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

ULP-INSA-ENGEES

THÈSE

présentée pour obtenir le grade de

Docteur de l'Université Louis Pasteur - Strasbourg I

Spécialité : Informatique

par

Olivier Hoenen

Parallélisation de Méthodes Adaptatives Semi-Lagrangiennes pour la Résolution de l'Équation de Vlasov

Membres du jury

Directeur de thèse : M. Éric Violard (MCF HDR, ULP, Strasbourg)
Rapporteur interne : Mme. Catherine Mongenet (Professeur, ULP, Strasbourg)
Rapporteur externe : M. Jacques Bahi (Professeur, IUT, Belfort)
Rapporteur externe : M. Stéphane Vialle (Professeur, SUPELEC, Metz)
Examinateur : M. Olivier Coulaud (Directeur de recherche, INRIA Bordeaux Sud-Ouest)
Examinateur : M. Éric Sonnendrücker (Professeur, ULP, Strasbourg)

Laboratoire des Sciences de l'Image, de l'Informatique et de la Télédétection (UMR ULP-CNRS 7005)

Remerciements

> Carambar – *Qu'est-ce qui est long, dur,*
> *et que les femmes n'ont pas ?*

Une thèse, c'est presque comme le service militaire : c'est long et c'est difficile. Si j'ai pu mener cette thèse à son terme, après quatres années jalonnées de hauts et de bas, c'est grâce à de nombreuses personnes que je voudrais remercier ici.

Je tiens en premier lieu à faire un grand merci à Éric Violard, qui m'a donné l'envie d'aller plus loin et qui a toujours été présent et disponible pour moi. Ses nombreux conseils, sa persévérance pour comprendre et offrir un œil critique sur mes algorithmes alambiqués, et sa patience pour lire et corriger mes phrases à rallonge m'ont été très précieux. Merci à Éric Sonnendrücker pour avoir rendu cette thèse possible, en finançant ma première année alors que je n'étais pas du tout assuré de pouvoir la continuer jusqu'au bout. Ensuite, je voudrais remercier Michel Mehrenberger, l'un des deux papa de YODA, pour m'avoir fait découvrir le monde des numériciens, pour ses indispensables explications sur les méthodes numériques et pour nos échanges de points de vue mêlant incompréhension et intérêt. Merci également à l'ensemble des membres du projet Calvi, pour m'avoir permis d'avoir une vision plus large et multi-disciplinaire sur ces travaux.

Je veux aussi remercier les membres de mon jury de thèse : Catherine Mongenet, Jacques Bahi et Stéphane Vialle pour avoir accepté d'être les rapporteurs de mon mémoire, et Olivier Coulaud pour avoir fait le déplacement depuis Bordeaux pour assister à la soutenance.

Ensuite, comme aller au labo tous les matins c'est plus facile quand l'ambiance y est bonne, je veux remercier tous les membres de l'équipe ICPS pour leur bonne humeur, leur sens de l'humour et les échanges de point de vue, qu'ils concernent le travail ou simplement l'actualité. Merci également pour les barbecues, les pauses café, le backgammon d'après repas et le basket quand vient l'été. J'aurai ici une pensée spéciale pour JC et Choopan (dans l'ordre de soutenance), mes deux compères depuis le DEA.

Je veux aussi remercier mes amis pour avoir su me changer les idées quand j'en avais besoin. Je pense notamment aux rôlistes, aux partenaires de badminton ainsi qu'à tous les autres. Merci pour tous ces bons moments que nous avons pu partager ensemble.

Je rajouterai un grand merci à ma famille, pour m'avoir soutenu pendant ces quatres dernières années ainsi que toutes celles qui les ont précédées.

Enfin, mon dernier merci (mais pas le moindre) sera pour ma Sarah, pour celle qui est à mes côtés depuis le début de cette thèse. Pour avoir su m'épauler dans les moments difficiles, avoir fait preuve de compréhension et de patience, alors même que la patience c'est pas vraiment ta première qualité, et pour toutes les joies partagées ces dernières années, merci ma puce.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des extraits de code

Introduction

Le calcul scientifique permet d'augmenter le niveau de compréhension des phénomènes naturels, avec des implications importantes pour la société. Ce domaine qui regroupe la simulation numérique, la modélisation et l'analyse numérique, permet de compléter les expérimentations réelles qui peuvent être extrêmement coûteuses. Dans le cas de phénomènes en physique des plasmas, par exemple, les enjeux sont très importants (nouvelles sources d'énergies ou de propulsion) et les expériences réelles sont particulièrement difficiles à réaliser et à reproduire. Ainsi, le projet phare ITER[1] destiné à démontrer la "faisabilité scientifique et technique de la fusion nucléaire" représente un investissement considérable estimé à quelques 10,3 milliards d'euros sur 30 ans. Parallèlement, la simulation numérique est une approche très économique et qui cependant offre la possibilité d'étudier la pertinence des modèles physiques utilisés. Ce travail de thèse est une contribution à cette autre approche.

Étant donnée la complexité des phénomèmes étudiés, obtenir des simulations réalistes, suppose de trouver la meilleure combinaison de modèles, de méthodes numériques et de ressources informatiques. La simulation numérique de phénomèmes physiques s'enrichit donc d'interactions fortes entre des chercheurs dans trois disciplines : physique, mathématique et informatique. Les recherches qui sont présentées ici s'inscrivent dans un projet pluridisciplinaire : le projet CALVI[2] de l'INRIA Nancy–Grand-Est, qui est consacré à la simulation de phénomèmes en physique des plasma et faisceaux de particules. Ces recherches sont axés sur l'exploitation des ressources informatiques et, plus précisément, ont pour but de développer des techniques de parallélisation permettant d'utiliser efficacement les ressources de calculs et de stockage.

La simulation numérique de systèmes en physique des plasmas et faisceaux de particules cherche à décrire l'évolution en temps de particules chargées. On s'intéresse particulièrement aux systèmes physiques qui interviennent lors d'une réaction de fusion thermonucléaire contrôlée, réalisée dans l'optique d'une production d'énergie. Parmi les réacteurs à fusion, on distingue principalement les systèmes par confinement magnétique (tokamak) et les systèmes par confinement inertiel (avec ou sans laser). La réaction de fusion ne se produit que lorsque la matière est portée à très haute température (plusieurs dizaines de millions de degrés) et entre dans l'état plasma. Dans ces conditions, le modèle est principalement l'équation de Vlasov couplée à d'autres équations aux dérivées partielles. Ce modèle décrit l'évolution des particules dans l'espace des positions et des vitesses appelé *espace des phases*. L'espace des phases a donc 6 dimensions dans le cas réel. Les méthodes numériques de résolution les plus précises discrétisent ce système sur un maillage de l'espace des phases. Parmi ces méthodes, les méthodes semi-Lagrangiennes [105] sont

---

[1] http://www.iter.org/

[2] https://www.inria.fr/recherche/equipes/calvi.fr.html

d'un grand intérêt en terme de qualité de la solution obtenue. Cependant, étant donné le grand nombre de dimensions du domaine physique, la résolution du modèle en utilisant ces méthodes, représente une énorme quantité de calculs et de mémoire. Le recours au parallélisme est donc indispensable pour réaliser des simulations numériques précises en utilisant le schéma semi-Lagrangien. Par exemple, pour une précision de 128 points par dimension, qui est à peine suffisante pour certains cas test, le maillage représente un volume de données de 32 téraoctets (To). À l'heure actuelle, les supercalculateurs les plus puissants embarquent une quantité de mémoire à peine suffisante pour atteindre cette résolution, à condition d'être entièrement dédiés à une telle simulation. De telles simulations ne sont donc pas envisageables pour l'instant, donc des modèles réduit ont été développés, notamment pour 2 et 4 dimensions de l'espace des phases.

Les méthodes adaptatives permettent de réduire considérablement le coût des simulations. Ces méthodes sont basées sur des maillages non uniformes et qui évoluent en temps. Leur principe est de concentrer l'effort de calcul dans les zones d'intérêt du domaine en augmentant le nombre de points de discrétisation dans ces zones, et en réduisant ce nombre dans les zones stables. Par contre, ces méthodes sont difficiles à implanter efficacement sur des architectures parallèles à mémoire distribuée. Ces difficultés découlent du caractère creux et dynamique du maillage, associé au grand nombre de dimensions du domaine de calcul.

Ces difficultés de parallélisation expliquent qu'à ce jour, les solveurs parallèles et adaptatifs de l'équation de Vlasov sont conçus uniquement pour des architectures à mémoire partagée pour lesquelles l'accès aux données n'a pas besoin d'être explicité. Un inconvénient majeur est que ces machines ne sont pas extensibles en mémoire. La taille des problèmes qu'elles peuvent simuler est donc limitée.

Les travaux effectués au cours de cette thèse concernent la parallélisation de méthodes adaptatives de résolution de l'équation de Vlasov. Les méthodes adaptatives que nous considérons utilisent le principe semi-Lagrangien. Notre objectif est de développer un code efficace sur une machine parallèle à mémoire distribuée. L'approche que nous utilisons consiste à élaborer des algorithmes *locaux*, au sens où les données nécessaires aux calculs sont voisines.

Le manuscrit comporte trois parties. La première partie situe le contexte de cette étude. Elle est composée de deux chapitres. Le chapitre 1 présente l'équation de Vlasov et les principes des méthodes numériques classiquement utilisées pour la résoudre. Nous y décrivons en particulier les méthodes utilisant un maillage de l'espace des phases et le principe semi-Lagrangien. Le chapitre 2 présente les élements constituant un solveur adaptatif et parallèle.

La deuxième partie de ce travail concerne la parallélisation d'une méthode adaptative appelée YODA (pour Yet anOther aDaptive Algorithm). Cette méthode est caractérisée par l'utilisation d'un opérateur d'interpolation local. Le chapitre 3 présente en détails cette méthode qui effectue l'adaptation du maillage en deux phases : *prédiction* et *compression*. La parallélisation de cette méthode est donnée dans le chapitre 4. Cette parallélisation utilise une décomposition du domaine de calcul en *régions*. Nous proposons un schéma de communication *à la volée* pour permettre les communications imprévisibles. Cette parallélisation est conçue pour un espace des phases réduit à 2 dimensions. Les techniques de parallélisation utilisées sont extensibles aux dimensions supérieures mais la structure de données sous-jacente doit être optimisée pour ces dimensions supérieures. Le chapitre 5 propose de telles optimisations pour un espace des phases à 4 dimensions. L'implantation parallèle de la méthode YODA offre des performances satisfaisantes pour une simulation 2D. Cependant, l'adaptation du maillage engendre un surcoût non négligeable en dimensions supérieures.

La troisième partie concerne la parallélisation d'une méthode obtenue à partir de YODA en changeant de méthode d'adaptation. Le chapitre 6 présente la nouvelle méthode d'adaptation basée sur un changement dans la phase de prédiction. Cette nouvelle méthode permet d'effectuer toutes les phases de la méthode localement et indépendamment sur chaque zone du domaine de calcul. Le chapitre 7 présente la parallélisation de cette nouvelle méthode. Le changement de méthode d'adaptation permet une parallélisation *par bloc*. Des hypothèses sur les paramètres de la simulation permettent de rendre l'ensemble des communications prévisibles et régulières.

La conclusion résume les différents résultats obtenus et donne quelques perspectives de travaux futurs.

Première partie

Contexte de l'étude

Chapitre 1

La résolution de l'équation de Vlasov

La physique des plasmas et des faisceaux de particules est un domaine de recherche très actif. La compréhension précise des phénomènes ayant lieu dans les plasmas est un challenge important, notamment dans la recherche de nouvelles sources d'énergie. Un des modèles les plus précis pour simuler de tels phénomènes est le modèle cinétique. Pour ce modèle, l'évolution des particules chargées dans le plasma est décrite par l'équation de Vlasov.

Dans ce chapitre, nous nous intéressons au problème complexe de la résolution numérique de l'équation de Vlasov. La première section replace l'équation de Vlasov dans le contexte physique de la modélisation des plasmas. Le deuxième section présente rapidement le système d'équation à résoudre dans le cadre du modèle cinétique pour l'équation de Vlasov. Enfin, la troisième section concerne les méthodes numériques utilisées pour résoudre un tel système, et plus particulièrement les méthodes semi-Lagrangiennes qui vont servir de base à notre étude. Des rappels sur les opérateurs différentiels utilisés dans ce chapitre sont donnés en annexe A.

1.1 Contexte physique

L'équation de Vlasov est utilisée pour décrire des phénomènes physiques ayant lieu dans un plasma ou des faisceaux de particules. La compréhension de phénomènes ayant lieu dans de tels milieux est primordiale dans l'optique de la production d'énergie à partir d'une réaction de fusion thermonucléaire contrôlée. Les plasmas considérés sont alors des plasmas générés artificiellement que l'on cherche à *confiner* le plus longtemps possible.

1.1.1 Le plasma

L'*état plasma* est souvent appelé le 4$^{\text{e}}$ état de la matière (après les états solides, liquides et gazeux). Cet état est obtenu en portant un gaz à très haute température (10^4 $^\circ K$). L'agitation thermique des molécules et des atomes qui composent le gaz devient alors suffisante pour que leur frottement arrache des électrons de la couche externe des noyaux auquels ils étaient rattachés : c'est le phénomène d'ionisation. Le gaz devient alors un mélange globalement neutre de particules chargées (électrons et ions) que l'on appelle *plasma*. Les plasmas représentent plus de 99% de la matière connue dans l'Univers (les étoiles et les nébuleuses gazeuses en sont composées), mais sont très rarement observables sur Terre compte tenu des conditions extrêmes nécessaires à leur apparition. Les aurores boréales et les éclairs sont parmi les rares phénomènes nous permettant de voir des plasmas à l'état naturel sur Terre. Des plasmas peuvent aussi être générés artificiellement, comme par exemple dans les néons, dans certains téléviseurs et dans des propulseurs spatiaux.

1.1.2 La fusion thermonucléaire

Un autre phénomène où l'on côtoie des plasmas est la fusion thermonucléaire contrôlée, qui nécessite des conditions de chaleur dépassant la centaine de millions de degrés Celsius. Le *facteur d'amplification* Q est défini comme le rapport entre la puissance produite et la puissance fournie au plasma. La production d'énergie de la réaction de fusion est reliée au produit $n.T.\tau_E$ par le critère de Lawson [77], avec n la densité du plasma (le nombre de particules par unité de volume, en $m-3$), T la température du plasma (en keV) et τ_E le temps de confinement de l'énergie (en s). Le temps de confinement de l'énergie représente la durée pendant laquelle la plasma est suffisamment chaud et dense pour permettre la réaction de fusion, en sachant que la tendance naturelle d'un plasma est de se disperser, de se refroidir et de perdre son énergie. Le confinement d'un plasma peut être obtenu par plusieurs méthodes :

- le *confinement gravitationnel* est un phénomène qui a lieu dans les étoiles et les soleils. La force gravitationnelle permet alors d'assurer une densité de plasma suffisante pour maintenir la réaction de fusion pendant un temps infini. Ce confinement n'est pas envisageable sur Terre.
- le *confinement inertiel* cherche à obtenir une densité élevée en tirant avec un puissant laser sur un petit volume ($\approx 10^{-3}m^3$) de matière. Une température ($\approx 10^7\ ^\circ K$) et une densité élevées ($10^6\ \times$ la densité de l'air) permettent de compenser un temps de confinement très court ($\approx 10^{-11}s$).
- le *confinement magnétique* consiste à utiliser un champ magnétique pour confiner le plasma dans une chambre ($\approx 10^3m^3$) généralement de forme toroïdale et appelée un *tokamak* ou un *stellarator* en fonction de la configuration magnétique utilisée. Le plasma (à température élevée $\approx 10^7\ ^\circ K$) est confiné sur un temps long ($\approx 10s$) ce qui permet de compenser sa faible densité ($\approx 10^{-5}\ \times$ la densité de l'air).

L'étude du confinement (inertiel ou magnétique) d'un plasma nécessite la compréhension de phénomènes très complexes, mettant en jeu des interactions non linéaires et des échelles de temps et d'espace multiples. Cette compréhension passe par la construction de modèles adaptés permettant des simulations numériques précises des différents phénomènes ayant lieu dans les plasmas de fusion.

1.1.3 La modélisation des plasmas

Pour modéliser un plasma, il faut avoir à disposition un modèle permettant de décrire l'interaction de particules chargées sous l'influence d'un champ électromagnétique. Ce champ est généré par les particules chargées elles-mêmes, c'est pourquoi il est nommé *champ auto-consistant.* Ce champ peut aussi être externe, par exemple le champ utilisé dans les tokamaks pour confiner le plasma, on parle alors de *champ appliqué.* En général on considère une combinaison de ces champs appliqués et auto-consistants. Il existe différentes classes de modèles permettant de décrire l'évolution de ces particules. On peut construire une hiérarchie à partir de ces modèles, selon qu'ils offrent plus de précision ou qu'ils sont moins coûteux numériquement (voir figure 1.1).

$$\begin{array}{ccccc} & \leftarrow + \text{précis} & & \text{- coûteux} \rightarrow & \\ \text{modèle à} & & \text{modèle} & & \text{modèle} \\ N\text{–corps} & \Longleftrightarrow & \text{cinétique} & \Longleftrightarrow & \text{fluide} \end{array}$$

Fig. 1.1 – Une hiérarchie de modèles pour les plasmas.

Modèle à N–corps

Toute particule chargée génère un champ électromagnétique qui a un effet dans tout l'espace. Bien sûr, l'intensité de ce champ décroit rapidement avec la distance, mais cette force n'est pas négligeable dans le voisinage de la particule et peut avoir une influence sur les autres particules du plasma. Le modèle complet pour décrire un plasma consiste alors à décrire explicitement le mouvement des N particules du plasma sous l'effet des champs générés par toutes les particules ainsi que d'éventuels champs appliqués externes. C'est le *modèle à N–corps*, qui correspond à une description du plasma au niveau microscopique. Le mouvement de ces particules est alors régi par la *loi fondamentale de la dynamique*, la loi de *Newton* :

$$\frac{d\mathbf{v}_i m_i}{dt} = \sum \mathbf{F}_{app} \tag{1.1}$$

avec $\mathbf{v}_i$ la vitesse d'une particule, m_i sa masse, $\mathbf{F}_{app}$ l'ensemble des forces appliquées sur la particule. Dans notre cas, le membre de droite est la *force de Lorentz*, issue des champs électromagnétiques appliqués et auto-consistants, qui s'écrit :

$$\sum_{j \neq i} \frac{q_j}{m_j} (\mathbf{E} + \mathbf{v}_j \times \mathbf{B})$$

où q_j et m_j sont respectivement la charge et la masse d'une particule j, $\mathbf{E}$ est le champ électrique et $\mathbf{B}$ est le champ magnétique. De plus, la vitesse $\mathbf{v}_i$ d'une particule i est liée à sa position $\mathbf{x}_i$ par la relation :

$$\frac{d\mathbf{x}_i}{dt} = \mathbf{v}_i$$

L'évolution des particules est donc complètement déterminée par le système :

$$\left\{ \begin{array}{rcl} \dfrac{d\mathbf{x}_i}{dt} & = & \mathbf{v}_i \\ \dfrac{d\mathbf{v}_i m_i}{dt} & = & \displaystyle\sum_{j \neq i} q_j (\mathbf{E} + \mathbf{v}_j \times \mathbf{B}) \end{array} \right. \tag{1.2}$$

à condition de connaître la position initiale $\mathbf{x}_0$ et la vitesse initiale $\mathbf{v}_0$ de chaque particule. Cette approche est la plus précise car elle est la plus proche de la physique. Mais dans un plasma où le nombre de particules est très important ($> 10^{10}$), une simulation numérique basée sur un tel modèle ne serait pas envisageable car beaucoup trop coûteuse pour la puissance des architectures actuelles.

Modèle cinétique

La représentation microscopique étant trop coûteuse pour être envisagée numériquement, des représentations de plus haut niveau ont été établies. Le *modèle cinétique* se base sur une représentation statistique des particules dans l'*espace des phases*. L'espace des phases est un espace utilisé en physique et qui permet d'interpréter géométriquement le mouvement d'un système mécanique. Il comprend les 3 dimensions de l'espace physique (ou dimensions de position) plus les 3 dimensions de l'espace des vitesses. Dans la suite, le nombre de dimensions considérées sera toujours le nombre de dimensions dans l'espace des phases, sauf s'il est précisé que l'espace considéré est l'espace des positions ou celui des vitesses.

Dans le modèle cinétique, chaque espèce de particules est caractérisée par la moyenne statistique de sa répartition dans l'espace des phases. La *fonction de distribution* est définie comme la fonction qui donne la probabilité de présence des particules pour un instant t et une position $(\mathbf{x}, \mathbf{v})$ de l'espace des phases. On note $f_s(\mathbf{x}, \mathbf{v}, t)$ la fonction de distribution pour une espèce s de particules. Le produit $f_s d\mathbf{x} d\mathbf{v}$ représente alors le nombre moyen de particules dont la position et la vitesse sont comprises dans un volume $d\mathbf{x}d\mathbf{v}$ centré en $(\mathbf{x}, \mathbf{v})$. On peut maintenant faire une des deux approximations suivantes :

- en considérant uniquement des collisions binaires avec des particules voisines, la fonction de distribution satisfait l'*équation de Boltzmann* :

  $$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_x f = Q(f, f) \tag{1.3}$$

  avec Q l'opérateur de collision non linéaire de Boltzmann. Cette équation est aussi utilisée dans la théorie cinétique des gaz.
- en supposant que les particules interagissent uniquement par le *champ moyen* qu'elles génèrent, la fonction de distribution est solution du *système Vlasov–Maxwell* :

  $$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_x f + \mathbf{F}(\mathbf{x}, t) \cdot \nabla_v f = 0 \tag{1.4}$$

  où $\mathbf{F}$ est la force de Lorentz. Le champ moyen, par opposition à l'ensemble des champs générés par les N particules (comme dans le modèle à N–corps), est le champ électromagnétique solution des équations de *Maxwell*, données en section 1.2.1.

Le choix entre ces deux approximations se fait en fonction du *temps de parcours moyen*, défini comme le temps moyen avant qu'une particule du plasma n'entre en collision avec une autre particule. Dans le cas où le temps de simulation est inférieur au temps de parcours moyen, il s'agit d'un *plasma sans collision* et on peut utiliser le modèle Vlasov–Maxwell.

Modèle fluide

Le *modèle fluide* est valable quand la fonction de distribution est proche de la distribution de Maxwell–Boltzmann, on dit alors qu'on a une fonction *Maxwellienne*. Ce modèle se base sur un ensemble de lois de conservation de grandeurs macroscopiques, incluant la *densité* n, la *vitesse moyenne* $\mathbf{u}$, et la *pression scalaire* p. Ces grandeurs peuvent être dérivées de la fonction de distribution f :

$$\begin{aligned}
n(\mathbf{x},t) &= \int f(\mathbf{x},\mathbf{v},t)d\mathbf{v} \\
\mathbf{u}(\mathbf{x},t) &= \frac{1}{n(\mathbf{x},t)}\int f(\mathbf{x},\mathbf{v},t)\mathbf{v}d\mathbf{v} \\
p(\mathbf{x},t) &= \frac{m}{3}\int f(\mathbf{x},\mathbf{v},t)(\mathbf{v}-\mathbf{u}(\mathbf{x},t))^2 d\mathbf{v}
\end{aligned}$$

Les équations fluides pour une espèce de particules du plasma sont alors les équations de conservation de la masse, de conservation de la quantité de mouvement et de conservation de l'énergie, couplées avec les équations de Maxwell.

1.2 L'équation de Vlasov

Dans ce travail nous allons nous intéresser essentiellement au modèle cinétique pour la description d'un plasma sans collision. Dans ce modèle, le plasma est régi par le système Vlasov–Maxwell dont les principes ont été brièvement énoncés dans le paragraphe 1.1.3. Dans cette section nous allons revenir sur ce système, et montrer comment il peut être réduit jusqu'à obtenir le système *Vlasov–Poisson* qui servira de base à notre étude.

1.2.1 Système Vlasov–Maxwell

En prenant la force de Lorentz sous la forme

$$\mathbf{F}(\mathbf{x},t) = \frac{q}{m}\left(\mathbf{E}(\mathbf{x},t)+\mathbf{v}\times\mathbf{B}(\mathbf{x},t)\right)$$

nous avons vu dans le paragraphe 1.1.3 que la fonction de distribution $f(\mathbf{x},\mathbf{v},t)$ (ou simplement f) est solution de l'équation de Vlasov (1.5) posée dans l'espace des phases $(\mathbf{x},\mathbf{v}\in\mathbb{R}^3\times\mathbb{R}^3)$ et dépendant du temps t :

$$\frac{\partial f}{\partial t}+\mathbf{v}\cdot\nabla_x f+\frac{q}{m}\left(\mathbf{E}(\mathbf{x},t)+\mathbf{v}\times\mathbf{B}(\mathbf{x},t)\right)\cdot\nabla_v f = 0 \tag{1.5}$$

L'équation de Vlasov est couplée aux équations de Maxwell (1.6) pour déterminer la valeur des champs électrique $\mathbf{E}$ et magnétique $\mathbf{B}$. Les équations de Maxwell pour un milieu homogène (ici le vide) sont les suivantes :

$$\begin{aligned}
-\frac{1}{c_0^2}\frac{\partial\mathbf{E}}{\partial t}+\vec{\mathrm{rot}}\,\mathbf{B} &= \mu_0\,\mathbf{J} \\
\frac{\partial\mathbf{B}}{\partial t}+\vec{\mathrm{rot}}\,\mathbf{E} &= \vec{0} \\
\mathrm{div}\,\mathbf{E} &= \frac{\rho}{\varepsilon_0} \\
\mathrm{div}\,\mathbf{B} &= 0
\end{aligned} \tag{1.6}$$

avec les inconnues $\mathbf{E}(\mathbf{x},t)$ et $\mathbf{B}(\mathbf{x},t)$ qui dépendent du temps t et de la position $\mathbf{x}$, et qui correspondent respectivement au champ électrique et au champ d'induction magnétique (que l'on appelera simplement *champ magnétique*). Les constantes c_0, ε_0 et μ_0 représentent quant à elles respectivement la vitesse de la lumière ($3.10^8\, m/s$), la permitivité électrique ($\approx 8,85.10^{-12}\, C/Vm$) et la permitivité magnétique du vide ($\approx 1,25.10^{-6}\, Vs/Am$). Enfin $\mathbf{J}(\mathbf{x},t)$ et $\rho(\mathbf{x},t)$ sont des données qui dépendent du temps et de la position dans l'espace physique. ρ est la *densité de charge* qui résulte de la présence de particules chargées électriquement et est définie par :

$$\rho(\mathbf{x},t) \quad = \quad q\int_{\mathbb{R}^3} f(\mathbf{x},\mathbf{v},t)d\mathbf{v} \tag{1.7}$$

$\mathbf{J}$ est la *densité de courant* qui est non nulle dès lors qu'il y a un déplacement de charge électrique et est définie par :

$$\mathbf{J}(\mathbf{x},t) \quad = \quad q\int_{\mathbb{R}^3} f(\mathbf{x},\mathbf{v},t)\mathbf{v}d\mathbf{v} \tag{1.8}$$

La difficulté lors de la résolution de ce système vient du fait que les deux données ρ et $\mathbf{J}$ sont calculées à partir des valeurs de la fonction de distribution comme l'indiquent les équations (1.7) et (1.8). Il s'agit donc d'un problème de couplage où les inconnues des deux systèmes dépendent les unes des autres. Le système Vlasov–Maxwell correspondant au couplage des équations (1.5) et (1.6) est par conséquent un système non linéaire très délicat à résoudre.

1.2.2 Système Vlasov–Poisson

La résolution précise et efficace des équations de Maxwell étant un problème non trivial, notamment pour des géométries complexes, on va chercher à simplifier le système. Pour cela on ne considère que des espèces de particules ayant des vitesses faibles par rapport à c_0, par exemple des ions lourds. Dans de telles conditions, $\mathbf{J}$ est petit et $\mathbf{B}$ est petit, ce qui implique, pour la force de Lorentz :

$$\frac{q}{m}(\mathbf{E} + \underbrace{\mathbf{v}\times\mathbf{B}}_{\approx 0})$$

et pour les équations de Maxwell :

$$\begin{aligned} \vec{\mathrm{rot}}\,\mathbf{E} &= \vec{0} \\ \mathrm{div}\,\mathbf{E} &= \frac{\rho}{\varepsilon_0} \end{aligned}$$

De plus, on sait que le champ électrique $\mathbf{E}$ dérive du potentiel électrique ϕ, ce qui se traduit par l'expression $\mathbf{E} = -\nabla\phi$ dont la divergence permet d'établir la relation suivante :

$$\left\{ \begin{aligned} -\Delta\phi &= \frac{\rho}{\varepsilon_0} \\ \mathbf{E} &= -\nabla\phi \end{aligned} \right. \tag{1.9}$$

Au final, nous avons un système réduit par rapport au système Vlasov–Maxwell. Le couplage des équations reste néanmoins non linéaire ce qui rend difficile la résolution efficace du système. Une simplification supplémentaire du système consiste à ne pas poser les équations dans l'espace des phases entier mais sur un domaine réduit 1D ($(\mathbf{x},\mathbf{v}) \in \mathbb{R}\times\mathbb{R}$) ou 2D ($(\mathbf{x},\mathbf{v}) \in \mathbb{R}^2\times\mathbb{R}^2$). Par la suite, nous utiliserons une écriture de l'équation de Vlasov *adimensionnée*, autrement dit les constantes physiques (q, m, ε_0, μ_0) seront normalisées à 1. Le système adimensionné s'écrira donc :

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_x f + \mathbf{E}(\mathbf{x}, t) \cdot \nabla_v f \;=\; 0 \tag{1.10}$$

couplé avec l'équation de Poisson (1.9).

1.2.3 Propriétés de l'équation de Vlasov

Dans ce paragraphe, nous allons énoncer maintenant quelques propriétés de l'équation de Vlasov. Nous détaillerons principalement la propriété fondamentale qui sera utilisée pour la construction de nombreuses méthodes numériques, puis nous passerons rapidement sur les propriétés de conservation de l'équation.

Propriété fondamentale

L'équation de Vlasov (1.10) peut être vue comme une équation de transport dans l'espace des phases. Elle peut alors s'écrire comme suit :

$$\frac{\partial f}{\partial t} + \mathbf{A} \cdot \nabla_{\mathbf{x},\mathbf{v}} f \;=\; 0, \quad \text{avec } \mathbf{A}(\mathbf{x}, \mathbf{v}, t) \;=\; \begin{pmatrix} \mathbf{v} \\ \mathbf{E}(\mathbf{x}, t) \end{pmatrix} \tag{1.11}$$

Pour résoudre une telle équation de transport, aussi appelée équation d'*advection*, on utilise les *courbes caractéristiques* (ou simplement *caractéristiques*) de l'équation. On note $(\mathbf{X}(t), \mathbf{V}(t))$ ces caractéristiques. Ces caractéristiques sont solutions du système d'équations différentielles ordinaires suivant :

$$\left\{ \begin{array}{lcl} \dfrac{d\mathbf{X}}{dt} & = & \mathbf{V}(t) \\ \dfrac{d\mathbf{V}}{dt} & = & \mathbf{E}(\mathbf{X}(t), t) \end{array} \right. \tag{1.12}$$

Avec les conditions initiales $\mathbf{X}(s) = \mathbf{x}$ et $\mathbf{V}(s) = \mathbf{v}$, la solution de ce système peut être notée $(\mathbf{X}(t;\, \mathbf{x}, \mathbf{v}, s), \mathbf{V}(t;\, \mathbf{x}, \mathbf{v}, s))$. Pour l'équation de Vlasov, les courbes caractéristiques représentent les trajectoires des particules dans l'espace des phases. Cette dernière notation exprime le fait que la valeur des caractéristiques, considérées au temps t, est égale à la position des particules qui, au temps s, étaient en position $\mathbf{x}, \mathbf{v}$. D'après le théorème d'unicité (Cauchy-Lipschitz), ces courbes ne peuvent pas se croiser dans l'espace des phases. En effet, deux particules ayant à un instant donné même position et même vitesse continueraient leur parcours ensemble, il s'agit donc d'une même et unique particule. Par contre ce n'est pas le cas dans l'espace physique classique.

La propriété fondamentale de l'équation de Vlasov dit que la fonction de distribution f est constante le long des courbes caractéristiques. La solution de l'équation d'advection s'exprime alors à l'aide de ces caractéristiques par :

$$f(\mathbf{X}(t;\, \mathbf{x}, \mathbf{v}, s), \mathbf{V}(t;\, \mathbf{x}, \mathbf{v}, s), t) \;=\; f(\mathbf{x}, \mathbf{v}, s)$$

En connaissant une approximation de la fonction de distribution au temps t, on peut donc déduire une approximation au temps $t + \Delta t$ par la formule suivante :

$$f(\mathbf{x}, \mathbf{v}, t + \Delta t) \;=\; f(\mathbf{X}(t;\, \mathbf{x}, \mathbf{v}, t + \Delta t), \mathbf{V}(t;\, \mathbf{x}, \mathbf{v}, t + \Delta t), t) \tag{1.13}$$

C'est cette propriété fondamentale qui est à la base des méthodes numériques *semi-Lagrangiennes.*

Propriétés de conservation

L'équation de Vlasov est une équation *conservative*, autrement dit il existe un certain nombre de grandeurs dont on peut contrôler la conservation au cours du temps pour évaluer la méthode numérique.

(i) Le *principe du maximum* dit que la fonction de distribution f est toujours positive et toujours inférieure à la valeur maximum de la fonction initiale f_0 :

$$0 \leq f(\mathbf{x}, \mathbf{v}, t) \leq \max(f_0(\mathbf{x}, \mathbf{v})) \tag{1.14}$$

(ii) La *conservation du volume* dit que l'intégrale de la fonction de distribution sur un volume V est égale à l'intégrale de la fonction initiale sur le même volume transporté par les caractéristiques au temps t_0 :

$$\int_V f(\mathbf{x}, \mathbf{v}, t) d\mathbf{x} d\mathbf{v} = \int_{F^{-1}(V)} f_0(\mathbf{x}, \mathbf{v}) d\mathbf{x} d\mathbf{v} \tag{1.15}$$

(iii) La *conservation des normes L^p*, avec $1 \leq p \leq +\infty$:

$$\frac{d}{dt}\left(\int (f(\mathbf{x}, \mathbf{v}, t))^p d\mathbf{x} d\mathbf{v}\right) = 0 \tag{1.16}$$

(iv) La *conservation de l'énergie* dit que l'énergie globale du système, à savoir l'énergie cinétique et l'énergie potentielle, est constante au cours du temps :

$$\frac{d}{dt}\Big(\underbrace{\int v^2 f(\mathbf{x}, \mathbf{v}, t) d\mathbf{x} d\mathbf{v}}_{\text{cinétique}} + \underbrace{\int (\mathbf{E}(\mathbf{x}, t)^2 + \mathbf{B}(\mathbf{x}, t)^2)\mathbf{x}}_{\text{potentielle}}\Big) = 0 \tag{1.17}$$

1.3 Les méthodes numériques de résolution

Qu'elle soit associée aux équations de Maxwell ou simplement à l'équation de Poisson, l'équation de Vlasov est non linéaire, et de ce fait n'a pas de solution analytique dans le cas général. Malgré cela, il existe des cas particuliers où l'équation de Vlasov peut être linéarisée et donc une solution analytique peut être trouvée. Ces cas sont généralement utilisés pour permettre la validation des codes de simulation numérique. Le reste du temps, l'équation de Vlasov est résolue numériquement en la couplant aux équations de Maxwell ou de Poisson. Il existe trois catégories principales de méthodes numériques pour résoudre ce système :

- les méthodes particulaires, PIC,
- les méthodes spectrales, basées sur des développements en série de Fourier,
- les méthodes basées sur un maillage de l'espace des phases.

Il faut avoir à l'esprit que ces différentes méthodes ont toujours dû prendre en compte les ressources de calcul et de mémoire disponibles. Certaines d'entre elles ont été développées dès les années 1960–1970, et seuls les problèmes sur 2 dimensions de l'espace des phases étaient envisageables à cette époque. L'essor de l'informatique dans les années 1980 a permis d'envisager des simulations plus réalistes, effectuées de manière quasi-exclusive avec des méthodes particulaires PIC. En effet, les méthodes particulaires utilisent un nombre de particules qui ne dépend pas de la dimension pour approcher un plasma. Elles sont par conséquent beaucoup moins gourmandes en ressources de calcul et de stockage que des méthodes spectrales (d'ordre élevé) et les méthodes sur des maillages (dont le nombre d'éléments augmente avec la dimension). Depuis une dizaine d'années, la rapide augmentation des capacités des architectures parallèles de calcul intensif permet d'envisager le développement de méthodes basées sur des maillages de l'espace des phases.

1.3.1 Méthodes PIC

Les seules méthodes numériques utilisées au cours des années 1980 voire 1990 étaient les méthodes *Particle-In-Cell* (PIC) [15]. Le principe de ce type de méthodes est de discrétiser la fonction de distribution par un nombre fini N de *macro-particules*. La fonction de distribution f_N est posée comme une somme de masses de Dirac d'un nombre N de macro-particules centrées aux positions $(\mathbf{x}_k(t), \mathbf{v}_k(t))_{1\leq k\leq N}$ et possédant chacune un poids w_k :

$$f_N(\mathbf{x}, \mathbf{v}, t) \;=\; \sum_{k=1}^{N} w_k \, \delta(\mathbf{x} - \mathbf{x}_k(t)) \, \delta(\mathbf{v} - \mathbf{v}_k(t))$$

Cette approche particulaire est couplée avec la résolution du champ électromagnétique. Ce champ est calculé en résolvant les équations de Maxwell (ou de Poisson) sur un maillage uniforme de l'espace physique, dont le coût est raisonnable par rapport à un maillage de l'espace des phases.

Dans la phase d'initialisation, les macro-particules approchent la fonction de distribution initiale f_0, qui est donnée analytiquement. Les positions de ces macro-particules sont alors tirées de manière aléatoire ou de manière déterministe. Dans la pratique, une répartition aléatoire suivant une loi de probabilité est souvent privilégiée par rapport à une répartition déterministe, qui n'est pas adaptée en présence de fortes différences de densité. Les champs électromagnétiques sont données à l'initialisation.

Le calcul d'un pas de temps par la méthode PIC se fait alors en 4 étapes. D'abord, les valeurs des champs électromagnétiques sont interpolées en la position de chaque particule. Ensuite les particules sont avancées en position et en vitesse à l'aide d'un schéma saute-mouton d'ordre 2 en temps. Une fois les particules avancées, les valeurs des densités de charge et de courant sont interpolées en chaque nœud du maillage uniforme de l'espace physique. La mise à jour des valeurs de ce maillage permet de calculer les champs électromagnétiques au nouveau pas de temps en résolvant les équations de Maxwell. On peut alors recommencer à la première étape. Ces étapes et les différentes méthodes utilisées pour chacune d'elles sont présentées plus en détails dans [10].

Ces méthodes PIC sont particulièrement intéressantes et extensibles puisque le nombre de particules ne dépend pas du nombre de dimensions. Actuellement, les simulations sur 6 dimensions de l'espace des phases ne sont envisageables dans des cas réalistes que par la méthode PIC. Mais pour certains problèmes importants, le bruit numérique engendré par ces méthodes ne permet pas d'obtenir une assez grande précision. Ce bruit numérique ne décroît qu'en $1/\sqrt{N}$, il faut donc rajouter beaucoup de particules pour le faire diminuer. Dans ces cas-là, il peut être plus avantageux d'avoir recours à des méthodes sur maillage.

1.3.2 Méthodes spectrales

Les méthodes spectrales sont des méthodes d'ordre élevé utilisées pour la résolution d'équations aux dérivées partielles. Contrairement aux méthodes par éléments finis, qui approchent une fonction avec des polynômes de faibles degrés sur un grand nombre d'éléments du domaine, les méthodes spectrales utilisent des polynômes de plus haut degré sur une seule ou quelques parties du domaine. Par conséquent, les méthodes spectrales sont considérées comme étant globales alors que les méthodes par éléments finis sont plutôt considérées comme étant locales. En général, la solution est approchée par des transformations en séries de Fourier, en polynômes d'Hermite ou encore par une combinaison des deux [37, 98].

Ces méthodes sont difficiles à développer, notamment dans la formulation des conditions aux bords du domaine, et ne sont pas bien adaptées à la décomposition du domaine dans l'optique de leur parallélisation.

1.3.3 Technique du splitting

L'équation de Vlasov nécessite de calculer les caractéristiques en résolvant le système d'équations différentielles ordinaires (1.12). On ne peut trouver de solution exacte à ce système étant donné que les inconnues sont liées. La technique du splitting d'opérateurs consiste à résoudre successivement deux équations sur chaque pas de temps au lieu d'une seule. L'équation de Vlasov s'écrit alors sous la forme de deux équations d'advection *à pas constant* (**v** est fixé pour l'équation (1.18), et c'est **x** qui est fixé pour l'équation (1.19)) :

$$\frac{\partial f}{\partial t} + \mathbf{v} \cdot \nabla_{\mathbf{x}} f = 0 \tag{1.18}$$

$$\frac{\partial f}{\partial t} + \mathbf{E}(\mathbf{x}, t) \cdot \nabla_{\mathbf{v}} f = 0 \tag{1.19}$$

De cette façon, chaque équation du système (1.12) peut être résolue séparément. Toutefois, cette technique du splitting peut, dans des cas où elle ne permet pas de maintenir certaines propriétés de conservation de l'équation, engendrer des erreurs d'approximation. Considérons le cas général du système :

$$\left\{ \begin{array}{rcl} \dfrac{du}{dt} & = & Au \\ \dfrac{du}{dt} & = & Bu \end{array} \right. \tag{1.20}$$

avec A et B deux opérateurs différentiels supposés constants entre t_n et t_{n+1}. Ce système correspond au splitting d'une équation de la forme $\frac{du}{dt} = (A + B)u$. Il peut être démontré que si A et B commutent ($A + B = B + A$), alors le splitting est exact. Cela signifie que le calcul sur un pas de temps de (1.18) puis de (1.19) est rigoureusement identique au calcul de l'équation sans splitting. Dans le cas contraire, une erreur est commise, localement d'ordre 2 et globalement d'ordre 1.

Le splitting de *Strang* [108] permet de réduire l'erreur commise par le splitting. Cette version du splitting consiste à effectuer le calcul de certaines des équations *splittées* sur un demi-pas de temps et à symétriser les opérateurs. Cela revient donc à calculer (1.18) sur un demi-pas de temps, puis à calculer (1.19) sur un pas de temps complet et à nouveau (1.18) sur un demi-pas de temps, ce qui correspond au schéma :

$$\frac{1}{2}A \rightarrow 1B \rightarrow \frac{1}{2}A$$

L'erreur obtenue est alors localement d'ordre 3 et globalement d'ordre 2. Au final, le splitting standard est qualifié d'ordre 1 en temps alors que le splitting de Strang est lui d'ordre 2 en temps. Un splitting d'ordre aussi élevé que voulu peut être obtenu en composant les opérateurs de manière adéquate. De plus, le splitting de Strang peut être généralisé à un nombre quelconque d'opérateurs en gardant le même schéma par demi-pas de temps et en symétrisant les opérateurs.

1.3.4 Méthodes semi-Lagrangiennes

Avec le développement des calculateurs parallèles, les méthodes basées sur des maillages de l'espace des phases ont à nouveau été considérées pour la résolution de l'équation de Vlasov. C'est le cas notamment pour des problèmes en 2 et 4 dimensions de l'espace des phases. Parmi les méthodes sur maillage, la méthode *semi-Lagrangienne* [105] est souvent considérée comme l'une des plus précises. Ces méthodes semi-Lagrangiennes, qui sont couramment utilisées en météorologie, ont été introduites dans le cadre de la résolution du système Vlasov–Poisson en 1976 [24].

Le principe de cette méthode est d'utiliser les caractéristiques de l'équation de transport pour mettre à jour la fonction de distribution à chaque pas de temps. La propriété fondamentale de l'équation de Vlasov (donnée en section 1.2.3) assure alors que toute valeur de la fonction de distribution reste la même le long de ces caractéristiques. Une fois les courbes caractéristiques calculées, il ne reste qu'à déterminer la valeur de la fonction de distribution aux points indiqués par les caractéristiques. Ces points ne coïncident pas forcément avec des nœuds du maillage, c'est pourquoi il est nécessaire de disposer d'un opérateur d'interpolation.

La méthode semi-Lagrangienne est une méthode de résolution directe, dans le sens où elle ne nécessite pas de raffiner le pas de temps pour assurer la convergence, comme c'est le cas pour d'autres méthodes de résolution d'EDP soumises aux conditions CFL [28]. La méthode semi-Lagrangienne est également utilisée en météorologie et en océanographie [79] pour résoudre les équation de Navier-Stokes. Il existe plusieurs variantes de la méthode semi-Lagrangienne, notamment les méthodes en avant, en arrière et volumique. Nous allons ici présenter en détails la méthode en arrière (ou méthode semi-Lagrangienne classique) qui sera la variante traitée dans ce travail de thèse.

La méthode semi-Lagrangienne classique se décompose en deux étapes pour la mise à jour de la solution à chaque pas de temps. Ces deux étapes sont retranscrites schématiquement sur la figure 1.2 qui montre un maillage posé dans un espace des phases à deux dimensions : l'axe horizontal représente les coordonnées en position (x) et l'axe vertical représente les coordonnées en vitesse (v). Pour alléger les notations, on pose f^{n+1} l'approximation de la fonction de distribution $f(\mathbf{x}, \mathbf{v}, t_{n+1})$ en chaque nœud $(\mathbf{x}, \mathbf{v})$ d'un maillage de l'espace des phases au temps t_{n+1}. Pour la même raison, les caractéristiques

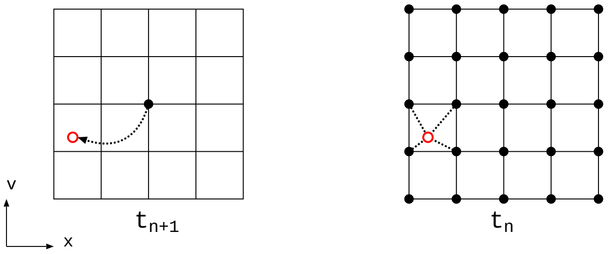

FIG. 1.2 – Les deux étapes d'une méthode semi-Lagrangienne.

$(\mathbf{X}(t_n;\, \mathbf{x}, \mathbf{v}, t_{n+1}), \mathbf{V}(t_n;\, \mathbf{x}, \mathbf{v}, t_n))$, qui donnent le point origine au temps t_n du point $(\mathbf{x}, \mathbf{v})$ au temps t_{n+1}, sont notées $(\mathbf{X}^n, \mathbf{V}^n)$. La fonction f^{n+1} est calculée à partir des valeurs de la fonction de distribution f^n (au temps t_n) de la manière suivante :

1. Pour chaque nœud $(\mathbf{x}, \mathbf{v})$ du maillage de l'espace des phases au temps t_{n+1}, on calcule $(\mathbf{X}^n, \mathbf{V}^n)$ pour identifier leur origine au temps t_n le long des courbes caractéristiques. Cette étape est représentée à gauche de la figure 1.2.
2. On sait que $f(\mathbf{x}, \mathbf{v}, t_{n+1}) = f(\mathbf{X}^n, \mathbf{V}^n, t_n)$, autrement dit la valeur en $(\mathbf{x}, \mathbf{v})$ au temps t_{n+1} est égale à la valeur de $(\mathbf{X}^n, \mathbf{V}^n)$ au temps t_n. Comme $(\mathbf{X}^n, \mathbf{V}^n)$ peut ne pas être un nœud du maillage, on calcule la valeur en ce point par interpolation avec des valeurs de nœuds proches. Cette étape est représentée à droite de la figure 1.2. L'interpolation est bi-linéaire pour ne pas surcharger la figure, mais le principe reste identique pour d'autres opérateurs d'interpolations.

On appelle *opérateur d'advection*, et on note $\mathcal{A}^-_{n+1}(\mathbf{x}, \mathbf{v})$, la fonction qui renvoie l'origine des caractéristiques $(\mathbf{X}^n, \mathbf{V}^n)$ au temps t_n, pour un point $(\mathbf{x}, \mathbf{v})$ au temps t_{n+1}. Cette fonction est bijective et son inverse, notée $\mathcal{A}^+_n(\mathbf{X}^n, \mathbf{V}^n)$, renvoie $(\mathbf{x}, \mathbf{v})$. Par extension, on appelle une *advection* la succession des deux étapes d'une méthode semi-Lagrangienne.

L'interpolation de la valeur au point $(\mathbf{X}^n, \mathbf{V}^n)$ de l'espace des phases (on parle parfois du *pied de la caractéristique*) peut s'effectuer avec des opérateurs d'interpolation tels que des splines, des polynômes de Lagrange ou d'Hermite. Il est malgré tout généralement admis que cet opérateur ne doit pas être d'un ordre trop faible, au risque de rendre la méthode semi-Lagrangienne trop diffusive : en particulier, une interpolation linéaire est à proscrire.

Schéma avec splitting

Nous présentons maintenant la méthode semi-Lagrangienne pour la résolution de l'équation de Vlasov posé dans l'espace des phases à 2 dimensions, telle qu'elle a été décrite dans l'article fondateur de Cheng et Knorr [24]. Il s'agit d'une méthode en arrière avec un splitting de Strang. Le principe du schéma avec splitting est de calculer f^{n+1} à partir de f^n en passant par plusieurs étapes intermédiaires. Chacune de ces étapes consiste à calculer les valeurs de la fonction de distribution sur un maillage intermédiaire. La solution approchée par chaque maillage intermédiaire n'est pas une solution physique.

1. Le maillage de l'espace des phases est inititialisé avec les valeurs de la fonction de distribution initiale $f_0(\mathbf{x}, \mathbf{v})$ en chacun de ces nœuds $(\mathbf{x}, \mathbf{v})$. La densité de charge au temps initial, notée ρ^0, est déduite en intégrant les valeurs de f_0 en $\mathbf{v}$, puis le champ électrique E^0 est calculé en résolvant l'équation de Poisson.

Une fois la phase d'initialisation effectuée et sachant que l'on connaît la valeur de f^n en tout nœud $(\mathbf{x}, \mathbf{v})$ du maillage, et la valeur de E^n en tout nœud du maillage sur le domaine physique, le passage d'une étape de temps $t_n = n\Delta t$ à l'étape $t_{n+1} = (n+1)\Delta t$ s'effectue de la manière suivante :

2. On commence par résoudre sur un demi-pas de temps l'équation (1.19) issue du splitting en vitesse. Cette résolution se fait selon la méthode semi-Lagrangienne vue précédemment. Le pied de la caractéristique est calculé, pour $\mathbf{x}$ fixé, par

$$\mathbf{v} = \mathbf{v} - \frac{\Delta t}{2}\mathbf{E}^n(\mathbf{x}) \tag{1.21}$$

et l'interpolation est calculée avec des valeurs f^n. On obtient ainsi une première solution intermédiaire (et donc un maillage intermédiaire) que l'on va appeler $f^{n\star}$.

3. Ensuite on résoud l'équation (1.18) issue du splitting en position sur un pas de temps complet. L'origine des caractéristiques est calculée, pour $\mathbf{v}$ constant, par

$$\mathbf{x} = \mathbf{x} - \mathbf{v}\Delta t \tag{1.22}$$

   La valeur au pied de ces caractéristiques est alors calculée par interpolation avec les valeurs de $f^{n\star}$ et la solution est stockée dans un nouveau maillage intermédiaire. On note $f^{n\star\star}$ cette solution intermédiaire.
4. On calcule ensuite le champ électrique dont les valeurs sont nécessaires lors de la prochaine phase du splitting de Strang (voir l'étape 5). La densité de charge ρ^{n+1} est calculée à partir de $f^{n\star\star}$, puis le champ $\mathbf{E}^{n+1}$ en est déduit en résolvant Poisson.
5. Enfin, la dernière étape consiste de nouveau à résoudre (1.19) sur un demi-pas de temps. Cette étape est analogue à l'étape 2 et l'interpolation des valeurs se fait avec les valeurs de la solution intermédiaire $f^{n\star\star}$. La solution obtenue au final est alors f^{n+1}.

Il est à noter que ρ^{n+1} et $\mathbf{E}^{n+1}$ calculés sur une solution intermédiaire (donc non physique) représentent vraiment la densité de charge et le champ électrique obtenus au temps t_{n+1}. Ceci découle du fait que la dernière étape en vitesse (l'étape 5) se fait à $\mathbf{x}$ constant, et donc ne modifie pas le calcul de la densité de charge.

Après ces différentes étapes, on possède $\mathbf{E}^{n+1}$ et f^{n+1} en tout point du maillage, on peut donc itérer le calcul pour le prochain pas de temps. Lors de ce passage entre deux pas de temps, on remarque que l'équation en vitesse (1.19) est résolue deux fois successivement sur un demi-pas de temps. Ces deux étapes peuvent par conséquent être remplacées par une seule étape où l'équation est résolue sur un pas de temps complet. Cela implique qu'en régime permanent (en dehors de la phase d'initialisation et des phases de diagnostics où on désire avoir la solution physique à un pas de temps donné, le schéma avec splitting consiste à enchaîner les étapes 2, 3 et 4 sur des pas de temps complets, comme le montre la figure 1.3.

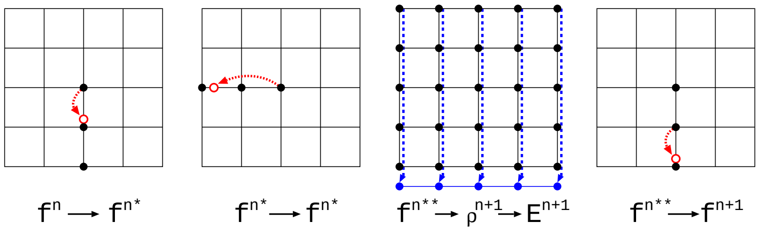

FIG. 1.3 – Décomposition du calcul d'une étape de temps avec le splitting.

Schéma sans splitting

Pour certains cas, voir par exemple [104], le splitting ne peut pas être utilisé. Dans ces cas-là, le calcul des caractéristiques peut se faire en utilisant un schéma prédicteur–correcteur d'ordre 2 en temps. Étant donnés le champ $\mathbf{E}^n$ et la fonction de distribution f^n au temps $t_n = n\Delta t$, le calcul de f^{n+1} se décompose en les opérations successives suivantes :

1. Tout d'abord, il faut prédire une première approximation $\overline{\mathbf{E}}^{n+1}$ du champ électrique en utilisant une valeur approchée de ρ^{n+1} déterminée comme suit :

$$\begin{aligned} \mathbf{J}^n &= \int f^n(\mathbf{x},\mathbf{v})\,\mathbf{v}\,d\mathbf{v} \\ \rho^{n+1} &= \rho^{n-1} - 2\Delta t\,\nabla\cdot\mathbf{J}^n \end{aligned}$$

2. Ensuite, grâce à cette approximation du champ électrique, l'origine au temps t_n des caractéristiques pour un nœud $(\mathbf{x},\mathbf{v})$ au temps t_{n+1} est calculée par un schéma saute-mouton décentré :

$$\begin{aligned} \mathbf{V}^{n+\frac{1}{2}} &= \mathbf{v} - \frac{\Delta t}{2}\,\overline{\mathbf{E}}^{n+1}(\mathbf{x}) \\ \mathbf{X}^n &= \mathbf{x} - \Delta t\,\mathbf{V}^{n+\frac{1}{2}} \\ \mathbf{V}^n &= \mathbf{V}^{n+\frac{1}{2}} - \frac{\Delta t}{2}\,\mathbf{E}^n(\mathbf{X}^n) \end{aligned}$$

   où $\mathbf{V}^{n+\frac{1}{2}}$ est une notation simplifiée de $\mathbf{V}(t_{n+\frac{1}{2}};\mathbf{x},\mathbf{v},t_{n+1})$.

3. L'interpolation des valeurs au pied des caractéristiques fournit une approximation de f^{n+1}. La densité de charge ρ^{n+1} est alors calculée à partir des valeurs de f^{n+1}, et le solveur de Poisson permet de calculer le champ $\mathbf{E}^{n+1}$.

4. Enfin, on calcule la différence $\|\mathbf{E}^{n+1} - \overline{\mathbf{E}}^{n+1}\|$ entre le champ électrique approché $\overline{\mathbf{E}}^{n+1}$ et le champ électrique $\mathbf{E}^{n+1}$, obtenu après calcul des valeurs de f^{n+1}. Si cette différence est supérieure à un certain seuil, cela signifie que la méthode ne converge pas encore. Alors, on itère l'algorithme à partir de l'étape 2 en remplaçant les valeurs du champ approché $\overline{\mathbf{E}}^{n+1}$ par les valeurs de $\mathbf{E}^{n+1}$.

Ce schéma d'ordre 2 permet de converger rapidement vers la solution. En pratique, seulement une ou deux étapes de correction sont nécessaires pour atteindre une erreur acceptable. Cela implique que le nombre d'étapes est au final à peu près équivalent au nombre d'étapes effectuées avec le splitting. Par contre, l'interpolation peut être beaucoup plus coûteuse que les interpolations faites avec le splitting, étant donné qu'elle s'effectue dans toutes les dimensions de l'espace des phases.

Chapitre 2

Mise en œuvre de solveurs adaptatifs parallèles

Les méthodes numériques basées sur un maillage de l'espace des phases et sur le schéma semi-Lagrangien permettent de résoudre de manière précise l'équation de Vlasov. Étant donnée l'énorme quantité de calculs et de mémoire nécessaires pour manipuler de tels maillages, le recours au parallélisme est indispensable si l'on veut envisager des simulations précises avec de telles méthodes numériques. Il existe un certain nombre de codes parallèles utilisant des maillages uniformes en 4D voire en 5D. Par contre, les maillages 6D représentent toujours une masse de données trop importante par rapport aux capacités des supercalculateurs actuels. À titre d'exemple, si une simulation pouvait être exécutée sur l'ensemble de leurs nœuds de calcul, 4 des 10 premiers supercalculateurs du top500 [111] n'auraient pas la mémoire cumulée nécessaire pour gérer les 32 To de données d'un maillage 6D avec une résolution de 128 points par dimension. Seul le recours à des méthodes numériques basées sur des maillages adaptatifs pourrait permettre d'envisager de telles simulations à l'heure actuelle. Mais le développement de méthodes numériques adaptatives et leur mise en œuvre parallèle pose de nombreux problèmes.

Ce chapitre reporte quelques unes des techniques utilisées pour construire un code parallèle de résolution d'équations aux dérivées partielles utilisant une méthode adaptative. L'objectif de ce chapitre est d'introduire les notions utiles pour les chapitres suivants. La première section de ce chapitre présente les techniques classiquement utilisées pour adapter le maillage. D'un certain point de vue, le maillage adaptatif peut être considéré comme une collection de données à accès parallèle. La parallélisation consiste alors à répartir les données sur l'espace de processeurs. Ce point de vue data-parallèle est présenté dans la deuxième section. Les techniques utilisées pour partitionner le maillage de façon à répartir convenablement les données sont présentées dans la troisième et dernière section.

2.1 Adaptation de maillage

Le principe des simulations numériques est de calculer une solution approchée d'un problème au lieu de la solution exacte. La solution approchée est souvent représentée par un maillage du domaine de calcul dont les nœuds identifient les points d'approximation. Selon le nombre et la disposition de ces points, le maillage est plus ou moins adapté par rapport à la solution exacte pour une précision donnée. En effet, le maillage peut être trop fin dans certaines régions du domaine alors que dans d'autres régions, il peut être trop grossier et ainsi laisser échapper des détails. La simulation numérique repose donc sur un compromis entre le coût de calcul engendré par le maillage et le niveau de précision qu'il permet d'atteindre par rapport à la solution exacte. L'objectif de l'adaptation de maillage est d'obtenir la précision recherchée pour un coût aussi faible que possible. Lorsque le maillage permet d'approcher la solution en tout point du domaine avec une précision recherchée (mais pas plus), on dit qu'il est *optimal*.

Dans cette section, après avoir rappelé quelques notions élémentaires sur les maillages, nous allons voir brièvement les différentes techniques qui permettent d'effectuer l'adaptation d'un maillage et ainsi d'obtenir un maillage proche de l'optimal.

2.1.1 Définition d'un maillage

Un maillage est une modélisation d'un domaine continu et des frontières qui l'entourent. Cette modélisation utilise des éléments géométriques discrets (par extension on parle de *discrétisation du domaine*). Ces éléments sont appelés généralement les *mailles* ou *cellules* du maillage. Ils forment une partition du domaine dans le sens où ils recouvrent entièrement le domaine sans qu'il n'y ait ni chevauchements ni espace vide. Ils possèdent des propriétés géométriques (coordonnées, tailles) et topologiques (connectivité, voisinage). Aux cellules sont ajoutés des *nœuds*, généralement associés aux valeurs approchées de la solution. Les maillages peuvent être classés en plusieurs catégories selon différents critères : la topologie (structuré, non-structuré), la forme des éléments (triangle, quadrangle), la régularité (uniforme, non uniforme), le système de coordonnées (cartésien, polaire), etc...

2.1.2 Méthodes d'adaptation de maillage

Après avoir construit un premier maillage selon des critères géométriques du domaine ou des critères inhérents à la solution approchée, le maillage peut être modifié. Ces modifications doivent permettre d'adapter le maillage au problème simulé pour le rendre plus efficace (dans le sens plus optimal). On appelle *raffinements* du maillage ces modifications. Ces raffinements se font habituellement en fonction d'un opérateur d'estimation de l'erreur. Si l'erreur due à la discrétisation est inférieure à un certain seuil, alors le mécanisme d'adaptation est stoppé et la solution est considérée suffisamment précise. Sinon, on construit une carte de l'erreur pour permettre un raffinement local du maillage dans les zones où la précision est insuffisante.

Le raffinement du maillage peut être obtenu par plusieurs méthodes qui peuvent influer sur la position du maillage (*r–adaptation*), sur la taille et la topologie du maillage (*h–adaptation*) ou encore sur l'ordre de la solution (*p–adaptation*). Ces différentes méthodes peuvent être combinées entre elles, les combinaisons les plus couramment utilisées étant les stratégies *hp* et *hr*.

r–adaptation

Dans les méthodes utilisant une stratégie de raffinement par *r*–adaptation, on change la position des nœuds du maillage, sans changer le nombre de mailles, de nœuds ou la topologie du maillage. Les nœuds des zones de faible gradient sont déplacés vers les zones du domaine à fort gradient. On obtient ainsi une forte concentration de points de discrétisation, et donc une augmentation de la précision dans les zones d'intérêt du domaine. Les méthodes utilisant cette stratégie sont souvent appelées méthodes par *grille* *mobile* [99, 3, 7] (ou *moving mesh*). On peut effectuer le déplacement des nœuds de manière uniforme (on parle alors de *moving grid*) ou non-uniforme (on parle alors de maillage de type *Lagrange*).

Cette stratégie est généralement peu onéreuse en terme de coût de calcul. Elle est surtout utilisée pour des problèmes en mécanique des fluides qui évoluent en temps. Mais ces méthodes ne permettent pas d'offrir à elles seules une solution convenable en fonction du problème considéré. En effet, au-delà d'un certain niveau, elles ne permettent pas d'améliorer la précision pour un maillage initial donné [51]. Si le nombre de points de discrétisation n'est pas suffisant pour obtenir la précision voulue, alors la méthode par r–adaptation doit être utilisée en conjonction avec une méthode permettant d'augmenter le nombre de nœuds (hr–adaptation).

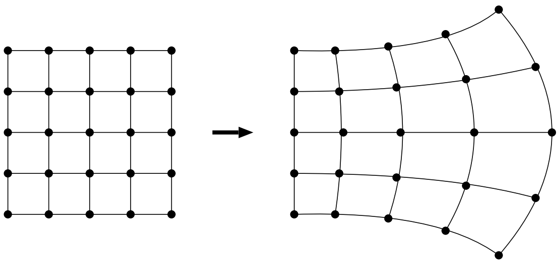

FIG. 2.1 – Adaptation d'un maillage par la stratégie r–adaptation.

p–adaptation

Les méthodes utilisant la stratégie de raffinement par p–adaptation permettent de faire évoluer le degré des fonctions d'interpolation utilisées pour reconstruire la solution sur chaque maille. Concrètement, le nombre de points de discrétisation par maille peut augmenter dans les zones où la discrétisation n'est pas assez précise par rapport à la solution exacte. Ces points supplémentaires permettent l'utilisation d'un opérateur d'interpolation d'ordre supérieur, et donc de se rapprocher de la solution exacte. Il est à noter que ces méthodes ne modifient pas la topologie du maillage.

Cette méthode offre de bons résultats et une meilleure convergence dans les régions où la solution est assez régulière. Mais elle n'est pas adaptée dans les zones du domaine où la solution est trop irrégulière. De plus, dans le cas d'une succession de maillages adaptés (évolution en temps), la rapidité de convergence peut être moins bonne que pour la h–adaptation. Cette stratégie est utilisée pour des méthodes basées sur des éléments finis [6, 5, 109] et pour des méthodes Galerkin discontinu [26]. Elle est souvent associée à la méthode h pour suivre plus facilement de fortes irrégularités (hp–adaptation [62]).

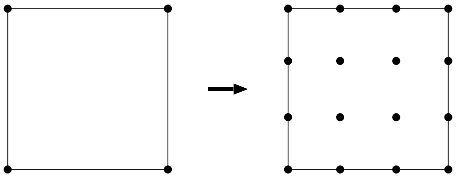

FIG. 2.2 – Adaptation d'un maillage par la stratégie p–adaptation.

h–adaptation

Avec la stratégie *h*–adaptation, c'est le nombre d'éléments du maillage qui évolue. Le *raffinant* et le *déraffinant* des mailles permet respectivement d'augmenter et de diminuer le nombre d'éléments du maillage. Lorsqu'elle est raffinée, une maille est remplacée par un certain nombre de mailles de taille plus petite et qui couvrent la même portion du domaine. Il en résulte une augmentation du nombre de points de discrétisation et donc une augmentation de la précision (et inversement quand des mailles sont déraffinées).

Il existe plusieurs méthodes pour raffiner le maillage. Le raffinement peut être appliqué de manière uniforme à l'ensemble des éléments du maillage ou bien de manière locale pour un ensemble d'éléments sélectionnés (raffinement *hiérarchique*). Le raffinement hiérarchique peut être ajouté assez facilement à un solveur existant, une structure de données hiérarchique permettant par exemple d'effectuer l'opération de déraffinement simplement en supprimant localement un niveau de la structure. Par contre ces méthodes peuvent engendrer des éléments excessivement fins autour des zones de discontinuité. Parmi les méthodes qui utilisent cette stratégie on peut citer notamment les méthodes d'éléments finis adaptatifs [4, 8, 94] et les méthodes AMR [13, 12]. Ces dernières utilisent un mécanisme de raffinement par blocs (on appelle *patch* ces sous-ensembles d'éléments) et non pas élément par élément.

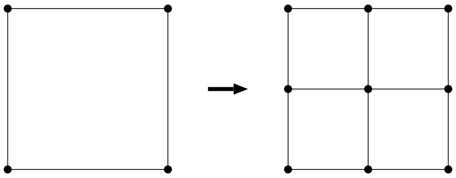

FIG. 2.3 – Adaptation d'un maillage par la stratégie *h*–adaptation.

2.2 Data-parallélisme et équilibrage de charge

Data-parallélisme

La parallélisation d'une méthode numérique qui discrétise les équations sur un maillage du domaine de calcul, utilise le paradigme du data-parallélisme [17, 46].

Une telle méthode numérique consiste en un algorithme séquentiel qui traite l'ensemble des données du maillage. Puisque ces données peuvent être accédées en parallèle, le principe de parallélisation est de répartir les données sur les processeurs (on parle de distribution des données). Comme le parallélisme de l'application repose sur la distribution des données et l'accès en parallèle aux données, ce type de parallélisme est appelé *parallélisme de données* ou *data-parallélisme.*

Les langages data-parallèles (dont le standard est HPF [74]) offrent des constructions syntaxiques et des facilités permettant au programmeur d'expliciter le parallélisme au moyen d'opérateurs dits globaux qui s'appliquent à toutes les données de l'ensemble, et de préciser la distribution des données sur les processeurs (soit en exprimant directement l'allocation des données aux processeurs au moyen de déclaration de type, soit au moyen de directives de distribution comme par exemple en HPF). Un programme data-parallèle se présente essentiellement comme un programme séquentiel utilisant des variables à accès parallèles (comme les tableaux). Le compilateur se charge alors de réaliser la distribution des données et de générer les communications pour y accéder. Cependant, lorsque la structure de données est irrégulière et dynamique comme dans le cas d'un maillage adaptatif, il peut être difficile d'obtenir un code parallèle efficace. La répartition des données de façon à équilibrer la charge entre les processeurs est notamment un problème délicat à résoudre par le compilateur.

2.2.1 Équilibrage de charge

Nous nous plaçons ici dans le cadre d'une parallélisation en utilisant le data-parallélisme, et de l'implantation d'un problème sur une architecture parallèle homogène à mémoire distribuée. Résoudre un problème en parallèle revient alors à décomposer l'ensemble des données du problème en autant de sous-ensembles qu'il y a de processeurs. Chaque sous-ensemble est alors affecté à un unique processeur.

Un programme parallèle efficace minimise, pour un nombre donné de processeurs, le temps total d'exécution du programme. Pour obtenir un tel programme, il est communément admis que les deux objectifs suivants doivent être atteints :

1. chaque processeur doit avoir la même quantité de calculs à effectuer (on parle aussi de charge de calcul).
2. le coût des communications doit être minimal (on identifie souvent ce coût au nombre de communications).

L'équilibrage de charge revient alors à trouver une décomposition de l'ensemble des données qui vérifie ces objectifs. Notons que pour un solveur, cette décomposition peut être définie à partir d'une décomposition du domaine de calcul : chaque sous-ensemble étant alors constitué des données du maillage situées à l'intérieur d'un sous-domaine. L'équilibrage de charge, au sein d'un solveur, peut être fait avant l'exécution du programme, on parle alors d'*équilibrage statique* ou au cours de l'exécution du programme, on parle alors d'*équilibrage dynamique.*

Équilibrage statique

Ce type d'équilibrage de charge est adapté aux solveurs utilisant un maillage uniforme, pour lesquels les données sont denses, et pour les solveurs utilisant un maillage qui n'évolue pas au cours du temps.

Équilibrage dynamique

Étant donnée une décomposition du domaine de calcul et la distribution des sous-domaines aux processeurs, il se peut que la charge de calcul correspondant à chaque sous-domaine varie au cours de l'exécution. Ainsi, au bout de quelques itérations de la méthode, la charge de calcul affectée à certains processeurs peut connaître une baisse (ou une hausse) significative. Une nouvelle décomposition du domaine doit alors être calculée pour respecter les deux objectifs permettant d'obtenir un programme parallèle efficace.

L'apparition de ce phénomène est prévisible pour un solveur donné, mais son influence sur l'efficacité et le moment où il se produira ne peuvent en général pas être prédits. Le calcul de la nouvelle décomposition doit donc s'effectuer au cours de l'exécution de manière dynamique en utilisant un *mécanisme d'équilibrage dynamique* de charge.

Les mécanismes d'équilibrage dynamique de charge peuvent être regroupés en deux familles en fonction de la fréquence à laquelle une nouvelle décomposition est calculée.

Cette fréquence caractérise la *dynamicité* de la charge pour une application donnée [27]. Une dynamicité *quasi-statique* correspond aux situations où la charge évolue très peu à chaque pas de temps. L'équilibrage de charge est donc peu fréquent et peut se faire après un nombre conséquent d'itérations. Si la charge varie beaucoup et à chaque itération du calcul, on parle alors de situations *hautement dynamiques*. Dans ces cas là, le coût de calcul de la nouvelle décomposition doit être amorti par le gain dû à la nouvelle décomposition.

La fréquence de l'équilibrage dépend essentiellement du solveur, mais elle peut aussi varier pour un même solveur en fonction du cas test ou du jeu de données.

2.3 Partitionnement du maillage

Dans la plupart des codes de calcul scientifique qui utilisent un maillage du domaine du calcul, la parallélisation repose sur une décomposition de ce maillage. Chaque sous-ensemble (ou *partition*) du maillage est ensuite alloué à un unique processeur. Cette décomposition (ou partitionnement) doit répondre aux deux objectifs de l'équilibrage de charge qui, dans la pratique, sont souvent en conflit. Pour atteindre l'objectif de minimisation du coût des communications, de nombreux travaux utilisent le graphe d'adjacence des éléments du maillage et cherchent à en minimiser le nombre d'arêtes coupées (*edge-cut*) pour chaque sous-ensemble.

Dans le cas où le maillage considéré est adaptatif, les partitions doivent être calculées en cours d'exécution du solveur (on parle d'*équilibrage dynamique*) sans que celui-ci soit ralenti. On ajoute donc les objectifs suivants :

3. L'algorithme de partitionnement et la redistribution du maillage doivent être peu coûteux, en tout cas de coût moindre par rapport au gain obtenu par l'équilibrage. Idéalement, l'algorithme doit également être parallélisable efficacement.
4. L'algorithme de partitionnement doit être *incrémental*. Autrement dit, un léger changement dans le domaine ne doit provoquer qu'un léger changement dans sa décomposition, et par conséquent un minimum de migration de données.
5. Le schéma de communication associé à la nouvelle décomposition doit être facile à déterminer. Autrement dit, les partitions de géométrie simple sont préférées à celle de géométrie complexe.

Répondre de manière optimale aux objectifs 1 et 2 est un problème reconnu comme *NP-complet* [44, 16], autrement dit dont on ne peut trouver de solution en temps raisonnable [1]. Ce problème étant crucial pour obtenir une parallélisation efficace, de nombreuses heuristiques ont été développées, chacune cherchant à approcher au mieux la solution optimale. On peut répartir ces heuristiques en deux grandes catégories : les méthodes géométriques et les méthodes topologiques. La plupart des heuristiques pour un partitionnement et un équilibrage de charge dynamique sont basées sur des heuristiques statiques légèrement modifiées. Nous décrivons dans la suite les heuristiques les plus communes et donnons leurs avantages et inconvénients.

2.3.1 Méthodes géométriques

De nombreuses méthodes dites *géométriques* ont été développées pour prendre en compte les spécificités de la géométrie du domaine de calcul. En outre, très souvent dans

---

[1] le temps de résolution n'est pas polynomial mais exponentiel en fonction de la taille du problème

les simulations de phénomènes physiques, les éléments d'un maillage interagissent avec les éléments qui sont géométriquement proches d'eux. C'est pour cela que mettre les éléments proches géométriquement dans une même partition et donc sur un même processeur peut permettre de réduire les communications.

Il existe deux familles de méthodes géométriques : celles qui cherchent à diviser récursivement le domaine, et celles qui cherchent à agglomérer des éléments fins et réguliers du maillage.

Bissections récursives

Le principe des méthodes de partitionnement par bissections récursives est de diviser le maillage par des coupes successives. Chaque coupe divise le maillage (ou une partie du maillage) en 2 sous-parties représentant une même charge de calcul. Cela implique que le nombre total de partitions générées est une puissance de deux, mais cette restriction peut être levée assez facilement, rendant ces algorithmes généralisables à un nombre quelconque de partitions. Les communications s'effectuent aux frontières de chaque partition donc au niveau de chaque coupe ce qui implique une contrainte supplémentaire sur le choix de la coupe. En effet, plus les éléments aux frontières d'une partition sont nombreux, plus il y aura de communications à effectuer pour obtenir les valeurs des éléments voisins appartenant à d'autres processeurs.

Les auteurs de [22] ont montré que des coupes régulières sont intéressantes pour limiter le nombre d'éléments aux frontières des partitions d'un maillage bien formé. Ils démontrent qu'il existe toujours une ligne droite en 2D, ou un hyperplan en dimensions supérieures, qui divise un maillage en deux parties contenant un même nombre d'éléments (à 1 élément près). Ils définissent la *graduation* (grading) du maillage par

$$\Delta = 1 + \log_2(M/m)$$

avec M la longueur du plus grand côté parmi les cellules du maillage, et m la longueur du plus petit côté. Cette métrique est utilisée pour montrer que l'hyperplan qui divise le maillage en 2 sous-ensembles de même taille coupe un nombre d'arêtes de l'ordre de

$$\mathcal{O}(\Delta^{1/d} n^{1-1/d} (\log n)^{1/d})$$

où d est le nombre de dimensions de l'hyperplan et n le nombre de nœuds du maillage. Ces travaux montrent que des coupes droites et des partitions de forme "carrée", autant que possible, permettent de réduire la taille des frontières et donc le nombre de communications engendrées par le partitionnement du maillage.

Parmi les méthodes utilisant des lignes ou des plans, la plus connue est sûrement la méthode ***Orthogonal Recursive Bisection*** [2] (ORB) introduite dans [11]. Dans cette approche, les lignes ou les plans de coupe sont orthogonaux aux axes de coordonnées. Chaque coupe est effectuée perpendiculairement à la plus longue direction de la géométrie à subdiviser. La position d'une bissection est calculée de manière à répartir la moitié des objets de la géométrie de chaque côté de la coupe. Chaque partition ainsi formée possède le même nombre d'objets du maillage, et on parle alors de partitions de même *poids*. Ces partitions sont alors subdivisées à leur tour par l'algorithme de bissection qui est exécuté

---

[2] également référencée sous le nom Recursive Coordinate Bisection (RCB)

récursivement sur chacune d'elles. La modification de la répartition des poids pour chaque bissection (au lieu de faire une répartition équitable à chaque fois) permet d'obtenir un nombre quelconque de partitions de même poids, et par conséquent de généraliser cette technique de partitionnement pour créer un nombre quelconque de partitions.

Plusieurs travaux basés sur cette méthode ont cherché à l'améliorer en terme d'aspect des partitions générées ou pour permettre d'atteindre un équilibre de la charge de calcul plus proche de l'idéal [3]. On peut citer notamment la méthode ***hierarchical-ORB*** (*H-ORB*) [114]. Dans cette méthode, les bissections sont effectuées dimension par dimension et sans alterner. Soit un domaine 2D à découper en 8 partitions, la méthode H-ORB consiste à effectuer dans un premier temps deux bissections récursives selon la première dimension. Ensuite, les deux bissections récursives suivantes s'effectuent selon la deuxième dimension pour chacune des 4 premières partitions. Cette méthode tend à former des régions plus "carrées". Une autre méthode, introduite dans [102], cherche à obtenir une meilleure répartition de la charge de calcul tout en préservant la contiguïté des partitions. Dans cette méthode, appelée *ORB-final* ou ***ORB-MM*** (pour *Median of Medians*), les cellules du maillage ne sont pas considérées au travers de leur point central mais comme des éléments de surface. Toutes les cellules qui chevauchent la ligne de coupe sont alors ordonnées puis affectées à un côté de la bissection tant que la moitié de la charge totale de départ n'est pas dépassée. Les partitions obtenues sont plus irrégulières mais elles restent contiguës et elles permettent d'atteindre un meilleur équilibre.

Une autre méthode, la méthode ***Unbalanced Recursive Bisection*** (*URB*) proposée dans [66], repose sur le choix des poids des partitions créées à chaque bissection pour obtenir des partitions ayant un meilleur aspect. L'aspect d'une partition (*aspect ratio*) est défini par $ar = \max(h/w, w/h)$, avec h la hauteur du rectangle et w sa largeur. Le principe de cette approche est de tester, en plus de la bissection menant à une répartition en $p/2 : p/2$ (comme pour les méthodes ORB), des bissections menant à des répartitions $(p-1) : 1$,$(p-2) : 2$, etc... Ces répartitions sont testées dans les différentes directions et la bissection qui génère des partitions ayant le plus petit ar est retenue. Enfin l'algorithme est appelé récursivement sur chacune de ces partitions, en prenant en compte le fait que chaque subdivision peut représenter une répartition différente de la charge de calcul. Les partitions construites par la méthode URB sont généralement considérées comme ayant un meilleur aspect que les partitions obtenues par simple ORB (on évite notamment la construction de partitions fines et longues). Cela se traduit en général par une légère baisse de communications.

Dans les méthodes de type ORB ou URB, chaque coupe est effectuée orthogonalement aux axes de coordonnées. La méthode ***Recursive Inertial Bisection*** (*RIB*) [39, 110], cherche à pallier cette rigidité en proposant des coupes perpendiculaires à l'axe principal d'inertie du maillage. La première étape de cette approche consiste à trouver cet axe d'inertie. Pour cela, le barycentre de chaque maille est calculé en considérant que la masse est uniformément répartie sur toute sa surface. Ensuite, on construit une matrice $d \times d$ (d est le nombre de dimensions du domaine) dont les coefficients correspondent au produit tensoriel des coordonnées des centres de gravité de chaque maille. La direction principale d'inertie est alors indiquée par un vecteur propre de cette matrice. Une fois que l'axe principal d'inertie est trouvé, les barycentres de toutes les mailles sont projetés sur cette droite, puis séparés en deux sous-ensembles contenant le même nombre d'éléments.

---

[3] la charge idéale est définie par la charge globale divisée par le nombre de processeurs

L'algorithme est ensuite répété récursivement sur chaque sous-ensemble jusqu'à obtenir le nombre de partitions souhaité. Cette méthode plus souple construit généralement de meilleures partitions que les méthodes basées sur ORB ou URB, d'autant plus en présence de géométries complexes. Par contre elle est plus coûteuse que les méthodes précédentes. De plus, les partitions obtenues ne sont plus de simples parallélépipèdes, ce qui peut rendre la détermination du schéma de communications plus complexe.

Octree et courbes de remplissage de surface

Les méthodes géométriques vues précédemment cherchent à construire les partitions en divisant le domaine global par un certain nombre de coupes. Une autre approche est de construire ces partitions par agglomération (ou *clustering*) d'éléments fins les uns avec les autres. Ces éléments fins sont généralement issus de la division simultanée de chaque axe de coordonnées en deux moitiés. Chaque division donne donc place à 2^d nouveaux sous-domaines, où d est le nombre de dimensions du domaine. Le domaine peut donc être représenté par un arbre binaire dont le nombre de branches issues de chaque nœud dépend de d : on parle de *quad-tree* en 2D et d'*octree* en 3D.

Une méthode directe pour effectuer le partitionnement d'un tel domaine est basée sur un parcours *en profondeur d'abord* de l'arbre binaire représentant la décomposition du domaine en éléments fins. Une telle approche est appelée partitionnement par *octree* [100, 43, 83]. Dans la pratique, deux parcours successifs de l'arbre sont nécessaires : le premier passage permet de déterminer le coût (par exemple le nombre de feuilles, chaque feuille pouvant être éventuellement valuée) de chaque sous-arbre et sauvegarde ce coût au niveau de chaque nœud de l'arbre. Le nœud correspondant à la racine de l'arbre contient alors la charge globale de l'arbre. Le deuxième passage consiste à effectuer un nouveau parcours et à accumuler les nœuds (et les sous-arbres associés) dans une même partition tant que la somme des coûts associés à ces nœuds est inférieure à la charge idéale d'une partition.

On peut alors remarquer que ce parcours de l'arbre revient à ordonner de manière unidimensionnelle les objets pour obtenir un ordre global sur l'ensemble des éléments fins constituant le domaine. Cet ordonnancement 1D est aussi appelé *linéarisation* des éléments, et permet de représenter l'ensemble des éléments le long d'une courbe. De plus, on dit que cet ordre *respecte la localité* si les éléments proches selon cet ordre 1D sont également proches dans l'espace de départ. Le partitionnement consiste alors à découper l'espace 1D ordonné pour former un ensemble de sous-ensembles 1D (un ensemble de segments). Chaque partition du maillage est formée des éléments traversés par un segment de la courbe. Ces partitions sont qualifiées de *bien formées* si l'ordre choisi respecte la localité des éléments.

Il existe un autre moyen couramment utilisé pour trouver une telle linéarisation, il s'agit des *courbes de remplissage de surface* ou *Space Filling Curves* (SFC) [96]. Une courbe de remplissage est une courbe qui occupe tout l'intérieur d'un carré sans jamais s'intersecter. Plus formellement, une courbe de remplissage est une surjection continue de $[0, 1] \times [0, 1]$ dans $[0, 1]$. Ces courbes sont rangées dans la catégorie des *fractals*. En pratique, une courbe comme celle de ***Peano*** (voir figure 2.4) est construite récursivement à partir d'un *motif* de base. À chaque niveau de la récursivité, les segments de la courbe sont remplacés par des copies à échelle réduite de ce motif. Chaque copie du motif peut éventuellement subir une rotation ou une réflexion. Le motif de départ dépend notamment du type de subdivision considéré. Dans le cas d'un carré divisé en 4 sous-domaines, on parle de *courbes de*

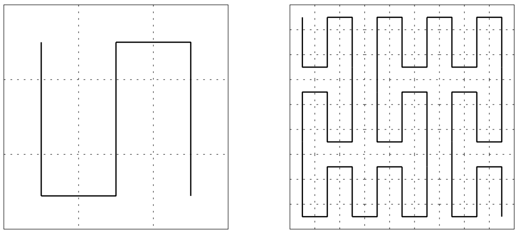

Fig. 2.4 – Construction récursive de la courbe de Peano.

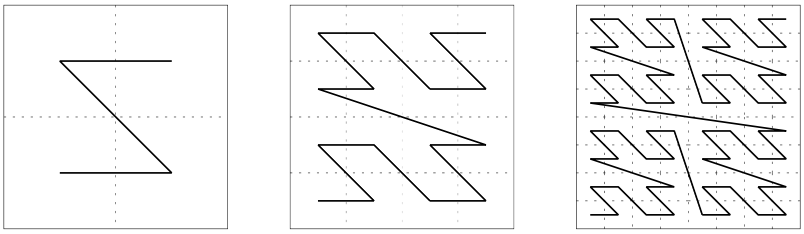

Fig. 2.5 – Construction récursive de la courbe de Lebesgue (ou z-curve).

Peano binaires. Parmi les courbes de Peano binaires on peut notamment citer la *courbe de Lebesgue* (ou *z-curve*), la *courbe de Hilbert* et la *courbe de Moore*. Dans le cas d'un domaine carré divisé en 9 sous-domaines, on parle de *courbes de Peano ternaires*, c'est le cas notamment pour la courbe de Peano telle qu'elle est donnée dans la figure 2.4. Mais il existe aussi des motifs adaptés à une subdivision en triangles homothétiques et dont la *courbe de Sierpinski* est issue. Dans la suite nous nous intéresserons essentiellement au cas des courbes binaires.

La courbe de Lebesgue (ou *z-curve*) est la plus simple, son motif de base est un Z qui est répété sans subir aucune transformation lors de la construction récursive de la courbe (voir figure 2.5). Sa simplicité la rend très facile à implanter, mais elle provoque aussi des *"sauts"* dans la linéarisation. Autrement dit il existe des éléments proches dans la courbe 1D qui ne sont pas proches dans l'espace de départ. La z-curve ne respecte donc pas toujours la localité, et cela est d'autant plus vrai que le nombre de dimensions augmente.

La courbe de Hilbert est aussi basée sur un motif simple qui prend la forme d'un crochet (ou d'un U). Mais cette fois le motif peut subir une rotation et une réflexion selon le quadrant dans lequel on raffine (voir figure 2.6). Ces transformations permettent de garantir que la courbe fait un "saut" minimum dès qu'elle change de quadrant. La courbe de Hilbert préserve donc la localité, y compris lorsque le nombre de dimensions augmente. C'est pour ces raisons que cette courbe est la plus couramment utilisée pour effectuer un partitionnement. La courbe de Moore, quant à elle, est construite à partir d'un motif de base carré. En simplifiant, elle peut être considérée comme une extension de la courbe de Hilbert permettant d'obtenir une courbe fermée (un circuit). En effet la courbe de Moore peut être construite à l'aide d'un certain nombre de courbes de Hilbert dont les extrémités sont reliées entre elles. Le nombre de courbes de Hilbert nécessaires dépend du nombre de dimensions du domaine.

Le partitionnement par courbe de remplissage a été utilisé pour la première fois avec une z-curve par les auteurs de [115]. Par la suite, de nombreux travaux ont utilisé ces

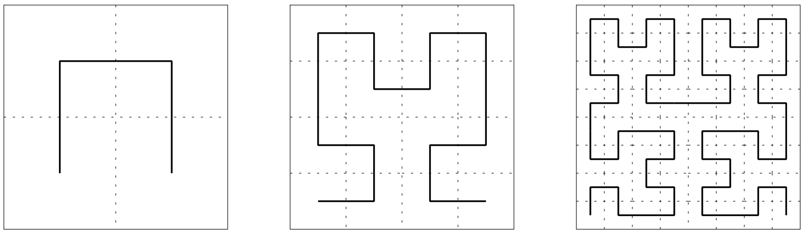

Fig. 2.6 – Construction récursive de la courbe de Hilbert.

différentes courbes, et notamment la courbe de Hilbert, pour partitionner des maillages adaptatifs [89, 87, 88, 43] ou des maillages uniformes pour lesquels la répartition de la charge de calcul évolue [91, 106]. Ces méthodes offrent en général une qualité de partitionnement équivalente ou légèrement inférieure aux méthodes de type ORB, avec en plus un léger surcoût en terme de stockage mémoire causé par les formes localement irrégulières des partitions. Par contre cette approche s'avère être aussi rapide que les méthodes ORB et le choix de la courbe de Hilbert peut permettre de conserver la localité géométrique pour obtenir une meilleure incrémentalité (objectif 4).

2.3.2 Méthodes basées sur les graphes

La principale limitation des méthodes géométriques est qu'elles n'utilisent pas d'informations sur la connectivité des éléments du maillage. Pour prendre en compte cette information, des méthodes de partitionnement basées sur la théorie des graphes ont été développées. Ces méthodes reposent sur l'observation suivante : le problème de partitionnement d'un maillage est équivalent au partitionnement d'un graphe non orienté $G = (V, E)$, où les sommets V représentent les éléments du maillage, et les arêtes E représentent les connections entre les éléments voisins dans le maillage. Le graphe G est appelé graphe d'adjacence (ou graphe dual) du maillage auquel il est associé. La figure 2.7 présente l'idée du partitionnement d'un maillage en se basant sur son graphe dual. Les sommets et les arêtes du graphe G peuvent être pondérés pour représenter la charge de calcul associée à un élément du maillage ou toute autre information supplémentaire. Enfin, le degré d'un sommet v_i est noté $\delta(v_i)$ et correspond au nombre d'arêtes incidentes à v_i.

Le k–partitionnement du graphe est défini comme une application $\mathcal{P} : V \to [1, k]$ des sommets du graphe vers les sous-ensembles $V_1, V_2, ..., V_k$ telle que $\mathcal{P}(v_i) = j$ si et seulement si le sommet v_i appartient à V_j. Les sous-ensembles V_i sont appelés des partitions et sont tels que $\cup_i V_i = V$ et $V_i \cap V_j = \emptyset, \forall i \neq j$. La somme $|V_i| = \sum_{v_j \in V_i} w(v_j)$ des pondérations associées aux sommets présents dans la partition est appelée *poids* de la partition V_i. $|V|$ représente le poids du graphe tout entier, soit la somme des poids des partitions. De plus,

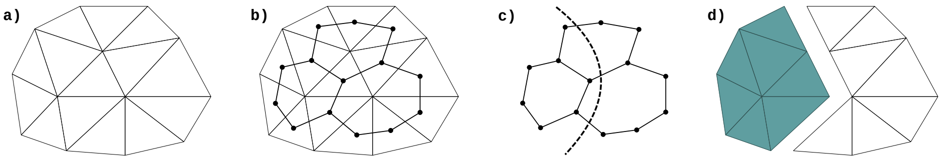

Fig. 2.7 – Maillage triangulaire (**a**), son graphe dual (**b**), la bissection du graphe dual (**c**) et les deux partitions engendrées (**d**).

une arête coupée (ou *cut-edge*) est définie comme une arête dont les deux extrémités se trouvent dans des partitions différentes. À chaque partitionnement est alors associé un ensemble d'arêtes coupées E_c. La *cut-size* du partitionnement correspond au cardinal $|E_c|$.

Le *problème du k–partitionnement* d'un graphe consiste alors à trouver un partitionnement tel que :

- le poids de chaque partition soit égal à $|V|/k$.
- la cut-size $|E_c|$ soit minimum.

Ce problème est NP-Complet. Les différentes approches développées pour le résoudre se basent sur des heuristiques permettant de trouver une approximation du résultat. Le domaine du partitionnement de graphe et de l'optimisation combinatoire étant extrêmement vaste, cette partie propose un tour d'horizon non exhaustif des techniques couramment utilisées, sans entrer dans les détails.

Méthodes globales

Les méthodes de partitionnement géométrique de type ORB ou RIB peuvent très bien être appliquées au graphe associé à un maillage. Mais cela implique que l'algorithme de partitionnement ne prenne pas en compte l'information de connectivité fournie par le graphe.

La méthode de partitionnement utilisant l'information sur la connectivité des éléments de la manière la plus directe est certainement la méthode ***Recursive Graph Bisection*** (RGB) présentée par exemple dans [101]. Cette méthode utilise la notion de *distance* entre deux sommets d'un graphe, définie par la longueur du plus court chemin entre les deux sommets. Cette notion de distance qui prend en compte la connectivité du graphe remplace la distance Euclidienne pour déterminer les emplacements des coupes. Dans un premier temps, les deux sommets les plus distants sont trouvés (la distance entre ces deux sommets est appelée le *diamètre* du graphe). Puis chaque sommet est trié dans l'ordre croissant de distance par rapport à l'un de ces sommets extrêmes. Les sommets sont alors répartis en deux sous-domaines de même taille selon leur proximité à l'une ou l'autre extrémité. Enfin l'algorithme est répété récursivement sur chaque sous-domaine jusqu'à obtention d'un nombre voulu de partitions. La principale difficulté de ce type de méthode est de déterminer les sommets les plus distants dans le graphe. Des heurisitiques existent pour trouver ces sommets en temps acceptable. C'est le cas de l'algorithme *Reverse Cuthill-McKee* [45], qui recherche deux sommets séparés par une grande distance sans être forcément de distance maximale.

Les algorithmes de type ***Greedy*** (GR) [38] sont assez semblables aux méthodes RGB. Cette fois, le sommet de plus faible degré est considéré comme le sommet de départ. À partir de ce sommet, tous les sommets voisins sont marqués, puis les voisins des voisins et ainsi de suite. Lorsque n/p sommets ont été marqués, ces sommets forment un sous-domaine et on recommence l'algorithme sur le reste du graphe jusqu'à ce que tous les sommets soient marqués. Cet algorithme donne des résultats très semblables à l'algorithme RGB. De plus, différentes stratégies peuvent être utilisées lors du marquage de nouveaux sommets, par exemple en utilisant des notions de gain pour ajouter les sommets par ordre croissant de l'augmentation de la cut-size [69].

Les méthodes spectrales ou de type ***Recursive Spectral Bisection*** (RSB) sont des approches beaucoup moins intuitives. Ces techniques sont largement utilisées [92, 55] et souvent considérées comme donnant les meilleurs résultats en matière de qualité du partitionnement. Le principe repose sur le calcul des valeurs propres de la *matrice de Laplace* associée au graphe. La matrice de Laplace L d'un graphe G contient les informations sur la connexité de G. C'est une matrice carrée de taille $|V| \times |V|$ définie par :

$$L_{i,j} = \begin{cases} \delta(v_i) & si\ i = j \\ -1 & si\ (v_i, v_j) \in E \\ 0 & sinon \end{cases}$$

Une fois la matrice construite, le vecteur propre x associé à la deuxième plus petite valeur propre λ de L est calculé. Ce vecteur propre, appelé *vecteur de Fiedler* [41], permet d'ordonner les sommets du graphe sur une droite tout en assurant que les sommets fortement connectés dans le graphe sont proches dans la droite. Les sommets sont alors triés dans l'ordre croissant des composantes correspondantes dans le vecteur de Fiedler. Ensuite il suffit de placer les $n/2$ premiers sommets dans un sous-domaine et le reste des sommets dans l'autre sous-domaine puis de recommencer récursivement sur chaque sous-domaine. L'approche spectrale fournit une bonne qualité de partitionnement, en revanche elle est très coûteuse, notamment pour le calcul du vecteur de Fiedler [9]. Son coût implique qu'elle est principalement utilisée pour effectuer un partitionnement statique en amont de la simulation.

Améliorations locales

Les algorithmes vus précédemment permettent tous de construire un nouveau partitionnement indépendamment de tout résultat préalable. Une approche différente consiste à partir d'un partitionnement valide et bien équilibré et d'essayer d'en améliorer chaque coupe en terme de cut-size. De nombreuses méthodes effectuent ces améliorations en se basant sur la notion de *gain* introduite par Kernighan et Lin [72]. Ce gain, pour un sommet donné, correspond à la réduction du nombre d'arêtes coupées par une bissection particulière, quand ce sommet est échangé de sa partition actuelle vers l'autre partition. Pour un graphe et une bissection donnés, le coût interne $int(v_i)$ d'un sommet v_i est alors défini comme le nombre d'arêtes (ou la somme des poids des arêtes si celles-ci sont pondérées) incidentes à v_i et dont l'autre sommet appartient à la même partition que v_i. De même le coût externe $ext(v_i)$ correspond au nombre d'arêtes dont l'autre sommet appartient à une autre partition. Le gain du déplacement de v_i vers l'autre partition est alors $g(v_i) = ext(v_i) - int(v_i)$. Le gain pour l'échange de deux sommets v_i et v_j appartenant à des partitions distinctes est la somme du gain pour chaque sommet moins deux fois l'arête (v_i, v_j) (ou deux fois son poids si elle est pondérée).

L'algorithme ***Kernighan-Lin*** (KL) [72] est un algorithme itératif d'amélioration locale basé sur cette notion de gain. Cette approche cherche à déterminer pour chacune des deux partitions des sous-ensembles de sommets dont l'échange donne un gain maximum. Initialement, chaque sommet est non-marqué. Dans un premier temps, le couple de sommets qui donne un gain maximum est marqué intransférable et ce gain (qui peut être négatif pour permettre dans une certaine mesure de sortir d'un minimum local) est enregistré. On répète cette étape sur les sommets non-marqués jusqu'à ce que tous les sommets du graphe soient marqués. Ensuite on détermine k tel que la somme des gains des k premiers échanges (dans l'ordre d'enregistrement des gains) soit maximum. On annule alors les échanges ultérieurs à k et on recommence jusqu'à ce que le maximum de la somme des gains soit négatif. Une somme des gains négative signifie que l'on a atteint un optimum local. Cet algorithme permet d'obtenir une bonne amélioration de la qualité de partition mais il reste très coûteux (en $\mathcal{O}(n^3)$, avec n le nombre de sommets). Une variante proposée dans [69] ne prend en compte que les sommets qui se trouvent sur la frontière, ce qui permet de réduire le coût de la méthode mais pour un résultat de moindre qualité.

L'algorithme de Fiduccia et Mattheyses (FM) [40] est une variante en temps quasi-linéaire de KL pour les graphes dont les arêtes ne sont pas valuées. Cet algorithme ne cherche pas à échanger des paires de sommets mais procède en trouvant et déplaçant les sommets de plus grand gain l'un après l'autre. Cela peut provoquer un déséquilibre de la partition, c'est pourquoi les mouvements d'une partition à une autre ne sont autorisés que s'ils ne perturbent pas l'équilibre au-delà d'une certaine tolérance. La structure de donnée utilisée (à base de listes chaînées) permet de trouver le meilleur sommet à déplacer en temps constant. L'algorithme FM a été adapté aux graphes valués, et il a été modifié pour pouvoir considérer k partitions au lieu d'une seule bissection. Pour effectuer cette modification, les auteurs de [54] ont généralisé la notion de gain pour que celle-ci prenne en compte chaque partition. Par contre il est à noter que les algorithmes FM de même que KL peuvent être amenés à effectuer un choix entre plusieurs échanges de sommets qui offrent un même gain. Ce choix, souvent arbitraire, peut détériorer le résultat obtenu. Pour cette raison, plusieurs exécutions de l'algorithme sont habituellement réalisées, pour ne conserver au final que la meilleure des solutions.

Méthodes multi-niveaux

Actuellement, les méthodes les plus utilisées pour effectuer le partitionnement de graphes sont sans doute celles qui utilisent l'approche ***multi-niveaux*** [9, 54, 70]. Cette approche peut être comparée à l'approche multi-grilles utilisée dans les solveurs d'équations linéaires [18]. Un algorithme de partitionnement multi-niveaux se décompose généralement en trois phases :

- la *phase de contraction* du graphe, durant laquelle la taille du graphe (le nombre de sommets) est réduite un certain nombre de fois jusqu'à atteindre une taille acceptable.
- la *phase de partitionnement* du graphe, durant laquelle le graphe, réduit lors de la phase de contraction, est partitionné.
- la *phase d'expansion* du graphe, durant laquelle le partitionnement est projeté progressivement vers le graphe initial. C'est durant cette dernière phase que l'on pourra ajouter un algorithme d'optimisation locale (notamment de type KL) pour améliorer le partitionnement à chaque étape.

L'enchaînement de ces trois phases ainsi que l'amélioration de la coupe lors de la projection sont schématisés par la figure 2.8 pour un 2-partitionnement.

La phase de contraction du graphe consiste à transformer un graphe G en une succession de graphes $G_1, G_2, ..., G_k$ contenant chacun moins de sommets que son prédécesseur ($|V_1| > |V_2| > ... > |V_k|$). Transformer un graphe en un graphe réduit consiste en général à regrouper deux à deux les sommets du graphe G_i. Chacune de ces paires est alors remplacée par un sommet G_{i+1} dont le poids est égal à la somme des poids de la paire de sommets qu'il remplace. Si l'appariement des sommets fait apparaître des arêtes multiples, celles-ci sont remplacées par une arête simple dont le poids est égal à la somme des poids initiaux. Le choix du sommet v_j à apparier à un sommet v_i peut être fait de plusieurs façons. Le sommet v_j peut être choisi aléatoirement parmi l'ensemble des sommets voisins

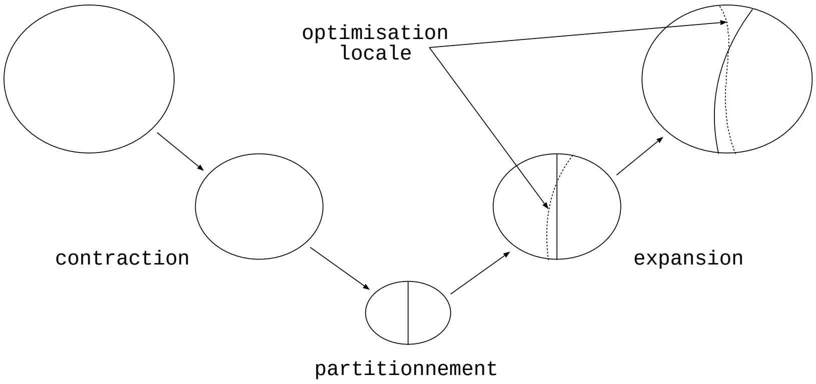

Fig. 2.8 – Représentation schématique de l'approche multi-niveaux pour le partitionnement d'un graphe.

de v_i, ou alors tel que l'arête (v_i, v_j) soit l'arête incidente à v_i ayant le poids le plus grand. Cette dernière méthode, introduite dans [69] sous le nom de *Heavy Edge Matching*, permet de mieux préserver la topologie du graphe, ce qui est un des points-clé pour obtenir un partitionnement de bonne qualité.

La phase de partitionnement considère uniquement le graphe le plus réduit G_k (on parlera aussi de graphe le plus *grossier*). Le nombre de sommets étant raisonnable dans ce graphe réduit, des heuristiques globales comme GR, RSB, ou même des algorithmes génétiques [2], peuvent être appliqués sans que le temps de calcul engendré ne soit trop important. Comme la phase de contraction préserve le poids total du graphe, il est à noter que l'équilibre atteint par le partitionnement de G_k sera identique pour chaque maillage intermédiaire et pour le graphe initial G.

Enfin, la phase d'expansion projette le partitionnement calculé pour le graphe G_k vers le graphe initial G. Cette projection est effectuée étape par étape, en projetant la solution d'un niveau sur le niveau précédent. À chaque projection, le partitionnement peut être raffiné à l'aide de méthodes d'améliorations locales comme KL ou FM. Il est à noter qu'un optimum local trouvé pour G_{i+1} par une méthode d'amélioration ne sera pas forcément le même pour G_i. Cependant, si la topologie de G_{i+1} est proche de celle de G_i, les deux solutions devraient être proches l'une de l'autre.

2.3.3 Méthodes locales

Contrairement aux méthodes globales de partitionnement, les méthodes locales se basent sur une vision volontairement réduite de la charge de l'application. Au lieu de considérer l'ensemble des processeurs sur lesquels tourne l'application, les méthodes locales cherchent à équilibrer la charge par petits *groupes* de processeurs. Ces groupes peuvent être formés en fonction de l'architecture de la machine cible ou en fonction des dépendances entre les données de l'application. De plus les différents groupes doivent se recouvrir pour permettre de transférer des données entre les groupes. Le principe des méthodes locales est le suivant : les processeurs les plus chargés transmettent une partie de leurs données aux processeurs les moins chargés appartenant au même groupe. Ces méthodes se décomposent habituellement en deux étapes :

- Déterminer la quantité de charge à faire migrer des processeurs les plus chargés vers les processeurs les moins chargés.

– Sélectionner les objets qui vont être effectivement transmis.

Une fois ces deux étapes réalisées, les objets sélectionnés sont transmis au cours d'une phase de diffusion. Le principe de cette diffusion est de n'effectuer les transmissions de données qu'avec des processeurs voisins pour que la redistribution des données soit locale. Si des objets doivent être transmis à des processeurs plus distants, il faut alors itérer cette phase de diffusion. Quand il n'y a que de légères variations dans la répartition de la charge de calcul, peu d'itérations de l'algorithme de diffusion sont nécessaires pour équilibrer la charge. Par contre, si la charge évolue rapidement et fortement, il se peut qu'un nombre important d'itérations soit nécessaire, rendant la méthode locale moins efficace qu'une méthode globale.

Il existe plusieurs méthodes pour déterminer la quantité de charge qui doit être déplacée (on parle aussi de migration des données entre les processeurs). Certaines méthodes proposent l'utilisation d'algorithmes de *diffusion* en se basant sur la connectivité entre les processeurs ou sur le schéma de communication de l'application. Pour ces algorithmes, le flux de la charge est représenté de manière locale [31] en résolvant une équation de type diffusion de la chaleur par un schéma de différences finies d'ordre 1. Des variations de cette méthode proposent soit d'en améliorer la convergence soit de réduire la quantité de données à déplacer. Pour améliorer la convergence de la diffusion, des travaux proposent l'utilisation de schémas de différences finies d'ordre supérieur [116]. D'autres travaux proposent de répartir virtuellement les processeurs sur un hypercube et de transmettre la charge en alternant entre les voisins dans chaque dimension [33] (*dimensional exchange*).

Pour réduire la quantité de données à déplacer, des travaux ont cherché à minimiser le flux de la charge représenté par les arêtes valuées d'un graphe des processeurs [64]. Cette méthode nécessite la résolution d'un système $L\lambda = b$, où L est la matrice de Laplace, b est un vecteur donnant pour chaque processeur la différence avec la charge moyenne, et λ est un vecteur des multiplicateurs de Lagrange. La valeur des coefficients multiplicateurs de Lagrange est utilisée pour déterminer la quantité de charge à transférer entre chaque paire de processeurs. D'autres méthodes, les méthodes *à la demande* [32, 85], transmettent la charge des processeurs surchargés vers les processeurs sous-chargés. Le transfert de la charge peut être initié soit par les récepteurs, soit par les émetteurs [118]. Ces méthodes ne considèrent qu'un sous-ensemble des processeurs alors que les méthodes par diffusion considèrent l'ensemble des processeurs voisins.

Une fois la quantité des objets à faire migrer calculée, la sélection de ces objets se fait le plus souvent selon une notion de gain pour le déplacement d'un objet vers un autre processeur. Ce gain, inspiré des algorithmes de type KL est généralement utilisé pour réduire la taille de la cut-size en s'intéressant à l'impact de la migration d'un objet sur le nombre d'arêtes coupées du sous-domaine considéré. Il existe néanmoins des variantes de cette notion de gain, permettant par exemple de réduire la quantité de données transférées. En plus des objets qui permettent de réduire la cut-size tout en maintenant l'équilibre et des objets permettant de maintenir la cut-size tout en améliorant l'équilibre, les auteurs de [97] autorisent la sélection d'objets qui maintiennent cut-size et équilibre si le transfert de ces objets se fait vers leur processeur initial. Une autre variante permet de considérer la forme de la partition dans la sélection des objets à déplacer. Le gain peut alors dépendre de l'impact du déplacement d'un objet sur l'équilibre et sur l'aspect de la partition construite. Par exemple, les objets sélectionnés en premier lieu sont ceux qui sont les plus éloignés des coordonnées moyennes de l'ensemble des objets du domaine [112]. Les performances des différentes variantes utilisées dépendent fortement du type d'application considéré.

2.3.4 Analyse des différentes méthodes de partitionnement

Les méthodes géométriques sont basées uniquement sur les coordonnées des éléments à partitionner. Les méthodes effectuant des bissections parallèles aux axes de coordonnées sont très rapides et implicitement incrémentales. L'incrémentalité est qualifiée d'implicite dans le sens où cette propriété n'est pas garantie (on parlera alors d'incrémentalité explicite) mais elle est vérifiée la plupart du temps car inhérente à la méthode. De plus ces méthodes décrivent des partitions rectangulaires, ce qui rend plus aisé le calcul du nouveau schéma de communications. Ces propriétés en font de bons candidats pour un mécanisme d'équilibrage dynamique de charge. Par contre, la qualité des décompositions générées par ces techniques est souvent plus faible qu'avec d'autres techniques, que cela soit en terme d'équilibre de la charge de calcul et surtout en terme de réduction de la quantité de communications. Cette remarque est encore plus vraie pour des domaines à géométrie complexe. Les méthodes à coupes rectilignes non parallèles aux axes permettent généralement une meilleure qualité de décomposition, notamment en présence de géométries complexes. Par contre elles ne sont plus incrémentales, ce qui peut rendre ces techniques mal adaptées à un mécanisme d'équilibrage dynamique. Enfin, les méthodes à base d'octree ou de courbes de remplissage ont des avantages assez semblables aux méthodes par bissections parallèles aux axes et peuvent donc être utilisées pour un mécanisme d'équilibrage dynamique. Elles nécessitent néanmoins souvent un surcoût en mémoire et la décomposition peut être de moindre qualité.

En ce qui concerne les méthodes basées sur les graphes, les méthodes globales comme RGB et les algorithmes de type GR offrent en général une qualité de décomposition légèrement supérieure aux méthodes RIB. Elles en partagent d'ailleurs le principal défaut, à savoir ne pas être incrémentales. Les deux méthodes sont assez peu coûteuses, mais la méthode GR est difficile à paralléliser contrairement à RGB. Enfin, la méthode GR respecte mieux l'aspect des partitions, mais les dernières partitions calculées ont tendance à être non connexes. L'approche spectrale est celle qui offre la meilleure qualité de partitionnement. Elle n'est pas incrémentale mais il existe des versions modifiées permettant d'inclure cette propriété [36]. Par contre, ces méthodes sont très coûteuses en calcul et donc très lentes. Elles sont utilisées largement pour effectuer un partitionnement initial, mais rarement dans le cadre d'un partitionnement dynamique au cours de l'exécution du programme. Les méthodes multi-niveaux permettent de réduire le coût d'un tel algorithme de partitionnement en l'exécutant sur un graphe de taille réduite puis en projetant la solution vers le graphe initial. Ces méthodes sont très populaires et sont actuellement implantées dans la plupart des outils de partitionnement statique [53, 68, 93, 90]. Leur parallélisation n'est pas facile, mais elle doit permettre leur utilisation en cours d'exécution du programme. Les diverses méthodes basées sur les graphes peuvent être combinées avec des algorithmes d'améliorations locales, qui cherchent à minimiser le nombre d'arêtes coupées par le partitionnement. Ces méthodes d'amélioration locales sont basées sur des notions de gain dérivées du gain de KL et permettent dans certains cas d'obtenir des solutions optimales localement.

Enfin, les méthodes locales se basent sur une décomposition existante et sont donc complémentaires d'une méthode d'équilibrage statique. Elles nécessitent l'accès aux éléments voisins et sont donc adaptées si l'information sur la connectivité des éléments est disponible, ce qui les rapproche des méthodes basées sur graphe. Contrairement à ces dernières, les méthodes locales sont incrémentales par construction, et par conséquent elles sont plutôt bien adaptées à un équilibrage dynamique. Ces méthodes sont peu coûteuses mais nécessitent en général plusieurs itérations pour atteindre l'équilibre sur l'ensemble du domaine de calcul. Elles peuvent par conséquent devenir coûteuses quand la charge de calcul évolue beaucoup ou rapidement : cela peut engendrer un grand nombre d'itérations et une grande quantité d'échanges de données. Il est alors préferable d'utiliser une méthode globale.

Plusieurs études [65, 52, 106, 63] ont cherché à comparer les différentes méthodes de partitionnement décrites précédemment. La conclusion commune de ces études est qu'il n'existe pas de meilleure méthode. Il ressort de ces travaux que l'efficacité réelle d'une stratégie de partitionnement est corrélée à la tolérance de l'application concernant sa rapidité, sa qualité d'équilibrage ou sa stabilité. Par exemple, une application qui effectue un unique partitionnement tout au long de la simulation pourra faire appel à des stratégies qui peuvent être très coûteuses mais qui fournissent un partitionnement de grande qualité, proche de l'optimal. Au contraire, une application hautement adaptative doit donner la priorité à la rapidité d'exécution du partitionneur et à sa capacité à ne pas générer trop de messages plutôt qu'à la qualité de l'équilibrage obtenu. Il faut de même considérer le degré d'informations accessibles par l'application. Par exemple, une application qui ne fournit pas de base une information sur la connectivité des éléments utilisera en général plus avantageusement les stratégies géométriques plutôt que les stratégies à base de graphes.

Concernant la qualité de la décomposition intrinsèque des différentes méthodes, les méthodes à base de graphes offrent en général une bien meilleure qualité de partitionnement que les méthodes géométriques. Cependant, la difficulté de leur implantation en parallèle, et leur coût parfois important en calcul en font un candidat parfois délicat à mettre en œuvre dans un mécanisme d'équilibrage de charge dynamique. Cette remarque est d'autant plus vraie si l'application visée ne permet pas de fournir facilement et à moindre coût l'information sur la connectivité des éléments.

Deuxième partie

Schéma adaptatif avec prédiction du maillage *en avant*

Chapitre 3

La méthode numérique YODA

Dans la première partie de cette thèse, nous avons vu que la résolution de l'équation de Vlasov est un problème extrèmement difficile. Cette difficulté vient de la non linéarité du système, de son évolution en temps et du grand nombre de dimensions. Pour répondre au défi posé par la taille du système, des méthodes adaptatives ont été envisagées depuis quelques années.

Dans ce chapitre nous allons présenter la méthode numérique de résolution de l'équation de Vlasov dont nous étudierons la parallélisation par la suite. Cette méthode numérique, appelée YODA [20] (Yet anOther aDaptive Algorithm), est une méthode semi-Lagrangienne basée sur un maillage adaptatif de l'espace des phases. Elle a été conçue en 2003 par Martin Campos Pinto et Michel Mehrenberger. Le lecteur pourra consulter la thèse de Michel Mehrenberger [80] pour de plus amples informations sur les fondements mathématiques de cette méthode. Dans un premier temps, nous allons présenter la décomposition en éléments finis hiérarchiques et le maillage adaptatif sous-jacent. Puis nous nous intéresserons au schéma de résolution en temps, qui comporte une méthode originale d'adaptation directe (sans itération) du maillage.

3.1 Motivations

Nous avons vu en section 1.3 que les méthodes numériques basées sur des maillages, comme les méthodes semi-Lagrangiennes, permettent généralement d'atteindre une meilleure précision que les méthodes particulaires traditionnelles. Leur principal défaut est le coût important de stockage et de traitement d'un maillage posé dans l'espace des phases. En effet, le nombre de points de discrétisation croît de manière exponentielle avec le nombre de dimensions. Pour donner un ordre de grandeur sur l'utilisation mémoire, voici l'espace mémoire requis pour stocker (en double précision) la fonction de distribution discrétisée sur différentes tailles de grille uniforme en fonction du nombre de dimensions :

- en 2D, 512 points par dimension ⇒ 2Mo de données
- en 4D, 256 points par dimension ⇒ 32Go de données
- en 6D, 128 points par dimension ⇒ 32To de données

De plus, les phénomènes simulés sont bien souvent inhomogènes et exhibent des zones à fort gradient (ou singularité) qui nécessitent localement une discrétisation de l'espace des phases par un nombre important de points. Bien souvent (surtout en physique des faisceaux de particules), des zones à faible gradient sont également présentes. Par conséquent, l'utilisation d'un maillage uniforme implique que de nombreux nœuds, nécessaires dans les zones à fort gradient, sont gaspillés dans les zones à faible gradient, ce qui rend la méthode numérique peu efficace. Le recours à des méthodes adaptatives doit permettre d'améliorer considérablement l'efficacité des méthodes numériques sur maillage. Leur principe est d'utiliser un maillage non-uniforme et en adéquation avec la distribution des particules.

Élaborer une méthode adaptative utilisant le schéma semi-Lagrangien est un problème difficile. En effet, le maillage doit être adapté à toutes les étapes de temps et son évolution en temps doit prendre en compte l'apparition de singularités et de phénomènes locaux complexes.

Différentes méthodes adaptatives ont été développées pour la résolution de l'équation de Vlasov. On peut citer notamment [104], où une grille mobile de résolution fixe est utilisée. Dans cette grille, l'espace entre les points de discrétisation est constant et c'est la position et la taille de la grille qui évoluent au cours de la simulation. Comme la grille change de position au cours du temps, subissant notamment des rotations, le splitting en temps de l'équation n'est pas applicable. De plus, la grille reste à un niveau de précision constant, ce qui engendre un gaspillage de points de discrétisation dans les zones à faible gradient, même si celles-ci sont moins nombreuses qu'avec un domaine fixe. Une autre méthode, introduite dans [14, 48] utilise une décomposition de la fonction de distribution sur une base d'ondelettes à différents niveaux de résolution. Cette méthode permet de réduire considérablement le nombre de points de discrétisation, les ondelettes étant bien connues pour leurs qualités en terme de compression de données. L'inconvénient de cette méthode vient de la difficulté de parallélisation : le support des bases d'ondelettes est assez large et par conséquent il rend difficile le partitionnement des données en vue de leur distribution sur les processeurs. Cette difficulté implique notamment que les parallélisations existantes de ces méthodes ont à l'heure actuelle essentiellement pour cible des architectures à mémoire partagée [50, 30]. La méthode YODA a été conçue dans l'optique d'une parallélisation par décomposition de domaine pour des machines à mémoire distribuée.

Le reste du chapitre est une description détaillée de la méthode YODA. Pour simplifier la présentation, la méthode est donnée dans le cas d'un espace des phases réduit à deux dimensions, mais elle peut facilement être généralisée à un nombre plus grand de dimensions. Le chapitre est organisé comme suit : la section 3.2 présente le type de maillage utilisé et de façon plus générale il montre comment la solution est répresentée à chaque étape de temps. La section 3.3 traite du point délicat de l'adaptation du maillage et la dernière section 3.4 donne le schéma de résolution complet qui se scinde en deux variantes : avec ou sans splitting.

3.2 Représentation de la solution

La méthode adaptative YODA est une méthode semi-Lagrangienne "locale" au sens où la solution peut être reconstruire sur chaque maille uniquement à partir des informations de la maille. Les sections suivantes décrivent les éléments de la méthode sur lesquels repose cette propriété.

3.2.1 Maillage

Le maillage de l'espace des phases, qui est utilisé dans la méthode, est un maillage *dyadique*. Un maillage dyadique est obtenu par subdivision du domaine en base 2 selon

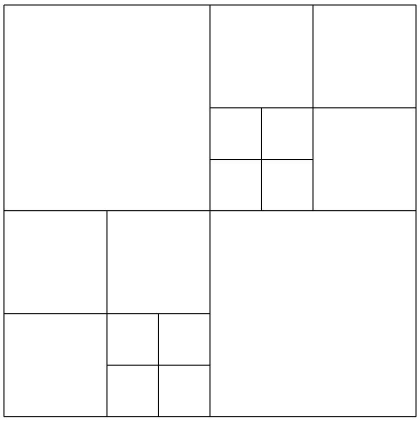

FIG. 3.1 – Maillage dyadique 2D obtenu par subdivision en base 2 du domaine.

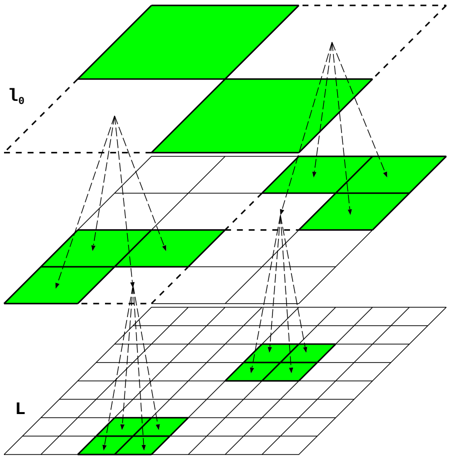

FIG. 3.2 – Construction hiérarchique d'un maillage dyadique.

toutes les dimensions. Un exemple de tel maillage est donné sur la figure 3.1. Chaque maille, on dit aussi *cellule*, appartient à une grille uniforme de taille $2^l \times 2^l$, où l est un entier positif appelé *niveau* de la grille ou de la cellule. Un tel maillage est caractérisé par deux entiers l_0 et L, qui correspondent respectivement au niveau le plus grossier et au niveau le plus fin. Par exemple, le maillage dyadique de la figure 3.1 vérifie $l_0 = 1$ et $L = 3$. Comme on le voit sur la figure 3.2, le maillage est construit hiérarchiquement en subdivisant certaines cellules à chaque niveau. Ainsi, chaque cellule de niveau $l < L$ peut être raffinée en 4 cellules de niveau $l + 1$. On introduit ainsi une filiation de cellules : la cellule de niveau l est appelée cellule *mère* en référence aux cellules de niveau $l + 1$ qui sont appelées cellules *filles*. Par extension, des cellules issues du raffinement d'une même cellule mère sont appelées cellules soeurs.

Notons que le raffinement d'une cellule signifie que la cellule est supprimée du maillage et remplacée par ses cellules filles. Il n'y a pas de recouvrement entre les cellules et l'union des cellules filles occupe le même espace que la cellule mère. Par extension, l'ensemble des cellules du maillage forme une partition du domaine. Cette propriété dite de *consistance* du maillage garantit la bonne définition de l'opération inverse du raffinement d'une cellule : le déraffinement qui consiste à remplacer les quatres cellules filles par la cellule mère. Les cellules seront notées par la suite en utilisant les lettres grecques α ou β. Chaque cellule est identifiée par trois indices entiers : son niveau l et ses deux coordonnées $i, j \in \{0, .., 2^l - 1\}^2$ dans la grille uniforme de niveau l.

À chaque cellule $\alpha_{l,i,j}$ sont associés 9 *nœuds* (3 par dimension) répartis uniformément comme le montre la figure 3.3. Les noeuds d'une cellule sont indicés par $N_{l,i,j}^{r,s}$ avec $r, s = 0, 1, 2$ (voir la figure 3.3). Il est important de noter que des valeurs différentes des indices (l, i, j), r et s peuvent désigner un même nœud. La plupart des nœuds sont en effet partagés par plusieurs cellules voisines, comme le montre la figure 3.4. Cela découle du fait que la majorité des nœuds d'une cellule sont situés sur ses bords. On verra par la suite que ceci est important pour l'implantation de la méthode, en particulier pour déterminer la structure de donnée à utiliser pour représenter le maillage.

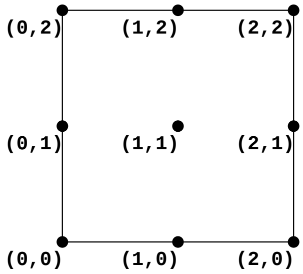

FIG. 3.3 – Position des noeuds dans une cellule.

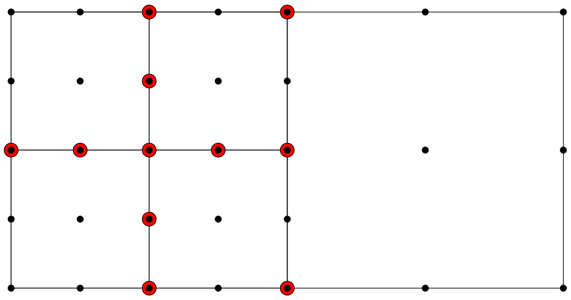

FIG. 3.4 – Les noeuds entourés sont partagés par plusieurs cellules voisines.

3.2.2 Élements finis de Lagrange

La méthode utilise une base de polynômes de Lagrange d'ordre 2 pour reconstruire la solution en tout point de chaque maille à partir des valeurs aux noeuds de la maille. Nous donnons ici la définition de cette base de polynômes d'interpolation. Nous reprenons pour cela la définition classique des éléments finis de Lagrange.

On se place dans le cas 1D : les définitions se généralisent en dimension en combinant les opérateurs par produit tensoriel. Un élément fini de Lagrange est défini par un triplet (K, P, Σ) :

- K est un élément géométrique simple (dans notre cas des quadrangles),
- P est un espace de polynômes (dans notre cas des polynômes de degré 2),
- Σ est un ensemble de points de K (appelés *degrés de liberté*).

L'élément fini (K, P, Σ) est qualifié d'unisolvant si le fait de spécifier une valeur pour chacun des degrés de liberté de Σ détermine de manière unique une fonction de $p \in P$. Soient $\Sigma = l_i, 1 \leq i \leq m$, où $m = \dim P = \dim \Sigma$ et $\{\phi_j\}_{1 \leq j \leq m} \subset P$, l'ensemble des *fonctions de base* telles que $l_i(\phi_j) = \delta_{ij}$, où δ_{ij} est le symbole de Kronecker défini par

$$\delta_{ij} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{sinon} \end{cases}$$

Alors, l'unisolvance implique que toute fonction de P s'écrit de manière unique sur la base des $\{\phi_j\}_{0 \leq j \leq m-1}$:

$$p(x) = \sum_{j=0}^{m-1} l_j(p)\phi_j(x), \ \forall p \in P \tag{3.1}$$

Pour les éléments finis de Lagrange, les fonctions de base valent 1 pour un noeud et 0 pour tous les autres. Pour des polynômes de degrés 2 et en 1D, on obtient donc les fonctions de base suivantes (représentées sur la figure 3.5) :

$$\left\{ \begin{array}{lcl} \phi_0(x) & = & 2\,(x - \frac{1}{2})\,(x-1) \\ \phi_1(x) & = & 4x\,(1-x) \\ \phi_2(x) & = & 2x\,(x - \frac{1}{2}) \end{array} \right. \tag{3.2}$$

Pour une maille donnée, le polynôme d'interpolation de degré 2 est donc défini par

$$p(x) = l_0(p)\phi_0(x) + l_1(p)\phi_1(x) + l_2(p)\phi_2(x), \ \forall p \in P, x \in [0, 1[$$

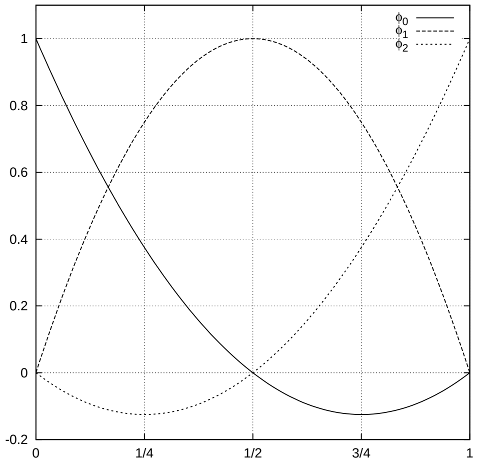

FIG. 3.5 – Représentation graphique des fonctions de base sur une cellule grossière.

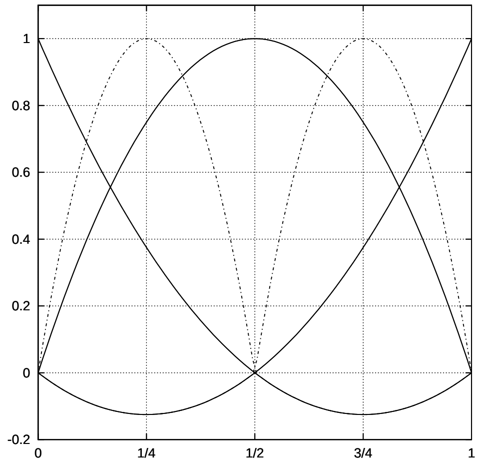

FIG. 3.6 – Ajout des fonctions de base aux noeuds issus des cellules fines obtenues par raffinement (en hachuré).

où les $l_i(p)$ sont les valeurs de p données aux noeuds de la maille.

Le polynôme d'interpolation pour un niveau fin est construit hiérarchiquement à partir du niveau supérieur jusqu'au niveau grossier. Le polynôme de niveau grossier est défini par les 3 fonctions de bases (3.2). Lors du raffinement d'une cellule (1D) au niveau grossier, deux nouvelles cellules sont créées : chacune contient un seul nouveau noeud au centre de l'élément. Cela implique que pour chaque nouvelle cellule raffinée, on introduit une unique fonction de base supplémentaire. Cette fonction vaut 1 au nouveau point et 0 aux points existants. La figure 3.6 décrit l'ajout des fonctions de base supplémentaires lors du raffinement d'une cellule grossière.

3.2.3 Représentation hiérarchique

La méthode YODA utilise une représentation hiérarchique de la solution. Le principe de cette représentation repose sur le calcul des *détails*. Le détail est la différence entre deux valeurs : la valeur de la fonction approchée par un polynôme à un niveau grossier l, et la valeur de la fonction approchée par un polynôme au niveau plus fin $l+1$. Ces détails sont calculés en chaque point correspondant à un noeud rajouté lors du raffinement d'une cellule. Le calcul des détails permet d'évaluer la perte de précision induite par l'approximation de la fonction à une résolution plus grossière. Si tous les détails, pour un ensemble de cellules soeurs au niveau $l+1$, sont inférieurs à un seuil donné, alors la fonction est approchée de manière suffisamment précise par la cellule mère de niveau l et il inutile de raffiner cette cellule.

La figure 3.7 décrit l'approximation d'une fonction f, par une hiérarchie de polynômes d'ordre 2. Chaque polynôme est défini sur un élément contenant 3 noeuds et a pour valeur, en chacun de ces noeuds (en rouge), la valeur de la fonction . Au niveau grossier (maille de taille $4h$), le polynôme d'interpolation p_l (dessiné en trait plein) est défini par les valeurs de f en $x_{4k}, x_{4k+2}, x_{4k+4}$. Au niveau plus fin (deux mailles de taille $2h$), les deux polynômes p^0_{l+1} et p^1_{l+1} (dessinés en pointillés) sont définis respectivement par les valeurs en $x_{4k}, x_{4k+1}, x_{4k+2}$ et $x_{4k+2}, x_{4k+3}, x_{4k+4}$. Les détails sont calculés aux points qui existent

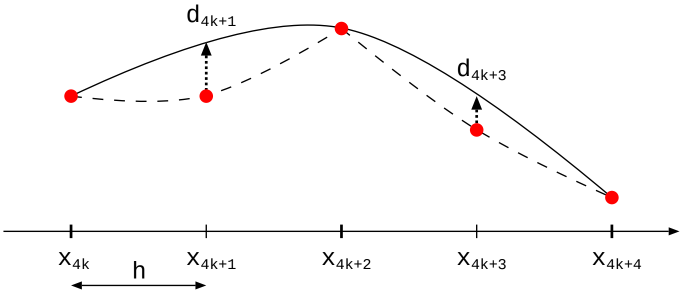

FIG. 3.7 – Approximation de la fonction par une hiérarchie de polynômes d'ordre 2 et calcul des détails.

au niveau fin mais qui ne sont pas définis au niveau grossier :

$$\begin{aligned} d_{4k+1} &= |p_l(x_{4k+1} - f(x_{4k+1}))| \\ d_{4k+3} &= |p_l(x_{4k+3} - f(x_{4k+3}))| \end{aligned}$$

Ces détails sont comparés à un seuil ϵ qui permet de décider si l'approximation au niveau fin doit être conservée ou si elle est inutile et pourra le cas échéant être reconstruite avec une précision suffisante.

Dans la méthode YODA, les détails ne sont pas conservés : ils sont seulement utiles au moment de déterminer si une cellule doit être raffinée ou non. Dans d'autres méthodes, au contraire, les détails sont conservés et sont utilisés pour reconstruire la fonction aux niveaux fins à partir des valeurs aux niveaux grossiers [47].

3.3 Adaptation du maillage

Nous décrivons ici les modifications qui s'opérent sur le maillage à chaque étape de temps dans le but de suivre l'évolution en temps de la simulation. On peut se référer à la thèse de Martin Campos Pinto pour l'analyse de tels schémas d'adaptation du maillage [19].

L'adaptation du maillage d'un pas de temps t_n au pas de temps suivant t_{n+1} s'effectue en deux phases.

1. La première phase consiste à prédire un nouveau maillage au temps t_{n+1}. Une fois les cellules du maillage prédit déterminées, les valeurs aux noeuds de ces cellules sont calculés selon le schéma semi-Lagrangien (en suivant les courbes caractéristiques en arrière). À l'issue de cette première phase, on obtient donc une nouvelle représentation hiérarchique de la solution au temps t_{n+1}. Durant cette phase, il est essentiel que le maillage prédit contienne suffisamment de cellules de sorte que la précision recherchée pour la solution approchée au temps t_{n+1} soit garantie. C'est pourquoi le maillage prédit contient en général plus de cellules que nécessaire. On dit que la prédiction est *pessimiste*.
2. Dans la deuxième phase, le maillage prédit est "corrigé" pour mieux correspondre à la solution approchée au temps t_{n+1}. Compte tenu de la précision recherchée, certaines cellules du maillage prédit sont inutiles et sont donc supprimées par déraffinement. Cette phase a pour objectif de réduire le nombre de points de discrétisation.

Ces deux phases sont détaillées dans les paragraphes suivants. Nous utiliserons les notations suivantes : on notera f^n, la solution approchée au temps t^n, $\tilde{\mathcal{M}}^n$, le maillage au temps t_n et $\tilde{\mathcal{F}}^n$, la fonction qui donne la valeur en tout noeud de $\tilde{\mathcal{M}}^n$. Ainsi, le couple $(\tilde{\mathcal{M}}^n, \tilde{\mathcal{F}}^n)$ désigne la représentation hiérarchique de f^n.

3.3.1 Prédiction du maillage

Dans la méthode YODA, la prédiction du maillage repose sur l'utilisation des courbes caractéristiques qui correspondent aux trajectoires des particules dans l'espace des phases. Le principe de cette prédiction est de "transporter" les cellules du maillage au temps t_n vers le temps t_{n+1} en suivant les caractéristiques en avant. Dans la suite, on notera $\tilde{\mathcal{M}}^{n+1}$, le maillage prédit, et $\mathcal{A}^+$ l'opérateur d'advection en avant qui permet de calculer le point au temps t_{n+1} étant donné le point au temps t_n sur la même trajectoire ou courbe caractéristique.

Le transport d'une cellule ne peut pas être réalisé de manière exacte : les cellules sont toujours de forme carrée alors que leur transport en suivant les caractéristiques a tendance à les déformer. Il faut donc approcher le transport des cellules en utilisant une heuristique. L'heuristique retenue pour approcher le transport d'une cellule consiste à appliquer l'opérateur d'advection en avant sur son centre. Ainsi, si on note α, cette cellule et c son centre, on fait correspondre à α, l'unique cellule β contenant le point $\mathcal{A}^+(c)$ et de même taille ou niveau que α.

Le maillage prédit $\tilde{\mathcal{M}}^{n+1}$ est donc construit à partir du "maillage vide" (réduit à la cellule de niveau 0) en y insérant successivement chacune des cellules β correspondant à une cellule α du maillage $\tilde{\mathcal{M}}^n$. Cette opération d'insertion d'une cellule est définie de manière à préserver la consistance du maillage. Il y a deux cas à considérer lors de l'insertion d'une cellule β dans le maillage en construction :

- s'il existe une cellule β' de $\tilde{\mathcal{M}}^{n+1}$ telle que $\beta' \subseteq \beta$, alors le maillage reste inchangé.
- s'il existe une cellule β' de $\tilde{\mathcal{M}}^{n+1}$ telle que $\beta \subseteq \beta'$, alors la cellule β' est raffinée (et des cellules supplémentaires sont donc ajoutées au maillage) jusqu'à ce qu'une cellule fille soit égale à β .

D'autre part, pour s'assurer que le maillage prédit comporte suffisamment de cellules pour capturer l'apparition de singularités et garantir la convergence de la méthode, chaque cellule β (de niveau $< L$) est raffinée d'un niveau. Ce raffinement systématique renforce le côté pessimiste de la prédiction. La deuxième phase d'adaptation du maillage a pour rôle de corriger ce côté pessimiste. L'algorithme 1 résume les différentes opérations effectuées dans la première phase.

3.3.2 Correction du maillage (ou compression)

La deuxième phase est la conséquence directe de la prédiction pessimiste présentée précédemment. En effet, le raffinement systématique d'un niveau des cellules du maillage prédit induit une augmentation rapide du nombre de cellules du maillage adaptatif. Il paraît évident qu'au bout de quelques étapes de temps, le maillage va tendre vers un maillage uniforme. La phase de correction du maillage permet d'éviter cela. Cette phase élimine les cellules qui ont été raffinées dans des zones où peu de points de discrétisation sont requis, et donc où la prédiction est trop pessimiste. Les cellules sont déraffinées si l'approximation de la fonction par leur mère offre une précision suffisante, autrement dit si les valeurs de ces cellules peuvent être calculées depuis leur mère avec une erreur inférieure

Algorithme 1 : Construction du maillage prédit $\tilde{\mathcal{M}}^{n+1}$ à partir de $\mathcal{M}^n$.

```
Entrées : $\mathcal{M}^n$
Sorties : $\tilde{\mathcal{M}}^{n+1}$ initialement vide
début
    pour chaque cellule $\alpha_{l,i,j} \in \mathcal{M}^n$ faire
        calculer son centre $c$
        calculer le point $\mathcal{A}^+(c)$
        ajouter à $\tilde{\mathcal{M}}^{n+1}$ l'unique cellule $\beta$ de niveau $l$ qui contient $\mathcal{A}^+(c)$
        si $l < L \wedge \beta \in \tilde{\mathcal{M}}^{n+1}$ alors
            raffiner $\beta$
        fin
    fin
fin
```

à un seuil donné. En pratique, ce déraffinement a lieu dans les zones où la solution est suffisamment stable pour que l'on puisse conserver une précision identique d'une étape de temps à la suivante. Ce déraffinement peut aussi avoir lieu dans les zones qui se stabilisent et où un déraffinement de plusieurs niveaux peut être effectué. Comme cette phase permet de réduire le nombre de cellules présentes dans le maillage, elle est aussi appelée phase de *compression* du maillage.

Le principe de la compression est de supprimer toutes les cellules dont les valeurs aux noeuds peuvent être retrouvées par interpolation avec une erreur d'approximation inférieure à un certain seuil ϵ. Il faut donc parcourir l'ensemble du maillage $\tilde{\mathcal{M}}^{n+1}$ à la recherche de groupes de cellules soeurs. Une fois un groupe sélectionné, on calcule la valeur obtenue par interpolation en utilisant les valeurs aux noeuds de la cellule mère en chaque point correspondant à un nœud de cellule fille ajouté lors du raffinement. Puis on calcule la norme de la différence entre la valeur interpolée et la valeur stockée aux nœuds des cellules filles. On appelle cette opération le *test de compression*. Si toutes les différences sont inférieures au seuil fixé, alors on peut remplacer les cellules soeurs par leur mère. Les cellules restantes après la phase de compression forment le maillage $\mathcal{M}^{n+1}$ au temps t_{n+1}. L'algorithme 2 résume les opérations effectuées lors de cette phase de compression.

3.4 Schéma de résolution

Notre schéma de résolution commence par une phase d'initialisation durant laquelle le maillage initial est construit. Ce maillage est obtenu à partir de la fonction initiale f_0 donnée analytiquement. Il est construit héirarchiquement en partant d'une grille uniforme de niveau L dont chaque noeud est associé à la valeur de f_0 en le point de l'espace des phases correspondant à ce noeud. Le test de compression est alors effectué sur chaque groupe de cellules soeurs. Chaque fois que le test échoue le groupe de cellules est ajouté au maillage initial.

Nous allons maintenant donner le schéma de résolution en régime permanent (hors initialisation, finalisation et phases de diagnostic), qui consiste à trouver la solution approchée f^{n+1} au temps t_{n+1} à partir de la solution approchée f^n au temps t_n. Deux schémas sont présentés, en fonction de la manière dont le calcul des caractéristiques est

Algorithme 2 : Correction du maillage $\tilde{\mathcal{M}}^{n+1}$, prédit de façon pessimiste.

```
Entrées : $\tilde{\mathcal{M}}^{n+1}$, $\tilde{\mathcal{F}}^{n+1}$
Sorties : $\mathcal{M}^{n+1}$, $\mathcal{F}^{n+1}$
tant que il y a des cellules soeurs $(\alpha_0, \alpha_1, \alpha_2, \alpha_3) \in \tilde{\mathcal{M}}^{n+1}$ non traitées faire
    soit $\alpha$, la cellule mère de $(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$
    soit $\Pi_\alpha^{\tilde{\mathcal{F}}^{n+1}}$, le polynôme d'interpolation sur $\alpha$
    pour chaque noeud $a \in (\alpha_0 \cup \alpha_1 \cup \alpha_2 \cup \alpha_3)$ faire
        calculer $\varepsilon_a = \tilde{\mathcal{F}}^{n+1}(a) - \Pi_\alpha^{\tilde{\mathcal{F}}^{n+1}}(a)$
    fin
    si $\forall a$, $|\epsilon_a| < \epsilon$ alors
        remplacer $(\alpha_0, \alpha_1, \alpha_2, \alpha_3)$ par $\alpha$
    fin
fin
```

effectué : dans un premier temps nous considérons l'utilisation d'un opérateur d'advection $\mathcal{A}$ posé directement sur tout l'espace des phases, et dans un deuxième temps nous considérons l'utilisation de la technique du splitting. Ce deuxième schéma basé sur le splitting de l'équation introduit dans la section 1.3.3, utilise deux opérateurs d'advection $\mathcal{A}_{\mathbf{x}}$ et $\mathcal{A}_{\mathbf{v}}$.

3.4.1 Schéma sans splitting

Le schéma sans splitting se décompose en quatre étapes successives pour mettre à jour la solution au temps t_{n+1} à partir de la solution au temps t_n.

1. Calcul de E^n : la densité de charge ρ^n au temps t_n est calculée pour le couple maillage–valeur $(\mathcal{M}^n, \mathcal{F}^n)$. Pour chaque cellule $\alpha \in \mathcal{M}^n$, la fonction de distribution est reconstruite en tout point de la grille fine (de niveau L) à partir des valeurs aux nœuds de la cellule. La contribution de α à ρ^n est alors calculée en intégrant en vitesse les valeurs de la fonction. Le champ électrique E^n est ensuite déduit de ρ^n à l'aide d'un solveur de Poisson.

2. Prédiction de $\tilde{\mathcal{M}}^{n+1}$: un nouveau maillage adapté pour le temps t_{n+1} est prédit par la méthode introduite dans la section 3.3.1. Les valeurs du champ E^n sont utilisées lors du calcul des caractéristiques par l'opérateur d'advection en avant $\mathcal{A}^+$. La figure 3.8 montre les étapes effectuées pour la prédiction d'une cellule dans le nouveau maillage : la cellule (A) est ajoutée dans le maillage prédit ainsi que les cellules nécessaires pour garder le maillage dyadique (B). Puis la cellule ajoutée est raffinée (C).

3. Évaluation de $\tilde{\mathcal{F}}^{n+1}$: la fonction de distribution au temps t_{n+1} est approchée en chaque point de l'espace des phases correspondant aux nœuds du maillage prédit $\tilde{\mathcal{M}}^{n+1}$. Le calcul de ces valeurs se fait selon le schéma semi-Lagrangien en arrière vu dans la section 1.3.4. La figure 3.9 montre l'étape d'évaluation de la solution par la méthode semi-Lagrangienne : le calcul de l'origine des caractéristiques au pas de temps précédent est effectué pour chaque point de l'espace des phases correspondant à un nœud de cellule du

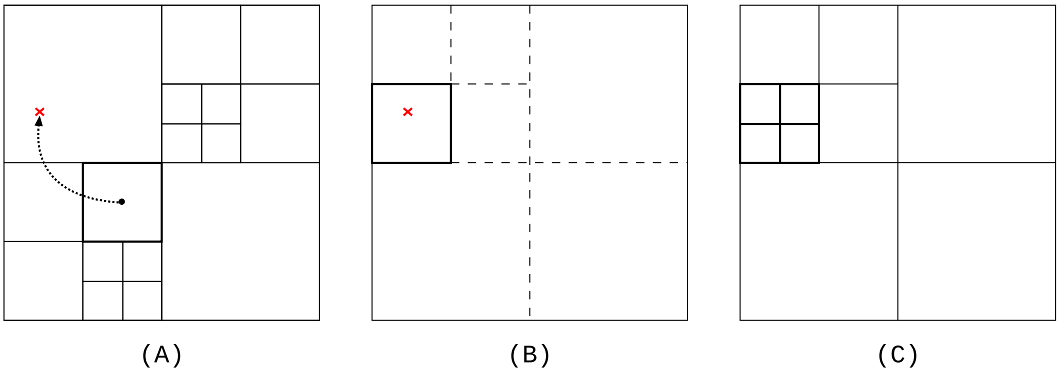

FIG. 3.8 – Étapes pour la construction du maillage prédit à partir d'un maillage initial.

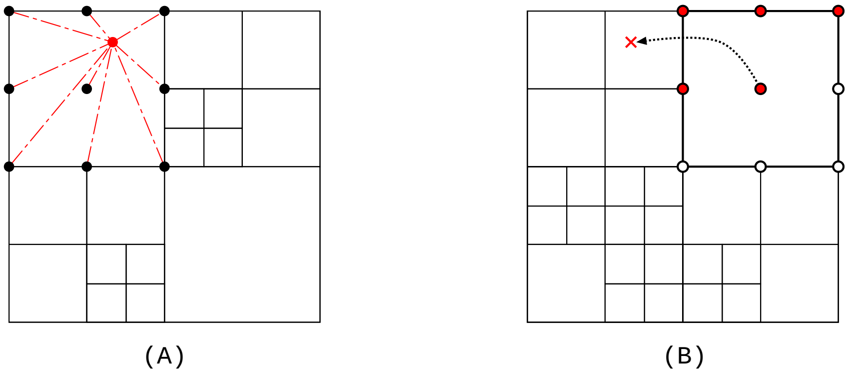

FIG. 3.9 – Calcul des valeurs de la fonction de distribution aux noeuds du maillage prédit par la méthode arrière du schéma semi-Lagrangien.

maillage prédit (B). Cette origine est alors considérée dans le maillage au pas de temps précédent (A), où l'unique cellule du maillage qui la contient est déterminée. Les valeurs aux nœuds de cette cellule sont utilisées pour calculer l'interpolation de la valeur en ce point.

4. Compression de $\tilde{\mathcal{M}}^{n+1}, \tilde{\mathcal{F}}^{n+1}$: la méthode de compression vue au paragraphe 3.3.2 est appliquée sur le couple maillage–valeur prédit de manière pessimiste. Cette compression permet d'obtenir la représentation adaptative $\mathcal{M}^{n+1}, \mathcal{F}^{n+1}$ de la fonction de distribution au temps t_{n+1}. La figure 3.10 montre l'étape de compression du maillage : un groupe de cellule soeurs du maillage prédit est sélectionné (A) et, si le test de compression réussit, ces cellules soeurs sont remplacées par la cellule mère (B).

On appelle *advection*, l'enchaînement des étapes de prédiction, d'évaluation et de compression décrites ci-dessus. La figure 3.11 résume l'enchaînement des différentes étapes de la méthode numérique adaptative depuis l'initialisation jusqu'à la boucle en temps, dans le cas où on n'utilise pas le splitting de l'équation.

3.4.2 Schéma avec splitting

Dans les cas où le splitting en temps de l'équation de Vlasov peut être utilisé (voir section 1.3.4), le passage d'un pas de temps au pas de temps suivant se fait par une

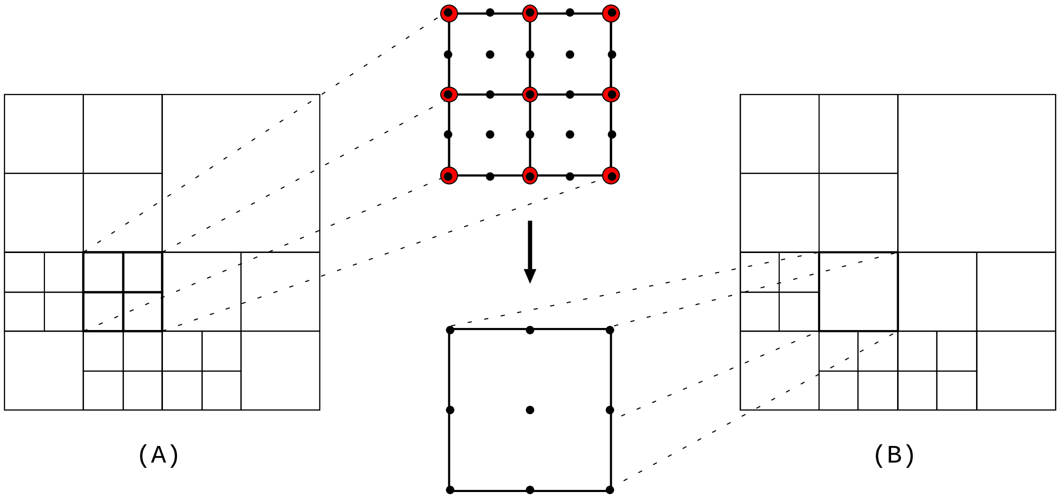

FIG. 3.10 – Compression du maillage prédit par groupes de cellules soeurs.

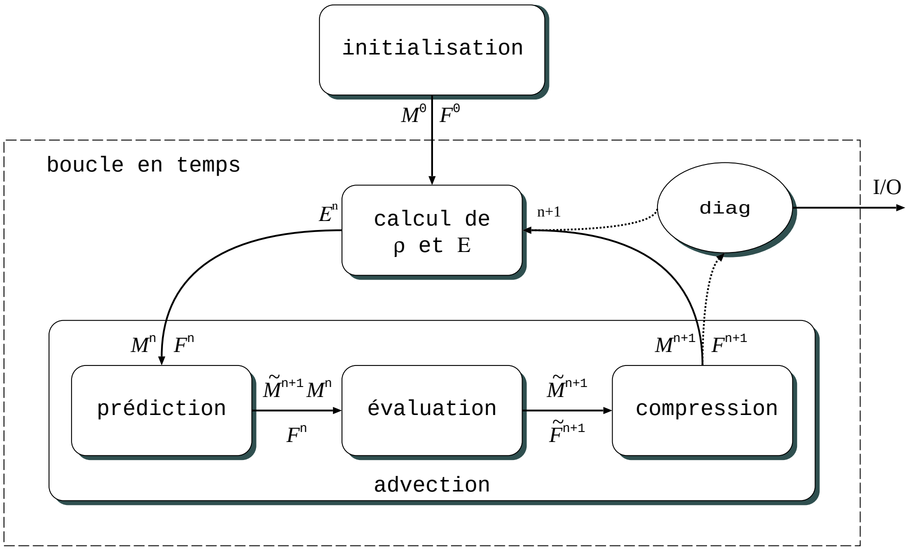

FIG. 3.11 – Schéma complet en temps de la méthode numérique adaptative YODA sans splitting.

succession d'advections. Ces advections diffèrent alors légèrement de celle introduite dans le paragraphe précédent : chaque phase d'advection utilisera un opérateur d'advection qui dépend de la dimension considérée lors de la phase de splitting. Posons $\mathcal{A}_x$ l'opérateur d'advection en dimension x, et $\mathcal{A}_v$ l'opérateur d'advection en dimension v. On appelle advection en x, une phase d'advection qui utilisent l'opérateur $\mathcal{A}_x$ sur un pas de temps complet. De même, on appelle *demi-advection en* x, une phase d'advection utilisant $\mathcal{A}_x$ sur un demi-pas de temps. Les advections intermédiaires calculent un couple maillage–valeur intermédiaire qui ne correspond pas à une solution physique. La solution n'a donc de sens qu'une fois l'ensemble des advections effectuées. Les phases d'advections pour passer de l'approximation de f^n par le couple $(\mathcal{M}^n, \mathcal{F}^n)$ à l'approximation f^{n+1} par le couple $(\mathcal{M}^{n+1}, \mathcal{F}^{n+1})$, s'enchaînent de la manière suivante :

1. Demi-advection en v : on possède la représentation $(\mathcal{M}^n, \mathcal{F}^n)$ depuis laquelle le champ électrique E^n a été déduit. On peut alors effectuer une advection avec $\mathcal{A}_v$ sur un demi-pas de temps pour obtenir la représentation intermédiaire $(\mathcal{M}^{n+1^*}, \mathcal{F}^{n+1^*})$.

2. Advection en x : à partir de la représentation intermédiaire en t^*_{n+1}, on effectue une advection avec l'opérateur $\mathcal{A}_x$ sur un pas de temps entier, et on obtient la représentation intermédiaire $(\mathcal{M}^{n+1^{**}}, \mathcal{F}^{n+1^{**}})$.

3. Mise à jour du champ E : on calcule alors le champ électrique E^{n+1} à partir de la représentation intermédiaire à t^{**}_{n+1} (le paragraphe 1.3.4 justifie le calcul de ρ sur une telle représentation intermédiaire).

4. Demi-advection en v : une fois les valeurs de champ électrique mises à jour pour le temps t_{n+1}, on peut effectuer une deuxième advection avec $\mathcal{A}_v$ sur un demi-pas de temps à partir de la représentation en t^{**}_{n+1}. L'approximation obtenue est alors $(\mathcal{M}^{n+1}, \mathcal{F}^{n+1})$, la représentation adaptative de f au temps t_{n+1}.

Les demi-pas de temps sont nécessaires pour retrouver la solution à un temps t_n donné. Mais pour tous les pas de temps où aucun diagnostic n'est prévu (en régime permanent), les demi-advections en v peuvent être regroupées en une seule advection en v. La figure 3.12 résume l'enchaînement des différentes advections pour le calcul d'un pas de temps. On notera bien que dans le cas où il n'y a pas de diagnostic, un cycle complet se fait en trois étapes.

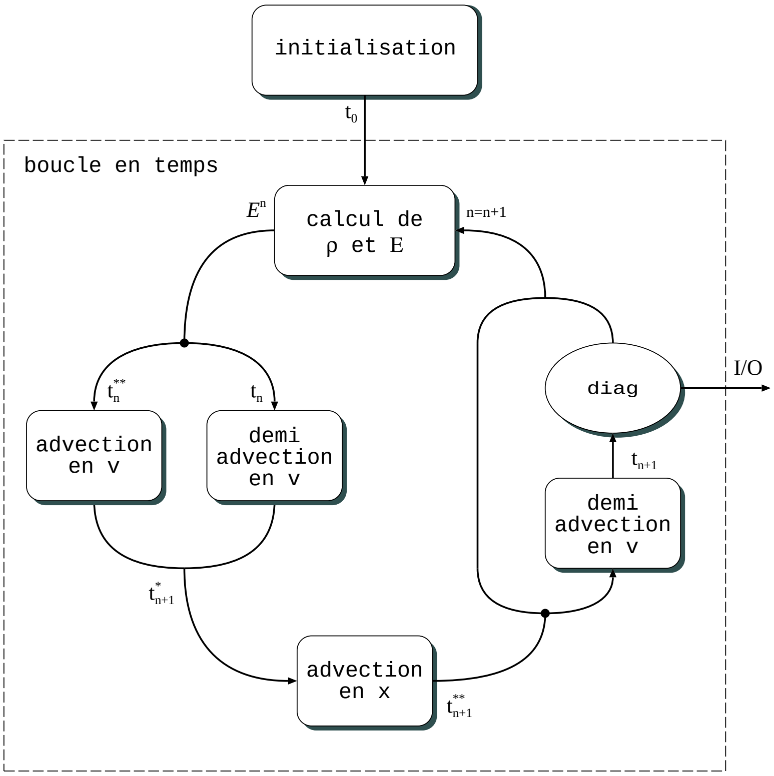

FIG. 3.12 – Schéma complet en temps de la méthode numérique adaptative YODA avec splitting et sans (choix de gauche) ou avec (choix de droite) l'étape de diagnostic.

Chapitre 4

Parallélisation de YODA en 2D

Nous avons vu dans le chapitre 3 une méthode numérique de résolution du système Vlasov–Poisson sur un maillage adaptatif de l'espace des phases. Cette méthode, basée sur le principe semi-Lagrangien, utilise un opérateur d'interpolation "local" au sens où cet opérateur permet de reconstruire la fonction de distribution en tout point d'une cellule, en utilisant uniquement les valeurs aux nœuds de cette cellule. Cette caractéristique rend la méthode particulièrement bien adaptée à une parallélisation pour machine à mémoire distribuée.

Dans ce chapitre, nous nous intéressons à la parallélisation de cette méthode, plus particulièrement de l'algorithme sans splitting donné en section 3.4.1. Cette parallélisation est effectuée dans le cas d'un espace des phases réduit à deux dimensions ($d = 2$). L'intérêt des méthodes adaptatives pour la résolution de l'équation de Vlasov est de permettre des simulations réalistes en dimensions supérieures ($d > 3$). C'est pourquoi les différentes techniques de parallélisation que nous considérons ici pour le cas $d = 2$ sont généralisables en dimension et extensibles aux cas $d = 4$ et $d = 6$.

Dans la première section, une analyse grossière de l'algorithme séquentiel est effectuée, conduisant à un algorithme data-parallèle qui exprime la décomposition parallèle du problème. Notre parallélisation repose sur une décomposition du domaine de calcul en *régions*. La deuxième section est consacrée à la distribution des données. La troisième section s'intéresse aux communications induites par cette distribution. La quatrième section décrit la structure de données mise en œuvre pour représenter le maillage qui est réparti entre les processeurs. La cinquième section concerne le développement d'un mécanisme d'équilibrage dynamique de charge pour gérer l'évolution en temps du maillage. Enfin la dernière section de ce chapitre est consacrée aux résultats obtenus avec le code parallèle.

4.1 Extraction du parallélisme

Nous montrons dans cette section que la méthode numérique YODA définit un algorithme data-parallèle en considérant le maillage comme une structure à accès parallèle aux cellules. Notre parallélisation pour machine à mémoire distribuée consiste à partager l'ensemble des cellules du maillage. Ce partage repose sur une décomposition du domaine de calcul.

4.1.1 Algorithme data-parallèle

Comme nous l'avons vu au chapitre précédent, la méthode numérique se décompose en quatre phases. Nous vérifions dans la suite que pour chaque phase, l'algorithme consiste à opérer un même traitement sur chaque cellule du maillage. Notre méthode peut donc être vue comme un algorithme data-parallèle puisque chaque phase consiste en le traitement séquentiel d'un conteneur (le maillage) de données à accès parallèle (les cellules).

Pour chaque phase, nous donnons l'algorithme et en rappelons les grandes lignes : l'objectif étant uniquement d'illustrer et de préciser notre point de vue "data-parallèle" de la méthode.

Phase 1. La première phase est le calcul du champ électrique E^n. Cette phase consiste à calculer la densité de charge puis à résoudre l'équation de Poisson. Pour chaque cellule de $\mathcal{M}^n$, on calcule sa contribution à ρ^n, puis on somme toutes ces contributions. Cette phase est décrite en détails par l'algorithme 3.

Algorithme 3 : Calcul de la densité de charge.

```
Entrées : $\mathcal{M}^n$,$\mathcal{F}^n$
Sorties : $\rho^n$
pour chaque cellule $\alpha \in \mathcal{M}^n$ faire
    soit $l_\alpha$ le niveau de $\alpha$ et $(x_\alpha, v_\alpha)$ ses coordonnées dans la grille de niveau $l_\alpha$
    récupérer les valeurs de l'ensemble des nœuds de $\alpha$ dans $\mathcal{F}^n$
    pour tous les $x \in \mathbb{N}$ tel que $2^{L-l_\alpha+1}x_\alpha \leq x < 2^{L-l_\alpha+1}(x_\alpha+1)$ faire
        intégrer en vitesse les valeurs de la cellule pour la position $x$
        additionner la valeur intégrée à $\rho^n(x)$
    fin
fin
```

Phase 2. La deuxième phase est la prédiction du maillage $\tilde{\mathcal{M}}^{n+1}$. Pour chaque cellule de $\mathcal{M}^n$, on transporte son centre et on insère dans $\tilde{\mathcal{M}}^{n+1}$ l'unique cellule de même niveau que α et qui contient le point issu du transport en avant du centre de α. L'algorithme 4 détaille cette phase.

Phase 3. La troisième phase est l'évaluation des valeurs $\tilde{\mathcal{F}}^{n+1}$. Pour chaque cellule β de $\tilde{\mathcal{M}}^{n+1}$, on calcule la valeur en chacun de ses noeuds. La valeur au noeud a est la valeur interpolée au point issu du transport en arrière de a en utilisant les valeurs aux noeuds de la cellule de $\tilde{\mathcal{M}}^n$ contenant ce point. L'algorithme 5 détaille cette phase.

Phase 4. Enfin la quatrième et dernière phase est la compression du maillage. Pour chaque groupe de cellules soeurs, on effectue le test de compression et s'il réussit, on remplace ce groupe de cellules par leur mère. On répète ce traitement tant qu'il existe un groupe de cellules non traité. L'algorithme 6 détaille cette phase.

Algorithme 4 : Prédiction du maillage.

```
Entrées : $\mathcal{M}^n$
Sorties : $\tilde{\mathcal{M}}^{n+1}$
pour chaque cellule $\alpha \in \mathcal{M}^n$ faire
    soit $l_\alpha$ le niveau de $\alpha$
    calculer le centre $c$ de $\alpha$
    calculer $\mathcal{A}^+(c)$
    déterminer la cellule $\beta$ de niveau $l_\alpha$ et telle que $\mathcal{A}^+(c) \in \beta$
    si $l_\alpha < L$ alors
        insérer les cellules filles de $\beta$ dans $\tilde{\mathcal{M}}^{n+1}$
    sinon
        insérer $\beta$ dans $\tilde{\mathcal{M}}^{n+1}$
    fin
fin
```

Algorithme 5 : Évaluation des nœuds du maillage.

```
Entrées : $\tilde{\mathcal{M}}^{n+1}$,$\mathcal{M}^n$,$\mathcal{F}^n$
Sorties : $\tilde{\mathcal{F}}^{n+1}$
pour chaque cellule $\alpha \in \tilde{\mathcal{M}}^{n+1}$ faire
    pour chaque nœud $a \in \alpha$ faire
        si $a$ n'a pas encore de valeur dans $\tilde{\mathcal{F}}^{n+1}$ alors
            calculer $\mathcal{A}^-(a)$
            déterminer la cellule $\beta \in \mathcal{M}^n$ telle que $\mathcal{A}^-(a) \in \beta$
            interpoler la valeur $v_a$ de $a$ avec les valeurs de $\beta$ dans $\mathcal{F}^n$
            ajouter $(a, v_a)$ à $\tilde{\mathcal{F}}^{n+1}$
        fin
    fin
fin
```

Algorithme 6 : Compression du maillage.

```
Entrées : $\tilde{\mathcal{M}}^{n+1}$,$\tilde{\mathcal{F}}^{n+1}$
Sorties : $\mathcal{M}^{n+1}$,$\mathcal{F}^{n+1}$
pour chaque cellule $\alpha \in \tilde{\mathcal{M}}^{n+1}$ faire
    si $\alpha$ est non marquée alors
        $cpt := 1$
        marquer $\alpha$
        pour chaque cellule $\alpha_k, k \in \{1..3\}$ soeur de $\alpha$ faire
            si $\alpha_k \in \tilde{\mathcal{M}}^{n+1}$ et $\alpha_k$ est non marquée alors
                $cpt++$
                marquer $\alpha_k$
            fin
        fin
    fin
    si $cpt = 4$ alors
        soit $\alpha'$ la cellule mère de $\alpha$
        effectuer le test de compression sur $\alpha'$
        si le test de compression réussit alors
            retirer $\alpha$ et $\alpha_{k\in\{1..3\}}$ de $\tilde{\mathcal{M}}^{n+1}$
            ajouter $\alpha'$ à $\mathcal{M}^{n+1}$
        fin
    fin
fin
$\mathcal{M}^{n+1} := \tilde{\mathcal{M}}^{n+1}$
$\mathcal{F}^{n+1} := \tilde{\mathcal{F}}^{n+1}$
```

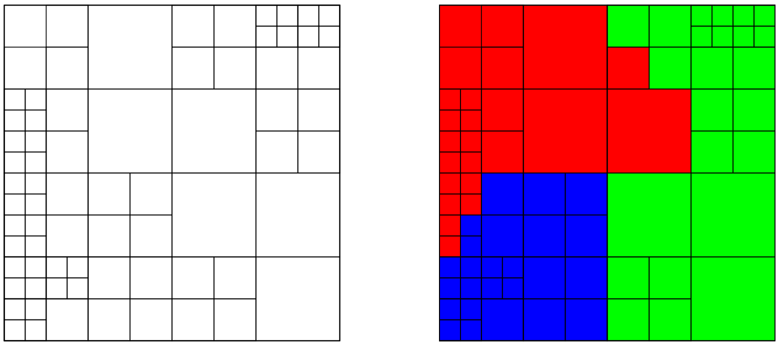

Fig. 4.1 – Distribution des cellules du maillage sur les processeurs.

4.1.2 Décomposition en tâches parallèles

Nous pouvons donc partitionner notre application en définissant une tâche parallèle comme le traitement d'une cellule (pour chaque phase de l'algorithme). Étant donné que le coût d'une tâche est faible par rapport au nombre de tâches à traiter, notre algorithme data-parallèle exhibe un parallélisme à grain fin. Pour obtenir un parallélisme à gros grain mieux adapté aux architectures parallèles à mémoire distribuée, on choisit de regrouper ces tâches : chaque tâche parallèle sera le traitement séquentiel d'un ensemble de cellules.

Notre décomposition est définie de la manière suivante : le domaine de calcul (l'espace des phases) est découpé virtuellement en *régions*. On appelle région une zone du domaine correspondant à une union de cellules d'un maillage dyadique. L'ensemble des régions forme une partition du domaine et une tâche parallèle correspond au traitement séquentiel des cellules à l'intérieur d'une région de l'espace des phases.

4.2 Distribution des données

Sur une architecture à mémoire distribuée, chaque tâche parallèle, correspondant à une région de l'espace des phases, est allouée à un processeur : les informations relatives aux cellules et aux noeuds à l'intérieur de la région sont stockées dans la mémoire locale du processeur, qui traite les informations de sa région selon la règle "the owner computes". La figure 4.1 montre un exemple de distribution du maillage pour 3 processeurs.

Comme nous l'avons vu dans le chapitre 3, certains nœuds peuvent être partagés par plusieurs cellules voisines. Dans le cas où ces cellules voisines appartiennent à des régions différentes, la valeur de chacun de ces noeuds partagés est dupliquée et stockée dans la mémoire locale de plusieurs processeurs. Ceci introduit un surcoût en terme de stockage et de calcul. Ce surcoût est faible dans le cas 2D car les nœuds concernés sont peu nombreux.

4.3 Gestion des communications

La distribution des données détermine les communications à réaliser. Dans un premier temps, nous expliciterons les communications engendrées par la distribution. Puis nous discuterons des techniques utilisées pour implanter et réduire le coût de ces communications.

4.3.1 Schéma de communications

Nous décrivons les communications engendrées par la distribution des données. Les communications requises sont données phase par phase :

Phase 1 : calcul du champ. La résolution de l'équation de Poisson (qui donne E à partir de ρ) est posée dans l'espace des positions (à 1 dimension dans le cas 2D). Son coût est donc négligeable par rapport au calcul de ρ, qui lui, est posé dans l'espace des phases. Par conséquent, on choisit de réaliser le calcul de E séquentiellement : on effectue le même calcul sur chaque processeur. Cela implique que chaque processeur doit disposer de la densité de charge ρ. On effectue la somme des contributions et la redistribution du résultat à l'aide d'une seule communication globale de type *all-to-all*. Cette communication implique une barrière de synchronisation.

Phase 2 : prédiction. La prédiction du maillage nécessite un traitement plus complexe puisqu'elle utilise le transport en avant. Prenons l'exemple d'une cellule $\alpha \in \mathcal{M}$ appartenant à la région du processeur p, et appelons $\beta \in \tilde{\mathcal{M}}$ l'unique cellule de même niveau que α et qui contient le point issu du transport en avant du centre de α. La cellule β est la cellule à ajouter au maillage prédit. Il y a deux cas à considérer selon la région à laquelle appartient β.

1. *Insertion locale* : si β se trouve dans la région du processeur p, alors aucune communication n'est nécessaire.
2. *Insertion distante* : si β se trouve dans la région d'un processeur, disons p', différent de p, alors le processeur p ne peut pas insérer β dans le maillage car il ne détient pas la partie du maillage où β doit être insérée. Par conséquent, le processeur p doit transmettre la cellule β au processeur p', et c'est p' qui va insérer cette cellule au maillage.

Phase 3 : évaluation. Considérons le calcul de la valeur d'un nœud a d'une cellule $\alpha \in \tilde{\mathcal{M}}$ appartenant à la région du processeur p. Comme pour la phase de prédiction, il y a deux cas à considérer selon que le point $\mathcal{A}^{-}(a)$, issu du transport en arrière de a, appartient à la région du processeur p ou à une autre région.

1. *Interpolation locale* : si le point $\mathcal{A}^{-}(a)$ se trouve dans la région du processeur p, alors p détient l'unique cellule du maillage qui contient ce point, ainsi que les valeurs aux nœuds de cette cellule. L'interpolation de la valeur de a peut donc être réalisée localement et ne nécessite aucune communication.
2. *Interpolation distante* : si le point $\mathcal{A}^{-}(a)$ se trouve dans la région d'un autre processeur, disons p', alors p' est le seul qui détient la cellule du maillage contenant ce point, et c'est également lui qui possède les valeurs aux nœuds de cette cellule. Dans ce cas, nous réalisons le protocole suivant : le processeur p envoie au processeur p' les coordonnées de $\mathcal{A}^{-}(a)$. À la réception du message, le processeur p' recherche, parmi les cellules de sa région, l'unique cellule $\beta \in \mathcal{M}$ qui contient ce point. A ce stade, deux approches sont possibles pour réaliser l'interpolation de la valeur en ce point.
    (a) *Rapatriement des données* : le processeur p' envoie à p l'identifiant de β et l'ensemble des valeurs aux nœuds de β et c'est le processeur p qui va calculer l'interpolation.

(b) *Calcul distant* : le processeur p' calcule lui-même la valeur au point qu'il vient de recevoir. Puis il renvoie au processeur p la valeur interpolée en ce point.

Dans notre cas, nous choisissons d'effectuer l'approche du calcul distant. Cette approche obéit à la règle *the owner computes* : c'est le processeur qui possède les données qui effectue le calcul. Ce choix a l'avantage de réduire le nombre de données transférées. En effet, le processeur distant renvoie une seule valeur, et non pas 3^d valeurs plus la clé d'une cellule comme c'est le cas pour le rapatriement des données. L'autre approche peut cependant être envisagée si les valeurs transmises sont stockées pour être réutilisées par la suite pour de futures interpolations. Mais le stockage et le traitement de l'ensemble des valeurs sont alors d'autant plus complexes.

Phase 4 : compression. On considère un groupe de cellules soeurs du maillage pour lesquelles le test de compression est effectué. Ce test nécessite l'accès à l'ensemble des valeurs aux nœuds de ces cellules. Il y a deux cas à considérer, selon que ces valeurs se trouvent toutes dans la mémoire locale du processeur ou pas. Lorsque toutes les cellules soeurs appartiennent à une même région, alors le processeur responsable de cette région peut effectuer le test de compression. Comme il possède localement l'ensemble des valeurs de ces cellules, aucune communication n'est requise. Si une des cellules soeurs est située dans une autre région, alors le processeur ne possède pas l'ensemble des valeurs requises pour effectuer le test de compression. On a alors le choix de rapatrier les valeurs distantes ou de restreindre la compression aux groupes composées de cellules de la même région.

Le rapatriement des valeurs permet d'assurer que la phase de compression est identique qu'elle soit effectuée en parallèle ou en séquentiel. Par contre, ce rapatriement engendre un surcoût de communication et de nombreux problèmes de gestion des données dans le cas où le test réussi. Ces traitement spécifiques augmentent la complexité de la phase de compression en fonction du nombre de processeurs utilisés, ce qui peut grandement affecter l'efficacité du programme. Pour éviter cela, nous décidons d'effectuer la phase de compression uniquement sur des groupes où toutes les cellules soeurs appartiennent à la même région. De cette façon, toutes les valeurs nécessaires au test de compression sont présentes localement. Par conséquent, la phase de compression ne nécessite plus aucune communication. Par contre, le résultat de la compression peut être différent en séquentiel et en parallèle. Cette approximation de la méthode peut notamment engendrer une baisse de taux de compression provoquant une légère augmentation de la charge globale de calcul. Mais cette approximation ne fait pas tendre le maillage vers un maillage uniforme puisque la compression peut s'effectuer sur plusieurs niveaux successivement. D'autre part, cette approximation ne remet pas en cause la convergence de la méthode.

4.3.2 Implantation des communications

Les communications sont réalisées à l'aide des fonctions de passage de message de la norme MPI [84]. Les phases de calcul de champ et de compression ne présentent pas de difficultés particulières du point de vue des communications. En effet, la phase de compression ne nécessite aucune communication alors que le calcul du ρ nécessite une seule communication collective réalisée par un appel à la fonction `MPI_Allreduce`.

Par contre, les phases de prédiction et d'évaluation exhibent des communications qui exigent une gestion particulière. En effet, ces communications sont déterminées par l'opérateur d'advection qui dépend du champ E qui, lui-même, dépend de la solution. Par conséquent, ces communications ne sont pas prévisibles : on ne connaît à l'avance ni leur nombre, ni l'identité des processeurs émetteurs et récepteurs. Pour implanter ces communications, nous définissons un schéma spécifique qui autorise les communications *à la volée* et le recouvrement de ces communications par des calculs.

Le recouvrement des communications par des calculs [117, 103] permet de réduire l'impact des communications sur le temps d'exécution d'un programme parallèle. Cette technique, bien connue en calcul haute performance, utilise des fonctions de communication *non-bloquantes* afin d'effectuer des calculs utiles pendant que le message transite sur le réseau. Elle permet notamment d'optimiser les algorithmes parallèles composés de phases de calcul sans communication et de phases de communication sans calcul [113].

Communications *à la volée*

Notre schéma de communications *à la volée* est utilisé lors de la phase de prédiction pour communiquer les cellules en cas d'ajout distant, et lors de la phase d'évaluation pour communiquer les nœuds et leurs valeurs en cas d'interpolation distante. Il utilise des fonctions de communication point-à-point non-bloquantes pour permettre le recouvrement par les calculs. Ce schéma utilise également un message spécifique appelé `fin-des-envois`, qui permet de spécifier l'arrêt des communications d'un processeur à un autre. Nous décrivons dans la suite ce schéma en séparant les parties émission et réception des messages.

1. *Émission* : durant la phase de prédiction, réaliser une insertion distante consiste simplement à initier un envoi de cellule (sans attendre la terminaison de l'envoi). Lorsque toutes les insertions (locales ou distantes) ont été réalisées, le processeur courant initie l'envoi du message `fin-des-envois` à tous les autres processeurs pour leur signaler qu'ils ne recevront plus de message provenant du processeur courant. Le schéma est similaire pour la phase d'évaluation : réaliser une interpolation distante consiste à initier un envoi de point et une réception de valeur (on initie la réception d'un message contenant la valeur interpolée au point). Lorsque toutes les interpolations (locales ou distantes) ont été réalisées, le processeur courant envoie le message `fin-des-envois` à tous les autres processeurs.
2. *Réception* : au début de la phase de prédiction, chaque processeur initie $P - 1$ réceptions de cellule – où P est le nombre de processeurs – en prévision de l'arrivée d'un message provenant d'un autre processeur. À chaque message réceptionné, une nouvelle réception de message est initiée pour vers le processeur émetteur. Cette opération est réitérée jusqu'à ce que le message reçu soit `fin-des-envois`. Lorsqu'un processeur a reçu le message `fin-des-envois` de la part de chacun des autres processeurs, alors sa phase de réception des messages est terminée (pour cette étape de prédiction). Le schéma est similaire pour la phase d'évaluation, excepté que pour chaque réception d'un message contenant un point, un envoi de la valeur interpolée en ce point est initié.

L'algorithme 7 reprend l'algorithme 4 en explicitant les communications *à la volée* telles qu'elles sont décrites ci-dessus.

Mise en œuvre du schéma de communications

Notre schéma de communications *à la volée* est réalisé en utilisant des appels à des fonctions MPI de communication non-bloquante. Parmi les modes de communication mis à

Algorithme 7 : Prédiction parallèle du maillage.

```
Entrées : $\mathcal{M}^n$
Sorties : $\tilde{\mathcal{M}}^{n+1}$
pour chacun des autres processeur faire
    initier la réception d'un message provenant de ce processeur
fin
fini := faux
tant que $\neg$ fini faire
    /* émission */
    si il reste une cellule $\alpha \in \mathcal{M}^n$ à traiter alors
        soit $l_\alpha$ le niveau de $\alpha$
        calculer le centre $c$ de $\alpha$
        calculer $\mathcal{A}^+(c)$
        déterminer la cellule $\beta$ de niveau $l_\alpha$ et telle que $\mathcal{A}^+(c) \in \beta$
        soit $p$ le processeur qui traite la région contenant $\beta$
        si $p$ est le processeur courant alors
            si $l_\alpha < L$ alors
                insérer les cellules filles de $\beta$ dans $\tilde{\mathcal{M}}^{n+1}$
            sinon
                insérer $\beta$ dans $\tilde{\mathcal{M}}^{n+1}$
            fin
        sinon
            initier l'envoi de $\beta$ au processeur $p$
        fin
    sinon
        initier l'envoi du message fin-des-envois à tous les autres processeurs
    fin
    /* réception */
    si la réception d'un message s'est terminée alors
        si ce message n'est pas fin-des-envois alors
            soit $\beta$ la cellule contenue dans le message et $l_\beta$ son niveau
            soit $p$ le processeur émetteur
            si $l_\beta < L$ alors
                insérer les cellules filles de $\beta$ dans $\tilde{\mathcal{M}}^{n+1}$
            sinon
                insérer $\beta$ dans $\tilde{\mathcal{M}}^{n+1}$
            fin
            initier une nouvelle réception d'un message en provenance de $p$
        fin
    fin
    mettre $fini$ à vrai si il ne reste plus de cellules à traiter ni de messages à
    réceptionner
fin
```

disposition par la norme MPI, le mode bufferisé, avec l'emploi de la fonction `MPI_Ibsend`, est bien adapté pour réaliser notre schéma de communication et maximiser le recouvrement. En effet, avec ce mode de communication, la valeur des données envoyées peut être modifiée dès le retour de cette fonction. Ce n'est pas le cas avec le mode synchrone et la fonction `MPI_Issend` pour lequel la valeur des données ne peut être modifiée qu'après la terminaison de la communication.

Le code 4.1 montre comment l'algorithme 7 peut être implanté en utilisant le mode bufferisé. Ce code est précédé d'une phase d'initialisation qui consiste à initier une réception de message pour chacun des autres processeurs. De plus, le mode bufferisé nécessite l'allocation, pour chaque processus MPI, d'un tampon d'envoi dans lequel les messages sont copiés à chaque appel à la fonction `MPI_Ibsend`. Un tampon de taille `nbcell*(sizeof(cell) +MPI_BSEND_OVERHEAD)` est attaché au processus MPI avec `MPI_Buffer_attach` appelée lors de cette phase d'initialisation.

Dans ce code, le maillage $\mathcal{M}^n$ (resp. $\tilde{\mathcal{M}}^{n+1}$) est représenté par la variable `Mold` (resp. `Mnew`). La fonction `next` renvoie l'adresse de la prochaine cellule non traitée, et `NULL` lorsque toutes les cellules sont traitées. La fonction `insert` réalise l'insertion d'une cellule ou de ses filles dans le maillage. Une cellule est représentée par une structure `cell` associée au type `MPI_CELL` : cette structure contient 3 entiers pour décrire les coordonnées x, v et le niveau de la cellule. Les tableaux `sreq` et `rreq` sont de taille P et servent à stocker les requêtes de communication associées respectivement aux envois et réceptions pour chacun des autres processeurs.

À chaque envoi de cellule à destination d'un processeur p, l'appel à `MPI_Ibsend` crée une nouvelle requête. Cette requête est stockée dans `sreq[p]`. Pour réutiliser cet emplacement dans le tableau `sreq`, il faut que la communication associée à la requête stockée à cet emplacement soit terminée. Nous assurons cette terminaison par un appel à `MPI_Wait`. Cette fonction retourne dès que les valeurs des données ont été copiées dans le tampon attaché au processus MPI. Pour simplifier le schéma de communication et ne transmettre que des messages contenant des données du même type, le message de fin des envois est lui aussi du type `MPI_CELL`.

La terminaison des réceptions de message est testée par `MPI_Testany`. Si une réception s'est terminée, `flag` vaut vrai et `idx` contient l'indice du tableau `rreq` où est stockée la requête associée à cette réception. Si aucune réception ne s'est terminée, `flag` vaut faux et `idx` contient la constante `MPI_UNDEFINED`. Enfin, si il n'y a aucune réception en cours, alors `flag` vaut vrai et `idx` vaut `MPI_UNDEFINED`.

Si un message a été reçu en provenance du processeur p, on vérifie que la cellule transmise est différente de `fin-des-envois`. Si c'est le cas, cette cellule est insérée dans le maillage, et une nouvelle réception d'un message provenant de p est initiée. On note que si un message `fin-des-envois` est reçu en provenance du processeur p, aucune réception ne sera plus initiée vers ce même processeur pendant cette phase de prédiction. Ainsi, lorsqu'un processeur a reçu le message `fin-des-envois` de la part de tous les autres, il n'y a plus de requêtes de réception valides dans le tableau `rreq`. Le prochain test de terminaison de message provoquera alors la mise à vrai de `endrecv`. La phase de prédiction termine lorsque les booléens `endrecv` et `endcell` sont à vrai.

Le principe de l'implantation du schéma de communications est quasiment identique pour la phase d'évaluation. Il y a deux différences avec l'implantation effectuée pour la phase de prédiction. La première est que les éléments transmis ne sont plus des identifiants

Extrait de code 4.1 – Phase parallèle de prédiction.

```
for (p=0; p<P; p++)
{
 if (p!=myproc)
  MPI_Irecv(rbuf[p], 1, MPI_CELL, p, CTAG, MPICOM, &rreq[p]);
}
fini = false;
while (!fini)
{
 if (!endcell)
 {
  if ((alpha = Mold->next())!=NULL)
  {
   // soit beta la cellule issue du transport en avant de alpha
   // soit p le processeur associe a la region contenant beta
   if (p == myproc)
    Mnew->insert(beta);
   else
   {
    MPI_Wait(&sreq[p],MPI_STATUS_IGNORE);
    MPI_Ibsend(beta,1,MPI_CELL,p,CTAG,MPICOM,&sreq[p]);
   }
  }
  else
  {
   endcell = true;
   for (p=0; p<P; p++)
   {
    if (p!=myproc)
    {
     MPI_Wait(&sreq[p],MPI_STATUS_IGNORE);
     MPI_Ibsend(fin-des-envois,1,MPI_CELL,p,CTAG,MPICOM,&sreq[p]);
    }
   }
  }
 }
 if (!endrecv)
 {
  MPI_Testany(P, rreq, &idx, &flag, &mps);
  if (!(idx==MPI_UNDEFINED || is_equal(rbuf[idx],fin-des-envois)))
  {
   Mnew->insert(rbuf[idx]);
   MPI_Irecv(rbuf[idx], 1, MPI_CELL, idx, CTAG, MPICOM, &rreq[i]);
  }
  else
   endrecv = flag;
 }
 fini = endcell && endrecv;
}
```

de cellules mais des coordonnées de points dans l'espace des phases : un message est donc composé de deux flottants. La deuxième différence correspond au renvoi de la valeur interpolée au point reçu. Après chaque envoi de point, une réception de valeur est initiée par un appel à `MPI_Irecv`. Cette valeur correspond à la valeur interpolée en ce point. Après chaque réception d'un point en provenance d'un processeur p, et juste après l'interpolation de la valeur en ce point, l'envoi de cette valeur est initié par un appel à `MPI_Ibsend` à destination de p. Deux tableaux de taille P sont alloués pour stocker les requêtes associées à ces réceptions et émissions de valeurs.

Avec cette implantation du schéma de communications, le nombre de messages échangés peut être très important. Par conséquent, traiter séparément chaque message peut conduire à une importante baisse des performances due à l'accumulation de la latence inhérente à chaque communication point-à-point. Pour améliorer les performances de ces implantations, les données sont regroupées avant d'être envoyées, ce qui augmente la taille des messages et permet d'en réduire le nombre. Pour cela, les données à envoyer sont accumulées dans un tampon, et l'envoi du message est initié lorsque ce tampon est plein. Pour signifier la fin des envois, l'envoi d'un message est initié immédiatement sans que le tampon soit rempli. À la réception, on détecte la fin des envois en testant la taille du message réceptionné à l'aide de la fonction `MPI_Get_count`. La réception d'un message de taille inférieure à celle d'un tampon plein signifie la fin des envois provenant d'un processeur.

La phase d'évaluation requiert un tableau de pointeurs ayant le même nombre d'éléments que le tampon. Pour chaque ajout d'un point $\mathcal{A}^-(a)$ (issu du transport en arrière d'un nœud a) en i-ième position dans le tampon, on stocke l'adresse de la valeur de a à l'indice i de ce tableau. Lorsqu'une réception de valeurs est terminée, la i-ième valeur reçue est copiée à l'adresse pointée par le i-ième élément du tableau.

4.4 Structure de données

Dans cette section, nous cherchons une structure de données convenable pour représenter le maillage. Nous nous livrons d'abord à une analyse des accès au maillage. Cette analyse permet d'identifier les propriétés requises pour la structure de données. Enfin, nous proposons une structure à base de tables de hachage qui possède les bonnes propriétés.

4.4.1 Analyse des accès

Nous étudions ici les accès aux données effectués lors de la résolution de l'équation de Vlasov en utilisant notre algorithme séquentiel. Plus précisément, nous évaluons le nombre et le type de ces accès. Le type d'un accès est défini à partir d'une classification selon trois critères :

- l'accès à la donnée est effectué en écriture (E) ou en lecture (L),
- la donnée accédée correspond à une cellule (C) ou à un nœud (N),
- les données sont accédées séquentiellement (S) ou de manière aléatoire (A).

Les accès séquentiels correspondent à des accès où le choix des données ne dépend que de la donnée précédemment accédée. Pour des accès de ce type, une correspondance peut être établie entre le placement des données en mémoire et l'ordre dans lequel ces données sont accédées.

Pour les accès aléatoires, le choix de la donnée accédée dépend de paramètres qui le rendent non déterministe. De tels accès ont lieu chaque fois que les opérateurs d'avection

$\mathcal{A}^+$ ou $\mathcal{A}^-$ sont utilisés. On note respectivement $\|\mathcal{C}^n\|$ et $\|\mathcal{N}^n\|$ le nombre de cellules et de nœuds du maillage $\mathcal{M}^n$. De même on note $\|\tilde{\mathcal{C}}^{n+1}\|$ et $\|\tilde{\mathcal{N}}^{n+1}\|$ ces valeurs pour le maillage intermédiaire $\tilde{\mathcal{M}}^{n+1}$.

Phase 1 Les cellules du maillage $\mathcal{M}^n$ sont toutes parcourues de manière arbitraire, ce qui engendre $\|\mathcal{C}^n\|$ accès séquentiels. Pour chaque cellule ainsi accédée, les 3^d valeurs aux nœuds de la cellule sont lues dans un ordre quelconque, ce qui engendre $\|\mathcal{C}^n\| \times 3^d$ accès séquentiels.

$$\begin{aligned}&\textit{nombre d'accès au maillage pour la phase de calcul de } E\\ &= \Big(\|\mathcal{C}^n\|\Big)_{L,C,S} + \Big(\|\mathcal{C}^n\| \times 3^d\Big)_{L,N,S} \ \textit{accès}\end{aligned} \tag{4.1}$$

Phase 2 Les cellules du maillage $\mathcal{M}^n$ sont à nouveau parcourues de manière arbitraire, d'où $\|\mathcal{C}^n\|$ accès séquentiels. Pour chaque cellule parcourue, on ajoute 2^d cellules soeurs dans $\tilde{\mathcal{M}}^{n+1}$, ce qui engendre $\|\mathcal{C}^n\| \times 2^d$ accès aléatoires. Nous négligeons ici le cas où la cellule parcourue est de niveau L (pour lequel une seule cellule est ajoutée dans $\tilde{\mathcal{M}}^{n+1}$), ainsi que le cas nécessitant l'ajout de cellules supplémentaires (pour maintenir la consistance du maillage). En effet, ces cas sont difficilement quantifiables et ont un faible impact sur ce décompte.

$$\begin{aligned}&\textit{nombre d'accès aux maillages pour la phase de prédiction}\\ &= \Big(\|\mathcal{C}^n\|\Big)_{L,C,S} + \Big(\|\mathcal{C}^n\| \times 2^d\Big)_{E,C,A} \ \textit{accès}\end{aligned} \tag{4.2}$$

Phase 3 Les nœuds du maillage $\tilde{\mathcal{M}}^{n+1}$ sont parcourus par l'intermédiaire des cellules : on parcourt toutes les cellules dans n'importe quel ordre (soit $\|\tilde{\mathcal{C}}^{n+1}\|$ accès séquentiels), et pour chaque cellule, on parcourt chacun de ses nœuds (soit $\|\tilde{\mathcal{C}}^{n+1}\| \times 3^d$ accès séquentiels). La recherche parmi $L - l_0 + 1$ de l'unique cellule qui contient un point nécessite au plus $L - l_0$ tests (soit $\|\tilde{\mathcal{N}}^{n+1}\| \times (L - l_0)$ accès aléatoires). Une fois que la cellule est trouvée, les valeurs de ses 3^d nœuds sont lues pour réaliser l'interpolation, ce qui engendre $\|\tilde{\mathcal{N}}^{n+1}\| \times 3^d$ accès aléatoires. Enfin, la valeur interpolée est stockée pour chaque nœud de $\tilde{\mathcal{M}}^{n+1}$, ce qui engendre $\|\tilde{\mathcal{N}}^{n+1}\|$ accès séquentiels.

$$\begin{aligned}&\textit{nombre d'accès aux maillages pour la phase d'évaluation}\\ &= \Big(\|\tilde{\mathcal{C}}^{n+1}\|\Big)_{L,C,S} + \Big(\|\tilde{\mathcal{C}}^{n+1}\| \times 3^d\Big)_{L,N,S} + \Big(\|\tilde{\mathcal{N}}^{n+1}\| \times (L - l_0)\Big)_{L,C,A}\\ &\quad + \Big(\|\tilde{\mathcal{N}}^{n+1}\| \times 3^d\Big)_{L,N,A} + \Big(\|\tilde{\mathcal{N}}^{n+1}\|\Big)_{E,N,S} \ \textit{accès}\end{aligned} \tag{4.3}$$

Phase 4 (Compression). Les cellules de $\tilde{\mathcal{M}}^{n+1}$ sont parcourues de manière arbitraire pour rechercher les groupes de cellules soeurs (soit $\tilde{\mathcal{C}}^{n+1}$ accès séquentiels). Le nombre de groupes de cellules soeurs est limité à $\frac{\|\tilde{\mathcal{C}}^{n+1}\|}{2^d-1}$. Pour chaque groupe trouvé, on accède dans un ordre arbitraire aux valeurs des 5^d nœuds des cellules soeurs pour effectuer le test de compression (soit $(\frac{\|\tilde{\mathcal{C}}^{n+1}\|}{2^d-1} \times 5^d)$ accès séquentiels). Notons $r_\epsilon \in [0, 1]$, le taux moyen de compression pour l'ensemble des groupes de cellules soeurs : le nombre de fois où le test réussit, rapporté au nombre de groupes de cellules. À chaque fois que le test de compression réussit, les cellules soeurs sont remplacées dans le maillage par leur cellule mère. Cela implique une limite de $\frac{\|\tilde{\mathcal{C}}^{n+1}\|}{2^d-1} \times r_\epsilon$ écritures dans $\tilde{\mathcal{M}}^{n+1}$ en respectant l'ordre utilisé pour parcourir les groupes de cellules. Nous faisons alors l'hypothèse que le nombre de cellules est approximativement le même d'une étape de temps à la suivante, autrement dit $\|\mathcal{C}^{n+1}\| \approx \|\mathcal{C}^n\|$. Cette hypothèse permet d'expliciter r_ϵ par $1 - \frac{\|\mathcal{C}^n\|}{\|\tilde{\mathcal{C}}^{n+1}\|}$. On a donc $(\frac{\|\tilde{\mathcal{C}}^{n+1}\|}{2^d-1} \times (1 - \frac{\|\mathcal{C}^n\|}{\|\tilde{\mathcal{C}}^{n+1}\|}))$ accès séquentiels en écriture.

$$\begin{aligned}&\textit{nombre d'accès au maillage pour la phase de compression}\\ &= \left(\tilde{\mathcal{C}}^{n+1}\right)_{L,C,S} + \left(\frac{\|\tilde{\mathcal{C}}^{n+1}\|}{2^d-1} \times 5^d\right)_{L,N,S} + \left(\frac{\|\tilde{\mathcal{C}}^{n+1}\|}{2^d-1} \times \left(1 - \frac{\|\mathcal{C}^n\|}{\|\tilde{\mathcal{C}}^{n+1}\|}\right)\right)_{E,C,S} \textit{accès}\end{aligned} \tag{4.4}$$

Ces formules permettent de comparer le nombre d'accès séquentiels au nombre d'accès aléatoires. En considérant que le nombre de cellules reste approximativement le même d'une étape de temps à la suivante, et que le nombre moyen de nœuds par cellule est supérieur ou égal (dans le cas d'un maillage uniforme) à 2^d, on obtient que le nombre d'accès aléatoires est au moins deux fois supérieur au nombre d'accès séquentiels. Et ce rapport augmente avec le nombre de dimensions.

En conclusion, la majorité des accès aux données sont aléatoires et les structures à base d'arbres (utilisées pour les méthodes AMR classiques [13]) ne sont donc pas les mieux adaptées pour représenter le maillage : un accès aléatoire nécessiterait le parcours d'une branche de l'arbre sur toute sa hauteur. Nous devons donc choisir une structure de données qui permet un accès en temps constant aux éléments quelque soit leur niveau de la hiérarchie.

4.4.2 Tables de hachage

Notre analyse nous amène à utiliser des structures de données basées sur des tables, qui permettent des accès aléatoires rapides aux éléments fins du maillage. Dans un premier temps, nous avons décidé de baser notre structure de données sur des tables de hachage qui permettent un temps d'accès identique quelque soit l'élément du maillage.

Les tables de hachage sont des structures classiques [73] où chaque *élément* stocké est associé à une *clé* au travers de laquelle il est accédé. L'accès à un élément repose sur le principe suivant : la clé est transformée en valeur de *hachage* via une *fonction de hachage.* La valeur de hachage permet de localiser la paire clé–valeur dans la table. Habituellement il s'agit d'une valeur entière représentant l'indice du tableau où l'élément est stocké. On appelle donc indice la valeur de hachage d'une clé. Pour stocker le couple $(\mathcal{M}^n, \mathcal{F}^n)$, on utilise deux tables de hachage :

1. *Table des cellules* : dans la première table de hachage, les clés sont composées du niveau de la cellule dans la hiérarchie et de ses coordonnées entières parmi les autres cellules de même niveau. L'élément associé à chaque clé est un entier qui nous permet d'identifier le processeur auquel est affectée la cellule. Typiquement, il s'agit du rang du processus MPI qui détient la cellule.
2. *Table des nœuds* : dans la deuxième table de hachage, les clés sont les coordonnées des nœuds dans la grille uniforme de taille $(2^{2(L+1)} + 1) \times (2^{2(L+1)} + 1)$. L'élément associé contient la valeur approchée de la fonction de distribution en ce nœud.

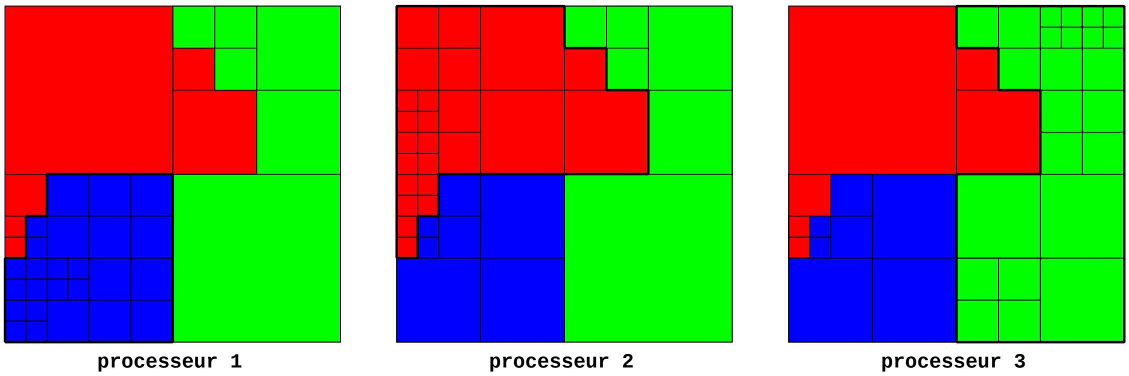

FIG. 4.2 – Distribution des cellules sur les processeurs : chaque processeur a une vue fine de sa région et une vue grossière des régions des autres processeurs.

Nous choisissons une fonction de hachage simple basée sur les valeurs de la clé qui correspondent aux coordonnées des éléments (cellules ou nœuds).

Chaque processeur possède une table des cellules et une table des nœuds qui stockent les informations de sa région. Dans la table des cellules, on retrouve toutes les cellules à l'intérieur de la région du processeur, plus un minimum de cellules "extérieures" qui appartiennent aux autres régions. L'ensemble des cellules dans la table forme une partition du domaine et constitue une vue de l'intégralité du maillage propre au processeur. Les cellules "extérieures" permettent au processeur de savoir quel autre processeur distant possède les informations sur les différentes parties du domaine.

Par exemple, si un processeur p souhaite connaître le rang du processeur p' qui détient une cellule α située à l'extérieur de la région de p, il recherche dans sa table une cellule "extérieure" α' telle que $\alpha \subseteq \alpha'$: le rang de p' est le rang associé à α' dans la table.

Cette information est particulièrement intéressante pour la gestion des communications et ne nécessite qu'un léger surcoût de stockage. Cela offre au processeur une vue fine de sa région et une vue grossière des autres régions. La figure 4.2 montre la vue que chaque processeur possède du maillage dessiné sur la figure 4.1.

4.5 Équilibrage de charge

Nous avons vu que la parallélisation repose sur le partitionnement de l'espace des phases en régions et la distribution des éléments du maillage entre les processeurs. Nous avons vu également que l'adaptativité du maillage (les phases de prédiction et de compression), provoquent des déplacements, des créations ou des disparitions de cellules dans chaque région. La charge de calcul étant proportionnelle au nombre de cellules, cette évolution de la répartition des cellules à chaque pas de temps peut mener à un fort déséquilibre de charge. Comme chaque étape de temps comporte une barrière de synchronisation au niveau de la phase de calcul du champ électrique, si la charge de calcul n'est pas bien répartie entre les processeurs, les processeurs les moins chargés vont attendre que les processeurs les plus chargés atteignent la barrière de synchronisation. Cette attente diminue l'efficacité du code parallèle. La charge de calcul doit donc être équitablement répartie tout au long de l'exécution du programme. C'est pourquoi nous ajoutons à notre algorithme parallèle un mécanisme de régulation dynamique de la charge. Ce mécanisme rédéfinit les régions au cours de l'exécution du code.

Dans la suite, nous commençons par définir les propriétés requises pour les régions et les techniques à mettre en oeuvre pour satisfaire ces propriétés. Puis nous présentons le mécanisme d'équilibrage et son intégration à l'algorithme parallèle.

4.5.1 Propriétés des régions

Les régions formées doivent satisfaire certaines propriétés pour réduire le surcoût en calculs et en communications dû à la décomposition en tâches parallèles. Il faut de plus que la construction de telles régions soit peu coûteuse étant donné qu'elle doit se faire en cours d'exécution. Le coût de la reconstruction doit en tous cas être inférieur au gain induit par les nouvelles régions.

Charge de calcul

Pour réduire le temps d'attente à la barrière de synchronisation à chaque étape de temps, la première propriété à vérifier est que les régions représentent une charge de calcul équivalente sur tous les processeurs. L'algorithme de résolution étant divisé en 4 phases distinctes qui ont chacune une incidence différente sur la charge de calcul, ce problème n'est pas simple et peut être appréhendé selon trois approches [52] :

1. Déterminer quelle est la phase du code la plus coûteuse et développer un mécanisme d'équilibrage pour cette phase. Cette phase doit s'exécuter avec un minimum de déséquilibre même si cela provoque un déséquilibre accru pour d'autres phases. Cette approche est efficace dans le cas où il existe un fort écart de coût entre les différentes phases. Ce n'est pas vraiment le cas avec notre algorithme : si la phase d'évaluation est plus coûteuse que les autres phases, ces dernières ne représentent quand même pas une quantité négligeable.
2. Développer un mécanisme d'équilibrage de charge spécifique pour chacune des phases de l'algorithme. Chaque algorithme doit permettre d'atteindre le meilleur équilibre pour la phase à laquelle il est dédié. L'inconvénient est que cette approche multiplie d'autant le surcoût de l'équilibrage. Dans notre algorithme, où le temps de calcul dépend davantage de la quantité de données que de la complexité des calculs effectués, le coût de l'équilibrage est trop important.
3. Développer un unique mécanisme d'équilibrage commun à toutes les phases de l'algorithme. Moins spécialisé, ce mécanisme fournit un résultat de moins bonne qualité qu'avec la deuxième approche, mais son faible coût permet d'améliorer les performances globales du code.

La troisième approche est celle que nous retenons dans notre cas. Cette approche nécessite de trouver une mesure de la charge qui soit commune à l'ensemble des phases de l'algorithme. Comme l'algorithme est dirigé par les cellules, nous décidons de retenir le nombre de cellules présentes dans la région comme mesure de la charge de calcul qu'elle représente. Cette approximation est satisfaisante étant donné que le nombre de nœuds est approximativement proportionnel au nombre de cellules.

Forme des régions

L'algorithme parallèle peut engendrer un surcoût par rapport à l'algorithme séquentiel. Ce surcoût est dû à l'approximation qui consiste à effectuer la compression uniquement sur les cellules situées à l'intérieur de la région du processeur. Ce défaut de compression augmente avec le nombre de groupes de cellules soeurs composés de cellules appartenant à des régions différentes. On doit donc former des régions qui réduisent le nombre de tels groupes de cellules soeurs.

Un autre surcoût provient du nombre de communications qui détermine le nombre d'appels aux fonctions de communication : chaque appel à un coût incompressible et pour un grand nombre de communications, le coût total peut devenir non négligeable. La quantité de communications est fonction du partitionnement de l'espace des phases. Pour réduire cette quantité, une technique classique est de réduire le nombre d'éléments sur la frontière des régions (ou *cut-size*). La réduction de la *cut-size* par les méthodes vues en section 2.3.2 pose deux problèmes inhérents à la nature de ces méthodes.

1. Les méthodes visant à réduire le nombre d'éléments frontières d'une partition sont des méthodes qui reposent sur la connexité des éléments, et par conséquent, qui utilisent le graphe dual associé au maillage. Dans notre cas, ce graphe doit être reconstruit, ce qui implique un surcoût d'autant plus important que le nombre de dimensions considérées est grand.
2. Ces méthodes permettent de diminuer la quantité de communications, mais peuvent générer des régions allongées et irrégulières si leur forme (*aspect ratio*) n'est pas prise en compte [34]. Or, la prise en compte de cette information sur la forme alourdit le mécanisme, ce qui n'est pas envisageable pour une utilisation fréquente et en cours d'exécution.

Pour ces raisons, nous allons chercher à diminuer la quantité de communications en nous basant uniquement sur des informations géométriques. Comme notre maillage est cartésien, on peut alors raisonnablement penser que des régions "carrées" ou aux frontières régulières permettent de diminuer le périmètre de chaque partition et par conséquent de diminuer le nombre d'éléments aux frontières.

Énoncé des propriétés

Les régions doivent donc satisfaire les propriétés suivantes :

- Les régions doivent représenter une même charge de calcul, ce qui signifie en l'occurrence qu'elles doivent contenir approximativement le même nombre de cellules (pour réduire le temps d'attente).
- Les régions doivent être connexes au sens où il doit être possible de relier deux cellules quelconques de la région par un chemin qui ne traverse que des cellules de la région (pour réduire le défaut de compression).
- Les régions doivent être compactes : de forme carrée ou en tout cas assez régulière (pour réduire le nombre de communications).

4.5.2 Courbe de Hilbert

La connexité d'une région et plus généralement le voisinage d'éléments du maillage est une propriété particulièrement difficile à obtenir et à maintenir lorsque le nombre de dimensions est élevé (> 3) et le maillage adaptatif. De même, les méthodes basées sur des informations géométriques peuvent être difficiles à appréhender ou à appliquer dès lors que le nombre de dimensions est supérieur à 3. Or, nous voulons que les différentes techniques de parallélisation utilisées soient extensibles en dimension pour nous permettre

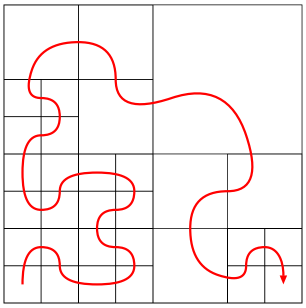

FIG. 4.3 – Courbe de Hilbert multi-résolution correspondant à un maillage dyadique 2D.

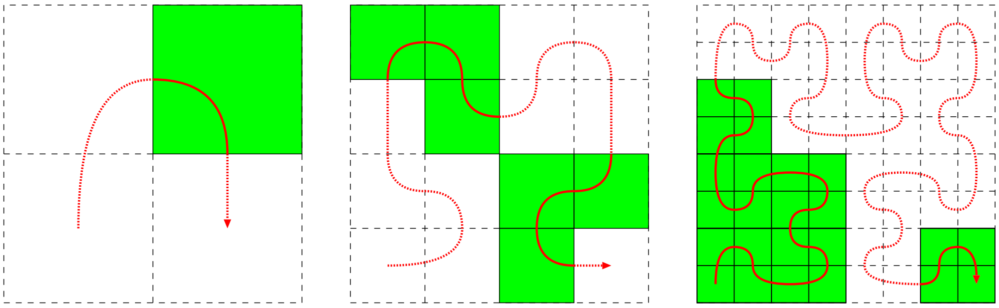

FIG. 4.4 – Construction hiérarchique d'une courbe de Hilbert 2D multi-résolution.

de développer des codes 4D ou plus. À cet effet, une des techniques de partitionnement vues en section 2.3.1 propose de réduire la dimensionnalité du problème : il s'agit des méthodes utilisant des courbes remplissantes, et plus particulièrement la courbe de Hilbert.

Parmi les différents types de courbes existantes (voir [96]), la courbe de Hilbert possède des propriétés particulièrement intéressantes. D'abord, il s'agit d'une courbe de type binaire (calculée en base deux) et elle permet par conséquent de décrire de façon naturelle les cellules d'un maillage dyadique. De plus, elle est généralisable en dimension, et peut donc permettre de linéariser un domaine 4D ou plus [1]. Enfin, elle permet de décrire des maillages adaptatifs : la figure 4.3 montre la courbe multi-résolution passant par chaque cellule d'un maillage dyadique. Cette courbe est construite récursivement, comme le montre la figure 4.4. C'est cette construction récursive, en conjonction avec les diverses rotations et symétries, qui donne à la courbe de Hilbert ses propriétés de localité.

Cette propriété de localité signifie que les éléments consécutifs dans la courbe de Hilbert sont toujours des éléments voisins dans l'espace de départ. La réciproque est fausse, mais cette propriété est suffisante pour assurer la connexité des régions. On peut noter que dans le cas d'une autre courbe de remplissage connue, la z-curve, le motif de base est toujours orienté de la même manière ; ce qui fait que la courbe peut faire des *sauts* qui peuvent affecter la localité des éléments.

Dans la pratique, nous allons implanter les courbes de Hilbert à l'aide du diagramme d'état introduit dans [75, 76] et donné dans la figure 4.5. Chaque état de ce diagramme représente une configuration du motif de base de la courbe de Hilbert. En dimension deux, il y a quatre configurations qui représentent les différentes manières de parcourir quatre cellules soeurs. Chaque cellule soeur est identifiée par sa position relative. Par exemple dans l'état 0, l'ordre imposé par la courbe est $(0,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (1,0)$.

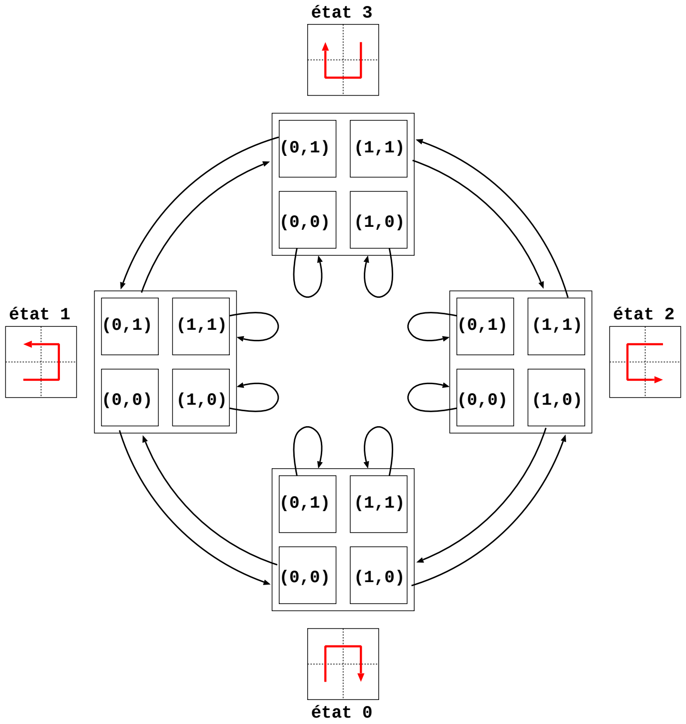

FIG. 4.5 – Diagramme d'état pour la construction récursive d'une courbe de Hilbert 2D.

Pour un état donné, chaque cellule est reliée à un état qui correspond à la configuration de la courbe lorsque cette cellule est raffinée. Par conséquent, ce nouvel état donne l'ordre de parcours sur les quatre cellules filles créées lors d'un raffinement. Prenons par exemple la construction de la courbe de Hilbert pour une configuration de départ à l'état 0. La cellule correspondant à l'état de départ est raffinée récursivement jusqu'à obtenir le maillage voulu. À chaque raffinement, l'état courant de la courbe de Hilbert est utilisé pour déterminer le motif correspondant aux cellules filles, et par conséquent l'ordre de parcours sur ces cellules.

Cette construction de la courbe peut être rapprochée de la description du maillage par un 4-arbre. Chaque nœud de cet arbre correspond à une cellule de la hiérarchique du maillage et peut stocker son état. Les quatres branches d'un nœud de l'arbre sont disposées dans l'ordre induit par cet état stocké au nœud. Parcourir cet arbre en *profondeur d'abord* revient alors à suivre l'ordre imposé par la courbe de Hilbert.

4.5.3 Mécanisme d'équilibrage dynamique

Dans le cas de notre algorithme parallèle, l'équilibrage de charge consiste à rédéfinir dynamiquement les régions. Dans cette section, nous commençons par montrer comment créer des régions ayant de bonnes propriétés. Puis nous définissons un moyen de détecter un déséquilibre de charge. Nous montrons ensuite comment mettre à jour les régions lorsqu'un déséquilibre est détecté. Enfin, nous montrons comment la détection du déséquilibre et la mise à jour des régions s'intègre à notre algorithme parallèle.

Création des régions

La création des régions est effectuée une seule fois. Elle peut donc être plus coûteuse que la mise à jour des régions qui elle est effectuée plusieurs fois au cours d'une simulation. Elle réalise un équilibrage statique au début de la simulation.

Les régions sont construites à partir du maillage initial. Chaque région construite doit avoir une charge approximativement égale à la charge idéale $\mathcal{L}_I$, qui est le nombre total de cellules du maillage, que nous noterons $\mathcal{L}$, divisé par P, le nombre de processeurs.

Pour former les régions, nous considérons la courbe de Hilbert qui passe par chacune des cellules du maillage dyadique et le 4-arbre qui représente la hiérarchie des cellules du maillage et dont les branches sont orientées par les motifs de la courbe. Chaque nœud de l'arbre est la racine d'un sous-arbre. Le nœud est associé à l'ensemble des cellules qui sont les feuilles du sous-arbre ainsi qu'à une charge égale au cardinal de cet ensemble.

L'algorithme de partitionnement consiste à parcourir l'arbre en profondeur d'abord tant que la somme des charges associées au nœud courant et à la région en construction est supérieure à la charge idéale. Si cette somme est inférieure à la charge idéale, alors on affecte le sous-arbre engendré par ce nœud à la région en construction et au processeur correspondant. On continue ensuite le parcours de l'arbre à partir du prochain nœud dans un parcours en largeur d'abord. Si la somme des charges est égale à la charge idéale plus ou moins un, alors la région est complète et on construit la prochaine région à partir du prochain nœud selon le parcours en largeur. (voir l'algorithme 8.)

Algorithme 8 : Création de régions équilibrées.

```
Entrées : $a$ = racine de l'arbre, $l = 0$, $p = 0$
tant que $p < P$ faire
    $l' = l +$ charge de $a$
    si $l' > \mathcal{L}_I + 1$ alors
        remplacer $a$ par son premier fils
    sinon si $l' < \mathcal{L}_I - 1$ alors
        affecter le sous-arbre de racine $a$ au processeur $p$
        remplacer $a$ par le nœud suivant
        $l = l'$
    sinon
        affecter le sous-arbre de racine $a$ au processeur $p$
        remplacer $a$ par le nœud suivant
        $p = p + 1$
        $l = 0$
    fin
fin
```

Cet algorithme permet de créer des régions équilibrées, mais qui ne sont pas forcément compactes. Cela peut engendrer un surcoût en communications dû à un trop grand nombre d'éléments sur les frontières. Cela peut aussi engendrer une baisse du taux de compression, et par conséquent une augmentation de la charge totale.

Pour améliorer la compacité des partitions, nous introduisons un facteur d'erreur $\varepsilon \in [0, 1]$ qui permet de réaliser un compromis entre la forme des régions et la charge idéale. Avec ce facteur d'erreur, la charge associée à la région d'un processeur p peut varier dans une certaine mesure :

$$(1-\varepsilon)(\mathcal{L}_I - 1) \leq \mathcal{L}_p \leq (1+\varepsilon)(\mathcal{L}_I + 1) \tag{4.5}$$

Ce degré de liberté supplémentaire permet de choisir des coupes de l'arbre effectuées au plus près de la racine. Par conséquent, il y a de plus fortes chances que des cellules soeurs aux niveaux fins se trouvent dans la même région et le taux de compression est meilleur.

Détection du déséquilibre

Pour détecter un déséquilibre de charge, nous nous basons sur une connaissance globale de la répartition de la charge sur les processeurs. Cela nous permet d'obtenir l'équilibre de charge en une mise à jour des régions.

Les P processeurs s'échangent la charge de leur région, ce qui nécessite une synchronisation. Chaque processeur $p_i, i \in \{0, \cdots, P-1\}$ peut alors déterminer la charge totale du système $\mathcal{L}_T$ et la charge idéale $\mathcal{L}_I = \mathcal{L}_T/P$. Chaque processeur calcule la différence de charge entre le processeur le plus chargé et le processeur le moins chargé :

$$\mathcal{L}_d = \max(\mathcal{L}_i)_{0 \leq i \leq P} - \min(\mathcal{L}_j)_{0 \leq j \leq P}$$

Quand la différence de charge $\mathcal{L}_d$ est supérieure à un certain seuil, il y a déséquilibre. Le seuil est fixé par l'utilisateur. en fonction du coût estimé pour la synchronisation.

Mise à jour des régions

La mise à jour des régions s'effectue après la détection du déséquilibre. A ce stade, nous disposons des charges locales $\mathcal{L}_p$ pour chaque processeur $p \in \{0, \cdots, P\}$. Nous disposons également de la charge totale $\mathcal{L}_T$ et idéale $\mathcal{L}_I$. Par conséquent, chaque processeur peut calculer la quantité de charge qu'il a en excès ou en défaut par $\bar{\mathcal{L}}_p = \mathcal{L}_p - \mathcal{L}_I$. Si $\bar{\mathcal{L}}_p$ est négatif, le processeur a un défaut de charge et ce nombre représente la quantité de cellules qu'il doit recevoir. Si $\bar{\mathcal{L}}_p$ est positif, le processeur a un excès de charge et ce nombre représente la quantité de cellules qu'il devra envoyer.

Pour atteindre l'équilibre en une seule étape de migration des éléments, nous utilisons le fait que les régions et les processeurs sont ordonnés par la courbe de Hilbert. Chaque processeur peut être considéré comme un pivot, et divise l'ensemble des processeurs en deux sous-ensembles : les processeurs précédents et les processeurs suivants dans la courbe.

Chaque processeur détermine le nombre de cellules qu'il va devoir communiquer avec ses deux voisins. Pour un processeur $p \in \{0, \cdots, P-1\}$, le calcul du nombre de cellules venant des processeurs précédents $\mathcal{L}_{prec}$ et du nombre de cellules venant des processeurs suivants $\mathcal{L}_{suiv}$ est effectué comme suit :

$$\mathcal{L}_{prec} = \sum_{i=0}^{p-1} \bar{\mathcal{L}}_i \quad , \quad \mathcal{L}_{suiv} = \sum_{i=p+1}^{P-1} \bar{\mathcal{L}}_i \tag{4.6}$$

Si le nombre de cellules $\mathcal{L}_{prec}$ (resp. $\mathcal{L}_{suiv}$) est positif, alors le processeur p doit recevoir $\mathcal{L}_{prec}$ (resp. $\mathcal{L}_{suiv}$) cellules du processeur voisin précédent (resp. suivant). Si le nombre de cellules $\mathcal{L}_{prec}$ (resp. $\mathcal{L}_{suiv}$) est négatif, alors le processeur p doit envoyer $\mathcal{L}_{prec}$ (resp. $\mathcal{L}_{suiv}$) cellules à son voisin précédent (resp. suivant).

Le choix des cellules à migrer se porte naturellement sur les cellules placées aux extrémités du morceau de la courbe de Hilbert qui définit la région du processeur p. De cette manière la connexité des régions est conservée par la mise à jour. Par contre les autres propriétés relatives à la forme des régions ne sont pas conservées.

4.5.4 Intégration du mécanisme à l'algorithme

Nous voulons effectuer un seul équilibrage par étape de temps pour en limiter le surcoût. De plus, cet équilibrage doit être intégré dans l'algorithme de manière à en minimiser le coût et à maximiser l'impact des nouvelles régions. Nous décidons de réaliser l'équilibrage (la détection du déséquilibre et la mise à jour des régions) entre les phases de prédiction et d'évaluation. Cet emplacement permet d'obtenir de meilleures performances pour plusieurs raisons :

- la prédiction du maillage est la phase qui modifie le plus la répartition des cellules sur les régions existantes. Donc, il est intéressant de placer l'équilibrage après la phase de prédiction pour tenir compte de la répartition la plus récente.
- les phases d'évaluation et de compression sont les phases les plus coûteuses de l'algorithme. Par conséquent ce sont celles qu'il convient d'équilibrer au mieux pour éviter d'importants temps d'attente aux barrières de synchronisation. On doit placer l'équilibrage avant ces phases.
- les valeurs sur le maillage prédit sont calculées uniquement au moment de l'évaluation. Par conséquent, la migration des éléments du maillage consécutive aux modifications sur les régions ne concerne que des cellules. Aucune valeur de nœud n'est transférée, ce qui réduit le surcoût en communications de l'équilibrage dynamique.

4.6 Résultats

L'implantation parallèle représente quelque 11000 lignes de code. Le code a été écrit en C++ avec appels à MPI [84]. Dans cette implantation MPI nous avons utilisé la fonction non-bloquante `MPI_Isend` plutôt que `MPI_Ibsend`, mais en nous assurant du fonctionnement selon le mode bufferisé en choisissant la taille des messages selon l'architecture considérée. La structure de données utilise les `hash_map` de la STL [107] pour stocker les informations sur les nœuds et sur les cellules, ce qui nous permet, dans un premier temps, de nous affranchir de la difficulté liée au développement d'une structure de données adaptative *ad-hoc*. Le code est paramétré en dimension grâce à l'utilisation de nombreux *templates*. Les expériences réalisées, dont les détails sont donnés en annexe, concernent le cas test de la focalisation uniforme d'un faisceau semi-Gaussien (voir B.1) sur un cluster d'Itaniums et une Origin 3800 (voir C). Ces résultats ont donné lieu à deux publications [56, 81].

La figure 4.6 montre la fonction de distribution dans l'espace des phases en deux dimensions après 10, 30 et 50 pas de temps de la simulation. Les positions x se trouvent en abscisse et les vitesses v se trouvent en ordonnée.

La figure 4.7 montre l'état du maillage adaptatif pour les même étapes de temps. On remarque que les cellules fines sont bien présentes au niveau des zones à fort gradient et le long des structures fines qui se créent au cours de la simulation.

La figure 4.8 montre les régions du domaine qui sont attribuées aux différents processeurs pour une simulation exécutée sur huit processeurs. On voit bien que la mise à

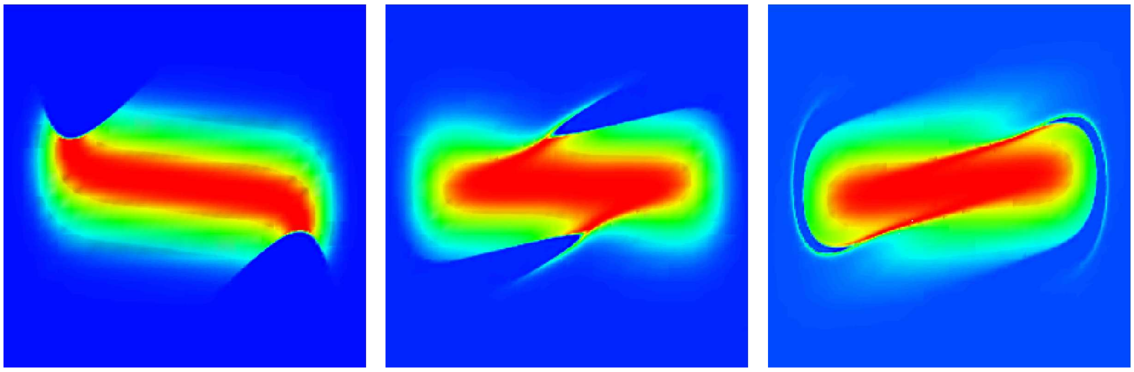

FIG. 4.6 – Valeurs de la fonction de distribution 2D aux pas de temps 10, 30 et 50.

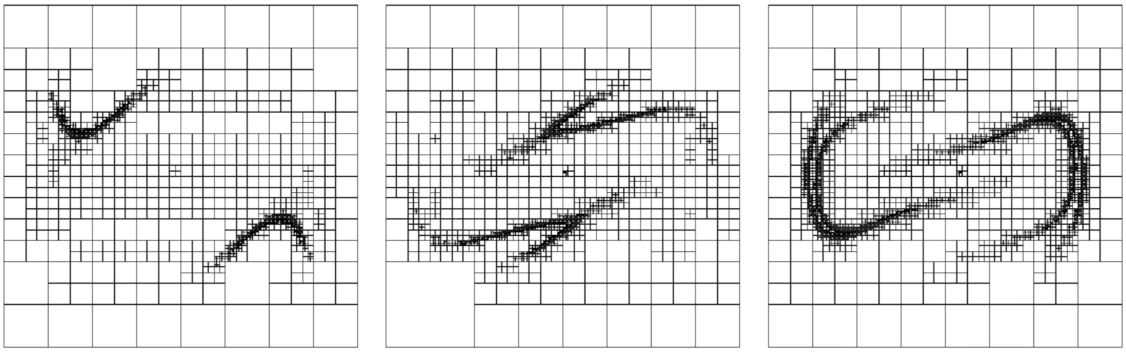

FIG. 4.7 – Maillage adaptatif 2D aux pas de temps 10, 30 et 50.

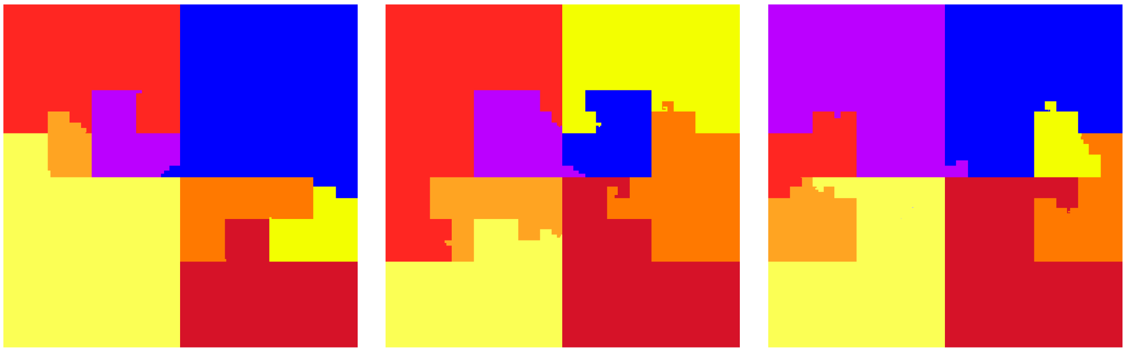

FIG. 4.8 – Régions pour 8 processeurs aux pas de temps 10, 30 et 50.

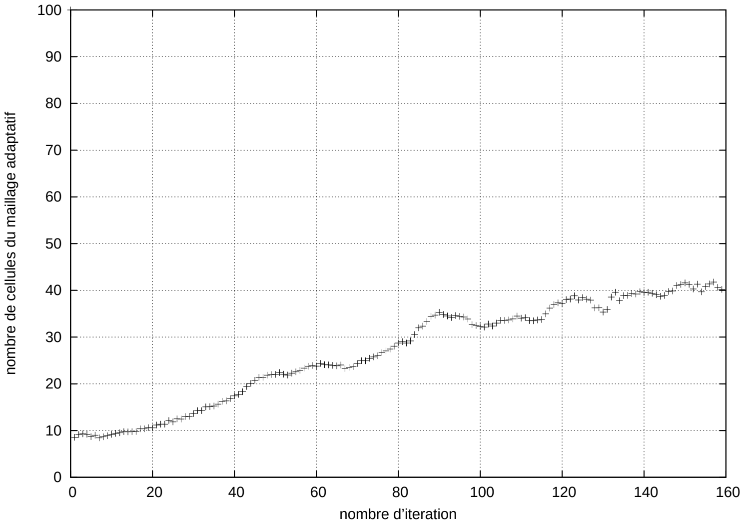

Fig. 4.9 – Évolution du nombre de cellules du maillage adaptatif au cours des 160 étapes de la simulation.

jour des régions par la courbe de Hilbert a tendance à respecter la construction par blocs des régions mais elle engendre aussi des petites irrégularités aux endroits où la courbe est coupée. Cette figure montre aussi que la courbe de Hilbert a l'inconvénient d'être une courbe ouverte dont l'origine est fixée dans le coin en bas à gauche, et dont la fin est fixée dans le coin en bas à droite. Ces points d'ancrage empêchent les régions d'évoluer en complète liberté pour suivre l'évolution des particules. Cela peut provoquer notamment d'importantes migrations de données pour équilibrer globalement la charge et donc un surcoût de communications qui peut faire baisser les performances du programme.

La figure 4.9 montre le taux de compression du maillage pour une simulation avec $L = 6$ sur 4 processeurs. Le nombre de cellules conservées par rapport à un code uniforme varie entre 10 et 40%.

Les figures 4.10 et 4.11 montrent le temps d'exécution respectivement sur le cluster HP et sur l'O3800. Chaque courbe correspond à un niveau de raffinement maximum L différent.

Les figures 4.12 et 4.13 montrent l'accélération obtenue respectivement sur les deux architectures précédentes. Notre code permet d'atteindre une efficacité de 70% pour une simulation sur 16 processeurs. Au delà de 16 processeurs, l'efficacité du programme baisse rapidement à cause du surcoût lié à la parallélisation. Le surcoût inhérent aux appels aux fonctions MPI pourrait être réduit par l'utilisation de la fonction `MPI_Testsome` à la place de `MPI_Testany`. De même l'emploi de communications persistantes peut réduire le surcoût induit par des appels répétés aux primitives système. Par contre, les calculs supplémentaires introduits par la parallélisation (défaut de compression, redondance des calculs des valeurs aux nœuds frontière) sont trop nombreux pour permettre d'obtenir un code extensible.

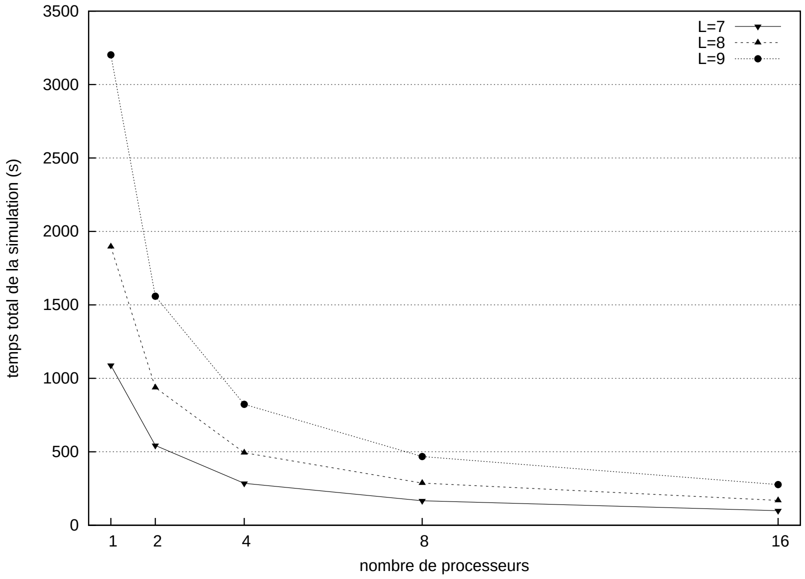

FIG. 4.10 – Temps total d'exécution sur le cluster d'Itaniums.

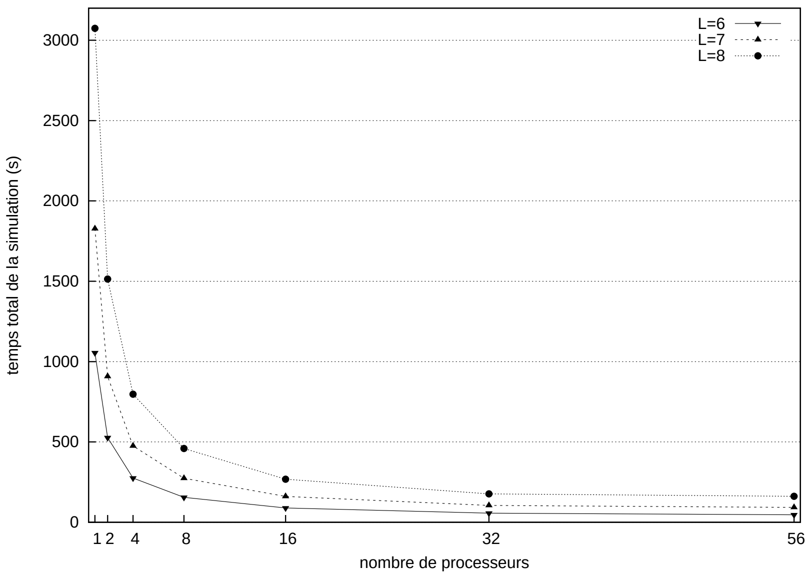

FIG. 4.11 – Temps total d'exécution sur Origin 3800.

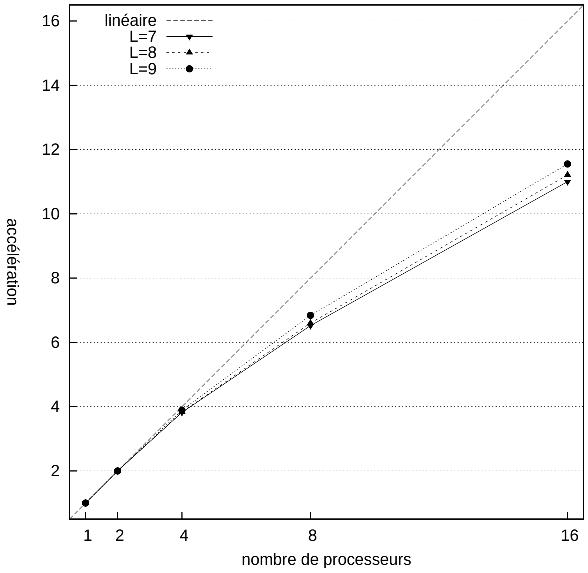

FIG. 4.12 – Accélération sur le cluster d'Itaniums.

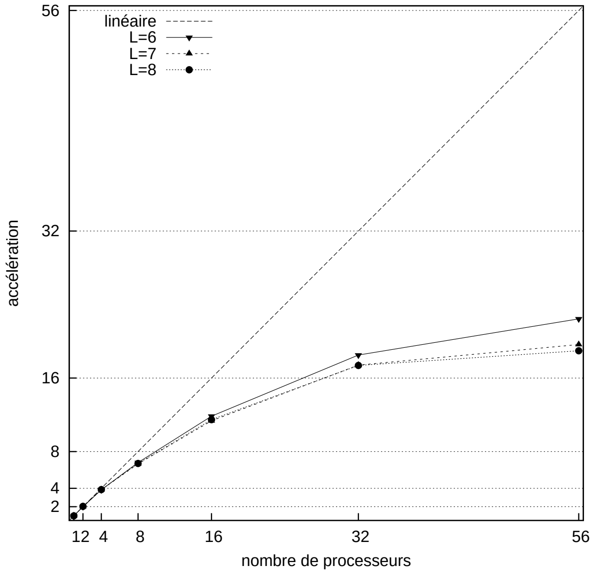

FIG. 4.13 – Accélération sur Origin 3800.

Chapitre 5

Extension du code au 4D

Les différentes techniques de parallélisation présentées dans le chapitre précédent sont extensibles aux dimensions supérieures. Cela nous a permis de développer un code qui prend le nombre de dimensions de l'espace des phases en paramètre. Il est donc théoriquement possible d'exécuter des cas test 4D avec ce code et d'étudier le comportement du code lorsqu'on augmente le nombre de dimensions. Cependant, les premières expériences ont montré que d'une part, l'usage mémoire impose de fortes limitations concernant la taille des problèmes qu'il est possible de traiter, et d'autre part, que le temps d'exécution du code, qui exhibe de bonnes performances en 2D, explose complètement en 4D.

Un profiling du code montre qu'en 4D, le temps passé à accéder aux éléments du maillage devient prépondérant. Ceci est dû à différentes causes : d'une part, le nombre d'accès à la structure augmente exponentiellement avec la dimension. D'autre part, les opérations de gestion du maillage (raffinement, déraffinement de cellules, recherche de la cellule contenant un point), dont la complexité dépend de l'indexation des éléments du maillage, sont plus coûteuses. Enfin, le coût d'un accès à un élément dans la structure de données augmente sensiblement. Ceci est dû notamment au fait que les éléments sont plus dispersés en mémoire. Il faut en effet résoudre des problèmes de localité en mémoire plus complexes, puisqu'il faut désormais ordonner dans un espace 1D (la mémoire) les données d'un espace 4D (le maillage). L'accumulation de ces différentes causes provoque l'effondrement des performances en 4D.

Nous considérons donc dans ce chapitre un ensemble d'optimisations qui ont pour but d'améliorer les performances du code dans le cas 4D. Dans un premier temps, nous nous concentrons sur des améliorations de la structure de données à base de tables de hachage. Nous montrons qu'un changement d'indexation des éléments du maillage permet de réduire la quantité de mémoire utilisée ainsi que le coût de l'accès aux éléments. Dans un deuxième temps, nous proposons des transformations plus radicales de la structure de données. Nous proposons divers extensions de notre structure de données visant à améliorer la localité des données et à utiliser plus efficacement le cache. Enfin nous remplaçons les tables de hachage par des tableaux. Ces modifications de la structure de données nous permettent de réduire considérablement le temps d'accès aux éléments du maillage au prix d'un faible surcoût en terme d'usage mémoire.

5.1 Optimisation de la table de hachage

Les optimisations que nous décrivons dans cette section ont pour but de réduire la quantité de mémoire requise pour stocker la table ainsi que le temps d'accès aux éléments.

Étant donné que les clés doivent être stockées dans la table (en plus des valeurs associées), la quantité de mémoire utilisée pour stocker la table est proportionnelle à la taille de la représentation en mémoire des clés. Donc, nous pouvons réduire l'usage mémoire en élaborant une nouvelle représentation mémoire plus compacte pour les clés. Cela signifie une nouvelle indexation des élements dans notre code. D'autre part, pour accéder à un élément, il faut calculer l'image de la clé par la fonction de hachage, et plus la fonction de hachage est complexe, plus le coût d'un accès à un élément est important. En modifiant l'indexation des éléments, on agit aussi sur la complexité de la fonction de hachage. La fonction de hachage est alors construite pour une avoir une complexité minimum par rapport à la nouvelle indexation et pour réduire le nombre de collisions dans la table.

5.1.1 Indexation des éléments

Chaque élément dans la table de hachage est un couple clé–valeur. Pour les cellules, une clé est un couple composé du niveau l de la cellule et de ses coordonnées dans la grille uniforme de niveau l. Pour les nœuds, une clé est formée des coordonnées du nœud dans une grille uniforme de taille $(2^(L+1)+1)^d$ avec d, le nombre de dimensions du maillage et L, son niveau le plus fin.

Dans notre code d'origine, une clé contient les coordonnées dans une grille et ces coordonnées sont stockées dans un tableau d'entiers. Lorsque le nombre de dimensions augmente, cette représentation devient très coûteuse en terme d'usage mémoire. De plus, l'utilisation des coordonnées sous cette forme accroît la complexité du calcul de la fonction de hachage.

Nous choisissons donc de stocker ces coordonnées sur un simple entier long. Pour un nœud, chaque coordonnée dans une dimension peut être représentée en utilisant $L+1$ bits (où L est le niveau le plus fin du maillage). La représentation binaire de la clé d'un nœud est le résultat de la concaténation des représentations binaires de chaque coordonnée, soit :

$$I_0 I_1 \cdots I_{d-1},$$

où $I_k, k \in \{0, \ldots, d-1\}$ est le code binaire de la k-ième coordonnée du nœud. Le nombre de bits nécessaires pour représenter une clé est donc de $d \times (L+1)$. La figure 5.1 montre comment la clé d'un nœud est construite à partir de la représentation binaire de ses coordonnées sur chaque dimension (dans le cas 2D). Cette représentation autorise donc, avec un entier long sur 64 bits, une résolution maximum de 65536 points par dimension en 4D, et de 1024 points par dimension en 6D. Ce qui est suffisant pour les simulations considérées.

De plus, avec cette représentation, retrouver les coordonnées d'un nœud consiste en un simple masquage de bits pour ne sélectionner que ceux qui correspondent à une dimension donnée.

De la même manière, les coordonnées des cellules peuvent également être stockées sur un seul entier long. Une idée intéressante pour réduire la complexité de la fonction de hachage est d'organiser les bits par niveau dans la hiérarchie de cellules (et non par dimension comme pour l'indexation des nœuds). La représentation binaire de la clé d'une

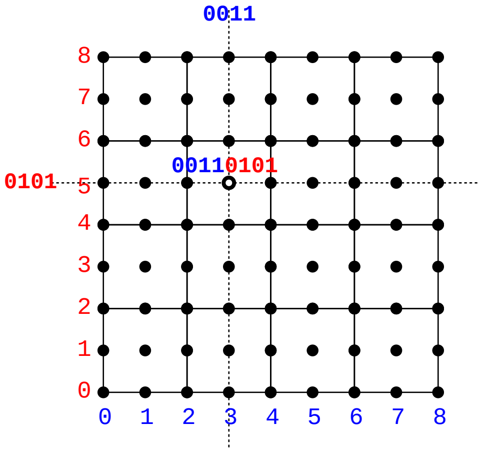

FIG. 5.1 – Indexation des nœuds en 2D par représentation binaire des coordonnées.

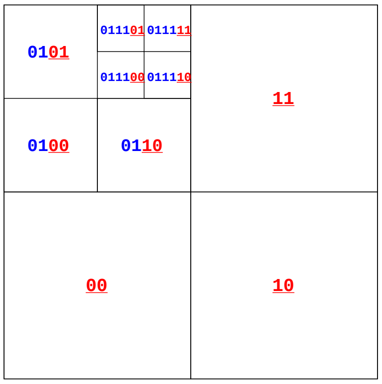

FIG. 5.2 – Indexation des cellules en 2D par représentation binaire de leur position dans la hiérarchie.

cellule, disons α, est alors :

$$J_0 J_1 \cdots J_{l-1},$$

où J_k, $k \in \{0, \ldots, l-1\}$ est le code binaire de la position de α dans une grille de taille 2^d si $k = 0$, ou sa position au niveau k relativement au niveau $k-1$ si $k > 0$. Chaque position peut être représentée en utilisant d bits (où d est le nombre de dimensions). La figure 5.2 montre comment la représentation binaire de la clé d'une cellule est contruitre hiérarchiquement (dans le cas 2D).

Le nombre de bits nécessaires pour représenter la clé d'une cellule dépend de son niveau : il est égal à $l \times d$, où l est le niveau de la cellule. Avec un entier long sur 64 bits, le nombre maximum de niveaux de la hiérarchie de cellules autorisé par cette indexation est de 16 en 4D, et de 10 en 6D. Cette représentation est suffisante pour les simulations considérées.

De plus, avec cette représentation, un simple décalage à droite de d bits permet de retrouver la clé de la cellule mère de α à partir de la clé de α.

5.1.2 Fonction de hachage

On a vu dans la section précédente que les clés sont stockées sur un entier (long). Pour la table des noeuds, nous choisissons la fonction de hachage la plus simple possible qui, étant donnée une clé, associe l'entier correspondant. La complexité de la fonction de hachage est donc minimum.

Concernant la table des cellules, nous avons vu que les clés de cellules ont des représentations binaires de différentes longueurs (en fonction du niveau des cellules). Si on stocke ces représentations binaires sur un entier long (en mettant des 0 dans les bits de poids le plus fort lorsque la longueur de la représentation est inférieure à la longueur de l'entier long), alors des cellules différentes dans le maillage peuvent être associées à des entiers identiques (par exemple en 2D, les cellules dont les clés sont 000100 et 0100 se situent dans des zones différentes). On ne peut donc pas choisir la même fonction de hachage que précédemment sauf à autoriser des collisions. Afin d'éviter les collisions, nous choisissons pour fonction de hachage, la fonction qui, étant donnée la clé d'une cellule, associe l'entier obtenu en décalant la représentation binaire de la clé de $(L - l)\times d$ bits vers la gauche, où l est le niveau de la cellule. Grâce à ce décalage, seules des cellules ne pouvant pas coexister dans le maillage (parce qu'elles se recouvrent) peuvent avoir le même hachage (par exemple en 2D, les cellules dont les clés sont 010000 et 0100 se superposent). Puisque les cellules d'un maillage dyadique forment une partition, il ne peut pas y avoir collision, on a donc un *hachage parfait.*

Une autre bonne propriété de ce décalage à gauche est que les bits de poids les plus forts correspondent aux niveaux les plus grossiers dans la hiérarchie de cellules. Cela implique que des éléments contigus dans la table de hachage correspondent à des cellules situées dans une même zone du maillage. En d'autres termes, cette fonction de hachage préserve la localité des cellules. Ainsi il est possible, en parcourant la table des cellules, de parcourir les cellules du maillage de proche en proche.

5.1.3 Autres optimisations

Nous avons réalisé d'autres optimisations liées à l'utilisation de la table des noeuds et qui concernent la phase d'évaluation de notre algorithme parallèle. Une première optimisation vise à éviter de calculer plusieurs fois la même valeur des noeuds à la frontière de plusieurs régions. Une deuxième optimisation vise à réduire le temps d'accès aux valeurs nécessaires pour l'interpolation en instaurant un cache logiciel.

Traitement des nœuds frontières. Dans notre algorithme parallèle, la valeur en un nœud a est calculée par interpolation par l'unique processeur P_a qui détient la cellule dans laquelle se trouve le point $\mathcal{A}^+(a)$ issu du transport du nœud a. Si le noeud a se trouve dans une région allouée à un autre processeur que P_a, disons P, alors P envoie les coordonnées du point $\mathcal{A}^+(a)$ au processeur P_a. À la réception de ces coordonnées, le processeur P_a calcule la valeur de a et retourne cette valeur au processeur P. Si le noeud a se trouve à la frontière de plusieurs régions et que ces régions sont alloués à des processeurs différents de P_a (voir figure), alors P_a calcule plusieurs fois la même valeur.

Notre optimisation de l'algorithme consiste à mémoriser la valeur de a dans la table des noeuds du processeur P_a. L'algorithme est modifié de la façon suivante : Le processeur P envoie non seulement les coordonnées du point $\mathcal{A}^+(a)$, mais aussi la clé du noeud a. A la réception, le processeur P_a consulte sa table des noeuds pour vérifier si le noeud a n'est pas déjà présent. S'il est présent, alors la valeur associée au noeud a dans la table est directement retournée au processeur P sans effectuer de calcul. Si le noeud a n'est pas dans la table, alors P_a calcule la valeur de a, ajoute le couple clé–valeur du noeud a dans sa table, puis retourne la valeur de a au processeur P.

Cette optimisation a malheureusement deux inconvénients : d'une part, le nouvel algorithme nécessite plus de mémoire puisque P_a stocke davantage de noeuds dans sa table des noeuds. D'autre part, le volume de communications est accru puisque la clé du noeud a doit être communiquée en plus des coordonnées du point $\mathcal{A}^+(a)$.

Accès rapide aux valeurs pour l'interpolation. Dans la phase d'évaluation, chaque processeur parcourt les noeuds de sa région. Ce parcours consiste en le parcours des cellules (dans l'ordre de la table des cellules) et pour chaque cellule, le parcours de ses noeuds. Puisque la fonction de hachage préserve la localité des cellules, alors des noeuds proches selon cet ordre de parcours sont également proches dans le maillage. Or, plus des nœuds sont proches, plus la probabilité est grande que les points issus de leur transport appartiennent à la même cellule. Cette propriété rend possible l'utilisation du cache matériel : lors du calcul de la valeur interpolée en ces points, les valeurs aux noeuds de la cellule peuvent être déjà présentes dans le cache. Cependant, les valeurs aux noeuds d'une cellule peuvent être très éloignées dans la table des noeuds, ce qui conduit à de nombreux défauts de cache et une mauvaise utilisation de celui-ci.

Notre optimisation consiste à instaurer un cache logiciel. Ce cache logiciel consiste en un petit tableau contenant les valeurs aux noeuds des cellules le plus récemment accédées. Dans ce tableau, les valeurs aux noeuds d'une cellule sont contiguës en mémoire ce qui autorise une bonne utilisation du cache matériel. La mise à jour de ce cache logiciel se fait selon une stratégie FIFO.

5.1.4 Résultats

Nous reportons ici les expériences réalisées pour mesurer l'impact des optimisations. Le simulation réalisé correspond au cas test de la focalisation d'un faisceau semi-Gaussien en quatre dimensions (voir B.3). La machine parallèle utilisée pour ces tests est le cluster d'Itaniums (voir C).

La tableau 5.1 montre que le temps gagné par le changement d'indexation est considérable. Ce gain est dû en grande partie à la réduction du coût de la fonction de hachage. Les fonctions de manipulation du maillage (calcul de la clé de la cellule mère, calcul de la cellule contenant un point donné, etc...) ont aussi vu leur coût réduit de manière importante. De plus, la table de hachage elle-même est moins volumineuse, ce qui permet de faire baisser le temps dû aux allocations dynamiques de mémoire qui sont non négligeables pour le traitement de grandes masses de données.

# procs	1	2	4	8	16
tableau d'entiers	3901.8	2036.1	1060.1	606.4	337.2
entier long	1933.7	1016.3	509.6	255.3	142.4

TAB. 5.1 – Effet du changement d'indexation

Le tableau 5.2 montre le gain obtenu grâce à l'utilisation du cache logiciel. Le cache utilisé pour cette expérience contient les valeurs aux nœuds de 7 cellules. Nous avons choisi ce nombre car il correspond au nombre moyen de cellules accédées pour l'interpolation des valeurs aux noeuds d'une même cellule. L'impact de ces différentes optimisations, en terme de temps d'exécution, est résumé par la figure 5.3.

# procs	1	2	4	8	16
sans cache logiciel	1933.7	1016.3	509.6	255.3	142.4
avec cache logiciel	1635.1	845.1	428.4	218.4	118.8
avec hachage optimisé	1563.3	805.7	407.1	207.9	113.8

TAB. 5.2 – Effet de l'utilisation du cache logiciel

Enfin, nous avons mené des expériences pour mettre en évidence l'influence de la fonction de hachage sur la localité des cellules. Nous avons mesuré le nombre moyen de cellules différentes du maillage $\mathcal{M}^n$ qui contiennent les points issus du transport des nœuds

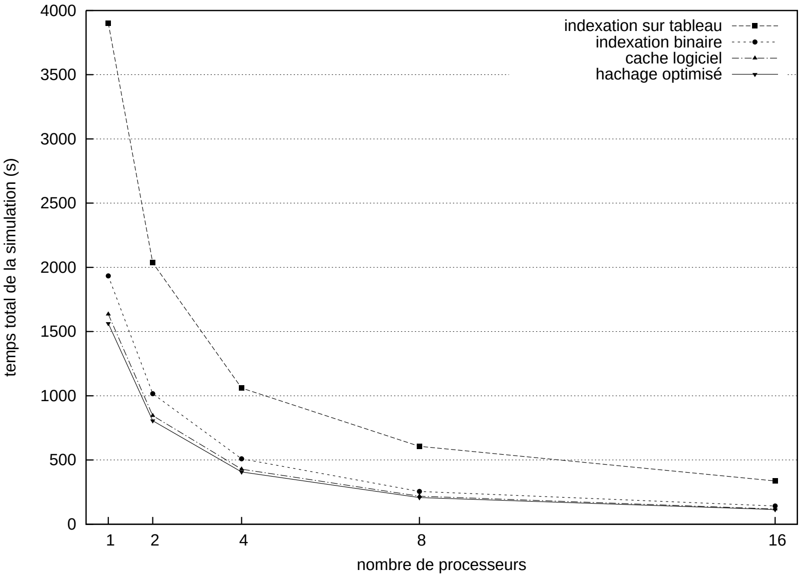

Fig. 5.3 – Impact des différentes optimisations de la structure de données sur le temps d'exécution.

d'une même cellule de $\mathcal{M}^{n+1}$. Nous obtenons une moyenne de 6.9 pour l'ancienne fonction de hachage, et de 4.3 pour la nouvelle. De même, nous avons mesuré le nombre moyen d'accès à une même cellule de $\mathcal{M}^{n}$ lors du traitement des nœuds d'une cellule de $\mathcal{M}^{n+1}$: on obtient une moyenne de 4.7 anciennement contre 6.1 avec la nouvelle fonction de hachage qui préserve la localité des cellules. Ainsi on remarque que ce changement de fonction de hachage permet non seulement d'améliorer l'utilisation du cache logiciel, mais aussi d'en réduire la taille. La dernière ligne du tableau 5.2 traduit l'influence de cette amélioration sur le temps d'exécution.

La figure 5.4 montre l'impact des différentes optimisations sur l'accélération du programme parallèle. Il faut noter que seule la version de référence n'implante pas l'optimisation sur les nœuds frontières. Or, cette optimisation vise directement à réduire le surcoût, en terme de calculs, de la parallélisation. Cela explique la différence entre l'accélération du code de référence et celle obtenue avec les codes implantant cette optimisation. Au final, on obtient une efficacité de 85% sur 16 processeurs avec la prise en compte des nœuds frontière.

5.2 Modification de la structure de données

Nous considérons ici des transformations de la structure de données. Ces modifications visent à obtenir une structure de données performante en terme de temps d'accès aux éléments et de quantité de mémoire utilisée. Nous commençons par présenter une modification dont le but est d'optimiser l'utilisation du cache matériel. Puis, nous proposons une nouvelle structure de données basée sur des tableaux qui privilégie le temps d'accès aux éléments par rapport à l'usage mémoire.

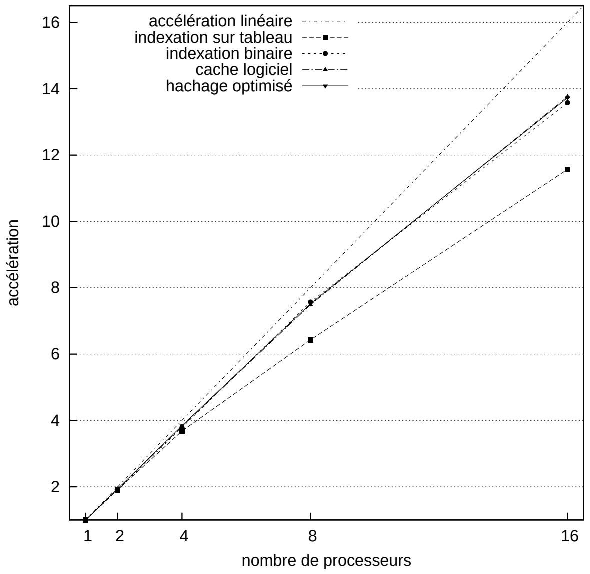

FIG. 5.4 – Impact des différentes optimisations de la structure de données sur l'accélération.

5.2.1 Réorganisation des valeurs en mémoire

Durant la phase d'évaluation, la table de hachage des noeuds ne permet pas une utilisation efficace du cache. Ceci est dû principalement à une mauvaise localité des valeurs en mémoire : la fonction de hachage pour les nœuds ne préserve la localité que selon une des dimensions du maillage.

Différentes approches peuvent être envisagées pour améliorer la localité des données. Parmi les approches classiques développées dans la littérature, on peut citer la transformation de boucles ou la réorganisation des données en mémoire. Le principe des techniques de transformations de boucles [23, 71] est de modifier l'ordre de parcours des éléments alors que celui de la réorganisation des données [25, 82], particulièrement dans les tableaux [78, 67], consiste à changer l'ordre de stockage des éléments.

Dans notre cas, le parcours des éléments du maillage adaptatif est complexe et évolue à chaque étape de temps, c'est pourquoi nous adoptons la deuxième approche.

Le principe de notre transformation est le suivant : pour éviter les défauts de cache lors d'interpolations successives de points contenus dans un groupe de cellules voisines, les valeurs aux nœuds de ces cellules doivent être contiguës ou suffisamment proches en mémoire. Ainsi le chargement d'une page en mémoire cache peut permettre un nombre important d'interpolations.

Pour réaliser cela, notre structure de données est transformée de la manière suivante : on ajoute un tableau (dit de réorganisation) dans lequel seront stockées les valeurs aux nœuds des cellules. Chaque valeur dans la table des nœuds est remplacée par un pointeur sur un élément dans ce tableau. De plus, on ajoute des pointeurs sur ces valeurs pour chaque élément dans la table des cellules. Les valeurs pointées sont les valeurs de la cellule correspondant à cet élément.

La figure 5.5 représente la structure de données une fois transformée. Le tableau de réorganisation met en relation la table des cellules et la table des nœuds. Il permet la réorganisation en mémoire des valeurs aux nœuds des cellules.

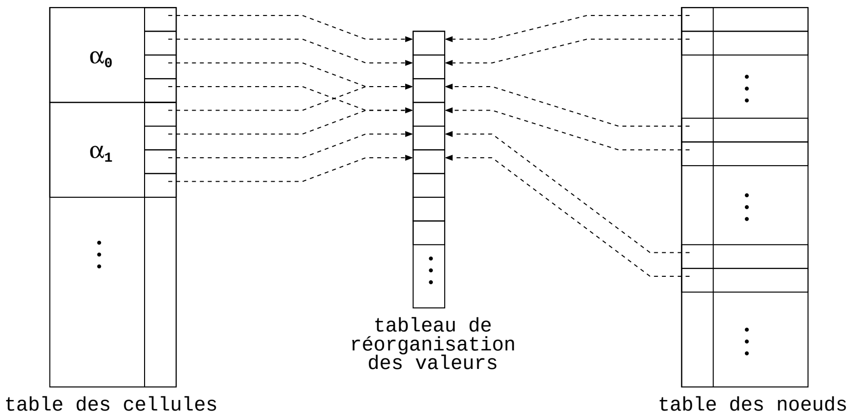

Fig. 5.5 – Réorganisation en mémoire des valeurs aux nœuds.

Notre heuristique pour approcher une localité en mémoire correcte consiste à placer les valeurs dans le tableau en respectant l'ordre introduit par le parcours des cellules qui se fait de proche en proche.

Algorithme 9 : Stockage des valeurs dans le tableau de réorganisation.

```
Données : $T_c$, la table des cellules (dont tous les pointeurs sont initialement à
          NULL), $T_n$, la table des nœuds (initialement vide) $T$, le tableau de
          réorganisation
début
    $i = 0$
    pour chaque cellule $\alpha \in T_c$ faire
        pour chaque nœud $a \in \alpha$ faire
            si $a \in T_n$ alors
                copier la valeur du pointeur $T_n(a)$ dans $T_c(\alpha).a$
            sinon
                ajouter $a$ dans $T_n$
                calculer la valeur au nœud $a$ (phase d'évaluation) et la stocker dans
                $T[i]$
                affecter l'adresse de $T[i]$ à $T_n(a)$ et $T_c(\alpha).a$
                $i++$
            fin
        fin
    fin
fin
```

L'algorithme 9 décrit le remplissage du tableau de réorganisation. La table des nœuds est utilisée pour vérifier si la valeur d'un nœud est déjà présente dans le tableau. Après le remplissage, les valeurs aux nœuds sont accédées directement en utilisant les pointeurs stockés dans la table des cellules, et la table des nœuds peut être désallouée. Étant donné que le nombre de nœuds n'est pas connu à l'avance et varie au cours de la simulation, le tableau de réorganisation des valeurs est en fait constitué de plusieurs tableaux alloués dynamiquement.

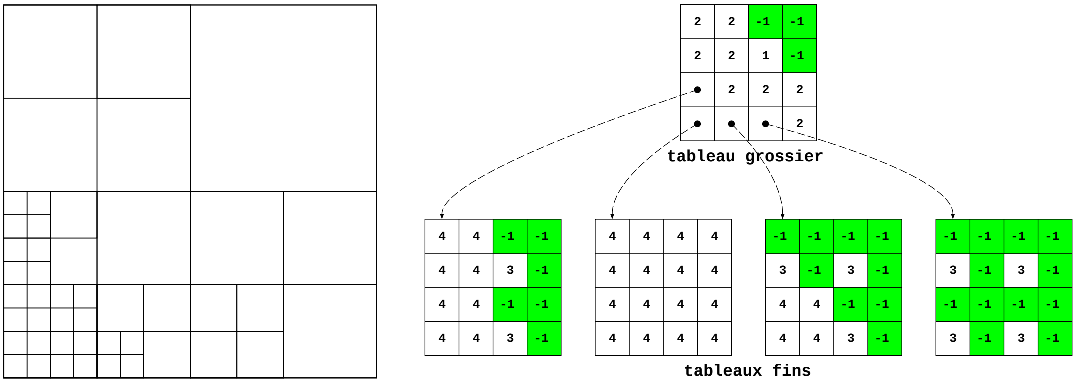

FIG. 5.6 – Maillage dyadique (à gauche) et tableau de cellules associé (à droite) : les éléments grisés sont ceux à -1.

5.2.2 Structure à base de tableaux

Les différentes optimisations de notre structure de données à base de table de hachage tendent à la rapprocher d'une structure à base de tableaux. L'avantage des tableaux est d'offrir un temps d'accès plus rapide aux éléments. Leur inconvénient repose sur leur structure uniforme qui se conforme donc mal à un maillage creux et adaptatif.

Nous pouvons cependant introduire du creux et de l'adaptativité en ajoutant un niveau d'indirection avec un tableau de pointeurs. Ceci permet de réduire la consommation mémoire dans les zones uniformément creuses du maillage. Notre structure de données est fondée sur ce principe. Il s'agit d'un cas particulier de *tableau multi-niveaux*.

Un tableau multi-niveaux [95] est une structure de données composée de tableaux, qui peut être définie récursivement de la manière suivante : un tableau multi-niveaux est un tableau où chaque élément est soit une valeur, soit un tableau multi-niveaux. Afin de réduire au maximum le temps d'accès aux éléments, notre structure est un tableau à deux niveaux seulement. De cette façon, l'accès à un élément ne requiert au maximum qu'une seule indirection et son coût est donc très faible.

Le principe de notre représentation est le suivant : on se donne un niveau intermédiaire de cellules l_c, avec $l_0 < l_c < L$. Le niveau l_c détermine à quel niveau de la structure les éléments du maillage sont stockés. Ainsi, pour une cellule du maillage de niveau l, si $l \leq l_c$, alors les informations relatives à cette cellule et à ses noeuds sont stockées dans le tableau au premier niveau, et si $l > l_c$, alors ces informations sont stockées dans un tableau au deuxième niveau.

La figure 5.6 illustre ce principe dans le cas 2D : elle montre un maillage dyadique (à gauche) et sa représentation par un tableau à deux niveaux (à droite). Dans cet exemple, $l_0 = 1$, $L = 4$ et $l_c = 2$. Pour simplifier la figure, on utilise des tableaux 2D et on ne représente que les informations relatives aux cellules. Au premier niveau, un tableau d'indirection dont chaque élément est soit est un niveau de cellule, soit un pointeur sur un tableau au deuxième niveau. Un -1 dans un tableau signifie que la cellule correspondante n'existe pas dans le maillage.

Nous décrivons maintenant en détails chacun des deux niveaux de notre structure à base de tableaux :

Premier niveau. Le premier niveau est composé de deux tableaux uni-dimensionnels permettant de stocker respectivement les informations sur les cellules (de niveau $l \leq l_c$) et les informations sur les nœuds associés à ces cellules.

Le tableau qui contient les informations sur les cellules, que nous appellerons *tableau des cellules grossières* (puisqu'il représente des cellules de niveau grossier), est un tableau de taille $(2^{l_c})^d$. Chaque élément de ce tableau contient un couple (l, r) de deux entiers et identifie une zone du maillage correspondant à une cellule de niveau l_c. Soit $alpha$, cette cellule. Si $l = -1$, alors α n'existe pas dans le maillage (elle est recouverte par une cellule plus grossière). Sinon, l est le niveau d'une cellule présente dans le maillage. Cette cellule est soit α, si $l = l_c$, soit la cellule ancêtre de α et de niveau l, si $l < l_c$. L'entier r n'est significatif que si $l \neq -1$ et cet entier est alors le rang du processeur qui détient la cellule de niveau l.

Le tableau qui contient les informations sur les nœuds est un tableau de taille $(2^{l_c+1} + 1)^d$. Chaque élément de ce tableau identifie un noeud d'une cellule de niveau l_c et est un réel dont la valeur est soit indéfinie, si le nœud n'existe pas dans le maillage, soit la valeur de ce nœud (c'est-à-dire la valeur approchée de la solution au point de l'espace des phases correspondant à ce nœud), si le nœud existe dans le maillage.

Tableau d'indirection. Chaque élément du tableau des cellules grossières contient également deux pointeurs qui réalisent un niveau d'indirection. Pour chacun de ces éléments, s'il existe des cellules du maillage de niveau $l > l_c$ à l'intérieur de la zone correspondante, alors le couple (l, r) n'est pas significatif et les deux pointeurs sont non nuls et pointent sur deux tableaux au deuxième niveau. Ces deux tableaux représentent la zone du maillage correspondant à l'élément du tableau des cellules grossières.

Deuxième niveau. Au deuxième niveau, une zone du maillage correspondant à une cellule de niveau l_c est représentée par deux tableaux uni-dimensionnels permettant de stocker respectivement les informations sur les cellules (de niveau $l > l_c$) et les informations sur les nœuds associés à ces cellules.

Le tableau qui contient les informations sur les cellules, que nous appellerons *tableau des cellules fines* (puisqu'il représente des cellules de niveau fin), est un tableau de taille $2^{L-l_c)^d}$. Comme pour le tableau des cellules grossières, chaque élément de ce tableau est un couple (l, r) de deux entiers et identifie une zone du maillage correspondant à une cellule de niveau L.

Le tableau qui contient les informations sur les nœuds est un tableau de taille $(2^{L-l_c+1} + 1)^d$. Comme pour le premier niveau, chaque élément de ce tableau identifie un noeud d'une cellule de niveau L et est un réel dont la valeur est soit indéfinie, si le nœud n'existe pas dans le maillage, soit la valeur de ce nœud (c'est-à-dire la valeur approchée de la solution au point de l'espace des phases correspondant à ce nœud), si le nœud existe dans le maillage.

5.2.3 Résultats

Nous comparons ici deux versions du code : avec la structure à base de tables de hachage (code 1) et avec la structure à base de tableaux (code 2).

Ces expériences ont eu lieu sur le cluster HP du CECPV pour un cas test jouet qui consiste à effectuer la rotation d'une Gaussienne 4D dans l'espace des phases pendant 100

tables de hachage	version témoin	avec accès directs et réorganisation
temps d'exécution (s)	1506.3	453.79
usage memoire (Ko)	1002240	2033600

TAB. 5.3 – Temps d'exécution et usage mémoire pour le code 1.

étapes de temps. Le niveau de raffinement maximum est $L = 5$ ce qui équivaut à une résolution maximum de 64 points par dimension.

Nous avons mesuré le temps d'exécution et la quantité de mémoire utilisée pour :

1. le code 1 avec ou sans les optimisations concernant les accès directs et la réorganisation (cf. tableau 5.3)
2. le code 2, sans optimisations, avec accès directs uniquement, avec accès directs et réorganisation, et avec différentes valeurs du niveau l_c.

La table 5.3 montre que le recours à des accès directs aux valeurs et le stockage de ces valeurs dans des tableaux de réorganisation en mémoire permet de diviser le temps de la simulation par 3. Ce gain très important est dû à une meilleure utilisation du cache et à l'économie du coût de la fonction de hachage pendant la phase d'évaluation. En effet, la table des nœuds n'est plus accédée que $2^d\mathcal{C}^{n+1}$ fois, alors qu'auparavant elle l'était $(2^d\mathcal{C}^{n+1})2^d$ fois, où $\mathcal{C}^{n+1}$ est le nombre de cellules du maillage prédit. En contrepartie, la version optimisée consomme deux fois plus de mémoire que la version de base. Ce surcoût est dû à l'augmentation de la taille de la table des cellules à cause du stockage pour chaque élément, des pointeurs sur les valeurs des nœuds de la cellule correspondante.

Le tableau 5.4 montre que la structure à base de tableaux est plus performante que la structure à base de tables de hachage, même si cette dernière a tendance à s'en rapprocher grâce aux différentes optimisations mises en oeuvre. Cette différence peut avoir plusieurs causes, notamment le coût de l'accès à un élément (cellule ou nœud), le coût de l'allocation d'un nouvel élément ou encore le nombre d'opérations élémentaires nécessaires au parcours de l'ensemble de la structure. On note également que pour le code 2, la version avec accès direct aux valeurs semble plus performante que la version avec réorganisation des éléments en mémoire. Cela montre que le surcoût de cette réorganisation est plus important que le gain qu'elle engendre. Les tableaux à deux niveaux préservent suffisamment la localité des données.

En ce qui concerne le niveau d'indirection dans la structure à base de tableaux, les meilleures performances sont obtenues pour un niveau $l_c = 3$. En effet, pour un niveau plus grossier, la taille des tableaux du deuxième niveau est plus importante, ce qui affecte négativement la localité des données en mémoire et l'utilisation du cache. Pour un niveau plus fin, les tableaux du deuxième niveau sont plus petits et beaucoup plus nombreux, ce qui implique une augmentation importante du nombre d'allocations et désallocations dynamiques.

tableaux à 2 niveaux	l_c	sans optimisations	accès directs et réorganisation mémoire	accès directs uniquement
temps d'exécution (s)	2	411.15	372.44	328.9
	3	403.15	314.18	262.97
	4	426.76	334.74	290.05
usage mémoire (Ko)	2	334560	502320	362624
	3	248000	1060784	921088
	4	580688	1474784	1335088

Tab. 5.4 – Temps d'exécution et usage mémoire pour le code 2.

Troisième partie

Schéma adaptatif avec prédiction du maillage *en arrière*

Chapitre 6

Une méthode numérique basée sur la prédiction en arrière

Dans le précédent chapitre, nous avons vu que le nombre d'accès aux éléments du maillage augmente exponentiellement avec le nombre de dimensions. Ces accès peuvent faire baisser de manière importante les performances du code si la structure de données n'est pas adéquate. Deux structures de données ont été implantées et des optimisations y ont été ajoutées pour en améliorer les performances. Une autre approche, qui est complémentaire, est de chercher à modifier la méthode numérique de sorte que l'algorithme de résolution provoque moins d'accès à ces structures.

La méthode numérique, présentée au chapitre 3, est basée sur un schéma de prédiction en avant, qui implique un grand nombre d'accès aux éléments du maillage. En effet, l'algorithme de résolution se compose de 4 phases, et chacune d'elles nécessite un parcours de l'ensemble du maillage. De plus, l'algorithme parallèle implique une barrière de synchronisation supplémentaire (en plus de celle dûe au calcul du champ E) : les processus doivent attendre que la phase de prédiction soit complètement terminée avant de commencer la phase d'évaluation. Dans le cas de l'utilisation du splitting, un pas de temps est décomposé en plusieurs étapes sur des pas de temps intermédiaires. Le nombre de parcours du maillage et de barrières de synchronisation est d'autant plus important.

Ce chapitre présente une nouvelle méthode basée sur un schéma de prédiction en arrière qui remplace avantageusement le schéma de prédiction en avant. La première section présente le principe de base du nouveau schéma de prédiction puis le décrit en détails. La deuxième section discute des modifications à apporter à la méthode numérique pour utiliser efficacement le schéma de prédiction en arrière. La section suivante propose des optimisations qui visent à améliorer les algorithmes récursifs que nous avons introduits.

6.1 Prédiction en arrière

6.1.1 Principe de base

Le principe de base du schéma de prédiction du maillage en arrière [19] est de construire le maillage en effectuant des raffinements successifs de cellules, à partir de la grille uniforme de niveau l_0. Un intérêt majeur de ce schéma en arrière, par rapport au schéma en avant, est que l'ensemble des cellules prédites est consistant par construction : l'opération de raffinement d'une cellule utilisée dans ce schéma induit automatiquement la consistance du maillage. L'opération d'insertion d'une cellule dans le cas du schéma en avant est plus coûteuse car elle doit gérer la consistance.

6.1.2 Schéma de prédiction

Nous détaillons ici le schéma de prédiction en arrière. Initialement, le maillage prédit $\tilde{\mathcal{M}}^{n+1}$ est la grille uniforme de niveau l_0. Les cellules du maillage sont ensuite raffinées jusqu'à ce qu'elles vérifient la *condition d'arrêt*. Cette condition d'arrêt est propre à chaque cellule. Elle porte sur le niveau de la cellule. Cette condition s'exprime de la manière suivante : soit α, une cellule et l_α, son niveau. Soit c_α, le centre de α et $\mathcal{A}^-(c_\alpha)$, le point issu du transport en arrière de c_α. Soit β, l'unique cellule de $\tilde{\mathcal{M}}^n$ qui contient le point $\mathcal{A}^-(c_\alpha)$ et l_β, le niveau de β. Alors, la condition d'arrêt est $l_\alpha > l_\beta \vee l_\alpha = L$.

6.1.3 Propriété

La prédiction en arrière offre une meilleure adaptation du maillage que la prédiction en avant. Pour s'en rendre compte prenons un maillage uniforme de niveau $l < L$ à l'étape de temps t_n. Le maillage prédit par la prédiction en arrière à l'étape de temps t_{n+1} sera lui aussi uniforme (de niveau $l + 1$). Cette bonne propriété n'est pas vraie pour la prédiction en avant. En effet, l'opération d'insertion d'une cellule (en préservant la consistance du maillage) peut mener à un maillage non uniforme. Le lecteur peut consulter [21] pour une analyse fine des propriétés d'un schéma de prédiction en arrière. Le schéma de prédiction en arrière apparaît donc comme "naturel" du point de vue de l'adaptation du maillage.

6.2 Modification de la méthode

6.2.1 Intégration du nouveau schéma de prédiction

Les phases de calcul du champ électrique, d'évaluation et de compression peuvent se dérouler de la même façon qu'avec la prédiction en avant. Cependant, il est intéressant de constater que chaque phase peut être réalisée en utilisant un même parcours en *profondeur d'abord* de l'arbre représentant la hiérachie des cellules. Ce que nous montrons ci-dessous :

Phase 1 (calcul du champ). Le calcul de la densité de charge ρ peut être réalisé en parcourant l'arbre en profondeur d'abord et pour chaque feuille rencontrée (correspondant à une cellule présente dans le maillage), en ajoutant à ρ la contribution de cette cellule.

Phase 2 (prédiction). Le maillage prédit est l'ensemble des feuilles de l'arbre construit en partant des cellules de niveau l_0 et en développant chaque branche selon un parcours en profondeur d'abord jusqu'à ce que la condition d'arrêt soit vérifiée.

Phase 3 (évaluation). Le parcours des noeuds du maillage prédit peut s'effectuer en parcourant l'arbre en profondeur d'abord et pour chaque feuille rencontrée (correspondant à une cellule), en parcourant chacun des noeuds de cette cellule.

Phase 4 (compression). La recherche de groupes de cellules soeurs s'effectue naturellement en suivant le parcours de l'arbre en profondeur d'abord. Le déraffinement consiste à élaguer une branche qui ne contient que des feuilles.

6.2.2 Algorithmes récursifs

Afin de préciser notre algorithme de résolution, nous cherchons à exprimer de façon plus formelle les algorithmes qui sont donnés informellement dans la section précédente.Étant donné que ces algorithmes effectuent un parcours du maillage en profondeur d'abord, ils sont définis de façon naturelle par une fonction récursive (voir l'algorithme 10 pour la prédiction et l'algorithme 11 pour la compression).

Algorithme 10 : Prédiction récursive du maillage.

```
Entrées : $\mathcal{M}^n$
Sorties : $\tilde{\mathcal{M}}^{n+1}$ initialement vide
pour chaque $\alpha_0$ de niveau $l_0$ faire
    initialiser $\alpha$ à $\alpha_0$
    prédiction($\alpha$) : :
    début
        soit $l_\alpha$ le niveau de $\alpha$
        calculer $c$ le centre de $\alpha$
        calculer $\tilde{c} = \mathcal{A}^-(c)$
        déterminer la cellule $\beta \in \mathcal{M}^n$ telle que $\tilde{c} \in \beta$
        soit $l_\beta$ le niveau de $\beta$
        si $\min(L-1, l_\beta) \geq l_\alpha$ alors
            pour chaque cellule $\alpha_k, k \in 0..(2^d-1)$ fille de $\alpha$ faire
                appeler récursivement prédiction($\alpha_k$)
            fin
        sinon
            ajouter $\alpha$ à $\tilde{\mathcal{M}}^{n+1}$
        fin
    fin
fin
```

6.3 Optimisation des algorithmes

6.3.1 Réduction du nombre d'accès

La section précédente montre qu'il est possible de combiner les phases de prédiction, d'évaluation et de compression et de réaliser une advection en un seul parcours du maillage. Ceci réduit considérablement les accès aux éléments du maillage et accroît la localité des données puisque les différents traitements s'enchaînent et s'effectuent localement sur une même cellule du maillage. En combinant les algorithmes récursifs précédents, nous obtenons l'algorithme récursif 12 qui réalise l'advection en une passe.

L'algorithme détermine (par effet de bord) la représentation $(\mathcal{M}^n, \mathcal{F}^n)$ de la solution à t_{n+1} à partir de la représentation $(\mathcal{M}^{n+1}, \mathcal{F}^{n+1})$ de la solution à t_n. C'est une fonction récursive qui, étant donnée une cellule (initialement une cellule de niveau l_0), retourne un booléen indiquant si la cellule existe dans le maillage au temps t_{n+1}, et les valeurs en ses nœuds, si la cellule est présente dans le maillage.

Algorithme 11 : Compression récursive du maillage.

```
Entrées : $\tilde{\mathcal{M}}^{n+1}$,$\tilde{\mathcal{F}}^{n+1}$
Sorties : $\mathcal{M}^{n+1}$,$\mathcal{F}^{n+1}$
pour chaque cellule $\alpha_0$ de niveau $l_0$ faire
    initialiser $\alpha$ à $\alpha_0$
    compression($\alpha$) : :
    début
        si $\alpha \notin \tilde{\mathcal{M}}^{n+1}$ alors
            $b := vrai$
            pour chaque cellule $\alpha_k, k \in 0..(2^d - 1)$ fille de $\alpha$ faire
                appeler récursivement compression($\alpha_k$)
                $b := b \wedge (\alpha_k \in \tilde{\mathcal{M}}^{n+1})$
            fin
            si $b \wedge$ testCompression($\alpha$) alors
                ajouter $\alpha$ à $\tilde{\mathcal{M}}^{n+1}$
            fin
        fin
    fin
fin
```

6.3.2 Dérécursivation

Les algorithmes récursifs sont souvent plus concis et élégants que les algorithmes itératifs. Par contre, leur implantation est la plupart du temps moins performante que l'implantation de leur version itérative. Ceci a pour cause le coût des nombreux appels récursifs et de la gestion de la pile d'appels. Pour obtenir de bonnes performances, il est nécessaire d'éliminer la récursivité de notre algorithme en une seule passe, sans pour autant modifier l'ordre de parcours en profondeur d'abord des éléments du maillage.

La dérécursivation s'applique à chaque partie de l'algorithme itératif. Elle résulte en l'algorithme 13.

Pour la partie prédiction, nous introduisons classiquement une pile (P) pour stocker les cellules du maillage prédit. Initialement, cette pile contient toutes les cellules de niveau l_0. La prédiction consiste à dépiler une cellule (α), puis à empiler les cellules issues du raffinement de α lorsque α ne vérifie pas la condition d'arrêt. Lorsque α vérifie la condition d'arrêt, l'algorithme enchaîne l'évaluation et la compression. L'algorithme termine lorsque la pile est vide, ce qui signifie que toutes les cellules ont été traitées.

Pour les parties évaluation et compression, étant donné que dans les algorithmes récursifs, le niveau de récursivité de chaque appel récursif identifie un niveau de cellule, nous stockons les données dans des tableaux d'indices $[l_0..L]$. Nous utilisons trois tableaux : $K[l_0..L]$, $B[l_0..L]$ et $V[l_0..L]$ dont chaque élément est respectivement un entier, un booléen et un tableau de 5^d valeurs réelles permettant de stocker les valeurs d'un groupe de cellules soeurs.

Le tableau V est utilisé pour la partie évaluation. L'évaluation consiste à calculer les valeurs de la cellule α par interpolation en suivant le principe semi-Lagrangien. Ces valeurs sont rangées en bonne place dans le tableau V.

Le tableau K stocke le nombre de cellules qui ont été traitées (i.e. dont les valeurs ont

Algorithme 12 : Advection récursive en une passe.

```
Entrées : $\mathcal{M}^n$,$\mathcal{F}^n$
Sorties : $\mathcal{M}^{n+1}$,$\mathcal{F}^{n+1}$
pour chaque cellule $\alpha_0$ de niveau $l_0$ faire
    initialiser $\alpha$ à $\alpha_0$
    advection($\alpha$) : :
    début
        $b := vrai$
        soit $l_\alpha$ le niveau de $\alpha$
        calculer $c$ le centre de $\alpha$
        calculer $\tilde{c} = \mathcal{A}^{-}(c)$
        déterminer la cellule $\beta \in \mathcal{M}^n$ telle que $\tilde{c} \in \beta$
        soit $l_\beta$ le niveau de $\beta$
        si $\min(L-1, l_\beta) \geq l_\alpha$ alors
            pour chaque cellule $\alpha_k, k \in 0..(2^d-1)$ fille de $\alpha$ faire
                appeler récursivement advection($\alpha_k$)
                soit $b'$ le booléen retourné par l'appel récursif
                $b := b \wedge b'$
            fin
            si $b \wedge$ testCompression($\alpha$) alors
                retourner $vrai$ et les valeurs de $\alpha$
            sinon
                pour chaque $k \in 0..(2^d-1)$ faire
                    ajouter $\alpha_k$ à $\mathcal{M}^{n+1}$
                    ajouter les valeurs de $\alpha_k$ à $\mathcal{F}^{n+1}$
                fin
                retourner $faux$
            fin
        sinon
            pour chaque nœud $a \in \alpha$ et $a \notin \tilde{\mathcal{F}}^{n+1}$ faire
                calculer $\tilde{a} = \mathcal{A}^{-}(a)$
                trouver la cellule $\beta \in \mathcal{M}^n$ telle que $\tilde{a} \in \beta$
                calculer la valeur $v$ en interpolant $\tilde{a}$ avec les valeurs aux nœuds de $\beta$
            fin
            retourner $vrai$ et les valeurs de $\alpha$
        fin
    fin
fin
```

Algorithme 13 : Advection itérative en une passe.

```
Entrées : $\mathcal{M}^n$,$\mathcal{F}^n$
Sorties : $\mathcal{M}^{n+1}$,$\mathcal{F}^{n+1}$
$K[l_0..L] := 0$
$B[l_0..L] := vrai$
pour chaque cellule $\alpha_0$ de niveau $l_0$ faire
    initialiser $\alpha$ à $\alpha_0$
    empiler $\alpha$ dans $P$
    tant que la pile $P$ n'est pas vide faire
        soit $\alpha$ le sommet de $P$
        dépiler $P$
        soit $l$ le niveau de $\alpha$
        calculer $c$ le centre de $\alpha$
        calculer $\tilde{c} = \mathcal{A}^-(c)$
        déterminer la cellule $\beta \in \mathcal{M}^n$ telle que $\tilde{c} \in \beta$
        soit $l_\beta$ le niveau de $\beta$
        si $\min(L-1, l_\beta) \geq l_\alpha$ alors
            pour chaque cellule $\alpha_k, k \in 0..(2^d-1)$ fille de $\alpha$ faire
                empiler $\alpha_k$ dans $P$
            fin
        sinon
            pour chaque nœud $a \in \alpha$ et $a \notin \tilde{\mathcal{F}}^{n+1}$ faire
                calculer $\tilde{a} = \mathcal{A}^-(a)$
                trouver la cellule $\beta \in \mathcal{M}^n$ telle que $\tilde{a} \in \beta$
                calculer la valeur $v$ en interpolant $\tilde{a}$ avec les valeurs aux nœuds de $\beta$
                ajouter $v$ dans $V[l]$ selon sa position parmi les nœuds des soeurs
            fin
            incrémenter $K[l]$
            tant que $K[l] = 2^d \wedge l > l_0$ faire
                $K[l] := 0$
                remplacer $\alpha$ par sa mère
                si $B[l] \wedge \text{testCompression}(\alpha)$ alors
                    sélectionner les valeurs de $\alpha$ dans $V[l]$ et les mettre dans $V[l-1]$
                    $l := l - 1$
                sinon
                    $B[l-1] := faux$
                    $B[l] := vrai$
                    pour chaque cellule $\alpha_k, k \in 0..(2^d-1)$ fille de $\alpha$ faire
                        ajouter $\alpha_k$ à $\mathcal{M}^{n+1}$
                    fin
                    pour chaque nœud $a_k, k \in 0..(5^d-1)$ des filles de $\alpha$ faire
                        ajouter $a_k$ et sa valeur $V[l][k]$ à $\mathcal{F}^{n+1}$
                    fin
                fin
            fin
        fin
    fin
fin
```

été calculées) à chaque niveau de la hiérarchie. L'élément $K[l]$ est incrémenté chaque fois qu'une cellule de niveau l a été traitée. Lorsque cet élément vaut 2^d, toutes les cellules soeurs ont été traitées et le test de compression peut être appliqué.

Le tableau B stocke des booléens qui indiquent l'appartenance au maillage des cellules traitées, à chaque niveau de la hiérarchie. Si le test de compression échoue au niveau l, l'élément $B[l-1]$ est mis à faux et l'élément $B[l]$ est remis à vrai en prévision du traitement de la prochaine cellule de niveau l et on réitère l'algorithme. Si le test de compression réussit au niveau l, les valeurs dans $V[l]$ de la cellule au niveau $l-1$ sont placées en bonne place dans $V[l-1]$ et on recommence la compression au niveau $l-1$.

Chapitre 7

Parallélisation de la méthode en arrière

Le chapitre précédent a présenté une modification de la méthode numérique YODA, basée sur un nouveau schéma de prédiction en arrière du maillage. Cette nouvelle méthode utilise un algorithme intrinsèquement récursif qui réalise un seul parcours de la hiérarchie des cellules du maillage pour chaque phase d'advection. Nous nous intéressons à présent à la parallélisation de cette nouvelle méthode.

Nous allons montrer dans ce chapitre que ce nouvel algorithme de résolution est mieux adapté à la parallélisation que la méthode avec prédiction en avant. La parallélisation proposée et son implantation, sont faites pour le cas d'un espace des phases à 4 dimensions et avec un splitting en temps de l'équation de Vlasov.

Nous utilisons une démarche similaire à celle du chapitre 4 pour la parallélisation de la première méthode numérique. Dans la première section, un algorithme data-parallèle est déduit de l'algorithme séquentiel et conduit à une décomposition *par bloc* du domaine de calcul. La deuxième section présente la distribution des données et l'analyse de dépendances qui nous permet de réduire la quantité de communications. La troisième section concerne l'implantation des communications engendrées par cette distribution. La structure de données utilisée pour réaliser cette distribution et les communications associées est présentée en quatrième section. La cinquième section est consacrée à l'ajout d'un mécanisme d'équilibrage dynamique de la charge pour améliorer l'efficacité de code. Enfin, les résultats du code parallèle sont exposés dans la dernière section.

7.1 Extraction du parallélisme

La méthode numérique que nous voulons paralléliser est la méthode avec prédiction en arrière, présentée au chapitre 6, et avec splitting en temps de l'équation, comme présenté au chapitre 3. Plus précisément, le splitting utilisé est 1D en position et 2D en vitesse comme le montre la figure 7.1 qui résume l'algorithme séquentiel.

D'après l'algorithme donné au chapitre 6, une advection s'effectue localement sur chaque zone de l'espace des phases définie par une cellule de niveau l_0. Dans la suite, nous appellerons *bloc*, une telle zone du domaine physique, et *bloc de données*, les données correspondant aux éléments du maillage contenus dans une cette zone.

Chaque advection consiste donc à appliquer un même traitement à chaque élément d'une collection de blocs de données. Par conséquent, notre méthode numérique définit

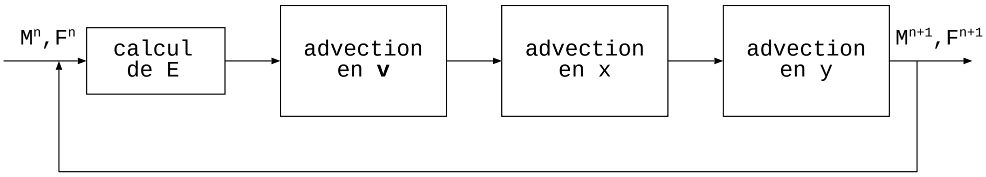

Fig. 7.1 – La boucle en temps de l'algorithme séquentiel.

un algorithme data-parallèle en considérant le maillage comme une collection de blocs de données à accès parallèle.

Cet algorithme data-parallèle explicite une décomposition en tâches parallèles. Chaque tâche parallèle (élémentaire) consiste à réaliser l'une des phases de l'algorithme séquentiel sur un bloc de données. Comme pour la méthode en avant, nous regroupons ces tâches en définissant une notion de région, de manière à obtenir des tâches plus grossières et donc mieux adaptées à une architecture à mémoire distribuée.

Le terme "région" utilisé ici a un sens différent du terme "région" utilisé pour la méthode en avant. On dira ici qu'une *région* est une zone du domaine correspondant à une union de blocs. L'ensemble des régions forme une partition du domaine et une tâche parallèle est le traitement séquentiel de tous les blocs dans une région de l'espace des phases.

7.2 Distribution des données et étude des dépendances

Notre distribution respecte naturellement le découpage en régions : si une donnée se trouve à l'intérieur (ou à la frontière) de la région affectée à un processeur, alors ce processeur détient cette donnée dans sa mémoire locale et il est responsable du traitement de cette donnée ("the owner computes").

Étant donné cette distribution, les régions et les dépendances entre les données déterminent les communications. Dans le cas de notre méthode avec splitting où les dépendances entre les données sont définies par les opérateurs d'advection (en x, en y et en (v_x, v_y)), nous pouvons éliminer des communications en choisissant des régions telles que des données qui dépendent les unes des autres se trouvent dans la même région.

Nous nous livrons maintenant à une analyse de dépendances afin de déterminer la meilleure façon de regrouper les blocs en régions afin de réduire le surcoût des communications.

7.2.1 Analyse de dépendances

Nous pouvons faire les observations suivantes :

1. Puisque les opérateurs d'advection (voir équations (1.22) et (1.22)) n'agissent que dans une certaine direction (x, y ou $\mathbf{v}$), le traitement d'un bloc ne requiert que des données de blocs situés dans l'alignement sur la même dimension. Considérons par exemple un bloc B de coordonnées (i, j, k, l) dans la grille uniforme de niveau l_0. Durant une advection en vitesse (en $\mathbf{v} = (v_x, v_y)$), le traitement du bloc B ne nécessite que des données contenues dans des blocs ayant les mêmes coordonnées (i, j) dans l'espace des positions. De manière analogue, lors d'une advection en position

($\mathbf{x} = (x, y)$), le traitement de ce bloc ne nécessite que des données contenues dans des blocs de mêmes coordonnées (k, l) dans l'espace des vitesses.

2. L'opérateur d'advection est défini à partir du champ électromagnétique E. Durant une advection en vitesse, les dépendances entre les données sont donc irrégulières et imprévisibles. Par contre, les opérateurs d'advection en position ne dépendent que des coordonnées en $\mathbf{v}$ et du pas de temps Δt. Les dépendances lors d'une advection en position sont donc linéaires et prévisibles.

Ces deux observations nous permettent de déduire la propriété suivante : Pour une région R, si tous les blocs de coordonnées en position (i, j) se trouvent dans R, alors une advection en vitesse n'engendre aucune communication pour ces blocs.

Notons que l'on peut faire la même remarque pour les blocs de mêmes coordonnées en vitesse et les advections en position. Par conséquent, on peut éliminer toutes les communications pendant les différentes phases d'advection. Pour cela, il faut que les processeurs opèrent un réarrangement global des données avant chaque advection et selon la dimension considérée lors de l'advection. Cet échange, qui correspond à une transposition de l'ensemble des données, est parfois effectué pour des codes parallèles uniformes [113]. Toutefois, la transposition d'une tel volume de données est une opération très couteuse.

Comme les phases d'advection en vitesse provoquent des dépendances irrégulières et imprévisibles, les communications associées sont difficiles à implanter. Par conséquent, nous choisissons de garder les communications lors des phases d'advection en position et d'éliminer celles des phases d'advection en vitesse.

Pour cela, nous définissons la notion de *tranche* : une tranche de coordonnées (i, j) est une zone du domaine physique correspondant à l'union de tous les blocs de coordonnées (i, j) en position. Nos régions sont donc une union de tranches. Cette restriction sur l'ensemble des régions que nous considérons, permet de simplifier la construction d'une région puisque celle-ci est désormais définie par un ensemble de coordonnées 2D. Les communications restantes sont issues de dépendances linéaires et prévisibles.

7.2.2 Communications régulières

D'après la section précédente, il n'y a que pendant les phases d'advection en x et en y que nous pouvons observer des dépendances entre les données de blocs qui appartiennent à des régions différentes. Ces dépendances sont linéaires et prévisibles. Elles sont définies par les opérateurs d'advection en position (en x et en y) et s'écrivent :

$$x \mapsto x - v_x \Delta t \quad \text{et} \quad y \mapsto y - v_y \Delta t$$

pour les advections en x et en y (respectivement).

Ainsi, lors d'une advection en x, la valeur en un point de coordonnées (x, y, v_x, v_y) de l'espace des phases dépend uniquement des valeurs aux nœuds de la cellule qui contient le point $(x - v_x \Delta t, y, v_x, v_y)$. La figure 7.2 représente géométriquement ces dépendances et leur projection sur le plan (x, v_x). Le calcul des points de coordonnées x (ligne discontinue verticale) dépend des points sur la ligne discontinue oblique. Par extension, l'advection en x de la colonne de blocs sombres dépend des valeurs contenues dans la zone hachurée. Comme cette zone est à cheval sur plusieurs blocs (coloriés en clair), il faut donc disposer de l'ensemble des valeurs de ces blocs pour effectuer le calcul.

Sur la figure 7.2, on voit que l'advection en x d'un bloc requiert des données dans trois blocs différents. Ce nombre représente le nombre maximum de blocs qu'il faudra

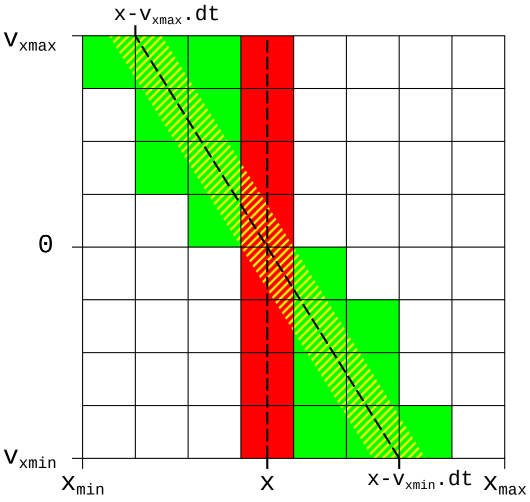

FIG. 7.2 – Dépendances lors d'une advection en x.

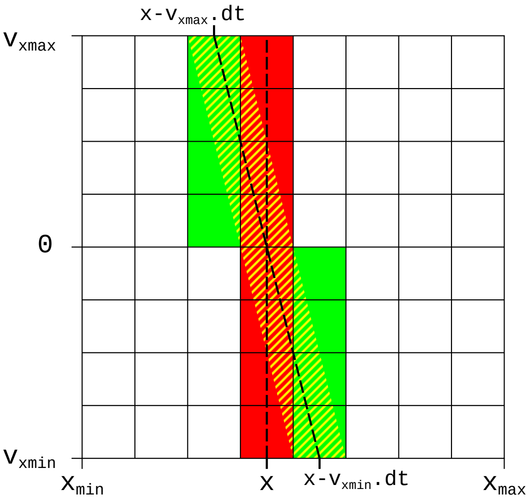

FIG. 7.3 – Dépendances lors d'une advection en x (sous certaines hypothèses).

recevoir pour pouvoir effectuer localement l'advection en x d'un bloc. En prenant quelques hypothèses sur les paramètres de la simulation, nous pouvons réduire ce nombre à un seul bloc, et ainsi diminuer considérablement le volume des communications.

Ces hypothèses concernent la taille d'un bloc, définie par $(x_{max} - x_{min})/2^{l_0}$, la vitesse maximum du domaine v_{xmax} et le pas de temps Δt. Le domaine doit être centré sur 0 en vitesse, donc $v_{xmax} = v_{xmin}$ et les paramètres doivent vérifier la relation :

$$v_{xmax}\Delta t < \frac{x_{max} - x_{min}}{2^{l_0}}, \tag{7.1}$$

Sous ces hypothèses, l'advection en x d'un bloc B ne dépend que des valeurs de B et d'un bloc voisin en x.

La figure 7.3 représente les dépendances sous ces hypothèses. Comme pour la figure 7.2, les dépendances sont projetées sur le plan (x, v_x). L'advection des blocs en zone sombre ne dépend que de leurs données et des données d'un autre bloc dont la position dépend du signe de la coordonnée v_x.

7.3 Gestion des communications

L'implantation parallèle présentée dans le chapitre 4 pour la méthode en avant, utilise un schéma *à la volée* pour gérer les communications. Ce schéma est adapté aux communications imprévisibles qui interviennent durant les différentes phases de cette méthode. Avec le nouvel algorithme parallèle et la distribution des données de la section précédente, les communications sont toutes prévisibles et régulières. Le schéma de communication *à la volée* est donc remplacé par un autre schéma de communication autorisant le recouvrement par les calculs. Ce nouveau schéma utilise la réplication des blocs de données.

7.3.1 Schéma de communication

La régularité des communications permet de déterminer exactement à quel moment chaque processeur a besoin de blocs distants de données, et quels sont ces blocs. On peut

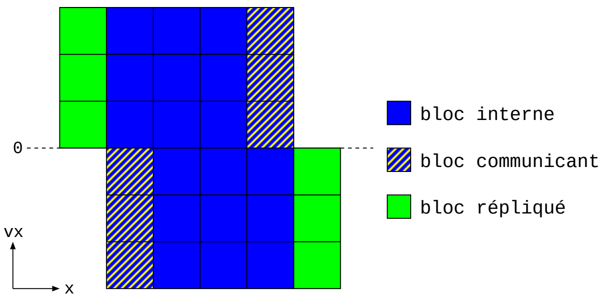

FIG. 7.4 – Les différents types de blocs dans une région.

donc rapatrier ces blocs de données *a priori*, pour que chaque processeur les possède dans sa mémoire locale au moment où il veut y accéder. Nous utilisons pour cela la réplication des blocs et la notion de *ghost cells*. Les *ghost cells* [35] représentent classiquement des zones frontières où les données sont partagées entre plusieurs processeurs pour en simplifier l'accès sur chaque processeur. Il faut alors assurer la cohérence des blocs données qui sont répliqués sur d'autres processeurs. Pour cela, les valeurs de ces blocs de données doivent être mises à jour à chaque pas de temps.

La figure 7.4 décrit, par une projection sur le plan (x, v_x), les différents types de blocs à l'intérieur de la région allouée à un processeur. Les blocs qui sont détenus par d'autres processeurs, et qui sont répliqués en mémoire locale sont appelés *blocs répliqués* (B_r). Les blocs qu'il faut envoyer, pour mettre à jour les blocs répliqués d'autres processeurs, sont appelés *blocs communicants* (B_c). Enfin, les blocs qui ne sont pas concernés par les communications sont appelés *blocs internes* (B_i). Notons que la répartition des blocs internes, communicants et répliqués dépend de la phase d'advection considérée. La figure 7.4 montre cette répartition pour la mise à jour qui précède la phase d'advection en x. On peut intervertir les dimensions (x, v_x) par (y, v_y) et obtenir la même répartition pour la mise à jour qui précède la phase d'advection en y.

La mise à jour des blocs répliqués doit se faire avant que les valeurs de ces blocs ne soient utilisées. Autrement dit, les blocs répliqués en dimension x (resp. y) doivent être transmis avant l'advection en x (resp. y). Ces phases de mise à jour ne peuvent être effectuées que lorsque les valeurs des blocs communicants ont été calculées. Par conséquent la boucle en temps est modifiée pour intégrer ces deux phases de communication, comme le montre la figure 7.5.

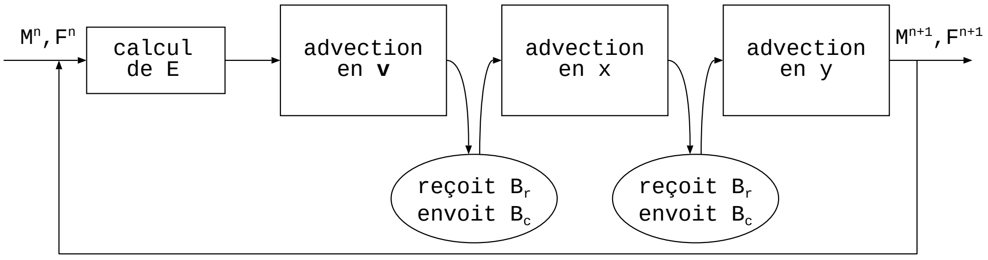

FIG. 7.5 – Ajout des communications dans la boucle en temps.

Si on attend la fin de toutes les communications au début des phases d'advections en x et en y, alors ces phases d'advections ne requièrent aucune communication. L'inconvénient est que ces mises à jour représentent alors un surcoût non négligeable. Pour en réduire l'impact sur les performances, nous recouvrons ces communications par des calculs.

7.3.2 Recouvrement des communications

Il existe plusieurs algorithmes classiques pour effectuer la mise à jour des ghost cells. Leur efficacité dépend la plupart du temps des caractéristiques de l'architecture sous-jacente, de la taille et de la dimension du problème considéré [86].

Nous avons développé un algorithme de mise à jour des blocs répliqués dont le but est de maximiser le recouvrement des communications par des calculs. Les communications effectuées lors d'une mise à jour sont recouvertes par les calculs d'une advection en $\mathbf{v}$ ou en x. Le principe de cet algorithme est d'initier dans un premier temps toutes les réceptions non-bloquantes, puis d'envoyer les blocs communicants aussitôt leur calcul terminé. De plus, l'ensemble des blocs communicants doit être traité avant les blocs internes. De cette manière, le temps de recouvrement du transfert des blocs sur le réseau est au moins égal au temps de traitement de l'ensemble des blocs internes.

L'algorithme 14 précise le déroulement de ces opérations. Il faut noter que l'attente de la fin des envois et des réceptions de messages, à la fin de l'algorithme, permet d'assurer que toutes les valeurs sont cohérentes au début de la phase d'advection qui suit. D'ailleurs, l'algorithme suppose que initialement, les données sont cohérentes : tous les blocs de données nécessaires à l'advection (les blocs de B^*) doivent être à jour.

Algorithme 14 : Mise à jour des blocs répliqués.

```
Entrées : $B^*$ : l'ensemble des blocs avant l'advection
Sorties : $B = B_r \cap B_c \cap B_i$ : l'ensemble des blocs après l'advection
début
    pour chaque bloc répliqué $b_r \in B_r$ faire
        initier la réception non-bloquante de $b_r$ venant d'un processeur voisin
    fin
    pour chaque bloc communicant $b_c \in B_c$ faire
        réaliser l'advection de $b_c$ avec les valeurs de $B^*$
        initier l'envoi de $b_c$ vers les processeurs où il est répliqué
        tester la réception des $b_r$
    fin
    pour chaque bloc interne $b_i \in B_i$ faire
        réaliser l'advection de $b_i$ avec les valeurs de $B^*$
        tester la réception des $b_r$
    fin
    attendre la fin des envois et réceptions de messages
fin
```

La figure 7.6 montre comment les phases de communication sont intégrées à la boucle en temps. Le recouvrement maximal pour la transmission des blocs répliqués est représenté en hachuré. Il représente le laps de temps pendant lequel les blocs peuvent transiter sur le réseau sans être attendus.

Enfin la figure 7.7 montre quels sont les blocs internes, communicants et répliqués pour chaque phase d'advection en les projetant sur le plan (x, y). On note que cette figure

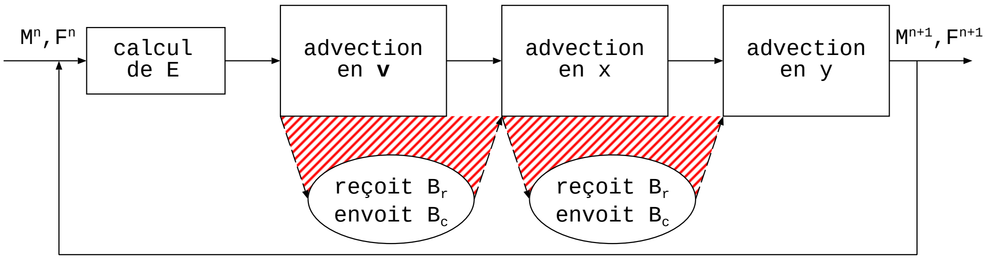

FIG. 7.6 – Recouvrement des communications dans la boucle en temps.

utilise la même nomenclature que la figure 7.4, au détail près que les blocs communicants et répliqués ne sont présents que pour les vitesses positives ou négatives selon les cas mis en évidence sur la figure 7.4.

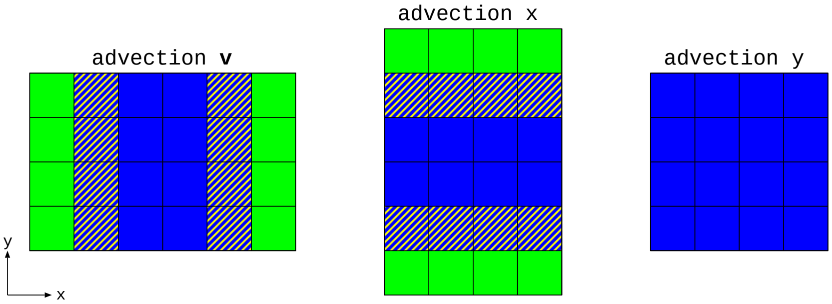

FIG. 7.7 – Les blocs d'une région et leur type à chaque phase d'advection.

7.3.3 Implantation des communications

L'implantation du schéma de communications que nous avons développé utilise des fonctions MPI de communication non-bloquante. Contrairement à l'implantation vue au chapitre 4, nous utilisons cette fois-ci le mode de communication synchrone (et non plus bufferisé) grâce à la fonction `MPI_Issend`. En effet, les blocs transmis sont cette fois de taille suffisamment importante, nous allons donc effectuer les envois bloc par bloc. De plus, les valeurs d'un bloc de données que nous allons envoyer ne sont pas modifiées avant la phase d'advection suivante, la recopie de ces valeurs dans un tampon par un envoi bufferisé n'est donc pas nécessaire et représente un surcoût que l'envoi synchrone nous permet d'éviter.

Le code source 7.1 décrit les différents appels aux fonctions MPI lors de l'implantation de l'algorithme 14. Le maillage $\mathcal{M}^n$ (resp. $\mathcal{M}^{n+1}$) est représenté par la variable `Mold` (resp. `Mnew`). Les tableaux `Br`, `Bc` et `Bi` (de tailles respectives `nBr`, `nBc` et `nBi`) contiennent respectivement les adresses des blocs répliqués, des blocs communicants et des blocs internes du maillage $\mathcal{M}^{n+1}$ pour un processeur. La fonction `proc_of` renvoie, étant donné un bloc, le rang du processeur qui effectue l'advection de ce bloc. La fonction `advection` calcule l'advection d'un bloc à l'aide des valeurs de $\mathcal{M}^n$. La fonction `assign_in_place` modifie $\mathcal{M}^{n+1}$ en positionnant dans la bonne zone, les données du bloc passé en argument. Cette fonction est appelée à chaque réception de bloc.

Comme pour le schéma de communications présenté dans le chapitre 4, l'utilisation de

Extrait de code 7.1 – Communications des blocs

```
r = 0;
for (n=0; n<nBr; n++) /* boucle sur les blocs repliques */
{
  if (Br[n] == NULL)
    alloc(Br[n]);
  MPI_Irecv(Br[n],1,MPI_BLOCK,proc_of(Br[n]),BTAG,MPICOM,&rreq[n]);
}

for (n=0; n<nBc; n++) /* boucle sur les blocs communicants */
{
  Mnew->advection(Bc[n],Mold);
  MPI_Issend(Bc[n],1,MPI_BLOCK,proc_of(Bc[n]),BTAG,MPICOM,&sreq[n]);
  if (r < nBr)
  {
    MPI_Testany(nBr, rreq, &idx, &flag, MPI_STATUS_IGNORE);
    if (idx != MPI_UNDEFINED)
    {
      Mnew->assign_in_place(Br[idx]);
      r++;
    }
  }
}

for (n=0; n<nBi; n++) /* boucle sur les blocs internes */
{
  Mnew->advection(Bi[n],Mold);
  if (r < nBr)
  {
    MPI_Testany(nBr, rreq, &idx, &flag, MPI_STATUS_IGNORE);
    if (idx != MPI_UNDEFINED)
    {
      Mnew->assign_in_place(Br[idx]);
      r++;
    }
  }
}

// attente de la fin des communications
MPI_Waitall(nBr,rreq,&stat[0]);
for (n=0; n<nBr; n++)
{
  MPI_Get_count(&stat[n],MPI_BLOCK,&count);
  if (count==1)
  {
    Mnew->assign_in_place(Br[n]);
  }
}
MPI_Waitall(nBc,&sreq[0],MPI_STATUS_IGNORE);
```

la fonction `MPI_Testsome` à la place de `MPI_Testany` peut permettre d'améliorer les performances de ce schéma.

7.4 Structure de données

Intéressons nous à présent à la structure de données à adopter pour représenter le maillage. Cette section est consacrée à l'élaboration de cette structure. Nous montrons comment les opérations de la méthode numérique peuvent être réalisées avec cette structure et comment elle supporte l'implantation des communications.

7.4.1 Stockage des nœuds uniquement

L'étude effectuée au chapitre 5, pour la méthode en avant, nous incite à considérer une structure à base de tableaux à 2 niveaux. Cependant, il y a une différence entre les deux méthodes, dont il faut tenir compte pour le choix de la structure de données : la méthode en arrière ne nécessite pas de stocker les informations sur les cellules pour les phases de prédiction et de compression. En effet, durant ces deux phases, seul un parcours en profondeur d'abord est requis. Ainsi, seule la phase d'évaluation a besoin de connaître les cellules qui existent dans le maillage. Or, cette information peut être retrouvée à partir des informations sur les nœuds. En effet, pour savoir si une cellule existe, il suffit de tester l'existence de son nœud central : ce nœud n'est partagé par aucune autre cellule de même niveau. C'est pourquoi nous utilisons une structure de données à deux niveaux qui ne stocke que les informations sur les nœuds.

Premier niveau

Le premier niveau est composé de deux tableaux unidimensionnels : un tableau de pointeurs de taille $(2^{l_0})^d$ et un tableau de réels de taille $(2^{l_0+1}+1)^d$.

Chaque élément du tableau de pointeurs correspond à une cellule de niveau l_0. Si le pointeur est nul, la cellule existe dans le maillage. Sinon, le pointeur renvoie sur un tableau au deuxième niveau qui contient l'ensemble des données relatives à la zone de l'espace des phases définie par la cellule.

Chaque élément du tableau de réels correspond à un nœud d'une cellule de niveau l_0. Si la valeur de l'élément est Nan (Not a number), alors le nœud n'existe pas dans le maillage. Sinon, le nœud existe et la valeur est celle de la solution approchée au point de l'espace des phases défini par le noeud.

Deuxième niveau

Le deuxième niveau est composé de tableaux unidimensionnels. Chacun de ces tableaux est un tableau de réels de taille $(2^{L-l_0+1}+1)^d$. Chaque élément correspond à un nœud d'une cellule de niveau L. De façon similaire au tableau de réels du premier niveau, si la valeur de l'élément est Nan (Not a number), alors le nœud n'existe pas dans le maillage. Sinon, le nœud existe et la valeur est celle de la solution approchée au point de l'espace des phases défini par le noeud.

7.4.2 Réalisation de la recherche d'une cellule

Nous montrons ici que, bien que notre structure de données ne stocke que les informations sur les nœuds, elle permet de réaliser les opérations requises pour la phase d'évaluation. En particulier, nous montrons comment peut être réalisée l'opération de recherche de la cellule contenant un point. Cette réalisation doit être particulièrement efficace car cette opération est effectuée autant de fois qu'il y a de nœuds dans le maillage.

Nous avons implanté trois algorithmes différents pour effectuer cette recherche étant donnés un point de l'espace des phases, le niveau L le plus fin et le niveau l_0 le plus grossier du maillage. Ces algorithmes prennent en compte le fait qu'un nœud, central pour un cellule de niveau l, peut être partagé par des cellules de niveau plus fin pour lesquelles il ne sera pas le nœud central. La figure 7.8 rappelle cette notion de partage de nœud.

Recherche linéaire. La première méthode consiste à parcourir les cellules qui contiennent le point à chaque niveau. Ce parcours est effectué des niveaux L à l_0 et tant que le nœud de la cellule de niveau l n'a pas de valeur (i. e est égal à Nan). Le premier nœud qui possède une valeur est le nœud central de la cellule présente dans le maillage. La recherche linéaire a donc une complexité en $\mathcal{O}(L - l_0)$.

Recherche dichotomique La seconde méthode consiste à tester la valeur du nœud central de la cellule de niveau médian $\lceil (L - l_0)/2 \rceil$ qui contient le point. Si le nœud n'a pas de valeur (il n'y a pas de cellule de niveau inférieur ou égal), alors on cherche la cellule de la même manière sur les niveaux strictement inférieurs. Sinon on effectue la recherche sur les niveaux supérieurs (niveau médian inclus). Lorsque l'intervalle de recherche est réduit à un seul niveau, ce niveau est celui de la cellule présente dans le maillage. La recherche dichotomique a une complexité en $\mathcal{O}(\log_2(L - l_0) + 1)$.

Stockage du niveau La dernière méthode consiste à utiliser un tableau additionnel pour chaque tableau de deuxième niveau. Chaque élément de ce tableau additionnel correspond à une cellule de niveau L et contient un entier qui a pour valeur le niveau de la cellule qui recouvre cette zone de l'espace des phases. La recherche par stockage du niveau a une complexité en $\mathcal{O}(1)$. L'inconvénient de cette méthode est qu'elle nécessite un surcoût en terme d'écriture de l'ordre de $\mathcal{O}(L - l + 1)^2$ pour chaque cellule de niveau l, et un surcoût en terme d'occupation mémoire.

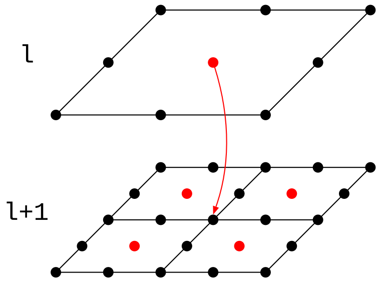

Fig. 7.8 – Partage du nœud central d'une cellule avec ses filles.

7.4.3 Extension de la structure de données pour les communications

Il faut maintenant que la structure de données soit étendue pour permettre le stockage des blocs répliqués. Pour cela, le tableau de pointeurs du premier niveau est augmenté d'un élément par dimension pour laquelle le processeur possède un voisin. De même, le tableau des valeurs au premier niveau de la structure est augmenté pour pouvoir contenir 2 nœuds supplémentaires par côté. La figure 7.9 schématise l'augmentation des régions affectées à chaque processeur.

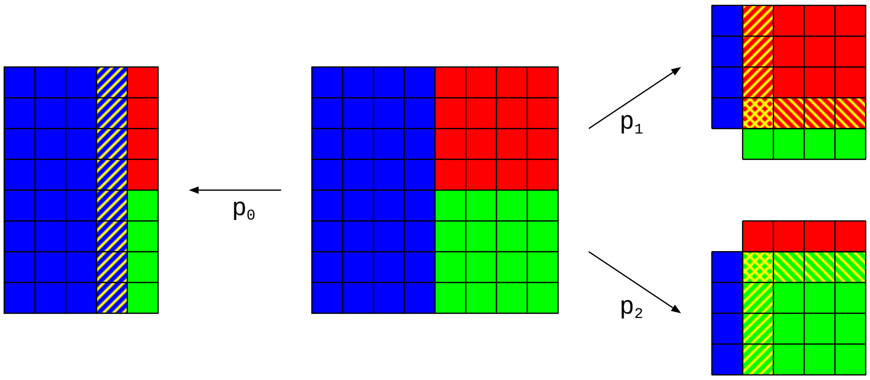

FIG. 7.9 – Augmentation des bords des régions pour permettre le stockage des blocs répliqués.

Il reste maintenant à déterminer comment envoyer les données des blocs à répliquer sur les différents processeurs. Tout d'abord, nous décidons d'envoyer le bloc en entier et tel quel, avec les valeurs de la fonction en certains nœuds mais aussi d'autres nœuds à Nan. Ce choix nous permet d'éviter les copies des données d'un bloc de la structure vers un tampon temporaire d'envoi, et les recopies d'un tampon temporaire de réception vers un bloc dans la structure. De plus, envoyer uniquement les valeurs des nœuds existants dans le maillage nécessite un parcours en entier du bloc et l'ajout pour chaque nœud d'un identifiant unique. Cet ajout peut engendrer des messages dont la taille est supérieure à celle d'un bloc entier (quand le bloc contient une grande majorité de nœuds existants).

Le premier problème à résoudre est celui de l'identification des blocs reçus. Une solution possible consiste à donner une valeur de *tag* différente aux messages MPI pour chacun des blocs. L'inconvénient de cette méthode est qu'il faut alors s'assurer que l'éventail de tags soit suffisamment grand pour permettre de différencier l'ensemble des blocs. Or, la norme MPI assure uniquement que ces tags soient des entiers positifs dont la borne supérieure (`MPI_TAG_UB`) doit être au minimum de 32767. Dans un souci de compatibilité avec différentes implantations de MPI, et comme un maillage 4D contenant 16 blocs par dimension dépasse déjà largement ce nombre, nous décidons de ne pas utiliser les tags pour différencier les messages.

Pour identifier un bloc de données, nous remplaçons les tableaux de deuxième niveau par une structure contenant les coordonnées du bloc représenté et le tableau des valeurs aux nœuds. Une réception de message peut donc être initiée avec pour tampon de réception un bloc de la structure étendue (un bloc hachuré dans la figure 7.9). Quand le message est effectivement reçu, il suffit de vérifier que l'identité du bloc correspond à l'identité de celui qui a servi de tampon de réception. Si les identités ne correspondent pas, il suffit d'inter-changer les adresses stockées dans le tableau des pointeurs au premier niveau de

Extrait de code 7.2 – Structures de données

```
typedef struct
{
 int size;
 int nbCells;
 Cell Co;
 double value[81];
}CoarseArray;

typedef struct
{
 int size;
 int nbCells;
 Cell Co;
 double value[1];
}FineArray;

// allocation d'un FineArray ...
fa = malloc(sizeof(FineArray)+SIZE_VAL*sizeof(double));
```

la structure.

Le second problème à résoudre est celui de la gestion commune des blocs qui correspondent à un tableau de deuxième niveau (que nous appelons blocs denses), et aux blocs qui représentent des cellules de niveau l_0 du maillage (que nous appelons blocs creux). Comme il est impossible de prédire quels blocs sont creux et quels blocs sont denses, il faut que les deux types de blocs soient réceptionnés de la même manière. On créé pour cela deux types de données MPI pour différencier les envois de blocs creux et les envois de blocs denses. Ces deux types ne diffèrent que sur le nombre de valeurs de nœuds qu'ils contiennent : 81 pour un bloc creux, $(2^{L-l_0+1})^4$ pour un bloc dense. La réception se fait toujours sur un bloc dense pour avoir une zone mémoire capable de stocker l'un ou l'autre type de message. Une fois le message réceptionné, le type du bloc est déterminé à l'aide de la fonction `MPI_Get_count` qui renvoie le nombre de données d'un certain type effectivement reçues. Si le bloc est creux, alors les 81 valeurs sont copiées dans le tableau du premier niveau et le bloc est désalloué.

Il faut noter que la structure contenant un bloc dense doit être allouée d'une manière spécifique pour permettre ce traitement par un type MPI. En effet, la construction d'un type MPI nécessite de fournir l'*offset* mémoire entre les différents champs de la structure. Or, si les champs qui identifient le bloc sont déclarés statiquement, le tableau des valeurs est lui déclaré dynamiquement, puisque que sa taille ne sera connue qu'à l'exécution du programme. Il se peut donc que l'offset entre le dernier champ statique et le premier élément du tableau dynamique varie d'un bloc à l'autre. Pour éviter ce phénomène, l'ensemble de la structure d'un bloc est déclarée dynamiquement comme une unique zone de mémoire contiguë, puis chaque champ est positionné de manière à respecter le même offset. Le code source 7.2 montre comment les deux structures sont définies pour que leur champ `value` soit positionné avec le même offset.

Le code source 7.3 montre comment est défini le type MPI `MPI_CA` correspondant aux blocs creux (`CoarseArray`). La définition du type `MPI_FA` pour les blocs denses, Extrait de code 7.3 – Définition de type MPI pour les blocs creux

```
CoarseArray CA;
int struct_length[4]={1,1,5,81};
MPI_Aint struct_offset[4];
MPI_Datatype struct_types[4]={MPI_INT,MPI_INT,MPI_INT,MPI_DOUBLE};
MPI_Address(&CA.size,&struct_offset[0]);
MPI_Address(&CA.nbCells,&struct_offset[1]);
MPI_Address(&CA.Co,&struct_offset[2]);
MPI_Address(&CA.value,&struct_offset[3]);
for(i=3; i>=0; i--)
  {
    struct_offset[i] -= struct_offset[0];
  }
MPI_Type_struct(4,struct_length,struct_offset,struct_types,MPI_CA);
MPI_Type_commit(MPI_CA);
```

s'effectue de la même manière, à la différence près que la taille du dernier champ correspond à $(2^{L-l_0+1})^4$.

7.5 Équilibrage de la charge

Comme le maillage adaptatif évolue au cours du temps, les régions affectées à chaque processeur doivent être mises à jour pour garantir un bon équilibre de la charge de calcul. La construction initiale des régions se base uniquement sur la valeur de la fonction pour proposer un partitionnement moins déséquilibré qu'avec un pavage régulier. Au cours de la simulation, nous avons accès à d'autres informations pour modéliser la charge de manière plus précise et proposer un mécanisme de détection du déséquilibre adapté à des architectures homogènes.

7.5.1 Création des régions initiales

La création des régions, à l'initialisation de la simulation, utilise une boîte englobante pour approcher la forme de la projection du faisceau de particules sur le plan (x, y). Cette boîte englobante peut être donnée en paramètre mais elle peut aussi être calculée automatiquement à partir de la valeur de la fonction de distribution initiale. Ce calcul correspond à celui utilisé dans certaines méthodes de *moving grid* [29] pour adapter la taille de la grille uniforme à la fonction de distribution.

Une fois obtenues taille et position de la boîte englobante, nous l'utilisons pour proposer un premier partitionnement du domaine alors que le maillage initial n'est pas encore construit. Pour cela, nous faisons l'hypothèse que la charge de calcul est concentrée uniformément sur toute la surface de cette boîte. La boîte englobante est alors subdivisée par des bissections récursives selon chaque dimension, jusqu'à l'obtention d'un nombre de partitions égale au nombre de processeurs.

Ces coupes sont alors reportées sur la grille 2D représentant l'ensemble des tranches de blocs, pour y définir un ensemble de régions rectangulaires. Chaque tranche est alors affectée à la région dans laquelle elle repose majoritairement.

7.5.2 Détection du déséquilibre

Le mécanisme de détection du déséquilibre repose sur le calcul du coût de traitement d'une région. Nous avons vu que pour être exhaustif, ce coût doit prendre en compte les opérations de calcul et les opérations de communication. Les communications étant recouvertes par les calculs, leur impact sur les performances est réduit par rapport à celui des calculs. C'est pourquoi nous avons choisi de négliger, au moins dans un premier temps, ces coûts de communication.

Dans ces conditions, la mesure de la charge de calcul que nous retenons est le temps de traitement d'un bloc. Ce temps est mesuré dynamiquement en microsecondes. Cette mesure est bien entendu uniquement adaptée aux architectures homogènes où les processeurs sont tous cadencés à la même fréquence. De plus, l'ensemble du traitement d'un bloc est effectué entièrement en local, le temps de traitement correspond uniquement à du temps de calcul et à des temps d'accès à la structure de données qui repose en mémoire locale. Nous définissons la charge d'une tranche de blocs par la somme des temps de traitement de tous les blocs qui composent cette tranche. De même, la charge de calcul associée à une région est définie par la somme des charges de calcul associées à toutes les tranches qui composent cette région.

Cette mesure est calculée à chaque pas de temps par l'ensemble des processeurs. Puis les processeurs s'échangent leur charge respective au cours d'une communication collective. Il faut noter qu'il existe déjà à chaque pas de temps une telle communication collective, lors de la phase de calcul de E. Nous pouvons donc utiliser la fonction `MPI_Allreduce` existante pour rapatrier la charge de calcul en même temps que les contributions de chaque processeur à ρ. Une fois cette charge de calcul connue pour chaque processeur, un déséquilibre est détecté si la différence entre le processeur le plus chargé et le processeur le moins chargé est supérieure à un certain seuil. Si tel est le cas, alors l'algorithme de partitionnement est exécuté pour construire des régions mieux équilibrées. Notons que d'autres mesures du déséquilibre peuvent être mises en place très facilement si elles utilisent ces mêmes informations, comme par exemple la différence avec la charge moyenne.

7.5.3 Algorithme de partitionnement

L'algorithme de partitionnement mis en œuvre doit construire des régions qui contiennent approximativement la même charge de calcul. Nous voulons également que ces régions soient de forme géométrique simple, pour que le schéma de communication soit facilement mis à jour. Enfin, il faut que le coût de cet algorithme soit le plus léger possible et que la redistribution des blocs ne génère pas un trop grand surcoût de communication.

L'heuristique que nous avons retenue est basée sur l'algorithme de bissections récursives des coordonnées (ORB) vu au chapitre 2. Cette méthode de partitionnement géométrique à l'avantage d'être rapide, et les coupes orthogonales aux axes sont bien adaptées à un maillage cartésien. De plus, elle est implicitement incrémentale, dans le sens où les régions affectées aux processeurs sont toujours approximativement dans la même zone du domaine de calcul. Cette dernière propriété permet généralement, mais sans le garantir, de réduire la quantité de données à transférer lors de l'équilibrage de charge.

On considère une grille uniforme 2D de l'espace des positions. Chaque cellule de cette grille, de coordonnées (i, j), représente une tranche de blocs dans l'espace des phases. Par conséquent, une région est définie par un ensemble de ces cellules. Ces ensembles sont formés par une approche classique "diviser pour régner". La grille de départ est coupée perpendiculairement à la plus grande dimension. La coupe génère deux sous-ensembles de tranches tels que la charge associée à chaque sous-ensemble est équivalente. Et comme nous ne voulons pas que les valeurs d'un même bloc soient calculées par plusieurs processeurs, la coupe doit être effectuée uniquement entre des cellules de la grille 2D, aucune cellule ne doit être coupée. Puis, les coupes sont effectuées récursivement sur chaque nouvel ensemble ainsi construit, jusqu'à obtenir un nombre de régions égal au nombre de processeurs. Il faut noter que ce nombre est pour le moment une puissance de 2 mais la technique est généralisable pour un nombre quelconque de processeurs.

Si nous effectuons uniquement des coupes rectilignes, la charge de calcul associée aux deux sous-ensembles générés peut être assez différente. En effet, comme nous ne voulons pas subdiviser un bloc (et par extension une tranche de blocs), une bissection qui coupe une cellule ne permet pas de répartir équitablement la charge. Pour améliorer le résultat de l'algorithme de partitionnement, nous autorisons des coupes présentant un *zigzag* à la place d'une coupe rectiligne, dans le cas où cette dernière passe à travers des cellules.

Cette optimisation rapproche notre méthode de partitionnement de la méthode ORB-MM présentée en section 2.3.1. Cette modification est schématisée par la figure 7.10. À gauche, la ligne discontinue, qui représente la division de la charge globale en deux moitiés égales, passe entre deux colonnes de cellules. Les régions sont donc créées avec une coupe rectiligne qui permet de répartir la charge de manière exacte. À droite, la ligne discontinue passe à travers une colonne de cellules. Par conséquent, on utilise une coupe avec un zigzag pour construire les régions. Le segment perpendiculaire est choisi entre deux lignes de cellules de telle manière à ce que la différence de charge entre les deux régions soit minimale.

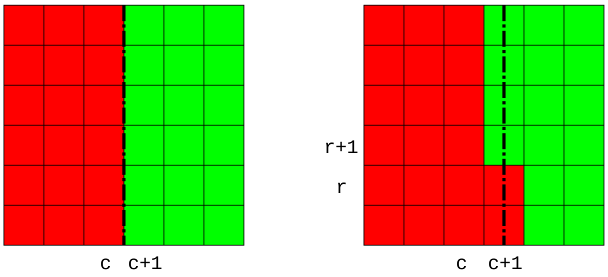

FIG. 7.10 – Coupe rectiligne et coupe avec *zigzag* d'un ensemble de tranches.

Notre variante de ORB est donnée plus en détails par l'algorithme 15. Dans cet algorithme, la variable *rank* correspond au rang du processeur associé à la région courante désignée par *part*. La variable *nbis* représente le nombre de bissections déjà effectuées, et *nproc* est le nombre de processeurs (donc le nombre de régions à construire). Il peut arriver que l'algorithme de partionnement génère des régions vides ou qui ne font pas diminuer le déséquilibre de charge entre les processeurs. Dans ces cas-là, les nouvelles régions ne sont pas validées et la simulation continue avec la même distribution des données. On note que le zigzag est toujours effectué dans le même sens, pour simplifier la description des régions générées à l'aide d'une union de sous-ensembles rectangulaires. Ainsi, la structure de données présentée en section 7.4 reste identique, mais chaque processeur peut gérer à présent un ensemble de régions rectangulaires.

Algorithme 15 : Algorithme récursif de bissection d'un ensemble tranches.

```
Données : à l'initialisation, $rank = 0$, $part$ est le domaine 2D en entier, et $nbis = 0$
bissection($rank, part, nbis$) : :
début
    si $2^{nbis} = nproc$ alors
        affecter la région décrite par $part$ au processeur $rank$
    sinon
        calculer la hauteur $H$ et largeur $W$ de la région décrite par $part$
        si $H > W$ alors
            intervertir les dimensions $x$ et $y$ dans le suite de l'algorithme
        fin
        soit $w(r, c)$ la charge associée à la cellule $(r, c) \in part$ (sinon $w(r, c) = 0$)
        calculer la coût de chaque colonne $c$ de $part$ par $w(c) = \sum_r w(r, c)$
        calculer la somme $sum$ de la charge contenue dans $part$
        trouver la colonne $c = \max\{k | S_k \leq sum/2\}$ avec $S_k = \sum_{j \leq k} w(j)$
        si $S_c = sum/2$ alors
            la coupe passe entre les colonnes $c$ et $c + 1$
        sinon
            la coupe passe à travers la colonne $c + 1$
            trouver la ligne $r$ qui minimise $|sum/2 - (S_c + s_r)|$ avec $s_r = \sum_{i \leq r} w(i, c)$
            la coupe perpendiculaire passe entre les lignes $r$ et $r + 1$
        fin
        soit $part_1$ et $part_2$ les sous-ensembles de $part$ définis par la bissection
        appeler récursivement bissection($rank, part_1, nbis + 1$)
        appeler récursivement bissection($rank + nproc/2^{nbis+1}, part_2, nbis + 1$)
    fin
fin
```

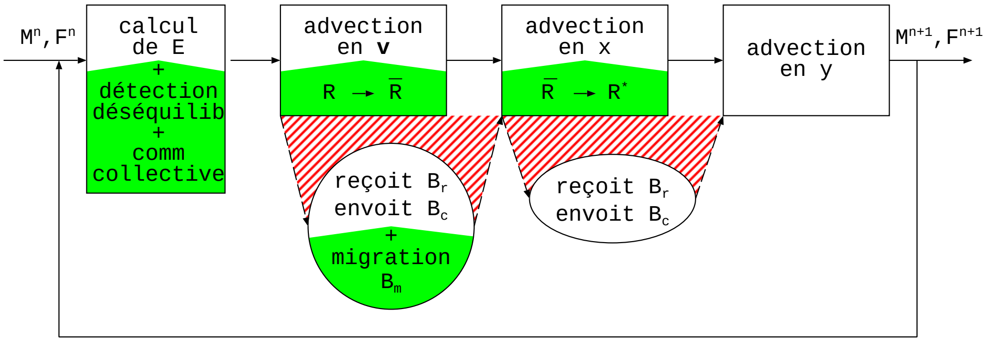

FIG. 7.11 – Ajout de l'équilibrage de charge dans la boucle en temps.

7.5.4 Intégration dans l'algorithme

L'affectation des nouvelles régions aux processeurs implique la communication de blocs de données qui n'appartiennent plus à la même région. Il ne s'agit pas de réplication des blocs mais bien d'un changement de propriétaire du bloc : on parle alors de migration de blocs. Il en découle un surcoût de communication qui peut ralentir le programme de manière significative et par conséquent gaspiller le gain de performance obtenu par un meilleur équilibrage de charge. Pour réduire ce surcoût, la redistribution des blocs de données et les autres mécanismes de l'équilibrage dynamique sont intégrés aux différentes phases qui composent une étape de temps de la simulation. Cette intégration nous permet, entre autre, de recouvrir le surcoût de communication par des calculs. Ce recouvrement n'est pas faisable si l'équilibrage de charge et la redistribution des données sont exécutés dans une phase séparée.

Lorsqu'un déséquilibrage de charge a lieu, la boucle en temps donnée dans la figure 7.5 est légèrement modifiée pour aboutir à celle exposée par la figure 7.11. L'équilibrage de charge ajoute des opérations dans les phases de calcul de E, d'advection en $\mathbf{v}$, et modifie légèrement les phases d'advection en $\mathbf{v}$ et en x. Décrivons en détails chacune de ces modifications.

Tout d'abord, la détection du déséquilibre a lieu au cours de la phase de calcul du champ électrique E, puisque la charge de calcul de chaque processeur est transmise dans le même message collectif que les contributions à ρ. Si un déséquilibre est détecté, alors une deuxième communication collective permet de rapatrier localement la charge de chaque bloc. Cette communication est peu coûteuse puisque les processeurs sont déjà synchronisés et que l'information à transmettre n'est pas volumineuse. Ensuite, chaque processeur exécute l'algorithme de partitionnement. Si les nouvelles régions créées permettent de réduire le déséquilibre, alors la migration des blocs est planifiée, sinon la boucle en temps reprend son cours normal.

Pour un processeur p, posons R son ancienne région et R^* la nouvelle région qui lui est affectée. Enfin notons $\bar{R}$ la région temporaire définie par $\bar{R} = R \cup R^*$. La figure 7.12 montre les différents types de blocs à considérer lorsqu'on passe de R à R^*. Le contour de la région R est représenté par un trait gras en pointillés, alors que celui de R^* l'est par un trait gras continu. Parmi les blocs appartenant à l'intersection $R \cap R^*$, les blocs internes sont notés B_i et les blocs communicants sont notés B_c. Les blocs appartenant à $R - R^*$ sont des blocs de migration, notés B_m, à transmettre à d'autres processeurs. Les blocs appartenant à $R^* - R$ sont des blocs de migration, notés B_m^*, à recevoir d'autres

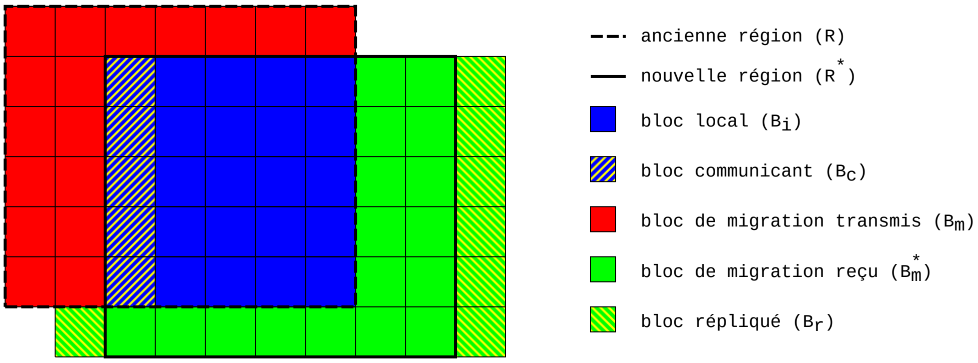

FIG. 7.12 – Différents cas lors de la migration des blocs.

processeurs. Enfin, les blocs hachurés et notés B_r sont des blocs répliqués en x de R^* nécessaires pour la phase d'advection en x.

Au cours de l'advection en $\mathbf{v}$, le processeur p réalise l'advection de tous les blocs de R, et envoie les blocs B_m et B_c aux processeurs qui en ont besoin. Contrairement aux phases de communications classiques, un même bloc peut être envoyé à deux processeurs différents : le processeur p_1 qui possède le bloc dans sa nouvelle région et le processeur p_2 qui a besoin de la réplication de ce bloc (si le bloc est communicant pour p_1). Le processeur p reçoit durant cette phase les blocs B_m^* et les blocs B_r en x, qui ne sont pas issus de R. Pour permettre le stockage des blocs calculés localement qui appartiennent à R, et des blocs reçus qui appartiennent à R^*, une nouvelle structure temporaire est allouée pour contenir l'ensemble des blocs de $\bar{R}$. Le stockage des blocs de $R - R^*$ est nécessaire car les communications non bloquantes ne permettent pas de désallouer ces blocs avant la fin de la phase de communication.

Enfin, la phase d'advection en x est effectuée normalement à l'exception des lectures des données nécessaires aux calculs d'un bloc qui sont faites dans la structure temporaire représentant $\bar{R}$. À la fin de cette phase, cette structure temporaire peut être désallouée et la boucle en temps reprend son cours normal.

7.6 Résultats

Dans cette section, nous allons dans un premier temps nous intéresser aux performances liées à la structure de données à deux niveaux que nous avons implantée. Nous réalisons d'abord une expérience pour mettre en évidence les performances de la recherche de la cellule qui contient un point de l'espace des phases. Puis nous nous intéressons au choix du paramètre l_0 et à son influence sur la quantité de calculs et l'occupation mémoire de la structure de données. Enfin nous montrons que le code parallèle 4D permet d'obtenir une bonne efficacité sur une machine à mémoire distribuée.

7.6.1 Performances de l'opération de recherche d'une cellule

Les trois méthodes de recherche d'une cellule dans la structure de données ont été implantées dans un code de test en 2D. Les performances obtenues correspondent à une simulation 2D du cas test (voir B.1) pour 500 itérations et avec un niveau de raffinement

$L = 10$ (soit 2048 points par dimension). La simulation est exécutée sur un simple PC linux (voir C).

Le tableau 7.1 montre l'impact de la méthode de recherche des cellules dans la structure à deux niveaux pour différents niveaux d'indirection l_0. Les résultats montrent que la recherche linéaire est l'algorithme qui donne les meilleures performances, ce qui est plutôt inattendu. Le fait que l'algorithme de stockage des niveaux soit légèrement moins performant que l'algorithme de recherche linéaire peut s'expliquer par le surcoût de l'écriture des niveaux, qui est supérieur au gain de la recherche de la cellule en $\mathcal{O}(1)$. Par contre, on aurait pu penser que le gain en nombre d'accès de la recherche dichotomique était supérieur au faible surcoût du calcul du niveau médian.

niveau grossier (l_0)	3	4	5	6	7	8
recherche linéaire	407	353	335	342	376	495
recherche dichotomique	438	387	369	374	402	529
stockage du niveau	409	356	344	346	387	510

Tab. 7.1 – Temps d'exécution (s) selon l'algorithme de recherche des cellules.

La tableau 7.2 montre qu'en réalité, l'algorithme de recherche dichotomique provoque plus d'accès à la structure de données que l'algorithme par recherche linéaire. Ce phénomène peut être expliqué en deux temps, en prenant $l_0 + 2 \leq L$.

Pour la recherche linéaire, le meilleur des cas correspond au cas d'une cellule de niveau L, trouvée en un unique accès à la structure. Pour la recherche dichotomique, le meilleur des cas correspond à une cellule l_0 pour $L = l_0 + 2$, la cellule est alors trouvée en un unique accès. Pour tous les autres cas, notamment pour une cellule de niveau L et quel que soit l_0, la recherche dichotomique nécessite au minimum deux accès pour trouver une cellule. Or, il y a beaucoup plus de cellules de niveaux fins que de cellules de niveaux grossiers dans le maillage (simplement du fait de leur taille). Par conséquent, il y a plus de cas favorables à la recherche linéaire que pour la recherche dichotomique.

Si la recherche dichotomique est plus performante que la recherche linéaire pour une certaine simulation, nous pourrions presque dire que le niveau maximum L est choisi trop grand.

niveau grossier (l_0)	3	4	5	6	7	8
recherche linéaire	1.50	1.54	1.50	1.49	1.40	1.21
recherche dichotomique	3	3	2.90	2.85	2	2

Tab. 7.2 – Nombre moyen des accès à la structure pour trouver une cellule.

7.6.2 Impact de la structure à deux niveaux

Le niveau grossier l_0 défini l'ensemble des blocs de données sur lesquels les phases de l'algorithme sont effectuées. En posant l_0 comme niveau d'indirection dans la structure, on assure que l'ensemble des données d'un bloc se trouve dans un unique tableau. Il s'agit du tableau de valeurs au premier niveau si le bloc correspond à la seule cellule de niveau l_0, ou d'un tableau de valeur au deuxième niveau si le bloc contient des cellules de niveau inférieur.

Le niveau l_0 doit être choisi avec précautions : de sa valeur peut dépendre une partie des performances. En effet, les valeurs à la frontière des blocs sont dupliquées pour simplifier la distribution des données entre les processeurs. Cela implique un surcoût, en terme de calcul et de stockage, qui peut ne pas être négligeable. Un niveau l_0 trop grossier entraîne une augmentation du nombre de nœuds stockés sans valeur, dans les tableaux de deuxième niveau, alors qu'un niveau l_0 trop fin augmente le nombre de nœuds calculés, et par conséquent rapproche la méthode adaptative d'une méthode uniforme.

Montrons l'influence de l_0 sur un exemple en deux dimensions. Prenons le maillage (A) donné à gauche dans la figure 7.13. La figure 7.14 montre le remplissage des tableaux de premier et deuxième niveaux dans la structure qui décrit le maillage pour $l_0 = 1$. Dans cet exemple, la structure contient 82% des nœuds d'un maillage uniforme, et seulement 27% des nœuds alloués représentent des nœuds existants dans le maillage et ont une valeur. Le nombre de nœuds calculés représente une augmentation de 3, 5% du nombre de nœuds total du maillage.

En prenant un niveau grossier $l_0 = 2$, on peut remarquer que la cellule dans le quadrant en haut et à droite du maillage ne peut plus être représentée telle quelle. Le maillage correspond alors au maillage (B), donné à droite de la figure 7.13. Cela implique une augmentation du nombre d'éléments à traiter. La figure 7.15 montre le remplissage des tableaux de niveaux 1 et 2 pour décrire un tel maillage. La structure ne contient plus que 37% des nœuds alloués pour un maillage uniforme. Les nœuds qui existent dans le maillage et possèdent par conséquent une valeur dans la structure, représentent désormais 65% des nœuds alloués. Le *gâchis* de mémoire est donc beaucoup moins important, alors que le surcoût en terme de nœuds à calculer est de 9%.

En conclusion, le choix du niveau l_0 exprime un compromis entre la charge de calcul et l'occupation mémoire. Les performances liées au choix de ce niveau dépendent fortement du cas test considéré, aussi nous laissons, pour l'instant, ce choix à l'utilisateur.

7.6.3 Performances du code 4D

Le code 4D correspondant à l'algorithme parallèle est écrit en C avec appels à MPI. Le cas test présenté est la focalisation uniforme d'un faisceau semi-Gaussien (voir B.2 pour les données du cas et la figure B.1 pour l'aspect de la fonction de distribution). Pour cette expérience, l'espace des phases est divisé en 16×16 tranches de 8×8 blocs chacune, avec un niveau de raffinement maximum $L = 2$ pour chaque bloc. Cette division permet d'obtenir au maximum une grille de $128 \times 128 \times 64 \times 64$ points. La simulation est exécutée sur le cluster d'Opteron du CECPV (voir C).

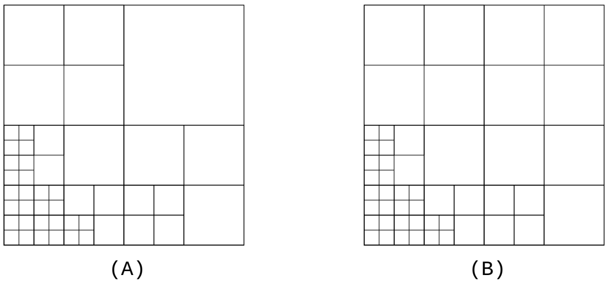
(A) (B)

FIG. 7.13 – Maillages dyadiques de référence.

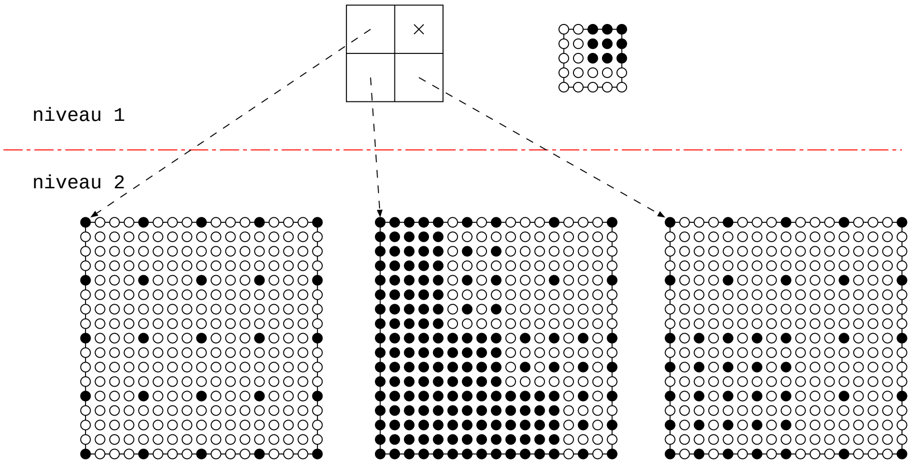

FIG. 7.14 – État de la structure de données pour $l_0 = 1$.

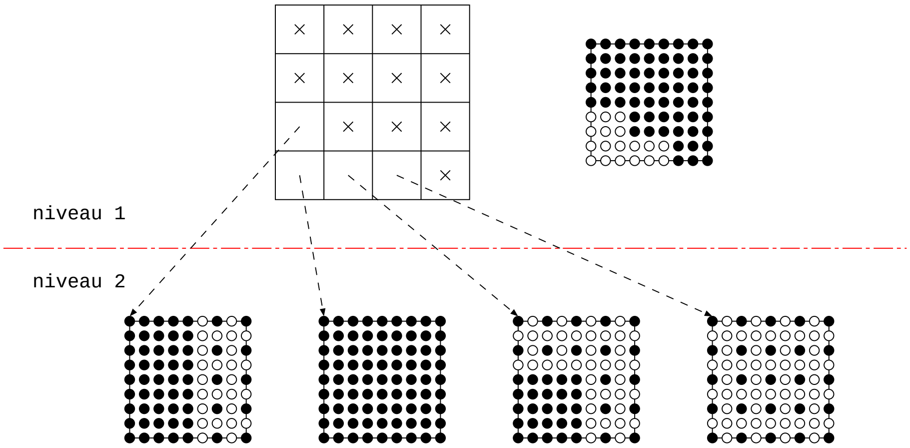

FIG. 7.15 – État de la structure de données pour $l_0 = 2$.

Impact de la méthode adaptative Nous nous intéressons tout d'abord à la réduction de la quantité de calculs grâce à la méthode adaptative. La figure 7.16 montre que le nombre de cellules du maillage varie au cours de la simulation. Les cellules traitées du maillage adaptatif représentent entre 5 et 9% des cellules d'un maillage uniforme. Le recours à des méthodes adaptatives est particulièrement intéressant pour simuler les cas test de faisceaux qui présentent de nombreuses zones “vides” (voir figure B.1).

Fig. 7.16 – Réduction des calculs : évolution du pourcentage de cellules restantes.

Nous avons vu que le structure de données à base de tableaux à deux niveaux ne permet pas de réduire au maximum la quantité de mémoire utilisée. La figure 7.17 montre l'évolution, au cours de la simulation, du pourcentage de blocs pleins. On remarque que la réduction de la mémoire est inférieure à la réduction du nombre de calculs. Ceci est dû au fait que les deux types de blocs permettent l'adaptativité en terme de calculs, alors que seuls les blocs creux permettent l'adaptativité en terme de mémoire. Le nombre de blocs pleins permet malgré tout une baisse significative de la consommation mémoire.

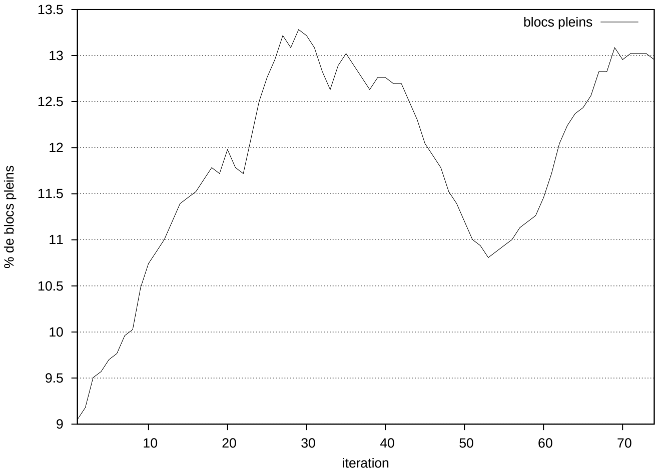

Fig. 7.17 – Réduction de l'usage mémoire : évolution du pourcentage de blocs pleins.

Impact de l'équilibrage de charge. La figure 7.18 montre le déséquilibre de charge de calcul (en secondes) pour la simulation lancée sur 16 processeurs et pour 3 différentes stratégies d'équilibrage de charge. La valeur du déséquilibre est donnée par la différence entre la charge associée à la région la plus coûteuse et la charge associée à la région la moins coûteuse. Ce déséquilibre est donné pour une simulation :

- sans équilibrage de charge dynamique (ECS).
- avec équilibrage de charge dynamique basé sur les bissections récursives de l'ensemble des tranches par des coupes rectilignes (rECD).
- avec équilibrage de charge dynamique basé sur des bissections récursives de l'ensemble des tranches par des coupes avec un zigzag autorisé (zECD). De plus, une marque indique chaque étape de temps où un équilibrage est réalisé.

La courbe correspondant à l'équilibrage dynamique zECD montre que cette stratégie permet un d'atteindre un déséquilibre légèrement inférieur à 1 seconde tout au long de la simulation.

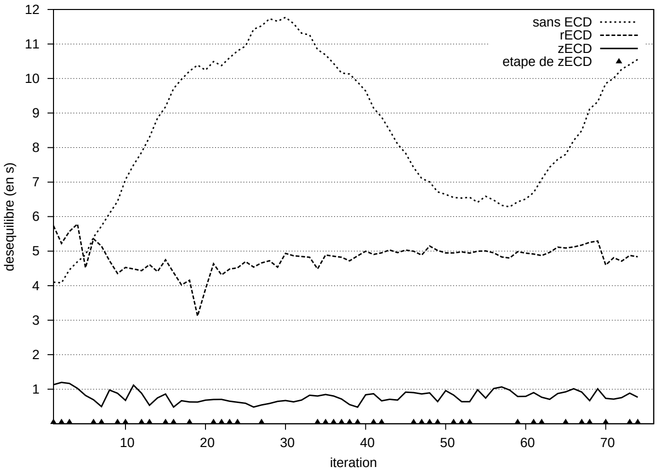

Fig. 7.18 – Évolution du déséquilibre pour différentes stratégies d'équilibrage de charge.

La partition des tranches de blocs avec la stratégie zECD est donnée dans la figure 7.19 avec la grille de 16 × 16 et pour 8 et 16 processeurs.

La figure 7.20 montre l'impact de la migration des blocs lors de certaines phases d'équilibrage. Les colonnes claires représentent le nombre total de blocs communiqués pendant une étape de temps. Les colonnes foncées représentent le nombre de blocs qui sont attendus aux barrières à la fin des phases de communications par la fonction `MPI_Waitall`. Il s'agit donc des blocs dont le transfert n'est pas totalement recouvert par les calculs. Enfin, les barres verticales représentent le temps d'attente passé dans la fonction `MPI_Waitall`. Nous observons que pour une grande majorité d'étapes de temps, tous les blocs transférés sont reçus avant la barrière d'attente des communications. Mais quelques étapes présentent une forte augmentation ($\times 1,5$ à $\times 2$) du nombre de blocs communiqués. Ces étapes correspondent à des phases de forte migration des blocs suite à un équilibrage de charge. Quand le nombre de blocs à communiquer est trop grand par rapport aux nombres de blocs à calculer, le temps de communication ne peut plus être entièrement recouvert par les calculs, ce qui peut engendrer des temps d'attente de fin de communication. Ce surcoût est inhérent à notre algorithme de partitionnement qui ne prend pas en compte le coût des communications pour définir les régions.

Performances. Nous comparons maintenant l'influence de notre algorithme d'équilibrage dynamique en fonction du nombre de processeurs utilisés. Le temps total d'exécution

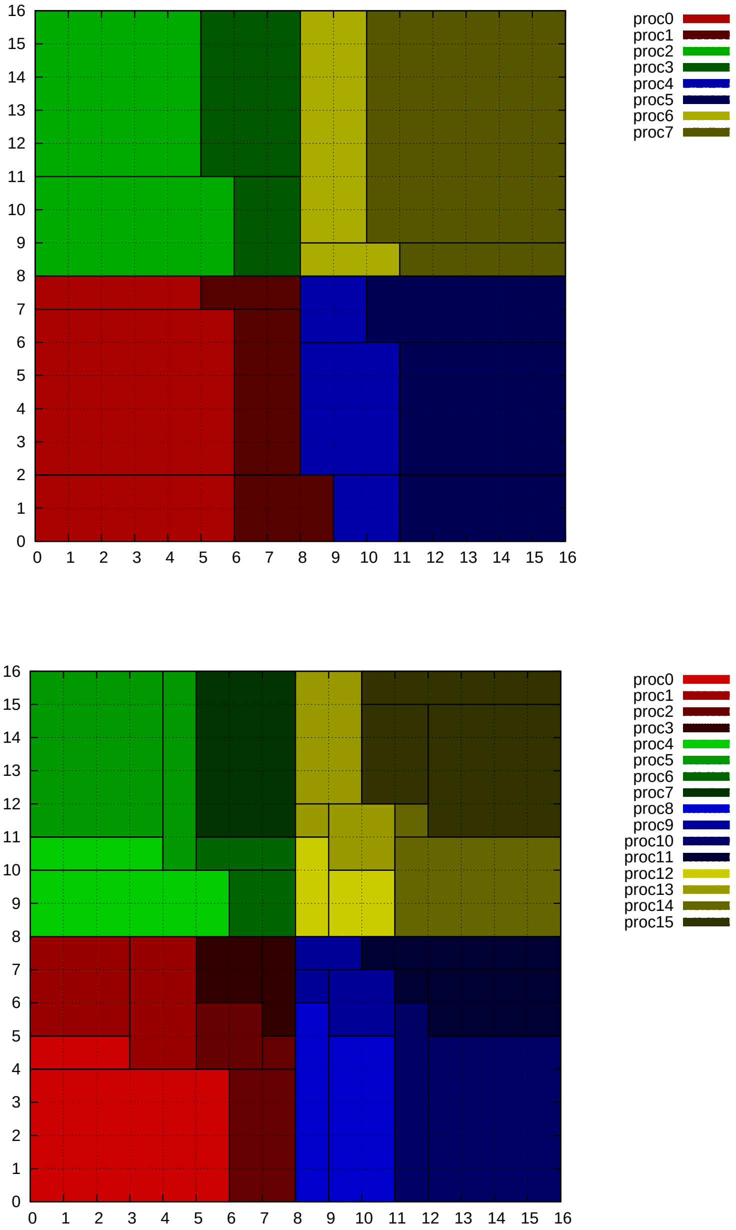

FIG. 7.19 – Partitionnement de la grille des tranches de blocs pour 8 (en haut) et 16 (en bas) processeurs à l'aide de l'algorithme zECD.

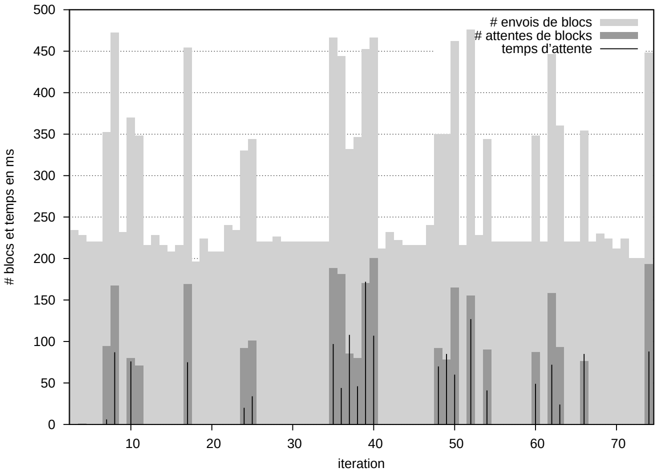

FIG. 7.20 – Impact de la migration des blocs de données au cours de la simulation.

de la simulation est mesuré pour la même simulation mais avec 4 stratégies différentes d'équilibrage de charge : les équilibrages dynamiques zECD et rECD, ainsi que les équilibrages statiques basés sur un partitionnement par zones de même taille (eECS) et un partitionnement basé sur la boîte englobante de la fonction de distribution initiale (bECS). Dans la figure 7.21, chaque colonne indique le temps d'exécution total de la simulation pour une stratégie donnée et pour un nombre de processeurs allant de 4 à 32. L'algorithme avec coupes en zigzag permet d'obtenir les meilleurs résultats quel que soit le nombre de processeurs utilisés.

La figure 7.22 montre l'efficacité obtenue jusqu'à 32 processeurs pour les stratégies rECD et zECD. Une simulation est aussi exécutée avec 32×32 tranches de blocs au lieu de 16×16. Les courbes d'accélérations montrent que les performances obtenues par la stratégie zECD sur 16×16 tranches sont meilleures qu'avec la stratégie rECD sur 16×16 ou 32×32. Quelle que soit la stratégie utilisée, l'augmentation du nombre de tranches permet de mieux répartir la charge de calcul entre les différents processeurs.

Cependant, cette augmentation implique également d'utiliser un Δt plus petit, puisque la valeur de Δ est proportionnelle à la taille d'un bloc d'après les hypothèses sur les paramètres de la simulation. Si nous voulons utiliser le même Δt, alors une solution serait d'augmenter le nombre de blocs répliqués en x et en y. Cela augmenterait le nombre de messages mais le volume de données transférées resterait identique.

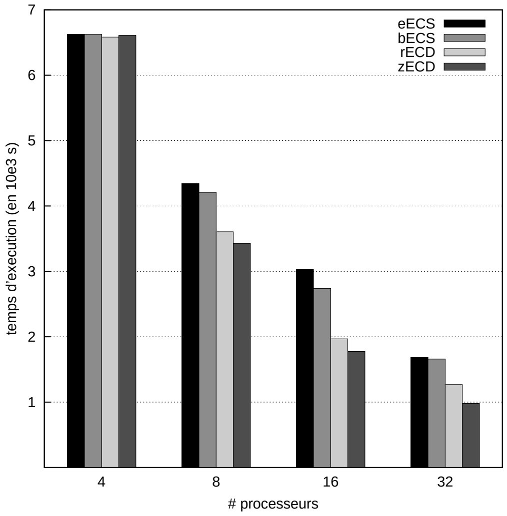

FIG. 7.21 – Temps d'exécution total pour les différentes stratégies d'équilibrage de charge.

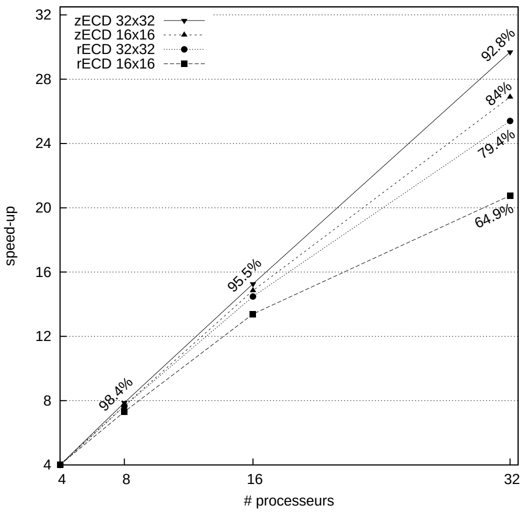

FIG. 7.22 – Accélération et efficacité pour la simulation avec 16×16 et 32×32 tranches de blocs.

Conclusion

Résultats obtenus

Dans la première partie de cette thèse (chapitres 4 et 5), nous avons étudié et parallélisé une méthode numérique basée sur une représentation hiérarchique des éléments du maillage et un opérateur d'interpolation local. Nous avons proposé, pour cette parallélisation, divers mécanismes permettant une grande dynamicité pour gérer l'adaptativité de la méthode. Les performances obtenues en 2 dimensions sont tout à fait correctes. Par contre, la complexité de ces mécanismes dépend du nombre de dimensions, et par conséquent, ne permet pas d'obtenir des performances satisfaisantes en 4 dimensions. Les résultats obtenus dans le chapitre 5 montrent que ces performances peuvent être nettement améliorées si la structure de données stockant le maillage adaptatif est plus régulière.

Dans la deuxième partie de cette thèse (chapitres 6 et 7), nous avons modifié la méthode d'adaptation du maillage pour que son surcoût soit moins pénalisant. Cette modification nous permet d'exprimer l'algorithme de résolution comme un algorithme récursif par bloc. La parallélisation de cette deuxième méthode est axée sur une régularisation de la structure de données et des dépendances entre les données. La maximisation du recouvrement des communications par le calcul permet de réduire considérablement le surcoût de la parallélisation. Alors que l'équilibrage dynamique de la charge est réalisé par un algorithme simple, nous avons obtenu avec ce code parallèle une efficacité de 92% pour une simulation 4D sur 32 processeurs.

Bilan

Les travaux effectués dans cette thèse montrent que le système Vlasov–Poisson peut être résolu par des solveurs à la fois adaptatifs et performants sur des architectures à mémoire distribuée.

Nous avons effectué la parallélisation de deux méthodes adaptatives. La deuxième méthode étant obtenue à partir de la première en modifiant l'adaptation du maillage. Cette modification a permis d'améliorer considérablement la localité des données et a un impact important sur les performances de son implantation parallèle. Ces travaux montrent que pour obtenir la meilleure efficacité, il faut prendre en compte très tôt, au niveau de la conception de la méthode numérique, les contraintes liées à la parallélisation.

Enfin, ces travaux permettent de mettre en évidence que la conception d'un solveur adaptatif pour la résolution de l'équation de Vlasov doit proposer des méthodes permettant de réguler cette adaptativité pour obtenir un code efficace. Il est intéressant de remarquer que les structures de données de notre code sont proches de celles du code adaptatif OBIWAN [50, 49] alors que les méthodes de résolutions utilisées sont très différentes. Cette structure de données permet en effet de conserver un degré d'adaptativité satisfaisant tout en simplifiant grandement la gestion dynamique des éléments.

Perspectives et travaux futurs

Les travaux effectués durant cette thèse sont la base de propositions pour réaliser des simulations pour le cas 6D en utilisant des architectures à mémoire distribuée et un grand nombre de processeurs. De telles simulations nécessitent la prise en compte de contraintes supplémentaires liées notamment à l'usage mémoire qui est considérable. Il est alors primordial d'exploiter au mieux la mémoire disponible pour chaque nœud de calcul. La distribution des blocs doit donc faire un compromis entre équilibre de charge de calcul et répartition équitable de la mémoire utilisée.

En complément, une perspective de ces travaux concerne l'extensibilité de notre code parallèle. Avec la multiplication des architectures basées sur des nœuds SMP et des processeurs multi-cœurs il devient intéressant d'utiliser à la fois la mémoire partagée et le passage de message. Une idée simple consiste à paralléliser le traitement d'une tranche en attribuant le calcul des blocs de cette tranche à des processeurs différents partageant le même espace mémoire. La plupart des implantations MPI actuelles permettent d'effectuer des communications par mémoire partagée quand les processus MPI sont présents sur un même nœud SMP. Nous envisageons alors l'étude comparative d'une approche par placement explicite des processus et par parallélisation mixte MPI/OpenMP.

Enfin, nous avons vu que la régularisation, dans une certaine mesure, de l'adaptativité de la méthode en terme de calculs, de communications et de stockage mémoire, permet d'améliorer les performances d'un solveur adaptatif sur des grandes dimensions. Il faut donc faire un compromis entre adaptativité et uniformité pour espérer obtenir les meilleures performances. Des travaux futurs pourront concerner l'élaboration de méthodes permettant de déterminer le meilleur compromis, ou de choisir de manière automatique un compromis acceptable en fonction du cas test considéré.

Bibliographie

[1] Jochen Alber and Rolf Niedermeier. On Multi-dimensional Hilbert Indexings. In *COCOON '98 : Proceedings of the 4th Annual International Conference on Computing and Combinatorics*, volume 1449 of *LNCS*, pages 329–338. Springer-Verlag, 1998.

[2] Shawki Areibi and Zhen Yang. Effective Memetic Algorithms for VLSI Design Automation = Genetic Algorithms + Local Search + Multi-level Clustering. *Evol. Comput.*, 12(3) :327–353, 2004.

[3] David C. Arney and Joseph E. Flaherty. A Two-dimensional Mesh Moving Technique for Time-dependent Partial Differential Equations. *J. Comput. Phys.*, 67(1) :124–144, 1986.

[4] Ivo Babuska, Jagdish Chandra, and Joseph E. Flaherty. *Adaptive Computational Methods for Partial Differential Equations.* SIAM, Philadelphia, PA, USA, 1983.

[5] Ivo Babuska and Manil Suri. The p- and h-p Versions of the Finite Element Method, an Overview. *Comput. Methods Appl. Mech. Engrg.*, 80 :5–26, 1990.

[6] Ivo Babuska, Barna A. Szabo, and I. Norman Katz. The p-version of the Finite Element Method. *SIAM J. Numer. Anal.*, 18(3) :515–545, June 1981.

[7] Michael J. Baines. Grid Adaptation via Node Movement. *Appl. Numer. Math.*, 26(1-2) :77–96, 1998.

[8] R. E. Bank, A. H. Sherman, and H. Weiser. Refinement Algorithms and Data Structures for Regular Local Mesh Refinement. *Scientific Computing*, pages 3–17, 1983.

[9] Stephen T. Barnard and Horst D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency : Practice and Experience*, 6 :101–107, 1994.

[10] Régine Barthelmé. *Le Problème de Conservation de la Charge dans le Couplage des Équations de Vlasov et de Maxwell.* PhD thesis, Université Louis Pasteur, 2005.

[11] Marsha J. Berger and Shahid H. Bokhari. A Partitioning Strategy for Non-uniform Problems on Multiprocessors. *IEEE Transactions on Computers*, 36(5) :570–580, 1987.

[12] Marsha J. Berger and Phillip Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *J. Comput. Phys.*, 82 :64–84, 1989.

[13] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.*, 53 :484–512, 1984.

[14] Nicolas Besse, Francis Filbet, Michael Gutnic, Ioana Paun, and Éric Sonnendrücker. An Adaptive Numerical Method for the Vlasov Equation Based on a Multiresolution

Analysis. In *Numerical Mathematics and Advanced Applications*, pages 437–446, 2001.

[15] Charles K. Birdshall and A. Bruce Langdon. *Plasmaphysics via Computer Simulation.* McGraw-Hill, 1985.

[16] Shahid H. Bokhari. On the Mapping Problem. *IEEE Trans. Computers*, 30(3) :207–214, 1981.

[17] Luc Bougé, David Cachera, Yann Le Guyadec, Gil Utard, and Bernard Virot. The Data Parallel Programming Model : A Semantic Perspective. In *LNCS Tutorial : The Data Parallel Programming Model*, volume 1132, pages 4–26. Springer-Verlag, 1996.

[18] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial (2nd ed.).* SIAM, 2000.

[19] Martin Campos Pinto. *Développement et Analyse de Méthodes Adaptatives pour les Équations de Transport.* PhD thesis, Université Pierre et Marie Curie, Paris 6, 2005.

[20] Martin Campos Pinto and Michel Mehrenberger. Adaptive Numerical Resolution of the Vlasov Equation. In *Numerical Methods for Hyperbolic and Kinetic Problems, CEMRACS*, pages 43–58, 2004.

[21] Martin Campos Pinto and Michel Mehrenberger. Convergence of an Adaptive Semi-Lagrangian Scheme for the Vlasov-Poisson System. *Numer. Math.*, 108(3) :407–444, 2007.

[22] Feng Cao, John R. Gilbert, and Shang-Hua Teng. Partitioning Meshes with Lines and Planes. Technical Report CSL-96-01, Xerox PARC, 1996.

[23] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler Optimizations for Improving Data Locality. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 252–262, San Jose, California, 1994.

[24] Chio-Zong Cheng and George Knorr. The Integration of the Vlasov Equation in Configuration Space. *J. of Comput. Phys.*, 22 :330–351, nov 1976.

[25] Michal Cierniak and Wei Li. Interprocedural Array Remapping. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '97)*, November 1997.

[26] Bernardo Cockburn, George E. Karniadakis, and Chi-Wang Shu. *Discontinuous Galerkin Methods, Theory, Computation and Applications*, volume 11 of *LNCSE.* Springer, 2000.

[27] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Diffusive Load-Balancing Policies for Dynamic Applications. *IEEE Concurrency*, 7(1) :22–31, 1999.

[28] Richard Courant, Kurt Friedrichs, and Hans Lewy. On the Partial Difference Equations of Mathematical Physics. *IBM Journal of Reseach and Developement : Difference Equations*, 11(2), 1967. English translation of the 1928 German original.

[29] Nicolas Crouseilles, Alain Ghizzo, and Stéphanie Salmon. Vlasov Laser-plasma Interaction Simulations in the Relativistic Regime with a Moving Grid. Technical Report RR-6109, INRIA Nancy – Grand Est, 2007.

[30] Nicolas Crouseilles, Michael Gutnic, Guillaume Latu, and Éric Sonnendrücker. Comparison of Two Eulerian Solvers for the Four Dimensional Vlasov Equation : Part II. In *Proc. of the 2nd international conference of Vlasovia*, volume 13(1) of *Communications in Nonlinear Science and Numerical Simulation*, pages 94–99, 2008.

[31] George Cybenko. Dynamic Load Balancing for Distributed Memory Multiprocessors. *J. Parallel Distrib. Comput.*, 7(2) :279–301, 1989.

[32] Hugues L. deCougny, Karen D. Devine, Joseph E. Flaherty, R. M. Loy, Can Özturan, and Mark S. Shephard. Load Balancing for the Parallel Adaptive Solution of Partial Differential Equations. *Appl. Numer. Math.*, 16(1-2) :157–182, 1994.

[33] Ralf Diekmann and Robert Preis. Load Balancing Strategies for Distributed Memory Machines. *Parallel and distributed processing for computational mechanics : systems and tools*, pages 124–157, 1999.

[34] Ralf Diekmann, Robert Preis, Frank Schlimbach, and Chris Walshaw. Shape-optimized Mesh Partitioning and Load Balancing for Parallel Adaptive FEM. *Parallel Comput.*, 26(12) :1555–1581, 2000.

[35] Chris Ding and Yun He. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems. In *Supercomputing'01 (CDROM)*, 2001.

[36] Rafael Van Driessche and Dirk Roose. An Improved Spectral Bisection Algorithm and its application to dynamic load balancing. *Parallel Comput.*, 21(1) :29–48, 1995.

[37] Bengt Eliasson. Numerical Modelling of the Two-dimensional Fourier Transformed Vlasov-Maxwell System. *J. Comput. Phys.*, 190(2) :501–522, 2003.

[38] Charbel Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Computer and Structures*, 28 :579–602, 1988.

[39] Charbel Farhat and Michel Lesoinne. Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics. *International Journal for Numerical Methods in Engineering*, 36 :745–764, 1993.

[40] C. M. Fiduccia and R. M. Mattheyses. A Linear-time Heuristic for Improving Network Partitions. In *ACM Ieee Nineteenth Design Automation Conference Proccedings*, pages 175–181, 1982.

[41] Miroslav Fiedler. A Property of Eigenvectors of Non-negative Symmetric Matrices and its Application to Graph Theory. *Czechoslovak Mathematical Journal*, 25, 1975.

[42] Francis Filbet and Éric Sonnendrücker. Modeling and Numerical Simulation of Space Charge Dominated Beams in the Paraxial Approximation. Technical Report RR-5547, INRIA Lorraine, 2005.

[43] Joseph E. Flaherty, R. M. Loy, Mark S. Shephard, Boleslaw K. Szymanski, James D. Teresco, and Louis H. Ziantz. Adaptive Local Refinement with Octree Load Balancing for the Parallel Solution of Three-Dimensional Conservation Laws. *Journal of Parallel and Distributed Computing*, 47(2) :139–152, 1997.

[44] Michael R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some Simplified NP-complete Problems. In *STOC '74 : Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63, 1974.

[45] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite.* Prentice Hall Professional Technical Reference, 1981.

[46] Philippe Gerner and Éric Violard. A Theoretical Framework of Data Parallelism and its Operational Semantics. In *EURO-PAR'2000*, volume LNCS 1900, pages 668–677. Springer-Verlag, September 2000.

[47] Michael Gutnic, Matthieu Haefele, and Guillaume Latu. A Parallel Vlasov Solver Using a Wavelet Based Adaptive Mesh Refinement. In *7th Workshop on High Perf. Scientific and Engineering Computing (ICPP'2005)*, pages 181–188, 2005.

[48] Michael Gutnic, Matthieu Haefele, Ioana Paun, and Éric Sonnendrücker. Vlasov Simulations on an Adaptive Phase-space Grid. *Comput. Phys. Commun.*, 164 :214–219, dec 2004.

[49] Matthieu Haefele. *Simulation Adaptative et Visualisation Haute Performance de Plasmas et de Faisceaux de Particules.* PhD thesis, Université Louis Pasteur, Strasbourg I, 2007.

[50] Matthieu Haefele, Guillaume Latu, and Michael Gutnic. A Parallel Vlasov Solver Using a Wavelet Based Adaptive Mesh Refinement. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops (ICPPW'05)*, pages 181–188, 2005.

[51] D. F. Hawken, J. J. Gottlieb, and J. S. Hansen. Review of some Adaptive Node-movement Techniques in Finite-Element and Finite-Difference Solutions of Partial Differential Equations. *J. Comput. Phys.*, 95 :254–302, aug 1991.

[52] Bruce Hendrickson and Karen D. Devine. Dynamic Load Balancing in Computational Mechanics. *Computational Methods in Applied Mechanics & Engineering*, 184 :485–500, 2000.

[53] Bruce Hendrickson and Robert W. Leland. The Chaco User's Guide Version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994.

[54] Bruce Hendrickson and Robert W. Leland. A Multi-Level Algorithm For Partitioning Graphs. In *Supercomputing.* ACM, 1995.

[55] Bruce Hendrickson and Robert W. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. *SIAM Journal on Scientific Computing*, 16(2) :452–469, 1995.

[56] Olivier Hoenen, Michel Mehrenberger, and Éric Violard. Parallelization of an Adaptive Vlasov Solver. In *11th European PVM/MPI Users' Group Conference (EuroPVM/MPI), ParSim Session*, volume 3241 of *LNCS*, pages 430–435. Springer-Verlag, 2004.

[57] Olivier Hoenen and Éric Violard. Parallélisation d'un Solveur Adaptatif de l'Équation de Vlasov. In *16ème Rencontres Francophones du Parallélisme (RenPar '05)*, pages 249–258, April 2005.

[58] Olivier Hoenen and Éric Violard. A Local Adaption Algorithm for Solving the Vlasov Equation. Technical Report 06-09, LSIIT–ICPS, August 2006.

[59] Olivier Hoenen and Éric Violard. An Efficient Data Structure for an Adaptive Vlasov Solver. Technical Report 06-02, LSIIT–ICPS, February 2006.

[60] Olivier Hoenen and Éric Violard. A Block-based Parallel Adaptive Scheme for Solving the 4D Vlasov Equation. In *the 7th Parallel Processing and Applied Mathematics Conference (PPAM)*, volume 4967. LNCS, September 2007.

[61] Olivier Hoenen and Éric Violard. Load-Balancing for a Block-based Parallel Adaptive 4D Vlasov Solver. to appear in proceedings of EUROPAR 2008, 2008.

[62] Paul Houston and Endre Süli. A Note on the Design of hp-Adaptive Finite Element Methods for elliptic partial differential equations. *Computer Methods in Applied Mechanics and Engineering*, 194(2-5) :229–243, 2005.

[63] Y. F. Hu and R. J. Blake. Load Balancing for Unstructured Mesh Applications. *Progress in computer research*, pages 117–148, 2001.

[64] Y. F. Hu and D. R. Emerson R. J. Blake. An Optimal Migration Algorithm for Dynamic Load Balancing. *Concurrency : Practice and Experience*, 10(6) :467–483, 1997.

[65] Peter K. Jimack. An Overview of Dynamic Load-Balancing for Parallel Adaptive Computational Mechanics Codes. *Parallel and Distributed Processing for Computational Mechanics : Systems and Tools*, pages 350–369, 1999.

[66] Mark T. Jones and Paul E. Plassmann. Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes. In *Proc. 1994 Scalable High-Performance Computing Conf.*, pages 478–485. IEEE Computer Society, 1994.

[67] Mahmut T. Kandemir, Alok N. Choudhary, J. Ramanujam, Nagaraj Shenoy, and Prithviraj Banerjee. Enhancing Spatial Locality via Data Layout Optimizations. In *European Conference on Parallel Processing (EuroPar '98)*, pages 422–434, 1998.

[68] George Karypis and Vipin Kumar. *MeTis : Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*, 1995.

[69] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1) :359–392, 1998.

[70] George Karypis and Vipin Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48(1) :96–129, 1998.

[71] Ken Kennedy and Kathryn S. McKinley. Optimizing for Parallelism and Data Locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC, 1992.

[72] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Tech. J.*, 49 :291–308, 1970.

[73] Donald E. Knuth. *The Art of Computer Programming : Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[74] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*, January 1994.

[75] Jonathan K. Lawder and Peter J. H. King. Using Space-Filling Curves for Multidimensional Indexing. In *Proceedings of the 17th British National Conferenc on Databases : Advances in Databases*, volume 1832 of *LNCS*, pages 20–35. Springer-Verlag, 2000.

[76] Jonathan K. Lawder and Peter J. H. King. Using State Diagrams for Hilbert Curve Mappings. *International Journal of Computer Mathematics*, 78(3) :327–342, 2001.

[77] John D. Lawson. Some Criteria for a Power Producing Thermonuclear Reactor. *Proceedings of Physical Society B*, 70(6), 1957.

[78] Shun-Tak A. Leung. Array Restructuring for Cache Locality. Technical Report TR-96-08-01, Department of computer science, University of Washington, 1996.

[79] Charles A. Lin, René Laprise, and Harold Ritchie (Eds.). *Numerical Methods in Atmospheric and Oceanic Modelling.* Canadian Meteorological and Oceanographic Society, 1997.

[80] Michel Mehrenberger. *Inégalités d'Observabilité et Résolution Adaptative de l'Équation de Vlasov par Éléments Finis Hierarchiques.* PhD thesis, Université Louis Pasteur, Strasbourg I, 2004.

[81] Michel Mehrenberger, Éric Violard, Olivier Hoenen, Martin Campos Pinto, and Éric Sonnendrücker. A Parallel Adaptive Vlasov Solver Based on Hierarchical Finite Element Interpolation. *Nuclear Inst. and Methods in Physics Research*, 558(A) :188–191, 2006.

[82] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing Non-Affine Array References. In *IEEE PACT*, pages 192–202, 1999.

[83] William F. Mitchell. A Refinement-tree Based Partitioning Method for Dynamic Load Balancing with Adaptively Refined Grids. *J. Parallel Distrib. Comput.*, 67(4) :417–429, 2007.

[84] *Official MPI (*Message Passing Interface*) standards documents, errata, and archives.*

[85] Can Özturan. *Distributed Environment and Load Balancing for Adaptive Unstructured Meshes.* PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1995.

[86] Bruce Palmer and Jarek Nieplocha. Efficient Algorithms for Ghost Cell Updates on Two Classes of MPP Architectures. In *Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 192–197, 2002.

[87] Manish Parashar and James C. Browne. Distributed Dynamic Data-Structures for Parallel Adaptive Mesh Refinement. In *International Conference on High Performance Computing (HiPC'95)*, 1995.

[88] Manish Parashar, James C. Browne, Carter Edwards, and Kenneth Klimkowski. A Common Data Management Infrastructure for Adaptive Algorithms for PDE Solutions. In *Supercomputing*, 1997.

[89] Abani K. Patra and J. Tinsley Oden. Problem Decomposition Strategies for Adaptive hp Finite Element Methods. *Computing Systems in Engineering*, 6(2) :97–109, 1995.

[90] François Pellegrini. Scotch and libScotch 5.0 User's Guide.

[91] John R. Pilkington and Scott B. Baden. Partitioning with Spacefilling Curves. Technical Report CS94-349, Dept. Computer Science and Engineering, University of California, 1994.

[92] Alex Pothen, Horst D. Simon, and Kan-Pu Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Matrix Anal. Appl.*, 11(3) :430–452, 1990.

[93] Robert Preis and Ralf Diekmann. The PARTY Partitioning-Library, User Guide - Version 1.1. Technical Report tr-rsfb-96-024, University of Paderborn, 1996.

[94] Maria-Cecilia Rivara and Patricio Inostroza. Using Longest-side Bisection Techniques for the Automatic Refinement of Delauney Triangulation. *Int. J. for Num. Meth. in Eng.*, 40(4) :581–597, 1997.

[95] Arnold L. Rosenberg and James W. Thatcher. What Is a Multilevel Array ? *IBM J. Res. Develop.*, 19(2) :163–169, 1975.

[96] Hans Sagan. *Space-Filling Curves.* Springer, 1994.

[97] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. *J. Parallel Distrib. Comput.*, 47(2) :109–124, 1997.

[98] John V. Shebalin. A Spectral Algorithm for Solving the Relativistic Vlasov-Maxwell Equations. *Comput. Phys. Comm.*, 156 :86–94, 2003.

[99] Mark S. Shephard. Automatic and Adaptive Mesh Generation. *IEEE tansactions on Magnetics*, 21(6) :2484–2489, 1985.

[100] Mark S. Shephard and Marcel K. Georges. Automatic Three-dimensional Mesh Generation by the Finite Octree Technique. *Int. J. for Numerical Methods in Engineering*, 32(4) :709–749, 1991.

[101] Horst D. Simon. Partitioning of Unstructured Problems for Parallel Processing. *Computing Systems in Engineering*, 2 :135–148, 1991.

[102] Jaswinder P. Singh, Chris Holt, John L. Hennessy, and Anoop Gupta. Parallel Adaptive Fast Multipole Method. In *Proceedings of the Supercomputing Conference 1993*, pages 54–65, Los Alamitos, 1993. IEEE, Computer Society Press.

[103] Andrew Sohn, Jui Ku, Yuetsu Kodama, Mitsuhisa Sato, Hirofumi Sakane, Hayato Yamana, Shuichi Sakai, and Yoshinori Yamaguchi. Identifying the Capability of Overlapping Computation with Communication. In *PACT '96 : Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, pages 133–138. IEEE Computer Society, 1996.

[104] Éric Sonnendrücker, Francis Filbet, Alex Friedman, Edouard Oudet, and Jean-Luc Vay. Vlasov Simulation of Beams with a Moving Grid. *Comput. Phys. Comm.*, 164 :390–395, 2004.

[105] Éric Sonnendrücker, Jean Roche, Pierre Bertrand, and Alain Ghizzo. The Semi-Lagrangian Method for the Numerical Resolution of the Vlasov Equation. *J. Comput. Phys.*, 149 :201–220, 1999.

[106] Johan Steensland, Stefan Söderberg, and Michael Thuné. A Comparison of Partitioning Schemes for Blockwise Parallel SAMR Algorithms. In *Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia (PARA '00)*, pages 160–169. Springer-Verlag, 2001.

[107] *Standard Template Library.*

[108] Gilbert Strang. On the Construction and Comparison of Difference Schemes. *SIAM Journal on Numerical Analysis*, 5(3) :506–517, 1968.

[109] Lei Tang and James D. Baeder. Uniformly Accurate Finite Difference Schemes for p-Refinement. *SIAM J. Sci. Comput.*, 20(3) :1115–1131, 1999.

[110] Valerie E. Taylor and Bahram Nour-Omid. A Study of the Factorization Fill-in for a Parallel Implementation of the Finite Element Method. *Int. J. for Num. Meth. in Engrg.*, 37(22) :3809 – 3823, 1994.

[111] Top 500 website, april 2008. `http://www.top500.org/`

[112] Denis Vanderstraeten, Charbel Farhat, P. S. Chen, Roland Keunings, and O. Ozone. A Retrofit Based Methodology for the Fast Generation and Optimization of Large-scale Mesh Partitions : Beyond the Minimum Interface Size Criterion. *Computer Methods in Applied Mechanics and Engineering*, 133(1-2) :25–45, 1996.

[113] Éric Violard and Françis Filbet. Parallelization of a Vlasov Solver by Communication Overlapping. In *PDPTA'02*, 2002.

[114] David W. Walker. The Hierarchical Spatial Decomposition of Three-Dimensional Particle-In-Cell Plasma Simulations on MIMD Distributed Memory Multiprocessors. Technical Report Technical Report ORNL/TM-12071, Oak Ridge National Laboratory, 1992.

[115] Michael S. Warren and John K. Salmon. A parallel Hashed Oct-Tree N-body Algorithm. In *Supercomputing*, pages 12–21, 1993.

[116] Jerrell Watts, Marc Rieffel, and Stephen Taylor. A Load Balancing Technique for Multiphase Computations. In *Proceedings of High Performance Computing'97*, pages 15–20. SCS, 1997.

[117] James B. White and Steve W. Bova. Where's the Overlap ? Ovelapping Communication and Computation in Several Popular MPI Implementations. In *Proceedings of Third MPI Developers' and Users' Conference*, 1999.

[118] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9) :979–993, 1993.

Annexe A

Opérateurs différentiels

Rappelons la définition de l'opérateur *nabla*, très utilisé en analyse vectorielle. Cet opérateur se note ∇ et est défini par :

$$\nabla = \begin{pmatrix} \dfrac{\partial}{\partial x} \\ \dfrac{\partial}{\partial y} \\ \dfrac{\partial}{\partial z} \end{pmatrix}$$

D'autres opérateurs vectoriels vont maintenant être définis grâce à l'opérateur ∇ : le *gradient*, la *divergence*, le *rotationnel* et le *Laplacien*. Le gradient est défini pour un champ scalaire f et est noté $\vec{\mathrm{grad}}$ (que l'on pourra simplifier par ∇).

$$\vec{\mathrm{grad}}\,(f) \quad = \quad \nabla f \quad = \quad \begin{pmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \\ \dfrac{\partial f}{\partial z} \end{pmatrix}$$

La divergence et le rotationnel notés respectivement div et $\vec{\mathrm{rot}}$, sont définis pour un champ de vecteurs $\vec{F} = (F_x(x,y,z)\ F_y(x,y,z)\ F_z(x,y,z))$.

$$\mathrm{div}\,(\vec{F}) \quad = \quad \nabla \cdot \vec{F} \quad = \quad \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z}$$

$$\vec{\mathrm{rot}}\,(\vec{F}) \quad = \quad \nabla \times \vec{F} \quad = \quad \begin{pmatrix} \dfrac{\partial F_z}{\partial y} - \dfrac{\partial F_y}{\partial z} \\ \dfrac{\partial F_x}{\partial z} - \dfrac{\partial F_z}{\partial x} \\ \dfrac{\partial F_y}{\partial x} - \dfrac{\partial F_x}{\partial y} \end{pmatrix}$$

Le gradient, la divergence et le rotationnel sont des opérateurs de premier ordre dans le sens où ils ne font intervenir que des dérivées partielles premières. Le Laplacien est un opérateur d'ordre 2 puisqu'il fait appel à des dérivées partielles secondes. Il peut être défini pour un champ scalaire f ou pour un champ de vecteurs $\vec{F}$.

$$\begin{aligned}
\Delta f &= \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \frac{\partial^2 f}{\partial z^2} \\
\Delta \vec{F} &= \begin{pmatrix} \dfrac{\partial^2 F_x}{\partial x^2} + \dfrac{\partial^2 F_x}{\partial y^2} + \dfrac{\partial^2 F_x}{\partial z^2} \\ \dfrac{\partial^2 F_y}{\partial x^2} + \dfrac{\partial^2 F_y}{\partial y^2} + \dfrac{\partial^2 F_y}{\partial z^2} \\ \dfrac{\partial^2 F_z}{\partial x^2} + \dfrac{\partial^2 F_z}{\partial y^2} + \dfrac{\partial^2 F_z}{\partial z^2} \end{pmatrix}
\end{aligned}$$

Ces opérateurs différentiels vérifient les propriétés suivantes :

$$\begin{aligned}
\mathrm{div}(\vec{\mathrm{rot}}) &= 0 \\
\vec{\mathrm{rot}}(\nabla) &= \vec{0} \\
\mathrm{div}(\nabla) &= \Delta \\
\vec{\mathrm{rot}}(\vec{\mathrm{rot}}) - \nabla(\mathrm{div}) &= -\Delta
\end{aligned}$$

Annexe B

Cas test

B.1 Focalisation uniforme d'un faisceau semi-Gaussien 2D

Le cas test considéré dans l'espace des phases à 2 dimensions est la focalisation uniforme d'un faisceau de particules semi-Gaussien. Pour un tel faisceau, la fonction de distribution initiale s'écrit

$$f_0(x,v) = \begin{cases} \dfrac{1}{\pi a^2 \sqrt{2\pi} b} e^{-1/2(v^2/b)} & \text{si } x < a \\ 0 & \text{sinon} \end{cases},$$

avec $a = 4/15$, $b = 1/(2\sqrt{15})$ et avec un pas de temps qui vaut 1/32$^{\text{e}}$ de la période, soit $\Delta t = 2\pi/32$. Cette simulation est effectuée sans splitting en temps, sur cinq périodes soit 160 itérations. Le maillage adaptatif est considéré avec $l_0 = 3$ et plusieurs valeurs de $L \in \{6, \cdots, 10\}$, soit une résolution entre 128 et 1024 points par dimension. Le seuil de compression vaut $1e^{-4}$.

B.2 Focalisation uniforme d'un faisceau semi-Gaussien 4D

Le cas test correspond à la focalisation uniforme d'un faisceau semi-Gaussien de protons dans l'espace des phases à 4 dimensions. L'emittence de ce faisceau de protons est de $2.36 \times 10^{-5}\, \pi\, m\, rad$, sont courant est de $100\, mA$, et son énergie est de $5\, MeV$. Les paramètres d'initialisation du code sont calculés en résolvant les équations d'enveloppe du faisceau KV équivalent (voir [42]) pour plus de détails. Ces paramètres physiques nous donne une dépression du nombre d'onde de 0.7. La longueur de la période est égale à $S = 26.6292\, m$. Le champ de focalisation appliqué est un champ magnétique uniforme $B_z = 0.218 T$.

Le faisceau de particules est donné par la fonction de distribution semi-Gaussienne

$$f_0(\mathbf{x},\mathbf{v}) = \begin{cases} \frac{1}{8\pi^2} e^{(-\frac{{v_x}^2 + {v_y}^2}{2})} & \text{if} \quad x^2 + y^2 < 6 \\ 0 & \text{else} \end{cases}$$

où $\mathbf{x}$ et $\mathbf{v}$ sont dans $[-6.5, 6.5]^2$. La figure B.1 montre la projection de la fonction de distribution sur le plan (x, y) (à gauche) et sur le plan (x, v_x) (à droite).

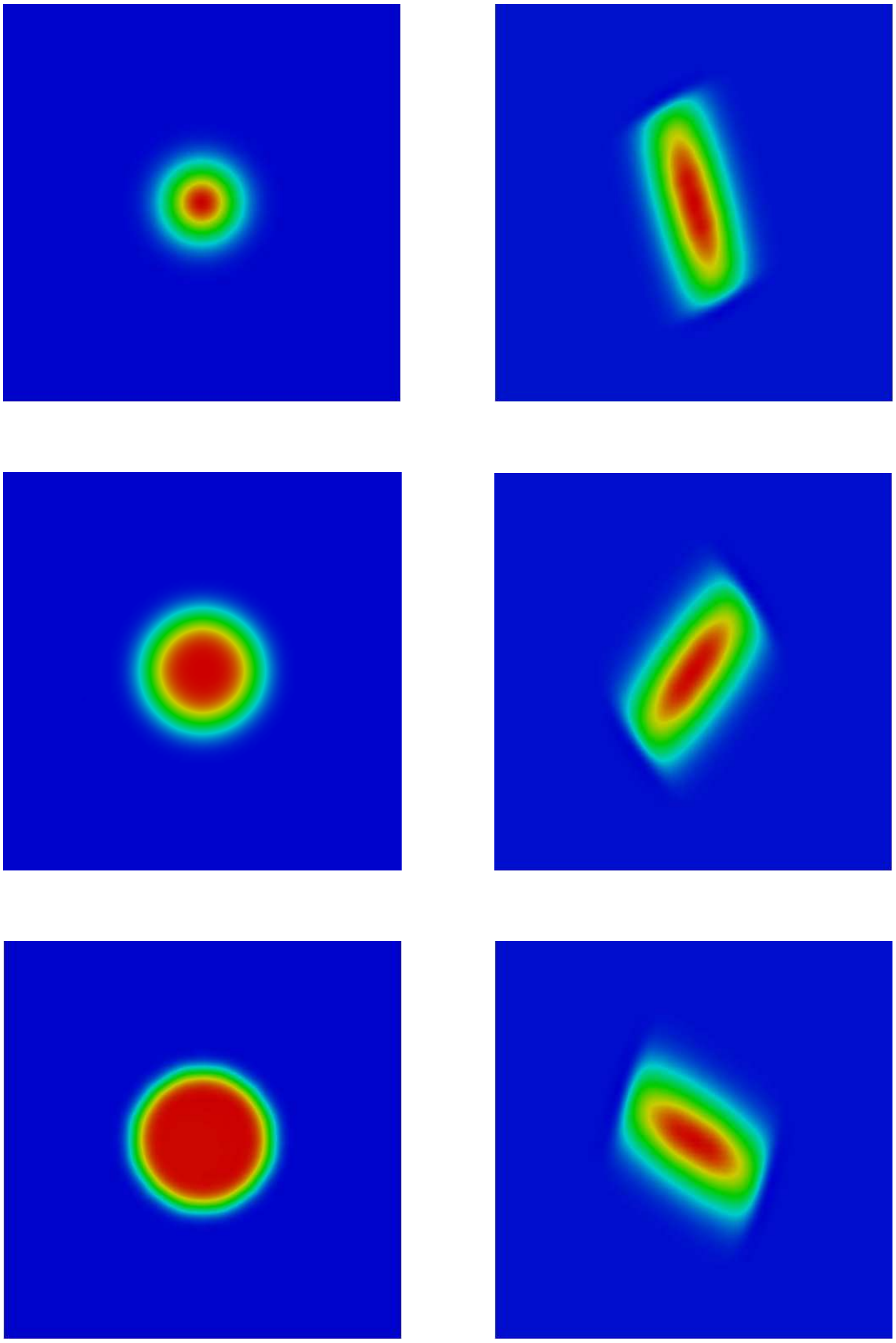

Fig. B.1 – Projection sur le plan (x, y) (à gauche) et le plan (v_x, v_y) (à droite) des valeurs de la fonction de distribution 4D pour les itérations 10 (en haut), 20 (au milieu) et 30 (en bas).

B.3 Faisceau semi-Gaussien en 4D

Ce cas test correspond à une simulation physique sur 10^{-8} secondes, soit 50 itérations de l'algorithme, d'un faisceau de particules semi-gaussien soumis à un champ électrique uniforme. Le niveau de raffinement maximum est 4, autrement dit le niveau le plus fin du maillage correspond à une grille uniforme de $32 \times 32 \times 32 \times 32$ points.

Annexe C

Plateformes d'exécution

Cluster Opteron

Le cluster d'Opterons du CECPV (Centre d'Étude du Calcul Parallèle et de la Visualisation de l'Université Louis Pasteur, Strasbourg. http://www-cecpv.u-strasbg.fr/), composé de nœuds bi-processeurs Opteron 250 cadencés à 2.4 Ghz. Chaque processeur dispose de 1024 Ko de mémoire cache de niveau L2. Chaque nœud dispose de 4 Go de mémoire et de deux interfaces Myrinet, qui permettent d'atteindre un débit théorique de 495 Mo/s. Le compilateur utilisé pour ce cluster est le compilateur Intel 9.0. Les fonctions MPI sont issues de l'implantation portable MPICH.

Cluster Itanium

Le cluster Beowulf HP du CECPV, composé de 30 nœuds bi-processeurs Itanium 2. Chaque processeur est cadencé à 1.3 Ghz et possède 32 Ko de cache de niveau L1, 256 Ko au niveau L2 et 3 Mo au niveau L3. Chaque nœud de calcul dispose de 8 Go de mémoire (partagée entre les deux processeurs du nœud) et l'ensemble est interconnecté par un réseau Myrinet offrant un débit théorique de 200 Mo/s. Le compilateur utilisé pour ce cluster est le compilateur Intel 9.0. Les fonctions MPI sont issues de l'implantation portable MPICH.

Origin 3800

La machine SGI Origin 3800 du CINES (Centre Informatique National de l'Enseignement Supérieur, Montpellier, http://www.cines.fr/) formé de 256 processeurs $R14000$ cadencés à 500 Mhz disposant chacun localement de 500 Mo de mémoire. La mémoire est virtuellement partagée par une architecture NUMA-cc.

PC linux

Un PC sous linux muni d'un processeur Athlon 64 DualCore 4800+, cadencé à $2,4$ GHz, possédant 2 Go de mémoire RAM et 1024 Ko de mémoire cache L2. Le compilateur utilisé sur cette machine est le compilateur GNU gcc 4.1.2.

Annexe D

Code source

Le code source ($\simeq$ 15000 lignes de code) de la version 4D avec prédiction en arrière se trouve dans le dépot svn de l'équipe CALVI sur la forge INRIA à l'adresse suivante : https://gforge.inria.fr/projects/calvi/

Résumé

La compréhension de phénomènes en physique des plasmas est un thème de recherche important, en relation avec plusieurs grands projets internationaux. Lorsque le modèle cinétique est utilisé pour simuler numériquement de tels phénomènes, l'évolution des particules chargées dans le plasma est alors décrite par l'équation de Vlasov. Il s'agit d'une équation aux dérivées partielles posée dans l'espace des phases, qui compte 6 dimensions dans le cas réel. La résolution numérique de cette équation représente par conséquent une énorme quantité de données et de calculs. Cette thèse s'intéresse à la mise en oeuvre informatique de méthodes de résolution basées sur des maillages adaptatifs. Nous nous sommes principalement concentrés sur la parallélisation de ces méthodes en visant des architectures à mémoire distribuée.

À l'aide d'un premier solveur parallèle disposant de mécanismes d'une grande adaptativité, nous avons mis en évidence le surcoût lié à l'adaptation du maillage et au caractère imprévisible des communications. Nous avons ensuite utilisé une méthode d'adaptation du maillage qui nous a permis de proposer un nouveau solveur par bloc, dont la parallélisation est axée sur la régularisation de la structure de données et des communications. Le recouvrement des communications par les calculs et l'équilibrage dynamique de la charge nous permettent alors d'obtenir un code adaptatif et parallèle efficace pour un espace des phases à 4 dimensions, résultat qui n'a été obtenu jusqu'à présent que pour des architectures à mémoire partagée.

Mots clés : parallélisation, maillage adaptatif, équilibrage de charge, recouvrement des communications, MPI, méthodes numériques, schéma semi-Lagrangien, équation de Vlasov

Abstract

Understanding phenomena in plasma physics is an important research area which is associated with several international projects. When numerical simulation of these phenomena is performed by using the kinetic model, charged particles evolution is given by the Vlasov equation. It is a partial differential equation that lies in phase space, which is a 6 dimensional space in the real case. Thus, the numerical resolution of such an equation represents a huge amount of data and computations. In this thesis, we are interested in the efficient implementation of numerical methods based on adaptative meshes. More precisely, we are concerned about their parallelization for distributed memory architectures.

We have developped a first parallel solver which uses some highly adaptive mechanisms. We have shown that the overhead of such a solver is associated with the mesh adaptation mechanism and with non predictable communications. Then we have used another mesh adaptation mechanism in order to propose a new block-based solver. For the parallelization of this second solver, priority is given to regularization of both data structure and communications. We have used communication overlapping with computations and dynamic load balancing to make this adaptive solver efficient for a 4 dimensional phase space. Currently, such results were acheived only on shared memory architectures.

Keywords : parallelization, adaptive mesh, load balancing, communication overlapping, MPI, numerical methods, semi-Lagrangian scheme, Vlasov equation