Parallélisation de simulations physiques utilisant un modéle de Boltzmann mullti-phases et multi-composants en vue d'un épandage de GNL sur sol

Julien Duchateau

École doctorale SPI Lille Nord-De-France

THÈSE

pour obtenir le grade de docteur délivré par

l'Université du Littoral Côte d'Opale

Spécialité doctorale "Informatique"

présentée et soutenue publiquement par

Julien DUCHATEAU

le ** décembre 2015

Parallélisation de simulations physiques utilisant un modèle de Boltzmann multi-phases et multi-composants en vue d'un épandage de GNL sur sol

Directeur de thèse : **Christophe RENAUD**

Co-encadrant de thèse : **François ROUSSELLE**

Jury

M. Daniel MENEVEAUX,	Professeur à l'Université de Poitiers	*Rapporteur*
M. Bruno RAFFIN,	Chargé de Recherche HDR à l'INRIA de Grenoble	*Rapporteur*
M. Benjamin GRAILLE,	Maître de conférences HDR à l'Université Paris-Sud	*Examinateur*
M. Jean-Charles NOYER,	Professeur à l'Université du Littoral Côte d'Opale	*Examinateur*

Laboratoire d'Informatique, Signal et Image de la Côte d'Opale – EA 4491

Maison de la Recherche Blaise Pascal – Centre Universitaire de la Mi-Voix

50 rue Ferdinand Buisson – B.P. 719, 62228 Calais Cedex, France

Remerciements

Les travaux de ce manuscrit de thèse ont été réalisés au sein du Laboratoire d'Informatique, Signal et Image de la Côte d'Opale (LISIC) de l'Université du Littoral Côte d'Opale (ULCO). C'est pourquoi je tiens à remercier en premier lieu l'intégralité des membres du laboratoire pour leur accueil et les excellentes conditions de travail qui y règnent.

Je tiens tout particulièrement à remercier le directeur de cette thèse, Christophe RENAUD*, pour m'avoir accueilli et pour l'ensemble de l'aide qu'il a pu m'apporter pendant ces trois années de thèse malgré le peu de disponibilité qu'impose sa fonction au sein du laboratoire. Il a su me soutenir et m'encourager dans les moments difficiles que ce soit sur le plan professionnel ou personnel. Je souhaite également remercier mon encadrant de thèse, François* ROUSSELLE*, pour son aide également très précieuse et du soutien qu'il m'a apporté tout au long de cette thèse malgré un emploi du temps toujours plus chargé. Il a su m'éclairer et me donner des pistes tout au long de ce travail. Leur convivialité et leur bonne humeur au quotidien m'ont permis de m'intégrer très facilement au sein de l'équipe de recherche et de travailler dans les meilleures conditions possibles.*

Je voudrais aussi remercier le dirigeant du projet Innocold-Simulation, Gilles ROUSSEL*, pour m'avoir permis de participer à ce projet et également pour son soutien tout au long de ces trois années de travaux. J'en profite également pour remercier mes deux collègues doctorants Nicolas et Kalyan ainsi que les partenaires académiques du projet au Laboratoire de Mécanique de Lille (LML). Je tiens à remercier l'ensemble des partenaires industriels du projet qui ont financé mon travail de thèse pour ces trois années. Les réunions trimestrielles permettant la bonne conduite du projet ont été productives et ont permis de mettre en relation les méthodes de travail industrielles avec les méthodes de travail académiques.*

Je souhaite maintenant remercier les rapporteurs M. Daniel MENEVEAUX*, M. Bruno* RAFFIN*, ainsi que les examinateurs M. Benjamin* GRAILLE *et M. Jean-Charles* NOYER *pour avoir accepté de juger ce travail de thèse.*

Je voudrais remercier l'ensemble des membres du laboratoire LISIC qui m'ont permis de m'intégrer très facilement au sein du laboratoire. Je tiens également à remercier les collègues doctorants des salles B125 et A205. Je voudrais tout particulièrement remercier Christopher et Florian pour l'ensemble du soutien qu'ils ont su m'apporter pendant ces années. L'ensemble des discussions que nous avons pu avoir pendant ces années ont été très productives et ont permis de créer un esprit très chaleureux et mature. Finalement, je voudrais remercier l'ensemble de ma famille et plus particulièrement ma compagne Marie pour le soutien et les encouragements permanents qu'ils ont pu me procurer tout au long de ce travail. Il est clair que je n'aurai pas pu aller jusqu'au bout de ces travaux sans leur soutien.

Table des matières

Liste des figures

Liste des tableaux

Première partie

Introduction générale et méthode de Boltzmann sur réseau

Chapitre 1

Introduction générale

Sommaire

1.1 Description du projet Innocold-Simulation

Le GNL est un gaz naturel constitué à 90% de méthane et d'un mélange d'hydrocarbures qui a subi une liquéfaction pour faciliter son transport sur de longues distances, généralement par voie maritime à bord d'un navire spécialisé que l'on appelle « Méthanier ». En passant d'un état gazeux à l'état liquide à -163°C, son volume est divisé par 600. Son utilisation domestique ou industrielle nécessite finalement que le GNL repasse à l'état gazeux. Un terminal méthanier est actuellement en construction à proximité du port de Dunkerque et sa mise en fonctionnement est prévue pour décembre 2015. Ces ports sont généralement connectés à un ou plusieurs gazoducs de grande taille, ou équipés en systèmes de traitement et de transport du GNL, avec notamment des réservoirs de stockage temporaire, ainsi que des unités de gazéification (par réchauffage du gaz qui peut alors être injecté dans les tuyaux des réseaux de distribution). L'exploitation d'une très grosse quantité de GNL nécessite l'évaluation du risque, car le GNL est un gaz inflammable voire explosif au-delà d'un certain niveau de concentration dans l'air. Maîtriser le risque passe alors par une meilleure connaissance du comportement du GNL. Les accidents sont généralement imprévisibles, mais l'expérience, la connaissance physique et la modélisation de son comportement ont pour objectif de mieux prévoir l'impact d'un scénario nouveau ou standard, voire de mieux prévenir les accidents ou de se protéger des conséquences. C'est dans ce cadre que le projet Innocold-Simulation s'inscrit.

L'objectif du projet vise à construire un modèle dynamique de simulation de dispersion d'un nuage de gaz de GNL lors d'un scénario d'épandage de GNL à l'état liquide sur le sol ou dans l'eau du bassin portuaire du terminal méthanier. L'épandage du GNL à l'état liquide constitue alors un terme source pour la dispersion dans l'atmosphère. Ce projet est découpé en trois parties. Il comprend les travaux effectués par trois équipes de recherche, chacune en charge d'un sujet d'étude, mais aussi d'une spécialité scientifique propre d'un projet global transversal. Ces trois spécificités scientifiques complémentaires sont l'Informatique, l'Automatique et la Mécanique des fluides.

La première partie du projet global concerne la modélisation d'un terme source pour l'épandage de GNL, son interaction avec plusieurs composants physiques (tels que l'air et l'eau) ainsi que l'évaporation en général et en particulier la transition rapide de phase lors

d'un épandage massif dans l'eau de mer. Ce noyau de simulation nécessite une connaissance importante des phénomènes physiques sous-jacents et impose des développements théoriques spécifiques, liés aux grandes différences de densités des composants et aussi à la présence de plusieurs fluides pouvant interagir entre eux. La validation des modèles représente également une préoccupation constante de ce travail de modélisation.

La seconde partie vise plus particulièrement l'implémentation et l'optimisation des calculs sur des structures multiprocesseurs du noyau de simulation issu de la modélisation du terme source sur de gros domaines de simulation liés à une géométrie complexe. La réalisation de simulations physiques complexes sur de grandes installations, telles qu'un terminal méthanier, soulève de nombreuses problématiques sur le plan informatique. La simulation de dynamique des fluides implique généralement une grosse quantité de calculs ainsi qu'une quantité extrêmement importante de mémoire à gérer. L'optimisation et la parallélisation des algorithmes de calculs sont donc des éléments indispensables afin de pouvoir réaliser ces simulations dans des temps acceptables. C'est dans cette seconde partie que s'inscrivent les travaux présentés dans ce manuscrit de thèse.

Le dernier sujet a en charge d'estimer la dispersion atmosphérique du GNL gazeux au sein de son environnement proche. La simulation doit se baser sur les résultats obtenus par la simulation d'épandage du GNL sur un sol ou dans l'eau de mer et ainsi permettre de déterminer le périmètre de danger en cas de dispersion de méthane gazeux dans l'atmosphère, qui devient inflammable au-delà d'une concentration supérieure à 5% environ. Comme le premier sujet, il impose une connaissance théorique physique importante.

1.2 Financeurs et partenaires du projet

Ce projet se déroule dans le cadre de la construction du terminal méthanier de Dunkerque qui est actuellement en voie d'achèvement. Il s'agit du deuxième plus gros chantier français en cours et il devrait être opérationnel dès la fin de l'année 2015. Les industriels à l'origine du projet et liés à la construction du site du terminal (Total, Dunkerque LNG et Sofregaz) ont donc financés les trois sujets décrits précédemment.

FIGURE 1.1 – Financeurs du projet Innocold-Simulation

Les partenaires académiques du projet sont composés du laboratoire LISIC (Laboratoire d'Informatique, Signal et Image de la Côte d'Opale) de l'Université du Littoral Côte d'Opale et le Laboratoire LML (Laboratoire de Mécanique de Lille) de l'Université de Lille 1. Les travaux de thèse de ce manuscrit ainsi qu'une thèse de modélisation physique ont été effectués au sein du LISIC, tandis que le troisième sujet lié à la dispersion se fait au LML.

Ce travail de thèse s'intègre dans une équipe composée de Christophe Renaud, Professeur en Informatique et directeur du LISIC et de François Rousselle, Maître de Conférences en Informatique. Le projet Innocold-Simulation est dirigé par Gilles Roussel, Maître de Conférences habilité à diriger la recherche en Automatique au LISIC et directeur de thèse du premier projet de modélisation physique.

1.3 Contexte et objectifs du travail de thèse

Le projet global comprend trois sujets dont les études sont fortement corrélées. En effet, la modélisation du terme source a un impact direct sur les deux autres sujets. L'optimisation et la parallélisation des calculs du terme source sont fortement dépendants de l'évolution de la modélisation. De la même façon, l'étude de la dispersion dépend essentiellement des résultats issus du terme source. De plus, les travaux ont en commun les méthodes numériques de simulation appelés modèles de Boltzmann sur réseau (LBM). Il existe également une volonté commune au sein du projet d'implémenter les algorithmes sur une cible de calculs de type SIMD. Même si les travaux de ce manuscrit se concentrent sur les calculs du terme source, ils peuvent être également applicables pour l'étude de la dispersion, voire pour d'autres champs d'applications.

Les simulations sur de grandes installations industrielles sont une première particularité de ce travail de thèse. En effet, l'utilisation d'une méthode de dynamique des fluides (CFD) pour réaliser des simulations physiques complexes implique de manipuler une importante quantité de données. Réaliser ces simulations sur de grandes installations soulève donc des problématiques de gestion de la mémoire informatique pour rendre ces simulations faisables sur une architecture de calculs limitée. La manipulation de plusieurs composants physiques implique également des problématiques informatiques sur le plan de la mémoire et du calcul. L'utilisation de plusieurs composants physiques à l'intérieur du noyau de simulation est en effet nécessaire, afin d'obtenir une modélisation plus réaliste et plus précise du comportement du GNL en interaction avec l'air ambiant ou encore avec l'eau de mer. Le premier objectif de ce travail de thèse est d'apporter des solutions informatiques de manière à pouvoir réaliser des simulations contenant plusieurs composants physiques sur des domaines géométriques étendus. Cela soulève plusieurs problématiques liées à la gestion de la mémoire. Un premier facteur concerne la réduction de la mémoire nécessaire à la méthode de Boltzmann sur réseau. Le noyau de simulation étant plus complexe qu'un modèle de Boltzmann standard, il implique également une gestion de la mémoire pour l'intégralité du modèle. Un facteur géométrique est également à prendre en compte lors d'un écoulement. Le maillage de l'intégralité d'un terminal méthanier implique une quantité de données extrêmement importante, qui n'est pas envisageable sur une architecture de calculs limitée comme celle dont nous disposons au sein du laboratoire et qui n'est sans doute pas souhaitable non plus, au vu de la localité prévisible de l'écoulement de GNL à simuler et donc d'une grande partie des calculs à effectuer. Des solutions doivent donc être apportées de manière à réduire considérablement la quantité de données nécessaires à la simulation.

L'utilisation d'une architecture de calculs adaptée est également une problématique importante pour l'implémentation des algorithmes du projet. En effet, l'accélération des calculs est une étape importante de manière à réaliser des simulations dans des temps acceptables. L'accélération des calculs liés au modèle d'écoulement par une parallélisation efficace des algorithmes est une seconde problématique du travail de thèse. L'utilisation de processeurs graphiques commence à se répandre dans de nombreux domaines et notamment la dynamique des fluides et la méthode de Boltzmann sur réseau. Réaliser des

calculs avec les processeurs graphiques soulève toutefois des problématiques importantes pour obtenir de bonnes performances. La complexité de ces problématiques s'accroît, dès lors que l'on envisage d'utiliser plusieurs processeurs graphiques, en vue d'accroître les performances des simulations. Les travaux réalisés dans le cadre de cette thèse auront ainsi pour objectif de proposer des solutions à ces problématiques.

1.4 Plan de la thèse

Ce manuscrit est découpé en plusieurs parties regroupant l'ensemble des travaux effectués pendant les trois années de thèse. L'évolution chronologique des travaux a conduit au plan suivant.

La première partie de ce mémoire de thèse inclut cette introduction concernant le projet Innocold-Simulation et traite de la méthode de Boltzmann sur réseau, commune à l'ensemble du projet. Un bref historique concernant la modélisation de Boltzmann est dans un premier temps présenté afin de bien comprendre l'origine de la méthode. En effet, la méthode de Boltzmann sur réseau tient son origine de plusieurs disciplines scientifiques comme la physique statistique, l'automatique et l'informatique. La mise en place progressive de la méthode de Boltzmann sur réseau usuellement utilisée dans des cas bidimensionnels et tridimensionnels est ensuite décrite. Plusieurs opérateurs de collision-propagation liés à la méthode de Boltzmann sur réseau sont présentés. La discrétisation et la paramétrisation des simulations liées à la modélisation de Boltzmann sont également abordées. Les conditions aux bords du domaine de simulation, ainsi qu'aux obstacles du domaine, sont ensuite définies. Une dernière partie traite de modélisations plus complexes, multi-phases et multi-composants. Des éléments de modélisation concernant l'interaction entre plusieurs fluides ou l'interaction entre plusieurs états physiques (phase) d'un même fluide sont d'abord décrits. L'algorithme résultant du travail de modélisation physique est finalement introduit.

La seconde partie de ce travail de thèse concerne une étude de la réduction de mémoire pour la méthode de Boltzmann sur réseau, ainsi que la mise en place d'une première parallélisation du noyau de simulation à l'aide d'un processeur central multi-cœurs. Les

techniques de réduction de mémoire liées à la modélisation de Boltzmann sont dans un premier temps introduites. En effet, l'utilisation d'une méthode permettant de réduire la quantité de mémoire est un élément important pour la réalisation de simulations sur de grandes installations. Le choix de cette méthode doit également être pris en compte de manière à choisir la solution la plus efficace. Le second aspect traité dans ce chapitre concerne la parallélisation de codes de simulation. La mise en place d'une parallélisation hybride SIMD du noyau de simulation par l'utilisation combinée de la librairie OpenMP et d'instructions SSE sur une architecture multi-cœurs est finalement étudiée.

La troisième partie de ce manuscrit concerne l'ensemble des travaux réalisés sur une architecture composée de plusieurs processeurs graphiques. Elle aborde dans un premier chapitre des problématiques de parallélisation sur processeurs graphiques (ou GPU, pour *Graphic Processing Unit*). Ces processeurs, utilisables dans le cadre de calculs génériques, fonctionnent en effet d'une manière totalement différente des processeurs centraux utilisés de manière usuelle. Cela implique également un modèle de programmation différent de la programmation classique. Une compréhension en profondeur de l'architecture des processeurs graphiques et de l'outil de programmation CUDA est dans un premier temps abordée. L'utilisation des processeurs graphiques pour réaliser du calcul générique a attiré l'attention de beaucoup de chercheurs dans le monde, notamment ceux liés à la modélisation de Boltzmann. Les travaux existants liés à la parallélisation de méthodes de Boltzmann sur réseau par l'utilisation de processeurs graphiques sont donc également étudiés. La mise en place d'une parallélisation du noyau de simulation sur une architecture composée de plusieurs processeurs graphiques est finalement abordée.

Le second chapitre de cette partie s'intéresse à la mise en place d'une méthode de maillage progressif au sein du noyau de simulation, utilisable pour différentes architectures. Cette méthode a pour vocation de mailler progressivement le domaine de simulation en fonction de la progression de la simulation. La définition d'un critère permettant de mailler progressivement le domaine est dans un premier temps abordée. Le choix du critère est une étape primordiale afin d'obtenir un maillage progressif qui soit efficace. L'intégration de cette méthode au sein d'une parallélisation sur un processeur graphique est ensuite étudiée. Le passage à une architecture composée de plusieurs processeurs gra-

phiques implique de nouvelles problématiques pour la méthode de maillage progressif, notamment en ce qui concerne le transfert de données. L'intégration de cette méthode sur ce type d'architecture est donc finalement traitée.

Le dernier chapitre de cette partie traite de cas de simulations où la mémoire nécessaire pour la simulation est trop importante pour être stockée sur la mémoire des différents processeurs graphiques. En effet, les processeurs graphiques disposent généralement d'une quantité de mémoire très limitée et nettement inférieure à la mémoire liée au processeur central de la machine. La réalisation de simulations sur de grandes installations demande pourtant une grande quantité de ressources informatiques. Des solutions doivent alors être apportées afin d'être en mesure de pouvoir réaliser les simulations dans les meilleures conditions possibles. Un mécanisme d'échanges avec la RAM du processeur central est donc étudié. L'intégration d'un algorithme de compression de données est finalement discutée de manière à réduire la quantité de données à transférer entre les GPUs et le CPU.

Chapitre 2

La méthode de Boltzmann sur réseau

Sommaire

2.1 Introduction

Comme la plupart des outils numériques classiques permettant de résoudre numériquement des problèmes de mécanique des fluides (méthode aux différences finies, méthode des éléments finis, méthode des volumes finis, ...), la méthode de Boltzmann sur réseau est une technique intéressante permettant de simuler des phénomènes physiques complexes. Contrairement à ces outils, la méthode de Boltzmann sur réseau ne s'intéresse toutefois pas directement à l'évolution globale des quantités macroscopiques d'un fluide (masse volumique, vitesse, ...). Elle s'intéresse plutôt au comportement microscopique du fluide et à l'évolution des particules qui le constituent. En effet, l'équation de Boltzmann est capable de décrire l'évolution spatio-temporelle d'une fonction représentant la distribution de particules ayant une vitesse donnée à une position et un temps fixés.

À la différence des modèles traditionnels, les quantités macroscopiques d'un fluide peuvent être exprimées à l'aide de cette unique fonction de distribution représentant la distribution des particules au sein du fluide. Ainsi, cette méthode permet de résoudre des équations aux dérivées partielles complexes telles que l'équation de Navier-Stockes (NS) à partir de calculs arithmétiques simples. De plus, les conditions aux limites peuvent généralement être exprimées par des règles relativement simples, permettant alors de rendre la méthode applicable sur des géométries complexes. Finalement, la méthode de Boltzmann sur réseau est également capable de modéliser des écoulements turbulents, des phénomènes où plusieurs états physiques (phases du fluide) interagissent ou encore plusieurs composants physiques en interaction.

2.2 Origine de la méthode

La construction de la méthode de Boltzmann sur réseau peut se résumer en deux étapes indépendantes :

1. Le développement de la physique statistique ;
2. L'apparition des automates cellulaires.

Les premiers éléments de la physique statistique sous la forme de la théorie cinétique des gaz remontent au milieu du XIX ^{e} siècle. Sur le plan algorithmique en revanche, il faut

attendre la seconde moitié du XX ^{e} siècle et l'apparition des outils informatiques et des automates cellulaires.

2.2.1 Théorie cinétique des gaz

Selon Wikipedia [Wik15] : « la physique statistique (appelée aussi « thermodynamique statistique ») fut introduite initialement sous la forme de la théorie cinétique des gaz à partir du milieu du XIX^{e} siècle, principalement dans les travaux de l'Américain Gibbs, des Anglais Kelvin et Maxwell et de l'Autrichien Boltzmann ». Elle ne s'intéresse toutefois pas à l'évolution globale des quantités macroscopiques d'un fluide (masse volumique, vitesse, pression, ...) mais plutôt à son comportement à l'échelle microscopique. « Cette première approche visait à proposer un modèle simple de la matière à l'échelle atomique, et en particulier des collisions entre atomes ou molécules, pour reproduire le comportement de certaines quantités macroscopiques ».

Boltzmann propose en 1872 son équation (équation 2.1), basée sur l'utilisation d'une fonction de distribution de particules, représentant la répartition des particules ayant une vitesse donnée à une position et un temps fixés.

$$\frac{\partial f}{\partial t} + c \cdot \nabla_x f + \frac{F_i}{m} \cdot \nabla_c f = \left(\frac{\partial f}{\partial t}\right)_{coll} \tag{2.1}$$

La fonction f représente la probabilité de trouver une particule à un instant et une position donnés. Cette fonction est donc dépendante du temps t, de la position x et de la vitesse c. Le terme F_i correspond aux forces agissant sur les particules et m à la masse moléculaire du gaz. Le membre de droite de l'équation 2.1 est appelé opérateur de collision. Il représente les collisions entre les différentes particules du fluide. Boltzmann n'avait à l'époque qu'une vague idée de cet opérateur et ne considérait que les collisions entre deux particules. Si le terme de collision est nul, les particules sont donc simplement propagées et subissent l'action de la force **F** présente dans la partie gauche de l'équation. En 1970, les travaux de Cohen et Dorfman [BCT73] ont démontré que l'équation de Boltzmann avec un opérateur de collision binaire ne pouvait pas s'appliquer ni aux liquides ni aux gaz.

Les travaux de Boltzmann n'ont pu être validés qu'après sa mort en 1906. Ce n'est que dans les années 1920 que les travaux de l'astronome anglais Sydney Chapman et d'un mathématicien suisse David Enskog s'intéressent à des collisions plus complexes intégrant plus de deux particules [Cer75]. Enskog eut l'idée d'effectuer un développement de la fonction de distribution de l'équation de Boltzmann [Bru13]. Chapman approfondit l'idée d'Enskog et choisit de considérer les particules comme des sphères pour prendre en compte des collisions complexes [Mar08]. À partir de 1921, les travaux de Chapman et Enskog ont été validés expérimentalement et ont permis d'établir un lien entre l'équation de Boltzmann et les équations de Navier-Stokes.

L'opérateur mis en place par Chapman et Enskog reste toutefois très spécifique. Une trentaine d'années sont nécessaires pour qu'un opérateur de collision simple soit mis en place. Ce sont les mathématiciens Bhatnagar, Gross et Krook qui le dévoilent en 1954 [BGK54]. Ce modèle est établi sur des développements théoriques très complexes. L'idée développée est que la collision des particules peut être décrite comme la relaxation en un temps donné des particules vers un état d'équilibre donné. L'opérateur de collision peut ainsi être écrit sous une forme simple qui porte le nom d'opérateur BGK. La méthode de Boltzmann sur réseau aurait pu voir le jour à cette époque. Bien que les éléments théoriques de la méthode soient posés, certaines notions propres à l'informatique sont encore inconnues à cette époque. C'est par exemple le cas du concept de simulation physique discrète. Il est donc encore trop tôt pour que la méthode de Boltzmann sur réseau puisse réellement voir le jour. L'ère de la simulation numérique et le développement intensif des automates cellulaires ont permis de changer considérablement les choses.

2.2.2 Automates cellulaires et modèle de gaz sur réseau

La notion de discrétisation apparaît dans la seconde moitié du XX^{e} siècle avec l'apparition des premières machines de calculs. Les premiers automates cellulaires sont mis en place pour la première fois dans les années 1950 par Von Neumann et Ulam [VNB+66]. Von Neumann cherche alors à savoir s'il est possible de concevoir une machine capable de s'auto-reproduire, c'est-à-dire capable de reproduire une machine de complexité équivalente par elle-même. Ulam propose alors de considérer un espace composé de cellules

réparties de manière uniforme dans lequel l'évolution de ces cellules est définie par l'état des cellules à son voisinage [L.10]. Les mêmes règles sont appliquées à chaque itération pour l'intégralité des cellules composant le système, produisant une nouvelle génération de cellules calculées en fonction de la génération précédente. Un comportement complexe global peut alors être réalisé à partir de règles simples. La figure 2.1 montre un premier exemple d'automate cellulaire. L'utilisation d'une règle simple et une initialisation adaptée des cellules permettent d'obtenir ici une structure fractale. L'initialisation des cellules et la définition des règles de calculs sont donc des éléments essentiels pour les automates cellulaires.

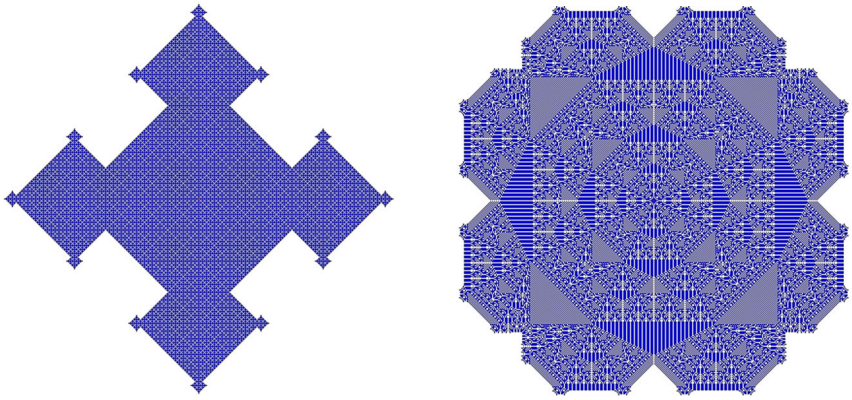


FIGURE 2.1 – Évolution d'un automate cellulaire engendrant une structure fractale.

L'évolution de la puissance des ordinateurs a permis d'utiliser les automates cellulaires dans de nombreux autres domaines. En 1970, la mathématicien Conway propose le célèbre « jeu de la vie » [Con70]. La figure 2.2 montre également un exemple d'automate cellulaire permettant de reproduire de manière très simplifiée un feu de forêt. Aucune règle physique n'intervient toutefois dans la définition de cet automate.

En partant du principe qu'un automate cellulaire peut s'appliquer à des systèmes complexes, les recherches se sont également orientées du côté de la simulation physique. C'est en 1973 que le développement des premiers automates cellulaires appliqués à la mécanique des fluides feront le lien entre les discrétisations de l'espace et du temps avec l'apparition des modèles de gaz sur réseau (LGCA pour Lattice Gas Cellular Automata). Le premier modèle fut développé par Hardy, Pomeau et Pazzis (modèle HPP) en 1973

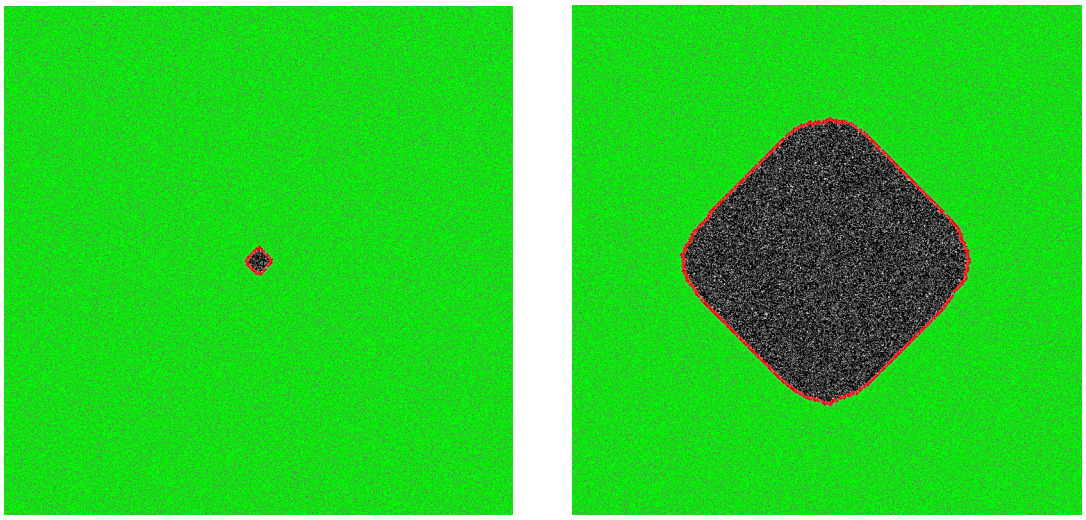


FIGURE 2.2 – Évolution d'un automate cellulaire simulant l'évolution d'un feu de forêt [LPBC].

[HPDP73]. Leur approche utilise une dynamique restreinte : seules quelques vitesses prédéfinies sont autorisées pour les particules. Autrement dit, un ensemble de particules peut suivre des trajectoires constantes et les particules peuvent interagir entre elles uniquement lorsque leurs trajectoires se croisent. Dans les premiers stades de développement, des lois d'interaction explicites ont été mises en place pour les collisions des particules [L.10]. Pour chaque unité de temps, une particule, en fonction de sa vitesse, se déplace d'une cellule à une autre de la grille régulière et subit des collisions avec d'autres particules. Une simplification supplémentaire, propre aux gaz sur réseau et appelée principe d'exclusion, est considérée. Cela signifie qu'à un temps donné, il ne peut pas y avoir plusieurs particules partageant la même position et la même vitesse. Ce modèle ne permet toutefois pas encore de retrouver les équations de Navier-Stokes. Il fut suivi en 1986 par le modèle bidimensionnel FHP (Frisch, Hasslacher et Pomeau) [FHP86] et du modèle 3D FCHC (pour Face Centered HyperCube), proposé par D'Humières [dLF86]. Ces modèles permettent de retrouver les équations de Navier-Stokes, mais des inconvénients importants sont à noter, comme la présence de bruit dû au caractère booléen des automates cellulaires ou encore le choix d'une symétrie de réseau adaptée pour des cas à trois dimensions [Suc01]. Enfin, ces modèles de gaz sur réseau sont victimes d'un manque de flexibilité dans la définition des paramètres physiques.

2.2.3 Méthode de Boltzmann sur réseau

À la fin des années 80, les travaux sur les gaz sur réseau se dirigent désormais vers l'équation de Boltzmann. Une étude reprenant l'ensemble des modèles de gaz sur réseau par Wolfram en 1986 [W+86], puis par Frisch l'année suivante [FdH+87], permet de faire le lien entre les gaz sur réseau et l'équation de Boltzmann. Les variables booléennes sont alors remplacées par des variables réelles et les distributions de Maxwell-Boltzmann sont utilisées pour l'état d'équilibre [Mar08]. Ces modifications entraînent la suppression des inconvénients majeurs des gaz sur réseau. On peut alors parler des premiers modèles de Boltzmann sur réseau, introduit par McNamara et Zanetti en 1988 [MZ88].

L'apparition des modèles de Boltzmann sur réseau actuels se fera progressivement avec la linéarisation de l'opérateur de collision par Higuera et Jiménez en 1989 [HJ89] et enfin, par l'utilisation de l'opérateur BGK par Koelman en 1991, puis par Chen [CCM92] et Qian en 1992 [QdL92]. L'équation de Boltzmann sur réseau est alors née et, au début des années 90 elle est présentée comme une amélioration des gaz sur réseau. On peut alors remonter aux équations de Navier-Stokes et à l'équation de Boltzmann continue par un développement de Chapman-Enskog. Pourtant, en 1997, He & Luo [HL97] montrent que l'équation de Boltzmann sur réseau peut être dérivée directement de l'équation de Boltzmann continue. On parle alors de dérivation *a priori* de la méthode de Boltzmann sur réseau [Mar08]. Cette dérivation *a priori* définit un cadre théorique pour la méthode de Boltzmann sur réseau sans passer par les gaz sur réseau.

L'ensemble des aspects historiques développés dans cette section montre bien la longue évolution qui s'est écoulée entre l'apparition de l'équation de Boltzmann en 1872 et l'équation de Boltzmann sur réseau en 1992, soit plus d'un siècle de recherches. Aujourd'hui, la méthode de Boltzmann sur réseau est utilisée dans de nombreux domaines d'applications [CLH+07] [Thü07] tels que l'industrie avec des simulations aérodynamiques ou acoustiques dans le domaine de l'automobile, le ferroviaire ou l'aéronautique. Des applications en synthèse d'images ont aussi pu voir le jour comme par exemple de la simulation numérique de flammes. Son utilisation est également possible au travers de logiciels tels que le logiciel de modélisation Blender ou encore PowerFLOW.

2.3 Discrétisation et paramétrisation de la méthode de Boltzmann sur réseau

Dans cette partie, la méthode de Boltzmann sur réseau est définie dans un cadre général, où le pas de discrétisation spatiale Δx et le pas de discrétisation temporelle Δt de la méthode sont quelconques. Les développements théoriques qui ont conduit à la méthode de Boltzmann sur réseau ne sont pas traités dans cette section. Les discrétisations associées à la méthode de Boltzmann sur réseau sont dans un premier temps présentées. La méthode de Boltzmann sur réseau utilisant principalement des valeurs non dimensionnées, une paramétrisation est nécessaire pour retrouver les valeurs réelles correspondant aux fluides à partir des valeurs issues de la simulation. Les équations discrètes de la méthode de Boltzmann sur réseau sont ensuite présentées. Finalement, les aspects de modélisation conduisant à pouvoir simuler plusieurs composants physiques ainsi que plusieurs états physiques en interaction sont introduits.

2.3.1 Développement dans un cadre bidimensionnel

Dans la méthode de Boltzmann sur réseau, le mouvement des particules est restreint à un nombre limité de directions. Le cas considéré ici concerne un modèle bidimensionnel à neuf vitesses discrètes, appelé communément modèle D2Q9. Dans cette configuration, l'espace est discrétisé par un réseau carré de pas spatial Δx possédant des vecteurs de directions c_i de $\mathbb{R}^2$ représentés par la figure 2.3 et définis de la manière suivante :

$$c_i = \begin{cases} (0,0)^t, & i = 0 \\ (\cos[(i-1)\pi/2], \sin[(i-1)\pi/2])^t, & i = 1,...,4 \\ (\cos[(2i-9)\pi/4], \sin[(2i-9)\pi/4])^t, & i = 5,...,8 \end{cases} \tag{2.2}$$

Il s'agit du modèle le plus utilisé dans un cadre à deux dimensions, car il considère l'ensemble des plus proches voisins. Le domaine de simulation est ainsi découpé selon une grille régulière composée d'un ensemble de cellules sur lesquelles les calculs de la méthode de Boltzmann sur réseau s'effectuent. Cette discrétisation peut également se généraliser au cas tridimensionnel.

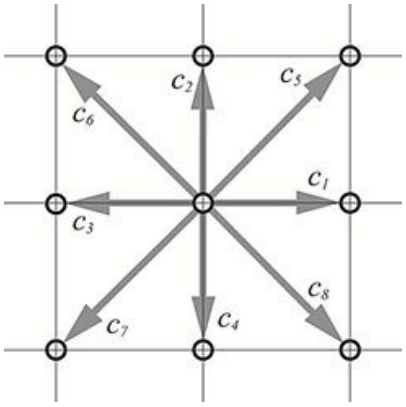


FIGURE 2.3 – Schéma de discrétisation D2Q9.

2.3.2 Généralisation aux cas à trois dimensions

Dans un cas à trois dimensions, l'espace est de la même façon découpé selon un pas de discrétisation Δx composant ainsi la grille de simulation sous la forme d'un réseau composé de cellules cubiques. Dans un cas tridimensionnel, le nombre de directions associé à une cellule du réseau est évidemment plus important que dans un cas à deux dimensions. Le modèle le plus couramment utilisé compte dix-neuf directions de propagation (modèle D3Q19). Les vecteurs de liaisons c_i, représentés par la figure 2.4 sont alors définis de la façon suivante :

$$c_i = \begin{cases} (0,0,0)^t, & i = 0 \\ (\pm 1,0,0)^t, (0,\pm 1,0)^t, (0,0,\pm 1)^t, & i = 1,...,6 \\ (\pm 1,\pm 1,0)^t, (\pm 1,0,\pm 1)^t, (0,\pm 1,\pm 1)^t, & i = 7,...,18 \end{cases} \tag{2.3}$$

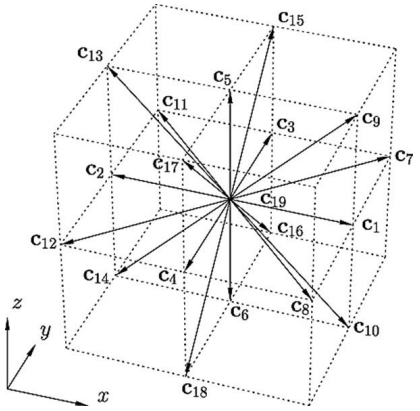


FIGURE 2.4 – Schéma de discrétisation D3Q19.

D'autres alternatives existent comme, par exemple, les modèles D3Q15 et D3Q27 comportant respectivement quinze et vingt-sept directions discrètes (Figure 2.5). Cependant, l'utilisation privilégiée d'une discrétisation à dix-neuf vitesses discrètes repose sur deux raisons majeures. La première est qu'en terme de stabilité le modèle D3Q15 se révèle être moins performant [KKH+99]. En revanche, le modèle D3Q19 s'avère généralement aussi efficace qu'un modèle D3Q27. L'apparition du modèle D3Q19 a en effet mis en avant que l'utilisation de vingt-sept vitesses discrètes n'est pas indispensable pour obtenir des performances similaires [WMK04]. Par conséquent, afin de ne pas augmenter inutilement le coût calculatoire, la modélisation D3Q19 est généralement choisie.

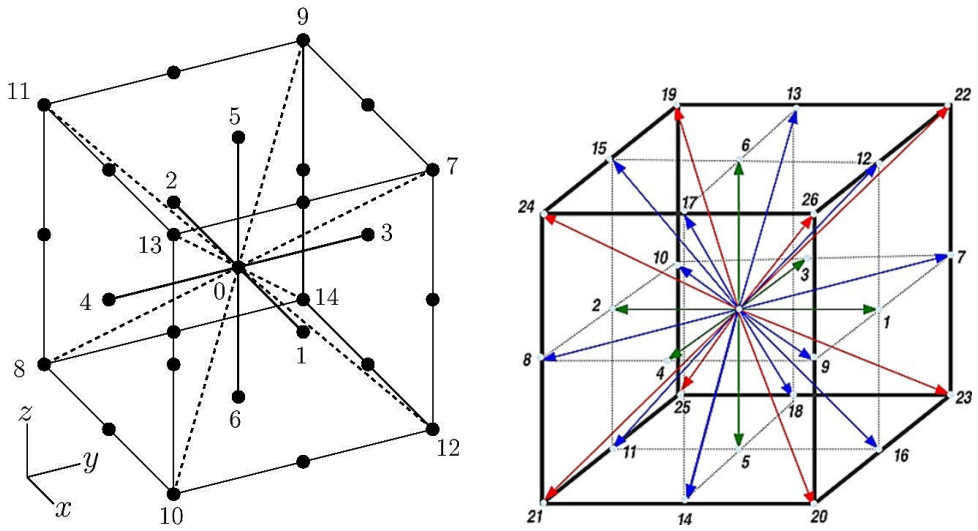


FIGURE 2.5 – Schémas de discrétisation D3Q15 et D3Q27.

2.3.3 Opérateur de collision à temps de relaxation simple

L'opérateur de collision pour la méthode de Boltzmann est en charge de l'évolution des particules au sein du réseau. La densité de particules à une position x du réseau, pour le vecteur de liaison c_i et au temps discret t est noté $f_i(x,t)$. L'évolution des particules entre les instants t et $t+\Delta t$ est composée de deux étapes :

— Une étape de collision locale en espace représentant les collisions entre l'ensemble des particules au sein d'une cellule ;

— Une étape de propagation vers les voisins proches définis par les vecteurs de liaison.

Ces deux étapes sont décrites par l'équation de Boltzmann discrète suivante :

$$f_i(x+c_i,t+\Delta t) = f_i(x,t) + \Omega(f_i(x,t)) \tag{2.4}$$

Ces deux étapes sont à réaliser pour l'ensemble des directions discrètes définies par la discrétisation (équations 2.2 et 2.3). L'opérateur de collision Ω est généralement défini suivant l'approximation de type BGK [CFL09], c'est-à-dire en réalisant une linéarisation autour d'un état d'équilibre :

$$\Omega(f_i(x,t)) = -\frac{\Delta t}{\tau}(f_i(x,t) - f_i^{eq}(x,t)) \tag{2.5}$$

Le retour à l'équilibre est caractérisé par un temps de relaxation τ vérifiant la condition de stabilité du schéma d'Euler explicite :

$$0 \leq \frac{\Delta t}{\tau} \leq 2 \tag{2.6}$$

Les fonctions de distribution à l'équilibre f_i^{eq} intervenant dans l'équation 2.5 sont définies à l'aide de l'équation suivante :

$$f_i^{eq}(x,t) = w_i \rho(x,t) \left(1 + \frac{c_i \cdot u(x,t)}{c_s^2} + \frac{1}{2}\left(\frac{c_i \cdot u(x,t)}{c_s^2}\right)^2 - \frac{u(x,t)^2}{2c_s^2}\right) \tag{2.7}$$

où :

$$w_i = \begin{cases} 4/9, & i = 0 \\ 1/9, & i = 1,...,4 \\ 1/36, & i = 5,...,8 \end{cases} \tag{2.8}$$

$$w_i = \begin{cases} 1/3, & i = 0 \\ 1/18, & i = 1,...,6 \\ 1/36, & i = 7,...,18 \end{cases} \tag{2.9}$$

pour le schéma D2Q9 (équation 2.8) et D3Q19 (équation 2.9) respectivement. c_s correspond à une vitesse de référence définie pour le réseau de la méthode de Boltzmann.

La densité $\rho(x,t)$ et la vitesse $u(x,t)$ correspondent aux quantités macroscopiques conservées du fluide dans un cas isotherme. Ces grandeurs sont déterminées à l'aide des fonctions de distribution f_i de la manière suivante :

$$\rho(x,t) = \sum_i f_i(x,t) \tag{2.10}$$

$$\rho(x,t)u(x,t) = \sum_i f_i(x,t)c_i \tag{2.11}$$

La figure 2.6 résume les étapes de propagation et de collision de la modélisation de Boltzmann sur réseau pour une cellule entre un instant t et $t + \Delta t$.

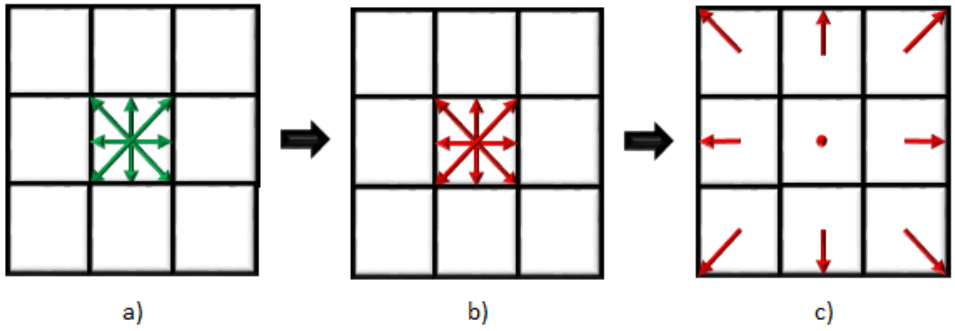


FIGURE 2.6 – Étapes de collision et de propagation pour une cellule : a) Sélection de la cellule à calculer, b) Collision des f_i de manière locale à la cellule, c) Propagation des f_i vers les plus proches voisins

L'opérateur de collision à temps de relaxation simple n'est toutefois pas la seule option pour modéliser les collisions entre les particules composant le fluide à simuler. D'autres approches ont été étudiées en littérature.

2.3.4 Opérateur de collision à temps de relaxation multiples

L'opérateur de collision à temps de relaxation multiples est apparu au début des années 1990 [d'H94]. Dans le modèle présenté précédemment, que l'on appellera modèle LBM-BGK, l'opérateur de collision associé à l'équation discrète de Boltzmann impose un temps de relaxation identique à chaque grandeur physique. Ainsi, chaque moment de la fonction de distribution requiert le même temps caractéristique pour revenir à son état d'équilibre. Cela peut paraître restrictif car dans la réalité, des phénomènes physiques différents peuvent retrouver l'équilibre en des temps différents. Ainsi, en parallèle du modèle LBM-BGK, D'Humières définit un nouveau modèle basé sur l'utilisation de plusieurs temps de relaxation [d'H02]. Ce modèle, que nous désignerons dans la suite par

LBM-MRT (MRT pour Multiple Relaxation Time), considère donc que chaque moment (c'est-à-dire les grandeurs physiques) possède un temps caractéristique propre. Le vecteur des fonctions de distribution $f = \left(f_i\right)_{i=0,\ldots,18}$ est alors associé au vecteur des moments $m = \left(m_i\right)_{i=0,\ldots,18}$. Autant de moments sont nécessaires que de fonctions de distribution. Un passage de l'espace des fonctions de distributions à l'espace des moments est alors assuré par l'équation suivante :

$$m = Mf \tag{2.12}$$

où M est une matrice de passage de taille 9x9 ou 19x19 selon la dimension. Les matrices de passage pour le schéma à temps de relaxation multiples sont données en annexe. L'équation de Boltzmann sur réseau associé au modèle LBM-MRT peut finalement être écrite de la façon suivante :

$$m(x+c,t+\Delta t) = m(x,t) - M^{-1}S[m(x,t)) - m^{eq}(x,t)] \tag{2.13}$$

avec S une matrice diagonale contenant les différents temps de relaxation. Les coefficients de la matrice S sont alors calculés de manière à avoir une stabilité optimale [d'H02]. Elle aura donc la structure suivante :

$$S = diag[0, s_1, s_2, 0, s_4, 0, s_4, 0, s_4, s_9, s_{10}, s_9, s_{10}, s_{13}, s_{13}, s_{13}, s_{16}, s_{16}, s_{16}] \tag{2.14}$$

Une amélioration en terme de stabilité numérique a vu le jour ces dernières années et a permis aux modèles de Boltzmann sur réseau à temps de relaxation multiples de surpasser les modèles à un seul temps de relaxation [LL00]. Cependant, des débats sont toujours présents au sein de la communauté scientifique afin de savoir quel opérateur il est préférable d'utiliser.

2.3.5 Conditions limites

Trois catégories de cellules doivent être prises en compte afin de pouvoir réaliser des simulations sur de véritables géométries. En fonction de ces catégories, différentes règles

peuvent être appliquées à la place de l'équation de Boltzmann discrète. La première catégorie est la plus triviale : elle regroupe les « cellules fluides ». Les étapes de collision et de propagation se font normalement sur ces cellules à l'aide de l'équation de Boltzmann discrète. Les deux autres catégories sont plus problématiques. Elles comprennent en réalité les « cellules solides » ou obstacles ainsi que les « cellules fictives », c'est-à-dire les cellules normalement situées en dehors du maillage et n'intervenant pas directement dans la simulation. Le but de ces dernières est de définir les conditions aux bords du domaine de simulation.

L'opérateur de collision ne peut pas s'appliquer pour ces deux catégories et des règles doivent être définies pour ce type de cellules. La figure 2.7 représente les trois catégories pour un réseau bidimensionnel.

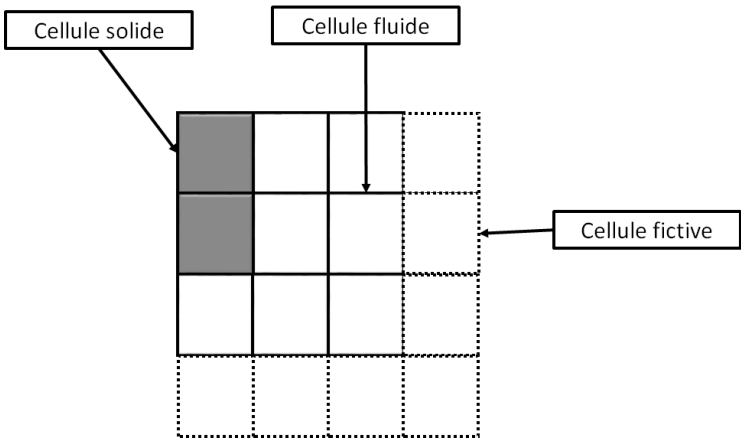


FIGURE 2.7 – Représentation des trois catégories de cellules

Il existe en littérature des solutions permettant de définir les règles pour les conditions aux obstacles. En général, la règle du rebond ou « Bounce-back » (No-Slip boundary condition) s'applique pour les obstacles du domaine de simulation, car elle impose une vitesse nulle sur ces obstacles [Zie93]. Cela consiste simplement à réaliser une inversion des fonctions de distribution proches des obstacles du domaine comme le montre l'équation suivante et la figure 2.8 :

$$f_i(x,t+\Delta t) = f_{\hat{i}}^{coll}(x,t) \tag{2.15}$$

avec $f_{\hat{i}}^{coll}$ étant la fonction de distribution de direction opposée à f_i à l'issue de l'étape

de collision.

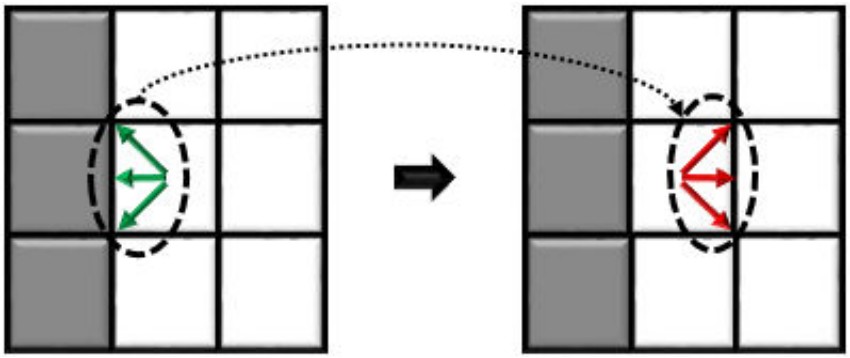


FIGURE 2.8 – Représentation du fonctionnement du Bounce-Back : les fonctions de distribution devant se propager dans un obstacle sont renvoyées dans leurs directions opposées.

Il existe toutefois d'autres règles en littérature permettant de traiter des conditions au niveau des obstacles comme par exemple les « Free-Slip boundary condition » [BFL01] ou encore la gestion d'obstacles mobiles [LL03]. Dans le cas où un obstacle est animé d'une certaine vitesse, la quantité de mouvement issue de son déplacement doit être transférée au fluide [Thü07]. Ainsi, le fluide peut être accéléré et repoussé au bord de l'obstacle. Concrètement, il suffit d'ajouter un terme de forçage pendant l'étape de propagation.

Aux bords du domaine, les fonctions de distribution issues des nœuds fictifs extérieurs sont inconnues. Il faut alors mettre en place des conditions de type entrée et sortie de fluide. Encore une fois, diverses méthodes sont à notre disposition [CMM96], [ZH97]. La solution la plus simple est de considérer que chacune des cellules extérieures au domaine fait office d'obstacle. Dans ce cas, toutes les conditions énoncées précédemment sont directement applicables. Cette situation particulière correspond au cas d'un domaine totalement fermé. Il est néanmoins possible de traiter le cas d'un domaine ouvert en imposant par exemple une vitesse et une densité d'entrée de fluide au niveau du bord. Ce profil de vitesse est alors facilement implémenté en mettant constamment à jour les cellules extérieures à partir des fonctions de distribution à l'équilibre (calculées en fonction de la vitesse et de la densité qui sont fixées). Une autre solution consiste à imposer sur le bord du domaine un gradient de vitesse nul. Il suffit pour cela de recopier les fonctions de distribution des cellules situées sur le bord dans les cellules situées à l'extérieur du domaine suivant la direction normale au bord. Il s'agit de la condition de type Neumann homogène.

Cette méthode requiert toutefois que les bords du domaine soient suffisamment éloignés les uns des autres pour éviter l'apparition d'instabilités numériques. Parfois, il est utile d'appliquer des conditions périodiques, c'est-à-dire que les particules sortant par un côté de l'espace sont réinjectées sur le côté opposé. Enfin, il est envisageable de refléter une partie des particules qui pointent vers les nœuds extérieurs. La quantité de particules réintroduites dans le domaine est déterminée de manière à assurer la conservation de la matière [BYS08].

2.3.6 Paramétrisation de la méthode de Boltzmann sur réseau

Comme la plupart des méthodes de dynamique des fluides, la méthode de Boltzmann sur réseau repose sur l'utilisation de valeurs non dimensionnées. Cela implique qu'il est nécessaire d'adimensionner le modèle. Pour cela, il faut avoir connaissance de plusieurs informations concernant la simulation à réaliser. Considérons un fluide de viscosité ν_r ayant pour vitesse caractéristique u_r dans un domaine de simulation ayant pour longueur caractéristique L_r. À partir de ces valeurs, il est alors possible de définir le nombre de Reynolds de la simulation (Équation 2.16). Ce nombre généralement noté Re est un nombre sans dimension utilisé en mécanique des fluides. Il caractérise un écoulement, en particulier la nature de son régime (laminaire, transitoire, turbulent).

$$Re = \frac{u_r L_r}{\nu_r} \tag{2.16}$$

L'utilisation de ce nombre permet en réalité de faire la jonction entre les valeurs réelles et les valeurs de simulation. Considérons maintenant une définition de la longueur caractéristique L_s et de la vitesse caractéristique u_s de la simulation exprimée en unité du réseau de Boltzmann. On peut ainsi en déduire le pas de discrétisation spatiale réel de la simulation :

$$\Delta^r x = \frac{L_r}{L_s} \tag{2.17}$$

Le pas de temps réel de la simulation peut être déduit de façon similaire :

$$\Delta^r t = \frac{u_s}{u_r}\Delta^r x \tag{2.18}$$

La connaissance de la viscosité du fluide, ainsi que des pas de discrétisations, nous permet alors de calculer la viscosité non dimensionnée du fluide :

$$\nu_s = \nu_r \frac{\Delta^r x}{\Delta^r t} \tag{2.19}$$

L'obtention de la viscosité de la simulation permet finalement d'obtenir le temps de relaxation τ de l'équation de Boltzmann discrète :

$$\tau = \frac{1}{2} + 3\nu_s \tag{2.20}$$

2.3.7 Stabilisation de la méthode de Boltzmann sur réseau

Comme la plupart des méthodes numériques, la méthode de Boltzmann sur réseau impose certaines conditions de stabilité. Le non-respect de ces conditions peut entraîner la divergence du modèle. Ces instabilités sont généralement liées à la valeur du temps de relaxation de la méthode de Boltzmann sur réseau (équation 2.6). Le calcul du temps de relaxation est cependant directement lié à la viscosité cinématique ν (équation 2.20). Une viscosité cinématique très petite peut donc facilement engendrer ces instabilités numériques.

Pour des nombres de Reynolds élevés (équation 2.16), la force d'inertie est beaucoup trop importante par rapport à la force de viscosité ce qui peut entraîner qu'un écoulement entre dans un régime turbulent. C'est ce phénomène qui se produit lorsque la viscosité cinématique est très faible. La méthode de Boltzmann est par conséquent mal adaptée pour gérer ces écoulements turbulents. Une solution envisageable consiste à réduire le pas de discrétisation spatiale afin de compenser la petite valeur de la viscosité cinématique. Cependant, agir de cette façon n'est pas sans conséquence sur le coût calculatoire de la simulation et s'avère finalement inexploitable. Une autre solution consiste à utiliser le modèle de sous-maille défini par Smagorinsky [Sma63]. Ce modèle permet de calculer le comportement des grandes échelles en modélisant l'action à petite échelle. L'idée de Smagorinsky était de modifier localement la valeur de la viscosité cinématique. Cette

modification consiste à ajouter une viscosité turbulente positive dont le calcul est effectué de manière spécifique [Egg96]. L'ajout de cette viscosité turbulente provoque une modification du temps de relaxation utilisé et permet ainsi de stabiliser certains phénomènes. En réalité, un temps de relaxation est calculé pour chaque cellule de la grille dans ces conditions. Cela permet ainsi de réduire les instabilités de manière locale.

2.4 Modélisation multi-phases et multi-composants

La méthode de Boltzmann classique est capable de réaliser des simulations physiques, mais elle reste toutefois limitée pour modéliser des phénomènes complexes faisant interagir plusieurs composants physiques ainsi que plusieurs états physiques d'un fluide. Toutefois, de nombreux travaux ont été développés au sein de la communauté scientifique de manière à pouvoir mettre en place des modèles physiques complexes basés sur l'utilisation d'une méthode de Boltzmann. C'est dans ce cadre que s'intègre le travail de modélisation physique du noyau de simulation effectué par Nicolas Maquignon dans le cadre de sa thèse [N.15]. Le but de cette section n'est pas de faire un inventaire des travaux existants, mais de définir la base de modélisation qui a été faite en vue du travail d'optimisation et de parallélisation. Ce manuscrit de thèse ne tient compte uniquement que des éléments de modèle qui ont été validés à l'issue du travail de modélisation du premier sujet du projet. Le noyau de simulation est ainsi capable de gérer plusieurs composants physiques pouvant être présents sous la forme de plusieurs états physiques. Il est de plus capable de gérer de gros ratios de densité entre deux phases d'un fluide, ce qui est extrêmement important pour des fluides tels que le GNL.

Néanmoins, les aspects thermiques du noyau de simulation permettant de mettre en place une dynamique de changement de phase ne sont pas encore assez aboutis et demandent un travail théorique supplémentaire très important. Leur incorporation a en effet tendance à rendre les simulations fortement instables, nécessitant une étude de méthodes théoriques et pratiques permettant leur stabilisation. C'est pourquoi nous ne tiendrons pas compte des effets thermiques dans ce manuscrit. Nous faisons donc l'hypothèse de nous placer dans un milieu totalement isotherme pour nos simulations.

2.4.1 Forces appliquées à un fluide

Les modèles de Boltzmann usuels ne tiennent généralement pas compte de forces agissant sur le fluide à simuler. Dans le cadre de la modélisation mise en place, l'interaction entre les particules composant un fluide doit être prise en compte. Un modèle couramment utilisé est un modèle développé par Shan & Chen [SC93]. Leur travail consistait dans un premier temps à calculer un pseudo-potentiel à partir de l'équation suivante :

$$\psi = \rho_0 \left(1 - \exp\left(-\frac{\rho}{\rho_0}\right)\right) \tag{2.21}$$

où ρ_0 est une constante de normalisation qui est généralement fixée à 1. La force d'interaction entre les particules d'un fluide peut alors être déduite de ce pseudo-potentiel à l'aide de l'équation suivante :

$$F(x) = -g\psi(x)c_s^2 \sum_{i=1}^{N} w(c_i)\psi(x+c_i)c_i \tag{2.22}$$

où N est le nombre de voisins de la cellule x. Les valeurs $w(c_i)$ correspondent aux facteurs de pondération définis dans les équations 2.8 et 2.9. Le paramètre g correspond à un facteur permettant de moduler la puissance de l'interaction entre les particules.

La force F définie dans l'équation 2.22 permet de modéliser les interactions entre les particules de fluides présentes sous différents états physiques au sein du domaine de simulation. Elle est responsable de la séparation des différentes phases présentes au sein du domaine. Toutefois, ce modèle dispose de plusieurs inconvénients comme le ratio de densité atteignable. En effet, le modèle dont nous avons besoin doit pouvoir atteindre un ratio de densité de 200 entre la phase liquide et la phase gazeuse et cette modélisation n'atteint pas ces ratios.

D'autres approches ont été étudiées afin de pouvoir atteindre d'importants ratios de densité (200 voire plus selon l'application). Le travail de modélisation de Nicolas Maquignon [N.15] propose une variante plus complexe du modèle de Shan & Chen. Il inclut une force d'interaction entre les particules d'un fluide et une force d'interaction entre les différents composants [BS13]. La première étape consiste à incorporer une équation d'état

sur l'ensemble du domaine de simulation. Dans le cadre de ce travail, c'est l'équation de Peng-Robinson qui a été retenue, mais il existe d'autres alternatives possibles. Elle est définie pour un composant physique σ à l'aide de l'équation suivante :

$$p_\sigma = \frac{\rho_\sigma R_\sigma T_\sigma}{1 - b_\sigma \rho_\sigma} - \frac{a_\sigma \alpha(T_\sigma) \rho_\sigma^2}{1 + 2 b_\sigma \rho_\sigma - b_\sigma^2 \rho_\sigma^2} \tag{2.23}$$

Les valeurs R_σ, b_σ et a_σ sont des valeurs constantes de l'équation de Peng-Robinson et peuvent se calculer à l'aide de la masse volumique critique du fluide et de sa température critique. T_σ représente la température du fluide. En attendant l'incorporation de la thermique dans le modèle, la température T_σ est considérée comme constante. De la même façon que le modèle de Shan & Chen, on calcule un pseudo-potentiel défini cette fois à partir de la pression précédemment calculée :

$$\psi_\sigma = \sqrt{\frac{2(p_\sigma - c_s^2 \rho_\sigma)}{c_s^2 g_{\sigma\sigma}}} \tag{2.24}$$

Le terme $g_{\sigma\sigma}$ est une constante connue du modèle. La force d'interaction peut être ensuite déduite de ce pseudo-potentiel. Une forme différente de la formulation de Shan & Chen est utilisée, basée sur les travaux de [GC12a], car elle permet d'obtenir une meilleure stabilité des simulations. Elle s'exprime sous la forme suivante :

$$F_{\sigma\sigma}(x) = -\beta \frac{g_\sigma}{2} c_s^2 \psi_\sigma(x) \sum_{x'} w(c_i) \psi_\sigma(x')(x' - x) - \frac{(1-\beta)}{2} \frac{g_\sigma}{2} c_s^2 \sum_{x'} w(c_i) \psi_\sigma^2(x')(x' - x) \tag{2.25}$$

Le terme β est une constante de pondération généralement fixée à 1.16 si l'on en croit [GC12a]. En addition à cette première force, une seconde force venant modéliser les interactions entre les différents composants physiques est définie. Elle est basée sur les travaux de [BS13] et se définit de la façon suivante :

$$F_{\sigma\sigma'}(x) = -\frac{g_{\sigma\sigma'}}{2} c_s^2 \psi_\sigma(x) \sum_{x'} w(c_i) \psi_{\sigma'}(x')(x' - x) \tag{2.26}$$

Une forme pondérée à la façon de l'équation 2.25 a également été étudiée mais la définition du facteur de pondération β est problématique pour l'appliquer aux interactions

entre composants.

La force de gravité est également incluse dans le modèle de simulation. Elle est définie de la manière suivante [GC12b] :

$$F_g = \rho g \left(1 - \frac{\hat{\rho}}{\rho}\right) \tag{2.27}$$

où g est la constante de gravitation et $\hat{\rho}$ la moyenne de la densité sur l'ensemble du domaine de simulation.

Une dernière force dont il faut tenir compte est la force d'adhésion entre le fluide et les obstacles. Elle peut être implémentée de la façon suivante dans le modèle :

$$F_{ads,\sigma}(x) = \frac{g_w}{2} \psi_\sigma(x) \sum_{x'} w(c_i) \psi_\sigma(\rho_w) s(x + c_i) c_i \tag{2.28}$$

où s est une fonction indicatrice pour définir si la cellule est un obstacle ou non, c'est-à-dire qu'elle vaut 1 pour les cellules solides et 0 pour les cellules fluides.

La somme des forces est ensuite calculée pour chaque composant. Ces forces doivent finalement être incorporées au sein du modèle de Boltzmann sur réseau.

2.4.2 Incorporation du terme force

L'incorporation du terme force est un facteur clé dans la modélisation de Boltzmann si l'on veut simuler des écoulements complexes incluant plusieurs composants et plusieurs phases. Il existe plusieurs méthodes en littérature permettant d'intégrer ces forces au sein du modèle. La première est une méthode nommée « Velocity shift » [SC93]. C'est cette méthode qui est utilisée pour incorporer les forces dans le modèle de Shan & Chen. Guo a mis également en place une méthode d'incorporation du terme de force dans [GZS02]. Le modèle développé repose sur une dernière méthode : la méthode aux différences exactes. Le terme de force est intégré dans l'équation de Boltzmann discrète à l'aide des équations suivantes :

$$f_{\sigma,i}(x + c_i, t + \Delta t) - f_{\sigma,i}(x,t) = -\frac{f_{\sigma,i}(x,t) - f^{eq}_{\sigma,i}(x,t)}{\tau_\sigma} + \Delta f_{\sigma,i}(x,t) \tag{2.29}$$

$$\Delta f_{\sigma,i}(x,t) = f^{eq}_{\sigma,i}(\rho_\sigma(x,t), u_\sigma(x,t) + \Delta u_\sigma(x,t)) - f^{eq}_{\sigma,i}(\rho_\sigma(x,t), u_\sigma(x,t)) \tag{2.30}$$

$$\Delta u_\sigma(x,t) = \frac{F_\sigma(x,t)\Delta t}{\rho_\sigma(x,t)} \tag{2.31}$$

L'inclusion est réalisée en appliquant une correction de la vitesse $u_\sigma + \Delta u_\sigma$. C'est une méthode coûteuse, car elle nécessite le calcul de deux fonctions d'équilibre pour obtenir le terme de collision. En effet, un calcul d'équilibre est nécessaire avant le calcul de correction de la vitesse et un second équilibre est calculé à l'issue de cette correction.

2.5 Conclusion

Cette section a pour but de résumer le travail de modélisation de Nicolas Maquignon sous la forme d'un premier algorithme. Dans le cadre du projet Innocold-Simulation, la mise en place d'un noyau de simulation permettant de simuler le comportement du GNL avec l'air ou l'eau a été étudiée. Ce noyau de simulation permet de modéliser les interactions entre plusieurs fluides mais également les interactions entre les différents états d'un fluide. Le but de ce travail était de fournir un modèle stable, capable de gérer les importants ratios de densité qu'il peut y avoir entre la phase liquide du GNL et sa phase gazeuse. La mise en place d'un tel modèle demande un travail théorique très important et au stade des avancées actuelles, des problèmes subsistent sur le plan de la thermique. La modélisation actuelle des effets thermiques entre les différents composants a pour effet de rendre les simulations fortement instables. De plus, les écarts de température entre le GNL liquide (-163°C) et l'air ambiant ($\approx$ 15°C) sont importants et de tels écarts sont extrêmement difficiles à gérer numériquement. C'est pourquoi le noyau de simulation n'inclut à l'heure actuelle aucun élément lié à la thermique. Il est alors défini à l'aide de l'algorithme 1.

De nombreuses pistes sont évoquées pour traiter les aspects thermiques comme l'utilisation d'un modèle de Boltzmann à double population ou encore un schéma aux différences finies. La difficulté réside également dans le choix d'un opérateur permettant le

Algorithme 1 : Algorithme décrivant les étapes du noyau de simulation.

↪ Initialisation de la simulation ;
pour *chaque itération* **faire**
  pour *chaque composant* σ **faire**
    ↪ Calcul de la pression p_σ à l'aide de l'équation (2.23) ;
    ↪ Calcul du pseudo-potentiel ψ_σ à l'aide de l'équation (2.24) ;
  fin
  pour *chaque composant* σ **faire**
    ↪ Calcul des forces à l'aide des équations (2.25)-(2.28) ;
    ↪ Calcul de la première fonction d'équilibre $f^{eq}_{\sigma,i}$ à l'aide de l'équation (2.7));
    ↪ Somme des forces et correction du terme de vitesse Δu_σ à l'aide de l'équation (2.31) ;
    ↪ Calcul de la seconde fonction d'équilibre $f^{eq}_{\sigma,i}$ à l'aide de l'équation (2.7);
    ↪ Calcul des temps de relaxations à l'aide d'un modèle de sous-maille de Smagorinsky ;
    ↪ Étape de collision-propagation des fonctions de distribution $f_{\sigma,i}$: application de règles différentes selon le type de la cellule ;
    ↪ Calcul des quantités macroscopiques ρ_σ et u_σ à l'aide des équations (2.10) et (2.11) ;
  fin
fin

changement de phase. Ces éléments seront considérés pour les travaux futurs à l'issue de ce travail de thèse.

Le noyau de simulation actuel a pour vocation d'être assez générique et est capable de simuler un grand nombre de phénomènes 2D et 3D. Il englobe l'ensemble des simulations réalisables à l'aide d'une modélisation de Boltzmann classique tels que le vortex de Karman (Figure 2.9), des écoulements de Poiseuille et bien d'autres « benchmarks » que l'on peut retrouver en littérature.

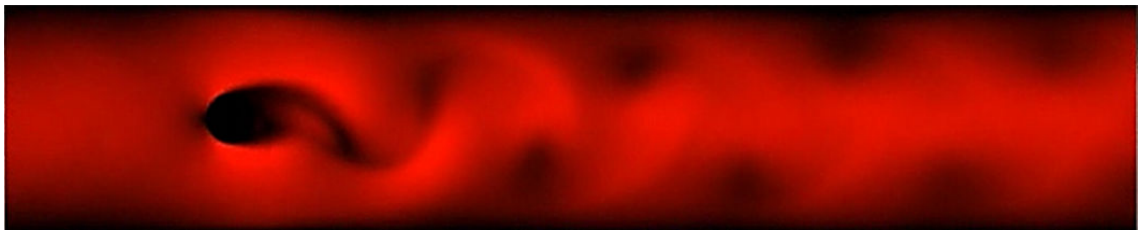


FIGURE 2.9 – Simulation des allées de Karman sur un domaine bidimensionnel à l'aide d'un schéma D2Q9 : un fluide entre au sein du domaine de simulation et vient heurter un obstacle circulaire. La présence de l'obstacle provoque l'apparition de turbulences et de vortex à l'intérieur du domaine.

Le noyau englobe également la possibilité de réaliser des simulations contenant un fluide présent sous la forme de plusieurs états physiques. De nombreuses simulations

sont également possibles telles que des simulations de condensation à une température constante (Figure 2.10), des simulations de remontée d'une bulle, etc.

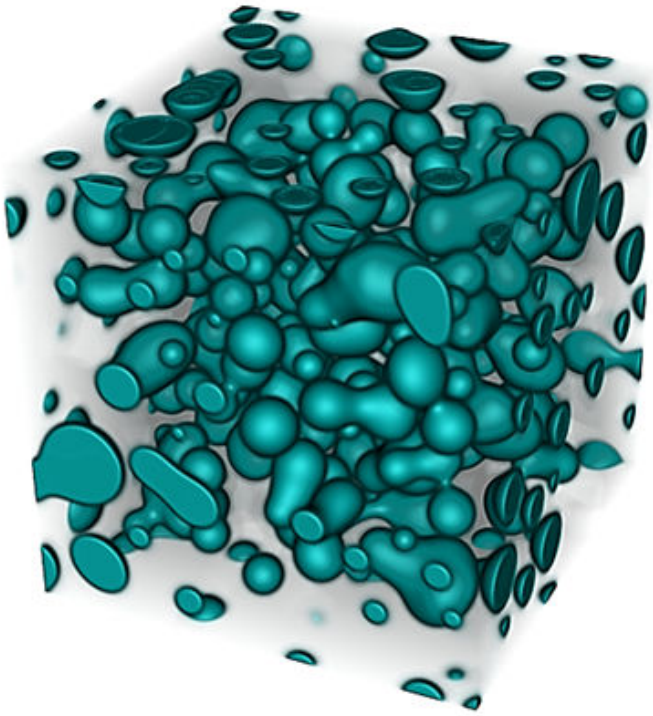


FIGURE 2.10 – Réalisation d'une simulation de condensation sur un domaine 3D à l'aide d'un schéma D3Q19 : la phase liquide et la phase gazeuse du fluide se séparent jusqu'à la formation de gouttes.

Finalement, ce modèle est capable d'intégrer plusieurs composants physiques au sein de la même simulation. L'interaction entre les particules des différents fluides est prise en compte. Cela permet de réaliser des simulations faisant interagir plusieurs fluides (Figure 2.11) ou encore des simulations d'écoulements sur des géométries complexes. En résumé, le noyau de simulation englobe de nombreuses possibilités de simulations incluant un ou plusieurs composants physiques en interaction, pouvant avoir eux-mêmes plusieurs états physiques également en interaction. Certaines vidéos représentant quelques simulations effectuées lors de ce travail de thèse sont disponibles à l'adresse suivante : http://www-lisic.univ-littoral.fr/ duchateau/.

La mise en place de ce noyau de simulation demande des connaissances importantes sur le plan de la physique théorique et numérique. Il a été construit de manière progressive

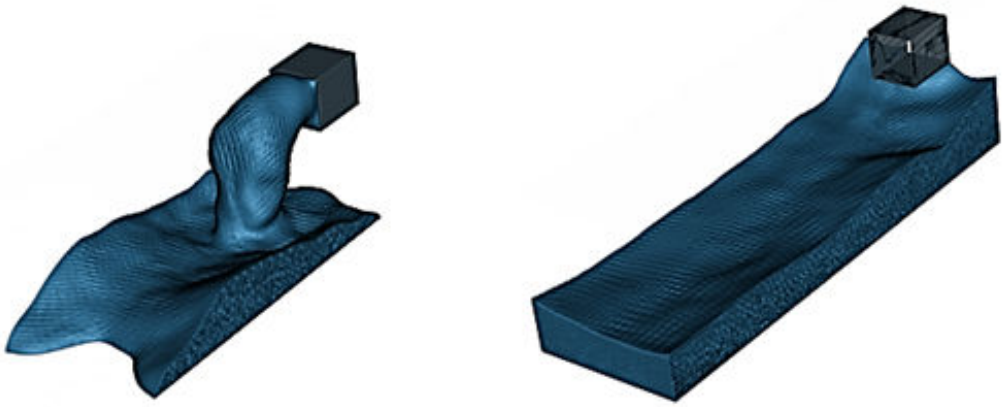


FIGURE 2.11 – Simulation d'épandage à deux composants pour un modèle D3Q19 : le premier composant est présent sous sa forme liquide et gazeuse alors que le second est présent uniquement sous forme gazeuse. Ce cas de figure peut représenter une simulation contenant du GNL sous sa forme liquide et gazeuse en interaction avec l'air présent uniquement sous forme gazeuse.

en parallèle aux autres travaux qui vont être présentés dans ce manuscrit. Il est toutefois important de constater que le noyau de simulation est extrêmement coûteux en ressources informatiques. En effet, la réalisation de simulations pouvant gérer plusieurs composants physiques sur de grandes installations soulève de nombreuses problématiques sur le plan informatique. La réduction de la mémoire et l'optimisation des calculs sont des problématiques importantes de ce travail de thèse. La parallélisation des calculs sur plusieurs architectures est également un facteur clé en vue d'obtenir des simulations réalisables en des temps acceptables. En effet, la méthode de Boltzmann sur réseau se prête extrêmement bien à la parallélisation. Favoriser et maximiser la parallélisation du modèle est par conséquent un élément essentiel de ce travail.

Deuxième partie

Réduction de la mémoire et parallélisation du noyau de simulation sur un processeur central multi-cœurs

Comme la plupart des méthodes numériques permettant de réaliser des simulations en dynamique des fluides, la méthode de Boltzmann sur réseau peut être extrêmement coûteuse en ressources informatiques. Cela inclut à la fois des problématiques liées à la rapidité des calculs, mais également à la gestion de la mémoire. De plus, l'utilisation d'un noyau de simulation comme celui présenté précédemment demande bien plus de ressources qu'une modélisation de Boltzmann classique. Toutefois, l'utilisation d'une méthode de Boltzmann dispose d'un avantage certain en comparaison d'autres méthodes numériques : son importante capacité de parallélisation.

Un premier objectif de cette partie est de mettre en lumière un certain nombre de techniques permettant de réduire de manière importante la consommation mémoire pour des simulations basées sur l'utilisation d'une méthode de Boltzmann sur réseau. Le choix de la méthode doit être fait de manière efficace, c'est-à-dire qu'elle doit être capable de réduire de manière importante la quantité de mémoire, mais également de ne pas perturber la capacité de parallélisation de la méthode.

La finalité de ce premier travail est de mettre en place une première version parallélisée du noyau de simulation et de réduire fortement sa consommation mémoire, afin de pouvoir réaliser nos premières simulations. La mise en place d'une parallélisation hybride du noyau de simulation par l'utilisation de la librairie OpenMP et d'instructions SSE (Streaming SIMD Extensions) disponible sur les microprocesseurs Intel y est étudiée. En effet, la méthode de Boltzmann est fortement adaptée au parallélisme SIMD. De ce fait, l'utilisation d'instructions SSE pour réaliser une parallélisation SIMD des calculs semble être une approche intéressante en vue d'exploiter l'ensemble des possibilités de calculs du processeur central. La mise en place d'un algorithme de calculs SIMD du noyau de simulation est finalement une première étape en vue d'un passage de ce noyau sur une architecture composée de processeurs graphiques.

Chapitre 3

Méthodes de réduction de la mémoire appliquées à la méthode de Boltzmann sur réseau

Sommaire

3.1 Introduction

La méthode de Boltzmann est une méthode hautement parallélisable intéressante pour simuler des phénomènes complexes. Cependant, pour d'importantes simulations, cela impose un coût en mémoire informatique qui peut s'avérer extrêmement important. En effet, l'utilisation d'un opérateur de collision comme celui présenté dans les sections (2.3.3) et (2.3.4) impose une dépendance spatiale et temporelle des données qui peut être problématique afin d'éviter toute perte d'informations.

L'approche standard utilisée généralement en littérature consiste en fait à considérer deux matrices de données pour éviter toute perte d'informations. Une grille **A** va se charger dans un premier temps du calcul de l'étape de collision et l'étape de propagation des fonctions de distribution se fait alors dans la grille **B**. À l'itération suivante, la situation est inversée. Le calcul de la collision se fait sur la grille **B** et l'étape de propagation se fait dans la grille **A** (Figure 3.1). C'est pourquoi cette méthode est communément appelée A-B pattern.

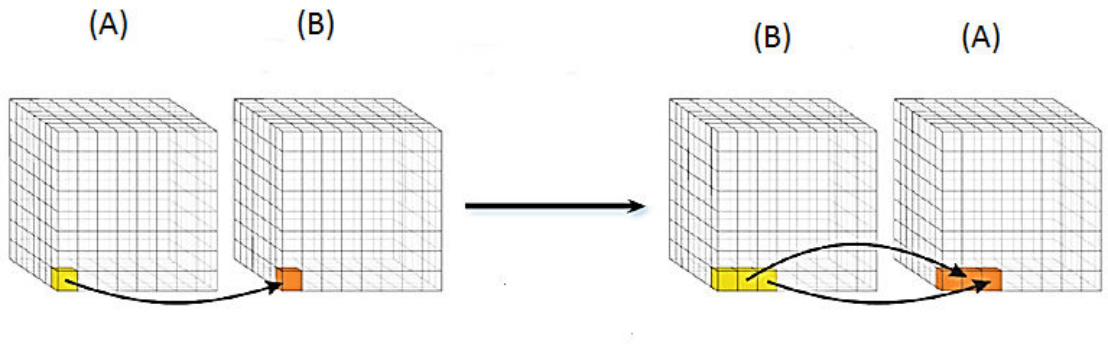


FIGURE 3.1 – Approche usuelle de stockage des données : deux grilles sont stockées en mémoire, la première gère la collision des fonctions de distribution et la deuxième grille gère la propagation de manière à ne perdre aucune information liée à la dépendance spatiale et temporelle.

Cette méthode peut être avantageuse, car elle est simple d'implémentation et elle offre une parallélisation optimale des calculs pour l'ensemble du domaine de simulation. En effet, l'indépendance des calculs pour chaque cellule offre une grande capacité de parallélisation. Cependant, elle souffre d'un gros désavantage qui est la consommation mémoire nécessaire pour une simulation. Considérons le cas d'une simulation utilisant le modèle présenté dans le chapitre précédent sur un domaine composé de 1024^3 cellules de calculs

utilisant deux composants physiques. Une estimation de la quantité de mémoire nécessaire pour traiter cette simulation est de **672 Gio**, soit une quantité très largement supérieure à celle présente sur l'architecture matérielle à notre disposition. Il est donc indispensable d'apporter des solutions permettant de réduire au mieux la quantité de données nécessaire, afin de pouvoir simuler la plus grande taille de domaine possible. La réduction du coût calculatoire est également une étape importante pour traiter des contextes de simulation à grandes échelles comme peut l'être un terminal méthanier. En effet, un terminal méthanier peut s'étendre sur des kilomètres et le pas de discrétisation spatiale de la méthode de Boltzmann sur réseau est limité et relativement faible pour obtenir des simulations stables.

La littérature propose toutefois un certain nombre de méthodes permettant de réduire la quantité de mémoire nécessaire pour les simulations. L'ensemble des méthodes connues a été étudié dans ce manuscrit, afin de choisir une méthode qui réduise au mieux la quantité de mémoire, tout en offrant une parallélisation efficace des calculs.

3.2 Algorithme de compression de grilles

La technique de compression de grilles de Pohl [PKW+03], qui peut être aussi connue sous le nom de *shift algorithm*, est une première méthode permettant de compresser les données. L'idée est principalement de réduire la consommation mémoire et d'améliorer la localité spatiale des données en mémoire. Contrairement à la méthode standard A-B pattern, il faut considérer uniquement une grille de calculs en mémoire contenant une légère extension, comme le montre la figure 3.2.

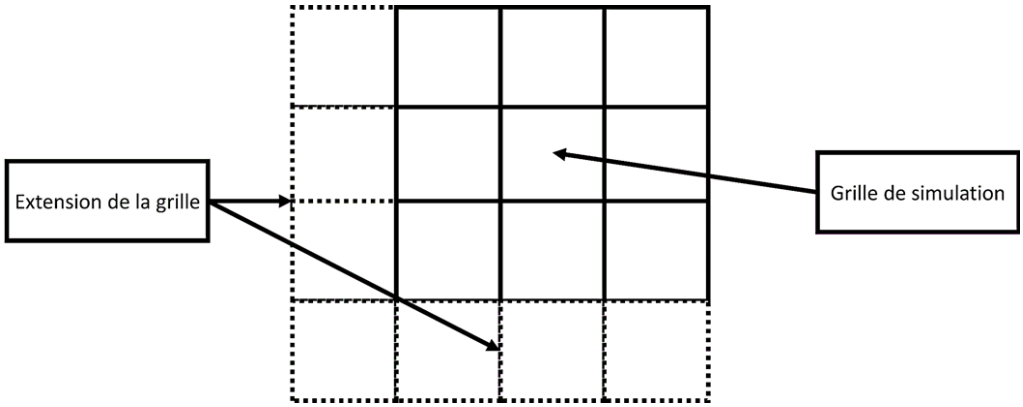


FIGURE 3.2 – Grille de simulation utilisant la méthode de compression de grilles : une légère extension de la grille est utilisée pour accueillir les informations dues à la propagation des données.

L'idée est ensuite d'initialiser les données dans la partie supérieure (ou inférieure respectivement) de la grille. Les calculs de collision se font ensuite sur la partie de la grille qui a été initialisée. La propagation des données se fait finalement par une translation en diagonale des données vers la zone mémoire non utilisée, comme peut le montrer la figure 3.3.

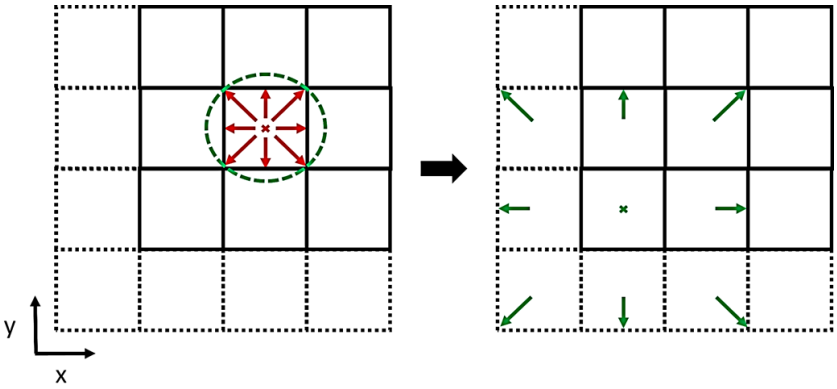


FIGURE 3.3 – Méthode de compression de grilles : l'étape de collision se fait au sein de la grille de simulation et la propagation se fait dans la diagonale montante ou descendante selon l'itération.

Après une première itération, les données de simulation se retrouvent alors dans la zone inférieure (ou supérieure respectivement) de la matrice de données globale. À la seconde itération, les calculs se font dans le sens inverse. L'étape de collision se fait dans la zone supérieure (ou inférieure respectivement) pour être ensuite propagée vers la partie inférieure (ou supérieure respectivement). Cela implique l'utilisation de deux pas de temps successifs distincts par un balayage du domaine de simulation qui diffère selon les itérations. La figure 3.4 résume les deux étapes de calculs de la méthode.

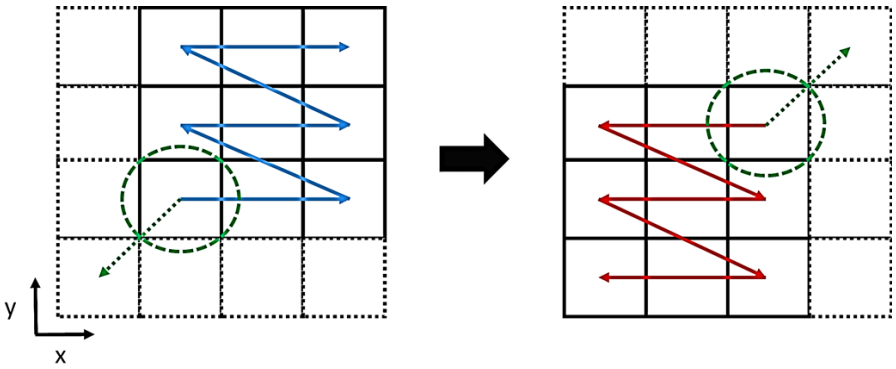


FIGURE 3.4 – Méthode de compression de grilles : le domaine de simulation est balayé de manière différente pour deux itérations successives de manière à ne perdre aucune donnée.

La méthode de compression de grilles est une méthode intéressante, car elle permet de réduire la consommation en mémoire liée aux fonctions de distribution d'environ 50%. Elle dispose toutefois d'un gros désavantage qui est sa faible capacité de parallélisation. En effet, l'utilisation d'un balayage particulier, ainsi qu'un stockage de l'étape de propagation en diagonal, a un impact plutôt néfaste sur la possibilité de parallélisation de la méthode. Celle-ci est en effet limitée à une seule tranche du domaine de simulation afin d'éviter une perte d'informations due au stockage des données de propagation en diagonal.

3.3 Méthode de « Swap algorithm »

La méthode de « Swap algorithm » de Matilla [MHR+07] et Latt [Lat07] appartient également aux méthodes de réduction de mémoire décrites en littérature. Cette méthode, comme la méthode de compression de grilles, ne requiert l'utilisation que d'une seule grille de simulation pour gérer la dépendance spatiale et temporelle des données. Toutefois, contrairement à la méthode précédente, celle-ci ne requiert aucune extension de la grille pour accueillir les données de propagation. La méthode diffère toutefois selon l'ordre de réalisation des étapes de collision et de propagation :

— L'étape de propagation suit l'étape de collision (Figure 3.5), communément appelé « push scheme ».

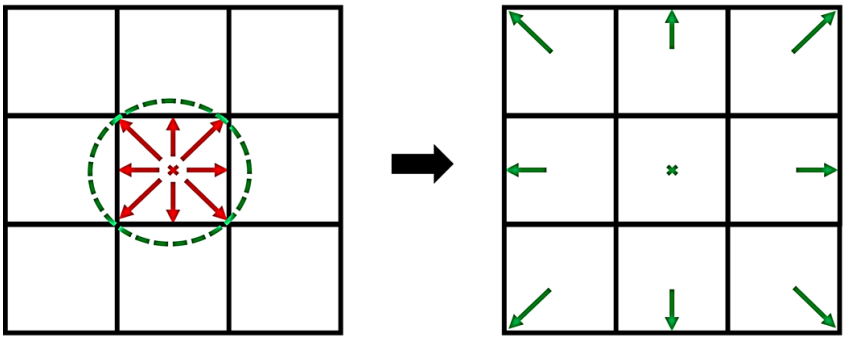


FIGURE 3.5 – Le « push scheme » : les données sont lues localement, la collision est appliquée et les données sont propagées vers les plus proches voisins.

— L'étape de collision suit l'étape de propagation (Figure 3.6), communément appelé « pull scheme ».

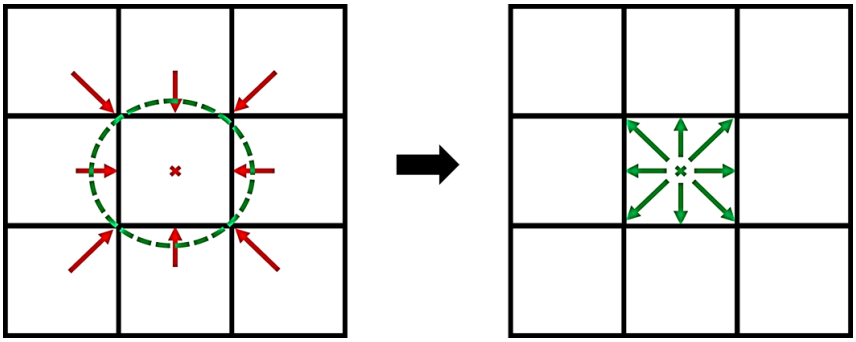


FIGURE 3.6 – Le « pull scheme » : les données sont lues depuis les plus proches voisins, propagées vers la cellule cible et finalement l'étape de collision est réalisée.

La méthode de « Swap algorithm » se base sur un jeu d'inversion organisé au niveau des fonctions de distribution de manière à conserver l'ensemble des informations en mémoire. L'idée est en réalité d'effectuer un échange pour la moitié de la cellule où le calcul est effectué, avec son voisinage. Dans le cas du « push scheme », la collision est effectuée dans un premier temps et une inversion au sein des fonctions de distribution en local est réalisée, c'est-à-dire que les fonctions de distribution de signes opposés sont échangées à l'issue de l'étape de collision. Un échange avec les cellules voisines est finalement réalisé pour la moitié de la cellule. Dans ce cas, les échanges se font avec les voisins d'indice inférieur comme l'indique la figure 3.7.

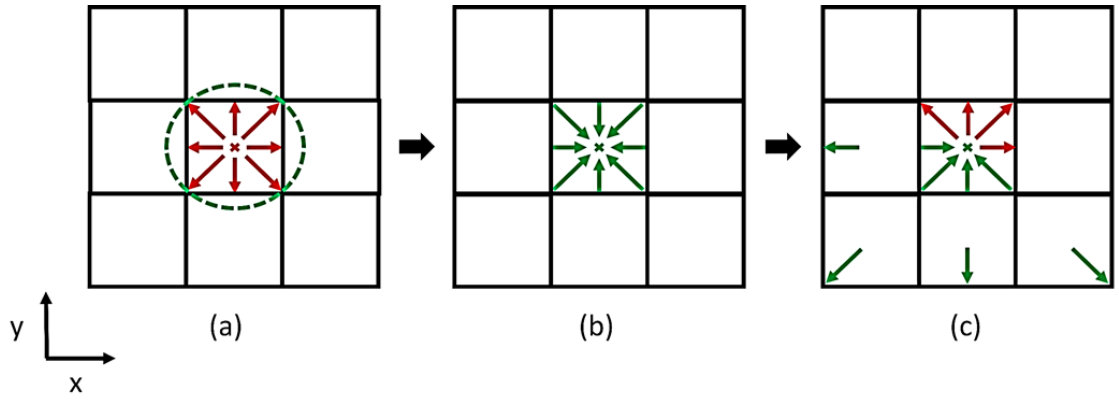


FIGURE 3.7 – Méthode de « Swap algorithm » appliquée au « push scheme » : (a) Définition de la cellule ciblée ; (b) la collision est appliquée localement sur la cellule et les fonctions de distribution opposées sont échangées ; (c) Une inversion avec les plus proches voisins d'indice inférieur est finalement réalisée.

Concernant le « pull scheme », l'inversion est dans un premier temps réalisée avec cette fois les voisins d'indice supérieur, comme le montre la figure 3.8. Ensuite, l'étape de collision est réalisée et les fonctions de distribution opposées sont inversées localement.

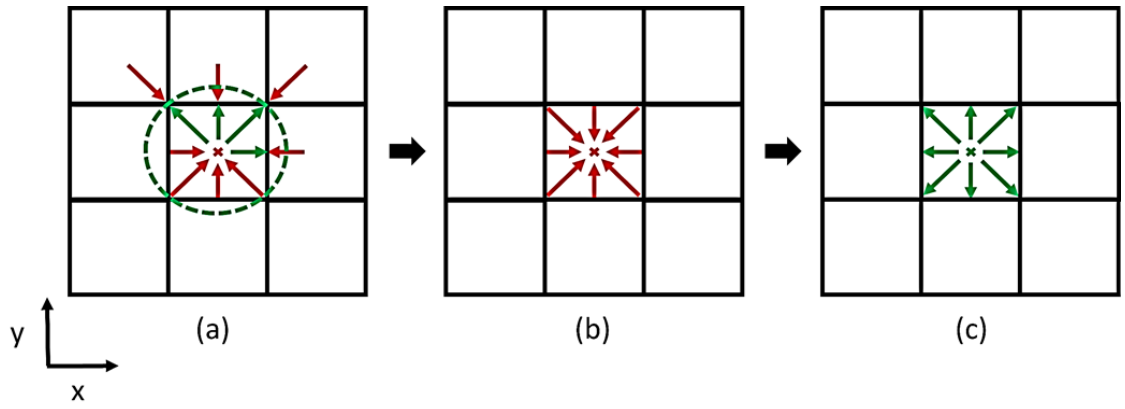


FIGURE 3.8 – Méthode de « Swap algorithm » appliquée au « pull scheme » : (a) Définition de la cellule ciblée ; (b) une inversion avec les plus proches voisins d'indice supérieur est tout d'abord réalisée ; (c) La collision est finalement appliquée localement sur la cellule et les fonctions de distribution opposées sont échangées de manière à retrouver leur ordre naturel.

L'avantage principal de cette méthode est qu'elle ne nécessite qu'une seule grille de calculs. Ainsi, un gain de 50% en mémoire est obtenu pour un modèle de Boltzmann sur réseau standard. De plus, la parallélisation n'est pas affectée par la méthode contrairement à la compression de grilles. Toutefois, les échanges avec les plus proches voisins doivent être bien séparés de l'étape de collision de manière à ne perdre aucune information. Cela implique la séparation de la méthode en deux étapes de calculs : une étape responsable de l'échange avec les plus proches voisins et l'autre responsable de la collision et de l'inversion en local des fonctions de distribution.

3.4 Algorithme « Esoteric twist »

L'algorithme « Esoteric twist » de Schönherr [SGK11] partage certaines caractéristiques avec la méthode de « Swap algorithm ». Elle ne requiert qu'une seule grille de simulation et est essentiellement parallèle. La méthode fait appel aux plus proches voisins d'indice supérieur de la cellule. L'étape de collision est réalisée sur certaines fonctions de distribution de la cellule à traiter ainsi que sur les plus proches voisins d'indice supérieur. Une inversion des fonctions de distribution opposées est également réalisée à l'issue de la collision (Figure 3.9). Finalement, une seconde inversion est réalisée localement sur les cellules de manière à retrouver leur ordre naturel.

Comme la méthode « Swap algorithm », cette méthode est intrinsèquement parallèle

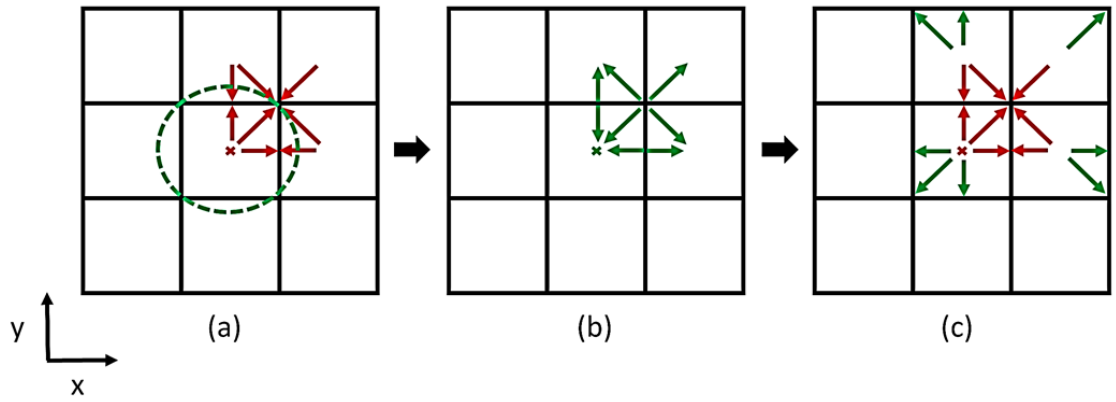


FIGURE 3.9 – Algorithme « Esoteric twist » : (a) Définition de la cellule ciblée ; (b) l'étape de collision est calculée sur une partie de la cellule à traiter ainsi que sur les plus proches voisins d'indice supérieur et les fonctions de distribution opposées sont inversées ; (c) Une seconde inversion est finalement réalisée localement sur les cellules de manière à retrouver leur ordre naturel.

à la condition de séparer une itération de l'algorithme en deux étapes : la collision et la première inversion peuvent être réalisées au sein de la même étape de calculs alors qu'une seconde étape est nécessaire pour la seconde inversion. En effet, cette séparation permet de conserver l'information sans fragiliser la parallélisation.

3.5 Méthode A-A pattern

La technique A-A pattern de Bailey [BMW+09] est également une méthode de réduction de mémoire présente en littérature. Son principe consiste en deux pas de temps successifs distincts qui permettent une meilleure parallélisation tout en réduisant le coût mémoire de manière importante. Le premier pas de temps, que l'on appellera « *A-step* » , est purement local sur l'ensemble du domaine. Il consiste à effectuer une étape de collision sur l'ensemble du domaine et à réaliser une inversion des valeurs au sein des fonctions de distribution opposées en local (Figure 3.10). En comparaison avec la méthode classique (Figure 3.5), le *A-step* effectue une advection camouflée par l'inversion locale des fonctions de distribution. Cependant, pour retrouver les quantités macroscopiques à l'issue de ce pas de temps, il faut tenir compte de l'état des fonctions de distribution comme si elles avaient été réellement propagées.

Le second pas de temps, que l'on appellera « *B-step* », consiste à réaliser les étapes de propagation, de collision puis à nouveau de propagation en tenant compte des valeurs

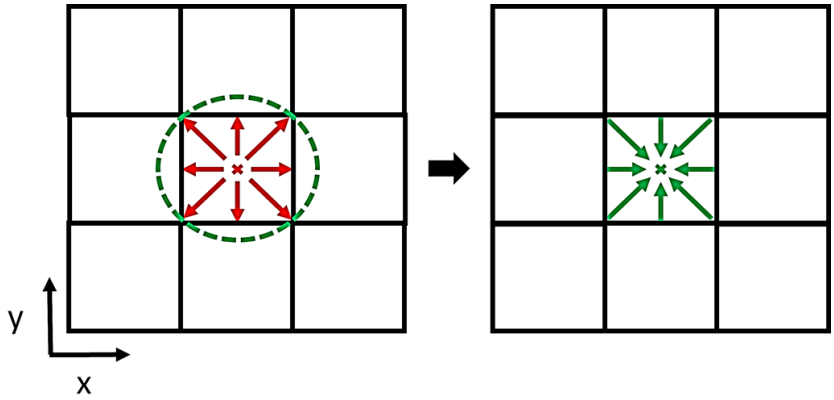


FIGURE 3.10 – Méthode A-A pattern : l'étape de collision est effectuée localement et les fonctions de distribution opposées sont inversées.

issues du *A-step*. L'idée consiste à prendre en compte les fonctions de distribution dans le voisinage de la cellule ciblée et à effectuer une collision sur ces valeurs. Une fois l'étape de collision réalisée, la double propagation se fait par échange des valeurs au niveau du voisinage de manière à retrouver leur ordre naturel.

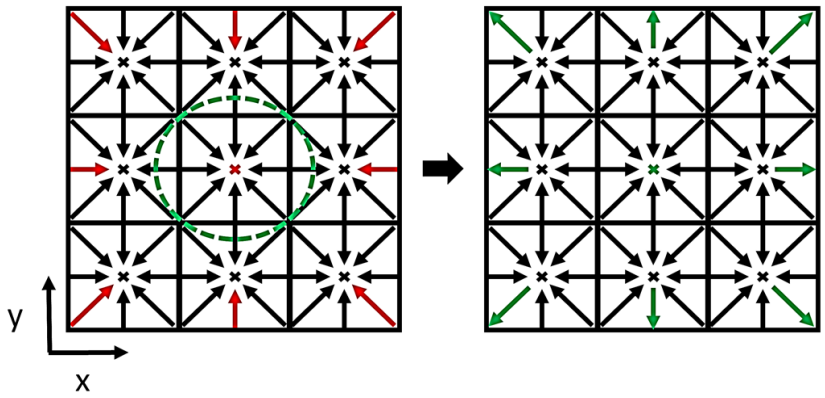


FIGURE 3.11 – Méthode A-A pattern : les données des plus proches voisins sont lues, on y applique l'étape de collision et les fonctions de distribution opposées sont inversées de manière à retrouver leur ordre naturel.

Une comparaison avec la méthode de collision-propagation classique nous montre que deux itérations successives de la méthode A-A pattern reviennent à réaliser deux itérations dans un cas standard. Cela implique donc qu'il n'y aucune perte d'information liée à l'utilisation de cette méthode. De manière similaire à la méthode « Swap algorithm », la mémoire est réduite de 50% pour un modèle de Boltzmann standard. De plus, aucune perturbation ne vient toucher la parallélisation. Les calculs peuvent se faire de manière

indépendante sur chaque cellule du domaine de simulation quelle que soit l'itération, ce qui est un réel avantage. Le seul réel inconvénient de cette méthode est que l'utilisation de deux pas de temps complètement distincts impose un développement du code de simulation plus laborieux. En effet, il faut bien tenir compte de l'ensemble des inversions qui sont faites sur les deux itérations, de manière à ne perdre aucune information de simulation, notamment pour les conditions limites, le traitement des obstacles et le calcul des quantités macroscopiques.

3.6 Choix d'une technique de réduction de mémoire adaptée pour le noyau de simulation

La littérature offre plusieurs possibilités en matière de gestion de la mémoire liée à la méthode de Boltzmann sur réseau. La première et la plus commune, consiste à utiliser deux matrices de données permettant de gérer la dépendance spatiale et temporelle des données. Elle offre un avantage incontestable sur le plan parallélisme, mais souffre d'un énorme défaut sur le plan de la consommation mémoire. L'utilisation d'une telle méthode n'est pas envisageable pour réaliser des simulations à plusieurs composants physiques sur de très grandes grilles de simulations.

Plusieurs solutions ont été décrites dans les sections précédentes et chacune d'entre elles dispose d'avantages et d'inconvénients. L'objectif est donc de choisir la méthode la plus adaptée à nos besoins, à savoir celle qui réduit au mieux la mémoire et qui ne perturbe pas la parallélisation des calculs. Pour cela, une comparaison de performances entre les différentes méthodes est réalisée à travers quelques simulations.

3.6.1 Simulations réalisées

Pour réaliser cette étude de performances, on définit deux simulations qui vont servir de base pour chaque technique de réduction de mémoire. Nous utilisons donc la simulation du vortex de Karman en deux et trois dimensions pour réaliser cette étude. Cette simulation fait entrer un fluide à l'intérieur du domaine de simulation ; ce dernier entre en collision avec un obstacle cylindrique situé au centre du domaine. La présence de cet obs-

tacle provoque l'apparition de turbulences et de vortex au sein du domaine de simulation (Figure 3.12).

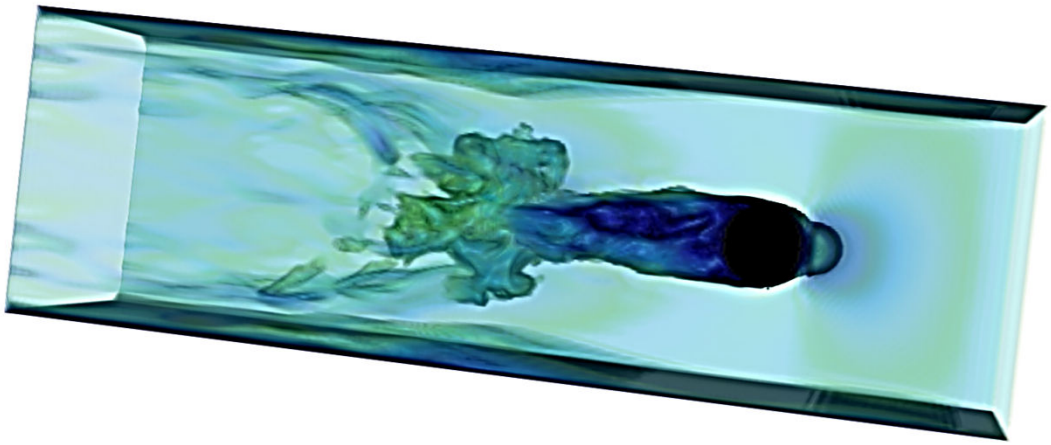


FIGURE 3.12 – Simulation du vortex de Karman en 3D à l'aide d'un schéma D3Q19.

Ces simulations ne font pas appel à l'intégralité du noyau de simulation, elles n'intègrent en effet pas de forces particulières et font simplement appel à un modèle de Boltzmann classique composé des étapes de collision et de propagation habituelles. Elles sont toutefois adaptées pour évaluer les performances des méthodes de réductions de mémoire car ces techniques ont essentiellement un impact sur ces deux étapes de calculs. Le domaine de simulation pour la simulation 2D est composée de 1000×200 cellules de calculs alors que la simulation 3D est composée de $500 \times 128 \times 128$ cellules de calculs. L'idée est donc d'incorporer chacune des méthodes dans le code de simulation et de réaliser une comparaison de performances.

3.6.2 Architecture de calculs utilisée

Les tests ont été réalisés à l'aide d'une machine composée de deux processeurs Intel Xeon X5660 dont les caractéristiques sont définies dans le tableau 3.1. L'objectif est dans un premier temps de paralléliser de manière simple les différents codes de simulations en utilisant l'ensemble des cœurs de calculs disponibles à l'aide de la librairie OpenMP.

Intel Xeon X5660	
Nombre de cœurs	6
Nombre de threads	12
Fréquence de base du processeur	2.8 GHz
Fréquence maximale	3.2 GHz
Instructions SIMD	SSE 4.2

TABLEAU 3.1 – Configuration technique du processeur Intel Xeon X5660

3.6.3 Comparaison des performances

La réduction de mémoire occasionnée par ces techniques est généralement la même, car elle permet de supprimer la seconde matrice de données nécessaire au stockage des données de propagation. La méthode de compression de grilles nécessite toutefois une légère extension de la grille pour accueillir les données de propagation à l'inverse de ses concurrentes. La mémoire réduite pour ces simulations est donc d'environ 49% pour la méthode de compression de grilles et d'exactement 50% pour les autres.

Concernant les performances, l'unité de mesure généralement utilisée pour la méthode de Boltzmann est le « Million Lattice nodes Update Per Second » (MLUPS). Cela représente le nombre de cellules mises à jour par seconde. Elle est calculée de la façon suivante :

$$P_{MLUPS} = \frac{X \times Y \times Z \times N_{iter}}{t_{sim}} \tag{3.1}$$

où X, Y, Z, définissent la taille du domaine, N_{iter} correspond au nombre d'itérations de la simulation et t_{sim} correspond au temps de simulation en secondes.

La figure 3.13 illustre la comparaison de performances pour les deux simulations du vortex de Karman. On constate dans un premier temps que la méthode de compression de grilles offre en moyenne des performances bien en dessous de ces concurrentes. En effet, elle offre une performance d'environ 17.8 millions de cellules mises à jour en 2D alors que les autres sont en moyenne à plus de 50 MLUPS ce qui représente une perte de performances d'environ un facteur 3. Le même constat est à noter pour la simulation 3D où les performances sont aussi bien inférieures pour cette méthode. L'obtention de performances si faibles provient du principal défaut de la compression de grilles. L'utilisation

d'une telle méthode bride en effet la capacité de parallélisation de la méthode de Boltzmann, car le stockage des données en diagonale limite la capacité de parallélisation à un seul niveau de la grille de simulation et non à l'intégralité de la grille. L'utilisation d'une telle méthode semble alors inadaptée pour son intégration au sein du noyau de simulation.

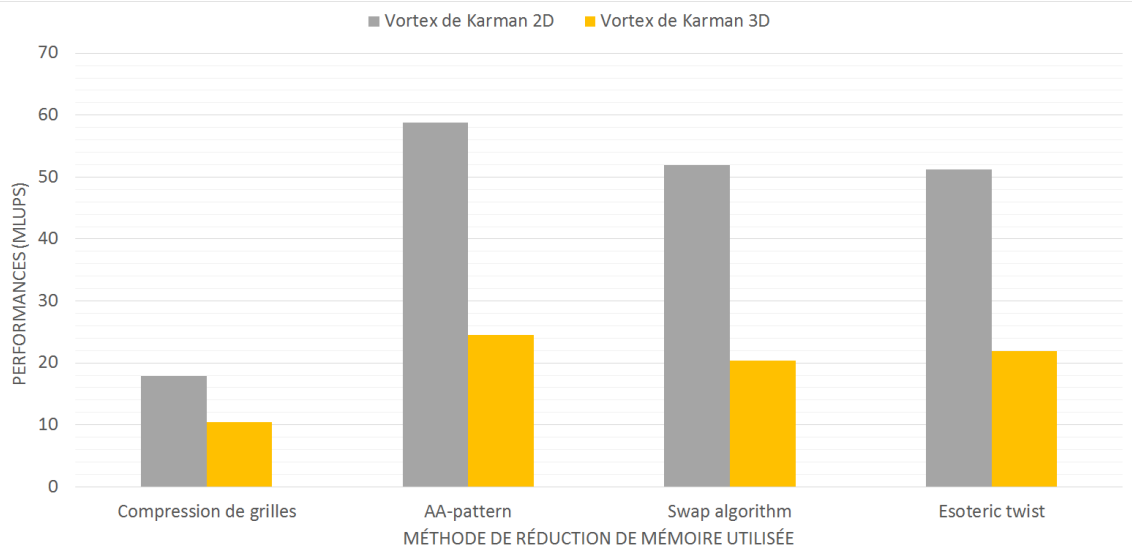


FIGURE 3.13 – Comparaison de performances pour des simulations du vortex de Karman en deux et trois dimensions associées à plusieurs méthodes de réduction de mémoire.

Les méthodes Swap algorithm et Esoteric twist offrent quant à elles des performances intéressantes et très similaires ($\approx$ 51 MLUPS en 2D et $\approx$ 20 MLUPS en 3D). Cela est dû au fonctionnement des deux méthodes qui est également fortement similaire. En revanche, la méthode A-A pattern offre des performances légèrement supérieures à ses deux principales concurrentes. En effet, la méthode arrive en moyenne à 59 MLUPS pour la simulation 2D et 24 pour la simulation 3D. Cela est dû au fait qu'une itération de la méthode A-A pattern peut se faire en une seule étape de calculs, alors que deux sont nécessaires pour les deux autres méthodes. En effet, la demande en calculs est la même pour l'ensemble de ces méthodes, mais les méthodes « Swap algorithm » et « Esoteric Twist » demandent un second balayage de la grille de simulation pour réaliser leurs échanges de données. Ce second balayage n'est pas nécessaire pour la méthode A-A pattern, ce qui lui permet d'obtenir les meilleures performances.

La méthode A-A pattern est par conséquent la technique qui semble offrir les meilleures performances sur le plan calculatoire et qui sera incluse dans le noyau de simulation. Elle est en effet intrinsèquement parallèle et réduit de manière considérable la quantité de mé-

moire nécessaire pour réaliser une simulation. Son seul inconvénient est qu'elle implique deux itérations successives distinctes, ce qui impose *a fortiori* un effort de développement supplémentaire.

Chapitre 4

Parallélisation hybride SIMD du noyau de simulation par l'utilisation combinée d'OpenMP et d'instructions SSE

Sommaire

4.1 Introduction

Le but de cette section est de mettre en place une première parallélisation de notre noyau de simulation sur une architecture du type processeur central multi-cœurs. L'objectif premier est de se familiariser avec la complexité du modèle, tout en maximisant les performances de calculs. Au meilleur de notre connaissance, la parallélisation SIMD du modèle de Boltzmann sur réseau n'a pas été exploitée sur un processeur central multi-cœurs. La parallélisation sur processeur graphique a très rapidement pris le dessus au sein de la communauté scientifique. Cependant, l'utilisation d'instructions SSE pour obtenir une parallélisation SIMD du moteur de calculs permet d'adapter l'algorithme de manière à obtenir de bonnes performances en vue d'un passage ultérieur sur une architecture massivement parallèle telle que celle d'un processeur graphique.

4.2 Parallélisation de méthode de Boltzmann sur CPUs : état de l'art

La méthode de Boltzmann sur réseau est une méthode de modélisation physique récente et qui a pris du temps à se faire accepter par la communauté scientifique. Son approche est complètement différente des méthodes de modélisation classiques du type éléments finis ou volumes finis. L'intérêt pour cette méthode s'est développé au fil des années, avec la validation de simulations dont le comportement est connu ou prévisible (appelés généralement « benchmarks ») et le développement de modèles plus complexes.

4.2.1 Parallélisme multi-cœurs sur architectures à mémoire partagée

Il faut attendre l'arrivée des premières architectures multi-cœurs pour voir arriver les premiers travaux concernant le parallélisme de la méthode de Boltzmann sur réseau. Les travaux de Bella [BFRU02] en 2002 ont permis de mettre en avant l'efficacité de la parallélisation de la méthode de Boltzmann en deux dimensions par l'utilisation de la librairie OpenMP [DE98]. L'idée est de découper le domaine de simulation selon le nombre de cœurs de la machine et de réaliser les calculs en parallèle sur les différents sous-domaines, comme peut le montrer la figure 4.1. Ces travaux mettent également en évidence le point

le plus critique de la parallélisation, qui concerne l'étape de propagation des données vers les plus proches voisins sur l'ensemble de la grille de simulation. La méthode A-B pattern, utilisant deux matrices de données, est alors choisie de manière à conserver une parallélisation optimale des calculs.

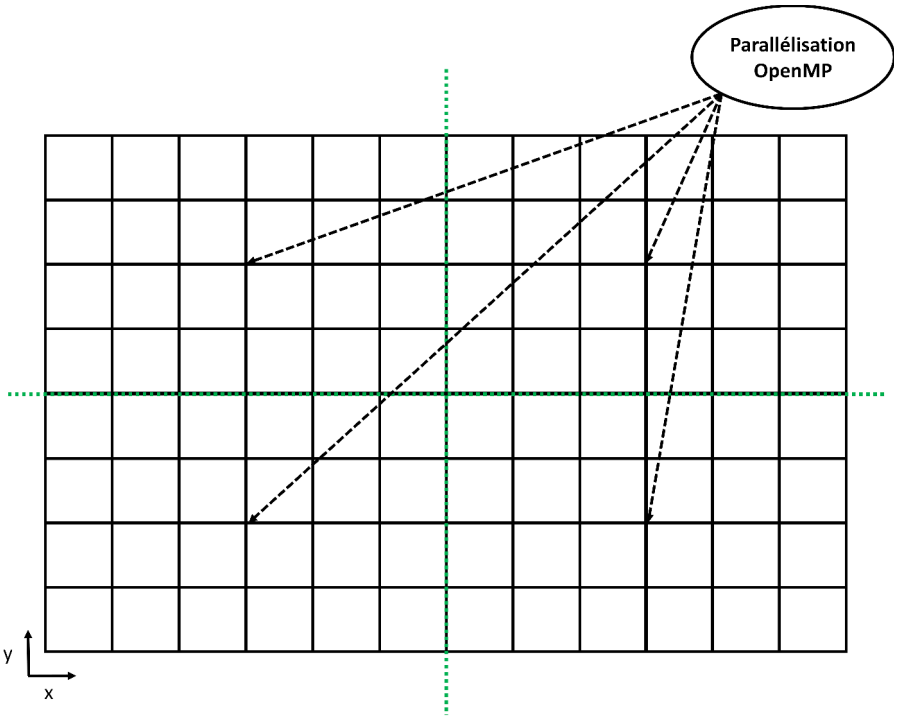


FIGURE 4.1 – Découpage d'un domaine de simulation 2D pour 4 cœurs.

Les travaux de Massaioli [MA02], également en 2002, se sont attardés sur la parallélisation de la méthode de Boltzmann dans un cas tridimensionnel. Le même constat est fait que pour les travaux en deux dimensions : la phase de propagation des données est la plus coûteuse. Les simulations en trois dimensions demandant plus de calculs, des solutions doivent être proposées de manière à améliorer les performances. Une première solution est apportée et consiste à fusionner les étapes de collision et de propagation de la méthode de Boltzmann. En effet, le caractère local à une cellule de l'étape de collision et l'utilisation d'une deuxième matrice de données pour accueillir les données de l'étape de propagation autorisent cette fusion sans perte d'informations. Cette solution permet d'économiser le nombre de balayages de la grille de simulation ce qui représente donc une amélioration de performances que ce soit de manière séquentielle ou parallèle. En effet, la parallélisation n'est pas affectée par cette fusion ce qui implique que les calculs peuvent toujours se faire de manière parallèle sur l'ensemble de la grille de simulation.

En 2003, les travaux de Pohl [PKW+03] s'attardent plus spécifiquement sur les aspects d'optimisations du cache du processeur central, en vue d'obtenir une simulation parallèle performante. La fusion des étapes de collision et de propagation est évoquée comme dans [MA02]. L'optimisation de la disposition des données en mémoire est également abordée. En effet, le rapprochement de données utilisées dans un intervalle de temps très court dans des zones mémoire très proches permet de réduire le trafic de données au sein du cache du processeur central et ainsi de gagner en performances. Deux idées sont évoquées dans ces travaux. La première consiste à fusionner les deux matrices de données d'une manière spécifique (Figure 4.2) de façon à rapprocher la lecture des données à un temps t et l'écriture à un temps $t+1$. L'autre idée consiste à utiliser la méthode de compression de grilles décrite dans la section 3.2. La fusion des deux matrices de données provoque un léger gain de performance en comparaison à l'utilisation de deux matrices de données. En revanche, les performances obtenues pour la méthode de compression de grilles seule sont légèrement inférieures à la fusion des deux matrices de données.

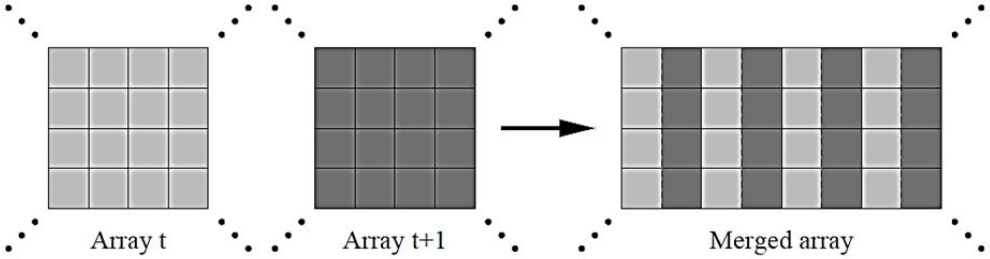


FIGURE 4.2 – Les deux tableaux de données 2D permettant de gérer la dépendance spatiale et temporelle des données sont regroupés au sein d'un unique tableau [PKW+03].

Un autre aspect que traite les travaux de Pohl concerne l'optimisation des calculs. L'idée est d'utiliser le fait que la méthode de Boltzmann soit une méthode de calculs interagissant uniquement avec les plus proches voisins. Ainsi, pour mettre à jour une cellule d'un instant t à un instant $t+1$, il suffit d'avoir les plus proches voisins à l'instant t. En exploitant cette idée, on peut considérer qu'il n'est pas indispensable d'attendre la fin d'une itération pour l'ensemble de la grille pour démarrer d'autres itérations à certains endroits de la grille. La méthode mise en place consiste alors à découper le traitement par blocs et d'effectuer plusieurs boucles de mises à jour à l'intérieur de ces blocs, comme le montre la figure 4.3. Cette méthode est généralisable pour des blocs de taille quelconque.

La taille du bloc a toutefois un impact sur le nombre de boucles de mises à jour à effectuer pour optimiser les calculs. C'est une méthode qui paraît efficace, mais qui semble plutôt lourde à gérer et qui demande beaucoup de précautions. En effet, la grille de simulation n'étant pas synchronisée à toutes les itérations, il y a possibilité d'être confronté à des cellules qui sont à des itérations différentes et cela demande une certaine prudence pour ne pas fausser la simulation.

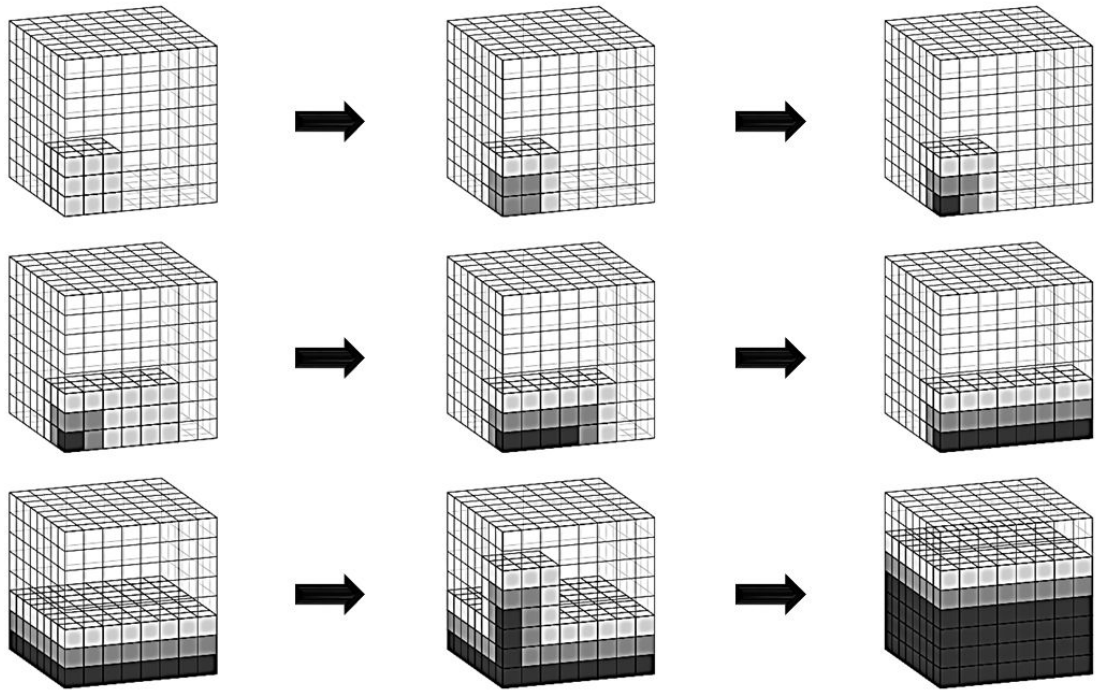


FIGURE 4.3 – Schématisation de l'optimisation des calculs pour un domaine 3D par l'utilisation de blocs de taille 3^3 : toutes les cellules du blocs sont mises à jour à la première boucle, les boucles suivantes ne mettent à jour que les cellules qui ont l'ensemble des plus proches voisins au même niveau de mise à jour [PKW+03]. L'intensité des niveaux de gris d'une cellule est plus forte selon le nombre d'itération réalisé au sein de celle-ci.

4.2.2 Parallélisme hybride sur architectures à mémoire distribuée

L'apparition de grosses architectures de calculs parallèles a également été bénéfique pour la méthode de Boltzmann sur réseau. En effet, l'utilisation d'architectures composées de plusieurs nœuds de calculs, chaque nœud étant composé de plusieurs cœurs, sont des aspects qui ont été abordés au sein de certaines références bibliographiques.

Les travaux de Thürey [Thü07] en 2007 ont dans un premier temps permis de définir une méthode de parallélisation hybride sur des architectures composées de plusieurs

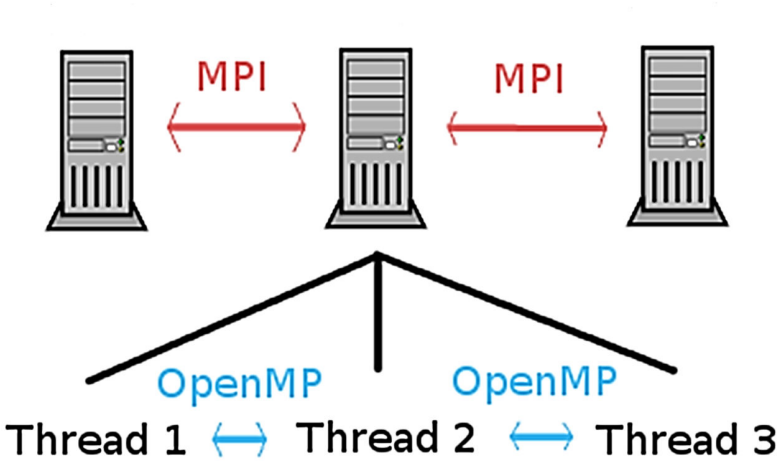


FIGURE 4.4 – Utilisation de la librairie MPI pour répartir les différents nœuds de calculs et de la librairie OpenMP pour répartir les cœurs de calculs de chaque nœud.

nœuds de calculs. Pour cela, il utilise de manière combinée la librairie MPI [GLT99] et la librairie OpenMP. En effet, MPI s'occupe de la répartition des différents nœuds de calculs alors que OpenMP se charge de la répartition des cœurs pour chaque nœud, comme le montre la figure 4.4. L'idée est dans un premier temps de découper le domaine de simulation en sous-domaines selon le nombre de nœuds de calculs disponibles et de réaliser de manière parallèle les calculs sur ces nœuds tout en communiquant les valeurs aux interfaces, comme le montre la figure 4.5.

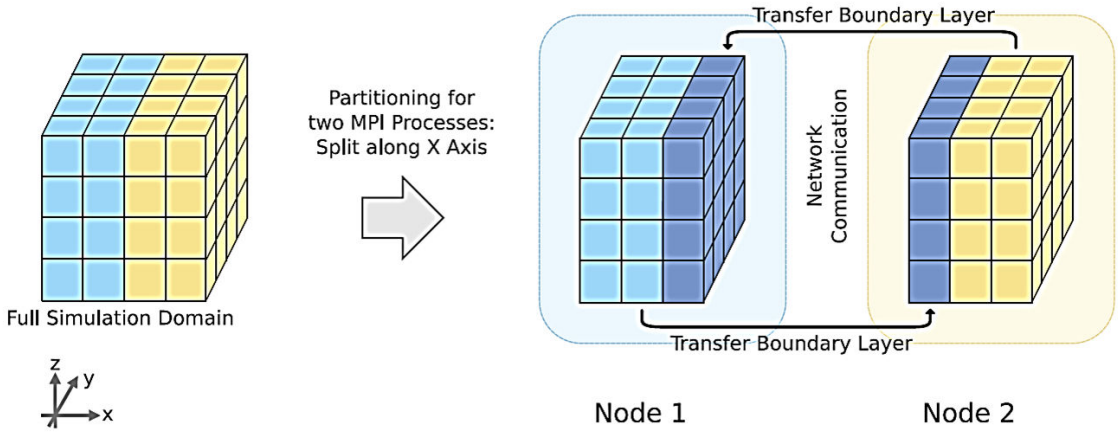


FIGURE 4.5 – Découpage de la grille de simulation selon le nombre de nœuds de calculs disponibles [Thü07].

Pour chaque nœud de calculs, l'utilisation d'OpenMP vient ensuite garantir l'utilisation de l'ensemble des cœurs de la machine. Chaque sous-domaine est alors divisé selon

le nombre de cœurs disponibles et les calculs se font alors de manière parallèle sur ces nouveaux blocs, comme le montre la figure 4.6. Cependant, les travaux de Thürey incluaient à l'époque la méthode de compression de grilles (section 3.2) ce qui limite la capacité de parallélisation d'OpenMP. En effet, des barrières sont nécessaires à chaque niveau de manière à ne pas fausser le comportement de la simulation.

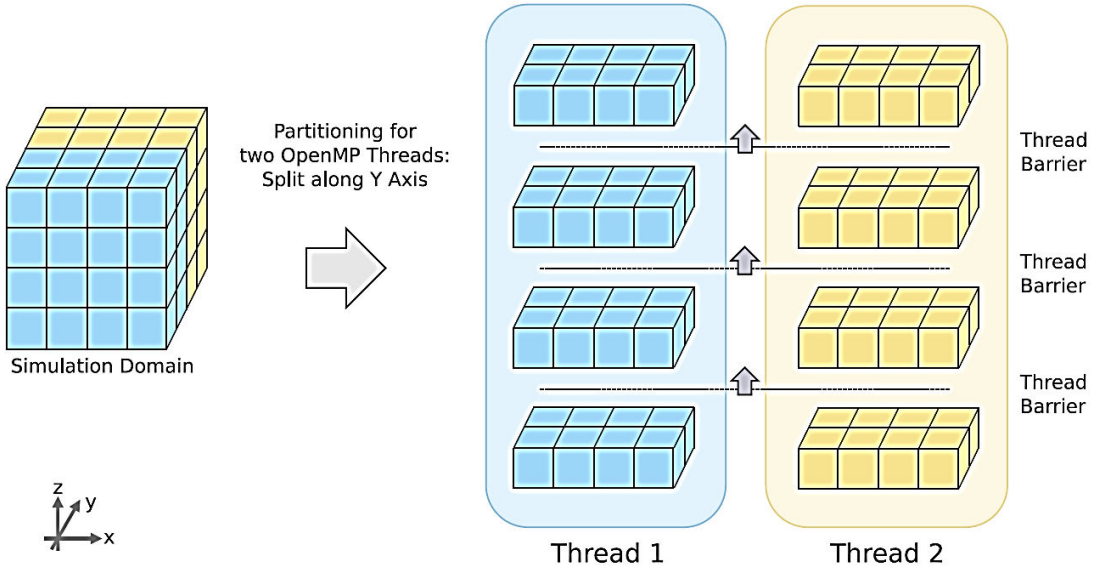


FIGURE 4.6 – Découpage d'un sous-domaine pour deux threads : chaque thread gère un bloc et des barrières sont mises en place afin de ne pas fausser les calculs dus à la méthode de compression de grille [Thü07].

Des travaux plus récents traitent également de parallélisme hybride sur CPU appliqué à des simulations plus complexes, comme la retombée d'une goutte sur un sol mouillé [SCL14]. Toutefois, l'apparition du calcul massivement parallèle utilisant le processeur graphique a déclenché un intérêt très important de la part de la communauté scientifique pour ce type d'architecture. De ce fait, les approches de calculs massifs liées à la méthode de Boltzmann sur réseau sur CPU se font de plus en plus rares.

Toutefois, le parallélisme peut encore être exploré de manière plus profonde sur les processeurs centraux. En effet, les processeurs centraux modernes sont capables de réaliser du calcul SIMD (pour « Single Instruction Multiple Data ») par l'utilisation d'instructions spécifiques. De plus, la méthode de Boltzmann semble fortement adaptée à ce type de parallélisme, les calculs étant identiques pour la majorité des cellules du domaine. Le chapitre suivant décrit la mise en place d'une méthode de parallélisation faisant appel à

ces instructions SIMD permettant d'accélérer les calculs du noyau de simulation.

4.3 Parallélisation hybride du noyau de simulation par l'utilisation combinée d'OpenMP et d'instructions SSE

4.3.1 Introduction

La parallélisation de la méthode de Boltzmann sur réseau sur une architecture composée de processeurs centraux permet d'exploiter de manière efficace les performances de la machine par une utilisation combinée des librairies MPI et OpenMP. En effet, l'ensemble des cœurs de calculs mis à disposition est exploité, chaque cœur étant associé à une partie du domaine de simulation. L'idée est maintenant de réaliser une étude des performances obtenues en utilisant le parallélisme interne des cœurs de calculs modernes. En effet, il est possible de plonger plus profondément dans l'architecture des processeurs modernes et d'en tirer une autre possibilité de parallélisme. La plupart des processeurs centraux modernes disposent d'un ensemble d'instructions spécifiques permettant de réaliser du calcul de type SIMD à l'aide de registres spécifiques internes à chaque cœur de calculs. L'objectif principal est d'exploiter l'utilisation de ces instructions en vue de fournir une parallélisation hybride du noyau de simulation en combinaison avec OpenMP. N'ayant qu'un seul nœud de calculs à notre disposition au sein du laboratoire, nous ne considérerons que l'utilisation d'OpenMP pour séparer les différents cœurs de calculs. Toutefois, l'approche que nous proposons ici pourrait très bien se généraliser sur une architecture composée de plusieurs nœuds de calculs. Cette section se propose d'expliciter les différentes démarches qui ont conduit à la mise en place d'un premier algorithme de calculs parallèle du noyau de simulation. La première étape consiste à définir les géométries de simulation et à les rendre exploitables pour la méthode de Boltzmann. La seconde étape concerne la mise en place d'une parallélisation hybride du noyau de simulation basée sur l'utilisation combinée d'OpenMP et d'instructions SSE (Streaming SIMD Extensions). Les performances issues de cette parallélisation hybride sont finalement étudiées.

4.3.2 Définition de la géométrie d'une simulation

La première étape importante consiste à définir une géométrie exploitable de manière à pouvoir réaliser nos simulations. La méthode de Boltzmann sur réseau fait généralement appel à une grille cartésienne composée de cellules pour réaliser les différentes étapes de calculs. De la même façon, la géométrie représentant le domaine de simulation doit être adaptée à cette grille cartésienne. C'est pourquoi il doit être possible de pouvoir associer à chaque cellule du domaine de simulation un « état géométrique ». Cet état doit rendre compte de la nature de la cellule, dans la majorité des cas obstacle ou cellule fluide. Pour cela, on associe au domaine de simulation un tableau de caractère qui va permettre de connaître l'état de chaque cellule du domaine (Figure 4.7).

C	W	W	W
C	C	W	W
C	C	W	W
C	C	C	C

FIGURE 4.7 – Représentation de l'état géométrique d'une cellule à l'aide d'un tableau de caractère : « C » pour une cellule fluide et « W » pour un obstacle.

Chaque cellule est alors associée à un unique caractère permettant de définir son état géométrique. L'utilisation d'un caractère se justifie par le fait que l'on peut multiplier les possibilités d'états d'une cellule (fluide, solide, entrée de fluide, sortie de fluide, ...) à la différence des booléens qui n'offrent que deux états possibles. De plus, la représentation de l'état géométrique par un caractère permet une consommation mémoire réduite par rapport à l'utilisation de nombres entiers. Toutefois, la représentation géométrique que nous avons définie ne correspond généralement pas à la représentation de modèles géométriques. En effet, un modèle géométrique est généralement défini par un ensemble de polygones représentant la surface des objets intervenant dans une scène.

La mise en place d'outils permettant de transformer un modèle géométrique en une géométrie exploitable pour les simulations est donc nécessaire. La littérature offre un

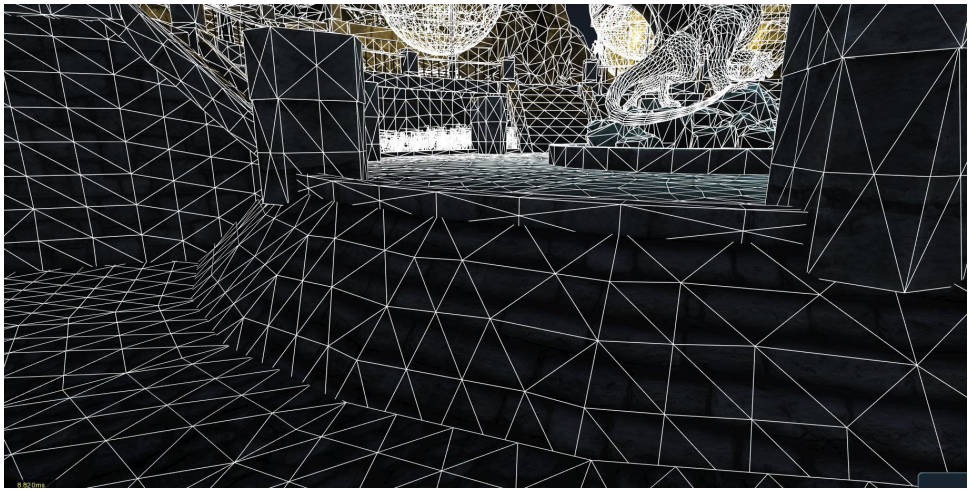


FIGURE 4.8 – Exemple de représentation géométrique surfacique (source internet).

grand choix de méthodes permettant de faire cette transformation à l'aide d'une voxelisation [TGR04]. Le but est principalement ici d'exploiter une idée simple à mettre en place et non de concurrencer les méthodes existantes. Elle consiste simplement à définir pour chaque cellule de la grille cartésienne si elle est située à l'intérieur d'un obstacle ou non. La difficulté principale est donc de déterminer si un point est situé à l'intérieur d'un objet de la géométrie ou non. Différentes solutions existent pour résoudre ce type de problème. L'idée que nous utilisons est de considérer dans un premier temps une demi-droite parallèle à un des axes du repère orthonormé usuel passant par le point à déterminer. Compter le nombre d'intersections de cette droite avec les objets de la scène permet ensuite de déterminer si le point est à l'intérieur de ces obstacles (Figure 4.9). Si le nombre d'intersections est impair, alors le point est à l'intérieur de l'obstacle et inversement si le nombre d'intersections est pair alors le point est en dehors.

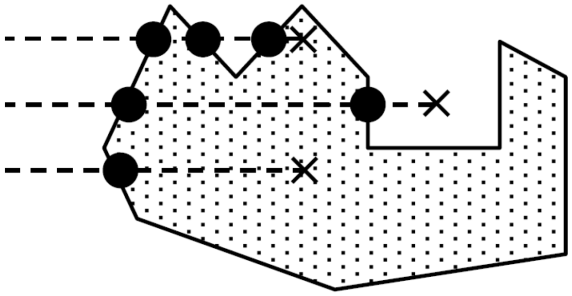


FIGURE 4.9 – Méthode permettant de trouver si un point est à l'intérieur d'un obstacle : un nombre impair d'intersections implique que le point est à l'intérieur d'un obstacle et un nombre pair implique que le point est en dehors de l'objet.

Les modèles géométriques couramment utilisés sont des géométries à surface triangulaire. Ainsi, pour trouver l'intersection entre une droite et un obstacle, il faut tester l'intersection de la droite avec l'ensemble des triangles composant l'obstacle. Pour accélérer les calculs, on divise la grille cartésienne en fonction du nombre de cœurs disponibles et les calculs se font en parallèle sur chaque sous-domaine.

Les géométries utilisées dans le cadre de ce manuscrit restent toutefois des géométries simplifiées dans le sens où le niveau de détail n'est pas le même que celui utilisé par exemple en informatique graphique. Une méthode plus poussée de voxelisation est par conséquent à envisager pour des simulations sur des domaines géométriques avec un niveau de détail important.

4.3.3 Répartition de la charge de calculs

L'étude des références bibliographiques a apporté un certain nombre de solutions permettant de paralléliser de manière efficace un code de modélisation de Boltzmann sur réseau. Comme l'ensemble des références étudiées en littérature, l'utilisation d'OpenMP est une alternative simple et efficace permettant de répartir la charge de calculs sur les différents cœurs du processeur central. En effet, de simples directives de programmation permettent de répartir la charge de calculs, de mettre en place des barrières de synchronisation et de partager des variables entre les threads. Le domaine de simulation est ensuite découpé en sous-domaines selon le nombre de cœurs disponibles, et ces sous-domaines sont ainsi calculés de manière parallèle [MA02] (Figure 4.1). L'utilisation de la méthode A-A pattern permet de réduire de manière importante la mémoire et de rapprocher les données en mémoire (section 3.5). L'utilisation d'une méthode de calculs par blocs (section 4.2.1) permettant une bonne gestion des caches du processeur central [PKW+03] pourrait également être intéressante, mais dans un cadre de calculs SIMD, la mise en place de ce type de méthode semble très complexe. En effet, pour des données voisines, on peut avoir une quantité de calculs qui est différente, ce qui est pénalisant pour un algorithme de calculs SIMD. Par conséquent, l'utilisation d'une telle méthode n'est pas considérée pour la suite des travaux.

En plus de cette première parallélisation, une seconde parallélisation basée sur l'utilisation d'instructions SSE [PW96] est mise en place. Le SSE est un lot d'instructions SIMD pour les architectures x86 développé par Intel en 1999. Il contenait à l'époque 70 instructions, la plupart fonctionnant en simple précision. Ces instructions SIMD peuvent améliorer de manière importante les performances quand les mêmes calculs sont réalisés sur plusieurs données. Plusieurs domaines d'applications sont possibles pour cette technologie et cette dernière semble adéquate dans le cas de la méthode de Boltzmann sur réseau.

Le SSE a originellement ajouté huit nouveaux registres 128 bits nommés XMM0 à XMM7. Les extensions x64 d'Intel et AMD ajoutent huit nouveaux registres de XMM8 à XMM15. Chaque registre compacte ensemble quatre nombres flottants de simple précision (32 bits) ou deux nombres flottants à double précision (64 bits). Les instructions liées au SSE regroupent un ensemble d'opérations arithmétiques permettant de réaliser du calcul sur ces registres. De manière simple, ces instructions permettent alors d'exécuter quatre opérations de manière parallèle pour la simple précision (ou deux pour la double précision), comme le schématise la figure 4.10. Les bénéfices qui peuvent être apportés en matière de performances par les instructions SSE sont par conséquent trop importants pour être ignorés.

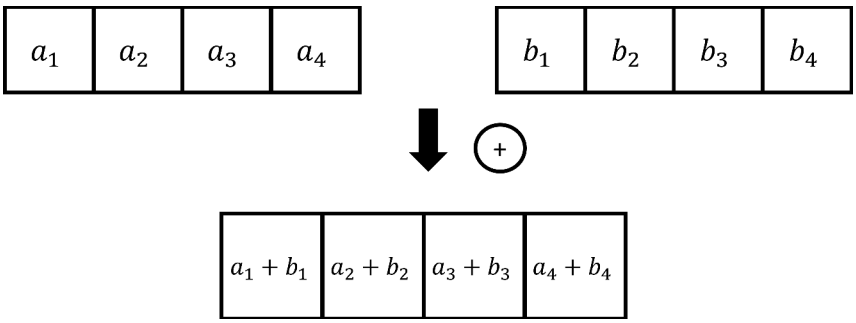


FIGURE 4.10 – Illustration simplifiée du fonctionnement des instructions SSE.

De nos jours, des outils permettent de vectoriser automatiquement un code à l'aide du compilateur. Cependant, ces outils offrent généralement des performances assez pauvres en comparaison à une solution manuelle. En effet, certains facteurs doivent être respectés de manière à obtenir de bonnes performances. Le premier concerne l'alignement des

données. De manière à réduire le trafic mémoire au sein des registres, il est préférable de lire des données alignées sur 128 bits. Une seule instruction est alors nécessaire pour lire les données alors que plusieurs sont nécessaires pour des données non alignées. L'organisation des calculs est également un facteur important pour obtenir des performances importantes. Comme dans un cadre de développement classique, la réduction du nombre de calculs est un facteur d'optimisation important. En effet, la réduction de la quantité de calculs permet un nombre d'instructions moins important et donc de meilleures performances, notamment dans un cadre d'applications impliquant du calcul intensif comme la méthode de Boltzmann sur réseau.

L'intégration de la parallélisation faisant appel aux instructions SSE va se faire naturellement au sein de chaque sous-domaine défini pour chaque thread OpenMP. Ainsi, pour chaque sous-domaine, les calculs vont être réalisés de manière SIMD (Figure 4.11). Les données de simulation étant stockées dans des tableaux, la direction d'alignement des données est choisie de manière à réduire le trafic mémoire.

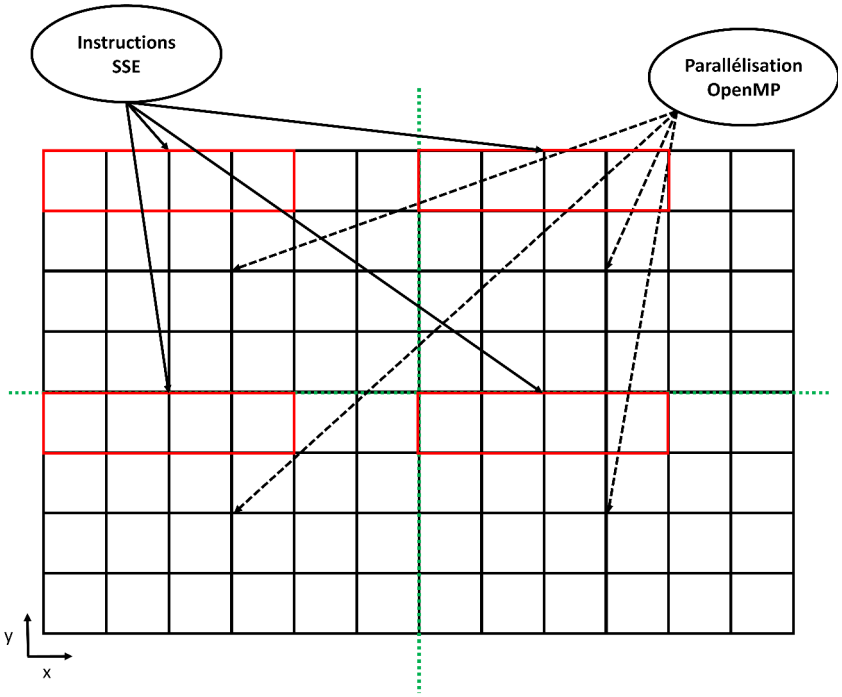


FIGURE 4.11 – Parallélisation hybride utilisant OpenMP et les registres SSE : chaque thread OpenMP se voit affecter un sous-domaine de la grille de simulation. Pour chaque sous-domaine, les calculs se font de manière SIMD par l'utilisation d'instructions SSE.

Suivant l'architecture utilisée, plusieurs évolutions de ces registres sont apparues. Sur

les processeurs centraux récents, la taille des registres a été doublée, pour passer de 128 bits à 256 bits, ce qui devrait aboutir à des performances encore plus importantes. Plus récemment encore, des coprocesseurs de calculs développés par Intel (coprocesseur Xeon Phi) désirant exploiter le parallélisme SIMD ont été commercialisés. Ces coprocesseurs disposent d'un grand nombre de processeurs vectoriels réalisant des calculs sur des registres de 512 bits. La parallélisation hybride mise en place dans cette section resterait valable pour ces architectures récentes et devrait offrir des performances intéressantes.

4.3.4 Traitement des conditions aux bords et des obstacles

L'avantage de la parallélisation SIMD est qu'elle permet de réaliser des calculs en parallèle de manière efficace à la condition que les calculs soient réalisés de la même façon sur toutes les données. De manière globale, la méthode de Boltzmann sur réseau et le noyau de simulation mis en place pourraient bien se prêter à ce type de parallélisation, car la plupart des calculs se font de la même façon sur l'ensemble des données. Toutefois, les conditions aux limites et aux obstacles doivent être généralement traitées de manière différente, ce qui peut être un facteur pénalisant pour ce type de parallélisation. En effet, cela impose de séparer les différents traitements à l'aide de structures conditionnelles et l'intérêt du calcul SIMD est ainsi diminué. Cette partie traite donc de la méthode mise en place pour traiter ces conditions de manière efficace.

La section 2.3.5 mettait en évidence la façon dont sont généralement traitées les conditions limites ou aux obstacles pour la méthode de Boltzmann sur réseau. Les règles particulières sont généralement limitées à une partie du domaine dépendante de la géométrie. Même si ces conditions concernent une infime partie du domaine de simulation, appliquer une structure conditionnelle implique un test de condition sur l'ensemble des cellules de la grille. Ce test peut alors s'avérer être une grosse perte de temps et donc de performances. Une manière différente de traiter ces conditions doit alors être définie. Elle doit respecter le fait que la parallélisation SIMD implique la réalisation de la même règle de calcul pour toutes les données. L'idée est donc d'appliquer la même règle de calcul sur l'ensemble des données et de venir à l'issue de cette règle corriger les données liées aux conditions de manière à ne pas propager les erreurs commises. Appliquer cette méthode

n'est toutefois possible qu'à la condition d'être en mesure de pouvoir corriger les erreurs commises.

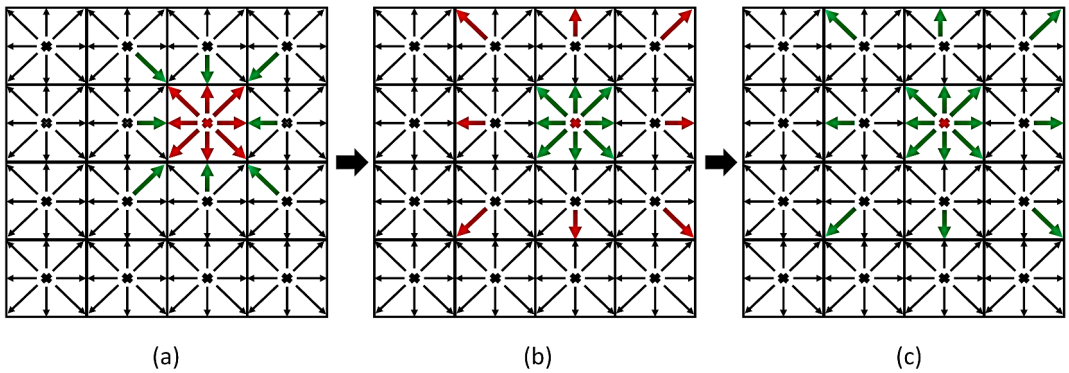


FIGURE 4.12 – Traitement de la condition aux obstacles pour la méthode de Boltzmann sur réseau avec utilisation de la parallélisation SIMD : (a) La cellule en rouge définit ici un obstacle ; (b) la collision et la propagation des données est appliquée de façon SIMD sur l'ensemble de la grille ; (c) les valeurs des fonctions de distribution au niveau des obstacles sont corrigées.

La figure 4.12 met en évidence le comportement du traitement des obstacles dans un cadre SIMD. L'idée est d'annuler les erreurs commises par les calculs SIMD au sein de la grille par une correction adaptée au niveau des obstacles et des conditions limites. En réalité, les étapes de collision-propagation des fonctions de distribution f_i au sein d'une cellule obstacle propagent une erreur vers ses plus proches voisins (flèches rouges de la figure 4.12 (b)). Les bonnes valeurs permettant de réaliser la condition au niveau de l'obstacle restent toutefois accessibles au sein de la cellule obstacle. Le but est par conséquent de reprendre ces bonnes valeurs et de les appliquer aux plus proches voisins afin de recouvrir les bonnes valeurs de simulation (flèches vertes de la figure 4.12 (c)). Pour cela, répertorier les obstacles est nécessaire au préalable afin de pouvoir aisément les cibler et leur apporter cette correction.

Cette méthode de correction est une méthode qui se prête bien à la méthode de Boltzmann sur réseau et également à l'intégralité du noyau de simulation. L'intégralité des étapes de calculs de l'algorithme décrit au sein de l'algorithme 1 peut être ainsi traité de manière SIMD. Une correction adaptée est ensuite réalisée à l'issue de ces étapes aux bords du domaine de simulation et au niveau des obstacles afin de ne propager aucune erreur pendant la simulation.

4.4 Résultats et performances

Cette section a pour but de résumer les précédentes sections par un algorithme et de mettre en évidence les performances obtenues pour cette première parallélisation. L'idée est de réaliser une comparaison de performances entre plusieurs modes d'implémentations. Le premier et le plus basique réalise l'ensemble des calculs de manière séquentielle sur le processeur central. Le second mode inclut la répartition des calculs sur les différents cœurs à l'aide d'OpenMP seul. Le dernier mode inclut finalement la combinaison d'OpenMP avec l'utilisation d'instructions SSE dans le noyau de simulation.

4.4.1 Algorithme

Cette section résume les différents changements apportés à l'algorithme du noyau de simulation de l'algorithme 1. L'inclusion d'une méthode de réduction de mémoire efficace ainsi qu'une parallélisation hybride SIMD incluant des instructions SSE provoquent des modifications majeures au niveau de l'algorithme du noyau de simulation. Ces modifications sont décrites à l'aide de l'algorithme 2.

Cet algorithme inclut les deux pas de temps successifs distincts *A-step* et *B-step* de la méthode de A-A pattern (section 3.5). Cela a pour effet de devoir gérer deux versions concernant plusieurs étapes de calculs de l'algorithme. Plusieurs étapes de corrections sont également ajoutées au sein de l'algorithme afin de pouvoir réaliser les calculs de façon totalement SIMD et de ne pas propager d'erreurs au niveau des conditions aux limites et des obstacles.

4.4.2 Simulations réalisées

Pour réaliser cette comparaison, plusieurs simulations exploitant l'ensemble du noyau de simulation ont été réalisées. La première est une simulation de condensation réalisée sur un domaine 2D et 3D (Figure 4.13). La formation de bulles à température constante se fait progressivement au sein du domaine jusqu'à la convergence de la simulation. La taille du domaine pour cette simulation est de 128^3 cellules.

La seconde simulation est une simulation 2D et 3D d'épandage sur une géométrie légèrement plus complexe (Figure 4.14). Une fuite est définie au sein du domaine et elle

Algorithme 2 : Algorithme du noyau de simulation incluant la méthode de réduction de mémoire A-A pattern ainsi qu'une parallélisation hybride SIMD.

↪ Initialisation de la simulation ;
↪ Découpage du domaine de simulation en sous-domaines selon le nombre de cœurs de calculs ;
pour *chaque itération* **faire**
  pour *chaque thread OpenMP* **faire**
    pour *chaque composant* σ **faire**
      ↪ Calcul SIMD (instructions SSE) de la pression p_σ à l'aide de l'équation (2.23) ;
      ↪ Calcul SIMD (instructions SSE) du pseudo-potentiel ψ_σ à l'aide de l'équation (2.24) ;
      ↪ Corrections des erreurs aux conditions aux limites et aux obstacles ;
    fin
  fin
  pour *chaque thread OpenMP* **faire**
    pour *chaque composant* σ **faire**
      ↪ Calcul SIMD (instructions SSE) des forces à l'aide des équations (2.25)-(2.28) ;
      ↪ Calcul SIMD (instructions SSE) de la première fonction d'équilibre $f^{eq}_{\sigma,i}$ à l'aide de l'équation (2.7));
      ↪ Somme des forces et correction SIMD (instructions SSE) du terme de vitesse Δu_σ à l'aide de l'équation (2.31) ;
      ↪ Calcul SIMD (instructions SSE) de la seconde fonction d'équilibre $f^{eq}_{\sigma,i}$ à l'aide de l'équation (2.7);
      si *itération paire* **alors**
        ↪ Calcul SIMD (instructions SSE) des temps de relaxations à l'aide d'un modèle de sous-maille de Smagorinsky pour le *A-step* de la méthode A-A pattern;
        ↪ Étape de collision-propagation SIMD (instructions SSE) des fonctions de distribution $f_{\sigma,i}$ pour le *A-step* de la méthode A-A pattern;
        ↪ Corrections aux conditions limites et aux obstacles pour le *A-step* de la méthode A-A pattern;
        ↪ Calcul SIMD (instructions SSE) des quantités macroscopiques ρ_σ et u_σ pour le *A-step* de la méthode A-A pattern à l'aide des équations (2.10) et (2.11) ;
      sinon
        ↪ Réalisation des mêmes étapes de calculs que pour les itérations paires mais dans le cas du *B-step* de la méthode A-A pattern ;
      fin
    fin
  fin
fin

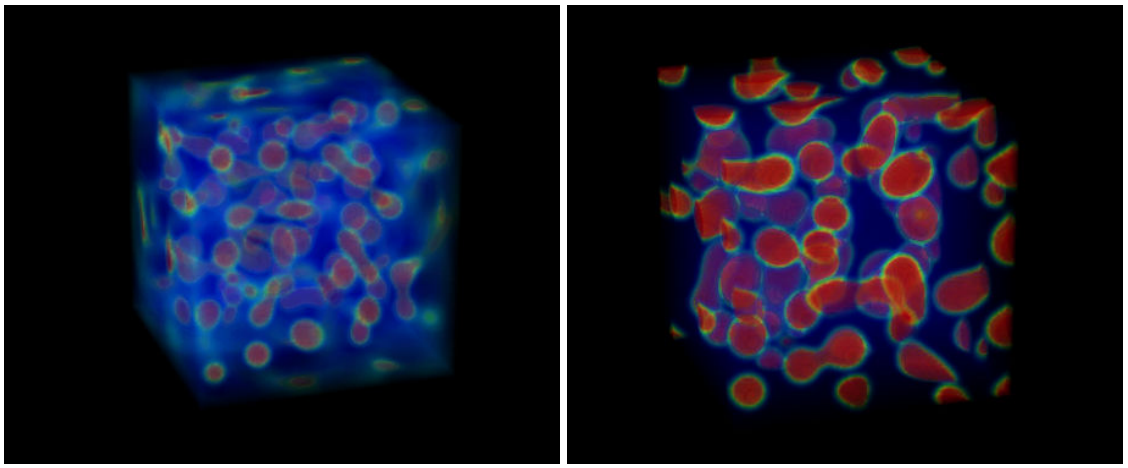


FIGURE 4.13 – Illustrations d'une simulation 3D de condensation avec un schéma D3Q19.

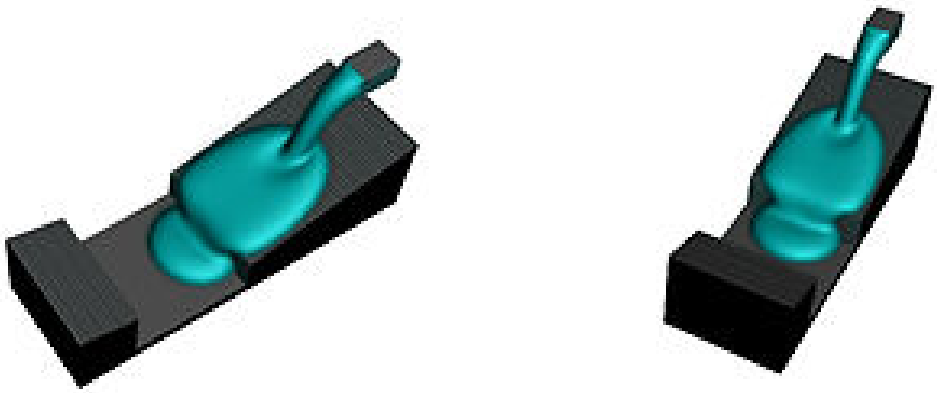


FIGURE 4.14 – Illustrations d'une simulation 3D d'épandage avec un schéma D3Q19.

se propage dans l'ensemble du domaine de simulation. La taille du domaine pour cette simulation est de $300 \times 128 \times 128$ cellules. Ces simulations impliquent la présence de deux composants physiques au sein du domaine de calculs.

4.4.3 Architecture de calculs

De la même façon que pour le choix de la méthode de réduction de mémoire, les tests ont été réalisés à l'aide d'une machine composée de deux processeurs Intel Xeon X5660 dont les caractéristiques sont définies dans le tableau 3.1. Notons toutefois que les

instructions SIMD de cette architecture sont limitées aux instructions SSE, contrairement aux nouvelles architectures pouvant utiliser des instructions AVX de plus grande capacité.

4.4.4 Performances

Cette section réalise une comparaison de performances entre plusieurs modes de parallélisation à l'aide des simulations décrites dans la section 4.4.2. Le but est d'évaluer l'efficacité de l'algorithme 2 et particulièrement l'inclusion des instructions SSE dans le code de simulation.

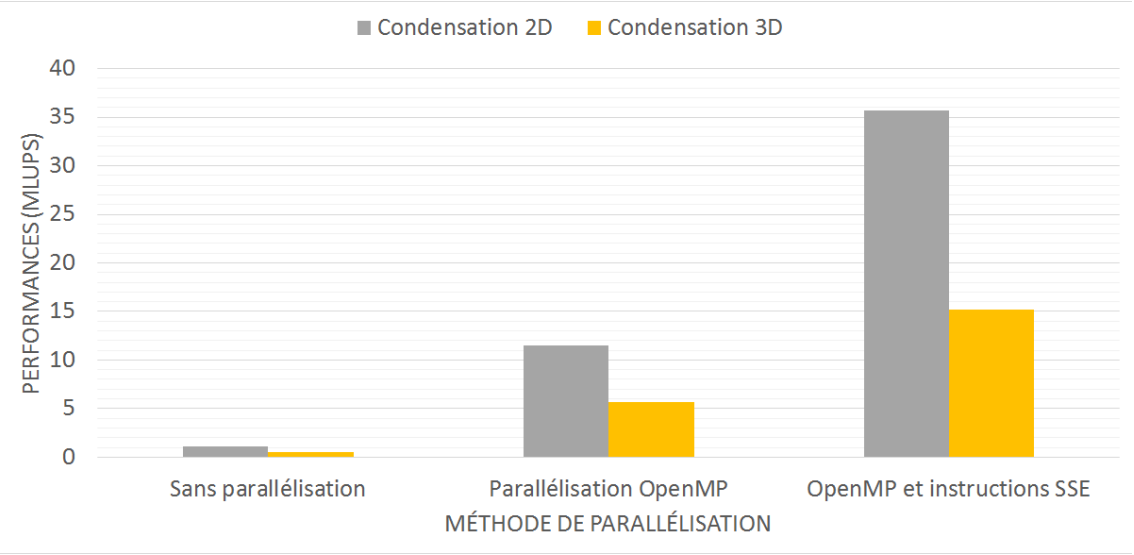


FIGURE 4.15 – Comparaison de performances pour des simulations 2D et 3D de condensation.

La figure 4.15 illustre les performances obtenues pour les simulations de condensation (Figure 4.13). Notons dans un premier temps que les performances obtenues pour une simulation purement séquentielle sont extrêmement basses (≈ 0.54 MLUPS pour la simulation 3D), ce qui rend cette approche difficilement exploitable dans le cadre d'une simulation de grande taille, comme celle qui est visée dans le cadre de notre projet. L'intégration de la répartition des différents cœurs de calculs à l'aide d'OpenMP provoque un gain de performances non négligeable. Le gain de performance obtenu est la plupart du temps proportionnel au nombre de cœurs à notre disposition (Figure 4.16). Ces performances tiennent toutefois compte de « l'hyperthreading » des cœurs de calculs. Une optimisation plus fine du code de simulation et de l'utilisation de la librairie OpenMP pourrait par conséquent mener à de meilleures performances et à une évolution parfaite

suivant le nombre de cœurs.

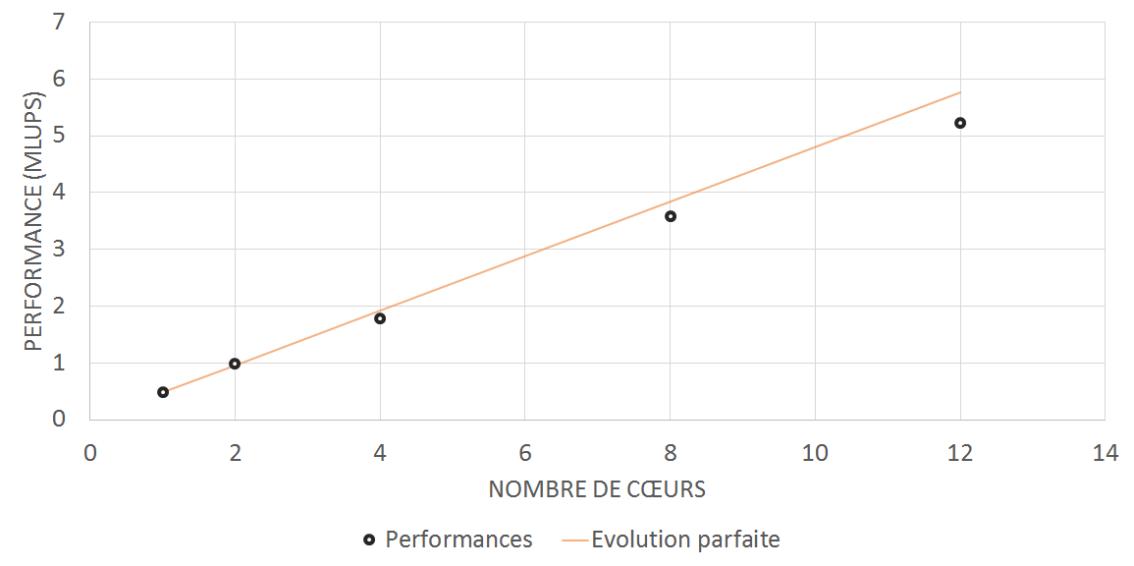


FIGURE 4.16 – Évolution de performances pour une simulation 3D de condensation en fonction du nombre de cœurs.

L'intégration d'instructions SIMD basées sur l'utilisation des registres SSE provoque finalement un gain supplémentaire important ($\approx$ 15.2 MLUPS pour la simulation 3D). Pour ces simulations, on note une gain de performances moyen d'un facteur d'environ 3 par le simple ajout de ces instructions. Le gain obtenu est intéressant mais toutefois pas optimal. En effet, le gain théoriquement atteignable est d'un facteur 4 car les calculs sont réalisés pour quatre nombres flottants en parallèle. L'importante quantité de calculs ainsi que de nombreuses lectures et écritures au sein des registres pourraient expliquer le fait que ce gain théorique n'est pas obtenu. En réalité, une opération arithmétique simple dans un cas normal est cette fois représentée par une ou plusieurs instructions SIMD (Listing 4.1).

Listing 4.1 – Addition de deux tableaux à l'aide d'instructions SSE

```
void SSE_ADD(float *a, float *b, float* c){
  for(i=0; i<SIZE; i+=4){
    __m128 a_sse = _mm_load_ps(&a[i]) ;
    __m128 b_sse = _mm_load_ps(&b[i]) ;
    __m128 c_sse = _mm_add_ps(a_sse,b_sse) ;
    _mm_store_ps(&c[i], c_sse) ;
  }
```

}

Cela conduit à des codes de calculs très volumineux et demande un gros travail de développement informatique notamment pour le noyau de simulation qui demande une grosse quantité de calculs. Des optimisations plus fines du code de simulation SIMD pourraient sans doute améliorer le gain. Une meilleure réduction du nombre d'instructions à utiliser tant sur le plan calculatoire que sur la lecture et l'écriture des données sur les registres devrait permettre un gain supplémentaire pour les performances.

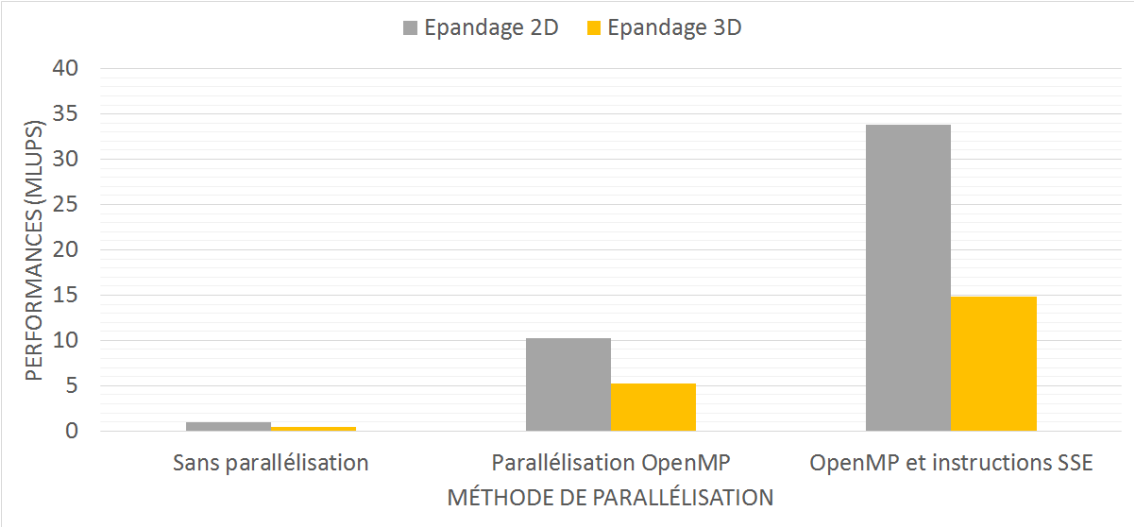


FIGURE 4.17 – Comparaison de performances pour des simulations d'épandage 2D et 3D.

Le même constat est à noter pour les simulations d'épandage (Figure 4.14). Nous constatons toutefois une légère diminution de performances en comparaison à la simulation de condensation ($\approx$ 14.8 MLUPS). Cela est principalement dû à la complexité de la géométrie qui est plus importante. En effet, cette simulation comporte un nombre d'obstacles beaucoup plus important que dans le cas de la condensation, les cellules correspondant aux obstacles devant être gérées en mode non SSE, ce qui impacte sur les performances.

4.5 Conclusion

Ce chapitre a permis d'établir différentes notions permettant la mise en place d'une première parallélisation du noyau de simulation sur une architecture composée d'un processeur central multi-cœurs. Cela passe tout d'abord par un choix adapté d'une méthode

de réduction de mémoire, de manière à réduire la quantité mémoire requise, tout en conservant de bonnes propriétés pour le parallélisme. Réduire la quantité de mémoire est également une étape importante de manière à pouvoir simuler sur de grandes grilles de calculs. Une étude de performances a ainsi conduit au choix de la méthode A-A pattern qui offre les meilleures performances en comparaison aux autres méthodes existantes en littérature.

De la même façon, ce chapitre a mis en évidence des méthodes permettant de paralléliser efficacement une méthode de Boltzmann sur réseau utilisant plusieurs processeurs centraux en parallèle. Dans le cadre de ce projet et des architectures de calculs à notre disposition, nous en avons déduit une parallélisation hybride du noyau de simulation basée sur l'utilisation d'OpenMP (pour répartir les calculs sur les différents cœurs disponibles) et d'instructions SSE pour obtenir une seconde parallélisation SIMD des calculs au sein de chaque cœur. Nous avons montré à travers plusieurs simulations que cela permettait d'améliorer de manière importante les performances de la simulation. Ces travaux peuvent évidemment s'étendre à une architecture plus complexe composée de plusieurs nœuds de calculs. La mise en place du code SSE demande toutefois un effort de programmation important pour obtenir des performances intéressantes, ce qui représente le principal inconvénient de cette technologie.

Ce premier travail a finalement permis de mettre en place un algorithme SIMD du noyau de simulation. Cela a permis d'appréhender un certain nombre de difficultés liées à la parallélisation SIMD. De ce fait, cette étude peut également servir d'étude préliminaire à un passage du noyau de simulation sur le processeur graphique qui utilise un mode de parallélisation similaire.

Troisième partie

Parallélisme et optimisations du noyau de simulation par l'utilisation de processeurs graphiques

Chapitre 5

Parallélisation de la méthode de Boltzmann par l'utilisation de processeurs graphiques

Sommaire

5.1 Motivation

L'utilisation des processeurs graphiques (GPU) pour réaliser du calcul générique à hautes performances a suscité l'attention de la communauté scientifique ces dernières années. La puissance de calculs du GPU est de nos jours capable de surpasser celle des processeurs centraux usuellement utilisés pour des applications de calculs intensifs. De plus, le parallélisme massif des GPUs semble très bien s'adapter à la méthode de Boltzmann sur réseau. Le but de ce chapitre est de définir un certain nombre de notions permettant la mise en place d'une parallélisation du noyau de simulation sur une architecture de calculs composée de plusieurs processeurs graphiques.

Une connaissance de l'architecture et du fonctionnement du GPU est en premier lieu nécessaire, afin de pouvoir en exploiter au mieux les capacités. Ensuite, le modèle de programmation de l'environnement CUDA (pour « Compute Unified Device Architecture ») est introduit. En effet, la façon de programmer un code de simulation sur un processeur graphique diffère de façon importante de la programmation classique sur processeur central. Un état de l'art des travaux existants en littérature concernant la méthode de Boltzmann sur GPUs est également réalisé. Cela comprend les travaux réalisés sur un seul processeur graphique, un nœud de calculs composé de plusieurs processeurs graphiques ou encore sur une grosse installation de calculs composée de plusieurs nœuds de calculs, eux mêmes constitués de plusieurs processeurs graphiques.

5.2 Introduction au parallélisme massif sur processeur graphique

Les processeurs graphiques programmables ont évolué au fil des années jusqu'à devenir des processeurs multi-cœurs massivement parallèles, fournissant ainsi une énorme puissance de calculs et une bande passante mémoire très élevée, comme l'illustrent les figures 5.1 et 5.2. Cette évolution est essentiellement due à la demande croissante d'applications en temps réel et en graphismes 3D haute définition.

La raison principale derrière la différence de performances entre le processeur central et le GPU provient du fait que ce dernier est spécialisé pour le calcul intensif, hautement

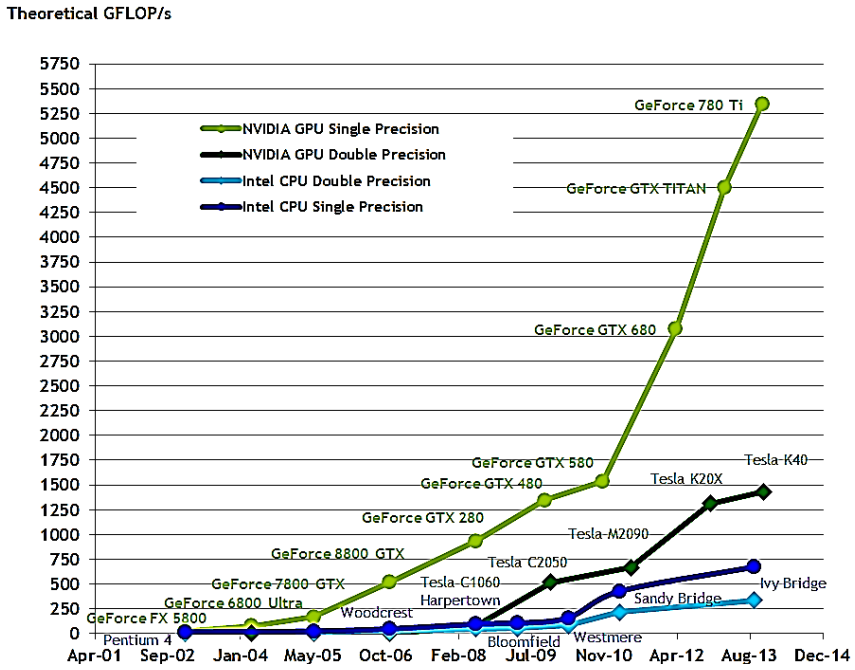


FIGURE 5.1 – Évolution au fil des années de la puissance de calculs des processeurs graphiques en comparaison au processeur central (source Nvidia).

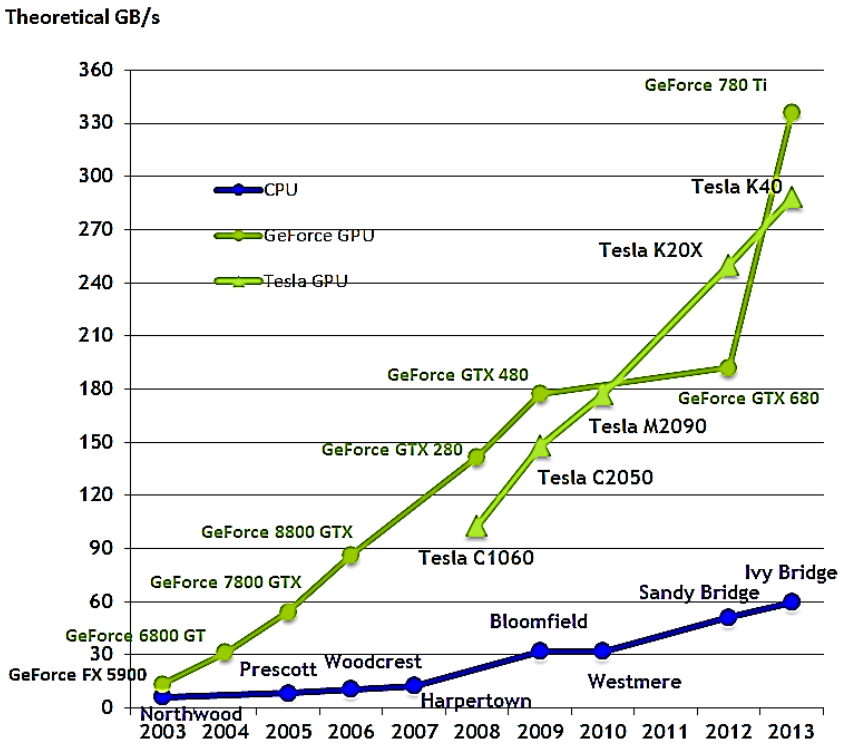


FIGURE 5.2 – Évolution au fil des années de la bande passante mémoire des processeurs graphiques en comparaison au processeur central (source Nvidia).

parallèle, c'est-à-dire exactement ce que les applications graphiques nécessitent. Il est donc conçu de façon à ce que les transistors soient utilisés au traitement direct des don-

nées plutôt qu'à leur mise en cache et au contrôle de flux d'instructions. Plus précisément, le GPU est particulièrement adapté pour traiter des problèmes qui peuvent être exprimés comme le calcul de plusieurs données en parallèle, c'est-à-dire que le même programme est exécuté sur plusieurs éléments de données en parallèle. Comme le programme est exécuté pour chaque élément de données, cela demande une exigence bien moins importante pour le contrôle des flux de données. De la même façon, comme le programme est exécuté sur un nombre important de données et de manière arithmétiquement intensive, la latence des accès mémoire peut être camouflée par du calcul au lieu d'utiliser de gros caches de données.

Cette section décrit en premier lieu un bref historique concernant l'évolution des GPUs et les éléments qui ont conduit à cette architecture de calculs massivement parallèle. L'architecture des GPUs Nvidia et le modèle de programmation CUDA sont finalement introduits, afin de se familiariser avec ce nouveau mode de programmation.

5.2.1 Historique extrait de [SK11]

Les processeurs graphiques sont normalement destinés à réduire la charge du processeur central pour ce qui concerne le rendu graphique. À la fin des années 1980 et au début des années 1990, la popularité grandissante des systèmes d'exploitation utilisant des ressources graphiques, tel que Microsoft Windows, a contribué à l'apparition de ce nouveau type de processeurs. Les ordinateurs personnels ont donc commencé à se doter d'accélérateurs graphiques 2D permettant d'améliorer les affichages issus de ces systèmes.

À peu près à la même époque, la société Silicon Graphics popularisa l'utilisation de graphismes 3D pour un grand nombre de domaines tels que le militaire, la science ou encore la médecine[Sin13]. Elle procurait également des outils permettant de créer des effets animés étonnants pour l'époque. Silicon Graphics introduisit finalement en 1992 l'interface de programmation OpenGL. L'objectif était de fournir une méthode uniformisée et indépendante du matériel afin de pouvoir créer des applications graphiques en trois dimensions.

L'évolution des jeux vidéos dans les années 1990 a beaucoup contribué à la demande croissante d'applications graphiques en 3D. À la même époque, des sociétés comme Nvidia, ATI Technologies et 3dfx Interactive ont démarré la production d'accélérateurs gra-

phiques pour le grand public. L'apparition de la carte Geforce 256 de Nvidia repoussa à nouveau les limites des possibilités des cartes graphiques. Elle permettait en effet d'exécuter directement des traitements liés aux transformations et à la lumière, offrant ainsi la possibilité d'obtenir des applications visuellement plus attrayantes. L'apparition de cette carte provoqua le début d'une progression vers laquelle certains traitements graphiques pourraient être directement implémentés au sein du processeur graphique. Du côté parallélisme, l'apparition de la série Geforce 3 de Nvidia en 2001 représente très certainement l'étape la plus importante en ce qui concerne la technologie GPU. En effet, c'était le premier circuit qui supportait le nouveau standard DirectX 8.0 de Microsoft. Ce standard imposait que les matériels compatibles devaient proposer à la fois des « vertex shaders », ainsi que des « pixels shaders » programmables. Cela permettait aux développeurs de pouvoir contrôler les traitements effectués sur le GPU.

Ce n'est finalement que cinq années plus tard que le traitement sur GPU va devenir accessible pour tous. En effet, Nvidia révéla en 2006 le premier GPU compatible DirectX 10, la Geforce 8800 GTX, qui fut également le premier GPU à intégrer l'architecture CUDA de Nvidia. Cette architecture a effectivement introduit de nouveaux composants conçus spécifiquement pour les traitements GPUs généraux et par conséquent supprimé la plupart des restrictions des GPUs précédents.

5.2.2 Architecture des processeurs graphiques Nvidia

Les processeurs centraux usuellement utilisés sont généralement composés de plusieurs cœurs de calculs ainsi que d'une hiérarchie mémoire composé de registres, de mémoires cache et d'une mémoire RAM. Il est de plus capable d'utiliser le processeur graphique comme un coprocesseur pour certains calculs adaptés aux architectures SIMD. L'architecture des GPUs Nvidia compatibles CUDA est toutefois bien différente de l'architecture d'un processeur central. Le processeur graphique a en effet pour but d'être massivement parallèle et adapté pour des applications de calculs intensifs. Il est pour cela composé d'un ensemble de N multiprocesseurs, chaque multiprocesseur étant également composé de M processeurs scalaires. Le GPU est ainsi constitué d'un nombre important de processeurs de calculs, atteignant 448 cœurs dans le GPU GF100 datant de 2010 des cartes Tesla C2050 utilisées dans le cadre de ces travaux, et plus de 3000 dans le GPU

GM200 équipant la carte GeForce GTX Titan X sortie en 2015 (Figure 5.3).

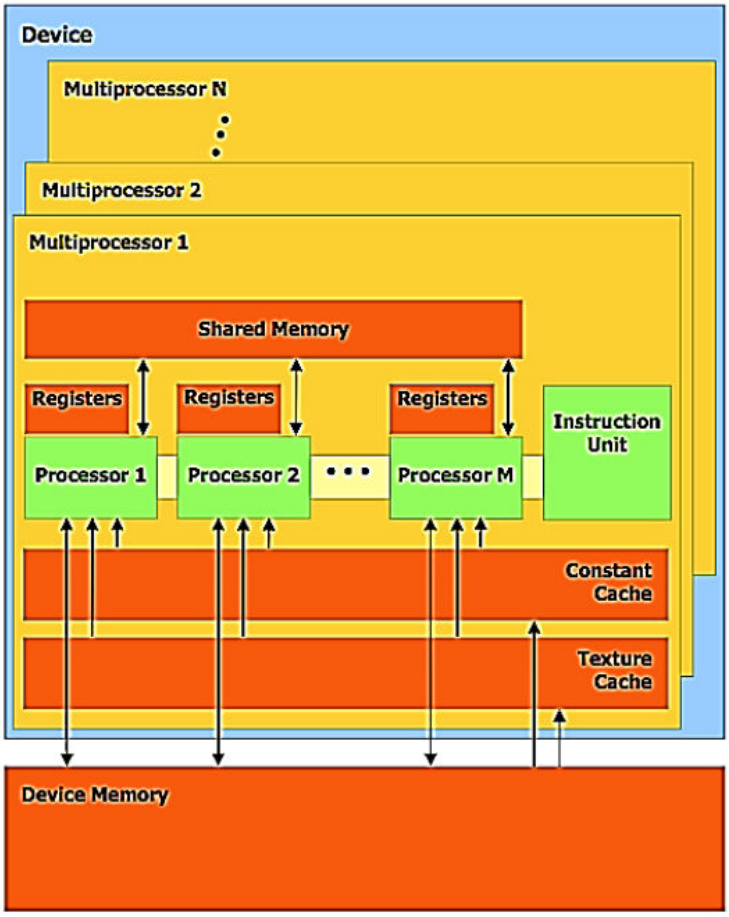


FIGURE 5.3 – Architecture d'un GPU Nvidia compatible CUDA (source Nvidia).

Le GPU dispose également d'une hiérarchie mémoire particulière. Il est constitué d'une mémoire globale de grosse capacité accessible par l'ensemble du GPU mais relativement lente d'accès. Des zones mémoire spécifiques pour les constantes et les textures sont également intégrées au sein du GPU. Chaque multiprocesseur dispose également d'une mémoire partagée de faible capacité mais accessible très rapidement. Des registres à accès rapide sont finalement associés à chaque processeur scalaire. Cette architecture mémoire impose de ce fait plusieurs restrictions. Les données stockées en mémoire globale sont accessibles par l'ensemble des cœurs du GPU alors que la mémoire partagée est locale à un multiprocesseur. Cela signifie que seul les processeurs de ce même multiprocesseur peuvent accéder aux données en mémoire partagée. Cela limite donc la possibilité d'accès à cette mémoire. De la même façon, les registres sont locaux à un processeur scalaire et les données présentes dans ces registres ne peuvent être utilisées en dehors.

5.2.3 Modèle de programmation CUDA

L'utilisation d'une architecture particulière comme celle du GPU impose également un modèle de programmation particulier. Cette section traite du modèle de programmation CUDA permettant de réaliser des applications utilisant le processeur graphique. CUDA C est un outil de programmation lancé par Nvidia en 2007 permettant de réaliser du calcul générique à l'aide du processeur graphique. Les programmes CUDA faisant appel au GPU (que l'on appelle généralement *device*) sont communément appelés *kernels* et sont lancés depuis le processeur central (que l'on appelle généralement *host*). Les *kernels* sont ensuite exécutés sur le GPU par le biais d'une *grille d'exécution*. Cette grille est composé d'un ensemble de *blocs* à plusieurs dimensions possibles (1, 2 ou 3). Chaque bloc est également constitué d'un ensemble de *threads* à plusieurs dimensions possibles (1, 2 ou 3), comme le montre la figure 5.4.

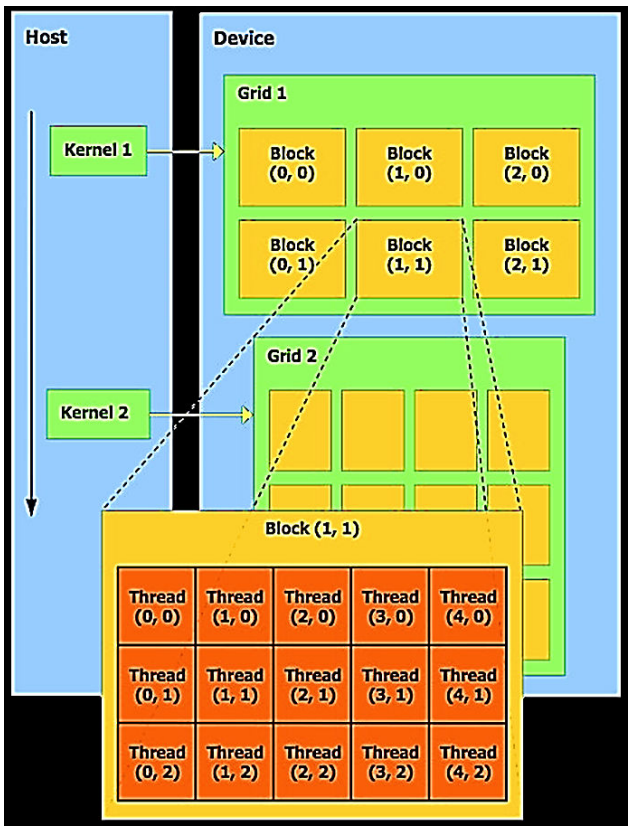


FIGURE 5.4 – Modèle de programmation CUDA (source Nvidia).

Ce modèle de programmation est en réalité très proche de l'architecture du GPU. Il en va de même pour la hiérarchie mémoire, un thread CUDA pouvant accéder à plusieurs

espaces mémoire pendant son exécution, comme le montre la figure 5.5. En effet, un thread peut accéder à des données privées locales à l'aide de registres. Il peut également accéder à des données partagées au sein d'un même bloc de threads. Pour cela, la mémoire partagée est utilisée. Pour partager des données entre des threads de blocs différents, la seule solution possible consiste à faire appel à la mémoire globale du GPU.

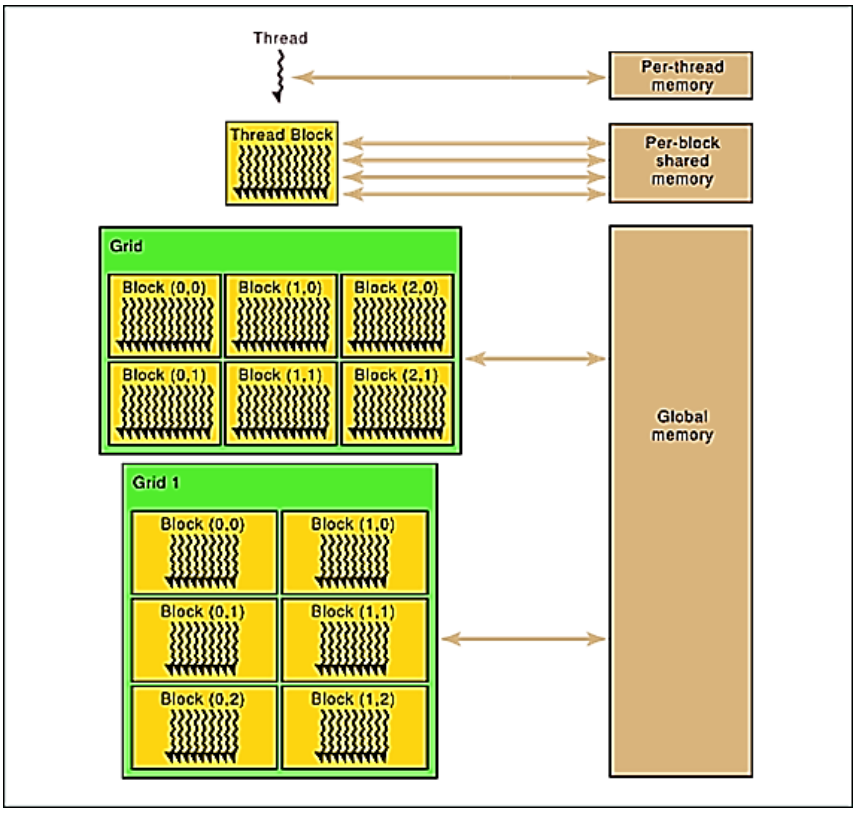


FIGURE 5.5 – Hiérarchie mémoire pour le modèle de programmation CUDA (source Nvidia).

5.3 Méthode de Boltzmann sur réseau et processeurs graphiques

Cette section réalise un inventaire détaillé des travaux liés à la méthode de Boltzmann sur GPU. Le but est de mettre en évidence un certain nombre de méthodes pouvant être utiles en vue d'une parallélisation du noyau de simulation sur une architecture de calculs composée de plusieurs processeurs graphiques. Cela implique dans un premier temps d'étudier des cas où un seul processeur graphique est utilisé. Les travaux faisant référence à l'utilisation de plusieurs processeurs graphiques sont finalement abordés.

5.3.1 Parallélisme mono-GPU

L'intérêt important concernant le GPU a rapidement été exploité de manière à réaliser des simulations 2D et 3D basées sur l'utilisation d'une méthode de Boltzmann sur réseau. Les principaux aspects concernant l'optimisation des performances utilisant le GPU sont décrits en plusieurs catégories [Mic12] [Woo13] :

1. Une utilisation adaptée du parallélisme de masse du GPU ;
2. Les accès aux différentes mémoires ;
3. L'utilisation des registres ;
4. Le chevauchement des transferts de données avec les calculs.

Deux facteurs sont nécessaires afin d'accroître le parallélisme d'une application. Le premier consiste à maximiser la quantité de travail qu'un thread peut faire de manière indépendante ; cela vaut à la fois pour les calculs mais également pour les accès mémoires. Le deuxième facteur consiste à maximiser le nombre de threads concurrents pouvant travailler de manière parallèle. Ce nombre détermine généralement l'*occupation* du GPU qui est défini comme le ratio entre le nombre de threads concurrents de l'application avec le nombre de threads concurrents maximum que le GPU puisse supporter. L'*occupation* est ainsi favorisée par plusieurs facteurs :

1. Le nombre de threads par bloc ;
2. Le nombre de registres utilisés par thread : les registres d'un multiprocesseur sont partitionnés en fonction du nombre de threads. Un nombre trop important de registres par thread peut ainsi conduire à une diminution du nombre de threads concurrents pouvant travailler en parallèle ;
3. L'utilisation de la mémoire partagée pour chaque bloc de thread : de la même façon que pour les registres, la mémoire partagée est partitionnée selon le nombre de blocs.

L'occupation du GPU n'est toutefois pas nécessairement un facteur de maximisation pour toutes les applications. Certaines applications peuvent avoir de meilleures performances pour une occupation plus faible du GPU [Vol10].

L'utilisation des mémoires du GPU est également un facteur d'optimisation important. L'optimisation des accès mémoire au sein du GPU semble même être un critère d'optimisation plus important que l'optimisation des calculs [OKT11]. Le processeur graphique dispose en effet de plusieurs mémoires à accès plus ou moins rapide (section 5.2.2). Étant la plus lente, l'utilisation de la mémoire globale est particulièrement importante. Réduire ses accès semble alors indispensable pour optimiser les performances d'une application. Cela passe par des accès « coalescents » de la mémoire globale. En réalité, les opérations mémoires peuvent se faire de manière conjointe pour un groupe de seize threads consécutifs (qu'on appelle généralement *demi-warp*). Réaliser ces échanges sur des portions de mémoire alignée permet ainsi de réduire le nombre d'accès à la mémoire globale et de maximiser l'efficacité de la bande passante par une utilisation optimale du bus mémoire.

L'utilisation de la mémoire partagée est également un facteur d'optimisation important. Pour des applications impliquant des déplacements de données importants, l'utilisation de la mémoire partagée à la place de la mémoire globale permet de réduire le temps d'accès aux données.

Concernant la méthode de Boltzmann sur réseau, Ryoo dans [RRB+08] a été le premier à tenter d'implémenter un code de simulation D3Q19. Des indications sont apportées concernant la structuration des données à adopter. Une structure de données de type *Array of Structures* (**AoS**) peut être utilisée sur le processeur central de manière à avoir les informations locales à une cellule sur des zones mémoire très proches. En revanche, cette structuration de données semble inadaptée sur le GPU. C'est pourquoi une structure de données du type *Structure of Arrays* (**SoA)** est généralement préférée dans ce cadre. En effet, les données proches en mémoire peuvent être traitées par des threads consécutifs ce qui favorise le parallélisme de masse et la coalescence des accès mémoire. Des problèmes d'alignement sont toutefois soulevés par les échanges de données dus à la propagation des données de la méthode de Boltzmann sur réseau.

Tölke dans [Töl08] a été le premier à mettre en avant, à l'aide de CUDA, une implémentation de la méthode de Boltzmann sur réseau pour un schéma D2Q9, ce qui a permis d'apporter une solution au problème d'alignement des données provoqué par le terme de propagation. L'idée était d'utiliser des zones de mémoire partagée pour réaliser

la propagation des données. Cela permettait de conserver la coalescence des données pour l'ensemble du modèle et ainsi d'améliorer de manière conséquente les performances. Toutefois, la quantité de mémoire partagée disponible dans les processeurs graphiques était très faible et cette approche ne pouvait ainsi pas se généraliser pour un modèle D3Q19 par manque de mémoire partagée. C'est pourquoi il utilisa dans [TK08] un modèle D3Q13 avec moins de fonctions de distribution de particules pour la généralisation au cas 3D. Il mit également en évidence que la fusion de la grille d'exécution CUDA avec la grille de calculs permettait également de favoriser la coalescence des données. Son implémentation du schéma D3Q13 avait conduit à une performance moyenne de 592 MLUPS sur une carte Geforce 8800 Ultra. Les performances sont intéressantes mais le modèle D3Q13 ne propose pas un nombre de propagation suffisamment grand pour obtenir une qualité scientifique suffisante. C'est pourquoi le modèle D3Q19 lui est généralement préféré.

De nombreuses approches sont ensuite apparues en littérature [KORR10] et se généralisent à des modèles plus complexes [LZWG13] [JH14]. Obrecht dans [OKT11] proposa une approche différente de la parallélisation de la méthode de Boltzmann sur GPU. Il mit en évidence une approche simple, basée sur l'idée que la lecture de données en mémoire globale est moins coûteuse que l'écriture de données. C'est pourquoi son approche se base sur un schéma de propagation inversé, où les fonctions de distribution sont d'abord lues en mémoire globale sans le respect de la coalescence des données. L'écriture des données en revanche se fait de manière coalescente en mémoire globale. Cette approche s'avère plus simple qu'une approche utilisant la mémoire partagée et offre des performances intéressantes. Rinaldi dans [RDVC12] reprit par la suite cette approche et utilisa de la mémoire partagée pour la lecture des données. Il mit également en avant des gains très importants par rapport à une implémentation CPU totalement sérialisée.

La plupart des travaux répertoriés ici se basent sur l'utilisation de deux matrices de données pour gérer l'étape de propagation. Peu de références s'intéressent à l'utilisation d'une technique de réduction de mémoire sur le processeur graphique. Les travaux de Bailey [BMW+09] permettent de mettre en évidence la méthode A-A pattern (section 3.5) et son intégration sur le GPU. Cependant, ces travaux manquent de clarté et d'explications concernant l'intégration de cette technique sur le GPU, même s'il semble que de bonnes performances soient obtenues. En effet, il obtenait une performance moyenne de 260

MLUPS sur une Geforce 8800 GTX pour un modèle D3Q19. Cela représente un gain en performances d'un facteur d'environ 28 par rapport à une parallélisation de son code de calculs utilisant 4 cœurs et OpenMP.

Habich dans [HFK+13] a récemment mis en évidence un facteur de performances limitant sur les cartes graphiques spécifiques aux calculs. Ces cartes sont généralement équipées d'une protection ECC (pour « Error Correction Code ») permettant d'obtenir une fiabilité totale des données de la mémoire interne du GPU. Toutefois, l'activation de cette protection a pour impact une réduction de la bande passante de la mémoire globale du GPU. Habich a ainsi montré dans ses travaux que la désactivation de cette protection sur les cartes de calculs Tesla C2050/C2070 permettait d'augmenter de manière conséquente la rapidité d'une simulation, un facteur d'accélération de 2 pouvant alors être obtenu. Toutefois, l'activation de la protection ECC semble inévitable pour obtenir des résultats de simulation corrects et ne doit pas être négligé pour obtenir de bonnes performances.

Différents travaux existants ont été évoqués dans cette section et ils mettent tous en évidence des performances extrêmement intéressantes de la méthode de Boltzmann sur réseau utilisant le GPU. La comparaison aux performances du processeur central est généralement abordée et montre que l'utilisation du GPU peut apporter un gain significatif en ce qui concerne les performances. Les gains obtenus sont évidemment différents selon le matériel utilisé mais un gros écart de performances apparaît généralement entre les deux architectures. C'est pourquoi l'utilisation du GPU pour notre noyau de simulation semble incontournable pour obtenir de bonnes performances. Ces références se limitent généralement à une modélisation de Boltzmann classique. Des références récentes traitent de méthodes de Boltzmann multi-phases utilisant le GPU [JH14] [LZWG13] mais les aspects physiques sont généralement mis en valeur.

5.3.2 Parallélisme multi-GPUs à un seul nœud de calculs

L'évolution des architectures de calculs utilisant le processeur graphique a permis de mettre en place des méthodes récentes de calculs pour la méthode de Boltzmann sur réseau en utilisant plusieurs GPUs. Les idées mises en avant consistent à diviser le domaine de simulation selon le nombre de GPUs disponibles [OKTR13b] [OKTR13a]. Dans ces références, le domaine de simulation est découpé suivant la plus grande dimension du

domaine de simulation et de manière parallèle à la direction d'alignement des données. En effet, ce type de découpage favorise les transferts de données efficaces entre les différents GPUs. Un thread CPU est ensuite associé à chaque GPU afin de gérer les différents « contextes » de calculs. Les processeurs graphiques disposent effectivement de mémoires distinctes et cela implique de communiquer les données entre les différents sous-domaines. La communication des données se fait généralement à l'aide de transactions basée sur la technologie « zéro-copy ». Cette technologie permet de réaliser des communications efficaces des données par un mapping entre les pointeurs CPU et GPUs. Cela permet de réaliser les transferts de données de manière implicite. Toutefois, cette technologie impose que les données ne doivent être lues et écrites qu'une seule fois afin d'obtenir de bonnes performances. Ces références indiquent également que les communications peuvent être camouflées avec du calcul à l'aide de transferts asynchrones des données. Cependant, les détails concernant ce camouflage ne sont généralement pas clairement décrits. Les performances obtenues sont généralement très intéressantes car ils permettent d'obtenir un gain quasiment linéaire pour un faible nombre de GPUs. Pour 6 GPUs travaillant en parallèle, l'auteur obtient une légère baisse de performances pour une efficacité d'environ 90 % de l'ensemble de la puissance de calculs disponible. Cette légère baisse de performances est principalement due au temps de communication des données qui devient trop important. Les calculs ne sont ainsi plus capables de camoufler l'ensemble des transferts de données d'où la perte de performances.

Très récemment, un environnement appelé Sailfish [JK14] permettant de réaliser des simulations LBMs sur plusieurs GPUs a été publié. Cet environnement inclut différents modèles tels que le BGK-LBM, MRT-LBM ou encore un modèle multi-phases type Shan & Chen. L'idée était de mettre en place un code générique de calculs pouvant travailler sur différents types d'architectures et de modélisations. Les résultats obtenus pour les simulations de cette référence sont également très intéressants. Toutefois, la désactivation de la protection ECC dans la référence implique nécessairement de bonnes performances pour les simulations au détriment de l'exactitude des calculs.

5.3.3 Parallélisme multi-GPUs à plusieurs nœuds de calculs

La généralisation à de très grosses architectures contenant plusieurs nœuds de calculs à plusieurs GPUs ont également été rapidement mis en avant. La combinaison des transferts de données à l'aide de CUDA et MPI ont permis d'obtenir des simulations efficaces [OKTR13c] [Ros11] [FHK+15]. L'environnement Sailfish [JK14] offre également la possibilité de réaliser des simulations utilisant plusieurs nœuds de calculs. Les communications entre les nœuds sont cette fois réalisées à l'aide de la librairie ZeroMQ qui semble plus simple d'utilisation que la librairie MPI.

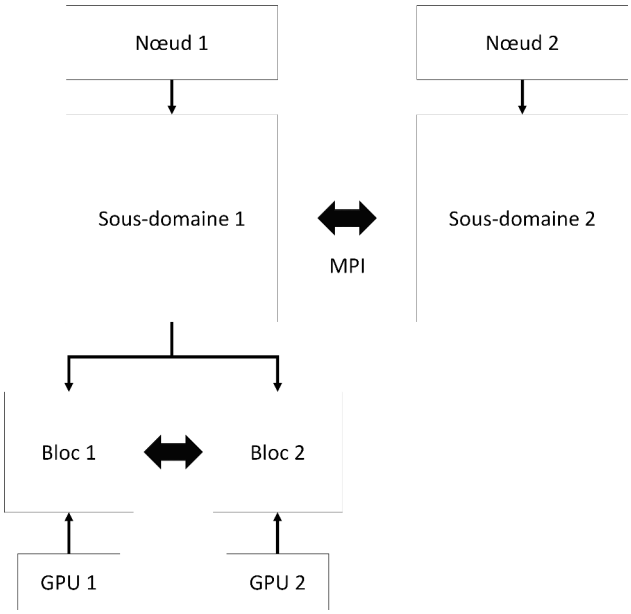


FIGURE 5.6 – Exemple de découpage d'un domaine de simulation pour une architecture à plusieurs nœuds de calculs composés de plusieurs GPUs : le domaine est dans un premier temps divisé en sous-domaines selon le nombre de nœuds de calculs. Chaque sous-domaine est de nouveau divisé selon le nombre de GPUs par nœud.

Le domaine est généralement découpé de la même façon que dans les approches à plusieurs nœuds de calculs pour les processeurs centraux. Le domaine est dans un premier temps divisé selon le nombre de nœuds de calculs disponibles. Ces sous-domaines sont finalement divisés de nouveau pour répartir la charge de calculs sur les GPUs présents au sein de chaque nœud (Figure 5.6).

Les communications entre les différents nœuds de calculs se font généralement par le biais des processeurs centraux à l'aide de messages MPI. Ces messages permettent de

faire transiter des données entre les processeurs centraux situés sur les différents nœuds. Cela implique par conséquent de communiquer les données présentes sur les GPUs vers le CPU au préalable (Figure 5.7).

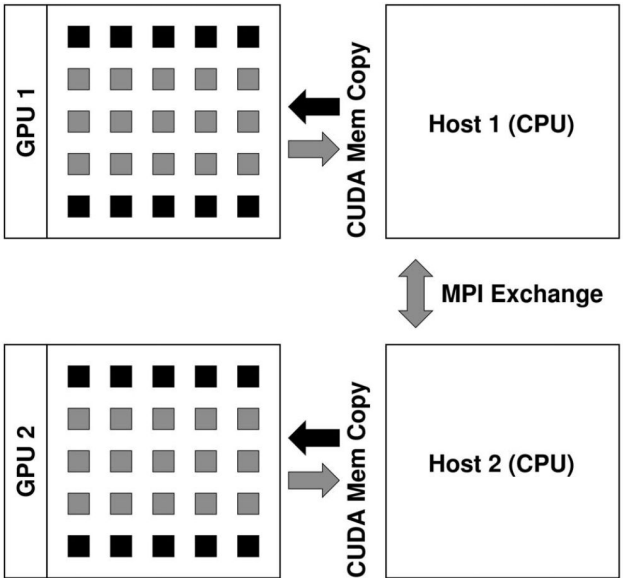


FIGURE 5.7 – Communications entre deux nœuds de calculs

Chapitre 6

Parallélisation du noyau de simulation sur un nœud de calculs composé de plusieurs processeurs graphiques

Sommaire

Ce chapitre traite de la mise en place d'une implémentation du noyau de simulation sur une architecture de calculs composée de plusieurs processeurs graphiques. L'utilisation de processeurs graphiques en littérature a montré qu'ils pouvaient être très efficaces pour réaliser des simulations basées sur la méthode de Boltzmann sur réseau. L'intégration du noyau de simulation au sein d'une architecture composée de plusieurs GPUs devrait ainsi offrir des performances bien au-delà des performances obtenues sur le processeur central.

Cela passe tout d'abord par une optimisation des performances sur un GPU. Pour cela, nous décrivons dans un premier temps la répartition des calculs et la gestion des accès mémoire internes à un GPU. Le passage sur plusieurs GPUs travaillant en parallèle sur un seul nœud de calculs est ensuite abordé. De la même façon, la répartition des calculs est décrite. Le chevauchement des transferts de données avec les calculs est ensuite présenté. L'inclusion de transferts en Peer-to-Peer en guise d'amélioration des transferts de données est finalement étudiée.

6.1 Optimisation des performances de calculs du GPU

Cette section reprend en détail quelques éléments permettant d'améliorer les performances du GPU en vue d'une implémentation du noyau de simulation. L'idée est dans un premier temps de répartir efficacement la charge de calculs sur les différents cœurs du processeur graphique. Les aspects d'optimisation des accès mémoire sont finalement traités dans une seconde partie.

6.1.1 Répartition des calculs

La méthode de Boltzmann sur réseau fait généralement appel à une grille de calculs cartésienne pour réaliser les différents calculs. L'utilisation d'une grille cartésienne s'adapte en réalité particulièrement bien au modèle de programmation du GPU.

L'idée naturelle [Töl08] [TK08] est donc de définir une grille d'exécution CUDA venant se coller à la grille cartésienne constituant le domaine de simulation, comme le montre la figure 6.1. Dans un cas 2D, la grille d'exécution est ainsi composée d'un groupe de blocs à une dimension, chaque bloc étant lui même composé d'un groupe de thread à une dimension. Dans un cas 3D, la grille d'exécution est composée d'un ensemble de

blocs à deux dimensions, chaque bloc étant toujours composé d'un ensemble de threads à une dimension. La majorité des calculs du noyau de simulation étant locale pour chaque cellule de la grille cartésienne, cette configuration permet de favoriser le parallélisme de masse du GPU en donnant à la fois une grosse quantité de calculs aux threads tout en permettant à de nombreux threads de pouvoir travailler de manière concurrente.

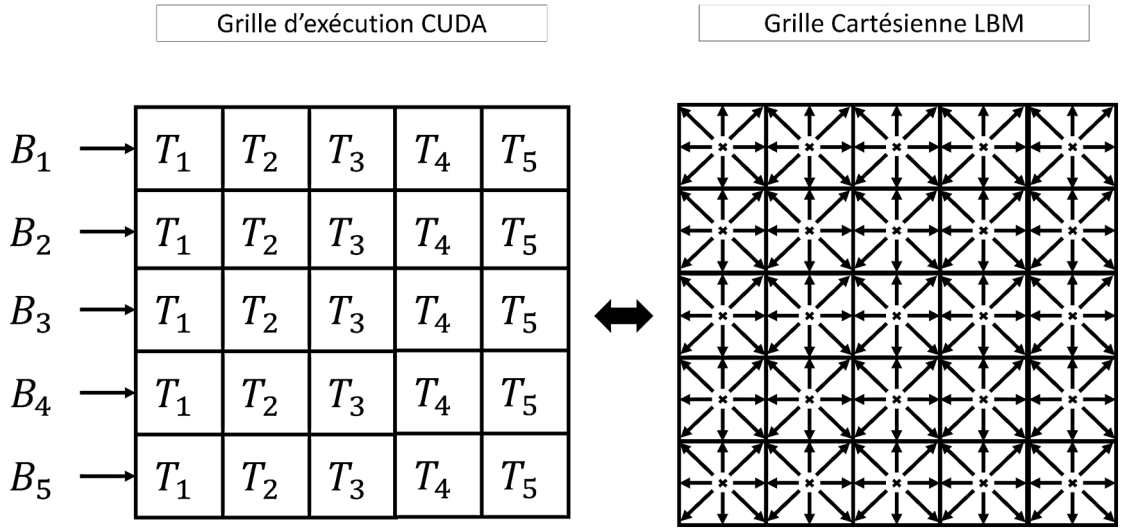


FIGURE 6.1 – Correspondance entre la grille d'exécution CUDA et la grille cartésienne LBM dans un cas 2D : les T_i représentent les threads CUDA et les B_i représentent les blocs CUDA.

Une attention particulière doit être portée au choix de la dimension correspondant aux groupes de threads. En effet, le choix du nombre de threads par bloc est important pour obtenir de bonnes performances, et un nombre trop petit ou trop grand peut être pénalisant [HFK+13] [RDVC12].

6.1.2 Optimisations mémoire

L'optimisation des accès mémoire est certainement le facteur le plus important pour réaliser des simulations performantes utilisant le processeur graphique. En effet, nous avons pu voir dans les sections précédentes que les accès à la mémoire globale sont généralement très lents et qu'il est important de réduire ces accès par des accès coalescents. Nous détaillons ici la méthode mise en place permettant de gérer de manière optimale les accès mémoire pour le noyau de simulation.

La coalescence des accès en mémoire globale impose en réalité deux conditions majeures :

1. L'alignement des données en mémoire ;
2. Le k^{eme} thread dans un demi-warp (groupe de seize threads) doit accéder au k^{eme} élément du bloc de données lu ou écrit en mémoire.

L'utilisation d'une configuration de la grille d'exécution comme celle présentée dans la section précédente (Figure 6.1) est favorable à la coalescence des données. En effet, l'utilisation d'une telle configuration permet à des threads voisins d'accéder à des données voisines en mémoire, permettant ainsi de respecter les conditions de la coalescence. Le choix de la direction d'alignement des données doit cependant bien correspondre à la dimension contenant les threads dans la grille d'exécution CUDA.

Certaines étapes de calculs du noyau de simulation ne sont toutefois pas entièrement locales pour chaque cellule de la grille de simulation et peuvent dépendre des données aux plus proches voisins. C'est le cas dans un premier temps du calcul des forces d'interaction au sein du même composant (Équation 2.25) ou entre différents composants physiques (Équation 2.26). Ces deux étapes font en réalité appel aux plus proches voisins de la cellule cible pour réaliser le calcul de ces forces, comme le montre la figure 6.2.

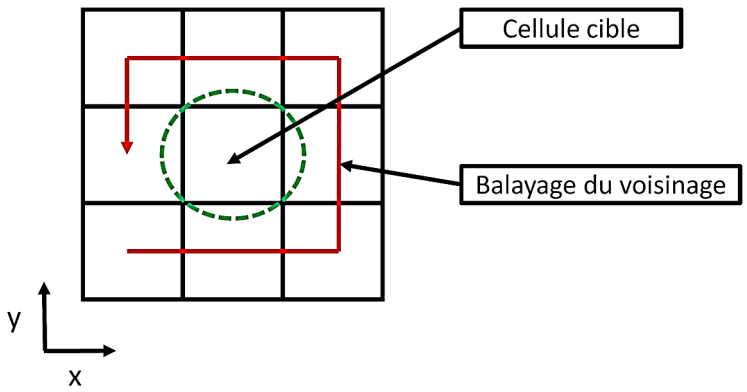


FIGURE 6.2 – Illustration du calcul de forces d'interaction à l'aide des plus proches voisins : la force sur la cellule cible (en vert) est calculée en balayant les valeurs du pseudo-potentiel aux plus proches voisins.

Cet appel aux plus proches voisins provoque ainsi un défaut d'alignement des données et de ce fait, une perte de la coalescence des accès à la mémoire globale. C'est pourquoi nous avons choisi de faire appel à la mémoire partagée du GPU pour régler ces problèmes d'alignement. L'idée consiste à faire appel à plusieurs buffers de mémoire partagée qui vont contenir l'ensemble des informations nécessaires aux calculs de ces forces pour un

bloc de threads. Il faut pour cela en premier lieu charger en mémoire partagée de manière coalescente les données du pseudo-potentiel ψ (Équation 2.24) aux plus proches voisins pour un bloc de threads. Une synchronisation entre les différents threads du bloc est alors nécessaire pour s'assurer que l'ensemble des données soit bien chargé en mémoire partagée. Une fois la synchronisation terminée, les calculs peuvent se faire naturellement à l'aide des données en mémoire partagée.

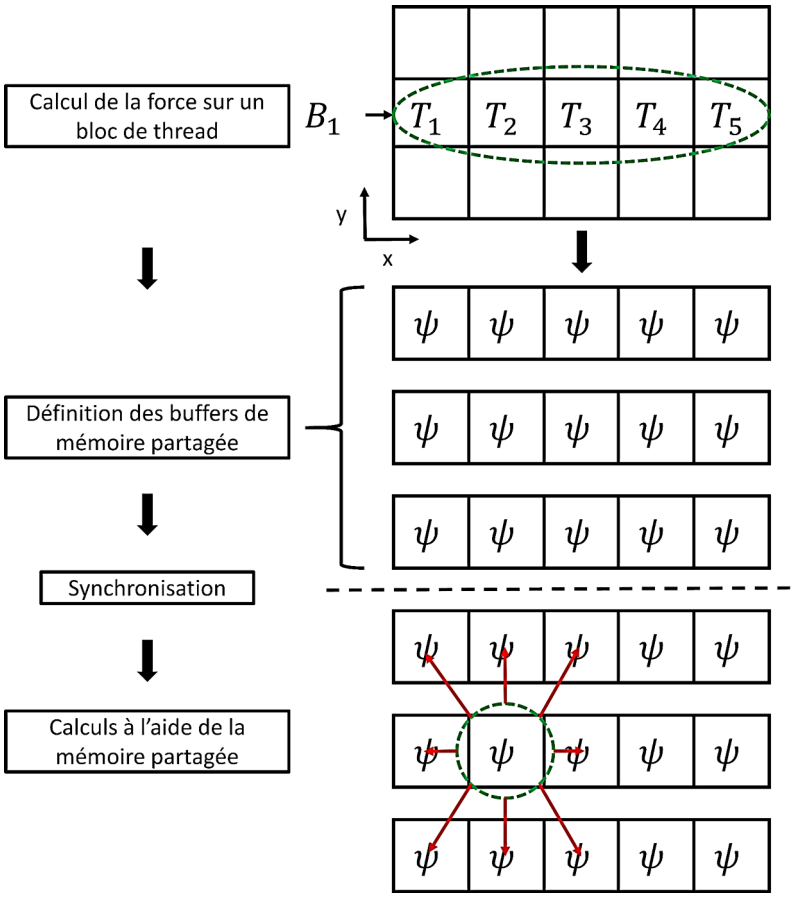


FIGURE 6.3 – Synthèse des étapes permettant le calcul des forces d'interaction : des buffers de mémoire partagée sont créés pour les données du pseudo-potentiel ψ ; les threads sont ensuite synchronisés ; les calculs se font finalement à l'aide des buffers de mémoire partagée.

La figure 6.3 résume de manière schématique l'utilisation de la mémoire partagée pour le calcul des forces d'interaction dans un cas 2D. Cette méthode est également valable dans un cas à trois dimensions et demande plus de mémoire partagée pour gérer l'ensemble des plus proches voisins. L'utilisation de la mémoire partagée permet ainsi de réduire le nombre d'appels à la mémoire globale et de conserver la coalescence des accès aux données.

L'étape de propagation des fonctions de distribution du schéma de Boltzmann impose également des problèmes d'alignement. En effet, cette propagation des données impose généralement une lecture et/ou une écriture des données de manière non alignée ce qui a pour effet de casser la coalescence des accès aux données. De la même manière, de la mémoire partagée est utilisée pour traiter ces problèmes d'alignement [Töl08] [TK08]. En revanche, dans notre cas particulier, l'intégration de la méthode de réduction de mémoire A-A pattern [BMW+09] change légèrement la façon de procéder. La succession de deux pas de temps distincts provoque cette légère modification. Pour la première itération (*A-step*), il n'est pas nécessaire de faire appel à la mémoire partagée car l'ensemble des calculs se fait de manière complètement locale pour chaque cellule. En effet, cette première itération ne faisant pas directement intervenir de propagation, les accès mémoire sont naturellement réalisés de façon coalescente ce qui n'impose pas l'utilisation de la mémoire partagée.

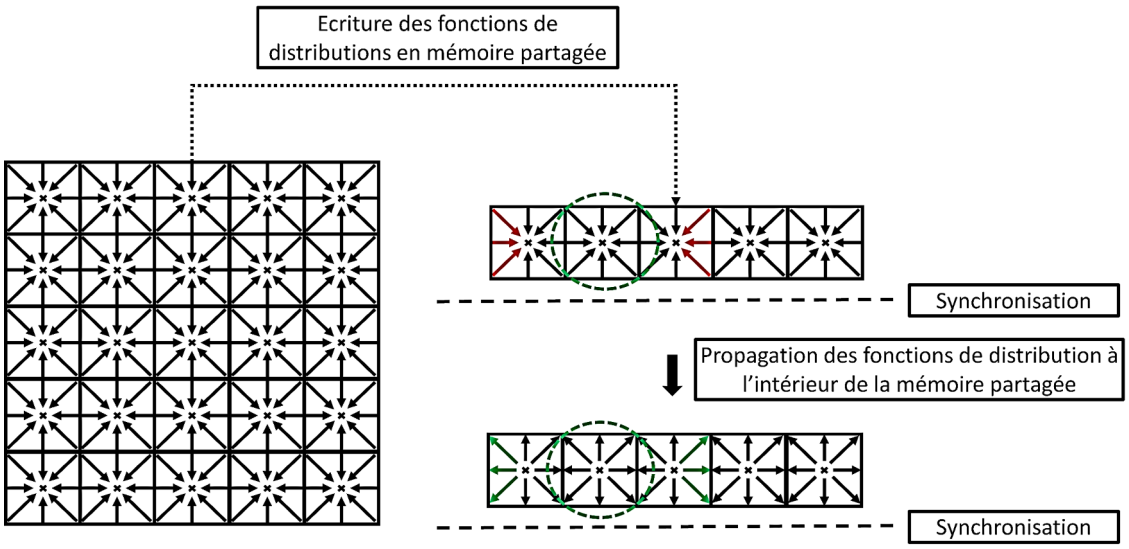


FIGURE 6.4 – Propagation des fonctions de distribution en mémoire partagée : les fonctions de distribution concernées par les problèmes d'alignement sont d'abord lues de manière coalescente et stockées en mémoire partagée ; la double propagation de ces fonctions de distribution se fait ensuite au sein de la mémoire partagée.

En revanche, la seconde itération provoque des problèmes d'alignement au niveau des fonctions de distribution à cause de leur double propagation. Cette double propagation est donc réalisée au sein de la mémoire partagée afin de conserver l'alignement des données, comme le montre la figure 6.4. De la même façon que précédemment, une étape de synchronisation est nécessaire pour charger les données en mémoire partagée. Une seconde

synchronisation des threads est finalement nécessaire à l'issue de la double propagation pour s'assurer qu'ils aient bien réalisé cette étape.

6.1.3 Conclusion

L'utilisation du processeur graphique impose des règles spécifiques permettant d'exploiter au mieux ses capacités. Ces règles permettent d'optimiser la puissance de calculs que propose cette architecture spécifique mais la principale préoccupation reste l'optimisation de ses accès mémoire. Les travaux issus de [Töl08] ont permis de mettre en évidence que l'impact des optimisations mémoire pouvait être d'un facteur d'environ 2 sur le plan des performances dans leurs conditions de simulation. En effet, le temps d'accès à la mémoire globale est généralement le facteur limitant pour les applications utilisant le processeur graphique. La réduction du nombre de transactions mémoire par des accès coalescents et une bonne utilisation de la mémoire partagée permettent ainsi d'optimiser les accès mémoire du GPU.

Les sections précédentes ont permis de mettre en évidence des méthodes permettant d'optimiser la charge de calculs et les accès à la mémoire globale pour notre noyau de simulation. En plus de ces méthodes, Nvidia a mis en place des outils du type Nvidia Visual Profiler [Pro11] ou Nvidia Nsight permettant de mesurer certaines métriques de l'application telles que le nombre de registres utilisés, la quantité de mémoire partagée utilisée, l'efficacité de la mémoire globale, etc. L'utilisation d'un tel outil peut être utile pour optimiser de manière plus fine les performances d'une application. Dans notre cas, cet outil a permis d'optimiser les accès mémoires et les performances de certaines simulations. Un usage intensif de ces outils peut améliorer de manière conséquente les performances d'une application.

6.2 Utilisation de plusieurs processeurs graphiques

L'utilisation de plusieurs processeurs graphiques en parallèle est une tâche complexe qui demande une bonne connaissance de CUDA. Cela soulève plusieurs problématiques. La première concerne la répartition de la charge de calculs entre les différents processeurs. Une répartition équitable de la charge de calculs est nécessaire de manière à ne défavoriser

aucun processeur et optimiser les performances. L'utilisation de GPUs impose également des problématiques liées à la mémoire. En effet, les différents processeurs graphiques disposent chacun de leur propre mémoire et cela impose de devoir communiquer des données à certaines étapes entre les différentes mémoires afin d'assurer le bon fonctionnement de la simulation. Ces communications représentent généralement un temps considérable et doivent être réduites et optimisées de manière à conserver de bonnes performances.

6.2.1 Répartition des calculs

Contrairement à l'usage d'un seul GPU, l'utilisation de plusieurs processeurs graphiques fonctionnant en parallèle impose une répartition des calculs qui doit être faite de manière à favoriser un certain équilibre de la charge de calculs. En effet, l'objectif est de fournir à chaque processeur graphique la même quantité de travail de manière à ne pas exploiter un processeur plus qu'un autre. Une répartition équitable de la charge de calculs permet ainsi d'éviter à certains processeurs d'attendre pendant que d'autres sont encore en train de calculer. L'idée pour la méthode de Boltzmann sur réseau consiste généralement à diviser équitablement le domaine de simulation en sous-domaines selon le nombre de processeurs disponibles. Un processeur est finalement associé à un sous-domaine et il est en charge des calculs au sein de ce sous-domaine (Figure 6.5).

La méthode de répartition est en réalité la même que dans un cas de simulation sur plusieurs nœuds de calculs composés de processeurs centraux [Thü07]. Cela implique une séparation du domaine sur des processeurs graphiques constitués de mémoires physiquement séparées. Cela signifie qu'une telle séparation implique que les zones mémoires correspondant à ces sous-domaines sont également séparées. Cette répartition entre GPUs impose par conséquent des communications entre ces différentes zones mémoires distinctes à certaines étapes du noyau de simulation. En effet, la méthode de Boltzmann sur réseau étant une méthode faisant généralement appel aux plus proches voisins, cette répartition impose de communiquer fréquemment des valeurs à l'interface des différents sous-domaines.

Une autre approche de répartition des calculs avait été envisagée dans ce cadre particulier de simulation à plusieurs composants physiques. La majorité des calculs étant indépendante entre les différents composants, il était envisageable de considérer une ré-

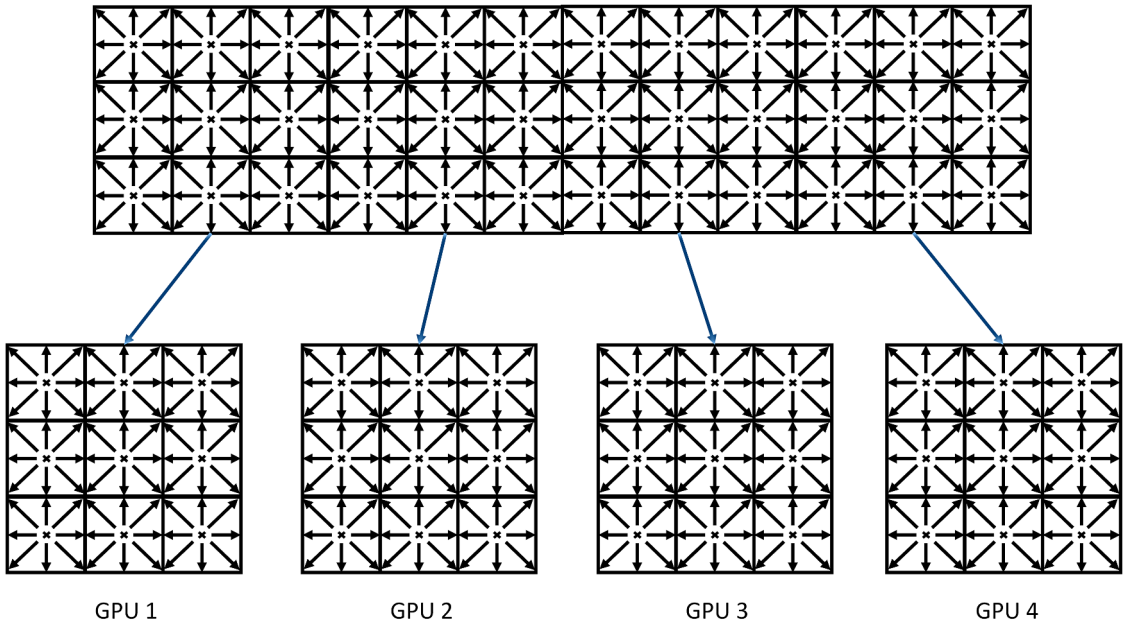


FIGURE 6.5 – Division d'un domaine de simulation 2D sur 4 GPUs : chaque GPU se voit attribuer un sous-domaine et est en charge de l'ensemble des calculs sur ce sous-domaine.

partition des calculs affectant un GPU à un composant physique (Figure 6.6).

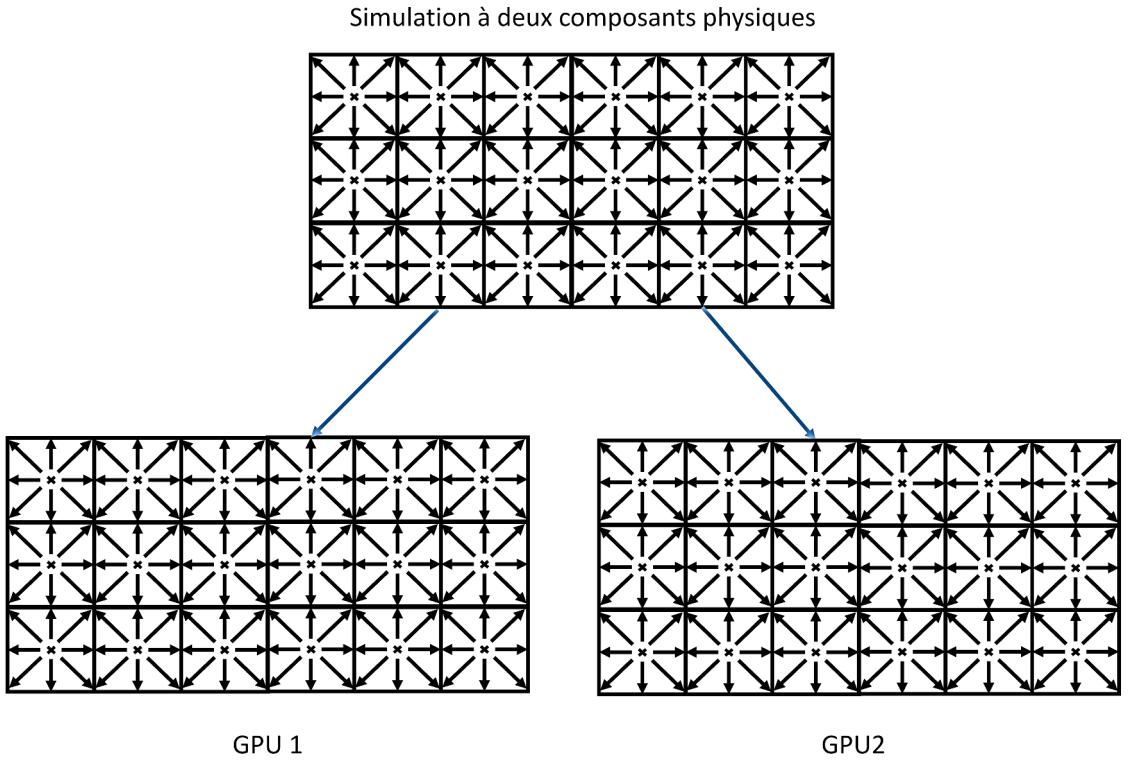


FIGURE 6.6 – Division d'un domaine de simulation 2D pour deux composants physiques sur 2 GPUs : chaque GPU se voit attribuer un composant et est en charge de l'ensemble des calculs pour ce composant.

Une simulation n'impose toutefois pas toujours un nombre de composants physiques aussi important que le nombre de GPUs disponibles. Pour ces cas-là, une combinaison de la répartition spatiale des calculs (Figure 6.5) avec une répartition par composant (Figure 6.6) avait été envisagée. Cette n'approche ne pouvait finalement pas être viable car la séparation des calculs par composant demandait des communications de données bien plus importantes que pour une séparation purement spatiale. Une estimation a mené à une augmentation d'environ 95% de la quantité de données à communiquer. Même si l'indépendance des calculs est plus importante pour une séparation par composant, cette trop grosse quantité de données à communiquer est trop pénalisante pour être viable. C'est pourquoi une répartition spatiale des calculs est envisagée pour le noyau de simulation.

De la même manière qu'en littérature [OKTR13b] [OKTR13a], des threads CPU sont finalement créés pour gérer les différents contextes de calculs. L'utilisation de threads est indispensable pour permettre le parallélisme entre les différents GPUs. En effet, le code de simulation est exécuté sur les processeurs graphiques mais ces exécutions sont lancées à l'aide du processeur central. L'utilisation de plusieurs threads permet ainsi de paralléliser le lancement des exécutions du code de simulation sur les différents GPUs.

6.2.2 Chevauchement entre les calculs et les transferts de données

L'efficacité des communications entre les différents GPUs est certainement la tâche la plus difficile à atteindre quand on réalise des simulations sur plusieurs GPUs. Le domaine de simulation étant découpé en plusieurs sous-domaines selon le nombre de GPUs, cela impose de devoir communiquer des données à certaines étapes du noyau de simulation. Les étapes faisant appel aux plus proches voisins sont concernées par ces communications. La première concerne le calcul des forces d'interaction F_{int} et F_{ext} (Équations 2.25 et 2.26) faisant appel aux valeurs du pseudo-potentiel ψ aux plus proches voisins. La seconde concerne la propagation des fonctions de distribution pour les deux itérations de la méthode A-A pattern. L'idée est simplement de communiquer les valeurs nécessaires entre les GPUs afin de garantir des calculs corrects (Figure 6.7).

La réalisation de ces communications peut être pénalisante pour les performances d'une simulation, car le temps de transfert entre des zones mémoires non directement connectées est généralement long. De plus, ces données étant absolument indispensables

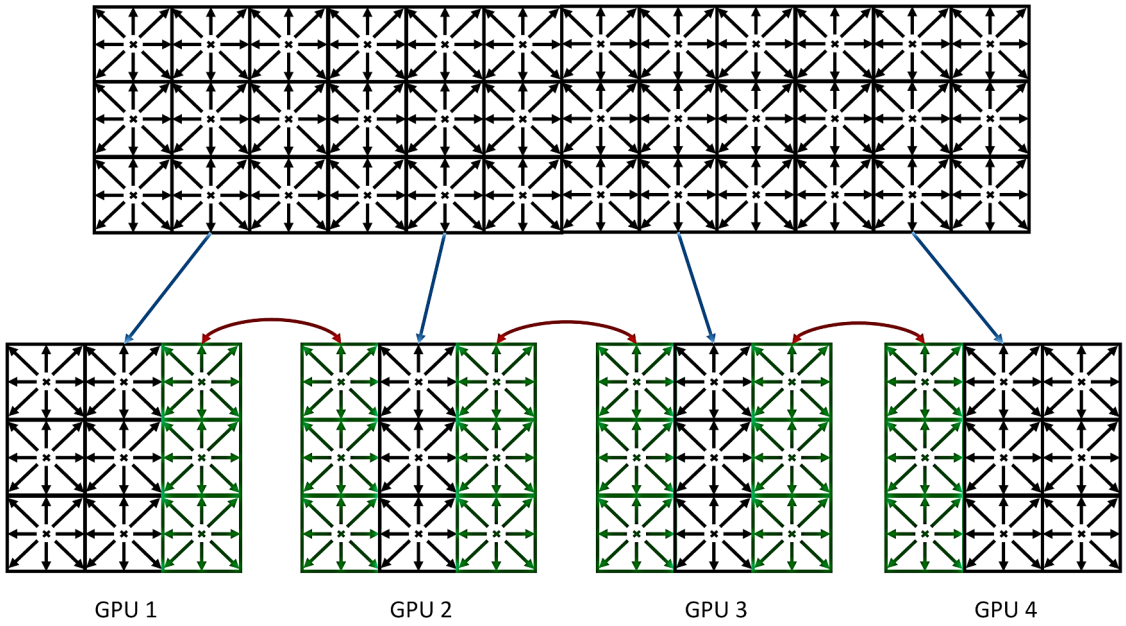


FIGURE 6.7 – Communication des valeurs à l'interface entre les sous-domaines pour un domaine de simulation 2D : certaines valeurs contenues dans les cellules colorées en vert doivent être échangées entre les GPUs concernés.

au bon fonctionnement de la simulation, il est indispensable d'avoir bien communiqué leurs valeurs avant de pouvoir calculer les étapes suivantes. Afin d'obtenir un temps de simulation le plus court possible, le chevauchement entre les copies mémoire et les calculs est ainsi indispensable. En effet, effectuer des copies mémoire de manière asynchrone permet un gain non négligeable en performance par la réduction du temps d'attente des données. Dans notre cas, l'idée est de séparer les phases calculatoires en deux étapes : les bords et l'intérieur du domaine. Les calculs sont d'abord réalisés sur les bords qui nécessitent par la suite une communication vers un sous-domaine voisin et le résultat est stocké à l'aide de buffers de mémoire alignée (Figure 6.8).

Ces buffers sont ensuite communiqués aux sous-domaines voisins de manière asynchrone pendant que les calculs sont appliqués sur l'intérieur du domaine (Figure 6.9). Ainsi, les différentes communications à réaliser s'effectuent en même temps que du calcul, ce qui permet de gagner en efficacité. En réalité, contrairement à la figure 6.9 qui sert ici d'illustration, seules les valeurs nécessaires (pseudo-potentiel ψ et fonctions de distribution f_i se propageant hors des sous-domaines) sont communiquées et non l'intégralité de la cellule.

L'idée paraît simple en théorie mais, à notre connaissance, aucune référence biblio-

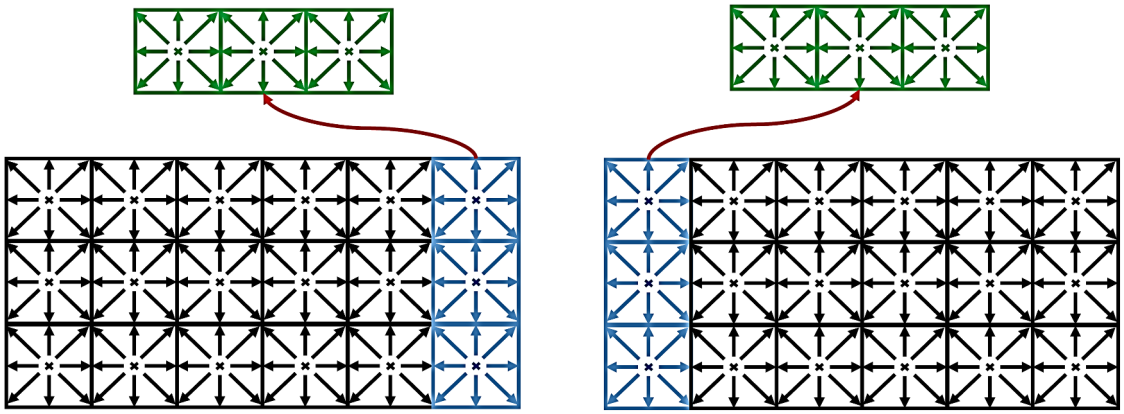


FIGURE 6.8 – Réalisation des calculs aux bords du domaine et stockage sur des buffers de mémoire alignée : les cellules en bleu correspondent aux cellules bords à calculer et les cellules en vert correspondent aux buffers résultant.

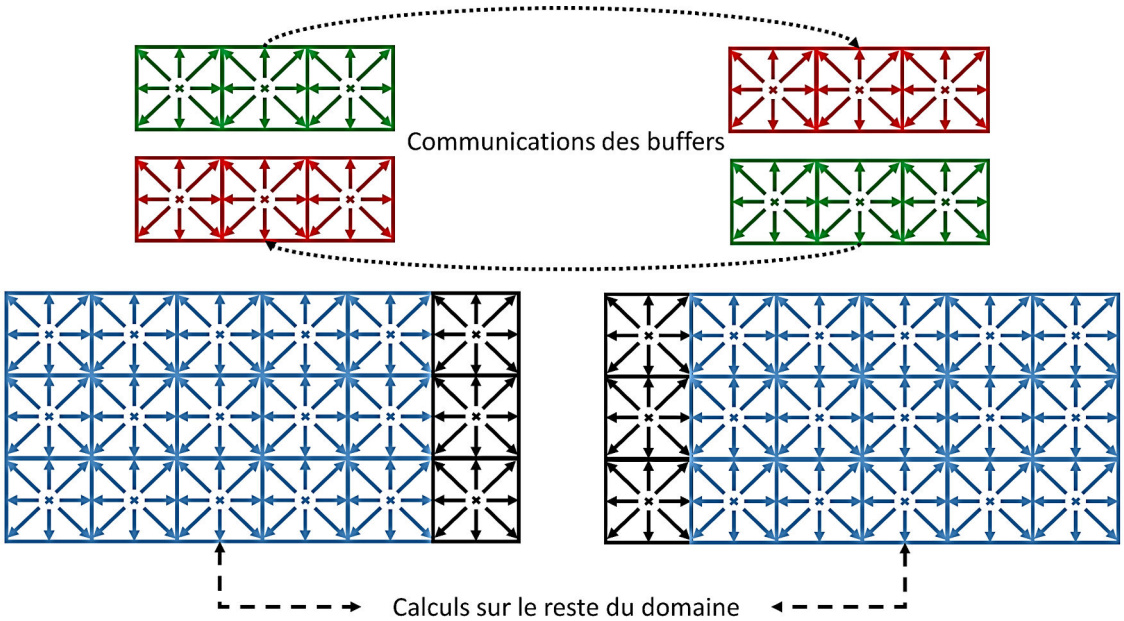


FIGURE 6.9 – Calculs sur le reste du domaine en même temps que les communications aux bords des sous-domaines : les calculs des cellules en bleu se font en même temps que la communications des buffers (buffers vert en direction du buffer rouge).

graphique ne parle de sa mise en pratique à l'aide de CUDA. En effet, aucune référence ne traite de la méthode employée pour chevaucher les communications avec les calculs. Dans notre cas, nous avons choisi de faire appel à des « *streams* ». Un stream en CUDA est une séquence d'opérations qui s'exécutent sur le processeur graphique dans l'ordre dans lequel elles sont émises par le code du CPU. Alors que les opérations au sein d'un même stream sont garanties pour être exécutées dans un ordre précis, les opérations de différents streams peuvent toutefois être entrelacées et, lorsque cela est possible, les streams peuvent

même fonctionner de manière simultanée. L'utilisation de ces streams permet dans un premier temps le chevauchement entre des calculs s'effectuant sur le processeur graphique avec du calcul s'effectuant sur le processeur central. Ils offrent également la possibilité de cumuler l'exécution de calculs avec des transferts de données. Pour cela, il faut affecter à un premier stream la communication des données et à un second stream l'exécution des calculs. Dans le cas de notre noyau de simulation, l'idée est de créer un premier stream en charge du calcul sur les bords des sous-domaines et des communications des bords, et un second stream en charge des calculs sur le reste du domaine, comme le montre la figure 6.10.

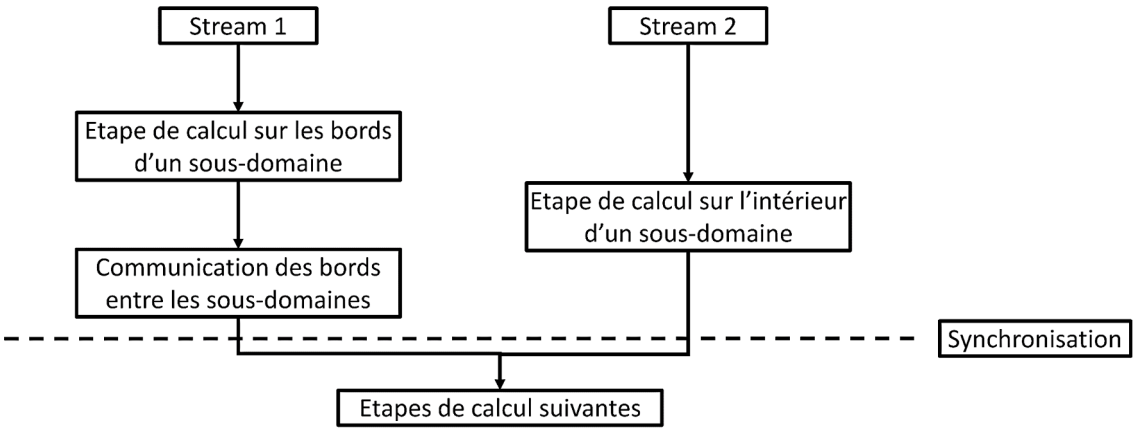


FIGURE 6.10 – Utilisation simplifiée de streams concurrents pour réaliser le chevauchement des données avec les calculs du noyau de simulation : le stream 1 est en charge des calculs aux bords et de la communication, tandis que le stream 2 se charge des calculs sur le reste du sous-domaine.

Des barrières de synchronisation entre les différents streams sont finalement nécessaires pour assurer la bonne continuité des calculs, car nous ne pouvons pas assurer quel stream est le plus rapide. En effet, il est indispensable d'attendre la fin des communications pour poursuivre les étapes de calculs suivantes.

6.2.3 Amélioration des transferts de données par l'inclusion de transferts en Peer-to-Peer

L'environnement CUDA offre la possibilité de réaliser des transferts de données à l'aide d'instructions spécifiques. En effet, l'utilisation de l'instruction `cudaMemcpy` permet de réaliser des transferts de données depuis le processeur central vers le GPU (à

l'aide de l'option `cudaMemcpyHostToDevice`) ou encore du GPU vers le processeur central (avec l'option `cudaMemcpyDeviceToHost`). L'utilisation de transferts de données asynchrones impose l'utilisation de l'instruction `cudaMemcpyAsync` en lui spécifiant un stream.

La première solution pour transférer des données entre deux processeurs graphiques consiste à séparer le processus de copies en deux étapes. La première étape consiste à réaliser un transfert du GPU de départ vers le processeur central et la seconde étape consiste à réaliser un transfert du processeur central vers le GPU cible (Figure 6.11). Le processeur central est en effet une interface nécessaire pour communiquer les données entre deux processeurs graphiques.

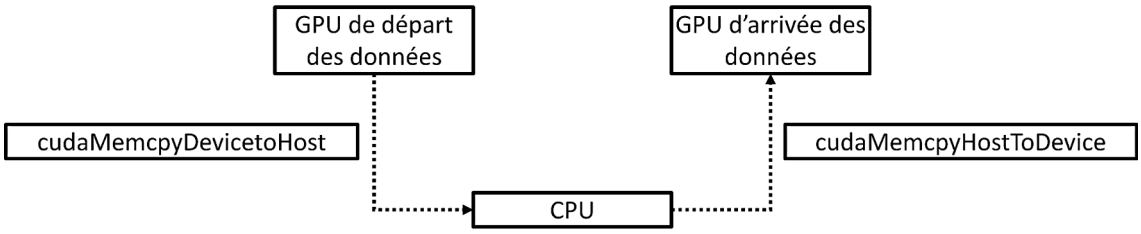


FIGURE 6.11 – Mécanisme de communication standard entre deux GPUs : les données sont envoyées du GPU de départ vers le CPU, puis repartent du CPU vers le GPU cible.

Ces transferts de données peuvent être accélérés par l'utilisation de mémoire non paginée (« *page-locked or pinned memory* »). L'utilisation de la mémoire non paginée (à l'aide de l'instruction `cudaHostAlloc` ou `cudaMallocHost`) permet d'avoir une bande passante plus élevée et par conséquent d'accélérer les transferts de données [OKTR13b]. Une utilisation abusive de mémoire non paginée peut toutefois ralentir le système car la quantité de mémoire physique allouée au système d'exploitation et à d'autres programmes est réduite.

CUDA offre une autre solution permettant de simplifier les transferts de données par l'utilisation de transferts zéro-copy. La technologie zéro-copy permet de réaliser des communications efficaces par un simple mapping des pointeurs mémoire du CPU et du GPU. L'utilisation d'un tel mapping entre les deux mémoires permet de réaliser des transferts de données de manière implicite. Pour que ces transferts soient efficaces, les pointeurs contenant les données à communiquer ne doivent être lus ou écrits qu'une seule fois. En ce qui concerne la méthode de Boltzmann sur réseau, c'est généralement ce type de transactions

qui est utilisé [OKTR13b] [JK14] [OKTR13a].

Nous proposons dans ce manuscrit une approche différente de transferts de données entre les processeurs graphiques, tirant partie des spécificités de notre architecture cible. Sur les architectures de calculs intensifs récentes, plusieurs processeurs graphiques peuvent être connectés au même port PCIe. Nvidia a ainsi mis en place une technologie nommée GPUDirect [Cud12] permettant de réaliser des accès mémoires et des transferts en Peer-to-Peer entre deux processeurs graphiques compatibles (Figure 6.12). L'utilisation de ce type de transfert permet alors d'éviter le passage par le processeur central et ainsi d'améliorer la rapidité des échanges. Toutefois, les communications en Peer-to-Peer sont restreintes uniquement pour des GPUs partageant le même chipset.

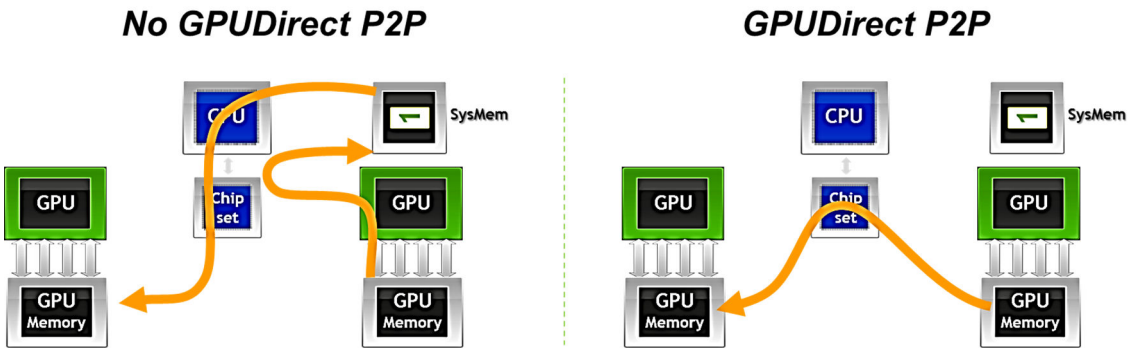


FIGURE 6.12 – Illustration de la technologie GPUDirect (source Nvidia).

L'objectif est donc de favoriser au maximum l'utilisation de transferts en Peer-to-Peer pour les GPUs compatibles et d'utiliser des transactions zéro-copy lorsque les GPUs ne sont pas compatibles. La compatibilité entre deux GPUs est un aspect important pour maximiser l'utilisation de transferts en Peer-to-Peer. En effet, la répartition spatiale des GPUs (section 6.2.1) doit être adaptée afin de maximiser ce type de transferts.

Pour cela, nous proposons d'utiliser un algorithme de clustering de type K-means [WCS+01] [M+67] permettant de maximiser l'utilisation de transferts en Peer-to-Peer dans un cadre général. L'idée est d'exploiter le fait que des sous-domaines très proches spatialement nécessitent généralement de communiquer des données. C'est exactement ce que propose l'algorithme de K-means. Il permet de regrouper dans des classes (dont le nombre est prédéfini) un ensemble d'observations. Le but est par conséquent de regrouper dans des classes (ou « *cluster* ») les différents sous-domaines.

La première étape consiste à définir le nombre de classes qui vont intervenir dans l'al-

gorithme de K-means. Cela peut être fait à l'aide de CUDA qui est en mesure de définir si des GPUs sont capables de communiquer en Peer-to-Peer (à l'aide de l'instruction `cudaDeviceCanAccessPeer`). Ainsi, l'ensemble des processeurs graphiques compatibles forme une classe (Figure 6.13).

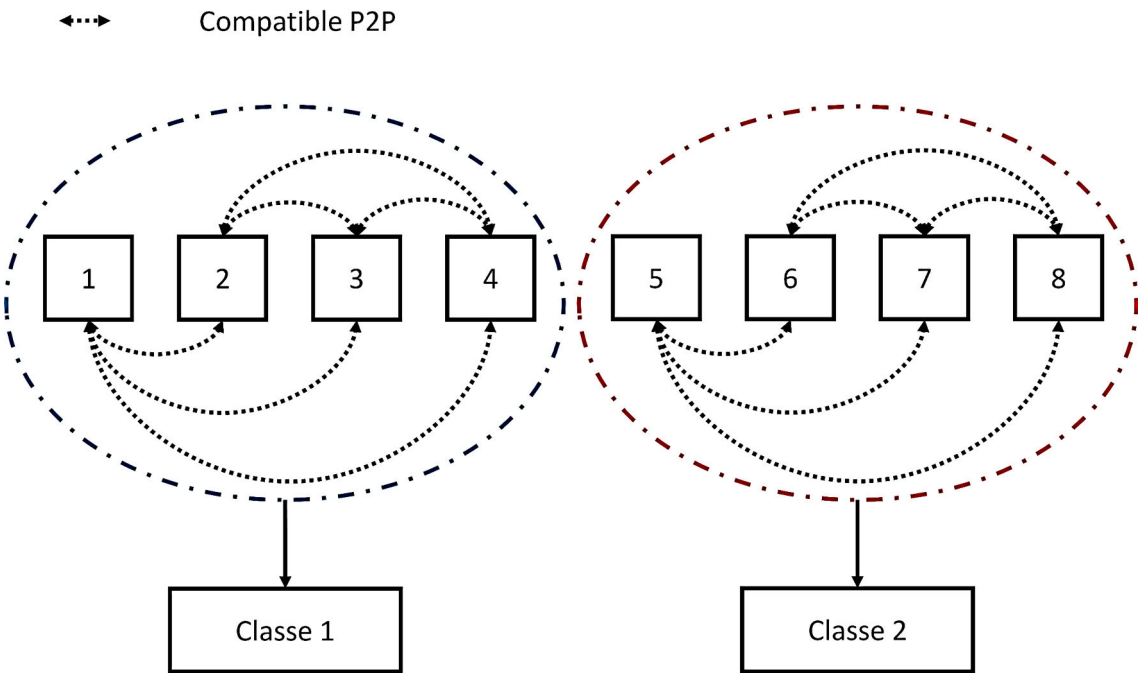


FIGURE 6.13 – Détermination du nombre de classes pour une architecture à 8 GPUs : les classe 1 et 2 sont composées de processeurs graphiques pouvant communiquer en Peer-to-Peer.

Une fois le nombre de classes défini, on peut alors appliquer l'algorithme de K-means sur les sous-domaines issus de la décomposition spatiale du domaine de simulation. Des centres de classes sont dans un premier temps définis pour chaque classe. Les classes sont ensuite séparées de manière itérative à l'aide de la distance euclidienne. Les GPUs sont finalement répartis au sein de chaque classe. L'algorithme 3 résume les différentes étapes permettant de répartir les GPUs sur les différents sous-domaines.

Cet algorithme se veut généraliste et adaptable à de nombreuses situations. En effet, il est valable dans le cas le plus simple où le domaine est découpé en un nombre de sous-domaines égal au nombre de GPUs mais également dans des cas où il y aurait un nombre de sous-domaines plus important que le nombre de GPUs. Une contrainte supplémentaire doit toutefois être prise en compte afin de favoriser une répartition de la charge de calculs équitable. L'idée est simplement de forcer l'affectation de chaque GPU de manière

Algorithme 3 : Algorithme de K-means et répartition des GPUs sur les différents sous-domaines.

```
↪ Définir le nombre de classes en fonction de la possibilité de communications en Peer-to-Peer ;
↪ Définir des centres de classe de manière aléatoire sur le domaine de simulation;
tant que divergence faire
    pour chaque sous-domaine faire
        ↪ Calcul de la distance euclidienne entre les centres de classes et le barycentre du
        sous-domaine ;
        ↪ Assigner le sous-domaine à la classe la plus proche ;
    fin
    pour chaque classe faire
        ↪ Mise à jour des centres de classes par une simple moyenne ;
    fin
    si Centre de classes stabilisé alors
        ↪ Convergence de l'algorithme ;
    fin
fin
pour chaque classe faire
    ↪ Répartition des GPUs aux différents sous-domaines ;
fin
```

consécutive de manière à favoriser un équilibrage de la charge de calculs.

L'utilisation d'un tel algorithme présente un réel avantage car il tient compte de la proximité spatiale des sous-domaines communicants afin de maximiser l'efficacité des communications en Peer-to-Peer. Cela permet une répartition automatique et efficace de la charge de calculs avant le lancement de la simulation pour un domaine de calcul donné.

6.3 Évaluation des performances

Cette partie a pour objectif d'évaluer les performances obtenues en réalisant plusieurs comparaisons. Il faut avoir conscience que réaliser une comparaison des performances par rapport à la littérature s'avère très compliqué. En effet, de nombreux facteurs sont susceptibles d'intervenir sur les performances. Le modèle utilisé est un premier facteur important car il peut être différent selon les références. Dans notre cas, le modèle utilisé est plus complexe et fait intervenir une quantité de calculs bien plus importante que ceux généralement utilisés en littérature. La littérature se limite la plupart du temps à l'utilisation d'une méthode de Boltzmann sur réseau standard [OKT11] [HFK+13] [RDVC12]

[OKTR13b] [BMW+09]. Le matériel utilisé est également un facteur majeur intervenant sur les performances. De plus, selon les références, le matériel utilisé ne bénéficie pas toujours d'une protection ECC [1], synonyme d'un gain important de performances au détriment de l'exactitude des calculs. Certaines références choisissent volontairement de désactiver cette protection afin d'accroître les performances de calculs [JK14] [HFK+13]. Dans notre cas, cette protection reste active afin d'être assuré de l'exactitude des calculs.

En réalité, pour réaliser une comparaison efficace, il faudrait se placer exactement dans les mêmes conditions que celles utilisées en littérature, ce qui est très difficile à réaliser. Nous proposons à la place de réaliser plusieurs comparaisons en tenant compte de nos implémentations précédentes. La première comparaison consiste à faire une comparaison des performances avec l'utilisation du processeur central. L'impact des communications des données entre les processeurs graphiques et l'influence de la taille du domaine sont également étudiés.

6.3.1 Algorithme

Le but de cette section est de reprendre l'ensemble des éléments mis en place permettant de réaliser des simulations en utilisant notre noyau de simulation sur une architecture composée de plusieurs processeurs graphiques travaillant de manière parallèle. Pour plus de clarté, l'algorithme global du noyau de simulation est décomposé sous la forme de plusieurs parties.

L'algorithme 4 décrit les calculs liés à la pression et au pseudo-potentiel permettant de calculer par la suite les différentes forces d'interaction. Les couleurs symbolisent les étapes de l'algorithme pouvant s'effectuer de manière concurrente. L'algorithme 5 quant à lui décrit la seconde partie de l'algorithme constituant au final un algorithme de simulation sur une architecture à plusieurs processeurs graphiques pouvant travailler en parallèle. De la même façon, les couleurs symbolisent les étapes pouvant être exécutées de manière concurrentes.

L'objectif est de mettre en place un algorithme efficace du noyau de simulation utilisant plusieurs processeurs graphiques pouvant travailler de manière parallèle. Pour cela, le chevauchement des communications avec le maximum de calculs possibles est réalisé. De

1. pour « Error Correction Code ».

Algorithme 4 : Première partie de l'algorithme : initialisation de la simulation et calcul de la pression p_σ et du pseudo-potentiel ψ_σ sur plusieurs processeurs graphiques ; la partie en rouge s'exécute de manière concurrente à la partie en bleu.

↪ Initialisation de la simulation ;
↪ Découpage du domaine de simulation en fonction du nombre de GPUs ;
↪ Utilisation de l'algorithme K-means (Algorithme 3) pour répartir de manière efficace les GPUs aux sous-domaines ;
↪ Création de threads CPU permettant de gérer les différents GPUs ;
pour *chaque itération* **faire**
  pour *chaque sous-domaine* **faire**
    pour *chaque composant* σ **faire**
      ↪ Calcul à l'aide d'un stream de copie sur le GPU associé de p_σ sur les bords concernés avec l'équation (2.23) ;
      ↪ Calcul à l'aide d'un stream de copie sur le GPU associé de ψ_σ sur les bords concernés avec l'équation (2.24) ;
      ↪ Communications à l'aide d'un stream de copie de ψ_σ aux interfaces entre les sous-domaines : Utilisation de transferts en Peer-to-Peer pour les GPUs compatibles et de transactions zéro-copy sinon ;
      ↪ Calcul à l'aide d'un stream de calculs sur le GPU associé de p_σ pour le reste du sous-domaine (Équation (2.23)) ;
      ↪ Calcul à l'aide d'un stream de calculs sur le GPU associé de p_σ pour le reste du sous-domaine (Équation (2.24)) ;
      ↪ Correction à l'aide d'un stream de calculs des erreurs aux conditions limites et aux obstacles ;
    fin
  fin
  ↪ Synchronisation des GPUs ;
  ↪ Suite des calculs : Algorithme 5 ;
fin

plus, la maximisation de transferts de données plus rapides tels que les communications en Peer-to-Peer est réalisée afin de réduire au mieux ce temps de transfert.

6.3.2 Simulations réalisées

Plusieurs simulations ont été réalisées de manière à étudier les performances obtenues par notre algorithme. La première simulation concerne une simulation à un seul composant de descente de bulle dans un domaine fermé de taille $160 \times 160 \times 1200$ cellules de calculs. Le but de cette simulation est d'initialiser une bulle d'un fluide liquide à l'intérieur du domaine. Par gravité, cette bulle liquide va progressivement descendre au fond du domaine (Figure 6.14).

La seconde simulation concerne une simulation d'épandage contenant deux compo-

Algorithme 5 : Seconde partie de l'algorithme : calcul des forces et collision-propagation des fonctions de distribution ; les étapes en bleu et en rouge peuvent s'effectuer de manière concurrente.

si *itération paire* **alors**
- **pour** *chaque sous-domaine* **faire**
  - **pour** *chaque composant* σ **faire**
    - ↪ Calcul des forces aux bords concernés à l'aide d'un stream de copie sur le GPU associé (Équations (2.25)-(2.28)) ;
    - ↪ Calcul de la première fonction d'équilibre $f^{eq}_{\sigma,i}$ aux bords concernés à l'aide d'un stream de copie sur le GPU associé (Équation (2.7)) ;
    - ↪ Somme des forces et correction du terme de vitesse Δu_σ sur les bords concernés à l'aide d'un stream de copie sur le GPU associé (Équation (2.31)) ;
    - ↪ Calcul de la seconde fonction d'équilibre $f^{eq}_{\sigma,i}$ sur les bords concernés à l'aide d'un stream de copie sur le GPU associé (Équation (2.7)) ;
    - ↪ Calcul des temps de relaxation avec un modèle de sous-maille de Smagorinsky aux bords concernés pour le *A-step* de la méthode A-A pattern à l'aide d'un stream de copie sur le GPU associé ;
    - ↪ Étape de collision-propagation des fonctions de distribution $f_{\sigma,i}$ aux bords concernés à l'aide d'un stream de copie sur le GPU associé pour le *A-step* de la méthode A-A pattern ;
    - ↪ Communiquer les valeurs $f_{\sigma,i}$ nécessaires aux interfaces entre les sous-domaines à l'aide d'un stream de copie : Utilisation de transferts en Peer-to-Peer pour les GPUs compatibles et de transactions zéro-copy sinon ;
    - ↪ Calcul des forces sur le GPU associé à l'aide des équations (2.25) à (2.28) ;
    - ↪ Calcul de la première fonction d'équilibre $f^{eq}_{\sigma,i}$ sur le GPU associé à l'aide de l'équation 2.7 ;
    - ↪ Somme des forces et correction du terme de vitesse Δu_σ sur le GPU associé à l'aide de l'équation (2.31) ;
    - ↪ Calcul de la seconde fonction d'équilibre $f^{eq}_{\sigma,i}$ sur le GPU associé à l'aide de l'équation 2.7 ;
    - ↪ Calcul des temps de relaxation avec un modèle de sous-maille de Smagorinsky sur le GPU associé pour le *A-step* de la méthode A-A pattern ;
    - ↪ Étape de collision-propagation des fonctions de distribution $f_{\sigma,i}$ sur le GPU associé pour le *A-step* de la méthode A-A pattern ;
  - **fin**
- **fin**
- ↪ Synchronisation des GPUs ;
- **pour** *chaque sous-domaine* **faire**
  - **pour** *chaque composant* σ **faire**
    - ↪ Corrections aux conditions limites et aux obstacles sur le GPU associé pour le *A-step* de la méthode A-A pattern;
    - ↪ Calcul des quantités macroscopiques ρ_σ et u_σ sur le GPU associé pour le *A-step* de la méthode A-A pattern à l'aide des équations (2.10) et (2.11) ;
  - **fin**
- **fin**

sinon
- ↪ Réalisation des mêmes étapes de calculs que pour les itérations paires mais dans le cas du *B-step* de la méthode A-A pattern ;

fin

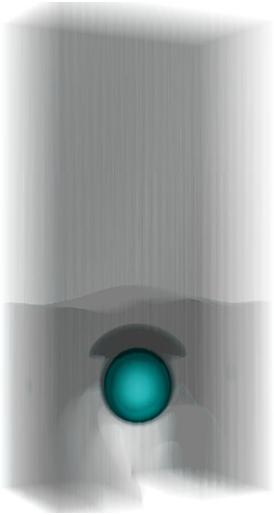


FIGURE 6.14 – Simulation 3D sur plusieurs processeurs graphiques d'une descente de bulle à l'aide d'un schéma D3Q19 : la bulle va progressivement descendre au fond du domaine de simulation par l'effet de la gravité.

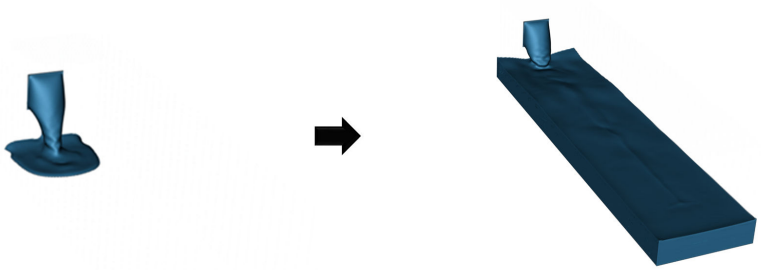


FIGURE 6.15 – Simulation sur plusieurs processeurs graphiques d'un épandage à deux composants à l'aide d'un schéma D3Q19 : le premier composant est présent sous sa forme liquide et gazeuse alors que le second est présent uniquement sous forme gazeuse. Ce cas de figure peut représenter une simulation contenant du GNL sous sa forme liquide et gazeuse en interaction avec l'air présent uniquement sous forme gazeuse.

sants sur un domaine de taille $128 \times 128 \times 1024$ cellules de calculs. Une fuite liquide est créée au bord du domaine de simulation et le fluide va progressivement se propager à l'intérieur du domaine et interagir avec le second composant présent dans le domaine de simulation (Figure 6.15).

6.3.3 Architecture de calculs

Nous avons utilisé huit cartes Nvidia Tesla C2050 (basées sur l'architecture matérielle Fermi) pour réaliser nos simulations. Le tableau 6.1 décrit quelques spécificités

techniques de la carte Tesla C2050.

Tesla C2050	
Version de la capacité de calcul CUDA :	2.0
Montant total de mémoire partagée :	2687 Mio
(14) Multiprocesseurs, (32) Cœurs CUDA :	448 cœurs CUDA
Fréquence d'horloge du GPU :	1147 MHz (1.15 GHz)
Taille du cache L2 :	786432 octets
Taille maximale de mémoire partagée par bloc :	49152 octets
Nombre total de registres par bloc :	32768

TABLEAU 6.1 – Spécifications techniques de la carte Tesla C2050

La possibilité de réaliser des communications en Peer-to-Peer est quant à elle décrite dans le tableau 6.2 (**X** implique que les deux GPUs concernés peuvent communiquer en Peer-to-Peer). Sur les 8 cartes présentes dans le serveur de calculs, les communications en Peer-to-Peer sont possibles entre les cartes 0 à 3 et entre les cartes 4 à 7. Ceci est déterminé automatiquement par le simulateur et reflète bien le fait que le serveur pilote les GPUs à l'aide de 2 chipsets indépendants gérant les communications avec 4 GPUs chacun.

GPU	0	1	2	3	4	5	6	7
0		**X**	**X**	**X**				
1	**X**		**X**	**X**				
2	**X**	**X**		**X**				
3	**X**	**X**	**X**					
4						**X**	**X**	**X**
5					**X**		**X**	**X**
6					**X**	**X**		**X**
7					**X**	**X**	**X**	

TABLEAU 6.2 – Possibilités de communication en Peer-to-Peer entre les différents GPUs (**X** pour compatible).

Nous considérons pour ces simulations que la protection ECC des cartes Tesla C2050 reste activée afin de s'assurer de l'exactitude des calculs.

6.3.4 Comparaison à l'utilisation du processeur central

Le but de cette section est de réaliser une comparaison des performances de notre algorithme utilisant plusieurs processeurs graphiques avec notre implémentation utilisant le processeur central (Chapitre 4). Le tableau 6.3 décrit les performances obtenues pour les simulations décrites précédemment pour différentes parallélisations du noyau de simulation. Les tailles des domaines de simulation décrites précédemment sont prévues pour pouvoir tenir en mémoire à l'intérieur d'un unique GPU.

	CPU-OpenMP	CPU-OpenMP-SSE	GPU	multi-GPU
Épandage	5.12	14.32	135.78	1076.75
Descente de bulle	11.43	35.63	281.33	2228.74

TABLEAU 6.3 – Comparaison de performances (en MLUPS) entre différentes implémentations utilisant le processeur central ou les processeurs graphiques.

Ce tableau permet dans un premier temps de mettre en évidence l'efficacité du processeur graphique pour réaliser nos simulations. En effet, la simulation de descente de bulle offre une performance moyenne de 281 MLUPS et la simulation d'épandage offre une performance d'environ 135 MLUPS. En comparant ces résultats avec ceux obtenus sur le processeur central (Chapitre 4), le gain occasionné par l'utilisation d'un processeur graphique avoisine un facteur entre 7 et 10. Dans le cas où seul OpenMP est utilisé, le gain obtenu est de l'ordre de 24 à 27. Cette première comparaison vient conforter le fait que l'utilisation du processeur graphique s'avère être un atout incontestable pour obtenir de bonnes performances de calculs.

Les mêmes remarques peuvent être données quant à l'utilisation de nos huit processeurs graphiques. Le gain obtenu pour ces simulations est de l'ordre de 62 à 74. Les chiffres donnés ici sont les meilleurs en terme de performances obtenues du point de vue des échanges de données. En réalité, les performances obtenues par l'utilisation de plusieurs processeurs graphiques sont fortement dépendantes de l'efficacité des communications entre les processeurs ainsi que de la taille du domaine de simulation.

6.3.5 Influence de l'efficacité des communications

Cette section a pour but d'étudier l'influence des communications entre les processeurs graphiques pour les performances des simulations. L'objectif est également de mettre en avant l'efficacité de l'amélioration des communications au sein de notre algorithme. Nous rappelons que le domaine de simulation est découpé en fonction du nombre de GPUs selon la dimension la plus grande (section 6.2.2). Les tableaux 6.4 et 6.5 permettent de mettre en évidence plusieurs résultats de performances à travers deux moyens de communications entre les GPUs.

Nombre de GPUs	1	2	4	8
Descente de bulle	281.33	541.1	1014.68	2076.38
Épandage à deux composants	135.78	262.96	501.76	997.45

TABLEAU 6.4 – Performance (MLUPS) pour nos simulations en utilisant uniquement des transferts de données à l'aide de la technologie zéro-copy.

Nombre de GPUs	1	2	4	8
Descente de bulle	281.33	542.41	1051.89	2228.74
Épandage à deux composants	135.78	265.46	531.16	1076.75

TABLEAU 6.5 – Performance (MLUPS) pour nos simulations en utilisant des transferts de données en Peer-to-Peer et une répartition efficace à l'aide d'un algorithme de K-means.

Le tableau 6.4 consiste à faire appel uniquement aux transactions zéro-copy généralement utilisées en littérature. Nous pouvons noter que l'utilisation de ce type de transfert offre de bons résultats avec une légère diminution pour plus de quatre GPUs. Cela implique que le temps de transfert des données devient trop important pour être camouflé à partir de ce nombre de GPUs. Les performances restent toutefois très bonnes avec une efficacité de 92 % pour huit GPUs travaillant en parallèle.

L'inclusion de transferts en Peer-to-Peer (Tableau 6.5) permet d'obtenir de meilleures performances. Pour ces cas particuliers de simulations, le gain en performance est de l'ordre de 4% à 6%. Ces résultats montrent également que nous sommes capables d'obtenir un gain presque linéaire selon le nombre de GPUs utilisés pour ces simulations avec une efficacité de 98,5% pour 8 GPUs (Figure 6.16). Le gain de performances occasionné

par les communications en Peer-to-Peer pourrait ainsi être plus important dans d'autres circonstances.

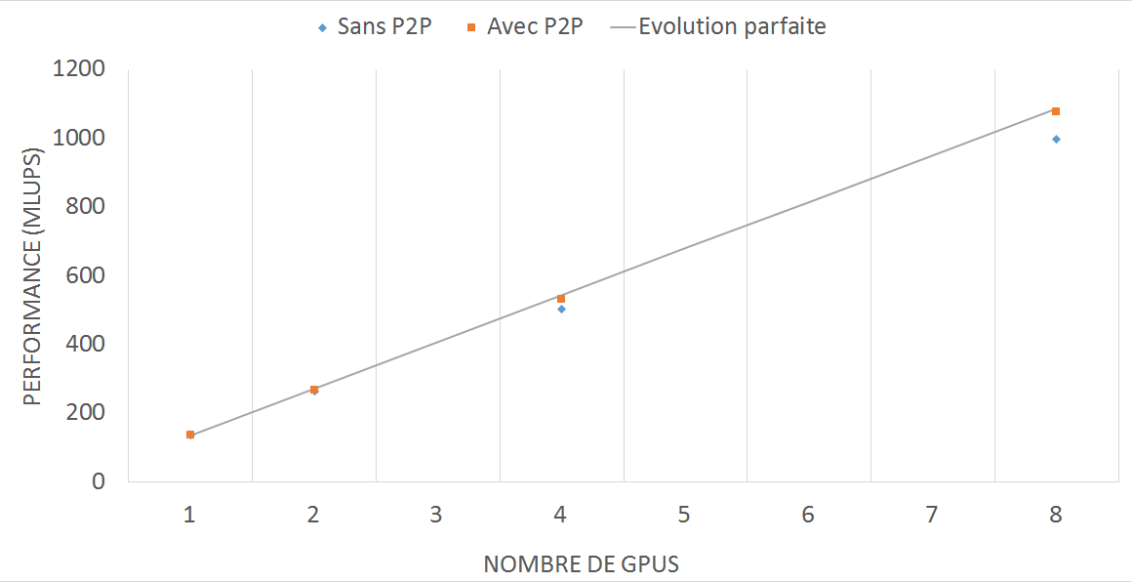


FIGURE 6.16 – Comparaison de performances entre deux moyens de communication de données pour la simulation d'épandage à deux composants.

Les bonnes performances obtenues pour nos simulations proviennent également en partie du fait que les GPUs sont répartis de manière à favoriser l'utilisation de transferts plus rapides en Peer-to-Peer. Le but est ici de mettre en évidence la différence de performances obtenue en comparant la répartition des GPUs à l'aide de l'algorithme K-means avec une répartition aléatoire des GPUs. La figure 6.17 illustre un exemple de comparaison de performances entre une répartition aléatoire des GPUs avec une répartition basée sur l'utilisation d'un algorithme de K-means. Lors des phases de tests, l'utilisation du K-means a toujours montré de meilleures performances qu'une répartition aléatoire. Cela s'explique par le fait qu'une répartition aléatoire ne tient pas compte de la possibilité de communications en Peer-to-Peer. Faciliter l'utilisation de communications en Peer-to-Peer à l'aide d'un algorithme simple et très rapide tel que le K-means est donc un facteur permettant d'améliorer légèrement les performances.

6.3.6 Influence de la taille du domaine

Cette section a pour but de mettre en évidence l'influence importante de la taille du domaine sur les performances d'une simulation. Pour cela, nous étudions les performances

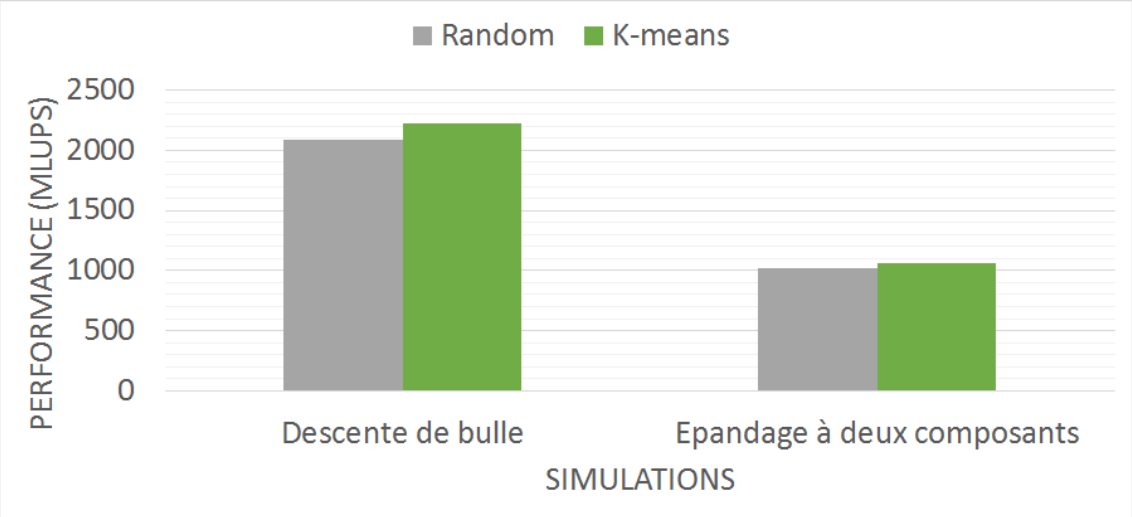


FIGURE 6.17 – Comparaison de performances entre une répartition aléatoire des GPUs avec une répartition basée sur l'utilisation du K-means.

obtenues sur huit GPUs pour les mêmes simulations en utilisant plusieurs tailles de domaine (Figure 6.18).

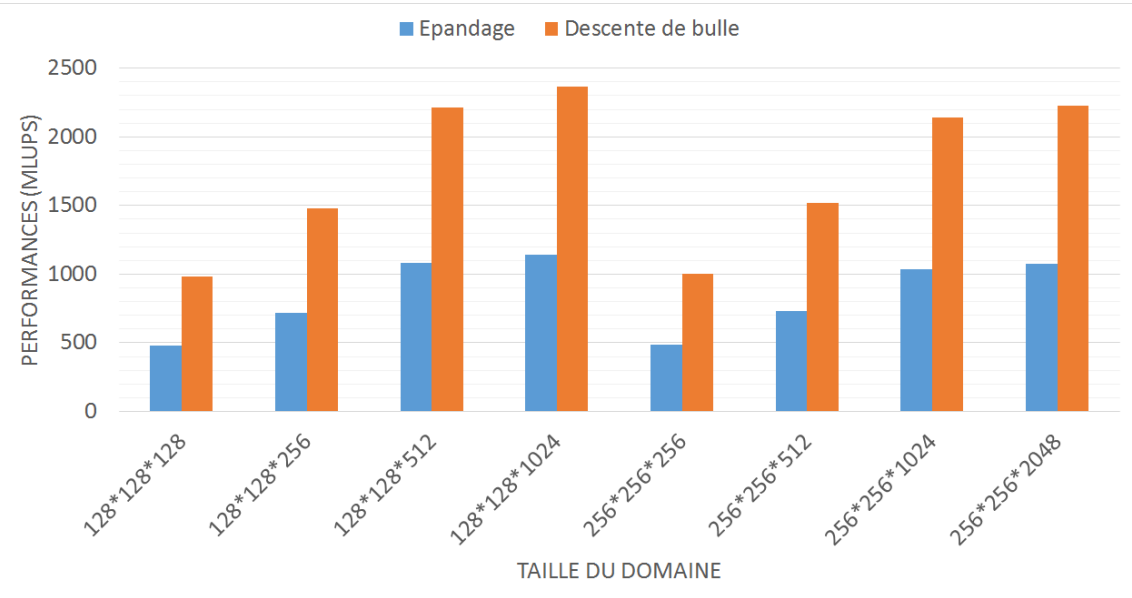


FIGURE 6.18 – Comparaison de performances sur huit GPUs pour des simulations de descente de bulle et d'épandage en considérant plusieurs tailles de domaine.

La figure 6.18 permet de mettre en évidence que réduire la quantité de calculs pour un processeur graphique diminue grandement la possibilité de chevauchement entre calculs et communications. Un découpage trop fin du domaine de simulation implique alors une perte importante de performances. Par exemple, un domaine de simulation de taille

128^3 découpé en 8 représente une perte d'efficacité d'environ 60%. Cette baisse importante de performance est due au fait que le domaine est découpé en sous-domaines de taille $128 \times 128 \times 16$ qui sont bien plus rapides à calculer que les transferts de données aux bords de chaque sous-domaine. L'augmentation progressive de la taille du domaine implique alors un découpage plus grand pour chaque processeur graphique, ce qui est alors synonyme d'une plus grande quantité de calculs pouvant être chevauchés par des échanges de données. Il reste malgré tout primordial de privilégier une quantité de calculs maximale pouvant être chevauché avec les échanges de données entre les processeurs. Dans la mesure du possible, il est important de favoriser un découpage maximisant la quantité de calculs entre les GPUs tout en minimisant les échanges de données à réaliser.

6.4 Conclusion

Ce chapitre a permis d'introduire un nouvel algorithme du noyau de simulation sur une architecture composée de plusieurs processeurs graphiques. L'utilisation d'une telle architecture a permis de mettre en évidence des performances nettement supérieures à une implémentation classique utilisant le processeur central. L'amélioration de la vitesse de simulation a permis de réaliser des simulations sur de grandes grilles de simulation en des temps considérablement réduits.

Les optimisations de calculs du GPU sont des choses bien étudiées en littérature. Nous avons voulu adapter ces optimisations en les associant à l'utilisation de la technique de réduction de mémoire A-A pattern. Des optimisations plus fines des performances restent certainement possibles en utilisant de manière intensive un outil de suivi de performance tel que Nvidia Visual Profiler. Les performances obtenues sur le GPU restent toutefois très intéressantes car elles ont permis d'améliorer de manière considérable les performances par rapport à l'utilisation traditionnelle du CPU.

L'utilisation de plusieurs GPUs en parallèle est une tâche complexe qui demande une bonne connaissance du fonctionnement de CUDA et des moyens de communications entre les différents processeurs. L'utilisation d'une méthode permettant des transferts plus rapides que celle utilisée de manière usuelle a été mise en place au sein de notre noyau de simulation. En effet, les transferts en Peer-to-Peer garantissent des communications

plus rapides à condition qu'ils soient favorisés. C'est pourquoi nous avons choisi de faire appel à un algorithme de K-means permettant de regrouper les sous-domaines proches spatialement à des GPUs pouvant communiquer de cette façon. L'utilisation de transferts en Peer-to-Peer a ainsi démontré un gain maximal de 12 % sur les performances lors des phases de tests sur plusieurs types de simulations. Le gain occasionné par l'utilisation de plusieurs GPUs est finalement fortement corrélé à la taille du domaine de simulation. Le gain parfait peut être obtenu dans des conditions de simulation où l'on favorise une quantité maximale de calculs avec une quantité minimale de communications.

Chapitre 7

Inclusion d'une méthode de maillage progressif du domaine de simulation au sein du noyau de calculs

Sommaire

7.1 Introduction

L'utilisation de la méthode de Boltzmann sur réseau en simulation se fait généralement à l'aide d'une grille cartésienne statique définie au préalable. En effet, l'intégralité des références en littérature se donne une grille de calculs statique et l'ensemble des calculs permettant la simulation est réalisé au sein de cette grille. La méthode a pour avantage d'être simple et d'être adaptée à tout type de simulation. Elle dispose toutefois d'inconvénients lorsque l'on s'attaque à de très grands domaines de simulation. En effet, la représentation de grandes installations industrielles, telles qu'un terminal méthanier, implique alors l'utilisation de grilles de simulation extrêmement grandes si l'on considère des pas de discrétisation réalistes [1]. L'utilisation de grilles de calculs extrêmement grandes impliquent alors une quantité de mémoire bien trop importante à gérer. L'utilisation d'une technique de réduction de mémoire du type A-A pattern permet de réduire ce coût mais la mémoire à gérer reste malgré tout trop importante. De plus, les processeurs graphiques disposent d'une capacité mémoire inférieure à la RAM du processeur central, limitant également la taille maximale de la grille de simulation sur GPU.

Nous proposons dans ce chapitre une solution alternative à l'utilisation d'une grille de simulation statique. Cette idée se base sur le fait qu'une simulation d'écoulement de fluides a généralement un caractère local en début de simulation et se propage au fur et à mesure de la simulation. Ce chapitre a donc pour objectif d'introduire une méthode de maillage progressif du domaine de simulation en fonction du comportement de la simulation. Cette méthode a pour but d'être généraliste et de s'adapter à tout type de simulations et de géométries. Elle a également pour vocation d'être applicable à de nombreux types de modèles utilisant la méthode de Boltzmann sur réseau.

Contrairement à un maillage statique du domaine de simulation, l'utilisation d'une telle méthode a pour objectif de localiser les calculs uniquement aux endroits où ils doivent nécessairement avoir lieu au sein du domaine de simulation. La mémoire est allouée dynamiquement et les calculs se font progressivement suivant l'évolution du maillage. Elle permet ainsi, selon la simulation et la géométrie du domaine, de pouvoir économiser à la fois une énorme quantité de mémoire mais également du temps de calculs. Le gain

1. Un pas de discrétisation de 0,5 cm à 1 cm a été envisagé selon les travaux réalisés par Nicolas Maquignon [N.15]

apporté pour une application sur des installations industrielles comportant généralement beaucoup de bassins d'écoulements fortement canalisés devrait alors être très important.

La section 7.2 décrit le principe de la méthode de maillage progressif dans un cas générique. Cela inclut également l'intégration de la méthode sur le processeur graphique pour notre noyau de simulation. La section 7.3 quant à elle, présente l'intégration de cette méthode sur un nœud de calculs composé de plusieurs processeurs graphiques. Cela soulève un certain nombre de contraintes et de problématiques en ce qui concerne la répartition de la charge de calculs ainsi que dans la gestion des communications entre les différents processeurs.

7.2 Principe de la méthode

L'ensemble des travaux référencés dans ce manuscrit fait généralement appel à une grille de calculs statique pour la réalisation de simulations physiques à l'aide de la méthode de Boltzmann sur réseau. L'approche que nous voulons mettre en avant dans cette section concerne la définition d'une méthode de maillage progressif du domaine de simulation. L'idée est de faire progresser automatiquement le maillage en fonction de la dynamique de la simulation. Pour cela, des petites parties du domaine de simulation vont être créées de manière dynamique et être connectées les unes aux autres en suivant un critère bien défini, de façon à suivre l'évolution de la simulation.

La mise en place d'une telle méthode impose alors plusieurs problématiques. Cela passe dans un premier temps par le placement d'un point de départ dans un maillage minimal afin de pouvoir initialiser la simulation. La définition d'un critère fiable et précis permettant de mailler progressivement le domaine de simulation est également une étape importante de la méthode.

7.2.1 Initialisation de la simulation et progression du maillage

L'initialisation de la simulation est une étape importante pour la méthode de maillage progressif, car c'est elle qui va être responsable du comportement de la simulation pour les premières itérations. L'idée, pour initialiser la simulation, est de définir un sous-domaine

au niveau de l'entrée du fluide et de démarrer les calculs uniquement au sein de ce sous-domaine (Figure 7.1).

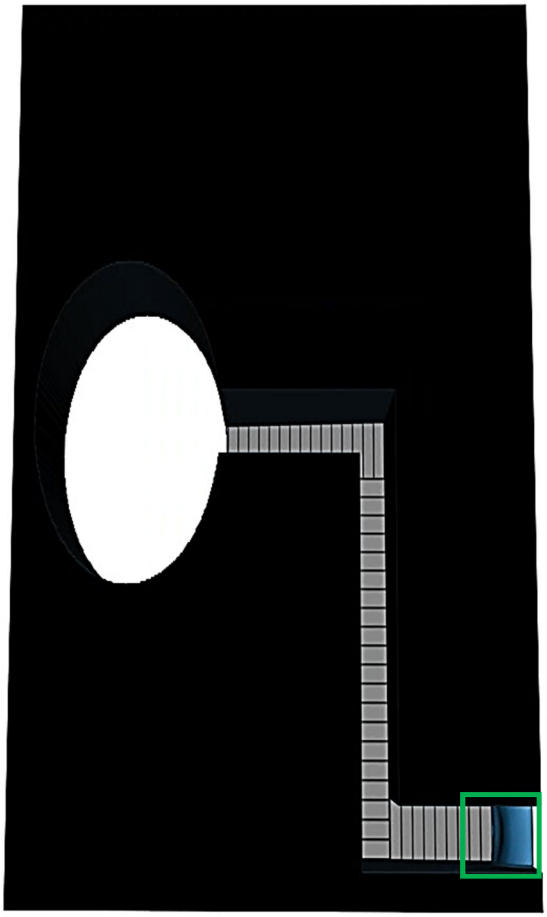
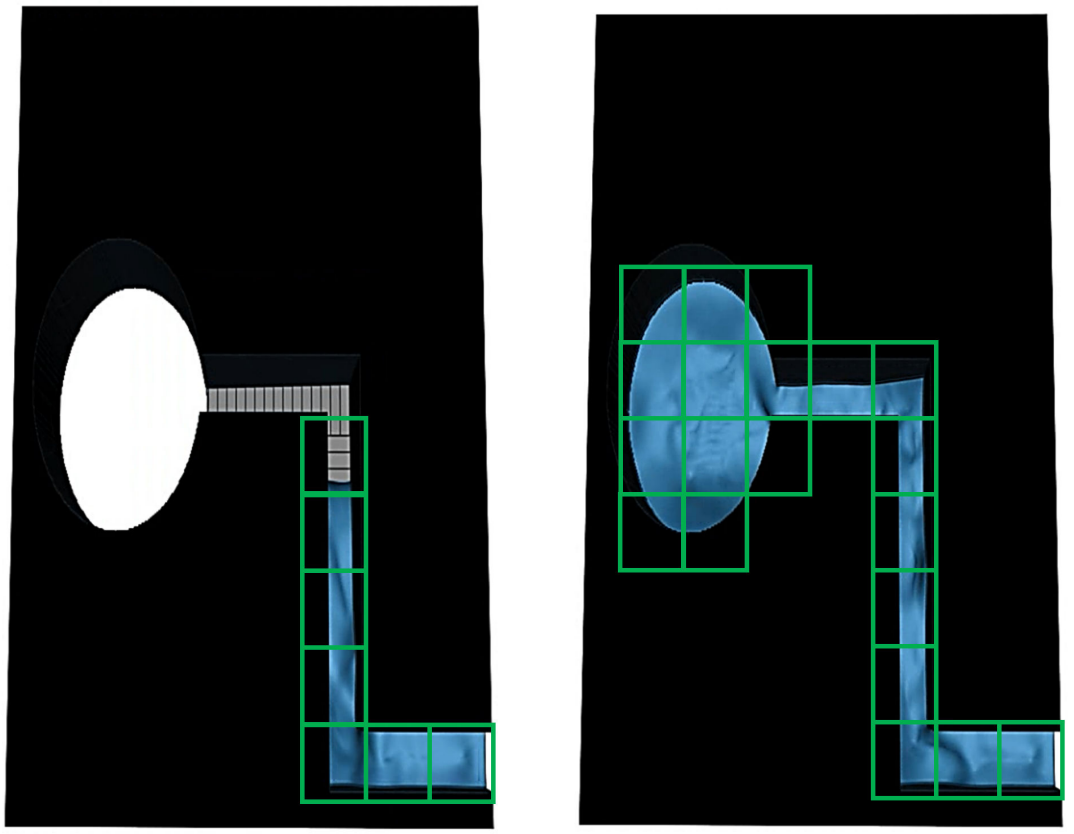


FIGURE 7.1 – Initialisation de la simulation en créant un premier sous-domaine (délimité par le carré vert) à l'intérieur du domaine de simulation : les calculs se font dans un premier temps uniquement au sein de ce sous-domaine.

Dans des cas de scénarii à plusieurs entrées de fluides, on définit de la même façon un sous-domaine pour chaque entrée. L'idée est simplement de couvrir toutes les entrées possibles de fluides avec un sous-domaine au début de la simulation. C'est à l'utilisateur de placer et de définir les sous-domaines nécessaires pour démarrer la simulation.

Une fois l'initialisation terminée, la simulation peut démarrer sur ces sous-domaines. En fonction de la propagation du ou des fluides, de nouveaux sous-domaines vont venir s'ajouter dynamiquement au sein du domaine de simulation (Figure 7.2(a)) jusqu'à atteindre un état de convergence dans lequel plus aucun sous-domaine n'apparaît. (Figure 7.2(b)).



(a) (b)

FIGURE 7.2 – Ajout progressif de sous-domaines en fonction de la dynamique de la simulation : des sous-domaines sont ajoutés progressivement jusqu'à atteindre un état de convergence dans lequel plus aucun sous-domaine n'apparaît.

L'ajout dynamique de sous-domaines implique dans un premier temps de pouvoir repérer spatialement un sous-domaine donné. Le domaine de simulation global est dans un premier temps voxelisé sur le processeur central de manière à définir l'état géométrique de toutes les cellules le constituant. Cela permet d'avoir une connaissance au préalable de l'état de chaque cellule du domaine de simulation global et ainsi d'initialiser les nouveaux sous-domaines plus facilement. De cette façon, seul l'état géométrique de l'intégralité du domaine est défini. Les valeurs liées à la simulation (fonctions de distribution, forces quantités macroscopiques, etc) ne sont définis que pour les sous-domaines ajoutés dynamiquement.

La création d'un nouveau sous-domaine implique de pouvoir le définir et le repérer spatialement. Il est par conséquent nécessaire de pouvoir faire le lien entre les coordonnées des cellules du domaine de simulation global avec les coordonnées locales d'un sous-

domaine. En effet, deux types de coordonnées sont utilisées pour repérer une cellule : les coordonnées globales du domaine de simulation permettant de trouver l'état géométrique de chaque cellule et les coordonnées locales permettant de réaliser les différents calculs de l'algorithme du noyau de simulation (Figure 7.3).

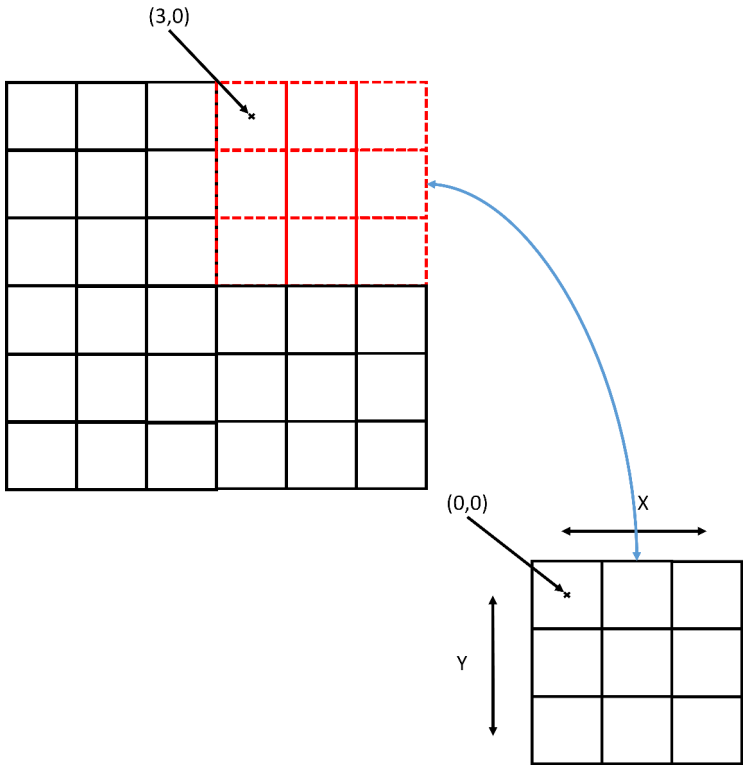


FIGURE 7.3 – Illustration en 2D de l'utilisation de deux espaces de coordonnées : les coordonnées du domaine de simulation global et les coordonnées locales du sous-domaine.

Les coordonnées globales sont utilisées pour définir dans un premier temps l'état géométrique de l'ensemble des cellules d'un sous-domaine. Pour cela, on utilise les coordonnées globales du coin supérieur gauche du sous-domaine et sa taille en X, Y, Z pour définir l'état géométrique de chaque cellule. Une fois l'état géométrique défini pour chaque cellule du sous-domaine, les calculs peuvent se faire naturellement à l'aide des coordonnées locales du sous-domaine. L'ajout dynamique de ces différents sous-domaines ne peut toutefois être réalisé qu'à l'aide de plusieurs conditions :

1. La définition d'un critère de progression adapté pour la progression du maillage.
2. Une bonne gestion des communications entre les sous-domaines.

7.2.2 Définition d'un critère de progression

La définition d'un critère efficace pour la progression du maillage est une étape importante pour notre méthode. En effet, un critère inadapté pourrait conduire à des ajouts inutiles ou à des ajouts trop tardifs ce qui pourrait ainsi conduire à une simulation faussée. Il doit alors représenter de manière efficace la propagation du ou des fluides à l'intérieur du domaine.

L'utilisation d'un critère temporel basé sur le nombre d'itérations semble par exemple inadapté car il est fortement arbitraire et peut conduire à des erreurs. Ce premier critère ne tient pas compte de la dynamique du fluide et ne peut par conséquent pas être représentatif du comportement de la simulation.

Le critère que nous avons mis en place se base sur la vitesse du ou des fluides intervenant au sein de la simulation. La vitesse d'un fluide est en effet une valeur exploitable pour représenter la dynamique de la simulation. L'idée est de considérer la différence de vitesse entre deux itérations de l'algorithme afin d'observer de manière efficace la dispersion d'un fluide. Il est par conséquent défini de la façon suivante :

$$||C_\alpha(x)||_2 = ||u_\alpha(x,t+\Delta t) - u_\alpha(x,t)||_2 \tag{7.1}$$

où le symbole $||.||_2$ correspond ici à la norme euclidienne et α à l'indice du composant.

L'idée est ensuite de calculer ce critère pour chaque sous-domaine existant et pour chaque composant présent au sein de la simulation. Pour réduire la quantité de calculs nécessaire pour le calcul du critère, il est calculé uniquement sur les cellules aux bords des différents sous-domaines. Si la valeur de ce critère dépasse un certain seuil S, alors un nouveau sous-domaine est créé à côté du bord concerné (Figure 7.4). La valeur de S est généralement fixée à 0 de manière à détecter tout changement pour chaque sous-domaine. De cette façon, aucune erreur n'est commise car un sous-domaine est créé pour tout changement de vitesse. L'utilisation d'une valeur $S > 0$ est également possible mais elle peut avoir un impact néfaste sur l'exactitude de la simulation car les sous-domaines peuvent alors être ajoutés trop tardivement.

La création des nouveaux sous-domaines étant réalisée de manière dynamique, ceux-ci sont alloués dynamiquement sur des zones mémoires différentes sur le GPU. Les zones

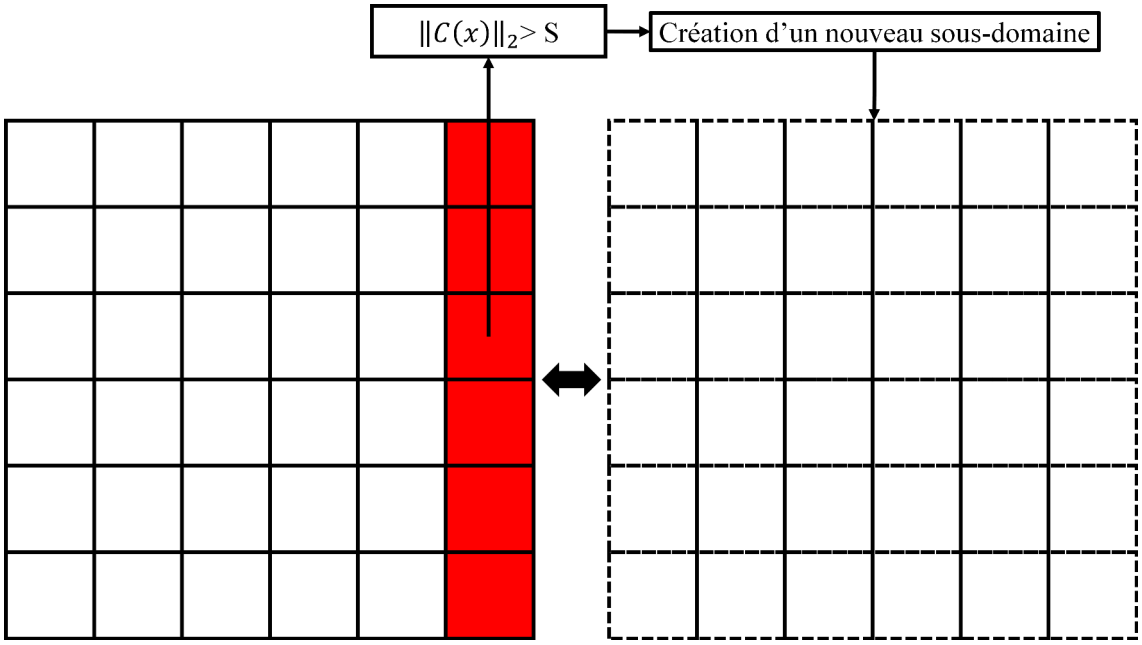


FIGURE 7.4 – Le critère $||C_\alpha(x)||_2$ est calculé sur un bord : si la valeur dépasse un certain seuil S, alors un nouveau sous-domaine est créé à côté de ce bord.

mémoires étant différentes, les sous-domaines devront par conséquent être amenés à échanger des données. Cela implique alors de mettre en place une gestion efficace des communications entre les sous-domaines.

7.2.3 Gestion des communications entre les sous-domaines

L'ajout dynamique de sous-domaines sur des zones mémoires distinctes impose une communication de données entre ces sous-domaines. Les échanges de données à réaliser sont les mêmes que pour les communications entre les GPUs lorsque le domaine est divisé (section 6.2). Cela impose donc un premier échange pour le pseudo-potentiel ψ et un second pour les fonctions de distribution f_i se propageant vers les sous-domaines voisins. Une différence importante apparaît toutefois ici. L'apparition dynamique de sous-domaines impose de définir un moyen permettant de savoir quels sont les échanges à réaliser. De manière simple, il faut savoir pour un sous-domaine donné avec quels autres sous-domaines il va être amené à échanger des données.

L'idée est alors de définir une notion de voisinage entre les sous-domaines. Le but est de connaître les connexions entre les différents sous-domaines. Pour cela, on définit dans un premier temps un indice pour chaque sous-domaine. Un nombre entier est géné-

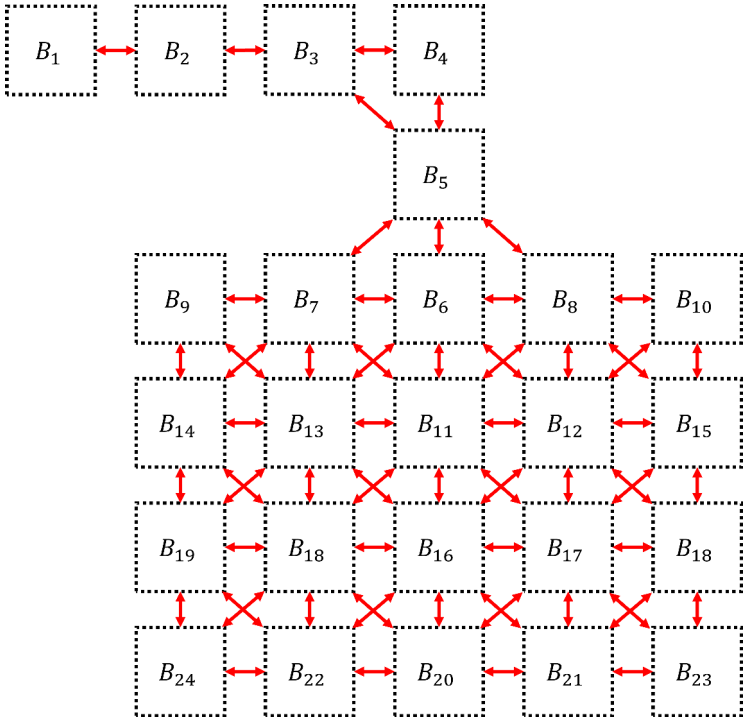


FIGURE 7.5 – Illustration 2D des communications à réaliser entre les différents sous-domaines B_i.

ralement utilisé pour définir cet indice. Les différents indices sont ensuite connectés les uns aux autres sous la forme d'un graphe (Figure 7.5). Le nombre de voisins possibles pour un sous-domaine est limité. En réalité, ce nombre est défini par le schéma utilisé pour la méthode de Boltzmann sur réseau. L'utilisation d'un schéma D3Q19 impose alors dix-huit voisins possibles pour un sous-domaine. La création dynamique d'un nouveau sous-domaine impose maintenant de connecter celui-ci aux sous-domaines existants. Pour cela, on compare la localisation spatiale de ce nouveau sous-domaine avec ceux existants aux alentours. Un tableau contenant dix-huit nombres entiers est alors défini. La valeur des entrées dans ce tableau est définie par défaut à -1 lorsque le voisin n'existe pas. Si un voisin existe, alors on inclut dans ce tableau la valeur de l'indice du sous-domaine (Figure 7.6). Un ordre précis de stockage de l'information dans ce tableau doit être respecté pour définir de manière cohérente les voisins. Cet ordre peut être défini de manière arbitraire par l'utilisateur mais nous préférons généralement suivre le même schéma de voisinage que pour les vecteurs de liaison de la méthode de Boltzmann.

La définition de ce tableau pour chaque sous-domaine permet ainsi de représenter les différentes connexions entre les sous-domaines. Ces connexions étant définies, il est par conséquent possible de savoir aisément quels sont les échanges de données à réaliser entre

les sous-domaines.

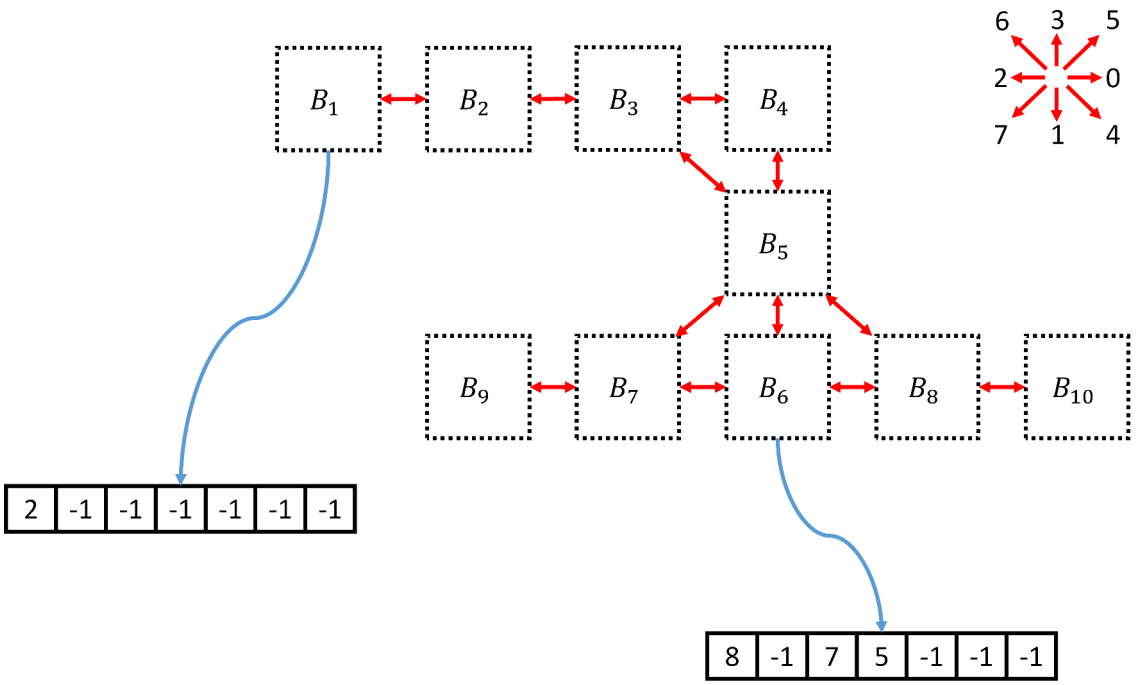


FIGURE 7.6 – Exemples de stockage des liaisons entre les différents sous-domaines dans un cas 2D : chaque sous-domaine dispose d'un tableau contenant les indices des voisins existants.

7.2.4 Algorithme

Le but de cette section est de mettre en évidence les différents apports concernant la méthode de maillage progressif pour une utilisation sur un processeur graphique, par le biais d'un algorithme (Algorithme 6) qui inclut des échanges de données entre les différents sous-domaines. Ces sous-domaines étant situés sur la mémoire globale du GPU, on ne parle pas ici d'échanges de données entre deux mémoires mais d'échanges entre deux zones d'une même mémoire. Cela implique que ces échanges sont très peu coûteux mais imposent ici d'être réalisés de manière séquentielle aux calculs. En revanche, une intégration sur une architecture composée de plusieurs processeurs graphiques impose de nouvelles problématiques en ce qui concerne les communications de données et la répartition de la charge de calculs.

Algorithme 6 : Algorithme du noyau de simulation utilisant une méthode de maillage progressif sur le processeur graphique

↪ Définition de la géométrie du domaine de simulation global ;
↪ Création d'un ou plusieurs sous-domaines en fonction des entrées de fluides ;
pour *chaque itération* **faire**
  pour *chaque sous-domaine existant* **faire**
    pour *chaque composant* σ **faire**
      ↪ Calcul sur le GPU de p_σ pour le sous-domaine (Équation (2.23)) ;
      ↪ Calcul sur le GPU de p_σ pour le sous-domaine (Équation (2.24)) ;
      ↪ Communications entre les différentes zones mémoires de ψ_σ aux interfaces entre les sous-domaines ;
      ↪ Correction des erreurs aux conditions limites et aux obstacles ;
    fin
  fin
  si *itération paire* **alors**
    pour *chaque sous-domaine existant* **faire**
      pour *chaque composant* σ **faire**
        ↪ Calcul des forces sur le GPU à l'aide des équations (2.25) à (2.28) ;
        ↪ Calcul de la première fonction d'équilibre $f^{eq}_{\sigma,i}$ sur le GPU à l'aide de l'équation 2.7 ;
        ↪ Somme des forces et correction du terme de vitesse Δu_σ sur le GPU à l'aide de l'équation (2.31) ;
        ↪ Calcul de la seconde fonction d'équilibre $f^{eq}_{\sigma,i}$ sur le GPU à l'aide de l'équation 2.7 ;
        ↪ Calcul des temps de relaxation avec un modèle de sous-maille de Smagorinsky sur le GPU pour le *A-step* de la méthode A-A pattern ;
        ↪ Étape de collision-propagation des fonctions de distribution $f_{\sigma,i}$ sur le GPU pour le *A-step* de la méthode A-A pattern ;
        ↪ Communiquer entre les différentes zones mémoires les valeurs $f_{\sigma,i}$ nécessaires aux interfaces entre les sous-domaines ;
        ↪ Corrections aux conditions limites et aux obstacles sur le GPU pour le *A-step* de la méthode A-A pattern;
        ↪ Calcul du critère de progression $||C_\alpha(x)||_2$;
        ↪ Calcul des quantités macroscopiques ρ_σ et u_σ sur le GPU pour le *A-step* de la méthode A-A pattern à l'aide des équations (2.10) et (2.11) ;
      fin
    fin
  sinon
    ↪ Réalisation des mêmes étapes de calculs que pour les itérations paires mais dans le cas du *B-step* de la méthode A-A pattern ;
  fin
  pour *chaque sous-domaine existant* **faire**
    pour *chaque composant* σ **faire**
      si $||C_\alpha(x)||_2 > S$ *sur un des bords* **alors**
        ↪ Création d'un nouveau sous-domaine ;
        ↪ Connexion avec les sous-domaines existants ;
      fin
    fin
  fin
fin

7.3 Intégration au sein d'une architecture composée de plusieurs processeurs graphiques

L'utilisation d'une méthode de maillage progressif associée à une architecture constituée de plusieurs processeurs graphiques travaillant de manière parallèle impose plusieurs problématiques. L'ajout dynamique des sous-domaines impose en premier lieu une contrainte sur la répartition de la charge de calculs. En effet, la quantité de calculs évolue de manière progressive et une répartition équitable devient beaucoup plus complexe que dans un cas purement statique. De la même façon, l'utilisation de plusieurs processeurs graphiques vient de nouveau poser une problématique de gestion des échanges de données. Les transferts de données représentent souvent un facteur handicapant en ce qui concerne les performances. Il est donc important de trouver un moyen d'optimiser ces transferts de données.

7.3.1 Répartition de la charge de calculs

Le principe du maillage progressif consiste à créer de manière dynamique des sous-domaines en fonction de la progression d'un ou de plusieurs fluides. La création de ces sous-domaines vient poser une problématique sur la répartition des calculs sur les différents processeurs graphiques. Apporter une solution permettant une répartition optimale des calculs est dans ce cas extrêmement complexe. En effet, l'augmentation progressive de la quantité de calculs est le facteur majeur rendant cette répartition très complexe. Dans ce cas, maintenir un équilibrage permanent de la charge de calculs est difficile. Le choix de la taille d'un sous-domaine est également un facteur influant sur la charge de calculs. Même s'il est préférable que la taille du sous-domaine soit cubique pour ne favoriser aucune direction de propagation, choisir une taille de sous-domaine optimale reste complexe. De plus, la progression du maillage, et donc de la charge de calculs, est fortement influencée par la géométrie constituant le domaine de simulation (généralement définie arbitrairement par l'utilisateur selon ses besoins). Bien qu'il semble ne pas exister de solution optimale pour répartir la charge de calculs, nous proposons trois approches différentes répondant à cette problématique.

La première solution consiste à diviser chaque sous-domaine suivant le nombre de GPUs disponibles. Cette idée permettrait en effet d'avoir une répartition équitable des calculs pour chaque processeur graphique. En revanche, cette méthode multiplie de façon extrêmement importante le nombre d'échanges à réaliser entre les processeurs. En effet, plus le nombre de sous-domaines sera important plus la quantité de communications à réaliser entre les processeurs graphiques sera importante (Figure 7.7). Cette première solution permet par conséquent de conserver un équilibrage de charge permanent au détriment de la quantité d'échanges de données à réaliser. En effet, le nombre d'échanges à réaliser dans un cas 3D peut devenir extrêmement important puisque dix-huit voisins sont considérés, soit dix-huit possibilités d'échange de données pour un unique sous-domaine.

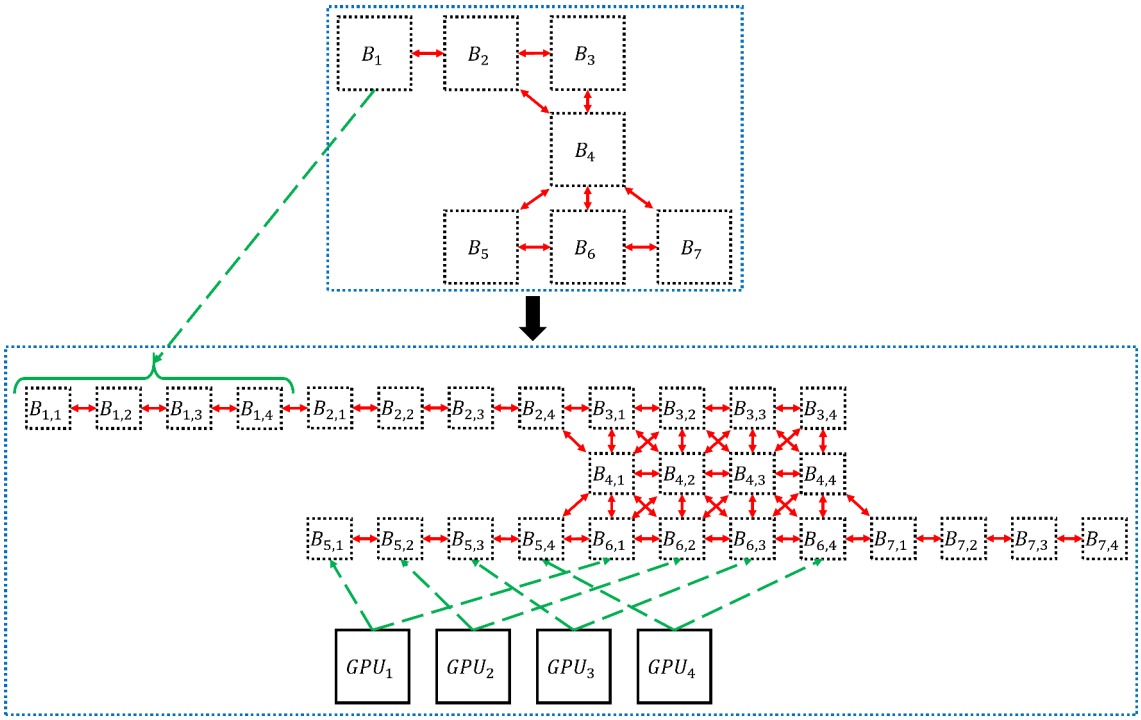


FIGURE 7.7 – Exemple 2D d'une première répartition de la charge de calculs : chaque sous-domaine est divisé en blocs suivant le nombre de GPUs disponibles. Un bloc d'un sous-domaine est alors assigné à un unique GPU.

La seconde solution étudiée consiste simplement à affecter un sous-domaine à un GPU. Dans ce cas, aucune communication supplémentaire n'est requise et les échanges de données restent au niveau des sous-domaines. L'utilisation d'une telle méthode dispose toutefois d'un inconvénient majeur concernant l'équilibrage de la charge de calculs. En effet, les sous-domaines s'ajoutant de manière progressive au fur et à mesure de la simu-

lation, l'ensemble des processeurs graphiques ne disposent généralement pas de la même charge de calculs (Figure 7.8).

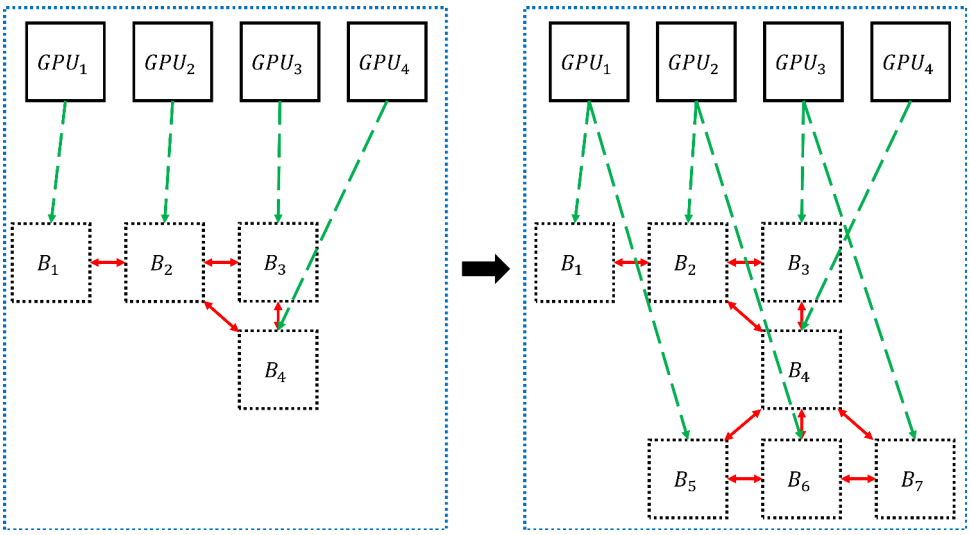


FIGURE 7.8 – Exemple 2D d'une seconde répartition de la charge de calculs : chaque sous-domaine est assigné à un GPU et ce GPU gère l'ensemble des calculs sur ce sous-domaine.

Une dernière solution consiste à réaliser un équilibrage de charges dynamique en tenant compte de la progression du maillage, c'est-à-dire réévaluer la répartition globale de la charge de calculs sur les différents processeurs graphiques à chaque fois qu'un nouveau sous-domaine est créé. Cette méthode implique en réalité de découper et répartir équitablement l'ensemble des sous-domaines en fonction de la charge. L'utilisation d'une architecture à mémoire distribuée rend cette approche également peu performante car cela impose de transférer des quantités très importantes de données entre les différentes cartes (Figure 7.9).

La répartition de la charge de calculs sur plusieurs processeurs graphiques pour la méthode de maillage progressif est une tâche complexe et il semblerait qu'aucune solution optimale ne soit envisageable à l'heure actuelle. Nous avons envisagé plusieurs solutions permettant de répartir la charge sur les différentes GPUs en tenant compte de l'aspect dynamique de la création du maillage. Dans notre cas, nous avons préféré opter pour la seconde solution qui, contrairement aux deux autres, n'augmente pas la quantité d'échanges de données, mais ceci au détriment de l'équilibrage de charge. Ce choix est basé sur le fait qu'il est préférable de minimiser la quantité de données à échanger entre les processeurs plutôt que de maximiser la puissance de calculs. En effet, les échanges corres-

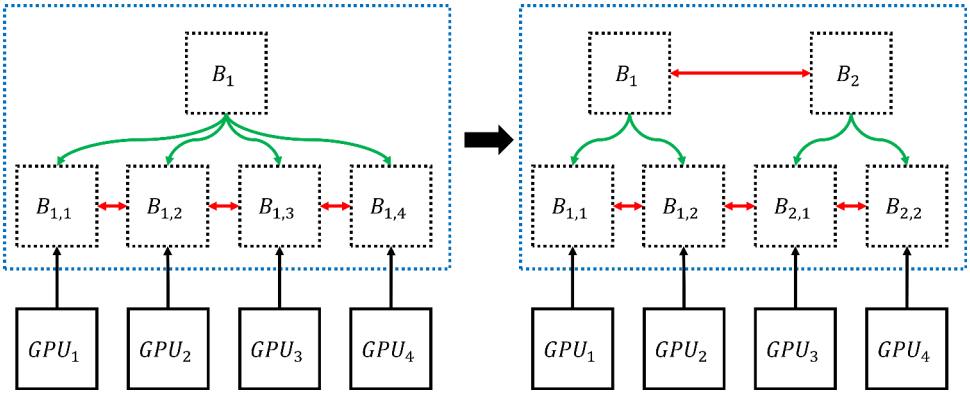


FIGURE 7.9 – Exemple 2D de répartition dynamique de la charge de calculs : les quatre GPUs sont dans un premier temps répartis sur le premier sous-bloc. À l'ajout d'un second sous-domaine, les quatre GPUs sont de nouveau répartis sur les deux sous-domaines.

pondent généralement au facteur le plus limitant pour ce type d'applications [OKTR13b]. C'est pourquoi nous favorisons une approche utilisant le moins de communications possibles à effectuer entre les différents processeurs. La mise en place d'un équilibrage de calcul optimal dans cette situation de maillage progressif s'avère extrêmement complexe et d'autres solutions seront recherchées à la suite de ce travail de thèse.

7.3.2 Optimisation des transferts de données

L'approche choisie dans la section précédente pour répartir la charge de calculs sur les différents processeurs graphiques vise à minimiser le nombre d'échanges de données à réaliser. Cette section vise maintenant à optimiser dynamiquement la répartition des GPUs aux différents sous-domaines, afin de maximiser l'efficacité des échanges à réaliser. Nous définissons pour cela trois moyens de communications entre les sous-domaines. Le premier consiste à échanger des données entre des sous-domaines appartenant au même GPU. Dans ce cas, le coût de communication est quasi nul car l'ensemble des données est stocké sur la même mémoire. Le deuxième et le troisième moyens de communication concernent en revanche les cas où les sous-domaines devant échanger des données appartiennent à des GPUs différents. Une distinction est toutefois réalisée entre les communications en Peer-to-Peer et les communications utilisant la technologie zéro-copy. Le but est de réaliser une optimisation des transferts à l'aide de ces trois possibilités. Pour cela, nous définissons une fonction F qui est définie pour un sous-domaine G de la manière suivante :

$$F(G) = \sum_{G'} \gamma(G, G') \tag{7.2}$$

où l'indice G' désigne l'ensemble des sous-domaines voisins à G. La fonction $\gamma(G, G')$ est quant à elle définie de la façon suivante :

$$\gamma(G, G') = \begin{cases} 0, & \text{si GPU}(G) = \text{GPU}(G') \\ 0,5 * sizeof(transfert), & \text{si GPU}(G) \neq \text{GPU}(G') \text{ et GPU}(G) \text{ P2P GPU}(G') \\ sizeof(transfert), & \text{si GPU}(G) \neq \text{GPU}(G') \text{ et GPU}(G) \text{ non P2P GPU}(G') \end{cases} \tag{7.3}$$

La fonction $\gamma(G, G')$ compare en réalité les trois moyens de communications entre le sous domaine G et son voisin G'. La valeur 0 est donnée si les deux sous-domaines sont situés sur le même GPU. La taille en octets de l'échange à réaliser avec le sous-domaine est considérée dans les autres cas. Une valeur arbitraire est incluse dans le cas de communications en Peer-to-Peer de manière à favoriser son utilisation. Dans notre cas, nous avons choisi de fixer cette valeur à 0.5 mais elle peut être modifiée au besoin. La fonction F regroupe quant à elle la somme des valeurs de γ pour l'ensemble des voisins existants. Elle doit par conséquent être minimisée de manière à obtenir le meilleur coût de communication. Pour cela, la fonction F est calculée pour l'ensemble des GPUs disponibles et le GPU avec la valeur minimale est affecté au nouveau sous-domaine. Pour un petit nombre de GPUs, cela implique une quantité de calculs négligeable qui peut être réalisée de manière très rapide en parallèle sur le processeur central.

De manière à répartir le plus équitablement possible la charge de calculs, un même GPU ne peut pas être affecté deux fois successivement tant que les autres processeurs n'ont pas été affectés. Cela veut dire que l'on affecte en priorité les GPUs les moins chargés de manière à garder un écart de la charge de calculs entre les GPUs limité au maximum à un sous-domaine.

Cela permet de faire travailler les différents GPUs en parallèle de manière successive. La figure 7.10 met en lumière le principe de notre optimisation par le biais d'un exemple simple. Dans cet exemple, deux sous-domaines apparaissent de manière successive et

notre optimisation est utilisée pour affecter un GPU à ces nouveaux sous-domaines.

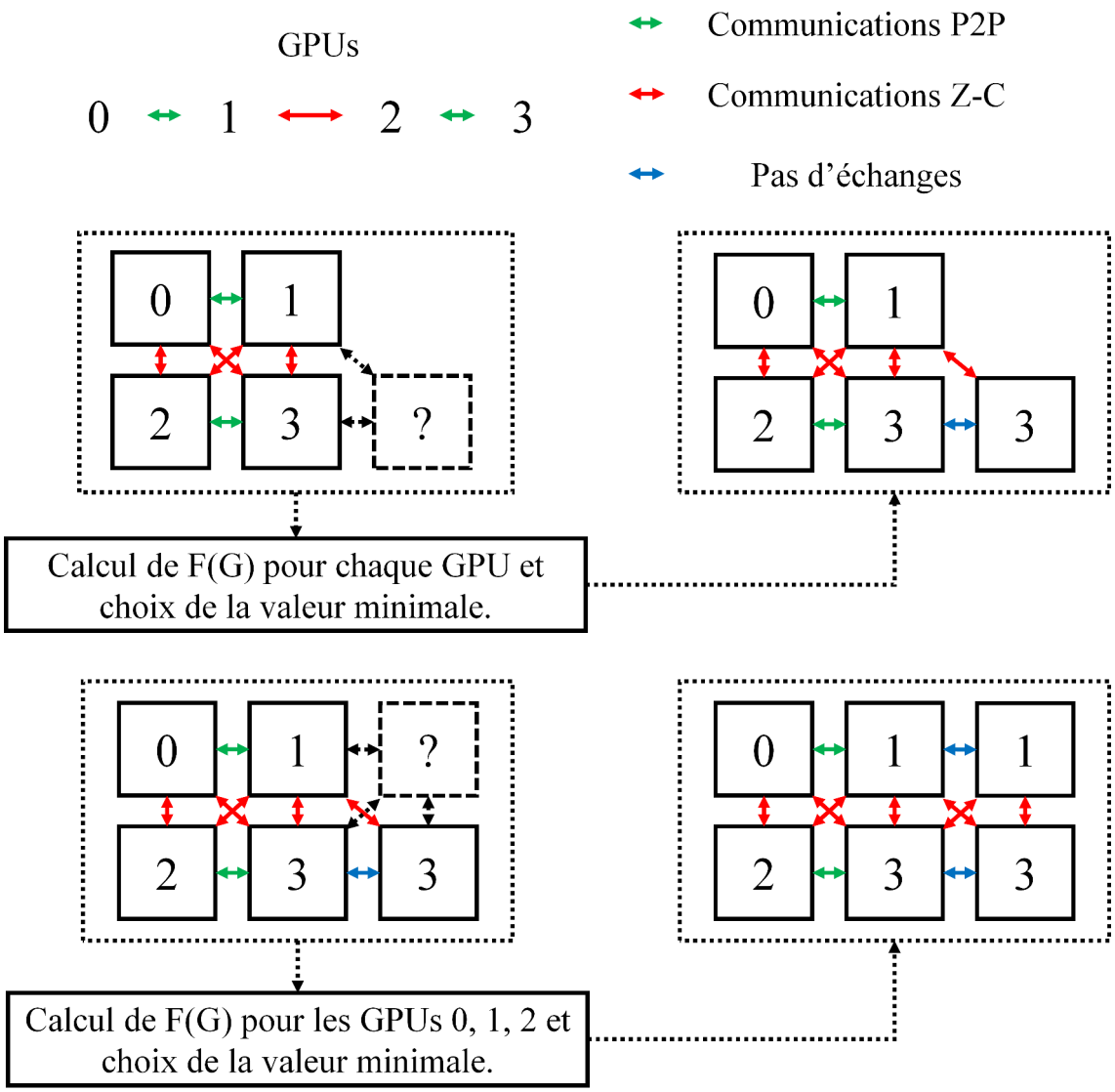


FIGURE 7.10 – Exemple 2D de l'optimisation de la répartition des GPUs : la fonction $F(G)$ est calculée pour chaque GPU disponible et le GPU avec la valeur minimale est choisi.

7.4 Évaluation des performances

7.4.1 Algorithme

Le but de cette section est de résumer et de combiner les différents apports présentés jusqu'à maintenant au travers d'un algorithme scindé en deux parties (Algorithmes 7 et 8). Il combine le travail d'intégration du noyau de simulation sur une architecture

composée de plusieurs processeurs graphiques (Algorithmes 4 et 5) avec la méthode de maillage progressif du domaine de simulation (Algorithme 6). L'algorithme intègre également les différentes optimisations réalisées pour améliorer les performances de la méthode de maillage progressif en combinaison avec l'utilisation de plusieurs processeurs graphiques.

Algorithme 7 : Première partie de l'algorithme : initialisation de la simulation, création des nouveaux sous-domaines et calcul de la pression p_σ et du pseudo-potentiel ψ_σ sur plusieurs processeurs graphiques ; la partie en rouge s'exécute de manière concurrente à la partie en bleu.

↪ Définition de la géométrie du domaine de simulation global ;
↪ Création d'un ou plusieurs sous-domaines en fonction des entrées de fluides ;
pour *chaque sous-domaine existant* **faire**
  ↪ Affectation par l'utilisateur d'un GPU ;
fin
↪ Création de threads CPU permettant de gérer les différents GPUs ;
pour *chaque itération* **faire**
  pour *chaque sous-domaine existant* **faire**
    pour *chaque composant* σ **faire**
      si $||C_\alpha(x)||_2 > S$ *sur un des bords* **alors**
        ↪ Création d'un nouveau sous-domaine ;
        ↪ Connexion avec les sous-domaines existants ;
        ↪ Calcul de la fonction F pour les GPUs disponibles ;
        ↪ Détermination de la valeur minimale et affectation du GPU associé ;
      fin
    fin
  fin
  pour *chaque sous-domaine existant* **faire**
    pour *chaque composant* σ **faire**
      ↪ Calcul à l'aide d'un stream de copie sur le GPU associé de p_σ sur les bords concernés avec l'équation (2.23) ;
      ↪ Calcul à l'aide d'un stream de copie sur le GPU associé de ψ_σ sur les bords concernés avec l'équation (2.24) ;
      ↪ Communications à l'aide d'un stream de copie de ψ_σ aux interfaces entre les sous-domaines : Utilisation de transferts en Peer-to-Peer pour les GPUs compatibles et de transactions zéro-copy sinon ;
      ↪ Calcul à l'aide d'un stream de calculs sur le GPU associé de p_σ pour le reste du sous-domaine (Équation (2.23)) ;
      ↪ Calcul à l'aide d'un stream de calculs sur le GPU associé de p_σ pour le reste du sous-domaine (Équation (2.24)) ;
      ↪ Correction à l'aide d'un stream de calculs des erreurs aux conditions limites et aux obstacles ;
    fin
  fin
  ↪ Synchronisation des GPUs ;
  ↪ Suite des calculs : Algorithme 8 ;
fin

Algorithme 8 : Seconde partie de l'algorithme : calcul des forces, collision-propagation des fonctions de distribution et calcul du critère de progression du maillage ; les étapes en bleu et en rouge peuvent s'effectuer de manière concurrente.

si *itération paire* **alors**
- **pour** *chaque sous-domaine* **faire**
  - **pour** *chaque composant* σ **faire**
    - ↪ Calcul des forces aux bords concernés à l'aide d'un stream de copie sur le GPU associé (Équations (2.25)-(2.28)) ;
    - ↪ Calcul de la première fonction d'équilibre $f^{eq}_{\sigma,i}$ aux bords concernés à l'aide d'un stream de copie sur le GPU associé (Équation (2.7)) ;
    - ↪ Somme des forces et correction du terme de vitesse Δu_σ sur les bords concernés à l'aide d'un stream de copie sur le GPU associé (Équation (2.31)) ;
    - ↪ Calcul de la seconde fonction d'équilibre $f^{eq}_{\sigma,i}$ sur les bords concernés à l'aide d'un stream de copie sur le GPU associé (Équation (2.7)) ;
    - ↪ Calcul des temps de relaxation avec un modèle de sous-maille de Smagorinsky aux bords concernés pour le *A-step* de la méthode A-A pattern à l'aide d'un stream de copie sur le GPU associé ;
    - ↪ Étape de collision-propagation des fonctions de distribution $f_{\sigma,i}$ aux bords concernés à l'aide d'un stream de copie sur le GPU associé pour le *A-step* de la méthode A-A pattern ;
    - ↪ Communiquer les valeurs $f_{\sigma,i}$ nécessaires aux interfaces entre les sous-domaines à l'aide d'un stream de copie : Utilisation de transferts en Peer-to-Peer pour les GPUs compatibles et de transactions zéro-copy sinon ;
    - ↪ Calcul des forces sur le GPU associé à l'aide des équations (2.25) à (2.28) ;
    - ↪ Calcul de la première fonction d'équilibre $f^{eq}_{\sigma,i}$ sur le GPU associé à l'aide de l'équation 2.7 ;
    - ↪ Somme des forces et correction du terme de vitesse Δu_σ sur le GPU associé à l'aide de l'équation (2.31) ;
    - ↪ Calcul de la seconde fonction d'équilibre $f^{eq}_{\sigma,i}$ sur le GPU associé à l'aide de l'équation 2.7 ;
    - ↪ Calcul des temps de relaxation avec un modèle de sous-maille de Smagorinsky sur le GPU associé pour le *A-step* de la méthode A-A pattern ;
    - ↪ Étape de collision-propagation des fonctions de distribution $f_{\sigma,i}$ sur le GPU associé pour le *A-step* de la méthode A-A pattern ;
  - **fin**
- **fin**
- ↪ Synchronisation des GPUs ;
- **pour** *chaque sous-domaine* **faire**
  - **pour** *chaque composant* σ **faire**
    - ↪ Corrections aux conditions limites et aux obstacles sur le GPU associé pour le *A-step* de la méthode A-A pattern;
    - ↪ Calcul du critère de progression $||C_\alpha||_2$ sur les bords ;
    - ↪ Calcul des quantités macroscopiques ρ_σ et u_σ sur le GPU associé pour le *A-step* de la méthode A-A pattern à l'aide des équations (2.10) et (2.11) ;
  - **fin**
- **fin**

sinon
- ↪ Réalisation des mêmes étapes de calculs que pour les itérations paires mais dans le cas du *B-step* de la méthode A-A pattern ;

fin

7.4.2 Simulations réalisées

Deux simulations sont ici étudiées de manière à évaluer les performances de notre algorithme. Ces simulations incluent toutes deux la présence de deux composants physiques. La géométrie et la taille du domaine diffèrent toutefois pour ces deux simulations. La première est basée sur une géométrie simple composée de $256 \times 256 \times 1024$ cellules de calculs. L'idée est de reprendre une simulation proche de celle effectuée dans la section 6.3.2 de manière à pouvoir comparer les performances entre les deux algorithmes. Dans cette simulation, un fluide va progressivement remplir le domaine de simulation (Figure 7.11).

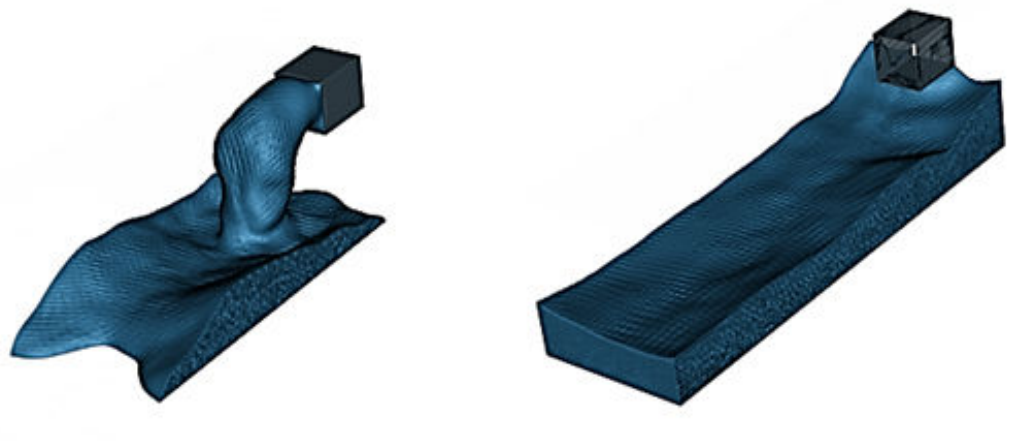


FIGURE 7.11 – Simulation d'un épandage à deux composants à l'aide d'un schéma D3Q19 : le premier composant est présent sous sa forme liquide et gazeuse alors que le second est présent uniquement sous forme gazeuse. Ce cas de figure peut représenter une simulation contenant du GNL sous sa forme liquide et gazeuse en interaction avec l'air présent uniquement sous forme gazeuse.

La seconde simulation inclut quant à elle une géométrie de taille $128 \times 1024 \times 1024$ plus complexe comportant plusieurs canalisations et un déversement dans un bassin à la manière de certaines installations industrielles. Un fluide arrive dans ce cas par deux entrées et vient progressivement se jeter dans le bassin de déversement (Figure 7.12).

7.4.3 Architecture de calculs

L'architecture de calculs utilisée est constituée de huit cartes Tesla C2050 dont les caractéristiques ont déjà été définies dans la section 6.3.3.

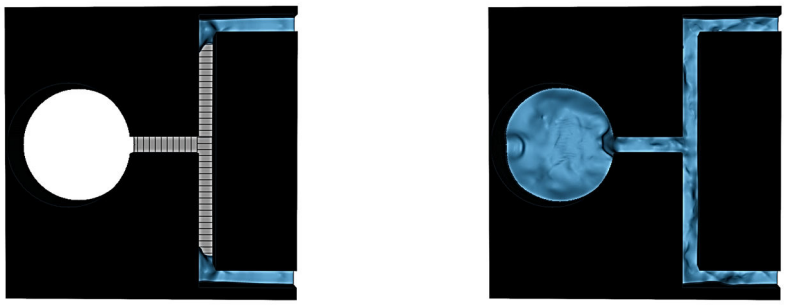


FIGURE 7.12 – Simulation d'un épandage à deux composants dans des canaux de déversement pour un modèle D3Q19 : le premier composant est présent sous sa forme liquide et gazeuse alors que le second est présent uniquement sous forme gazeuse. Ce cas de figure peut représenter une simulation contenant du GNL sous sa forme liquide et gazeuse en interaction avec l'air présent uniquement sous forme gazeuse.

7.4.4 Comparaison des performances avec l'approche statique

Cette section détaille une comparaison des performances obtenues entre la méthode de maillage progressif et la méthode de maillage statique. Le but est de montrer l'impact de la méthode de maillage progressif au sein d'une architecture composée de plusieurs processeurs graphiques. Nous considérons ici que des sous-domaines de taille $128 \times 128 \times 128$ sont utilisés pour nos simulations. Dans un premier temps nous présentons les résultats obtenus pour la simulation présentée dans la figure 7.11.

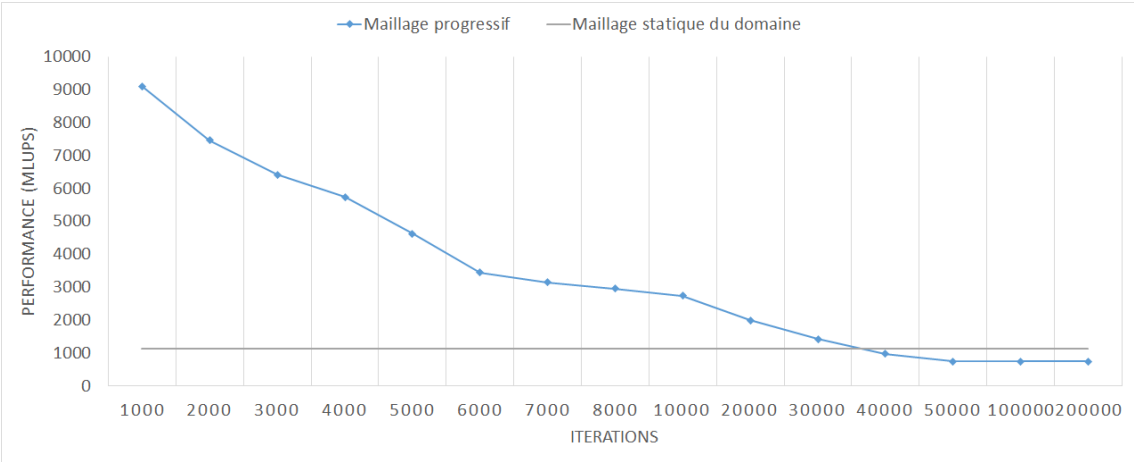


FIGURE 7.13 – Comparaison de performances entre la méthode de maillage progressif et la méthode de maillage statique pour la simulation issue de la figure 7.11.

Nous pouvons en premier lieu constater que les performances obtenues en début de simulation sont très importantes ($\approx$ 9100 MLUPS). En effet, la présence d'une petite

quantité de sous-domaines implique moins de calculs à réaliser et par conséquent de meilleures performances que dans le cas statique. L'augmentation progressive des sous-domaines pendant la simulation a pour conséquence directe une diminution progressive des performances. Cette diminution a lieu jusqu'à atteindre un état de convergence où un ensemble de sous-domaines est créé et qu'aucun autre ne vient s'y ajouter. Dans cette situation particulière, l'ensemble du domaine de simulation finit très vite par être complètement maillé, comme le montre l'évolution de la quantité mémoire dans la figure 7.14, et cela conduit par conséquent à une performance moins intéressante que dans le cas statique ($\approx$ 730 MLUPS) . En effet, l'utilisation du maillage progressif demande dans ce cas plus d'échanges de données entre les GPUs ce qui conduit à une baisse de performances non négligeable (ici de l'ordre de 30% pour ce cas particulier). Cette première simulation représente le cas le plus défavorable pour le maillage dynamique mais n'est pas représentative de la grande majorité des simulations d'épandage dans un complexe industriel qui sont généralement localisées.

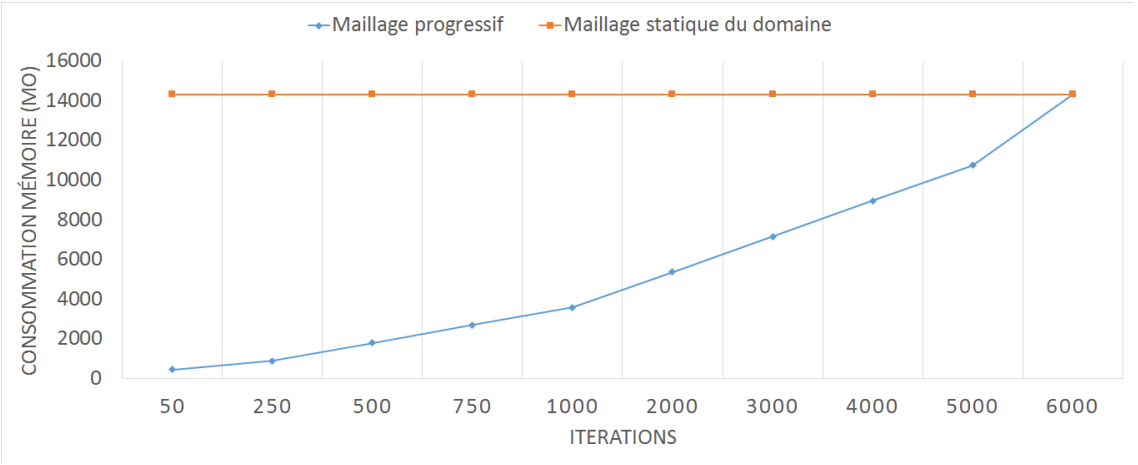


FIGURE 7.14 – Comparaison de la consommation mémoire entre la méthode de maillage progressif et la méthode de maillage statique pour la simulation issue de la figure 7.11.

Les mêmes comparaisons sont maintenant réalisées sur la simulation issue de la figure 7.12. La différence principale dans ce cas est la complexité de la géométrie qui est beaucoup plus canalisée à la manière des infrastructures industrielles. Dans ce cas, la méthode de maillage progressif offre des performances beaucoup plus intéressantes (Figure 7.15). Elle permet d'obtenir des performances très importantes avec une performance moyenne de 4569 MLUPS. Elles dépassent de loin les performances obtenues généralement dans

le cas statique. Sur le plan de la consommation mémoire (Figure 7.16), l'utilisation de la méthode de maillage progressif a permis de simuler un écoulement sur un domaine de simulation normalement composé de $1024 \times 1024 \times 128$ cellules, alors que la méthode de maillage statique ne le peut pas. En effet, la taille de la grille de simulation est trop importante pour tenir sur la mémoire des huit GPUs. On note ainsi un gain de l'ordre de 48% pour la consommation mémoire pour cette simulation par la seule utilisation d'un maillage progressif. Ce gain est principalement dû au fait que seules les zones réellement nécessaires au bon fonctionnement de la simulation sont considérées.

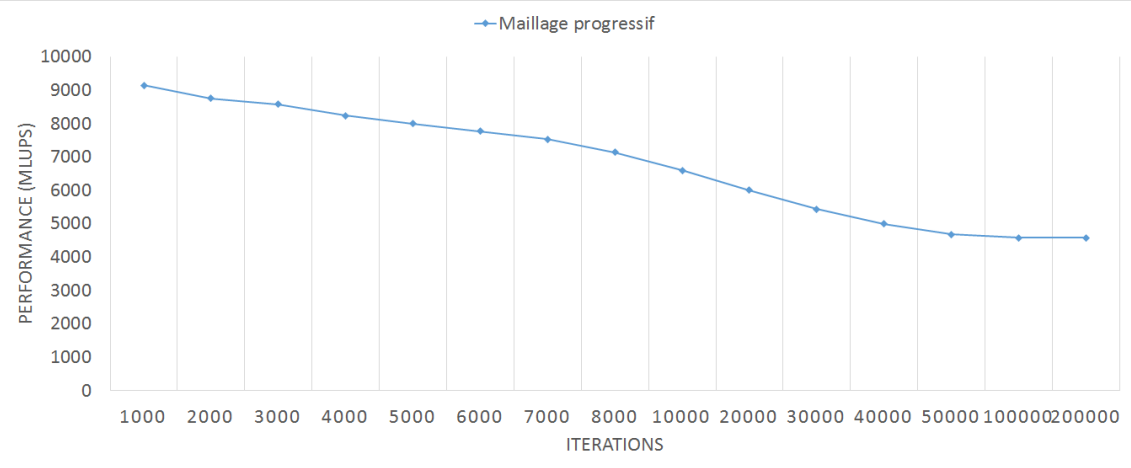


FIGURE 7.15 – Évolution des performances de la méthode de maillage progressif pour la simulation issue de la figure 7.12 : le cas statique n'est pas décrit car la quantité de mémoire est insuffisante pour effectuer la simulation.

7.4.5 Influence de l'amélioration des communications

Le but de cette section est de réaliser une comparaison entre deux méthodes d'affectation des GPUs. La première est une affectation simple qui assigne à un nouveau sous-domaine le premier GPU disponible (dans l'ordre de leur numérotation). La seconde fait appel à l'optimisation de la répartition des processeurs en vue de maximiser les performances des échanges de données présentées dans la section 7.3.2.

La figure 7.17 décrit les performances obtenues pour la simulation de la figure 7.11.

La comparaison entre ces deux méthodes conduit à une différence importante des performances de l'ordre de 26% au maximum dans ce cas. Cette différence est due au fait que l'utilisation d'une simple optimisation permet de réduire dynamiquement les échanges de

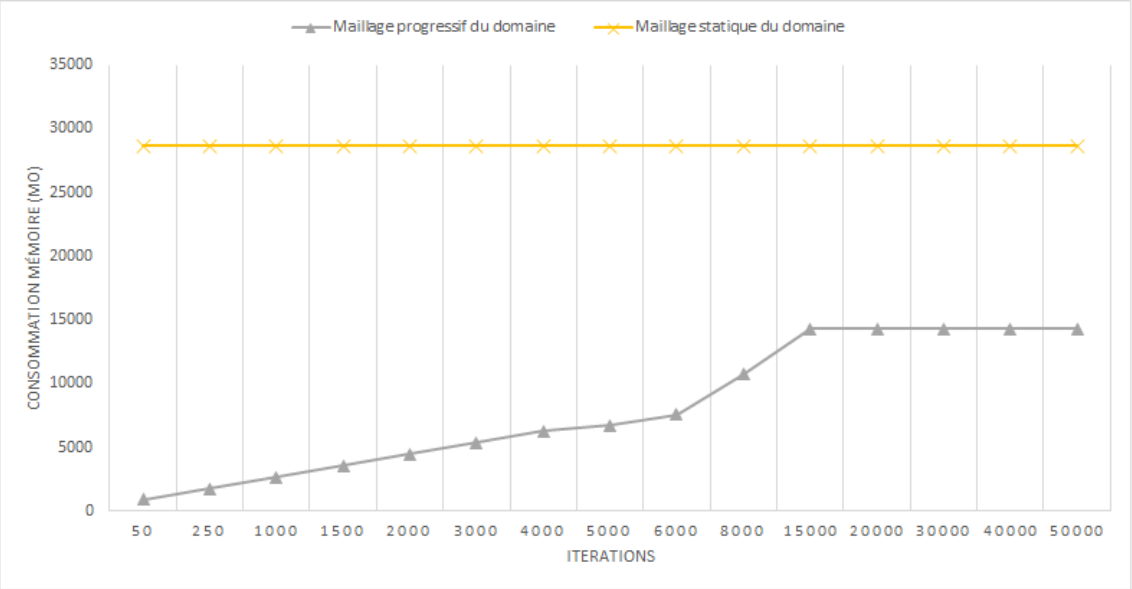


FIGURE 7.16 – Comparaison de la consommation mémoire entre la méthode de maillage progressif et la méthode de maillage statique pour la simulation issue de la figure 7.12.

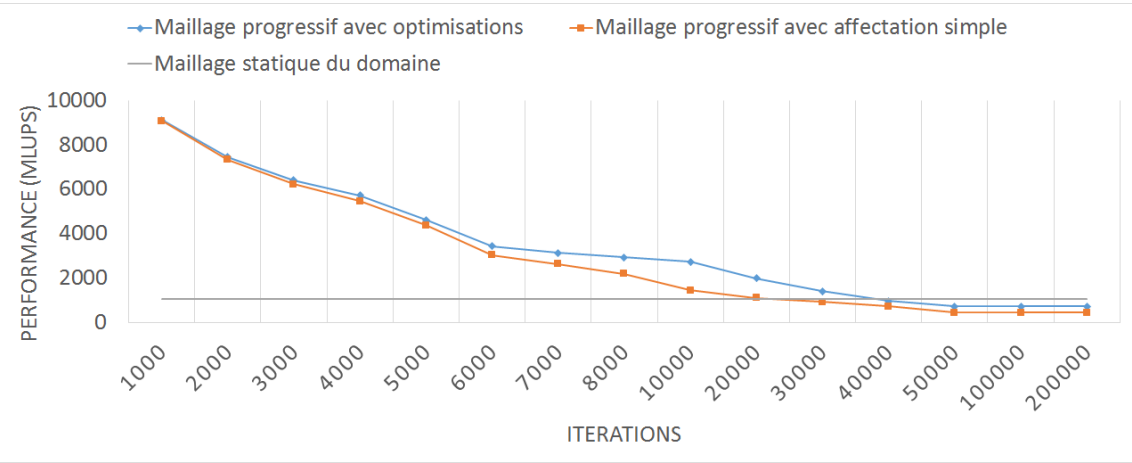


FIGURE 7.17 – Comparaison de performances pour la simulation 7.11 entre une affectation simple des processeurs graphiques avec une affectation améliorée réduisant les communications entre les processeurs.

données et de favoriser des moyens de communications plus rapides (Peer-to-Peer). Cette simple optimisation a pour impact d'affecter au mieux les GPUs de manière dynamique et permet ainsi une réduction du coût de communication.

La comparaison de la répartition des GPUs est également décrite dans le cas de la simulation 7.12 dans la figure 7.18. Une différence de 13% est obtenue entre les deux types de répartition des GPUs. Le gain obtenu est par conséquent inférieur à celui obtenu pour notre première simulation. Ce gain est en réalité variable selon la nature de la simulation

et de la géométrie composant le domaine de simulation. En effet, le gain obtenu est dépendant du nombre de sous-domaines qui seront créés tout au long de la simulation. La création des sous-domaines étant directement reliée à la géométrie composant le domaine de simulation et la simulation en elle-même, le gain obtenu sera inévitablement dépendant de tous ces facteurs.

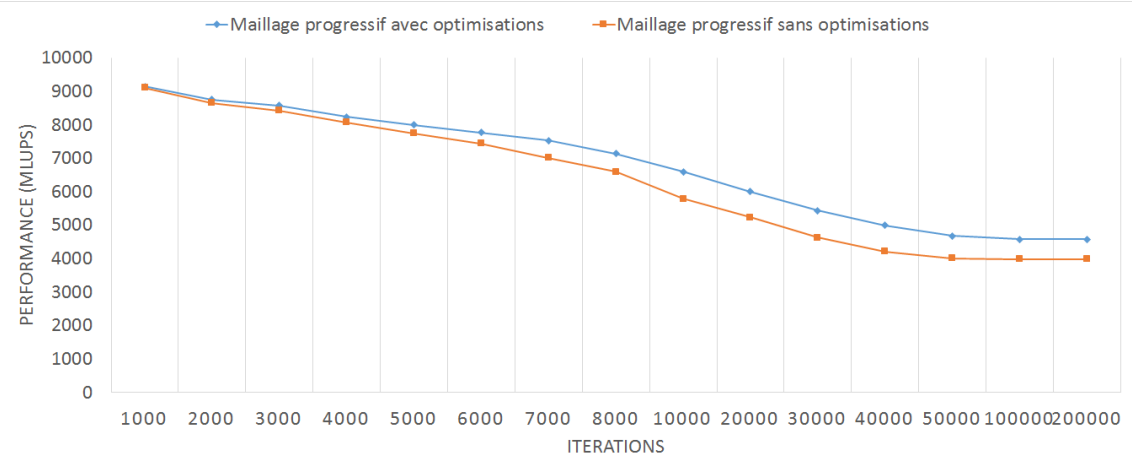


FIGURE 7.18 – Comparaison de performances pour la simulation 7.12 entre une affectation simple des processeurs graphiques avec une affectation améliorée réduisant les communications entre les processeurs.

7.4.6 Influence de la taille du sous-domaine

La méthode de maillage progressif fonctionne sur l'ajout dynamique de sous-domaines en fonction de la propagation du ou des fluides. La taille des sous-domaines est à l'heure actuelle arbitraire et est fixée par l'utilisateur. Le but de cette section est de faire une comparaison des performances en considérant plusieurs tailles de sous-domaine (32^3, 64^3, ou 128^3). La figure 7.19 décrit les performances obtenues pour la simulation 7.11 pour ces tailles de sous-domaine. Elle permet de mettre en évidence que l'utilisation d'une taille de sous-domaine plus petite offre de meilleures performances en début de simulation. En effet, un seul sous-domaine est initialisé sur un GPU en début de simulation, puis de nouveaux sous-domaines sont créés petit à petit. De ce fait, plus les sous-domaines sont de petite taille, plus ils sont créés et distribués rapidement entre les GPUs en début de simulation, améliorant dans un premier temps l'équilibrage de charge par rapport aux simulations utilisant des sous-domaines de plus grande taille.

En revanche, la progression de la simulation réduit de manière conséquente les performances obtenues pour des petits sous-domaines. En effet, la progression de la simulation implique une augmentation importante du nombre de sous-domaines. Pour des sous-domaines de petite taille (32^3 ou 64^3), ce nombre grandit de manière plus importante ce qui a pour impact d'augmenter la demande en communications entre les sous-domaines. Cette augmentation importante de communication implique alors une diminution importante des performances. Le choix d'une taille de sous-domaine suffisamment grande est alors un aspect important pour les performances.

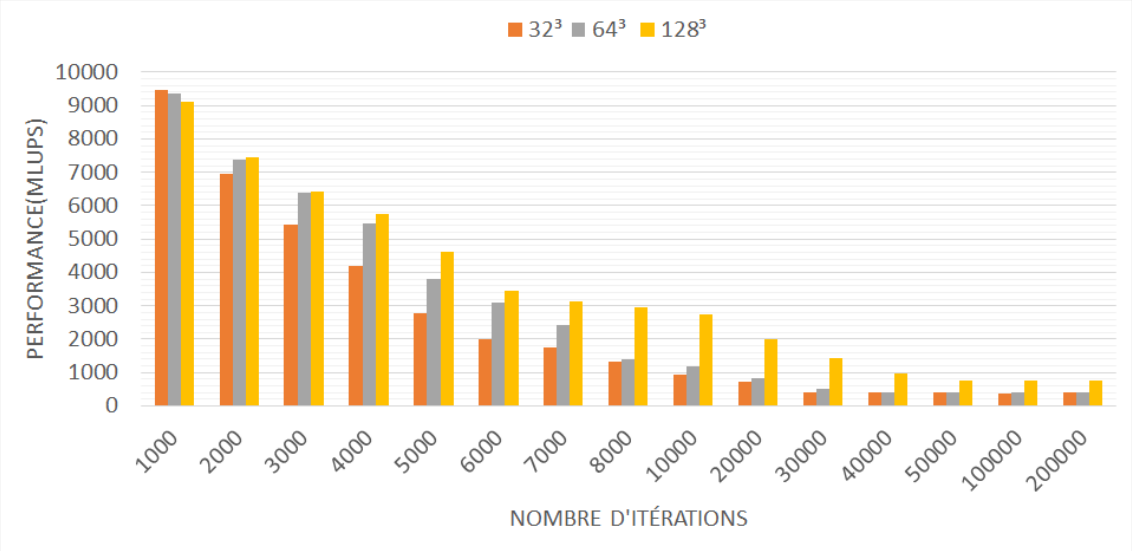


FIGURE 7.19 – Comparaison de performances pour la simulation 7.11 pour différentes tailles de sous-domaine.

Dans notre cas, une taille de 128^3 offre le meilleur compromis en ce qui concerne les performances parmi les tests effectués. L'utilisation de sous-domaines de taille supérieure a également été considérée mais implique alors une augmentation de la consommation mémoire par sous-domaine et un encadrement moins précis de la zone réelle de simulation par le maillage progressif. Ainsi, une taille trop grande offre un nombre de sous-domaines très réduit limitant la progression dynamique.

7.5 Conclusion

Dans ce chapitre, nous avons introduit une méthode de maillage progressif utilisée pour notre noyau de simulation et qui s'intègre à plusieurs processeurs graphiques cal-

culant en parallèle. Cette méthode peut s'adapter à de nombreux cas de simulations, ce qui en fait son premier avantage. La création dynamique des sous-domaines par l'utilisation d'un critère adapté est également un avantage permettant d'obtenir des gains plus ou moins importants en performances et en consommation mémoire selon les cas. Elle est particulièrement efficace pour des simulations fortement canalisées et notamment sur des infrastructures industrielles (écoulement dans des canaux de déversement, écoulement dans des pipelines, ...). Le maillage étant dynamique, la méthode s'adapte automatiquement en cas de débordement des canaux, fuite d'un pipeline, etc., contrairement aux maillages statiques qui doivent prévoir a priori une zone de simulation suffisamment grande au prix d'un coût mémoire et d'un temps de simulation beaucoup plus important.

Des améliorations doivent toutefois être prises en compte à la suite de ces travaux. Une première concerne la taille du sous-domaine qui est fixée à l'initialisation de la simulation. Dans notre cas, on la fixe généralement de manière cubique pour ne favoriser aucune direction de propagation du fluide. Travailler avec une taille fixe empêche toutefois de s'adapter parfaitement à la géométrie et cela peut alors provoquer la présence de sous-domaines qui ont uniquement une petite quantité de calculs utiles à réaliser. Dynamiser la taille des sous-domaines permettrait alors de s'adapter plus facilement à la géométrie du domaine et ainsi de gagner en efficacité. Travailler avec des sous-domaines de taille différentes impose alors une problématique supplémentaire de répartition équilibrée de la charge de calculs sur les GPUs. La répartition progressive des calculs sur les GPUs est également un élément discutable et améliorable. Maintenir de manière dynamique un équilibrage de charge permanent entre les différents processeurs tout en minimisant les échanges de données est une problématique complexe nécessitant encore des recherches approfondies.

La méthode de maillage progressive est également limitée par l'ensemble de la mémoire composant nos processeurs graphiques. La quantité de mémoire dans ce cas est généralement beaucoup moins importante que la RAM du processeur central. Cela implique par conséquent que le nombre de sous-domaines utilisables pour une simulation est dès le départ limité. Pour atteindre un nombre de sous-domaines plus important, la mise en place de méthodes dites « out-of-core » permettant des échanges de données efficaces avec la RAM du processeur central doit être considérée.

Chapitre 8

Mise en place d'une méthode « out-of-core » du noyau de simulation

Sommaire

8.1 Introduction

L'utilisation de processeurs graphiques est un réel avantage pour l'accélération des calculs de simulation. Ils souffrent toutefois d'un inconvénient majeur qui est la quantité de mémoire à leur disposition. En effet, dans notre cas, les cartes Tesla C2050 disposent chacune de 3 Gio de RAM, ce qui nous offre une quantité globale d'environ 24 Gio en tenant compte des huit processeurs à notre disposition. Une telle quantité permet déjà de simuler des domaines de simulation conséquents, mais cela reste insuffisant. En effet, en considérant un pas de discrétisation spatial de 0,5 cm, nous sommes capables de reproduire des écoulements sur quelques dizaines de mètres. Or, en considérant une installation industrielle telle qu'un terminal méthanier, les écoulements peuvent atteindre plusieurs centaines de mètres. La quantité de mémoire que nous offrent les processeurs graphiques est alors insuffisante.

La quantité de mémoire qu'offre le processeur central est généralement beaucoup plus importante que la mémoire des processeurs graphiques. L'objectif de ce chapitre est alors d'exploiter cette ressource afin d'utiliser l'ensemble des capacités qu'offre notre architecture de calculs. Pour cela, la mise en place d'un système d'échanges efficaces entre les processeurs graphiques et la mémoire centrale est indispensable.

Ce chapitre propose d'apporter une solution permettant de réaliser des simulations sur un domaine de simulation nécessitant une quantité de mémoire très importante par la mise en place d'une méthode d'échanges de données entre la mémoire centrale et les processeurs graphiques, qu'on appelle communément méthode « out-of-core ». Pour cela, nous proposons dans un premier temps une approche « naïve » permettant de réaliser ces simulations avec l'utilisation d'un maillage statique du domaine de calculs. Une deuxième approche plus efficace est ensuite introduite. L'extension de cette méthode en combinaison à la méthode de maillage progressif est ensuite discutée et fera l'objet de travaux à l'issue de ce travail de thèse.

8.2 Travaux existants

Les algorithmes à mémoire externe [Vit01], généralement appelés algorithmes « out-of-core », sont liés à la structure hiérarchique de la structure mémoire des ordinateurs modernes. La gestion et l'utilisation au mieux de la mémoire sont importants lorsque l'on traite de gros volumes de données qui ne peuvent pas tous tenir au sein de la mémoire accédée directement et rapidement pour les calculs. Cette section traite de quelques travaux existants en littérature concernant les méthodes « out-of-core » de manière à s'en inspirer pour notre méthode.

L'utilisation de ce type de méthode est majoritairement liée au domaine de l'informatique graphique, où la quantité de données à manipuler pour réaliser le rendu d'un objet ou pour visualiser des grosses scènes peut être extrêmement importante. Lindstrom dans [LP02] propose par exemple une méthode out-of-core permettant de faire de la visualisation de terrains (Figure 8.1).

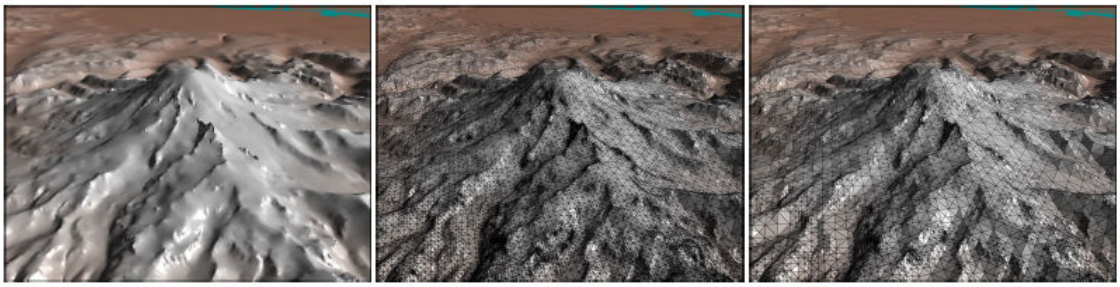


FIGURE 8.1 – Illustration d'une visualisation de terrain issue de [LP02].

Son objectif était de mettre en place une méthode permettant de visualiser rapidement les données issues du maillage du terrain. Ces données pouvaient représenter plusieurs Gio ce qui était à l'époque intenable sur un processeur central usuel. Son idée était alors d'exploiter la capacité du disque dur et d'y réaliser des accès cohérents en vue de visualiser le plus rapidement possible les données de terrain. Son idée était d'adopter une organisation des données adaptée qui permettait de charger des données proches spatialement pour réaliser les calculs de rendu. Pour cela, une classification des données est réalisée de manière à regrouper les données proches spatialement. De cette façon, il pouvait charger progressivement ses données, calculer son rendu et ensuite recharger les données suivantes.

De la même façon, Isenburg dans [IG03] décrit un travail permettant la mise en place d'une méthode de compression pour faire du rendu « out-of-core » d'objets. Son but était de mettre en place une méthode de compression-décompression qui convertit les données de maillage représentant un objet en un maillage fortement compressé. L'utilisation de la compression permet d'obtenir des échanges de données plus rapides entre la mémoire centrale et le disque dur ce qui permet d'accélérer le rendu. La décompression se fait finalement de manière progressive et structurée de manière à reconstruire efficacement l'objet considéré (Figure 8.2).

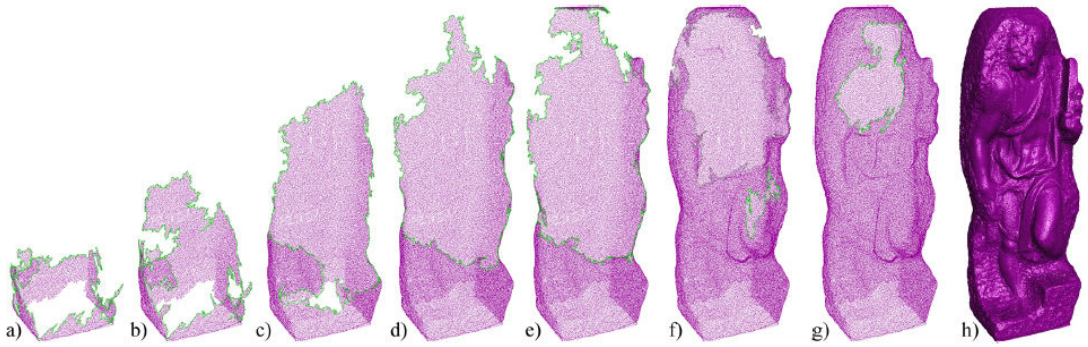


FIGURE 8.2 – Processus de décompression progressif pour un objet et rendu « out-of-core » de l'objet issu de [IG03].

L'utilisation de ces méthodes ne se limite toutefois pas seulement à l'informatique graphique. Elles sont également utilisées également en algèbre linéaire [Tol99] pour réaliser par exemple de la transposition de grosses matrices [SP02] ou encore de la factorisation matricielle [DD00].

La plupart des références présentées traitent d'échanges entre la mémoire centrale et le disque dur. Dans notre cas, c'est un peu différent car nous cherchons à réaliser ces échanges entre la mémoire centrale et les processeurs graphiques [1]. Certaines idées doivent toutefois être exploitées dans notre méthode. L'organisation des données est dans un premier temps un facteur important qui doit être pris en compte pour réaliser des échanges efficaces. L'utilisation d'une méthode de compression-décompression est également une idée intéressante qui pourrait être exploitée de manière à réduire la quantité de données à échanger entre les processeurs graphiques et la mémoire centrale.

1. Nous supposons que les données correspondant au maillage peuvent être stockées dans la mémoire centrale. Si ce n'est pas le cas, il serait alors nécessaire d'envisager un niveau supplémentaire d'échanges entre celle-ci et un espace sur disque.

8.3 Définition d'une méthode « out-of-core » dans un cadre de simulations à maillage statique

8.3.1 Introduction

Cette section traite de la mise en place d'une méthode d'échanges de données entre les processeurs graphiques et la mémoire centrale en vue d'exploiter le maximum des ressources à notre disposition. Dans le cadre de ce manuscrit de thèse, la méthode est uniquement décrite dans un cadre de simulations faisant appel à un maillage statique du domaine de simulation. L'extension à l'utilisation de la méthode de maillage progressif est discutée et fera l'objet de travaux à l'issue de ce travail de thèse. Le but est dans un premier temps de décrire une première approche « naïve » permettant de réaliser ces échanges. Des améliorations sont ensuite apportées de manière à obtenir une méthode plus performante. Cela passe dans un premier temps par une réduction de la quantité de données à échanger. De plus, une approche permettant de réduire le nombre de transferts à réaliser est étudiée. Une organisation adaptée des données à échanger est finalement décrite afin d'améliorer l'efficacité de la méthode.

8.3.2 Approche naïve

Cette section décrit une première méthode permettant de réaliser des échanges de données en vue de réaliser des simulations sur de grands domaines de calculs. Pour cela, considérons un domaine de simulation décomposé en un ensemble fini de sous-domaines (Figure 8.3). Nous faisons de plus l'hypothèse que les processeurs graphiques ne peuvent contenir en mémoire qu'un nombre limité de sous-domaines inférieur au nombre total de sous-domaines. Dans ce cas, à l'initialisation de la simulation, une partie des sous-domaines sont présents sur les processeurs graphiques alors que la totalité est présente sur la mémoire centrale.

Le noyau de simulation étant basé sur la méthode de Boltzmann sur réseau, les sous-domaines sont amenés à échanger des données sur leurs bords au cours de chaque itération (section 6.2.2). Une première problématique apparaît alors lorsque les sous-domaines sont présents au sein de différentes mémoires. En effet, les échanges entre des sous-

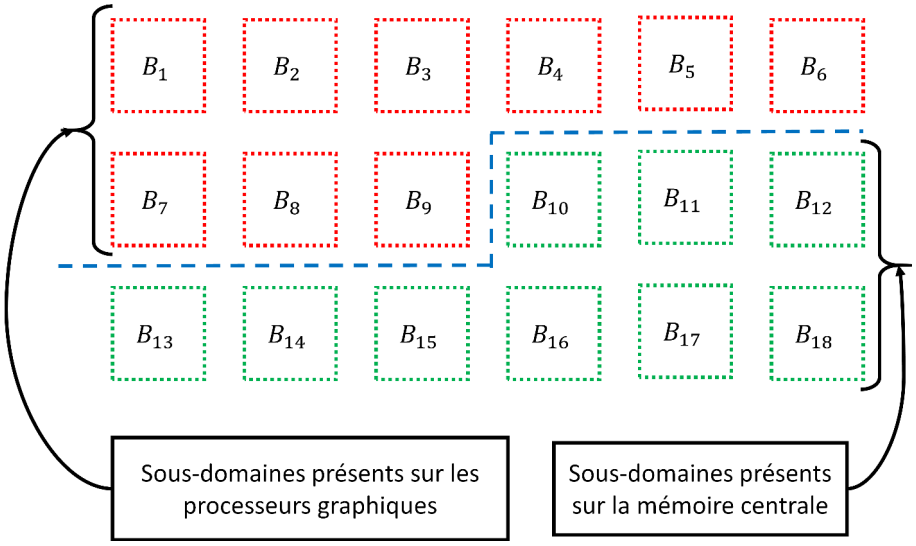


FIGURE 8.3 – Exemple illustrant un cas où le nombre de sous-domaine est trop important pour entrer dans la mémoire des processeurs graphiques : à un instant donné, seule une partie des sous-domaines est dans la mémoire des GPUs.

domaines présents sur les processeurs graphiques peuvent être effectués de manière normale, comme expliqué dans les chapitres précédents. En revanche, les échanges avec les sous-domaines présents sur la mémoire centrale ne peuvent être réalisés car il est impossible de les calculer sur les GPUs à cet instant.

Notre première approche consiste à séparer en premier lieu les calculs en plusieurs « niveaux ». Chaque niveau est composé d'un ensemble de sous-domaines sur lesquels les calculs vont s'effectuer au même instant (Figure 8.4).

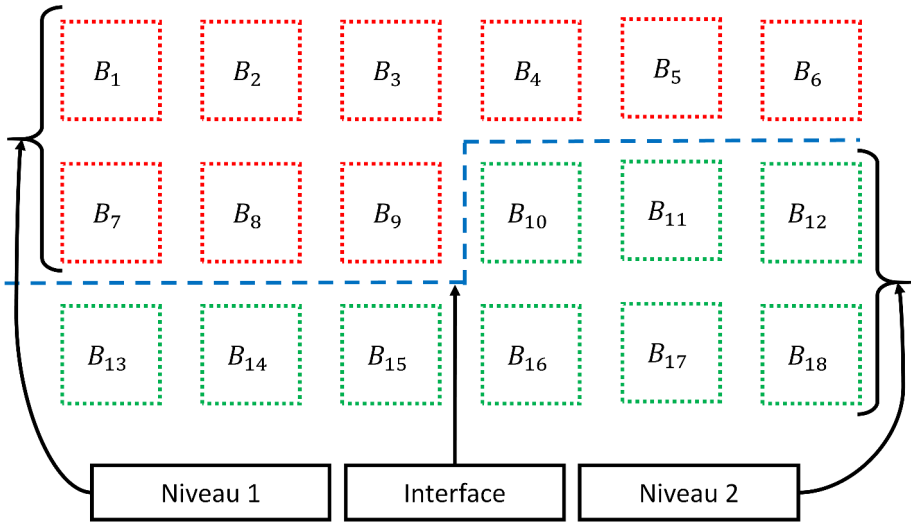


FIGURE 8.4 – Exemple illustrant la séparation des sous-domaines en plusieurs niveaux de calculs : deux niveaux sont présents et une interface apparaît entre les deux niveaux.

Le but est par conséquent d'alterner les calculs entre les différents niveaux et ainsi de charger-décharger la mémoire des sous-domaines de ces niveaux entre les processeurs graphiques et la mémoire centrale pour chaque itération. Une interface apparaît alors entre les niveaux ce qui peut être problématique pour le bon fonctionnement de la simulation. Il est donc indispensable de considérer en premier lieu les calculs au bord de cette interface de manière à assurer les bonnes communications entre les sous-domaines. De cette façon, chaque niveau peut effectuer une itération sans perte d'information et la simulation est en permanence correcte (Figure 8.5). En réalité, l'intégralité des bords situés à l'interface est conservé en permanence à l'intérieur de la mémoire des GPUs de manière à pouvoir aisément les calculer.

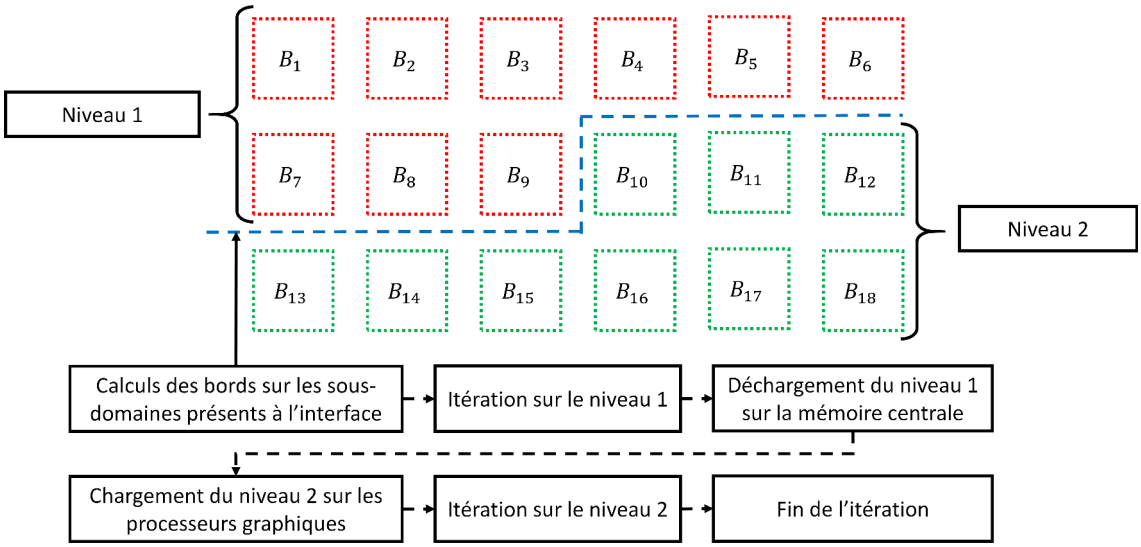


FIGURE 8.5 – Exemple illustrant une itération pour une simulation composée de deux niveaux : les calculs sont dans un premier temps réalisés sur les bords des sous-domaines à l'interface et les niveaux sont finalement chargés sur les GPUs, calculés puis déchargés sur la RAM.

Cette approche souffre d'un inconvénient majeur. En effet, chaque niveau doit être chargé puis déchargé des processeurs graphiques à chaque itération. Cela représente une très grande quantité de données à échanger entre les processeurs graphiques et la mémoire centrale. Une telle méthode ne peut par conséquent pas être performante car les performances de calculs des processeurs graphiques vont être complètement noyées par le temps de communication des données. Même si cette méthode semble inappropriée, cela permet de mettre en évidence ses faiblesses. Elle peut être sujette à des améliorations en vue d'obtenir une méthode plus efficace. En effet, plusieurs aspects sont à prendre en

compte pour les améliorations. La première consiste à réduire la quantité de données à transmettre à chaque échange. La seconde, et probablement la plus importante, consiste à réduire de manière importante le nombre d'échanges à réaliser entre les processeurs graphiques et la mémoire centrale. De ce fait, en tenant compte de ces deux améliorations, nous pourrions alors obtenir une méthode efficace d'échanges de données.

8.3.3 Réduction de la quantité de données à transférer

Une cellule de calculs est composée d'un ensemble de valeurs définissant de nombreux paramètres physiques pour chaque composant intervenant dans la simulation (densité, vitesse, forces, fonctions de distribution, ...). Le déplacement de l'ensemble des informations d'une cellule entre les processeurs graphiques et la mémoire centrale représente une quantité très importante de données. La réduction de la taille du transfert est par conséquent un premier facteur d'amélioration indispensable.

La méthode de Boltzmann sur réseau est basée sur une modélisation fortement centrée sur l'utilisation des fonctions de distribution f_i. En effet, ils représentent les éléments centraux de la méthode et permettent de recouvrer les quantités macroscopiques des fluides intervenant dans la simulation. Ce facteur reste valable pour notre noyau de simulation. En effet, l'intégralité des valeurs peuvent être retrouvées par la simple conservation de ces fonctions de distribution. L'objectif est par conséquent d'éviter tout échange de données inutile et de ne considérer que le transfert des valeurs absolument nécessaires. L'idée est alors de ne considérer que ces fonctions de distribution f_i pour les échanges de données entre les GPUs et la mémoire centrale. L'ensemble des quantités physiques peuvent ainsi être retrouvées par l'intermédiaire de calculs sur le GPU, ce qui est plus efficace que d'échanger ces données. Une telle approche permet ainsi de réduire la taille du transfert de 32% ce qui est non négligeable.

En plus de cette première réduction, l'ajout d'une méthode de compression sans perte est également à l'étude et sera introduite à l'issue de ces travaux de thèse. L'inclusion d'une telle méthode pourrait encore réduire la taille du transfert. Pour cela, la mise en place d'un algorithme de compression de données à virgule flottante sur le GPU est indispensable. Cet algorithme se doit d'être rapide sur le plan calculatoire et se doit d'offrir de bons ratios de compression de manière à obtenir de bonnes performances. Les travaux

issus de [OB11] pourraient nous conduire à un algorithme rapide de compression. Cet algorithme est basé sur le fait que des données peuvent avoir des valeurs très proches et que le stockage peut alors être réduit en les combinant.

L'idée est ainsi de calculer un résidu entre deux valeurs à compresser et de conserver ce résidu qui peut être stocké sur moins de bits qu'un nombre à virgule flottante. Dans notre cas, l'idée serait de comparer des valeurs proches spatialement, de calculer un résidu par la différence entre ces deux valeurs et de ne conserver que ce résidu dans autant ou moins de bits selon le besoin (Figure 8.6). La compression peut se faire de manière parallèle sur le GPU en réalisant une parallélisation par tranche. L'idée est de débuter la compression pour chaque tranche d'un point de départ qui est conservé, et de calculer successivement les résidus à l'aide des valeurs voisines.

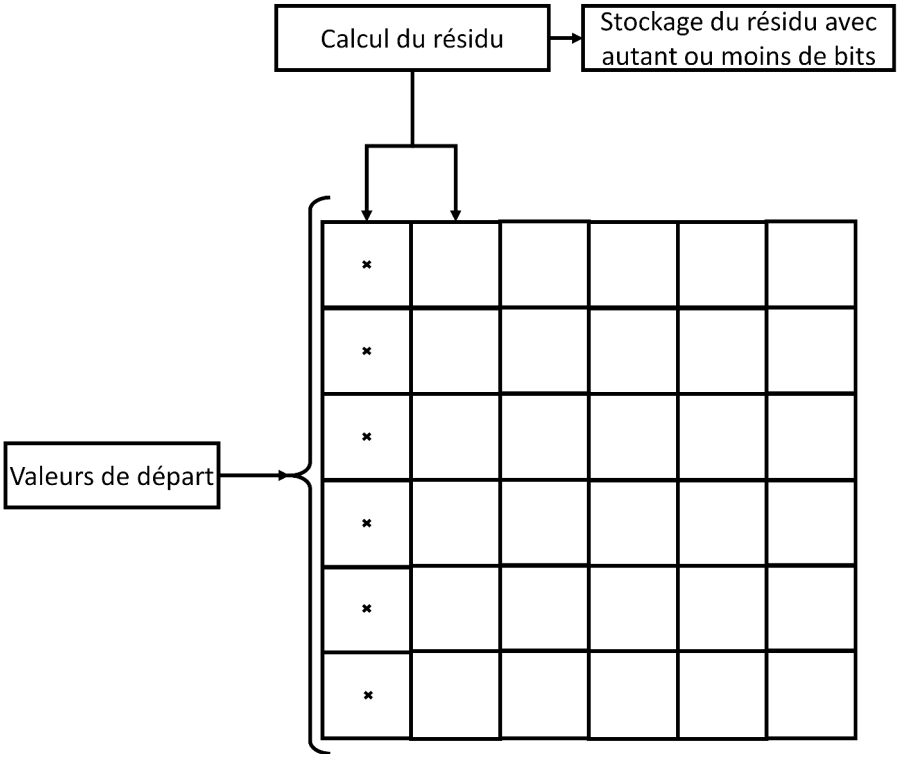


FIGURE 8.6 – Illustration du processus de compression pour des valeurs à virgule flottante : deux valeurs proches sont comparées, un résidu en est déduit et peut être stocké avec moins ou autant de bits que la valeur d'origine.

Le processus de décompression peut quant à lui se faire dans le sens inverse en partant de la valeur de départ et en recouvrant les véritables valeurs à l'aide des résidus (Figure 8.7).

Sur le plan théorique, l'algorithme de compression est bien défini. C'est dans un cadre

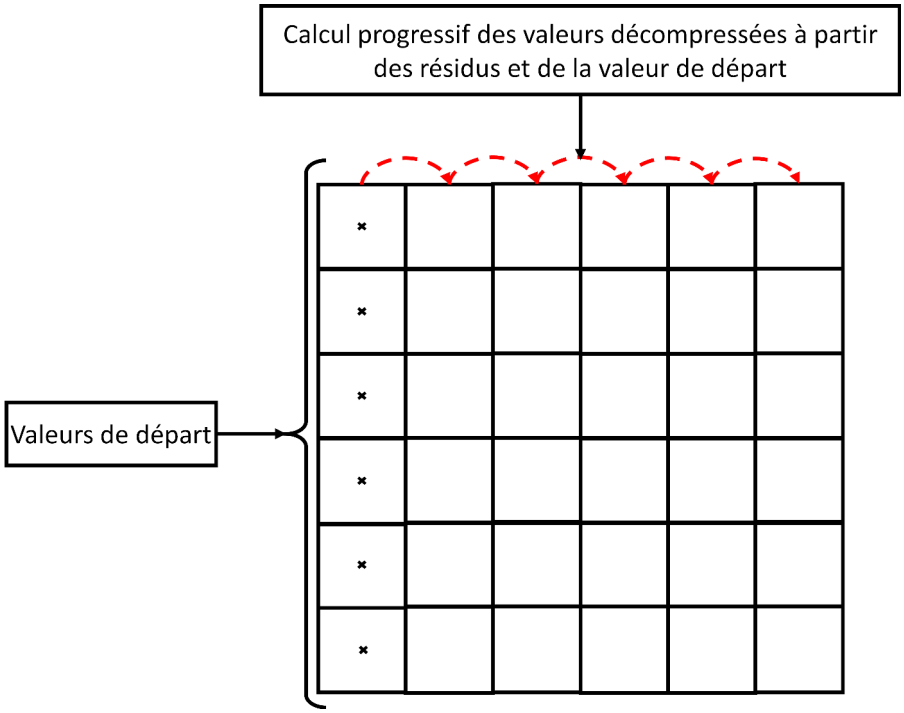


FIGURE 8.7 – Illustration du processus de décompression pour des valeurs à virgule flottante : les valeurs d'origine sont retrouvées progressivement à partir des valeurs de départ et des résidus.

pratique que le problème intervient. La question est surtout de savoir de quelle manière stocker les valeurs compressées en un nombre de bits différents sur le GPU. Ce problème est principalement lié au développement de l'algorithme et sera le sujet d'une étude à l'issue de ce travail de thèse.

8.3.4 Réduction du nombre de transferts par la propagation et la correction d'erreurs

Le nombre important d'échanges de données à réaliser dans notre première approche de la section 8.3.2 est également un facteur pénalisant pour les performances de la simulation. En effet, les performances sont ici totalement dépendantes du temps d'échanges des données qui est beaucoup trop important. Cette approche avait tout de même pour avantage de conserver une simulation correcte à chaque itération en s'assurant de bien réaliser les calculs aux interfaces entre les différents niveaux au préalable.

Nous proposons maintenant une approche totalement différente. L'idée est toujours de regrouper les sous-domaines par niveaux et de réaliser les calculs sur ces niveaux en chargeant et déchargeant successivement les données. Néanmoins, dans cette nouvelle

approche, plusieurs itérations de la méthode de Boltzmann sur réseau sont réalisées sur un même niveau avant de l'échanger avec un autre niveau en mémoire centrale. Le but est alors de fixer un nombre d'itérations appelé n et de réaliser ces itérations sur le niveau se trouvant dans la mémoire des GPUs sans tenir compte des sous-domaines extérieurs. La réalisation de ces itérations va par conséquent avoir pour impact la création d'erreurs sur les sous-domaines présents à l'interface dues à la non prise en compte des informations (fonctions de distribution f_i et pseudo-potentiel ψ) devant arriver à cet endroit (Figure 8.8).

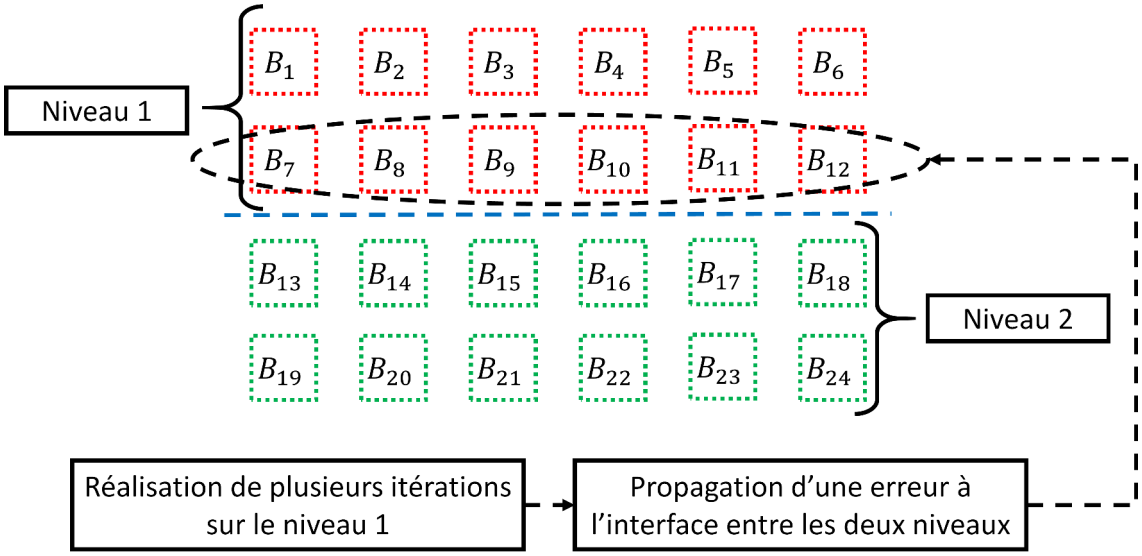


FIGURE 8.8 – Exemple de propagation de l'erreur entre deux niveaux : l'erreur se propage sur les sous-domaines présents à l'interface.

La méthode de Boltzmann sur réseau étant basée sur l'utilisation des plus proches voisins, l'erreur va donc se propager linéairement de cellule en cellule à partir des bords situés à l'interface en fonction du nombre d'itérations effectuées au sein d'un sous-domaine (Figure 8.9). L'idée est alors de réaliser le plus d'itérations possibles afin de propager au plus loin l'erreur sur les sous-domaines situés à l'interface sans affecter les autres. Le but est en fait de restreindre les erreurs propagées à ces sous-domaines situés à l'interface de manière à ce qu'ils soient les seuls à devoir être corrigés par la suite. Plus le nombre d'itération est important, moins le nombre d'échanges entre les processeurs graphiques et la mémoire centrale est important, tout en limitant les erreurs aux sous-domaines situés à l'interface.

La propagation d'une erreur permet en effet de réduire de manière conséquente le

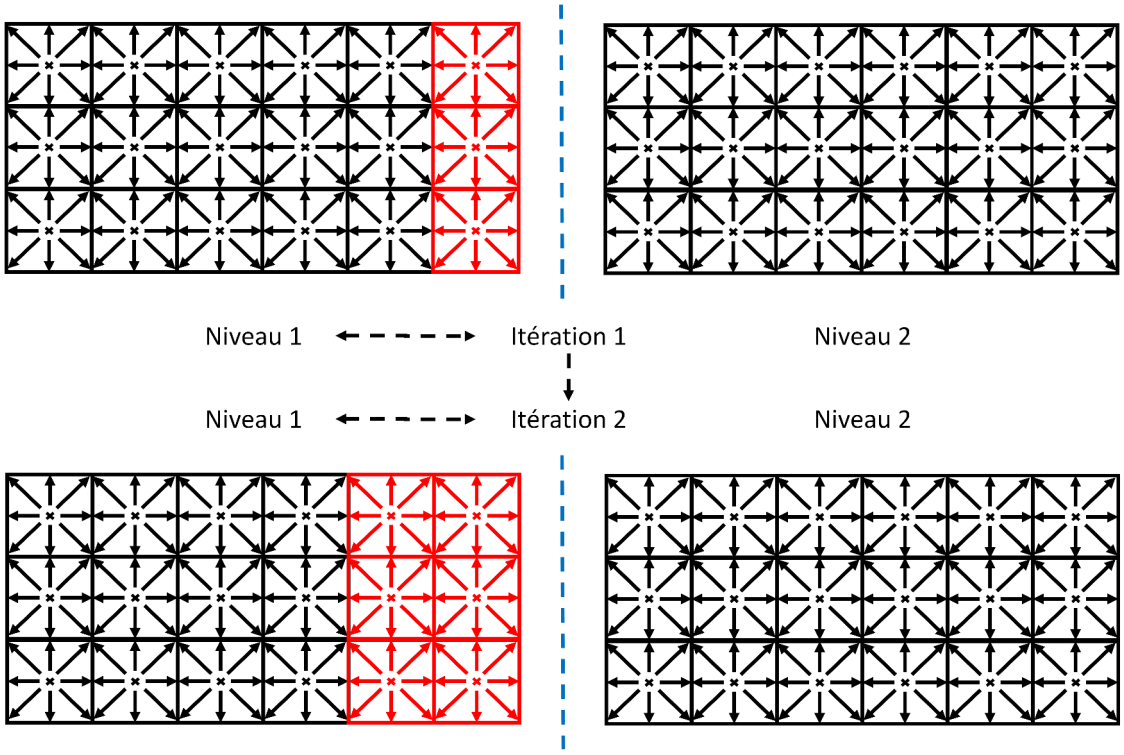


FIGURE 8.9 – Exemple de propagation de l'erreur entre deux niveaux : l'erreur se propage linéairement sur les sous-domaines en fonction du nombre d'itérations effectuées sur un niveau.

nombre d'échanges à effectuer entre les GPUs et la mémoire centrale. Il faut maintenant s'assurer que nous sommes en mesure de pouvoir corriger ces erreurs de manière à conserver une simulation correcte. En réalité, à l'issue de n itérations sur un niveau, une première partie des sous-domaines a été correctement calculée et une seconde partie est constituée de sous-domaines contenant des erreurs. L'idée est alors de décharger sur la mémoire centrale les sous-domaines correctement calculés et de venir remettre les sous-domaines contenant des erreurs à leur état avant ces n itérations à l'aide du processeur central. D'autres sous-domaines viennent finalement s'ajouter de manière à remplir la mémoire des processeurs graphiques. Ainsi, tous les sous-domaines chargés en mémoire sont au même instant sur le plan de la simulation. La correction d'une grille située à l'interface implique toutefois de sauvegarder les informations correctement calculées pour les bords des sous-domaines au voisinage de cette interface. La figure 8.10 résume par un exemple les différentes étapes permettant la bonne continuité de la simulation tout en réduisant le nombre d'échanges.

La méthode que nous avons mis en place permet ainsi de réduire le nombre d'échanges

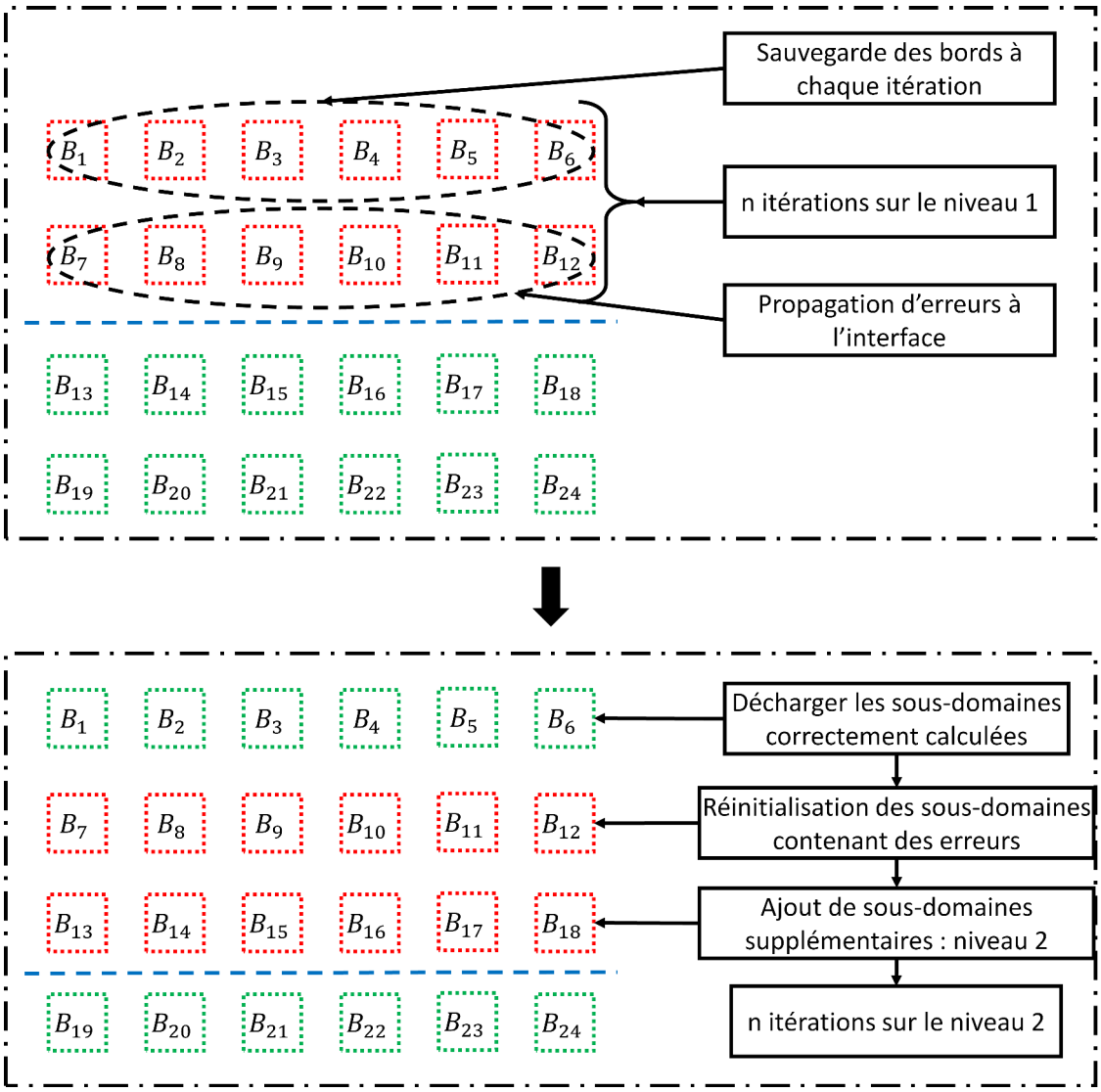


FIGURE 8.10 – Exemple illustrant le principe de notre méthode pour deux niveaux.

à faire entre la mémoire centrale et les processeurs graphiques d'un facteur n par la propagation et la correction d'erreurs. Le fait de réaliser plusieurs itérations sans apport d'informations à l'interface provoque des erreurs qui doivent par la suite être corrigées. C'est pourquoi les sous-domaines présents à l'interface sont remis à leur état d'origine pour être recalculés ensuite tout en ayant tenu compte cette fois d'avoir sauvegardé les informations devant arriver à ces sous-domaines. Les sous-domaines correctement calculés sont quant à eux déchargés sur la mémoire centrale pour être échangés avec d'autres sous-domaines devant être calculés. Le fait de recalculer les sous-domaines à l'interface augmente par conséquent la quantité de calculs à réaliser mais elle devrait être bénéfique sur le plan des performances, car elle permet une diminution du nombre d'échanges à réaliser entre les

processeurs graphiques et la mémoire centrale. L'idée est ainsi de favoriser les calculs aux échanges de données, ce qui devrait permettre un gain important de performances.

8.3.5 Organisation des données

La méthode présentée dans la section précédente ne peut s'avérer efficace que si le nombre de sous-domaines correctement calculé à l'issue de n itérations pour chaque niveau est maximal. Ce nombre est maximal si et seulement si l'interface avec les sous-domaines externes est minimale. En effet, une interface minimale implique une propagation moindre de l'erreur ce qui ne peut être que positif pour les performances. Le but de cette section est de fournir une méthode permettant d'organiser au mieux nos niveaux de calculs de manière à minimiser cette interface. Pour cela, nous allons définir une méthode simple de clustering permettant de regrouper au mieux les sous-domaines en plusieurs niveaux.

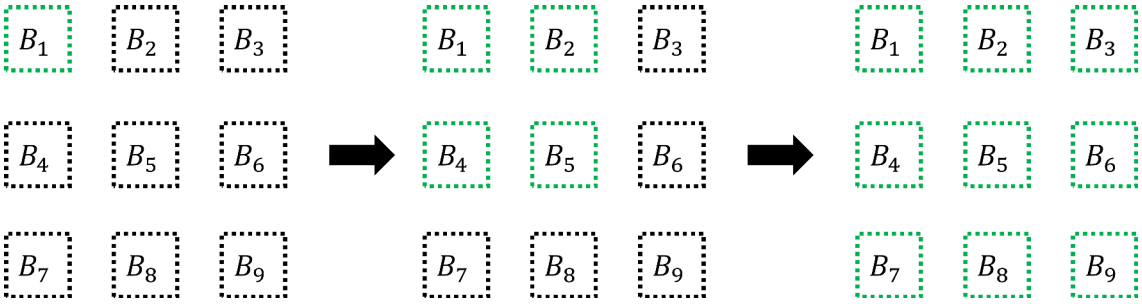


FIGURE 8.11 – Exemple illustrant l'évolution de la création d'un niveau : un point de départ fixe est initialisé et les voisins sont ajoutés successivement jusqu'à atteindre le nombre maximal de sous-domaines pour un niveau.

Le facteur clé pour obtenir un bon regroupement est la proximité spatiale des sous-domaines. L'idée est donc de définir progressivement nos niveaux en ayant pour critère une distance minimale pour des sous-domaines appartenant au même niveau. Un algorithme de type K-means avec une contrainte de taille sur les niveaux pourrait fonctionner mais nous proposons une méthode plus simple. L'idée est simplement d'exploiter la notion de voisinage que nous définissons pour les communications entre les sous-domaines (section 7.2.3). Le but est donc de définir un point de départ à la construction du niveau et d'ajouter l'ensemble des voisins successivement jusqu'à obtenir le nombre de sous-domaines limite (Figure 8.11). Ce point de départ est essentiel ici car c'est lui qui va

guider la construction du niveau de calculs et cela ne garantit pas pour autant une interface minimale. Nous décidons alors de tester toutes les possibilités comme point de départ et de conserver la solution minimisant l'interface.

Une fois le premier niveau défini, la même opération est réalisée pour définir les autres niveaux de calculs en ne considérant que les sous-domaines restants. Les sous-domaines qui sont définis comme interface doivent être réintroduits pour être classés car ils peuvent être calculés plusieurs fois selon l'arrivée ou non des informations. Un sous-domaine est en réalité défini en tant qu'interface tant qu'il a au moins un sous-domaine voisin non chargé qui n'a pas été calculé (Figure 8.12).

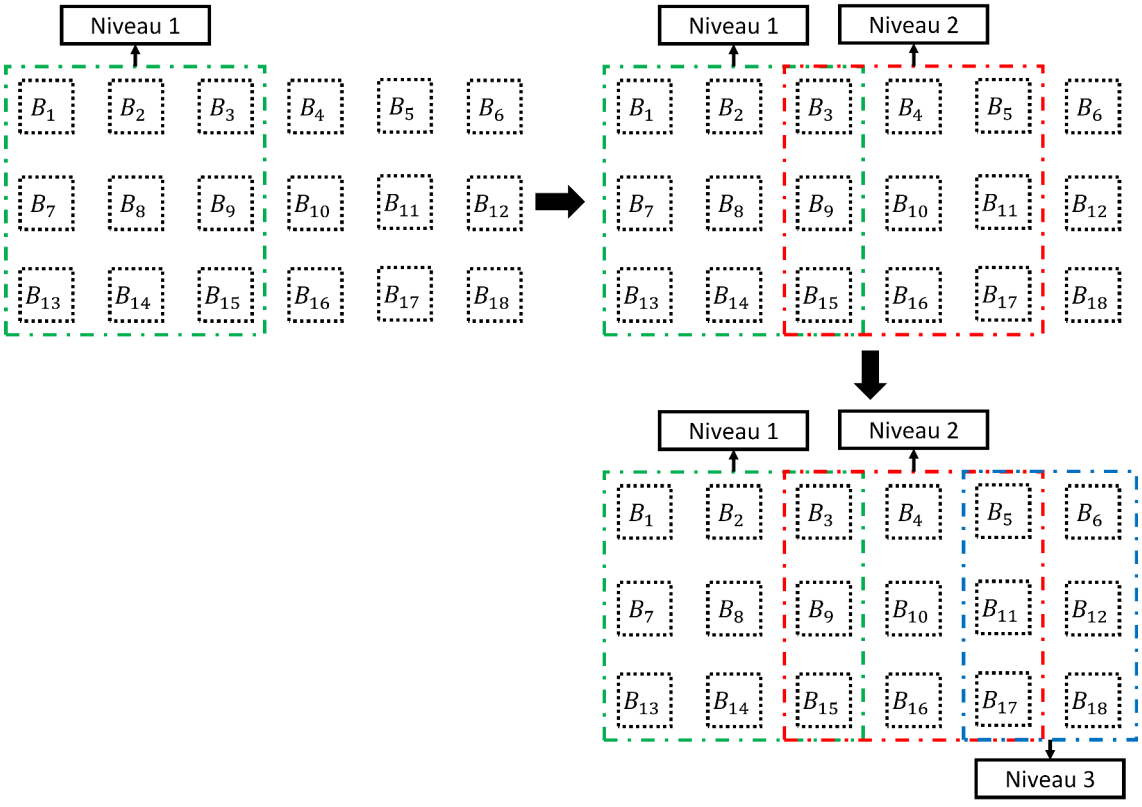


FIGURE 8.12 – Exemple illustrant l'évolution de la création de plusieurs niveaux : les niveaux sont définis progressivement et les interfaces sont classées deux fois de manière à assurer le bon fonctionnement de la méthode définie dans la section précédente.

8.4 Évaluation des performances

8.4.1 Simulation

L'objectif de cette section est de faire une évaluation des performances obtenues par notre méthode d'échange de données pour une simulation faisant appel à un maillage statique du domaine de simulation. Pour cela, nous allons considérer une première simulation d'épandage à l'intérieur d'un domaine simple de taille $128 \times 1024 \times 1024$ cellules. Une fuite se produit sur un des bords et va progressivement se propager à l'intérieur du domaine (Figure 8.13). Nous faisons également l'hypothèse que le domaine de simulation est découpé en sous-domaines de taille 128^3. En sachant que le choix d'une telle taille de sous-domaine nous permet d'avoir au maximum six sous-domaines par GPU (soit quarante-huit sous-domaines sur les huit processeurs graphiques), l'algorithme de classification décrit dans la section 8.3.5 nous donne alors le résultat présenté au sein de la figure 8.14.

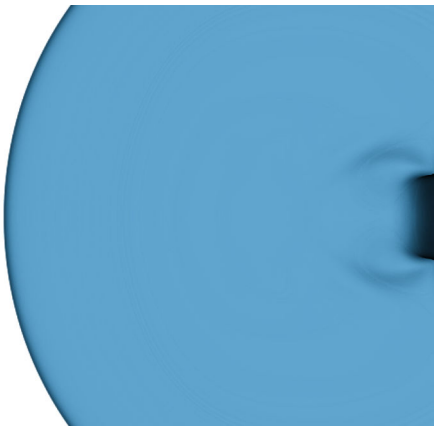


FIGURE 8.13 – Simulation d'écoulement à l'intérieur d'un gros domaine de simulation composé de $128 \times 1024 \times 1024$ cellules : une fuite se produit sur un des bords et se propage à l'intérieur de tout le domaine.

Cette simulation permet de mettre en évidence un exemple simple où seulement deux niveaux de calculs interviennent. Elle a pour but de vérifier en premier lieu l'efficacité de la classification réalisée. Cette classification nous offre en effet une interface de six sous-domaines ce qui permet d'avoir quarante-deux grilles correctement calculées pour le premier niveau, soit une efficacité de 87.5% des calculs pour ce niveau.

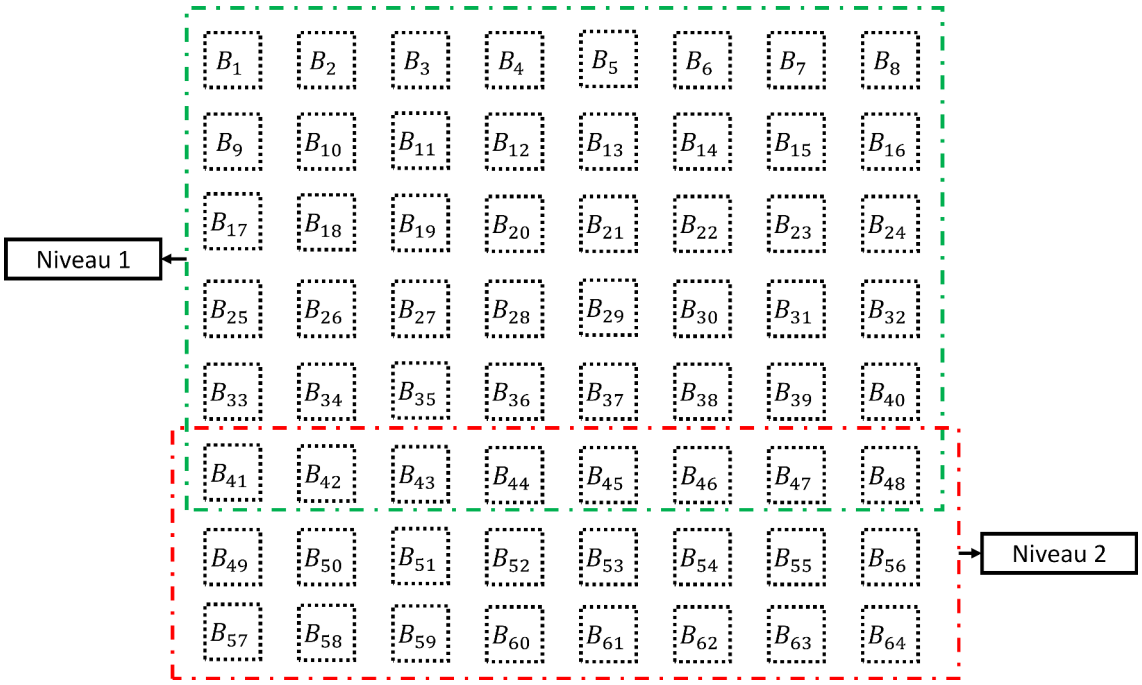


FIGURE 8.14 – Classification des niveaux pour un domaine de taille $128 \times 1024 \times 1024$ cellules décomposé en sous-domaines de taille 128^3.

8.4.2 Comparaison à l'approche naïve

Le but de cette section est de comparer les performances obtenues entre l'approche naïve décrite dans la section 8.3.2 avec notre méthode incluant les améliorations décrites dans les sections 8.3.3 et 8.3.4. Le tableau 8.1 décrit les performances obtenues pour les deux méthodes dans le cas de la simulation décrite ci-dessus. Les performances sont estimées après 100 000 itérations du noyau de simulation.

	Approche naïve	Notre méthode
Performances (MLUPS)	18	451
Temps de transfert (%)	96.2	24.4

TABLEAU 8.1 – Comparaison de performances entre l'approche naïve et notre méthode pour la simulation de la figure 8.13.

Nous pouvons constater dans un premier temps que les performances obtenues par l'approche naïve sont extrêmement faibles. Une performance moyenne de 18 MLUPS représente une perte extrêmement importante de l'efficacité de l'algorithme en comparaison à l'utilisation des processeurs graphiques dans des cas où la mémoire des GPUs est suffisante pour éviter les transferts avec la mémoire centrale (Chapitre 6). En réalité,

cette baisse importante de performances est essentiellement due au fait que les communications entre les processeurs graphiques et la mémoire centrale ont complètement noyé les performances de calculs qu'offrent les GPUs. En effet, le temps de transfert des données représente ici environ 96 % du temps global de la simulation. L'utilisation d'une telle méthode fait ainsi perdre tout intérêt à l'utilisation de GPUs. Puisque l'utilisation d'un processeur central peut offrir des performances quasi similaires à celles obtenues ici (Chapitre 4). Cela vient confirmer le fait qu'une telle méthode ne peut être viable pour obtenir des résultats de simulation performants.

En revanche, notre méthode offre ici des performances bien plus importantes. En effet, pour la même simulation, on obtient une performance moyenne de 451 MLUPS ce qui représente un gain d'un facteur environ 25 en comparaison à l'approche naïve. Nous pouvons constater que ce gain est dû à une importante réduction du temps de transfert entre les processeurs graphiques et la mémoire centrale. Cela vient conforter le fait qu'une telle approche semble plus appropriée pour obtenir des performances de simulation plus importantes. En comparaison avec la puissance de calculs des GPUs sans échange de données, on obtient alors une perte de performance d'un facteur d'environ 2.4 ce qui semble acceptable. L'ajout d'une technique de compression de données telle que présentée dans la section 8.3.3 pourrait encore faire baisser ce facteur.

8.4.3 Influence du nombre d'itérations réalisées par niveau

Le cœur de notre méthode d'échanges de données vient du fait qu'il est possible de réduire la quantité de ces échanges par la propagation d'erreurs en fonction du nombre d'itérations effectuées par niveau. Ce nombre est un facteur clé pour les performances de la méthode. C'est pourquoi nous réalisons une comparaison des performances obtenues en faisant varier ce nombre d'itérations (Figure 8.15).

Nous pouvons noter que l'augmentation du nombre d'itérations a effectivement un impact positif sur les performances. L'idée est en fait de faire propager l'erreur au plus loin sur les sous-domaines situés à l'interface sans affecter les autres. De ce fait, cela permet de maximiser le nombre d'itérations effectuées par niveau tout en conservant un nombre maximal de sous-domaines correctement calculés. L'augmentation du nombre d'itérations permet en réalité de diminuer de manière importante la quantité de transferts

à effectuer entre les processeurs graphiques et la mémoire centrale (Figure 8.16).

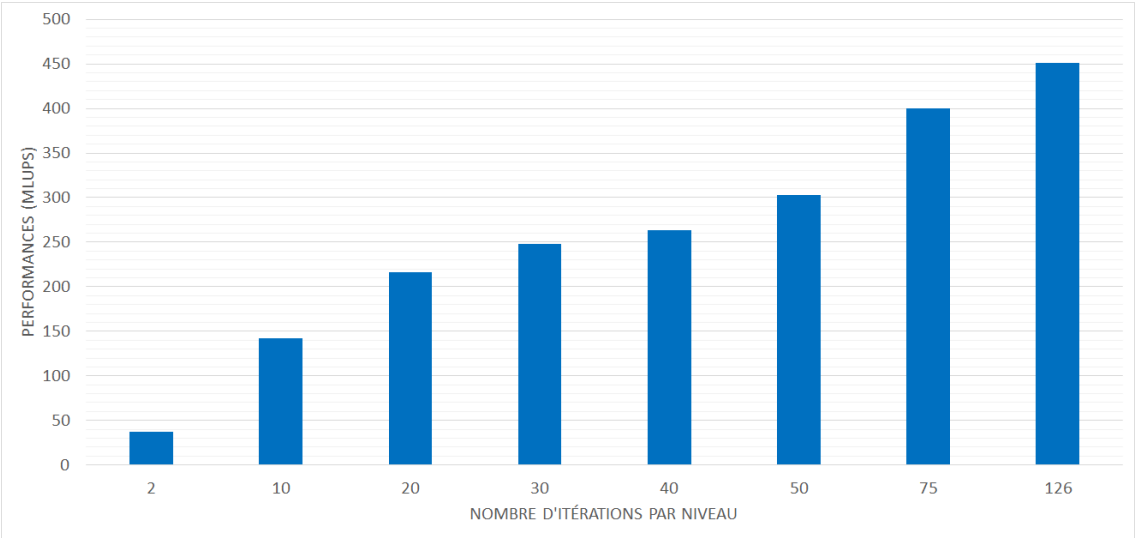


FIGURE 8.15 – Évolution des performances obtenues en faisant varier le nombre d'itérations réalisées pour chaque niveau.

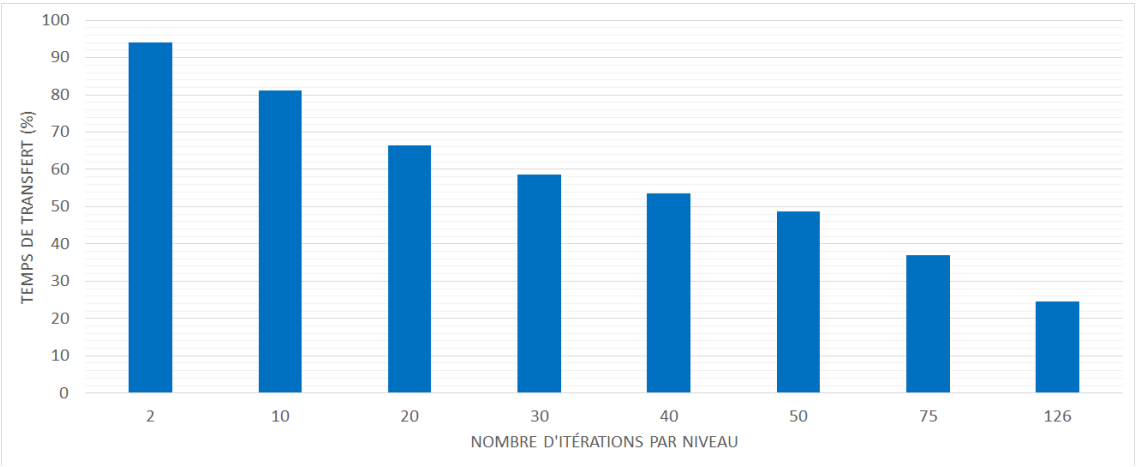


FIGURE 8.16 – Évolution du temps de transfert obtenu en faisant varier le nombre d'itérations réalisées pour chaque niveau.

Une telle réduction du temps de transfert permet de mettre à profit la puissance de calculs des processeurs graphiques et d'obtenir de meilleures performances.

8.4.4 Influence de la taille du sous-domaine

Le but est ici de comparer les performances obtenues pour différentes tailles de sous-domaines. Il est légitime de penser qu'une taille de sous-domaine plus petite peut impliquer une réduction de l'interface et pourrait donc conduire à de meilleures performances.

Le tableau 8.2 décrit les performances obtenues pour des tailles de sous-domaines de 32^3, 64^3 et 128^3.

Taille du bloc	Performances (MLUPS)	Temps de transfert (%)
32^3	237	42
64^3	337	27.1
128^3	451	24.4

TABLEAU 8.2 – Comparaison des performances obtenues pour différentes tailles de sous-domaines.

Nous constatons qu'une diminution de la taille du sous-domaine n'est pas nécessairement un critère permettant d'améliorer les performances. Il semblerait même au contraire que les performances soient meilleures avec une taille de blocs plus importante. En réalité, la diminution de la taille du sous-domaine a pour impact de diminuer le nombre maximal d'itérations possibles pour chaque niveau mais il augmente le nombre de communications entre les sous-domaines calculés ce qui conduit généralement à une baisse de performances (section 7.4.6). Augmenter de façon trop importante la taille des sous-domaines est également un facteur pouvant réduire les performances. En effet, une augmentation trop importante conduit nécessairement à une diminution du nombre de sous-domaines pouvant tenir dans un niveau. Une diminution de ce nombre contribue alors à une augmentation du nombre de niveaux ce qui est nécessairement un facteur pénalisant pour les performances. Dans notre cas, le choix d'une taille de sous-domaine de 128^3 offre un bon compromis en ce qui concerne les performances de calculs des processeurs graphiques et elle offre également un nombre de sous-domaines acceptable par niveau (48 dans notre cas) ainsi qu'un nombre important d'itérations par niveau.

8.4.5 Étude préliminaire pour une simulation à plus de deux niveaux

L'objectif de cette section est de réaliser une étude préliminaire de performances pour une simulation utilisant plus de deux niveaux. Cette simulation est basée sur la même que la figure 8.13 en considérant cette fois un domaine de calculs de $128 \times 2048 \times 2048$ cellules. En considérant toujours une taille de sous-domaine de 128^3, le domaine est alors découpé en 256 sous-domaines. L'algorithme de classification des niveaux nous donne

alors la répartition présentée dans la figure 8.17. Cette répartition a découpé le domaine de simulation en sept niveaux de calculs successifs. La différence majeure avec un cas à deux niveaux est la gestion des interfaces. En effet, dans ce cas, certains sous-domaines peuvent être considérés comme interface plus d'une fois. Pour qu'un sous-domaine ne soit en réalité plus considéré comme une interface, il faut pouvoir s'assurer que l'ensemble des informations nécessaires lui parviennent. Étant donné que les niveaux sont définis successivement dans un ordre précis, il faut savoir si les informations calculées précédemment sont suffisantes pour assurer le calcul de ce sous-domaine. Si c'est le cas, la grille peut être calculée en tant que correction, sinon elle est de nouveau définie en tant qu'interface.

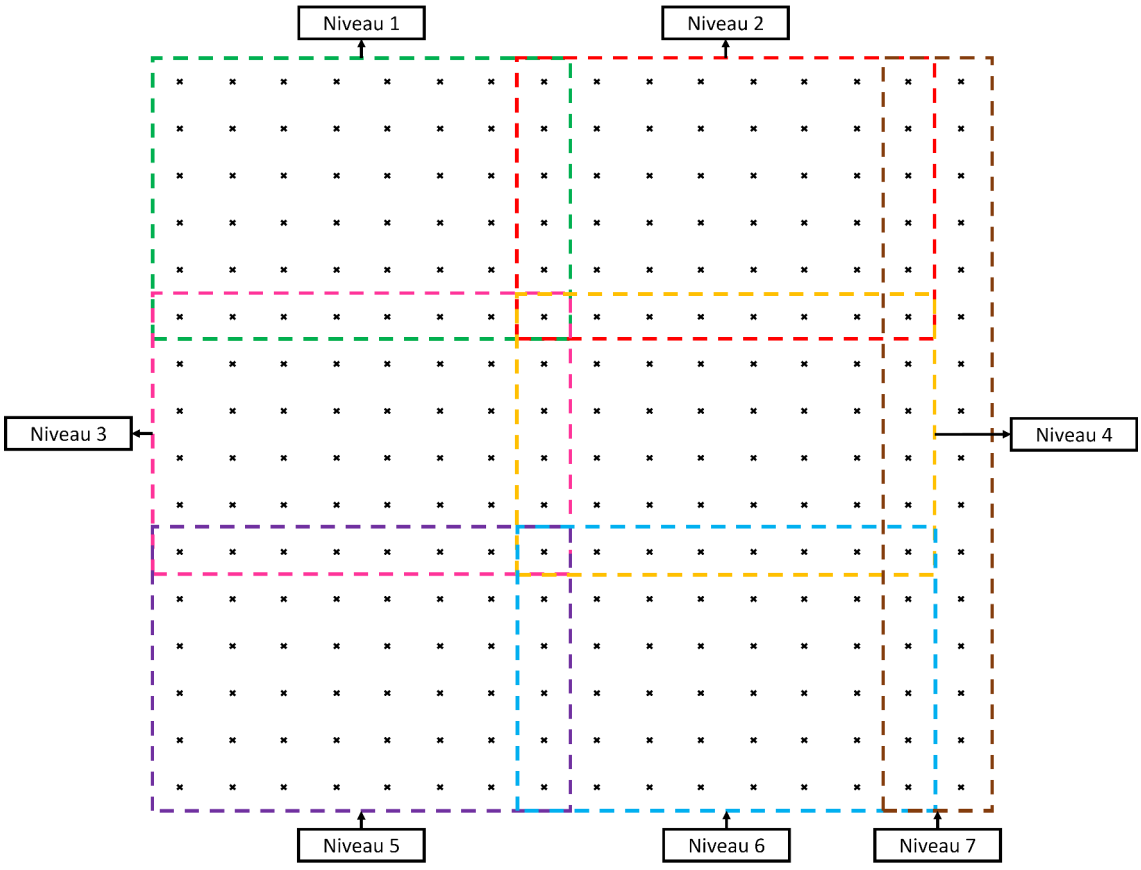


FIGURE 8.17 – Classification des sous-domaines en sept niveaux pour un domaine de taille $128 \times 2048 \times 2048$: une croix représente ici un sous-domaine.

L'augmentation du nombre de niveaux a un impact direct sur les performances de la simulation. Pour celle ci, nous avons obtenu une performance moyenne de 295 MLUPS en considérant 126 itérations successives par niveau, ce qui représente une perte de per-

formance d'environ 35% en comparaison à la première simulation sur un domaine de taille $128 \times 1024 \times 1024$. Cette baisse importante de performance est probablement due à l'augmentation du nombre de sous-domaines situés à l'interface. Cette augmentation implique un accroissement de la demande de re-calculs de sous-domaines, ce qui provoque cette diminution de performances. Pour cette simulation, cela représente un calcul équivalent à 320 sous-domaines alors qu'il est en réalité composé de 256 sous-domaines ce qui provoque une augmentation de 20% de la quantité de calculs. Les performances obtenues restent cependant intéressantes et permettent d'obtenir des résultats de simulations relativement rapidement.

L'extension du partitionnement à des niveaux en 3D est envisagée et sera étudiée à l'issue de ce travail de thèse. L'approche étant basée essentiellement sur la proximité spatiale des sous-domaines, elle devrait s'avérer également efficace.

Cette approche pourrait de la même façon être étendue à des cas de simulations sur une architecture du type CPU multi-cœurs car le principe de fonctionnement est général et n'est pas relié directement au fonctionnement du GPU.

8.4.6 Intégration au sein de la méthode de maillage progressif

La section précédente a permis de définir une méthode « out-of-core » permettant des échanges efficaces entre les processeurs graphiques et la mémoire centrale par l'utilisation de plusieurs niveaux de calculs. Cette méthode a été conçue pour des simulations utilisant un maillage statique découpé en un ensemble de sous-domaines préalablement définis. La méthode se prête bien à des cas de simulations où le maillage est statique mais le but final serait de l'intégrer au sein de la méthode de maillage progressif décrite dans le chapitre 7. Le but de cette section est de mettre en évidence les problématiques intervenant pour la combinaison de nos deux méthodes.

Une première problématique concerne la définition des niveaux de calculs. En effet, la méthode de maillage progressif crée les sous-domaines de manière dynamique. L'ajout des sous-domaines se faisant en fonction de la progression de la simulation, cela garantit une certaine proximité spatiale des sous-domaines appartenant au même niveau. Toutefois, la méthode de maillage progressif ne garantit pas une interface minimale entre les niveaux. Les interfaces sont en effet dépendantes de l'ordre dans lequel les sous-domaines

sont ajoutés.

Une seconde problématique importante concerne les échanges de données entre les processeurs graphiques et la mémoire centrale. Dans le cas où un seul niveau intervient, on peut alors ajouter un certain nombre de sous-domaines jusqu'à ce que le nombre maximal soit atteint. Ainsi, lorsqu'un second niveau apparaît, il est important de savoir quelle quantité de mémoire doit être déchargée sachant que ce deuxième niveau doit également pouvoir progresser par la méthode de maillage dynamique. Une solution simple serait de décharger complètement la mémoire des processeurs graphiques de manière à pouvoir ajouter facilement de nouveaux sous-domaines. Cependant, cette solution risque de ne pas être optimale car cela impliquerait dans la plupart des cas un déchargement de données qui est inutile. Une solution plus adaptée à la méthode de maillage progressif mais demandant un effort de développement conséquent fera l'objet d'une étude plus poussée à l'issue de ce travail.

Conclusion générale et perspectives

Ce travail de thèse aura permis d'imaginer, de concevoir et d'évaluer les performances d'un noyau de simulation utilisant la méthode de Boltzmann sur réseau sur plusieurs architectures de calculs. La modélisation mise en place se devait de pouvoir inclure les interactions entre plusieurs fluides présents au sein d'une simulation. Ce noyau devra à terme inclure des échanges thermiques entre les différents composants mais également réaliser ces échanges avec le sol. L'utilisation d'une équation de Boltzmann discrète spécifique permettant de gérer l'évolution de la température des fluides et du sol a été évoquée. Cela impose une condition spécifique entre la température issue du sol et celle des fluides intervenant dans la simulation. L'intégration d'un terme de changement de phase au sein du noyau de simulation a également été évoquée et testée. Ce terme semble adapté pour faire des simulations permettant la formation de bulles gazeuses dans un environnement liquide en 2D mais son passage en 3D le rend très instable. De plus, le gros écart thermique entre le GNL liquide et l'air ambiant rend également les phénomènes très instables. Des solutions du type raffinement de maillage sont envisagées pour résoudre ces problèmes, mais leur mise en place demande encore un effort théorique important.

L'utilisation d'un tel noyau de simulation impose une demande en ressources informatiques très importante, notamment en vue d'une application sur de très grandes installations industrielles comme un terminal méthanier. Son intégration sur plusieurs architectures de calculs a été étudiée au cours de ces travaux de thèse. La parallélisation des calculs est un facteur clé pour améliorer les performances de simulation. L'utilisation du processeur central a permis de mettre en place des premières simulations mais les performances obtenues sur une architecture multi-cœurs à mémoire partagée imposent très vite des contraintes en terme de vitesse de simulation. En effet, les performances de calculs

des processeurs centraux sont des facteurs très limitants pour ce type d'applications. Une parallélisation utilisant l'ensemble des ressources du processeur central a permis d'obtenir des premiers gains de performances. L'utilisation de l'ensemble des cœurs de calculs à notre disposition, associée à des instructions SIMD de type SSE, ont permis de mettre en évidence des gains de performances importants. Même si l'utilisation d'instructions SIMD semble efficace pour ce noyau, cela nécessite toutefois de très gros efforts de développement et rendent le code de simulation difficilement maintenable en cas d'ajout de nouvelles fonctionnalités au noyau de simulation.

Le processeur graphique a démontré ces dernières années qu'il pouvait être un excellent compromis sur le plan calculatoire. Il est vrai que son utilisation pour des applications spécifiques peut permettre d'obtenir des performances de calculs largement supérieures à celles des processeurs centraux. L'utilisation et l'optimisation des performances du GPU pour notre noyau de simulation a ainsi été une grosse préoccupation lors de ce travail de thèse. Son utilisation diffère de celle du processeur central. Une bonne connaissance de son architecture est indispensable afin d'en tirer les meilleures performances. Les optimisations du parallélisme et des accès mémoires du GPU sont en effet des éléments indispensables pour obtenir des calculs performants. Le GPU a ainsi démontré qu'il pouvait offrir des gains de performances largement supérieurs à l'utilisation du processeur central. L'extension à un cadre de simulation parallélisée sur plusieurs processeurs graphiques a ajouté une problématique supplémentaire. La gestion des communications entre les processeurs graphiques a certainement été la tâche la plus complexe à mettre en œuvre. Nous avons pu mettre en évidence de bonnes performances par un chevauchement efficace des calculs avec les communications. Le gain obtenu est variable en fonction du domaine de simulation et de son découpage. L'amélioration des communications par l'intégration de communications en Peer-to-Peer est un avantage non négligeable pour obtenir un gain de performances supplémentaire. L'utilisation efficace de ce type de transferts nous a permis d'obtenir un gain de performances maximal de l'ordre de 12% en comparaison à l'utilisation de transferts zéro-copy.

Les installations industrielles, et plus particulièrement un terminal méthanier, disposent d'infrastructures spécifiques très canalisées pour limiter la dispersion du danger en cas de fuite accidentelle. L'utilisation d'un découpage statique de la globalité du domaine de simulation, comme la littérature le propose, peut ainsi s'avérer trop coûteux sur le plan calculatoire et sur le plan de la mémoire. C'est pourquoi nous avons proposé la mise en place d'une méthode de maillage dynamique permettant de mailler progressivement le domaine de simulation en fonction de la propagation du ou des fluides. Cette méthode peut s'avérer très avantageuse car elle n'utilise que les zones du domaine de simulation qui sont absolument nécessaires pour le bon déroulement de la simulation. Cette méthode a également pour but d'être généraliste et de s'appliquer à de nombreux types d'écoulements et de géométries. Nous avons pu montrer que cette méthode permettait un gain de performance important pour des simulations fortement canalisées mais qu'elle était moins performante pour des simulations où l'intégralité du domaine se remplissait progressivement. Cela montre que certains éléments de la méthode sont améliorables en vue de maximiser ses performances. La réduction du coût de communications entre les sous-domaines est certainement le facteur à améliorer pour espérer gagner encore en performances. Nous avons proposé dans ce manuscrit une méthode permettant d'optimiser dynamiquement la répartition des sous-domaines en vue d'améliorer ces communications, mais il est possible que d'autres méthodes soient plus adaptées. Exploiter les informations issues de la géométrie pourrait également être un facteur permettant de savoir à l'avance où les sous-domaines seront créés et ainsi de permettre une meilleure répartition des calculs.

La gestion et l'utilisation de la mémoire a également été un des aspects les plus étudiés au cours de ce travail de thèse. Le fait de chercher à réaliser des simulations à grande échelle, sur une architecture de calculs limitée, impose des problématiques concernant la réduction de la mémoire nécessaire pour simuler, mais également des problématiques basées sur l'exploitation de l'ensemble des ressources mémoires à notre disposition. En littérature, l'idée est généralement de grossir la puissance de calculs afin de pouvoir grossir la taille du domaine. Dans notre cas, nous avons cherché à exploiter des techniques permettant une réduction importante de la mémoire lors de la simulation. L'utilisation de la méthode A-A pattern s'est révélée être le meilleur compromis en terme de perfor-

mances et de réduction de la quantité de mémoire.

L'utilisation d'une telle méthode reste malgré tout insuffisante pour réaliser de très grosses simulations. L'utilisation de processeurs graphiques à faible quantité de mémoire peut alors rapidement s'avérer problématique pour assurer une simulation trop importante. C'est pourquoi nous avons recherché et mis en place une solution de type « out-of-core » permettant d'exploiter la grosse quantité de mémoire dont dispose le processeur central. L'objectif de cette méthode est de stocker le maximum de données à la fois sur la mémoire des processeurs graphiques et la mémoire centrale et de mettre en œuvre des échanges efficaces entre les mémoires en exploitant toujours les performances de calculs des processeurs graphiques. Notre méthode a donc permis d'obtenir des résultats limitant la baisse de performances par une réduction de la quantité de données à échanger, mais aussi par la diminution du nombre d'échanges à réaliser. Cette méthode reste encore en chantier et doit être testée sur d'autres cas de simulation. De plus, de nombreuses pistes d'améliorations s'offrent à nous. La combinaison avec la méthode de maillage progressif devrait offrir des performances intéressantes, notamment dans des cas fortement canalisés comme ceux d'un terminal méthanier. En plus de cela, l'ajout d'un algorithme de compression-décompression permettant de réduire la quantité de données à échanger entre les niveaux est également une piste d'amélioration intéressante. En effet, l'utilisation d'un tel algorithme pourrait conduire à une réduction importante de la quantité de données à communiquer ce qui offrirait alors une amélioration conséquente des performances.

Annexe A

Annexes

A.1 Matrice de passage pour le modèle D2Q9 à temps de relaxation multiples

La matrice de passage M permettant de relier les fonctions de distribution aux moments pour les modèle D2Q9 s'exprime de la façon suivante :

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & -1 & -1 & -1 & -1 & 2 & 2 & 2 & 2 \\ 4 & 2 & 2 & 2 & 2 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 2 & 0 & 2 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 2 & 0 & 2 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

A.2 Matrice de passage pour le modèle D3Q19 à temps de relaxation multiples

La matrice de passage M permettant de relier les fonctions de distribution aux moments pour les modèle D3Q19 provient de [d'H02] et s'exprime de la façon suivante :

$$M = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
-30 & -11 & -11 & -11 & -11 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 & 8 \\
12 & -4 & -4 & -4 & -4 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 1 & -1 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & -4 & 4 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & -4 & 4 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & -4 & 4 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & 2 & 2 & -1 & -1 & 1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & -4 & -4 & 2 & 2 & 1 & 1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 & -1 \\
0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & -2 & 2 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & -1 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & -1 & 1
\end{pmatrix}$$

Bibliographie

[BCT73] J.J.M. Beenakker, E.G.D. Cohen, and W. Thirring. The boltzmann equation, theory and applications. *Springer, Wien*, 1973. 13

[BFL01] M. Bouzidi, M. Firdaouss, and P. Lallemand. Momentum transfer of a boltzmann-lattice fluid with boundaries. *Physics of Fluids (1994-present)*, 13(11) :3452–3459, 2001. 25

[BFRU02] G. Bella, S. Filippone, N. Rossi, and S. Ubertini. Using openmp on a hydrodynamic lattice-boltzmann code. In *Proceedings of the Fourth European Workshop on OpenMP, Roma*, 2002. 54

[BGK54] P. L. Bhatnagar, E. P. Gross, and M. Krook. A model for collision processes in gases. i. small amplitude processes in charged and neutral one-component systems. *Physical review*, 94(3) :511, 1954. 14

[BMW+09] P. Bailey, J. Myre, S. D.C. Walsh, D.J. Lilja, and M.O. Saar. Accelerating lattice boltzmann fluid flow simulations using graphics processors. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 550–557. IEEE, 2009. 45, 86, 97, 109

[Bru13] S. G. Brush. *Kinetic Theory : The Chapman–Enskog Solution of the Transport Equation for Moderately Dense Gases*, volume 3. Elsevier, 2013. 14

[BS13] J. Bao and L. Schaefer. Lattice boltzmann equation model for multicomponent multi-phase flow with high density ratios. *Applied Mathematical Modelling*, 37(4) :1860–1871, 2013. 29, 30

[BYS08] J. Bao, P. Yuan, and L. Schaefer. A mass conserving boundary condition for the lattice boltzmann equation method. *Journal of Computational Physics*, 227(18) :8472–8487, 2008. 26

[CCM92] H. Chen, S. Chen, and W. H. Matthaeus. Recovery of the navier-stokes equations using a lattice-gas boltzmann method. *Physical Review A*, 45(8) :R5339, 1992. 17

[Cer75] Carlo Cercignani. *Theory and application of the Boltzmann equation*. Scottish Academic Press, 1975. 14

[CFL09] B. Chopard, J.L. Falcone, and J. Latt. The lattice boltzmann advection-diffusion model revisited. *The European Physical Journal-Special Topics*, 171(1) :245–249, 2009. 21

[CLH+07] S. Chen, Z. Liu, Z. He, C. Zhang, Z. Tian, and C. Zheng. A new numerical approach for fire simulation. *International Journal of Modern Physics C*, 18(02) :187–202, 2007. 17

[CMM96] S. Chen, D. Martinez, and R. Mei. On boundary conditions in lattice boltzmann methods. *Physics of Fluids (1994-present)*, 8(9) :2527–2536, 1996. 25

[Con70] J. Conway. The game of life. *Scientific American*, 223(4) :4, 1970. 15

[Cud12] C Cuda. Programming guide, 2012. 106

[DD00] E. D'Azevedo and J. Dongarra. The design and implementation of the parallel out-of-core scalapack lu, qr, and cholesky factorization routines. *Concurrency - Practice and Experience*, 12(15) :1481–1493, 2000. 151

[DE98] L. Dagum and R. Enon. Openmp : an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1) :46–55, 1998. 54

[d'H94] D. d'Humieres. Generalized lattice-boltzmann equations. *Rarefied gas dynamics- Theory and simulations*, pages 450–458, 1994. 22

[d'H02] Dominique d'Humières. Multiple–relaxation–time lattice boltzmann models in three dimensions. *Philosophical Transactions of the Royal Society of London A : Mathematical, Physical and Engineering Sciences*, 360(1792) :437–451, 2002. 22, 23, I

[dLF86] D. d'Humieres, P. Lallemand, and U. Frisch. Lattice gas models for 3d hydrodynamics. *Europhysics Letters*, 2(4) :291–297, 1986. 16

[Egg96] J. G.M. Eggels. Direct and large-eddy simulation of turbulent fluid flow using the lattice-boltzmann scheme. *International journal of heat and fluid flow*, 17(3) :307–323, 1996. 28

[FdH+87] U. Frisch, D. d'Humieres, B. Hasslacher, P. Lallemand, Y. Pomeau, J.-P. Rivet, et al. Lattice gas hydrodynamics in two and three dimensions. *Complex systems*, 1(4) :649–707, 1987. 17

[FHK+15] C. Feichtinger, J. Habich, H. Köstler, U. Rüde, and T. Aoki. Performance modeling and analysis of heterogeneous lattice boltzmann simulations on cpu–gpu clusters. *Parallel Computing*, 46 :1–13, 2015. 89

[FHP86] U. Frisch, B. Hasslacher, and Y. Pomeau. Lattice-gas automata for the navier-stokes equation. *Physical review letters*, 56(14) :1505, 1986. 16

[GC12a] S. Gong and P. Cheng. A lattice boltzmann method for simulation of liquid–vapor phase-change heat transfer. *International Journal of Heat and Mass Transfer*, 55(17) :4923–4927, 2012. 30

[GC12b] S. Gong and P. Cheng. Numerical investigation of droplet motion and coalescence by an improved lattice boltzmann model for phase transitions and multiphase flows. *Computers & Fluids*, 53 :93–104, 2012. 31

[GLT99] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2 : Advanced features of the message-passing interface*. MIT press, 1999. 58

[GZS02] Z. Guo, C. Zheng, and B. Shi. Discrete lattice effects on the forcing term in the lattice boltzmann method. *Physical Review E*, 65(4) :046308, 2002. 31

[HFK+13] J. Habich, C. Feichtinger, H. Köstler, G. Hager, and G. Wellein. Performance engineering for the lattice boltzmann method on gpgpus : Architectural requirements and performance results. *Computers & Fluids*, 80 :276–282, 2013. 87, 94, 108, 109

[HJ89] F.J. Higuera and J. Jimenez. Boltzmann approach to lattice gas simulations. *EPL (Europhysics Letters)*, 9(7) :663, 1989. 17

[HL97] X. He and L.-S. Luo. Theory of the lattice boltzmann method : From the boltzmann equation to the lattice boltzmann equation. *Physical Review E*, 56(6) :6811, 1997. 17

[HPDP73] J. Hardy, Y. Pomeau, and O. De Pazzis. Time evolution of a two-dimensional classical lattice system. *Physical Review Letters*, 31(5) :276, 1973. 16

[IG03] M. Isenburg and S. Gumhold. Out-of-core compression for gigantic polygon meshes. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 935–942. ACM, 2003. xvii, 151

[JH14] F. Jiang and C. Hu. Numerical simulation of a rising co2 droplet in the initial accelerating stage by a multiphase lattice boltzmann method. *Applied Ocean Research*, 45 :1–9, 2014. 86, 87

[JK14] M. Januszewski and M. Kostur. Sailfish : a flexible multi-gpu implementation of the lattice boltzmann method. *Computer Physics Communications*, 185(9) :2350–2368, 2014. 88, 89, 106, 109

[KKH+99] D. Kandhai, A. Koponen, A. Hoekstra, M. Kataja, J. Timonen, and P.M.A. Sloot. Implementation aspects of 3d lattice-bgk : boundaries, accuracy, and a new fast relaxation method. *Journal of Computational Physics*, 150(2) :482–501, 1999. 20

[KORR10] F. Kuznik, C. Obrecht, G. Rusaouen, and J.J. Roux. Lbm based flow simulation using gpu computing processor. *Computers & Mathematics with Applications*, 59(7) :2380–2392, 2010. 86

[L.10] Bourgois L. *Automates cellulaires et estimation état-paramètres pour la modélisation semi-physique : application à l'assimilation de données environnementales*. PhD thesis, Université du Littoral Côte d'Opale, 2010. 15, 16

[Lat07] J. Latt. Technical report : How to implement your ddqq dynamics with only q variables per node (instead of 2q). Technical report, Technical report, Tufts University, 2007. 42

[LL00] P. Lallemand and L.-S. Luo. Theory of the lattice boltzmann method : Dispersion, dissipation, isotropy, galilean invariance, and stability. *Physical Review E*, 61(6) :6546, 2000. 23

[LL03] P. Lallemand and L.-S. Luo. Lattice boltzmann method for moving boundaries. *Journal of Computational Physics*, 184(2) :406–421, 2003. 25

[LP02] P. Lindstrom and V. Pascucci. Terrain simplification simplified : A general framework for view-dependent out-of-core visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 8(3) :239–254, 2002. xvii, 150

[LPBC] C. Le Page and F. Bousquet (Cirad). Automate cellulaire feu de forêt. ix, 16

[LZWG13] X. Li, Y. Zhang, X. Wang, and W. Ge. Gpu-based numerical simulation of multi-phase flow in porous media using multiple-relaxation-time lattice boltzmann method. *Chemical Engineering Science*, 102 :209–219, 2013. 86, 87

[M+67] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967. 106

[MA02] F. Massaioli and G. Amati. Achieving high performance in a lbm code using openmp. In *The Fourth European Workshop on OpenMP, Roma*, 2002. 55, 56, 63

[Mar08] S. Marié. *Etude de la méthode Boltzmann sur Réseau pour les simulations en aéroacoustique.* PhD thesis, Université Pierre et Marie Curie-Paris VI, 2008. 14, 17

[MHR+07] K. Mattila, J. Hyväluoma, T. Rossi, M. Aspnäs, and J. Westerholm. An efficient swap algorithm for the lattice boltzmann method. *Computer Physics Communications*, 176(3) :200–210, 2007. 42

[Mic12] P. Micikevicius. Gpu performance analysis and optimization. In *GPU Technology Conference*, 2012. 84

[MZ88] G. R McNamara and G. Zanetti. Use of the boltzmann equation to simulate lattice-gas automata. *Physical Review Letters*, 61(20) :2332, 1988. 17

[N.15] Maquignon N. *Vers un modèle multi-phases et multi-composants (MPMC) de type Lattice Boltzmann Method (LBM) pour la simulation dynamique d'un fluide cryogénique dans l'eau.* PhD thesis, 2015. 28, 29, 122

[OB11] M. A O'Neil and M. Burtscher. Floating-point data compression at 75 gb/s on a gpu. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 7. ACM, 2011. 156

[OKT11] C. Obrecht, F. Kuznik, and J.J. Tourancheau, B.and Roux. A new approach to the lattice boltzmann method for graphics processing units. *Computers & Mathematics with Applications*, 61(12) :3628–3638, 2011. 85, 86, 108

[OKTR13a] C. Obrecht, F. Kuznik, B. Tourancheau, and J.J. Roux. Multi-gpu implementation of a hybrid thermal lattice boltzmann solver using the thelma framework. *Computers & Fluids*, 80 :269–275, 2013. 87, 101, 106

[OKTR13b] C. Obrecht, F. Kuznik, B. Tourancheau, and J.J Roux. Multi-gpu implementation of the lattice boltzmann method. *Computers & Mathematics with Applications*, 65(2) :252–261, 2013. 87, 101, 105, 106, 109, 135

[OKTR13c] C. Obrecht, F. Kuznik, B. Tourancheau, and J.J. Roux. Scalable lattice boltzmann solvers for cuda gpu clusters. *Parallel Computing*, 39(6) :259–270, 2013. 89

[PKW+03] T. Pohl, M. Kowarschik, J. Wilke, K. Iglberger, and U. Rüde. Optimization and profiling of the cache performance of parallel lattice boltzmann codes. *Parallel Processing Letters*, 13(04) :549–560, 2003. xi, 40, 56, 57, 63

[Pro11] NVIDIA Visual Profiler. Nvidia corporation, 2011. 98

[PW96] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16(4) :42–50, 1996. 64

[QdL92] Y. H. Qian, D. d'Humières, and P. Lallemand. Lattice bgk models for navier-stokes equation. *EPL (Europhysics Letters)*, 17(6) :479, 1992. 17

[RDVC12] P.R. Rinaldi, E.A. Dari, M.J. Vénere, and A. Clausse. A lattice-boltzmann solver for 3d fluid simulation on gpu. *Simulation Modelling Practice and Theory*, 25 :163–171, 2012. 86, 94, 108

[Ros11] C. Rosales. Multiphase lbm distributed over multiple gpus. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 1–7. IEEE, 2011. 89

[RRB+08] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008. 85

[SC93] X. Shan and H. Chen. Lattice boltzmann model for simulating flows with multiple phases and components. *Physical Review E*, 47(3) :1815, 1993. 29, 31

[SCL14] Z. Shang, M. Cheng, and J. Lou. Parallelization of lattice boltzmann method using mpi domain decomposition technology for a drop impact on a wetted solid wall. *International Journal of Modeling, Simulation, and Scientific Computing*, 5(02) :1350024, 2014. 59

[SGK11] M. Schönherr, M. Geier, and M. Krafczyk. 3d gpgpu lbm implementation on non-uniform grids. In *International Conference on Parallel Computational Fluid Dynamics*, 2011. 44

[Sin13] Graham Singer, 2013. 79

[SK11] J. Sanders and E. Kandrot. *Cuda par l'exemple*. Pearson Education France, 2011. vi, 76, 79

[Sma63] J. Smagorinsky. General circulation experiments with the primitive equations : I. the basic experiment*. *Monthly weather review*, 91(3) :99–164, 1963. 27

[SP02] J. Suh and V. K. Prasanna. An efficient algorithm for out-of-core matrix transposition. *Computers, IEEE Transactions on*, 51(4) :420–438, 2002. 151

[Suc01] S. Succi. *The lattice Boltzmann equation : for fluid dynamics and beyond*. Oxford university press, 2001. 16

[TGR04] S. Thon, G. Gesquière, and R. Raffin. A low cost antialiased space filled voxelization of polygonal objects. *GraphiCon 2004*, pages 71–78, 2004. 62

[Thü07] N. Thürey. *Physically Based Animation of Free Surface Flows with the Lattice Boltzmann Method : Physikalische Animation Von Strömungen Mit*

Freien Oberflächen Mit Der Lattice-Boltzmann-Methode. Verlag Dr. Hut, 2007. xii, 17, 25, 57, 58, 59, 99

[TK08] J. Tölke and M. Krafczyk. Teraflop computing on a desktop pc with gpus for 3d cfd. *International Journal of Computational Fluid Dynamics*, 22(7) :443–456, 2008. 86, 93, 97

[Tol99] S. Toledo. A survey of out-of-core algorithms in numerical linear algebra. *External Memory Algorithms and Visualization*, 50 :161–179, 1999. 151

[Töl08] J. Tölke. Implementation of a lattice boltzmann kernel using the compute unified device architecture developed by nvidia. *Computing and Visualization in Science*, 13(1) :29–39, 2008. 85, 93, 97, 98

[Vit01] J. S. Vitter. External memory algorithms and data structures : Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2) :209–271, 2001. 150

[VNB+66] J. Von Neumann, A. W. Burks, et al. Theory of self-reproducing automata. *IEEE Transactions on Neural Networks*, 5(1) :3–14, 1966. 14

[Vol10] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10. San Jose, CA, 2010. 84

[W+86] S. Wolfram et al. *Theory and applications of cellular automata*, volume 1. World Scientific Singapore, 1986. 17

[WCS+01] K. Wagstaff, S. Cardie, C.and Rogers, S. Schrödl, et al. Constrained k-means clustering with background knowledge. In *ICML*, volume 1, pages 577–584, 2001. 106

[Wik15] Wikipedia. Physique statistique, 2015. 13

[WMK04] W. Wei, X.and Li, K. Mueller, and A. E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *Visualization and Computer Graphics, IEEE Transactions on*, 10(2) :164–176, 2004. 20

[Woo13] C. Woolley. Gpu optimization fundamentals, nvidia, 2013. 84

[ZH97] Q. Zou and X. He. On pressure and velocity boundary conditions for the lattice boltzmann bgk model. *Physics of Fluids*, 9(6) :1591–1598, 1997. 25

[Zie93] D.P. Ziegler. Boundary conditions for lattice boltzmann simulations. *Journal of Statistical Physics*, 71(5-6) :1171–1177, 1993. 24