Pa Vo Un tri parallèle adaptatif

Marie Durand

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Mathématiques et Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Marie DURAND

Thèse dirigée par **Bruno RAFFIN**
et codirigée par **François FAURE**

préparée au sein **d'INRIA**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'information, Informatique**

PaVo
Un tri parallèle adaptatif

Thèse soutenue publiquement le **25 octobre 2013**,
devant le jury composé de :

Mme Marie-Paule CANI
Prof. Grenoble INP, Présidente

M. Christophe CÉRIN
Prof. LIPN, CNRS, Rapporteur

M. Fabrice JAILLET
MCF UCBL, HDR, LIRIS, Rapporteur

M. Vincent FAUCHER
Chercheur CEA, Examinateur

M. Bruno RAFFIN
CR INRIA, HDR, Directeur de thèse

M. François FAURE
Prof. UJF, Co-Directeur de thèse

À mon grand-père.

Remerciements

Merci à mes deux rapporteurs, Christophe Cérin et Fabrice Jaillet pour la lecture approfondie de mon manuscrit. À Vincent Faucher avec qui nous avons également échangé dans le cadre de l'ANR Repdyn. Merci à Marie-Paule Cani de m'avoir, il y a déjà très longtemps, donné envie de faire une thèse et merci beaucoup d'avoir accepté de présider mon jury. Merci à tous pour vos commentaires enthousiastes.

Un merci très chaleureux à mes deux directeurs de thèse, François et Bruno qui m'ont initialement recrutée pour *mettre les mains dans le cambouis*. Merci à tous les deux pour votre humanité, votre compréhension, votre soutien à différents moments cruciaux... et bravo Bruno pour ta Transvésubienne 2013 !

Merci également à Joseph, mon compatriote de rédaction, qui a gagné la course à la soutenance et s'est échappé depuis.

Merci à toute l'ancienne équipe Evasion, aux différents ingés Sofa, aux assistantes super qui ont à tour de rôle pris soin de cette équipe et de ses dérivés (Anne Pierson, Elodie Toihein, Florence Polge), aux chercheurs plein d'humour : FF, Georges-Pierre...

De l'autre côté de la rue, merci au groupe coinche d'alors : Fred, PF, Jean-Marc, Philippe W., Pierre Ne., Brice, Vincent, Swann, Jean-Noël, Guillaume et les autres pour ces intenses moments de *pause*. Merci à tous ceux qui ont chaleureusement veillé à ce que tout se passe bien : en particulier, merci Annie, merci Christian !

Merci au groupe Kaapi, aux nombreuses personnes qui sont intervenues dans la résolution du problème des *5 minutes bonnes idées*, à François B. et Thierry pour la deadline iWomp 2013 ainsi qu'à FiFi qui a courageusement accepté d'aller en Australie à notre place, à Clément pour la motivation ski de rando, à Ziad pour m'avoir prêté *sa* machine...

J'aimerais remercier également tous mes différents co-bureaux dont les stagiaires du printemps 2013, Alexis, David : mention spéciale pour votre bonne humeur et votre enthousiasme. Et puis merci aux deux derniers qui malgré l'exclusion mutuelle m'ont bien soutenue et ont également donné un grand coup de main à l'occasion du pot : FiFi aux Kouign Amann, Mathias à la Brasseuse...

Merci à mes anciens colocs, à l'époque thésards en astrophysique, au groupe Nîmois étendu : merci de m'avoir supportée tout ce temps et d'être venus de loin pour fêter ça ! Un merci particulier à Yom qui m'a motivée à l'époque à me lancer dans ce projet incongru. Fin stratège, il est parti suffisamment loin pour n'avoir de la dernière année que des echos, mais de l'autre bout du monde a tout de même continué à m'encourager.

Et puis, merci à Titou de t'être dit qu'une rencontre en pleine rédaction ne manquerait certainement ni de piment ni de couleurs. Merci de ton soutien sans faille, de m'avoir amicalement embrigadée au début de l'été dans un stage *rédactions intensives* avec vue sur la mer... nous aurions dû y rester !

Enfin, merci à toute ma famille qui s'est déplacée en nombre et de loin le jour J et s'est chargée de l'organisation du pot. Merci du fond du cœur à mes parents et à mon *petit* frère chéri, Frédéric.

Les illustrations à l'aquarelle ont été réalisées
par Mme Brigitte Thibaudier.

Il y a fort longtemps

Il était une fois, non loin de ce que nous appelons maintenant la Cornouaille à la pointe ouest de l'Angleterre, une petite île bien verte sur laquelle régnait Thabor, un troll végétarien. De gentils petits êtres habitaient l'île et lui obéissaient tant bien que mal. Thabor était obnubilé par l'ordre. Tout devait être parfaitement rangé sinon il se mettait dans de folles colères et les habitants devaient alors le fuir sous peine de se retrouver changés en rochers et plantés à jamais au milieu des flots qui bordent l'île.

La période la plus difficile de l'année était la récolte des poires. Thabor adorait cet excellent fruit. Et pour pouvoir se régaler à la mesure de sa gourmandise du moment, il exigeait que les jeunes poires justes cueillies soient systématiquement parfaitement rangées dans l'ordre de leur taille, les unes à côté des autres. Comme le troll mangeait beaucoup, la récolte des poires occupait tous les habitants. Et comme il ne supportait pas le désordre, il fallait que les poires soient triées avant l'aube du jour suivant.

Le jour de la récolte, on préparait un alignement en ordre des précieux fruits. Les habitants couraient ajouter les poires qu'ils avaient trouvées. Seulement, il fallait souvent déplacer toutes les poires déjà en ordre pour insérer la nouvelle à sa place. Comme ils arrivaient tous en même temps, c'était une grande pagaille et ils devaient finalement attendre leur tour et déplacer plusieurs fois une grande partie de la récolte. Ainsi chaque année, plusieurs membres du petit peuple de l'île étaient changés en rochers pointus par Thabor mécontent que le rangement ne soit pas terminé à temps.

Vint une année où une jeune morgane s'aventura sur l'île. Elle sentit la présence des âmes emprisonnées et fut très malheureuse en entendant leur histoire. Elle n'avait pas assez de pouvoir pour défaire les sorts du troll ni pour les aider à ranger autant de poires en aussi peu de temps. Mais, alors que le jour de la récolte s'approchait dangereusement, elle eut une idée. Elle demanda quelle était la longueur de la ligne de poire les années précédentes. Puis avec l'aide d'un jeune habitant, elle prépara sur la place du village de nombreuses lignes. Elle divisa le nombre attendu de fruits par le nombre de lignes pour estimer la place nécessaire. En fait, elle prévit plus de place que nécessaire sur chaque ligne pour pouvoir espacer les poires.

Elle expliqua ensuite que cela fonctionnait comme une bibliothèque. Mais sur cette île, personne n'avait jamais vu de livres, encore moins de bibliothèque. Elle expliqua alors qu'au début de chaque ligne, on mettrait une poire qui serait la plus petite de la ligne. D'une ligne à l'autre en partant du nord de la place, la première poire est plus grande que celle de la ligne précédente. Ainsi, tout le monde pourrait rapidement savoir dans quelle ligne insérer son butin.

Quand vint le moment de tester son stratagème, la morgane proposa que sur chaque ligne, une personne soit désignée pour ranger les poires en ordre. Les habitants non assignés à une ligne apportaient les fruits jusqu'à la bonne ligne. Le temps passait mais les poires se rangeaient correctement... et rapidement ! Il y eut une fois où il fallut réorganiser toute une partie car il y avait plus de petites poires que prévu. Deux heures avant l'aube, les habitants amenaient les deux dernières ! Mais ce n'était pas fini : pour que Thabor soit content, il fallait encore rassembler toutes les lignes et surtout supprimer les trous. Ce fut rapidement fait et les habitants eurent même le temps de faire une petite sieste.

À l'aube, Thabor vint inspecter le travail et ne trouva rien à redire. Pour la première fois, nul ne fut changé en rochers et les habitants préparèrent une grande ronde. Mais lorsqu'ils voulurent remercier la morgane qui leur avait rendu ce bien grand service, elle avait déjà disparue...

Préface

Objectif : présenter le plus simplement possible la simulation numérique parallèle.

Mardi 8 avril 2008. Léo et Léa descendent du car. Ils sont devant une énorme étendue d'eau. En haut d'une grande butte de terre sur laquelle il y a même une route. La maîtresse leur a dit hier que des milliers de milliers de milliers de litres d'eau sont retenus par cette digue.

Après avoir marché derrière le parapet qui surplombe l'eau, la classe se dirige vers le bâtiment principal. La visite commence par la projection d'une petite animation qui explique à quoi sert le barrage et comment il fonctionne.

Dans le bureau de Théo, l'ingénieur qui reçoit ensuite les enfants, une maquette du barrage et de ses conduits leur permet de s'imaginer l'intérieur du bâtiment. Sur les murs se trouvent des posters représentant d'autres barrages.

Très vite, des questions fusent mais une d'entre elles revient souvent :

– comment on sait que le barrage ne va pas se casser ?

L'ingénieur explique brièvement que les concepteurs ont des connaissances sur la résistance des matériaux, sur les lois physiques qui régissent la matière. Des laboratoires spéciaux mènent des études et réalisent des expériences pour déterminer les paramètres physiques et vérifier qu'il est possible d'appliquer les lois. Tout le monde fait des calculs savants, de diverses méthodes. Et puis lorsque les comportements sont suffisamment connus mais qu'il n'est pas possible de faire une expérience grandeur nature pour vérifier que cela fonctionne, ce sont des modélisations qui prennent le relai.

Les modèles utilisés sont des dessins, puis des maquettes à l'échelle. Puis lorsque l'allure de la structure finale est suffisamment précise, des simulations sont construites. Une simulation est un outil qui permet de prédire le comportement d'une expérience sans la réaliser réellement. En général, lorsque les structures à modéliser sont complexes, les simulations sont numériques. C'est-à-dire qu'elles nécessitent l'utilisation d'ordinateurs. Les personnes qui travaillent dans les laboratoires écrivent des programmes informatiques qui utilisent les connaissances physiques.

Problème : pour simuler d'énormes structures suffisamment précisément il faut parfois de nombreux jours de calcul avec des ordinateurs. A la fin du siècle dernier, la rapidité des ordinateurs augmentait et l'on pouvait effectuer des calculs de plus en plus complexes. De nos jours, la technologie a atteint ses limites en terme de vitesse. En revanche, on sait construire et utiliser ce qu'on peut appeler pour résumer des accélérateurs. Il en existe de plusieurs types. Mais pour vous donner une idée, c'est souvent comme si on mettait plusieurs cœurs d'ordinateurs ensemble.

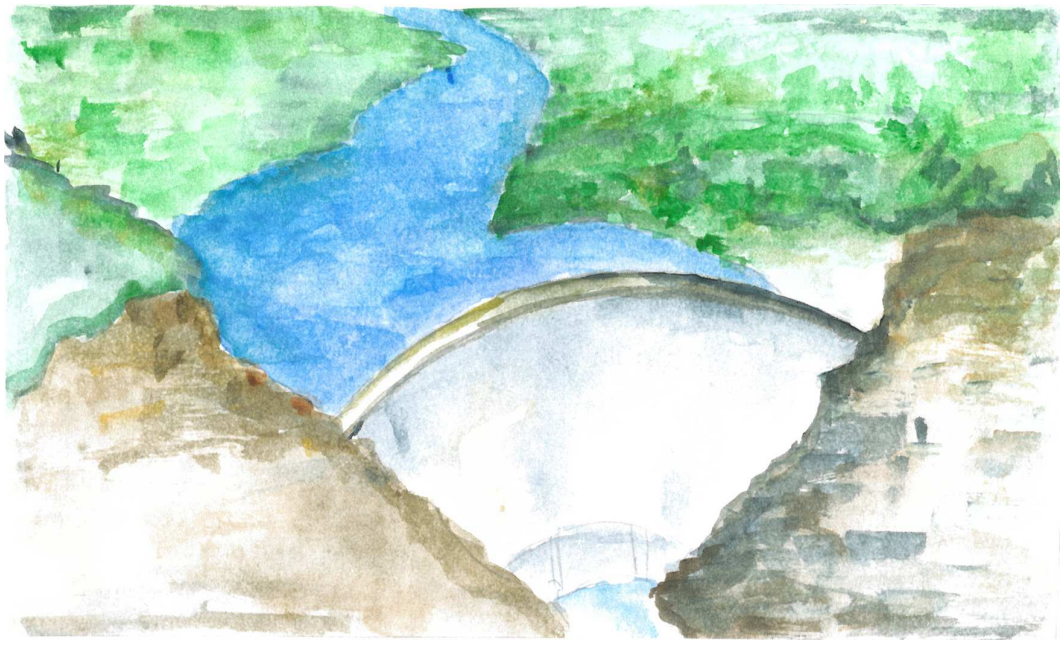


FIGURE 1 – L'eau est contenue par des enceintes en béton (pacifiquement) armé.

Léa et Léo se regardent, perplexes.

Toutes les unités de calcul se répartissent le travail et on aura le résultat plus vite. "Ah, comme lorsque j'apprends la moitié de la poésie et Léo l'autre moitié ?" intervient Léa. "Exactement !" s'exclame Théo en souriant, "cela s'appelle la parallélisation d'un programme".

Bien entendu, les plus gros accélérateurs demandent beaucoup d'argent pour les construire et pour leur permettre de fonctionner. Tous les laboratoires ne peuvent pas avoir accès à n'importe quel moment à ces ressources spéciales. Alors les scientifiques essaient d'utiliser efficacement tous les ordinateurs disponibles couramment. Ainsi seules les simulations gigantesques seront exécutées sur les plus gros accélérateurs.

"Mais moi, je suis spécialisé en mécanique, pas en informatique. Venez, je vais vous montrer d'autres maquettes du barrage." La classe s'ébranle alors en direction de la salle d'exposition juste à côté.

A la fin de la journée, les enfants sont tous impressionnés par la visite du complexe de Grand'Maison. Léa et Léo aimeraient bien avoir un programme pour faire des simulations où l'on voit passer l'eau dans les turbines.

Table des matières

Introduction

De nos jours... Non. L'essor et l'importance du parallélisme en informatique ne sont plus à justifier. Les nombreux traitements, calculs, prédictions traités par des outils, applications ou logiciels informatiques tentent tous de tirer partie des possibilités des machines contemporaines. Pour autant, la question est loin d'être entièrement traitée car la solution universelle, qui conviendrait à tous les problèmes pour n'importe quelle architecture, n'a pas encore été gravée dans le marbre [1].

Le GPU, ou carte graphique, qui dispose d'assez de puissance pour afficher effroyablement rapidement un jeu ou toute autre application sur des écrans de plus en plus finement résolus est l'architecture phare depuis la fin des années 2000 dans de nombreux domaines du calcul générique. L'utilisation de ce support particulier s'est étendue parce qu'il était déjà présent sur la plupart des ordinateurs du monde courant, relativement peu coûteux et proposait un nombre extraordinaire d'unités de calcul. Suite à l'avènement d'interface(s) de programmation accessible(s) au plus grand nombre, des accélérations démoniaques ont été rapportées. Petit à petit ces informations sont parvenues aux oreilles bien intentionnées des décideurs et ainsi de nombreuses équipes se sont lancées dans l'aventure. Bref, le *General-Purpose Processing on Graphics Processing Units* est (était ?) à la mode.

Le type de parallélisme offert par un GPU se prête bien à tout ce qui est régulier, suffisamment grand mais pas trop et surtout indépendant. Cela en fait a priori une cible parfaite pour la simulation. En effet, les problèmes traités sont souvent résolus par la répétition de la même opération sur une multitude de briques élémentaires. Cependant, les interactions entre les éléments introduisent des dépendances dans les calculs qui pénalisent l'efficacité parallèle.

Que ce soit en simulation ou en informatique graphique, les interactions entre les objets sont pourtant la manne de l'application. Une personne placée dans un environnement virtuel s'attend à introduire du mouvement lorsqu'elle "touche" un objet placé dans le monde dans lequel elle est immergée. Un astronome sait que l'étoile lointaine intéragit gravitationnellement avec les autres de la galaxie étudiée. L'astéroïde se rapprochant d'une planète est attiré par tous les corps environnants et suite à un mauvais calcul des interactions nous pourrions le croire prêt à s'écraser sur un continent.

Les interactions entre les corps font ainsi l'objet de moult études passionnantes. La difficulté est de les repérer, opération appeléee "détection de collision". Des centaines de méthodes ont été proposées pour résoudre la question. Elles sont généralement très efficaces pour un domaine ou une situation donnés et s'adaptent plus ou moins bien aux variations. La difficulté augmente d'un cran encore lorsqu'il s'agit de les traiter

1. À l'heure où nous éditons, durant l'été 2013

rapidement de manière parallèle et les solutions envisagées dépendent de l'architecture cible.

Lorsqu'un GPU est utilisé, il est souvent fait usage de tri car cela permet d'éviter le recours à des structures de données complexes usant d'indirections et souvent peu efficaces en cache. Les méthodes de détection de collision utilisant un tri sont suffisamment génériques pour être également intéressantes même sur des processeurs plus classiques. Plus largement, le tri est un des piliers de l'informatique. Trier peut être nécessaire pour retrouver plus facilement les opérations bancaires réalisées par Monsieur Durand ou pour éditer la liste des ouvrages d'une bibliothèque... Au sein d'une application, nous pouvons trier aussi bien pour améliorer les performances d'un algorithme que pour diminuer le nombre d'accès à la mémoire en rapprochant les informations proches. Si tous les livres d'un auteur sont regroupés sur le même rayon d'une bibliothèque, il suffit de se rendre devant ce rayon pour se rendre compte de l'importance de l'œuvre de l'auteur... sinon il faudra envisager de compter le nombre de pas entre les différents rayons où sont rangés les différents livres après avoir repéré chacun d'entre eux, puis de répéter la mesure pour les autres auteurs.

Quand nous évoquons le tri, ce sont probablement plusieurs milliers de versions qui ont été proposées et étudiées. Ce qui importe dans un algorithme c'est qu'il soit efficace en général mais également qu'il ne soit pas trop mauvais lorsque cela se passe mal. D'autres critères, comme la place supplémentaire nécessaire, aident à déterminer quelle méthode sera la plus adaptée à une situation donnée.

Lorsque l'application cible relève de la simulation physique, il se produit généralement des changements de l'ordre entre éléments impliquant une mise à jour des structures. C'est la question fondamentale que nous nous posons principalement dans cette thèse : **est-il possible de réordonner des éléments rapidement en cours d'exécution et ce, même en parallèle ?**

Il existe des algorithmes de tris pour le maintien d'un ordre entre des éléments. Cependant, le coût des déplacements mémoire associés peut être prohibitif. Nous nous intéressons à une structure de données particulière, le *Packed Memory Arrray* (PMA) dont le principe est d'utiliser des trous dans un tableau pour diminuer le nombre de déplacements en mémoire lors de l'ajout ou du retrait d'éléments triés. Cette structure est utilisée dans le domaine des bases de données pour augmenter l'efficacité des multiples insertions et suppressions.

Convaincus qu'une telle approche peut aider à résoudre le dilemne entre l'efficacité algorithmique et le coût en mémoire, nous l'avons adoptée. Nous avons développé un algorithme, PaVo, pour effectuer des mises à jour de l'ordre, autrement dit des voyages d'objets au sein de la structure à trous. En traitant ces voyages par paquets, nous factorisons au maximum les coûts générés pour éliminer le désordre dans l'ensemble d'éléments. De plus, nous identifions rapidement les zones pouvant être rangées indépendemment, qualité essentielle pour aboutir à des exécutions parallèles performantes.

Contexte architectural

Les architectures utilisées dans cette thèse ont en commun de posséder des mémoires hiérarchiques. Dans le cadre d'applications irrégulières qui génèrent d'importants flux de données, savoir s'adapter aux hiérarchies mémoires est une condition nécessaire de l'efficacité.

Les architectures multi-cœurs (multi-CPU) sont constituées de plusieurs cœurs répartis en un ou plusieurs processeurs. Sur une machine *Non-Uniform Memory Accesses* NUMA, il existe une mémoire de grande taille à laquelle toutes les unités de calcul peuvent accéder. Cependant, pour être efficaces, les accès à la mémoire sont mis en cache, c'est-à-dire copiés dans une mémoire plus proche du cœur et ainsi plus rapide. Les cœurs disposent généralement de plusieurs espaces dont la taille augmente avec la distance. Certains de ces caches peuvent être partagés entre plusieurs cœurs. L'enjeu est d'utiliser et de réutiliser les données déjà copiées dans l'une de ces mémoires proches afin de limiter les accès à la mémoire distante.

L'architecture d'un GPU est légèrement différente. Les *threads* groupés dans un *block* exécutent la même suite d'instructions sur des données différentes. Les caches sont peu présents sur ces architectures mais différents niveaux de mémoire doivent être gérés explicitement, comme la mémoire globale et la mémoire partagée. Cette dernière est, un peu à la manière d'un cache partagé, une mémoire rapide pouvant être accédée par tous les threads assemblés dans un même *block*. Pour être efficaces, les lectures ou écritures en mémoire globale doivent être fusionnées (*coalesced*) : tous les threads d'un bloc accèdent à des données contiguës et alignées en mémoire.

Dans les deux cas, les coûts liés aux échanges mémoire ne peuvent pas être négligés pour des applications *réelles* comme celles de simulations physiques à portée industrielle. Afin de comparer des algorithmes du point de vue de l'efficacité en mémoire, les deux principaux modèles de mémoire représentent des mémoires à deux niveaux. Des algorithmes pensés dans le modèle dit *Cache-Aware* utilisent les paramètres du système : les taille B des données pouvant être échangées en une opération entre les deux niveaux et la taille M du plus petit espace sensé représenter le cache. Dans le modèle *Cache-Oblivious* les valeurs B et M sont ignorées, d'où son nom. Un algorithme *Cache-Oblivious* efficace le sera sur diverses architectures ayant des valeurs différentes de B et M. De par sa nature, ce modèle est aussi adapté aux hiérarchies mémories multi-niveaux.

Par exemple, parcourir l'espace en suivant une courbe de recouvrement de l'espace (*space-filling curve*) dite en Z permet d'accéder aux données efficacement dans le modèle *Cache-Oblivious*. La courbe en Z est construite par découpes récursives de l'espace. Ainsi en découpant suffisamment il se trouvera systématiquement des blocs de taille adaptée à la taille du ou des différents caches. Autre exemple, la structure PMA que nous utilisons dans cette étude peut être vue comme un équivalent *Cache-Oblivious* aux listes chaînées. Nous en reparlerons...

Contributions

Dans une première phase, nous nous sommes intéressés aux architectures de type GPU dans le contexte de la simulation physique d'impact sur des murs de béton armé. Le problème est passionnant, autant par ses enjeux que par les réalisations techniques nécessaires pour permettre des simulations rapides. Nous avons réalisé une implantation complète de ce type de simulation numérique basée sur des particules particulières sur GPU (chapitre 2). Les résultats ont été publiés sous le titre *DEM Based Simulation of Concrete Structures on GPU* [Durand et al., 2012b]. À cette occasion, nous proposons une technique pour limiter l'empreinte mémoire des structures utilisées pour le stockage de la grille de détection de collision.

De cette étude est ressortie l'importance en termes de complexité et, en conséquence, de temps de calcul de la détection de collision. Dans des simulations particulaires comme celle que nous avons étudiée, les interactions entre tous les éléments voisins doivent être détectées. Nous indiquerons comment ce problème est généralement résolu dans différents contextes. Plusieurs méthodes utilisent des algorithmes de tris, soit pour éliminer rapidement des possibilités, soit pour accéder directement aux éléments d'une même zone géographique.

Quelle que soit la technique utilisée, lorsque les données des éléments voisins géographiques sont proches en mémoire, les opérations de calcul sont nettement accélérées. Nous le re-mettrons en évidence dans le cadre de la simulation de fluides par SPH (section 4.5.3). Cela a grandement motivé notre intérêt pour le tri.

Ainsi, dans un deuxième temps, avons-nous proposé d'utiliser un PMA dans le cadre de la simulation numérique. En profitant de cette structure, nous avons mis au point un algorithme original de tri de séquences presque triées. PaVo, algorithme des Paquets de Voyageurs, est performant car le nombre de déplacements en mémoire est limité grâce à la présence des trous dans la structure. L'intérêt de cette approche a été étudié pour des particules SPH en mouvement et divulgué dans un article intitulé *A Packed Memory Array to Keep Moving Particles Sorted* [Durand et al., 2012a]. Nous approfondissons l'analyse et produisons une généreuse étude expérimentale permettant d'appréhender les subtilités de l'algorithme ainsi que son comportement.

Enfin, nous présentons une implantation parallèle sur multi-cœurs de PaVo. L'extensibilité de l'algorithme est limitée par le coût dominant des accès mémoire. Nous avons examiné l'influence de différentes stratégies de placement en mémoire. Nous avons également proposé de guider la répartition du travail afin de diminuer le coût des échanges mémoire en fonction de la hiérarchie mémoire de la machine utilisée. Cette politique d'ordonnancement a été évaluée dans deux contextes : la parallélisation par tâches, et les boucles parallèles adaptatives. Nous avons utilisé PaVo pour contribuer à une évaluation montrant, entre autres, l'intérêt du partage de travail avec des cœurs proches dans le cadre des boucles parallèles, résultats publiés dans *An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines* [Durand et al., 2013]

Organisation du manuscrit

Le manuscrit s'articule en trois parties. La première, "Contexte" est l'écrin du premier chapitre intitulé **Un soupçon de physique**. Nous introduisons les méthodes numériques adaptées à la simulation physique.

La partie "Des particules en mouvement" est composée d'un unique chapitre, **Interactions sur GPU**. Nous nous intéressons à l'accélération par carte graphique de simulations physiques basées sur des éléments discrets.

En trois chapitres, la partie "Vers des structures adaptatives", fournit une solution complète au problème du maintien de l'ordre d'éléments en mémoire. Le chapitre **Jeux avec des trous** (3) met en évidence l'intérêt de l'ajout d'espaces vides et présente la structure que nous utilisons. Le chapitre suivant, **Les dessous de PaVo** (4), précise l'algorithme au cœur de notre solution et évalue ses performances. Dans le chapitre 5, **Les voyageurs tirent partie de l'infrastructure**, nous évaluons et discutons l'implantation parallèle de PaVo.

La dernière partie, "Croisée des chemins", est -surprise !- réservée à une synthèse des contributions et une présentation des perspectives.

I

Contexte

Un soupçon de Physique 1

Introduction. Dirigeons-nous dans le laboratoire grenoblois L3SR, CNRS, Grenoble. L'équipe RVo, Risques et Vulnérabilité des Ouvrages s'intéresse aux problèmes posés par les risques sismiques, anthropiques et environnementaux aux ouvrages de génie civil.

Les enjeux sont évidemment conséquents puisque les bâtiments considérés peuvent être des barrages aussi bien que des enceintes de centrales nucléaires. Les risques considérés vont de la dégradation naturelle des structures dans le temps à l'impact d'un missile sur une paroi. Cela signifie qu'il faut pouvoir représenter les fissures se formant dans les murs des bâtiments. Ces derniers sont fabriqués avec du sable, des graviers, du ciment, de l'eau, des armatures en acier, le tout assemblé en ce qui est nommé couramment un *béton armé* (figure 1.1).

Comment représenter la destruction de ce béton ? Et d'abord qu'est-ce que c'est ?

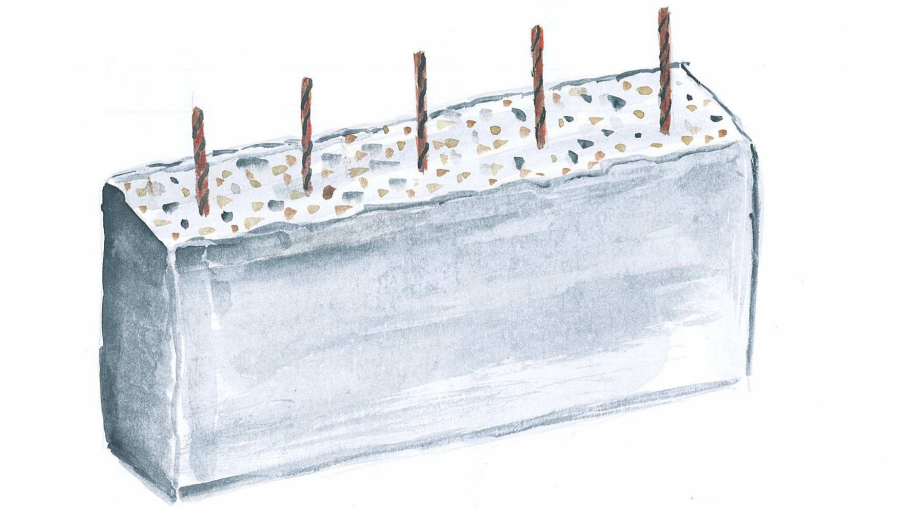


FIGURE 1.1 – Représentation d'un mur en béton armé. Les armatures en acier (en marron) sont coulées dans le béton, un mélange de ciment, d'eau, de gravier et de sable.

Une vertigineuse descente à l'intérieur de la matière nous plonge dans un royaume de *particules* élémentaires, les constituants fondamentaux de l'univers. Ces quarks et autres électrons qui forment à plus grande échelle ce qu'on appelle les *atomes* sont étudiés par la *physique des particules* qui s'intéresse également aux rayonnements et à l'interaction de ces derniers et des particules élémentaires.

Bien que ce domaine soit passionnant, ces objets à la base de tous les autres sont bien trop petits pour que nous les utilisions pour représenter une dalle de béton de plusieurs

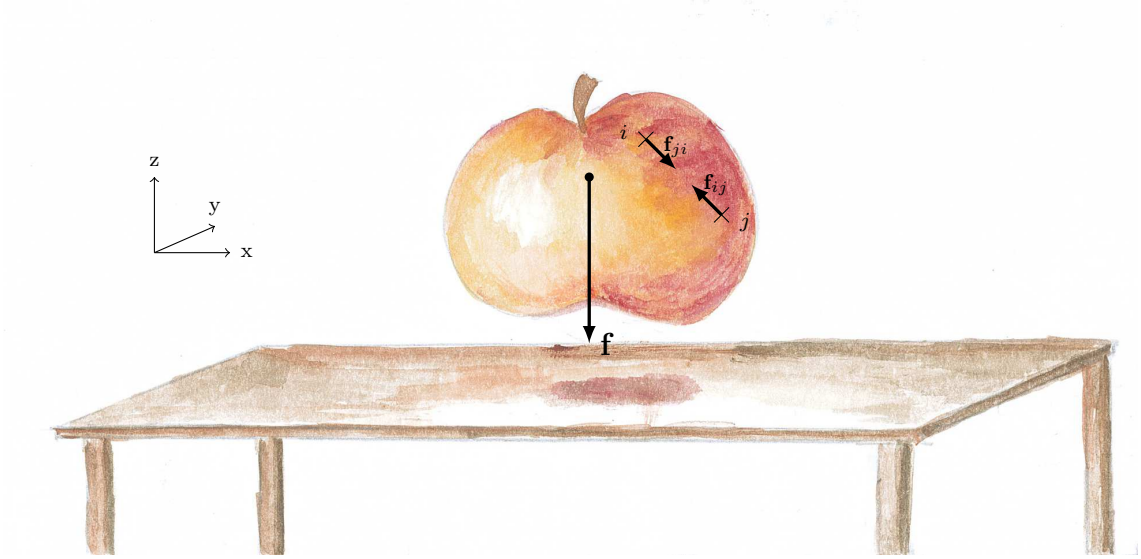


FIGURE 1.2 – Exemple de système physique soumis à la force de gravitation, notée $\mathbf{f}$ ainsi qu'à des forces internes $\mathbf{f_{ij}}$ et $\mathbf{f_{ji}}$.

mètres de large. Pourtant c'est dans la constitution même de la matière que réside le concept fondamental d'une grande partie des simulations numériques : la *particule*.

Avant d'allonger les particules sur un canapé rouge pour les analyser, avant de les introduire dans la danse et de les rendre omniprésentes, parlons de l'évolution d'un *système physique* dans le temps.

Guide de lecture. Nous présentons dans ce chapitre les principaux ingrédients d'une simulation numérique réussie. Un zeste de physique permet de s'accorder sur les notations (section 1.1). Nous discutons ensuite de l'aspect intégration numérique des lois décrivant le mouvement (section 1.1.1) ainsi que des différents types de modèles utilisés. Les modèles les plus classiques (section 1.1.2) sont présentés puis replacés dans leur contexte applicatif courant. Il s'agit en mécanique des fluides (section 1.2) de modèles basés sur des maillages, mais également de modèles hybrides (section 1.2.1).

Le concept de *particule* est introduit (section 1.3), détaillé dans le cadre de la dynamique moléculaire (section 1.4) puis spécialisé avec la méthode des éléments discrets pour la simulation de milieux continus cohésifs comme le béton (section 1.5). Nous décrivons la phase de *détection de collision*, une brique fondamentale des applications qui nous intéressent (section 1.6). Nous terminons par la présentation des simulateurs physiques que nous avons utilisés pour les différents travaux présentés dans ce document (section 1.7).

1.1 Le mouvement

L'évolution d'un système classique est représentée par le principe fondamental de la dynamique ou deuxième loi de Newton :

$$\sum \mathbf{f} = m\mathbf{a} \tag{1.1}$$

avec $\sum \mathbf{f}$ la somme des forces s'appliquant sur le système, m sa masse et $\mathbf{a}$ son accélération. Cette expression est valable si l'on suppose que la masse du système ne varie pas au cours du temps. L'intégration de l'accélération dans le temps permet d'obtenir la vitesse $\mathbf{v}$ du système puis la position $\mathbf{x}$. Cette formulation utilisant des vecteurs, $\mathbf{x}$, représente toutes les dimensions de l'espace considéré, généralement 3.

Les forces s'appliquant sur un corps peuvent être *externes* (notées **fe** par la suite) comme la force de gravité $\mathbf{f}$ sur une pomme (figure 1.2). Il peut également s'agir de forces *internes* (**fi**) entre des éléments du système, par exemple des morceaux de pomme, forces assurant alors la cohésion globale de l'objet. A l'échelle du système global, ici la pomme, la somme des forces internes s'annule en vertu de la troisième loi de Newton, le principe d'action et de réaction :

$$\mathbf{f}_{ij} = -\mathbf{f}_{ji} \tag{1.2}$$

avec i et j deux corps en interaction, $\mathbf{f}_{ij}$ la force exercée par i sur j et $\mathbf{f}_{ji}$ la force exercée par j sur i sont directement opposées, leur somme est nulle.

La formulation de la deuxième loi de newton (équation 1.1) permet d'analyser un mouvement de manière continue dans le temps. Pour reproduire le mouvement de manière numérique, le temps est *discrétisé* et l'état du système est calculé aux différents instants considérés. Il s'agit de l'*intégration numérique* des lois physiques.

1.1.1 Intégration numérique

Ainsi que nous allons l'entrevoir dans plusieurs exemples comme en mécanique des fluides (section 1.2), en électromagnétisme (section 1.2.1) ou encore en dynamique moléculaire (section 1.4), l'expression des lois physiques gouvernant un problème conduit généralement à établir des équations complexes non solvables analytiquement. Deux familles de schémas d'intégration permettent d'approcher la solution de ces équations, les schémas dits *explicite* et *implicite*.

Un schéma d'intégration est explicite lorsque les solutions au temps t_{i+1} peuvent être déterminées à partir des solutions au temps t_i. La valeur $\mathrm{dt} = \mathrm{t}_{i+1} - \mathrm{t}_i$ est appelée le *pas de temps* ou *pas d'intégration*. Ce schéma doit satisfaire une condition de stabilité telle que le pas de temps dt choisi n'excède pas le temps caractéristique du processus étudié.

Au contraire, un schéma d'intégration est implicite lorsque la solution au temps t_{i+1} implique la résolution d'une équation. L'avantage est que le schéma peut être stable quelque soit le pas de temps *dt* choisi. En revanche, la résolution de l'équation peut être ardue.

Parmi les schémas d'intégration explicites, une spécialisation est souvent utilisée pour l'intégration des équations du mouvement, il s'agit du schéma de saute-mouton *leap-frog*. Le principe est de réduire l'erreur en calculant les vitesses en décalage d'un demi pas de temps par rapport aux positions. A chaque nouvelle itération $n+1$, les déplacements sont mis à jour par $\mathbf{x}_{n+1} = \mathbf{x}_n + \mathrm{dt}\mathbf{v}_{n+1/2}$. Puis l'accélération est déterminée en appliquant le principe fondamental de la dynamique (équation 1.1). Dans sa forme la plus simple, l'équation s'écrit : $\mathbf{f}_n = \mathrm{m}\mathbf{a}_{n+1}$. Enfin les vitesses sont mises à jour : $\mathbf{v}_{n+3/2} = \mathbf{v}_{n+1/2} + \mathrm{dt}\mathbf{a}_{n+1}$. Et tout est à recommencer !

1.1.2 Modèles numériques

Différents modèles numériques sont utilisés pour résoudre les équations aux dérivées partielles lors de l'étude des milieux continus. Nous distinguons les méthodes basées sur des maillages telle que la méthode des *Éléments Finis*, des méthodes particulaires comme la méthode des *Éléments Discrets*.

Nous présentons dans les sections suivantes une sélection de méthodes parmi les plus courantes. Nous indiquons les domaines dans lesquels elles sont le plus intensément utilisées ainsi que leurs limitations.

1.2 Mécanique des fluides

Les gaz et les liquides, deux états de la matière correspondant à un milieu sans mémoire de forme, ainsi que d'éventuels états intermédiaires sont rassemblés par le terme *fluide*. L'étude des fluides est une sous-partie des milieux continus.

Les forces internes sont dues à la pression ainsi qu'à la viscosité du fluide dont l'effet dépend du gradient de la vitesse du fluide. Les forces externes peuvent être des forces de gravitation ou électromagnétiques mais aussi des forces liées aux obstacles solides. Les équations de la dynamique du fluide sont établies par l'application de l'équation fondamentale de la dynamique (1.1) en tenant compte de la contrainte de conservation de grandeurs physiques comme la quantité de mouvement. Ces équations sont les équations de *Navier-Stokes*, un parfait exemple d'équations aux dérivées partielles non linéaires (parmi les plus sadiques !).

Deux méthodes principales basées sur des maillages sont utilisées pour décrire les écoulements de fluide. Si l'intérêt se porte sur la forme du fluide, une description Lagrangienne dans laquelle le maillage se déforme avec le fluide est généralement choisie. L'intégration se fait par la *méthode des éléments finis* (FEM). Le référentiel se déplace alors avec l'écoulement ce qui rend complexe l'estimation des propriétés en un point donné de l'espace. Cette méthode est également utilisée dans le domaine de la mécanique du solide.

Si l'intérêt se porte principalement sur les propriétes cinématiques du fluide, une vision Eulérienne peut être utilisée. C'est ici le fluide qui se déplace à l'intérieur d'un maillage de l'espace. L'intégration utilise typiquement la *méthode des différences finies* (FDM), les propriétés du fluide étant calculées aux sommets du maillage.

Dans les simulations où ces deux aspects doivent être étudiés, ainsi que lorsque les écoulements présentent des zones inhomogènes, de fortes irrégularités comme des ruptures, ces méthodes classiques ne sont pas adaptées. Par exemple, dans l'approche Lagrangienne, une rupture dans l'écoulement impose une nouvelle génération de maillage, une opération complexe et coûteuse.

Nous présentons deux exemples de méthodes visant à s'affranchir des inconvénients des méthodes classiques : la méthode *Particle-In-Cell* (PIC) (section 1.2.1),puis la méthode des *Smoothed-Particle Hydrodynamics* (SPH) (section 1.2.2).

1.2.1 Particle-in-cell (PIC)

Cette méthode hybride entre les formulations Lagrangienne et Eulérienne a été conçue par [Harlow, 1957] pour le cas des larges déformations de fluides. La méthode est largement utilisée pour la simulation de *plasma*, un fluide qui transporte des particules chargées.

Le principe de la méthode est la combinaison de particules et de champs. Les trajectoires des particules virtuelles respectent le principe fondamental de la dynamique (1.1). Les champs sont considérés comme des quantités continues et sont définis en différents points d'une grille rectangulaire uniforme. Les interactions entre les particules ne sont considérées qu'à travers leur interaction avec les champs.

Les forces appliquées sur les particules sont déduites des champs. Ces derniers doivent alors être évalués pour la position de chaque particule. Cette étape d'interpolation est généralement appelée *Grid-To-Particle Interpolation*. Les positions et vitesses des particules sont calculées d'après l'équation de Newton. Alors la densité de particules et le courant doivent être mis à jour en chaque point de la grille. Cette étape est appelée *Particle-To-Grid Interpolation*. L'essentiel du temps de calcul est réparti entre ces deux étapes d'interpolation.

Le principal inconvénient de la méthode est son coût en temps de calcul et en mémoire. L'utilisation de paramètres physiques réalistes limite fortement la taille des problèmes pouvant être représentés.

1.2.2 Smoothed-Particle Hydrodynamics (SPH)

La méthode SPH est une méthode particulaire inventée pour la simulation de problèmes d'astrophysique comportant des écoulements. La méthode a plus tard été utilisée pour la dynamique des fluides classiques Elle est intensivement utilisée dans le domaine de *l'informatique graphique*. Son avantage par rapport à la méthode PIC est l'absence de grille pour calculer les dérivées spatiales. Les équations du moment peuvent être exprimées à l'aide d'équations différentielles ordinaires qui sont plus simples à manipuler. Par exemple, le gradient de pression est représenté par une force entre des paires de particules [Monaghan, 1992].

En pratique, en tout point du fluide ses propriétés sont données par l'évaluation des caractéristiques pondérées des particules proches. Les particules utilisées sont

dites *smoothed*, douces, car au-delà d'une certaine distance, elles cessent d'exercer de l'influence. Les lois physiques sont définies à travers des fonctions d'interpolation appelées *kernel*.

La mise à jour des propriétés physiques se fait en deux temps par le calcul des densités puis par celui des forces, utilisant les densités. Lors de chacune de ces étapes, les valeurs pour une particule sont obtenues en fonction des particules voisines suffisamment proches. L'étape de détermination du voisinage est donc effectuée deux fois par pas de simulation, sauf dans les cas où les les interactions sont conservées d'une étape à l'autre.

De nombreux logiciels de simulation implantent une méthode SPH. Parmi ces logiciels, SPHysics est un logiciel open-source dédié à la simulation par SPH des phénomènes hydrodynamiques [SPHysics, 2011]. Fluids, propose un code simple et efficace, également libre, dédié à l'informatique graphique [Hoetzlein, 2012]. Nous présentons ce logiciel utilisé dans le cadre de nos expérimentations ultérieurement (section 1.7.3).

L'inconvénient de cette méthode est qu'il n'existe pas de cadre pour représenter la cohésion de la matière. En effet, si les fluides ou les solides simples peuvent être simulés de cette manière ce n'est pas le cas des solides complexes formés par agrégation de différents matériaux.

1.3 Un point sur les particules

Dans cette section, nous formalisons le concept de particule.

Dans notre contexte, le terme *particule* désigne un objet virtuel localisé dans l'espace par un point. Différentes propriétés comme un volume ou une masse peuvent lui être associées. Les particules sont souvent représentées par des points et peuvent modéliser des atomes dans un fluide, des étoiles dans une galaxie aussi bien que des personnes dans une foule. Les particules peuvent également être traitées statistiquement si elles sont trop nombreuses.

L'utilisation des particules est une brique de base de la physique numérique. Cette modélisation simple permet d'éviter le traitement d'un problème physique complet. De manière générale, l'équation du mouvement (1.1) pour une particule i devient :

$$\sum_j \mathbf{f}_{ji} + \mathbf{fe}_i = m_i \mathbf{a}_i, \tag{1.3}$$

où $\mathbf{fe}_i$ représente les forces externes appliquées sur la particule i. Les forces internes $\mathbf{fi}_i$ appliquées sur la particule i est donnée par $\sum_j \mathbf{f}ji$, c'est-à-dire la somme sur chacune des autres particules de la force qu'elles exercent sur i.

Dans un ensemble de particules interagissant entre elles de manière gravitationnelle, chaque particule attire chacune des autres. Cette situation est le problème dit des *N-corps* (N est le nombre de particules du système). Ce problème, très important notamment en cosmologie et en mécanique des fluides, est l'exemple typique d'un problème ne pouvant pas être résolu de manière exacte (section 1.4.1.2).

L'introduction des particules permet de traiter les cas de ruptures dans la matière continue. La *dynamique moléculaire* (MD) présentée dans la section suivante

(section 1.4) est une méthode basée uniquement sur des particules. La méthode est utilisée dans de nombreux domaines de la simulation physique. La méthode des *éléments discrets* (DE) détaillée ensuite peut être vue comme une spécialisation de la dynamique moléculaire (section 1.5).

1.4 Dynamique moléculaire

La dynamique moléculaire (MD) est la simulation numérique des mouvements physiques des atomes et des molécules. Le principe de base est la résolution numérique des équations de Newton pour un système de particules en interaction où les forces et les potentiels d'énergie sont définis par les lois de la mécanique moléculaire. Cette méthode scientifique traite de la mécanique *classique* uniquement (non relativiste, non quantique).

Initialement conçue pour la recherche fondamentale en physique à travers l'étude des transitions de phase [Alder and Wainwright, 1957], la discipline est devenue commune en sciences des matériaux ainsi qu'en biochimie et en biophysique.

En pratique c'est à un wagon d'applications que cette discipline a ouvert le jour. Dans les grandes lignes : lorsqu'il s'agit d'étude théorique, la dynamique moléculaire permet de s'intéresser à des problèmes d'équilibre aussi bien que des problèmes de diffusion. Cette méthode permet aussi l'étude des transitions de phase, des fluides complexes comme lors de l'étude de la structure et de la dynamique du verre, de l'eau pure et des solutions aqueuses. C'est également un champ pour investiguer les polymères, les solides, les biomolécules (protéines, etc...). Également, la dynamique des fluides et ses écoulements sont régulièrement traités grâce à la MD.

1.4.1 Forces

Pour les simulations de MD, la tâche la plus intensive est le calcul des forces en fonction des caractéristiques internes des particules. Les forces auxquelles sont soumises les particules à l'intérieur du système sont distinguables en deux catégories, les forces issues des liaisons entre les atomes qui composent une molécule (section 1.4.1.1) et les forces dues aux interactions entre plusieurs molécules (section 1.4.1.2).

La sous-partie la plus coûteuse est le calcul des interactions non issues de liaisons car elle implique la détection de ces interactions.

1.4.1.1 Forces de liaisons

La partie correspondant aux forces de liaison contient par exemple les forces des liaisons covalentes entre les atomes.

La modélisation de la matière à l'échelle microscopique se base sur une description précise de la constitution des particules. Une telle description doit en principe utiliser la mécanique *quantique*. En MD, les atomes et les molécules sont représentés par des points massifs interagissant à travers des forces qui dépendent de la distance entre

les objets. L'aspect quantique des interactions issue de la superposition des nuages électroniques est représenté par un système de masses couplées avec des ressorts. Pour les applications plus complexes, des structures étendues sont utilisées pour prendre en compte d'autres phénomènes, notamment l'orientation relative des particules.

1.4.1.2 Forces dues aux interactions

Les forces représentant à la fois l'attraction et la répulsion entre molécules (ou entre différentes parties de la même molécule) sont décrites par les forces de *van der Waals*. Une manière de prendre en compte ces forces complexes est d'utiliser le *potentiel de Lennard-Jones* qui s'écrit :

$$U(r_{ij}) = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{1.4}$$

avec ϵ la profondeur du puits de potentiel, σ la distance finie à partir de laquelle le potentiel s'annule, r_{ij} la distance entre les particules i et j.

Ces forces sont dites à *courte portée*. Pour les établir il faut déterminer les interactions entre les particules suffisamment proches. Il existe une distance de coupure à partir de laquelle les forces sont considérées comme négligeables. Cette étape particulièrement coûteuse et qui relève du champ de la détection de collision sera expliquée un peu plus tard (section 1.6.1).

Des forces à *longue portée* peuvent être introduites pour prendre en compte les forces électrostatiques. Dans ce cas, toutes les interactions entre les particules doivent être considérées. La situation est alors similaire à celle du problème des *N-corps* dont la complexité en temps est $O(N^2)$. La susmentionnée méthode *Particle-in-Cell* (section 1.2.1) peut être adaptée et utilisée dans ce contexte. La technique de *Barnes-Hut* dont la complexité est en $O(N \log N)$ se base sur une division récursive de l'espace en cellules cubiques de plus en plus petites tant qu'elles contiennent des particules. Les cellules sont stockées dans une structure d'arbre [Barnes and Hut, 1986]. La méthode multipolaire rapide (*Fast Multipole Method*) qui consiste à regrouper les sources proches et à les traiter comme une source unique [Rokhlin, 1985 ; Greengard and Rokhlin, 1987] est une deuxième technique classique et fonctionne en $O(N)$.

Nonobstant la pression subie (et bue !), nous n'en dirons pas plus sur ces gloutonnes interactions à longue portée.

1.4.2 Évolution des forces

La plupart du temps, le nombre de liaisons n'évolue pas car les ruptures de liaisons covalentes ne sont pas considérées dans le cadre de la dynamique moléculaire classique.

En revanche, à chaque pas d'intégration, les interactions entre particules non liées doivent être de nouveau déterminées pour appliquer les forces correspondantes.

1.5 Éléments discrets

Les particules sont aussi utilisées pour la simulation de matière continue solide par la *méthode des Éléments Discrets*. Cette représentation initialement proposée pour les matériaux non-cohésifs comme le sable [Cundall and Strack, 1979] a ensuite été enrichie par la définition de lois d'interaction adaptées aux matériaux cohésifs [Hentz et al., 2004].

Le modèle utilise un ensemble de particules de différentes masses, types et tailles de manière à reproduire un comportement isotropique et homogène à l'échelle macroscopique. L'évolution de la matière est ainsi définie par le mouvement des particules selon 6 degrés de liberté dans l'espace, 3 pour la translation et 3 pour la rotation. Les interactions sont gérées par deux raideurs, normale et tangentielle.

Nous distinguons deux types d'interaction entre les éléments, les *liens* et les *contacts*. Les liens sont les interactions initiales entre les éléments, c'est-à-dire celles qui représentent la cohésion de la matière. Les contacts sont les interactions qui seront créées en cours de simulation entre deux éléments suffisamment proches. Dans les deux cas, l'évolution de l'interaction va dépendre de la position et des propriétés des deux éléments. Mais ces interactions obéissent à des lois différentes : les forces de contact sont des forces de répulsion uniquement.

Contrairement au cadre général de la dynamique moléculaire, les liens peuvent se rompre. D'autre part, l'information de la création d'un contact entre deux éléments doit être conservée car les contacts ont une histoire comme les liens.

1.6 Détection de Collision

Dans cette section, nous nous intéressons à la *détection de collision*, c'est-à-dire la détermination des interactions entre les éléments, ce uniquement dans le cadre d'interactions à courte portée.

1.6.1 Dynamique moléculaire

La méthode la plus simple consiste à tester toutes les paires possibles de particules entre elles et à déterminer celles qui interagissent effectivement. Si cette méthode est simple, sa complexité en $O(N^2)$ est tout à fait rédhibitoire dès qu'il s'agit de déterminer les interactions de plus de quelques dizaines de particules entre elles.

Il existe deux familles de méthodes pour rendre cette étape moins gourmande en temps de calcul.

Liste de cellules. La première famille, dite *listes de cellules* construit une grille de détection par la division en cellules de l'espace physique dans lequel le système étudié est plongé (figure 1.3). Les particules dont le centre de masse se trouve à l'intérieur des limites d'une cellule lui sont associées par un simple calcul algébrique. Pour limiter

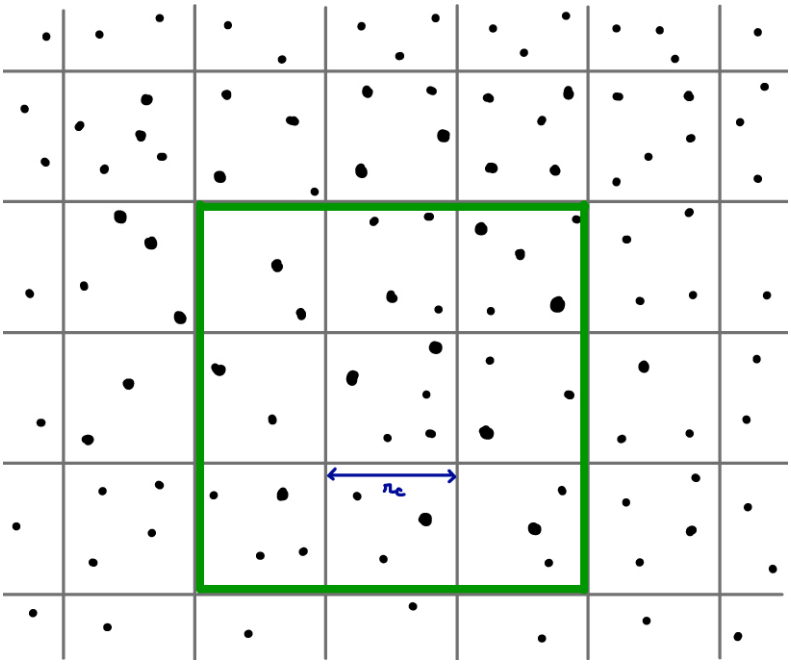


FIGURE 1.3 – Représentation d'une grille de détection, les cellules ont pour dimension le rayon de coupure r_c.

l'espace mémoire nécessaire pour stocker ces informations, les références des particules appartenant à une même cellule sont stockées dans des listes dynamiques. Puisqu'il s'agit d'interactions à courte-portée le nombre de voisins à considérer est limité, il existe une distance de coupure r_c au delà de laquelle deux éléments ne peuvent pas interagir. Si la taille des cellules correspond à cette distance limite alors le nombre de cellules voisines à considérer pour chaque cellule vaut 14 en trois dimensions si la réciprocité des interactions (équation 1.2) est prise en compte, 27 sinon.

Un inconvénient est que l'appartenance des particules aux cellules doit être reconstruite à chaque étape. De plus, en supposant une répartition homogène des particules, $\frac{4/3\pi r_c^3}{14 r_c^3} = \frac{4\pi}{3\times 14} \approx 30\%$ des particules contenues dans le volume cubique constitué des 14 cellules sont véritablement dans le rayon d'interaction. Autrement dit, beaucoup de tests réalisés sont inutiles.

Listes de voisins. Aux échelles considérées, les systèmes changent lentement. La deuxième famille, les *listes de Verlet* [Verlet, 1967] ou *listes de voisins* consiste à tirer partie de cette dernière propriété en préparant, pour chaque particule, une liste de *voisins* qui restera valide pendant plusieurs itérations (de l'ordre de quelques dizaines). Il suffit pour cela de repérer toutes les particules situées à une distance inférieure à une distance Δr supérieure au rayon de coupure r_c (figure 1.4). Conserver le déplacement maximal de toutes les particules au cours de la simulation permet de déterminer quand remettre à jour les listes de voisins.

Le principal défaut est l'espace mémoire nécessaire qui augmente drastiquement avec la distance Δr choisie. Il s'agit finalement d'un compromis à établir entre le temps

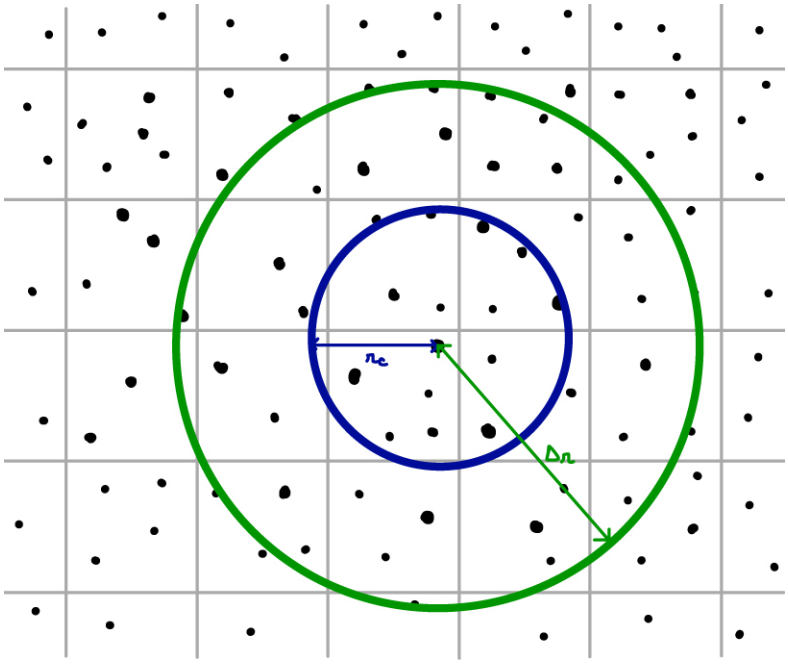


FIGURE 1.4 – Méthode de Verlet : les particules situées à moins de $\Delta r > r_c$ sont considérées comme voisines.

de calcul, via la fréquence de mise à jour des voisins, et l'empreinte mémoire.

En pratique, les deux méthodes sont souvent combinées, la liste des voisins étant établie par division de l'espace en cellule de taille $\Delta_r > r_c$.

1.6.2 Éléments discrets

Dans le cadre de la simulation de structures par éléments discrets, la situation est un peu différente de celle de MD. Lorsque deux éléments s'approchent suffisamment, une nouvelle interaction est créée et peut plus tard être détruite si la distance dépasse un seuil de rupture établi à la création.

La méthode choisie utilise la division de l'espace en cellules constituant une grille de détection. Lors du parcours des interactions potentielles, il faut s'assurer qu'une interaction (lien ou contact) n'a pas déjà été créée entre les deux particules candidates. La réponse aux collisions consiste donc en la création d'une nouvelle interaction, c'est cette interaction qui permettra le calcul des forces à appliquer sur toutes les particules.

En raison de l'évolution lente des positions des éléments, la détection de collision n'est effectuée que de manière périodique, suffisamment toutefois pour ne pas engendrer d'erreurs significatives.

1.7 Simulateurs numériques

Nous présentons dans cette section les principaux simulateurs numériques avec lesquels nous avons travaillé à l'occasion des travaux présentés dans ce document.

1.7.1 Europlexus

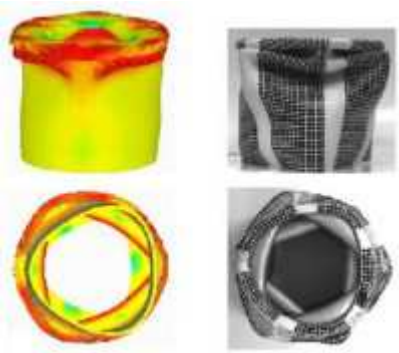


FIGURE 1.5 – Simulations de compression de cylindre réalisées avec Europlexus.

Europlexus (EPX) est un code de dynamique rapide destiné à l'analyse des phénomènes transitoires brutaux non-linéaires impliquant des fluides et des structures en interaction (figure 1.5). Il repose sur une intégration temporelle explicite sans inversion systématique de système linéaire, avec une limitation du pas de temps imposée par la condition de stabilitié de Courant [Courant et al., 1928]. De fait, il est particulièrement adapté lorsque le chargement et les conditions aux limites requièrent une finesse de discrétisation comparable avec le pas limite autorisé, ce qui est le cas pour des problématiques d'impact ou d'explosion.

Le logiciel, écrit en **FORTRAN**, permet de mettre en oeuvre et faire interagir de nombreuses formulations différentes. Par exemple, il est possible de représenter les structures par des éléments finis et le fluide par des volumes finis ou encore de modéliser une enceinte en utilisant pour trois murs sur quatre des éléments finis et des éléments discrets pour le dernier. Dans cette dernière situation, il faut définir le couplage entre éléments finis et éléments discrets [Rousseau et al., 2009].

1.7.2 Sofa

SOFA est un logiciel de simulation modulaire qui permet de représenter aussi bien des opérations chirurgicales, des modèles mécaniques que des fluides (figure 1.6). Conçu pour des applications temps-réel du domaine médicale, le logiciel est développé par plusieurs équipes projets de l'INRIA à Lille, Grenoble et Sophia.

SOFA est écrit en **C++**, il a bénéficié précocemment d'accélération par GPU utilisant le langage **CUDA** pour de nombreux composants ainsi que pour les calculs courant d'intégration numérique. Une interface en **OpenCL** est également disponible.

FIGURE 1.6 – Exemples de simulations réalisées avec le logiciel SOFA [Allard et al., 2011].

Des travaux ont été menés pour tirer partie de la combinaison des architectures multi-CPU et multi-GPU dans le cadre de la simulation physique interactive [Hermann et al., 2010].

1.7.3 Fluids

Fluids est un simulateur de fluides SPH pour CPU et GPU (figure 1.7). Ce simulateur open source a été écrit pour permettre à la communauté un accès à un simulateur de SPH simple et efficace. La version courante, **Fluids v.3** a été publiée en décembre 2012 et fait suite à la version **Fluids v.2** en apportant des améliorations comme l'augmentation du nombre de particules traitées, un code simplifié, une meilleure utilisation de la mémoire.

A l'époque des travaux présentés (section 4.5), seule la version 2 était disponible.

Pour la phase de détection de collision, l'implantation de **Fluids** se base sur deux tableaux : un pour toutes les cellules de l'espace considéré et un autre pour les particules. Le code utilise une opération de parcours des particules pour les placer dans les cellules. À la fin de l'opération, les cellules non vides pointent sur leur première particule et les particules d'une même cellule sont chaînées par indice entre elles.

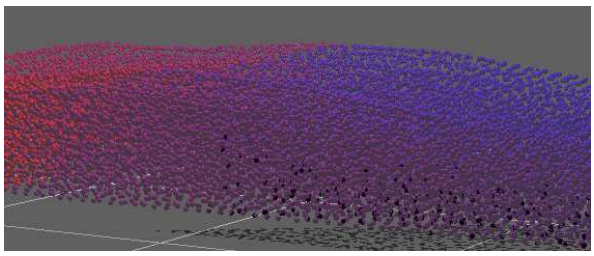


FIGURE 1.7 – Simulation de fluide SPH avec Fluids. Le dégradé de couleurs des particules rend compte de leur répartition en mémoire.

II

Des particules en mouvement

Interactions sur GPU 2

Dans ce chapitre, nous nous intéressons à l'apport de l'accélération par carte graphique pour des simulations de niveau industriel basées sur des éléments discrets (figure 2.1). Différents types d'interactions considérés, des particules complexes ainsi que la détection de collision sont autant de difficultés à surmonter pour pleinement tirer partie de l'architecture d'un GPU.

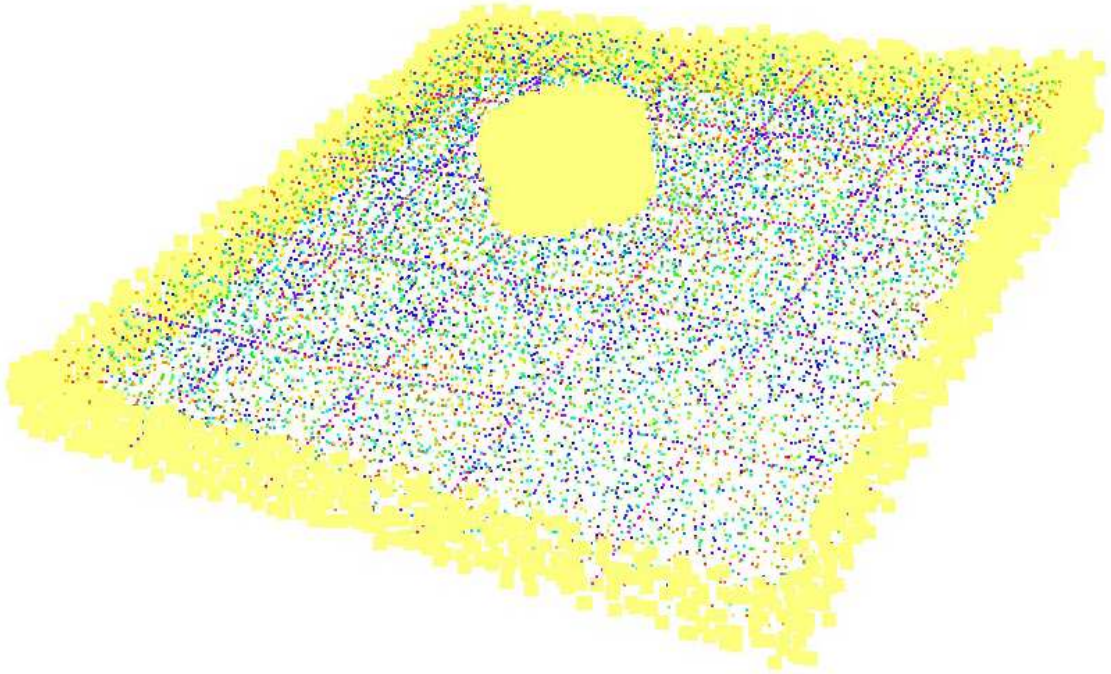


FIGURE 2.1 – Simulation avec Sofa d'un impact sur une dalle de béton armé. Ici ne sont figurés que le centre de masse des éléments discrets, leur couleur symbolisant leur emplacement relatif en mémoire. Les points entourés de jaune sont des éléments soumis à une contrainte. Les éléments sur le bord de la dalle sont immobilisés et ceux qui composent le disque d'impact subissent un déplacement imposé.

Pour représenter du béton armé, l'acier présent dans les armatures des structures considérées est modélisé par des alignements d'éléments de même taille (figure 2.2). Le comportement global d'une armature est analogue à celui d'une poutre. Le matériau béton est reproduit par des éléments de tailles différentes, répartis autour des armatures jusqu'aux frontières de la dalle simulée. Pour aboutir à des simulations pertinentes, l'interface acier-béton a été spécifiquement étudiée [Rousseau, 2009].

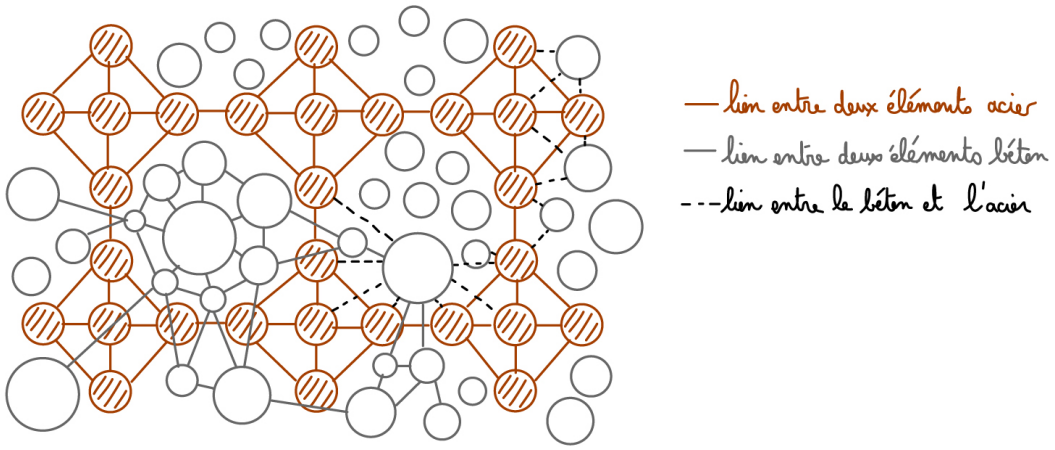


FIGURE 2.2 – Représentation du béton armé par des éléments discrets de béton (en gris) et d'acier (en marron, hachuré). Certaines interactions sont figurées par des traits entre les éléments.

Nous présenterons la simulation en détail (section 2.1) puis les différentes structures utilisées par les mécaniciens (section 2.2). Nous développerons l'algorithme sur GPU en commençant par une revue des simulations comparables (section 2.3.1). Nous exposerons ensuite les aspects pratiques fondamentaux pour adapter le code au GPU (sections 2.3.2, 2.3.3, 2.3.4). Enfin nous présenterons notre implantation et ses spécificités (sections 2.3.5, 2.3.6, 2.3.7). Nous terminerons par une mise en perspective des résultats obtenus (section 2.4) et des enseignements que nous en avons tirés (tant que ce ne sont que les enseignements...).

2.1 Simulation avec les éléments discrets

La simulation susmentionnée suit une boucle principale dont nous expliquons ci-dessous les étapes principales (figure 2.3).

Le schéma d'intégration dans le temps est explicite par différences centrées (1.1.1). Au début d'un nouveau pas de temps, les vitesses $\mathbf{v}$ correspondant au demi pas de temps sont calculées et stockées. L'opération exécutée correspond schématiquement à :

$$\mathbf{v} += \mathbf{a}\frac{dt}{2} \tag{2.1}$$

avec $\mathbf{v}$ les vitesses de tous les éléments dans chacun de leurs 6 degrés de liberté, et $\mathbf{a}$ les accélérations correspondantes. Le temps est avancé de la valeur du pas de temps dt calculé à la fin de l'itération précédente. Les positions $\mathbf{x}$ sont ensuite mises à jour en fonction des nouvelles vitesses et de la valeur du pas de temps. L'opération est alors :

$$\mathbf{x} += \mathbf{v}dt \tag{2.2}$$

Ces deux opérations seront triviales à paralléliser puisque tous les degrés de liberté sont indépendants.

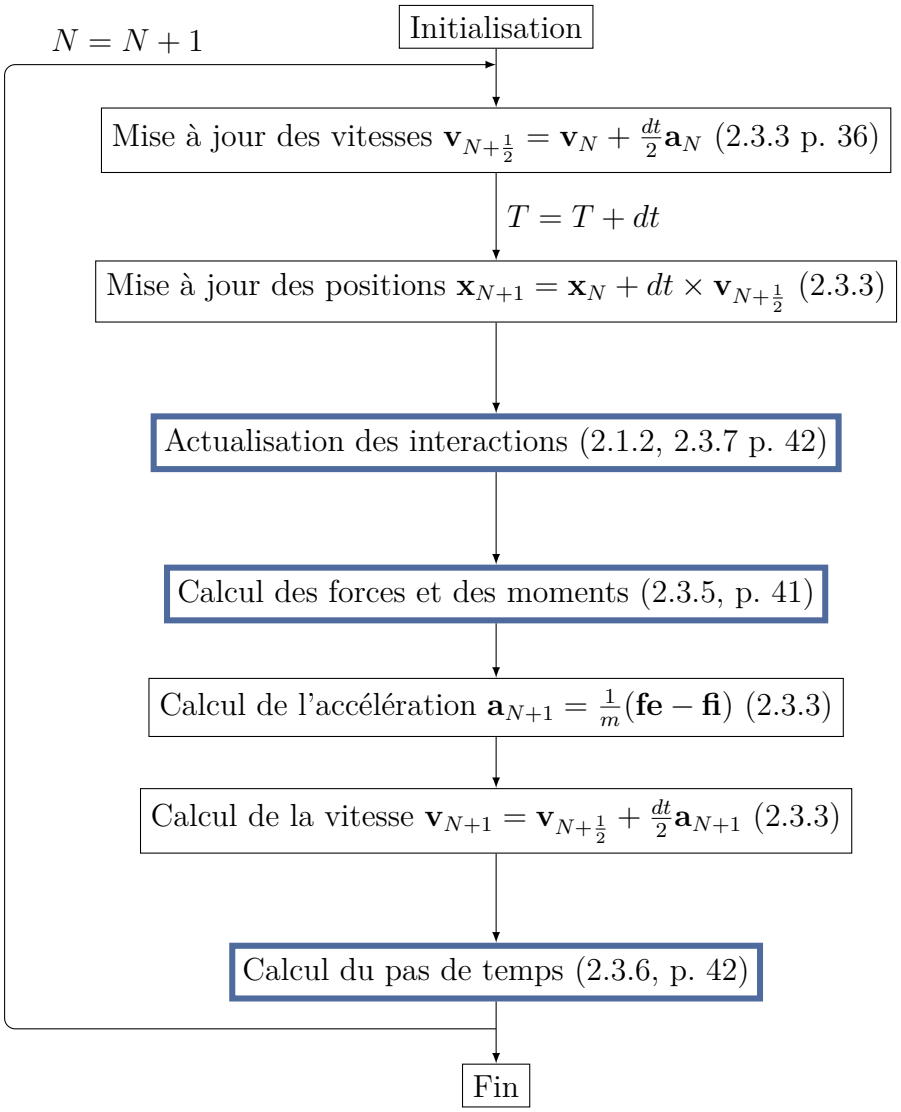


FIGURE 2.3 – Schéma de la simulation. Les positions **x**, vitesses **v** et accélérations **a** sont calculées pour les 6 degrés de liberté de chacun des éléments. L'accélération de chaque élément est obtenue à partir du bilan des forces externes **fe** et internes **fi** (incluant les moments) et de la masse m de l'élément. L'avancée dans le temps T se fait d'un pas de temps dt à chaque nouvelle itération (le numéro de l'itération est conservé dans la variable N).
Les étapes entourées en **bleu épais** sont les étapes spécialisées pour les ED. Les autres étapes sont traditionnellement traitées par le noyau du simulateur. Entre parenthèses figurent les références et les pages des sections se rapportant aux différentes étapes.

Dans un deuxième temps, les interactions entre les éléments sont actualisées. En fait, il s'agit de la détection de collision entre les particules de la simulation (1.6). Nous reviendrons sur la description de cette étape coûteuse plus en détail dans la section (2.1.2). Parce que les pas de temps sont très petits, il est possible d'effectuer cette étape périodiquement (typiquement une fois tous les dix pas de temps) sans altérer la conformité des résultats.

Ensuite, les forces **f** (6 composantes) à appliquer sur les éléments sont déterminées. Pour calculer les forces internes dues aux interactions entre les particules, l'ensemble des interactions (liens et contacts) entre les éléments est traité. L'état courant de l'interaction considérée est établi à l'aide de ses nombreux paramètres (figure 2.4(b)). Les interactions peuvent se rompre, auquel cas elles sont otées de la liste. Les forces engendrées par chaque interaction sont appliquées aux deux éléments concernés de manière locale aux éléments. Pour assurer la stabilité du système, une fois toutes les forces internes calculées, une étape supplémentaire est nécessaire pour certains types d'éléments. Les forces internes sont ensuite accumulées dans les vecteurs forces globaux.

L'accélération **a** est déduite du bilan des forces externes **fe** et internes **fi** par :

$$\mathbf{a} = \frac{1}{m}(\mathbf{fe} - \mathbf{fi}) \tag{2.3}$$

Les vitesses sont alors mises à jour à partir de leur valeur au demi pas de temps précédent et de la valeur des nouvelles accélérations. L'opération est ici :

$$\mathbf{v}+ = \mathbf{a}\frac{dt}{2} \tag{2.4}$$

Comme lors de la mise à jour des positions et des vitesses en début de pas de temps, ces opérations sont fortement *data parallèles*. En effet, le calcul de toutes les coordonnées des vecteurs sont indépendantes.

Enfin, le nouveau pas de temps critique est calculé en fonction des raideurs équivalentes appliquées sur chaque élément (2.1.1).

2.1.1 Stabilité et pas d'intégration

Comme nous l'avons expliqué précédemment (1.1.1), lors de l'intégration numérique et particulièrement avec un schéma explicite, il convient d'adapter le pas de temps pour garantir la stabilité du système.

Ici, comme l'explique [Rousseau, 2009] :

> Les équations à résoudre pour un système composé d'éléments discrets peuvent être comparées à celles obtenues pour une somme de systèmes masse-ressort (de masse m et de raideur k) à un degré de liberté. La pulsation propre de chaque système est $\omega = \sqrt{\frac{k}{m}}$. Pour l'intégration par le schéma explicite par différences centrées de ce système à un degré de liberté non amorti, la condition de stabilité [Courant et al., 1928] s'exprime par :
>
> $$dt \leq \frac{2}{\omega} \tag{2.5}$$

En pratique, la pulsation propre totale du système n'est pas déterminée mais une raideur équivalente est calculée. Pour chaque interaction, une fois les forces calculées, une raideur critique est évaluée selon l'état de l'interaction pour chacune des composantes en fonction des raideurs normale k_n et rotationnelle k_r. Cette raideur critique est accumulée sur chacun des éléments de l'interaction. Lorsque toutes les interactions ont été passées en revue, tous les éléments sont parcourus et pour chacune de leurs coordonnées, le temps critique est obtenu par : temps $= \sqrt{\frac{\text{masse}}{k_n}}$ pour la partie normale ou temps $= \sqrt{\frac{\text{inertie}}{k_r}}$ pour la partie tangentielle. Le pas de temps maximal pouvant être choisi comme pas de temps d'intégration est le minimum des temps critiques calculés ainsi.

2.1.2 Détection de collision

Au fur et à mesure de la simulation, des interactions peuvent se rompre à cause de l'éloignement des particules. Mais ce n'est pas le seul changement à apporter à la machiavélique liste des interactions ! Des contacts (type d'interaction généré au cours de la simulation) peuvent se créer lorsque deux éléments sont trop proches. A nous de détecter ces nouvelles interactions.

Pour la simulation considérée, la méthode utilisée se base sur la création d'une grille de détection régulière (méthode décrite section 1.6). La taille des cellules est choisie égale au plus grand rayon d'interaction entre les particules. Dans ce cas, seule la cellule courante et ses 26 voisines devront être parcourues pendant la phase de recherche. Le nombre de cellules voisines à vérifier peut être réduit de moitié pour ne pas détecter deux fois une interaction entre deux mêmes éléments.

À chaque détection de collision, la grille de détection est recréée. Cette opération consiste en une recherche des positions minimales et maximales des éléments sur les trois directions de l'espace. La grille pourrait être conservée au cours de la simulation, si l'espace physique était statique, comme dans le cas de la démonstration simple de NVIDIA où des particules évoluent dans une boîte fixe [Green, 2010]. Il faut ensuite déterminer dans quelle cellule se trouve quelle particule et pouvoir rapidement accéder aux particules d'une même cellule.

Lorsque ces informations sont établies, il est possible de procéder à la détection des nouvelles interactions. Pour chaque paire de candidats au contact (i.e. deux éléments sont suffisamment proches), il faut vérifier qu'il n'existe pas déjà une interaction (lien ou contact) entre les deux éléments. Il est ainsi nécessaire de conserver la liste des interactions d'un élément. Si un nouveau contact est établi, il doit être ajouté à la liste des interactions des deux éléments. Il sera ainsi pris en compte lors du prochain calcul de forces.

2.2 Les éléments discrets en mémoire

Les ED utilisés transportent de nombreux paramètres physiques utilisés pour calculer les forces ou lors de la création d'interaction (figure 2.4(a)). Le module d'EPX utilisé

(a) élément discret (b) interaction

FIGURE 2.4 – Paramètres physiques d'un élément discret (a), d'une interaction (b) avec leur taille en mémoire : `d` signifie `double` (8o), `i` est utilisé pour `integer` (4o). `3d` indique un vecteur : `double[3]` (24o).

comme référence stocke ainsi 208o pour chaque élément. A ces données il faut ajouter de 96o pour le stockage des identifiants des *interactions* de chaque élément.

Pour représenter les interactions, 36 paramètres étaient utilisés (dont 7 en trois dimensions) (figure 2.4(b)) par l'implantation dans EPX. Une interaction nécessite ainsi 316o en mémoire. Dans le cadre de la modélisation des structures de béton armé, chaque élément est en moyenne en interaction avec 12 autres éléments. Autrement dit, il faut compter 6 interactions par élément.

Au total, pour 260 000 éléments il faudra stocker de l'ordre de 5Go en mémoire. Atteindre le million d'éléments implique 21Go en mémoire uniquement pour les structures contenant les paramètres mécaniques.

En fait, certains des paramètres transportés avec les interactions dans la version EPX ne sont pas nécessaires après la phase d'initialisation. Nous verrons dans la suite qu'une réorganisation avec réduction de l'empreinte mémoire est possible autant qu'utile sur GPU (section 2.3.3).

2.3 EDs sur GPU

Dans cette section, nous détaillons les algorithmes mis en place pour l'implantation de cette simulation sur GPU. Le logiciel Sofa (section 1.7.2) a été choisi pour la mise en pratique des ED sur GPU car ce simulateur physique modulaire proposait déjà une implantation sur GPU de certaines étapes typiques : mise à jour des vitesses, des positions, des accélérations.

Dans le cadre de l'étude présentée ici, l'espace de simulation peut devenir gigantesque par rapport au rayon d'interaction minimum, lorsque des débris sont éjectés. Aussi avons-nous proposé pour la détermination des listes de particules par cellule, une méthode qui tout en résidant sur le GPU et permet néanmoins de représenter un domaine infini (section 2.3.7 p.42).

2.3.1 Etat des lieux

Lors de la mise en place de ce projet, aucun portage sur GPU de simulations basées sur des ED n'avait été rendu public. Les simulations les plus proches étaient celles issues de la dynamique moléculaire (MD), et parmi elles les simulations considérant les interactions à courte portée. D'autres types de simulations ont été étudiées sur GPU (section 2.3.1.3). Les exemples que nous avons sélectionnés sont intéressants par les techniques utilisées pour la détection de collision.

Nous avons par la suite trouvé une référence à des travaux contemporains des notres s'intéressant à des simulations ED sur GPU (section 2.3.1.4).

2.3.1.1 Dynamique moléculaire

[van Meel et al., 2008] et [Anderson et al., 2008] ont proposé dès 2008 les premières implantations presque intégralement sur GPU de simulations de MD. Ces deux travaux

contemporains présentent des techniques générales d'utilisation du GPU similaires mais utilisent pour la détection de collision des méthodes différentes. Il s'agit des listes de cellules pour [van Meel et al., 2008] et des listes de Verlet pour [Anderson et al., 2008] (section 1.6.1 p.17). Dans les deux cas, une grille de détection est construite pour la détermination soit des interactions, soit des voisins. Les listes chaînées habituellement utilisées sur CPU pour stocker les informations nécessaires à cette étape sont remplacées par des matrices organisées pour accéder aux données de manière fusionnée [Anderson et al., 2008]. Les structures les plus difficiles à construire pour cette étape sont celles qui associent à chaque cellule la liste de ses particules.

[van Meel et al., 2008] proposent de maintenir cette liste par une mise à jour avec une technique en deux temps : détection des particules quittant une cellule puis intégration des particules arrivant dans la cellule. Pour limiter le nombre de lectures de la mémoire principale du GPU, cette opération est appliquée au moment du calcul des forces et pas juste après le calcul des nouvelles positions. L'inconvénient est que la correspondance particules - cellules n'est pas parfaitement à jour. Pour ne pas oublier d'interaction, la taille des cellules est alors légèrement agrandie. Dans un travail ultérieur présentant l'utilisation de particules plus complexes de type ellipsoïdales, [Sunarso et al., 2010] utilisent le CPU afin d'établir ces listes à chaque pas de temps.

[Anderson et al., 2008] qui revendiquaient à l'époque sur la même architecture de meilleures accélérations que [van Meel et al., 2008] déportent également vers le CPU l'opération de constitution des listes de particules de chaque cellule. [Rapaport, 2011] propose une amélioration de la méthode grâce à une meilleure organisation des données en mémoire mais ils utilisent tout de même le CPU pour cette étape. Ces travaux utilisant la méthode des listes de Verlet (ou listes de voisins), la construction de la grille de détection est nécessaire moins souvent que lors de l'utilisation des listes de cellules directement (section 1.6.1 p.17).

[Anderson et al., 2008] et [Rapaport, 2011] constatent pareillement l'importance de l'ordre des particules en mémoire et discutent l'utilisation de différentes méthodes comme un tri des particules suivant une courbe de Hilbert[Butz, 1969] pour recouvrir l'espace 3D en conservant une bonne localité [Moon et al., 2001 ; Dai and Su, 2003]. L'accélération obtenue en 2008 est de l'ordre de 60 pour la partie de calcul des forces et 30 pour la génération des listes de voisins conduisant à une accélération globale moyenne de 36 grâce au GPU (la GeForce 8800 GTX de NVIDIA est comparée à un processeur Intel Xeon E7525 à 3.6GHz [Anderson et al., 2008]).

Pour l'établissement de la liste des particules appartenant à une cellule, tous les travaux présentés ci-dessus utilisent des structures dont la construction est complexe à paralléliser, d'où le recours régulier au CPU.

Dans le cadre des systèmes hétérogènes multi-GPU, [Brown et al., 2011] ont proposé une méthode utilisant le tri des particules et permettant de ne stocker pour chaque cellule que sa première particule, réduisant ainsi l'espace mémoire nécessaire. Cette méthode se ramène à celles présentées un peu plus loin dans le cadre des simulations SPH (section 2.3.1.3).

2.3.1.2 Limites de la comparaison

Bien que la MD et les EDs soient similaires dans les grandes lignes, d'importantes différences font que l'apport du GPU pour les ED ne peut pas être déduit directement des résultats en MD.

En premier lieu, dans le cas des ED, plusieurs types d'interactions sont utilisées. De plus, les forces correspondant aux interactions de type contact ne sont pas une simple réponse. Un contact détecté est créé et doit être conservé. Son histoire au cours du temps permettra d'estimer les forces subies par les particules concernées. Ainsi il est nécessaire de maintenir la liste des interactions **établies** de chaque élément et de ne pas recréer d'interaction entre deux éléments déjà en interaction.

D'autre part, ce que nous nommons, dans le cadre des ED, phase de détection de collision, comprend la génération des listes de cellules ainsi que la détection des **nouvelles** interactions entre les particules. Dans le cadre MD, la détection des particules en interaction proprement dite est soit incluse dans le calcul des forces (méthode de la liste de cellule) soit considérée dans la détermination des listes de voisins. Le calcul des forces est alors réalisé par le parcours de la grille de détection ou par le parcours des voisins, avec dans les deux cas une vérification de la distance entre les particules et une annulation éventuelle de la force. Dans le cadre de la simulation par ED, le calcul des forces intervient dans un deuxième temps et est réalisé par un parcours des interactions établies auparavant.

En outre, les phases de calcul des interactions des ED sont plus gourmandes en terme d'opérations et de nombre de registres que dans les simulations de MD. Pour les simulations en MD, il est généralement plus efficace sur GPU de ne pas prendre en compte la troisième loi de Newton (1.2) et de calculer les forces pour chaque atome (chaque force est calculée deux fois). Ce n'est pas le cas pour les ED dans notre situation : les forces dues à chaque interaction sont établies puis chaque élément parcourt la liste des interactions qui le concernent et accumule ainsi les forces subies.

Nous reviendrons sur ces différences à l'occasion de la présentation des algorithmes utilisés.

2.3.1.3 Dans les autres domaines

Un des premiers exemples de mise en place d'une méthode de grille de détection sur GPU date de 2007 [Harada et al., 2007]. Le contexte était la simulation de fluides par SPH (section 1.2.2) et l'implantation était faite en OpenGL, les langages de programmation génériques tels que CUDA n'étant pas encore disponibles. L'espace devait être découpé en cellules stockées dans des voxels pouvant contenir ainsi uniquement 4 particules. Les auteurs notaient déjà le problème du domaine de simulation qui ne peut pas être infini si l'espace est découpé uniformément. Cette technique a par la suite été améliorée pour supprimer la limitation du nombre de particules par cellule [Bayraktar et al., 2009 ; Zhao et al., 2010].

A l'occasion de la publication de CUDA, [Green, 2010] a décrit à l'aide d'un exemple, nommé *Particles*, comment efficacement simuler un système de particules en mouvement

dans une boîte fixe sur GPU. Dans cette simulation, la détection de collision à courte portée à l'aide d'une grille uniforme est traitée en utilisant le tri des particules selon l'indice de la cellule à laquelle elles appartiennent. La première et la dernière particule de chaque cellule de l'espace simulé sont facilement déterminées par un parcours du tableau des particules triées. Dans le cas de cellules vides, une valeur par défaut est stockée. [Zhang et al., 2011] ont utilisé cette méthode pour la simulation interactive de fluides SPH sur multi-GPU.

[Goswami et al., 2010] trient les particules selon leur indice dans une courbe en Z (illustration en page 57, figure 3.1). L'espace physique est divisé en un nombre de blocs égal à une puissance de deux. Les blocs correspondent à de grosses cellules d'une grille de détection classique. Leurs dimensions sont en fait liées à la taille des blocks du GPU. Les auteurs montrent qu'il est possible de limiter l'empreinte mémoire à deux attributs par bloc. De plus, leur méthode permet de ne pas lancer l'exécution sur le GPU des blocs vides, c.-à-d. des blocs qui ne contiendraient pas de particules.

2.3.1.4 Résultats récents sur les ED

Des algorithmes pour une version multi-coeurs (dont GPU) de la méthode ED ont finalement été présentés par [Shigeto and Sakai, 2011]. En ce qui concerne l'étape de détection de collision, ils proposent l'utilisation d'une liste chaînée pour stocker les éléments appartenant à chaque cellule en évitant l'augmentation gargantuesque de la taille mémoire lorsque les cellules contiennent plusieurs particules. Pourtant, le nombre de cellules est tout de même fixé. L'inconvénient de leur approche est que les particules ne peuvent pas être lues de manière fusionnée à cause des indirections de la liste chaînée. De plus, la construction de la liste au sein de la structure n'est pas décrite en parallèle.

Au demeurant, la simulation présentée est très simple : un seul type d'élément simulé et seules des réponses aux contacts sont appliquées, c'est-à-dire qu'ils ne supportent pas la gestion d'un historique d'interaction entre les éléments. Ils obtiennent avec le GPU NVIDIA Tesla C1060 des accélérations de l'ordre de quelques unités ($2,5$x) par rapport à une parallélisation avec OpenMP sur un Xeon W5590 (8 cœurs).

2.3.2 Préliminaires : gestion des différents types d'interactions

Les sections suivantes détaillent les structures et algorithmes utilisés pour la simulation d'ED sur GPU. Le schéma d'intégration est inchangé par rapport au schéma séquentiel (page 27) et les différentes étapes sont exprimées à l'aide d'un ou plusieurs *kernels* GPU (figure 2.5).

Les EDs sur lesquels nous travaillons peuvent représenter du béton ou de l'acier. Lorsque ces éléments interagissent entre eux, les sournoises lois mécaniques ne sont pas les mêmes entre deux éléments de béton, d'acier ou encore entre un élément de béton et un d'acier. Le comportement des liens créés dès le début et représentant la cohésion de la matière n'est pas similaire à celui des contacts établis par la suite.

Vous l'auriez compris, des lois mécaniques différentes entraînent des lignes de code différentes. Or, comme nous l'avons noté en introduction, pour bien utiliser un GPU,

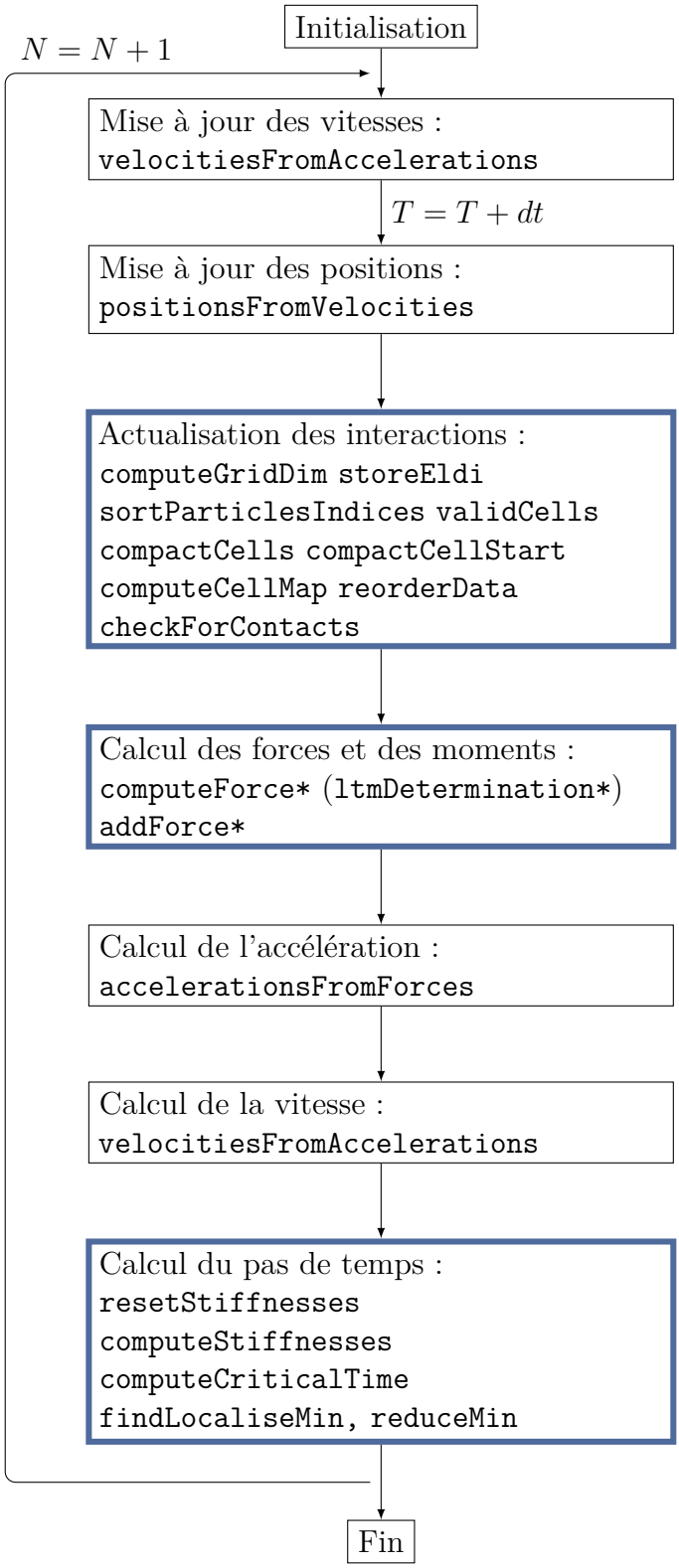


FIGURE 2.5 – Schéma de la simulation avec le nom des kernels GPU utilisés pour les différentes étapes.

un ensemble de threads d'un même *block* exécutent les mêmes instructions. Pire, les blocks sont divisés en *warp*, un ensemble de threads ordonnancés simultanément. Tout branchement conditionnel à l'intérieur du code entraînera une attente d'une partie des travailleurs. C'est une première difficulté par rapport à des codes de dynamique moléculaire ou de fluides par SPH dans lesquels tous les éléments suivent le même chemin en ce qui concerne le flot d'exécution du programme.

Comment faire ?

Plutôt que de stocker toutes les interactions pêle-mêle dans une même liste, les interactions sont organisées sur le GPU selon leurs types. Ainsi, les travailleurs suivent tous des chemins proches, notamment lors du calcul des forces. Il y a un deuxième intérêt à cette réorganisation : les paramètres inutiles pour certains types d'interaction ne sont pas conservés. Pour aller un peu plus loin, nous avons également rassemblé les paramètres en structure selon leur utilisation (lecture, écriture et à quel moment) (figure 2.6).

Les interactions de cohésion (*liens*) sont calculées en phase d'initialisation de la simulation à partir des propriétés stockées dans les structures CPU (figure 2.4). Ces interactions suivent des lois différentes selon les éléments. Leurs caractéristiques peuvent être conservées avec les paramètres de l'interaction. Les nouvelles interactions créées par le rapprochement d'éléments en cours de simulation (*contacts*) ne dépendent pas, elles, du type des éléments en interaction. Il n'est ainsi pas nécessaire de stocker toutes les caractéristiques des éléments discrets sur le GPU.

Un ED requiert maintenant 64o en mémoire et une interaction 260o ou 332o. Le gain par rapport à la version CPU (section 2.2) est (malheureusement !) compensé par l'ajout de structures supplémentaires notamment pour le stockage des interactions établies de chaque particule.

2.3.3 Précautions en mémoire

Nous avons mentionné en introduction l'importance de prendre ses précautions pour accéder à la mémoire du GPU.

Pour travailler efficacement sur les données stockées dans la mémoire globale du GPU, les accès à cette mémoire doivent être *fusionnés*En pratique, les threads d'un même bloc doivent accéder à des adresses mémoires contiguës pour que la lecture (ou l'écriture) soit faite en une seule opération et ne soit pas séquentialisée. Les adresses accédées doivent également être alignées sur des multiples de `B` le nombre de threads dans un block.

Pour atteindre cet objectif, les données sur le GPU sont organisées en tableaux de structures (figure 2.7). Par exemple, l'objet GPU_Eldi qui contient la masse et l'inertie de chaque élément est stocké par blocs. C'est-à-dire que les informations concernant `B` threads sont regroupées dans une structure. Les différents champs scalaires d'une structure sont ainsi représentés en mémoire par des tableaux de taille `B`. Lorsqu'un accès est nécessaire, les adresses chargées par les threads sont contiguës. Les différents blocs sont organisés en un grand vecteur de `N/B` structures de ce type.

Au sein de chaque bloc, les threads peuvent lire en mémoire uniquement les adresses qui les intéressent (programme 2.1). `threadIdx.x` est une variable interne contenant le

GPU_Eldi
- masse
- inertie

Caractéristiques statiques des éléments.

GPU_Eldi_Raideurs
- raideur_translation_(x-y-z)
- raideur_rotation_(x-y-z)

Raideurs des éléments pour calcul du pas de temps critique.

GPU_Interaction_Résultat
- force_normale_(x-y-z)
- force_tangentielle_(x-y-z)
- point_contact_(x-y-z)
- vecteur_normal_new_(x-y-z)

Pour stocker les données de chaque lien avant écriture dans les vecteurs globaux.

GPU_Interaction_LTM
- flag_ltm
- flag_limitite_plastique
- raideur_flexion
- moment_plastique
- vecteur_normal_old_(x-y-z)
- angle_roulement_(x-y-z)

Paramètres utiles pour les interactions devant être corrigées par la Loi de Transfert des Moments.

GPU_Interaction
- indice_1
- indice_2
- rayon_1
- rayon_2
- alpha
- armature
- distance_initiale_(x-y-z)
- direction_armature_(x-y-z)
- nature

Propriétés calculées à l'initialisation et seulement lues par la suite.

GPU_Interaction_Data
- surface
- cohe
- adoucissement
- phi_int
- phi_cont
- allongement_max
- limite_traction_locale
- limite_compr_normale_locale
- distance_glissement
- raideur_normale_traction
- distance_adoucissement
- raideur_ecrouissage
- distance_sur_norme
- distance_sur_armature
- statut

Caractéristiques modifiées seulement lors du test de fracturation.

GPU_Interaction_Propriétés
- raideur_normale
- raideur_tangentielle
- vecteur_normal_(x-y-z)
- distance_équivalente
- distance_ij

Propriétés mises à jour à chaque pas de temps.

FIGURE 2.6 – Organisation des paramètres physiques pour la simulation avec le GPU.

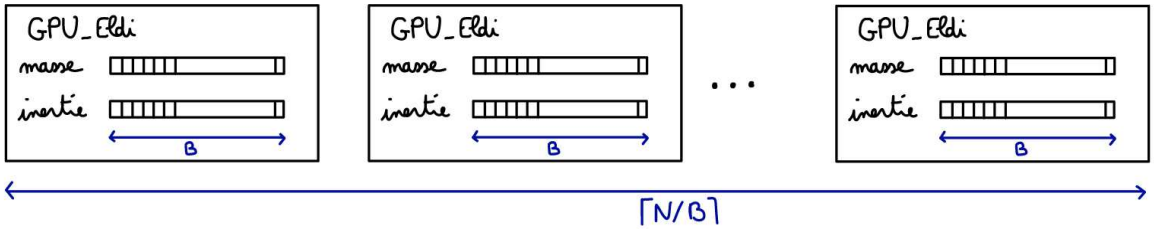


FIGURE 2.7 – Représentation du stockage d'une des structures sur le GPU.

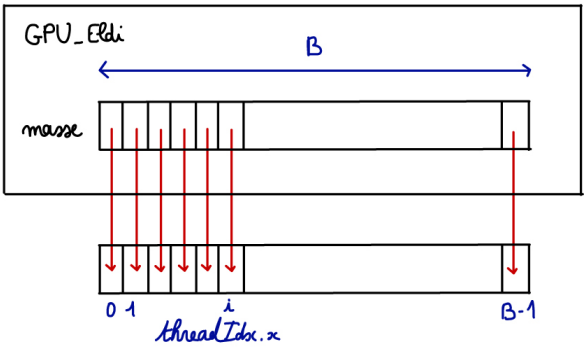


FIGURE 2.8 – Exemple d'accès aux structures de manière fusionnée.

numéro dans le bloc du thread courant. Les threads ont également accès au numéro du bloc auquel ils appartiennent, noté `blockIdx.x`. Dans ce code, les threads se positionnent sur leur bloc (ligne 4) puis accèdent chacun à une case du tableau des masses (ligne 11) puis du tableau des inerties (ligne 12). Le chargement des masses par les `B` threads d'un même bloc se fait alors en un seul accès (figure 2.8).

```
1  __global__ void acces_fusionne_kernel(const GPU_Eldi* eldis)
2  {
3    // acces a la zone memoire du bloc courant
4    const GPU_Eldi *bloc_courant = eldis + blockIdx.x;
5
6    // variables locales
7    double masse, inertie;
8
9    // acces par chacun des threads aux valeurs correspondant
10   // a son numero de thread
11   masse = bloc_courant->masse[threadIdx.x];
12   inertie = bloc_courant->inertie[threadIdx.x];
13
14   // traitement
15
16 }// acces_fusionne_kernel
```

Listing 2.1 – Organisation en mémoire pour accès fusionnés.

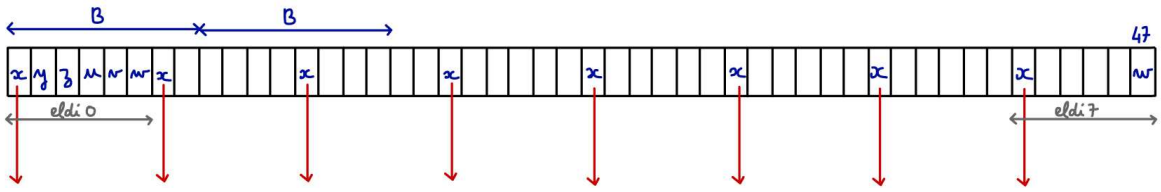


FIGURE 2.9 – Un vecteur d'état en mémoire. Si chaque thread accède aux données d'un élément différent alors les accès ne sont pas fusionnés.

Ces aménagements sont possibles uniquement pour les structures internes liées du module des éléments discrets (structures utilisées seulement pour les étapes en bleu du schéma d'intégration 2.3). Les vecteurs d'état représentant typiquement les positions $\mathbf{x}$, les vitesses $\mathbf{v}$ ou encore les forces $\mathbf{f}$ sont gérés par le simulateur. Pour ces vecteurs, les 6 coordonnées de chacun des ED sont stockées à la suite les unes des autres dans des tableaux denses (figure 2.9). Ainsi, pour un vecteur de forces, il y a $6 \times N$ coordonnées au total. Pour un bloc de threads qui traitent B éléments, $6 \times B$ coordonnées sont consultées. La taille du vecteur total est de $\lceil \frac{6 \times N}{B} \rceil \times B$ avec $\lceil x \rceil$ la plus petite valeur entière supérieure ou égale à x.

Si les threads traitent chacun un élément et en chargent les coordonnées, les accès ne seront pas fusionnés (figure 2.9). Pour éviter ce problème, une méthode classique est d'allouer de la mémoire partagée par les threads (programme 2.2, ligne 7). Au lancement du kernel, tous les threads copient en 6 accès de taille B le contenu de la mémoire globale dans la mémoire partagée (lignes 10-15). Ensuite, après une synchronisation (ligne 17) garantissant que toutes les données ont été chargées par les différents *warps* du bloc, les threads peuvent lire les données dont ils ont besoin dans la mémoire partagée (lignes 23-28).

Dans le cas du kernel `accelerationsFromForces`, il faut lire la masse et l'inertie de chaque élément, ce qui est réalisée de la manière expliquée précédemment (programme 2.1, et ici 2.2 lignes 31-34). À la fin du kernel, les accélérations locales sont copiées en mémoire partagée (lignes 48-51). Puis, après une synchronisation les threads travaillent de concert pour transférer les accélérations dans le vecteur d'état correspondant (lignes 55-60). Cette organisation assure des transferts mémoires efficaces tout en conservant des structures GPU stockées en mémoire globale cohérentes pour des communications éventuelles avec le CPU ou avec d'autres modules.

```
1  __global__ void accelerationsFromForces(int nbElements, const
       GPU_Eldi* eldis, double *forces, double *acc)
2  {
3    int iext = blockIdx.x*6 + threadIdx.x;
4
5    // reservation de memoire partagee contenant
6    // 6 * le nombre de threads coordonnees
7    __shared__ double temp[B*6];
8
9    // chargement des donnees en memoire partagee
10   temp[threadIdx.x      ] = forces[iext      ];
11   temp[threadIdx.x +   B] = forces[iext +   B];
12   temp[threadIdx.x + 2*B] = forces[iext + 2*B];
13   temp[threadIdx.x + 3*B] = forces[iext + 3*B];
```

```
14    temp[threadIdx.x + 4*B] = forces[iext + 4*B];
15    temp[threadIdx.x + 5*B] = forces[iext + 5*B];
16
17    __syncthreads();
18
19    double f[6];
20    int index6 = umul24(threadIdx.x, 6);
21
22    // lecture depuis la memoire partagee
23    f[0] = temp[index6  ];
24    f[1] = temp[index6+1];
25    f[2] = temp[index6+2];
26    f[3] = temp[index6+3];
27    f[4] = temp[index6+4];
28    f[5] = temp[index6+5];
29
30    // lecture des parametres de l'element courant
31    const GPU_Eldi *bloc_courant = eldis + blockIdx.x;
32    double masse, inertie;
33    masse = bloc_courant->masse[threadIdx.x];
34    inertie = bloc_courant->inertie[threadIdx.x];
35
36    // traitement
37    double a[6];
38    if (threadIdx.x+blockIdx.x*B < nbElements)
39    {
40            for(int i = 0; i < 3; i++)
41      {
42        a[i] = f[i]/masse;
43        a[i+3] = f[i+3]/inertie;
44      }
45    }
46
47    // ecriture en memoire partagee
48    for(int i = 0; i < 6; i++)
49    {
50      temp[index6+i] = a[i];
51    }
52
53    __syncthreads();
54
55    acc[iext      ] = temp[threadIdx.x      ]
56    acc[iext +   B] = temp[threadIdx.x +   B]
57    acc[iext + 2*B] = temp[threadIdx.x + 2*B]
58    acc[iext + 3*B] = temp[threadIdx.x + 3*B]
59    acc[iext + 4*B] = temp[threadIdx.x + 4*B]
60    acc[iext + 5*B] = temp[threadIdx.x + 5*B]
61  }// accelerationsFromForces
```

Listing 2.2 – Kernel complet : `accelerationsFromF`, illustrant la lecture et l'écriture des vecteurs d'état.

2.3.4 Calculs flottants normalisés

Pour des calculs dans lesquels la précision est critique pour la stabilité et/ou la reproductibilité, il est essentiel de prendre en considération la manière dont sont réalisées les opérations à virgule flottante.

Toutes les opérations standard ne sont pas conformes à la norme IEEE-754 dans l'implantation de CUDA [NVIDIA, 2011]. Cependant, pour la plupart des opérations il est possible de se restreindre à des opérateurs conformes.

En pratique, cela revient par exemple à éviter la combinaison des multiplications et des additions en une seule instruction qui ne respecte pas la norme car le résultat intermédiaire est tronqué. Des fonctions spéciales, en simple précision (resp. double précision) `__fadd_rn` et `__fmul_rn` (resp. `__dadd_rn` et `__dadd_rn`), permettent d'empêcher le compilateur d'utiliser cette combinaison. Certaines opérations avec arrondi respectant la norme IEEE-754 ne sont disponibles en double précision que sur les cartes de capacité supérieure ou égale à 2.0. Sur les GPUs de capacité inférieures, seules les versions non conformes avec la norme IEEE-754 sont disponibles.

2.3.5 Calcul des forces

Puisque nous avons séparé les informations concernant les ED en structures cohérentes, différents kernels sont utilisés pour le calcul des forces correspondant aux différents types d'interaction. Cette organisation permet d'exploiter au mieux la puissance parallèle du GPU en diminuant le nombre de branchements divergents.

Dans une première étape, les forces relatives à chaque interaction sont déterminées pour chacun des types d'interaction (figure 2.5, page 35) : `computeForceConcrete` pour les liens entre deux éléments de type béton, `computeForceSteal` pour les éléments acier, `computeForceBima` pour les liens entre un élément béton et un élément acier. Pour les contacts, `computeForceContact` est utilisé quelques soient les types des deux éléments.

Chaque thread calcule la force à appliquer sur les deux éléments concernés par une interaction. Ce résultat est écrit en mémoire partagée puis tous les threads d'un bloc se synchronisent pour recopier les forces dans un buffer contenant toutes les forces des interactions du type concerné. Les écritures en mémoire sont donc efficaces. En revanche, les chargements en mémoire comprennent la lecture des positions des particules qui ne sont pas ordonnées de manière cohérente avec les interactions.

Lors du calcul des forces, les interactions peuvent être rompues. Elles ne sont pas retirées pour limiter les changements de la structure mais il est nécessaire de les invalider afin que les forces correspondantes soient nulles dans la suite des calculs.

Pour tous les éléments aciers, il est nécessaire d'effectuer des calculs supplémentaires pour prendre en compte le transfert du moment le long des armatures. Ces calculs sont possibles lorsque les forces et les moments à appliquer ont été déterminés pour chaque interaction impliquant des éléments acier (kernel `ltmDetermination`).

Les forces sont ensuite accumulées sur les EDs concernés à l'aide d'un nouveau kernel générique, `addForce`, qui est appliqué aux différents types un à un. Chaque thread

parcourt la liste des interactions d'un ED et accumule la force subie par l'élément pour chacune de ses interactions. La méthode est adaptée de méthodes déjà validées dans d'autres modules du logiciel Sofa réalisant le calcul de champ de forces d'éléments finis tétrahédriques.

[Shigeto and Sakai, 2011] ont expliqué accumuler les forces sur les éléments concernés en une seule étape. Cela suppose d'utiliser des opérations atomiques qui sont coûteuses et ne devraient être utilisées qu'avec parcimonie.

2.3.6 Calcul du pas de temps

Pour définir le nouveau pas de temps, la première étape (`resetStiffnesses`) est de remettre à zéro les raideurs critiques calculées au pas de temps précédent. Ce genre d'opération est simple sur le GPU, des fonctions classiques de gestion de la mémoire comme `memset` étant fournies par le language CUDA. Le kernel GPU `computeStiffnesses` calculant les nouvelles raideurs critiques fonctionne avec un thread par élément. Chaque élément parcourt la liste de ses interactions et calcule une raideur par composante en utilisant les raideurs normales et tangentielles ainsi que le vecteur unitaire de l'interaction. Ici, au lieu d'utiliser des opérations atomiques, nous effectuons les calculs deux fois par interaction. Les calculs de cette étape sont peu nombreux et nécessitent peu de registres. Aussi il est plus efficace de privilégier la régularité des opérations.

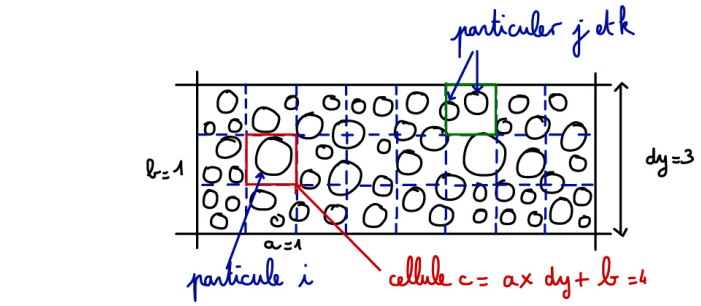
Le kernel `computeCriticalTime` permet de calculer les temps critiques correspondant à chaque degré de liberté (6 par élément) de tous les éléments de la manière expliquée précédemment (section 2.1.1). Ils sont stockés dans un buffer de taille $6 \times N$. Une recherche de minimum est ensuite exécutée pour localiser le degré de liberté qui contraint le pas de temps et récupérer la valeur de ce dernier. Nous utilisons la librairie **CUDPP** qui propose une interface de recherche de minimum via deux kernels GPU (`findLocaliseMin` et `reduceMin`).

2.3.7 Détection de collision par compactage des cellules

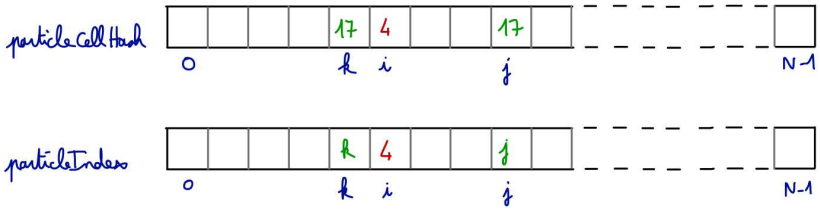
Nous avons présenté comment les différentes étapes du schéma d'intégration sont implantées sur le GPU. Il en manque une : la détection de collision.

L'étape de détection de collision fait intervenir plusieurs kernels CUDA. Un premier kernel, `computeGridDimensions`, calcule les dimensions de la boîte englobant l'ensemble des EDs du système simulé (figure 2.10(a)). Cela est fait par une recherche parallèle des minima et maxima des positions physiques des EDs. Cette donnée permet de déterminer le nombre de cellules dans les trois directions de la grille de détection.

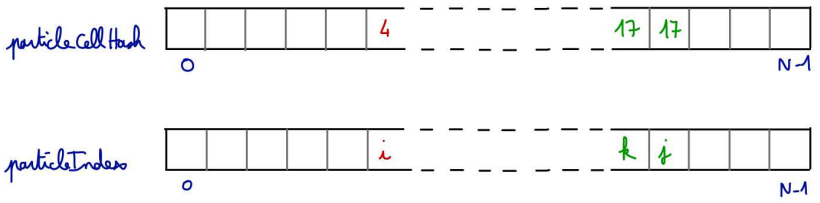
Ensuite, `storeEldi` détermine pour chaque ED à quelle cellule il appartient en fonction de sa position dans l'espace (figure 2.10(b)). Les numéros de cellules ainsi que les identifiants des EDs sont stockés en mémoire. Les indices des particules sont triés en fonction de leur indice de cellule (`sortParticlesIndices`) (figure 2.10(c)). Nous appliquons sur le tableau dense des indices de particules le tri de la librairie **CUDPP** [Satish et al., 2009].



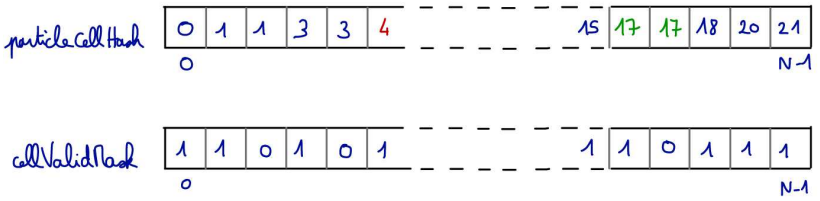
(a) `computeGridDim`



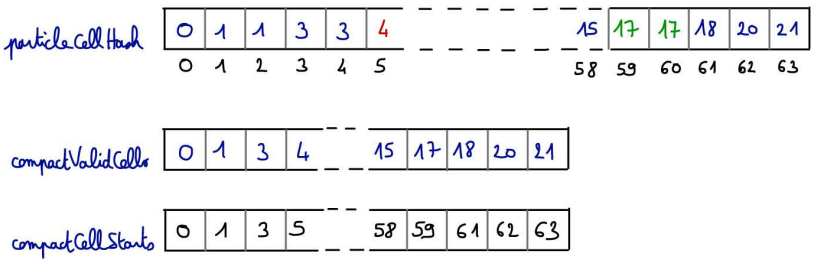
(b) `storeEldi`



(c) `sortParticlesIndices`



(d) `validCells`



(e) `compactCells` et `compactCellStart`

FIGURE 2.10 – Étapes de la détection de collision : calcul des dimensions de la boîte englobante (a), détermination du numéro de cellule de chaque élément (b). Tri des indices et des numéros de cellule dans l'ordre des numéros de cellule (c). Détermination des cellules valides (d). Compactage des cellules et des débuts de cellule valides (e).

Puisque nous ne souhaitons pas dépendre du domaine de simulation, il est primordial de déterminer les cellules qui sont occupées. Cette information est obtenue en parcourant la liste triée précédemment. Une nouvelle cellule commence lorsque l'indice de cellule change entre deux particules consécutives. Un kernel permet de créer un masque des changements des numéros de cellules (`validCells`). Un tableau contenant l'indice des cellules non vides est créé, avec un bitmask pour les repérer (sous la forme d'un tableau d'entiers) et l'emplacement de la première particule de chaque cellule est stocké (figure 2.10(d)). Les structures nécessaires pour cette opération ont la taille du nombre de particules et ne dépendent pas du nombre de cellules dans l'espace simulé. Le postulat est qu'il ne peut pas exister plus de cellules occupées que de particules.

Une opération de compactage est appliquée pour rassembler les cellules utiles et leur point de départ (`compactCells`, utilisant la fonction `compact` de la librairie **CUDPP**) (figure 2.10(e)).

Pour rendre plus efficace la phase de recherche, une carte de cellules voisines est établie pour chaque cellule (`computeCellMap`). Pour les 13 cellules voisines d'une cellule donnée, les threads vérifient que la cellule n'est pas en dehors de la grille puis cherchent si elle est occupée. Cette étape fonctionne avec une recherche dichotomique parmi les cellules occupées, ce qui va générer des branchements divergents au sein du kernel. Grâce à la conservation de la carte de cellules voisines, cette phase est factorisée.

Pour améliorer la localité mémoire lors du calcul, les positions et les rayons des EDs sont triés dans le même ordre que les identifiants des cellules des EDs (`reorderData`), comme dans le code de démonstration de simulation de particules de NVIDIA [Green, 2010].

Ensuite seulement le kernel `checkForContacts` gère la détection des nouveaux contacts. La recherche est parallélisée par ED. Chaque thread s'occupe d'effectuer les tests de candidats potentiels présents dans sa cellule et dans les cellules voisines. Pour éviter les calculs redondants, seulement les EDs avec des identifiants supérieurs à celui de l'élément courant sont testés dans la cellule courante et seule la moitié des cellules voisines sont testées. Cela permet de ne vérifier la nouveauté de l'interaction que dans la liste des interactions établies lors des recherches précédentes. Lorsqu'une nouvelle interaction est détectée elle est poussée dans une liste globale par bloc de threads, maintenue par un incrément atomique.

Pour limiter le nombre de réallocations mémoires, les contacts sont alloués par bloc. Il n'était pas possible avec les cartes de capacité inférieure à 2.0 d'allouer de la mémoire GPU depuis un kernel. A l'époque il fallait ainsi transférer la liste des nouveaux contacts sur le CPU afin d'allouer éventuellement de nouveaux blocs de mémoire pour les nouvelles interactions. Les nouvelles interactions sont ainsi créées par le CPU. Les structures sont ensuite transférées de nouveau dans la mémoire du GPU afin que le calcul puisse reprendre comme si de rien n'était.

Nous avons maintenu une compatibilité avec la carte de capacité 1.3 que nous avons utilisée pour le développement du code. Cependant à partir des cartes de capacité 2.0, les allocations dynamiques de mémoire sont possibles à l'intérieur d'un kernel. Cette partie pourrait tirer partie d'une réécriture utilisant cette fonctionalité pour ainsi limiter le nombre de transferts mémoires.

2.4 Résultats

2.4.1 Efforts et validation

Saperlipopette ! Bien que la programmation des GPUs se soit démocratisée grâce à des langages comme CUDA ou OpenCL, il faut prévoir une bonne dose de persévérance lorsque l'application visée est plus alambiquée que les applications jouets des démos.

Ce doux euphémisme énoncé, notre cible demeure la simulation d'impacts sur des murs par exemple des enceintes de centrales nucléaires. Ainsi, devons-nous pouvoir garantir la validité des résultats obtenus.

Dans notre cas, la validation du code GPU a nécessité le développement de tout un éventail de tests unitaires de manière à certifier l'application de lois physiques correctes pour les différents types d'interactions. En ce qui concerne les simulations plus conséquentes, la validation peut se faire en suivant l'évolution dans le temps de quantité scalaire comme l'énergie, mais également par la résultante de la force sur tout ou partie des éléments simulés. Il est aussi possible de comparer visuellement l'état des structures à différents instants lors du déroulement des expériences.

Malheureusement, nous avons rencontré de grosses difficultés pour reproduire le comportement du code séquentiel avec le code parallèle sur GPU. Tout ce qui a pu être facilement testé a été validé mais il reste néanmoins quelques divergences lors d'expériences avec un grand nombre d'éléments dont la source est délicate à déterminer.

Les problèmes rencontrés n'entravent pas la conduite d'expériences pour déterminer le gain de performance sur lequel compter. En effet, nous avons pris soin de vérifier que les calculs sont corrects suffisamment longtemps pour permettre des comparaisons représentatives entre les deux architectures.

2.4.2 Apport du GPU

Nous avons mesuré la performance de la version GPU dans le cadre de la simulation d'un impact sur une dalle de béton armé (figure 2.11). La scène est constituée de 14 274 éléments discrets dont 784 pour représenter les armatures en acier, 12 718 pour le béton de la dalle et 772 pour représenter l'impacteur. 79 211 liens sont créés pour représenter la cohérence de la matière et plusieurs dizaines de milliers de contacts sont générés pendant la simulation.

La carte utilisée est une NVIDIA Tesla C2050 avec 448 coeurs à 1.147GHz et 2.6Go de mémoire. Cette carte, de capacité 2.0 était utilisée avec Cuda 3.0. Le CPU est un 6-coeurs Intel Xeon X5650 @2.67GHz dont un seul coeur est utilisé pour la simulation.

Nous avons mesuré le temps d'exécution des itérations au cours de la simulation (figure 2.12). Les pics réguliers correspondent aux pas de temps avec détection de collision (toutes les 10 iterations). Au début de la simulation (figure 2.12(a)), les temps d'exécution sont constants car l'impacteur ne touche pas encore la dalle. Autrement dit, peu de changements se produisent, et aucun nouveau contact n'est créé. La détection de collision est à ce stade 10 fois plus coûteuse que le reste des opérations d'un pas de temps sur le CPU (6 fois sur le GPU).

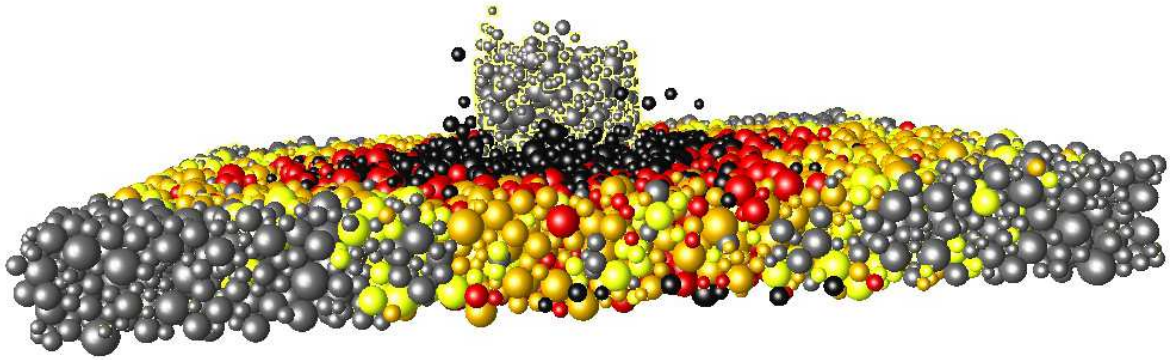
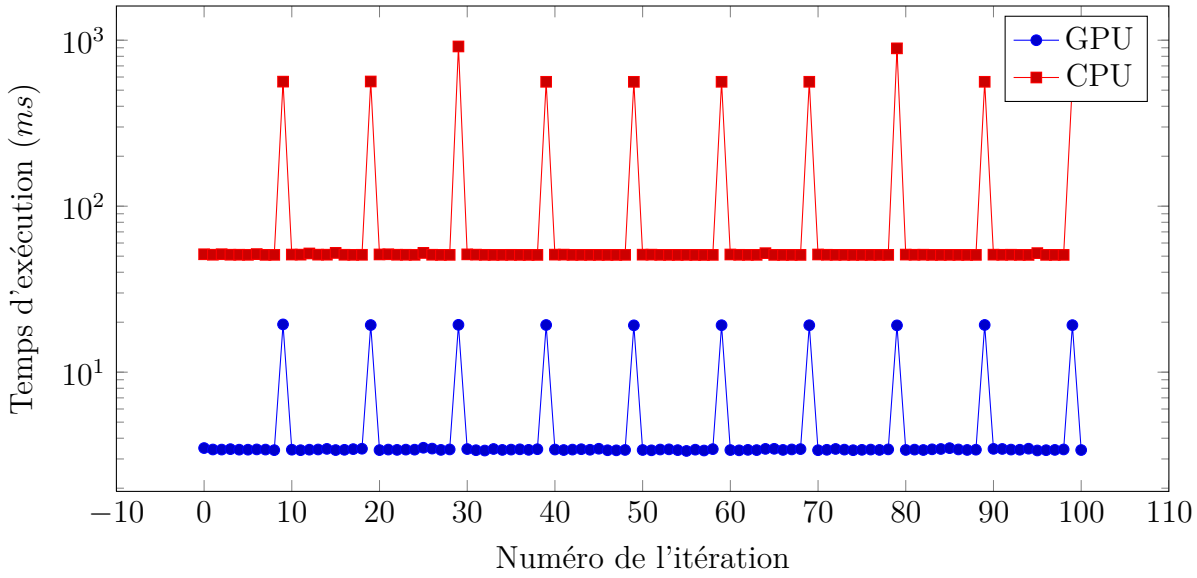


FIGURE 2.11 – Simulation d'un impacteur lancé à vive allure contre une dalle en béton armé. L'image a été prise au pas de temps 2330, durant l'impact lui-même. 14 274 éléments discrets sont utilisés dans cette scène de démonstration. Les couleurs servent à représenter l'endommagement des éléments : en noir, les particules dont toutes les interactions se sont rompues. Du rouge vers le jaune en passant par la couleur orange de plus en plus de liens sont encore actifs. Les éléments en gris clair n'ont aucun lien cassé.

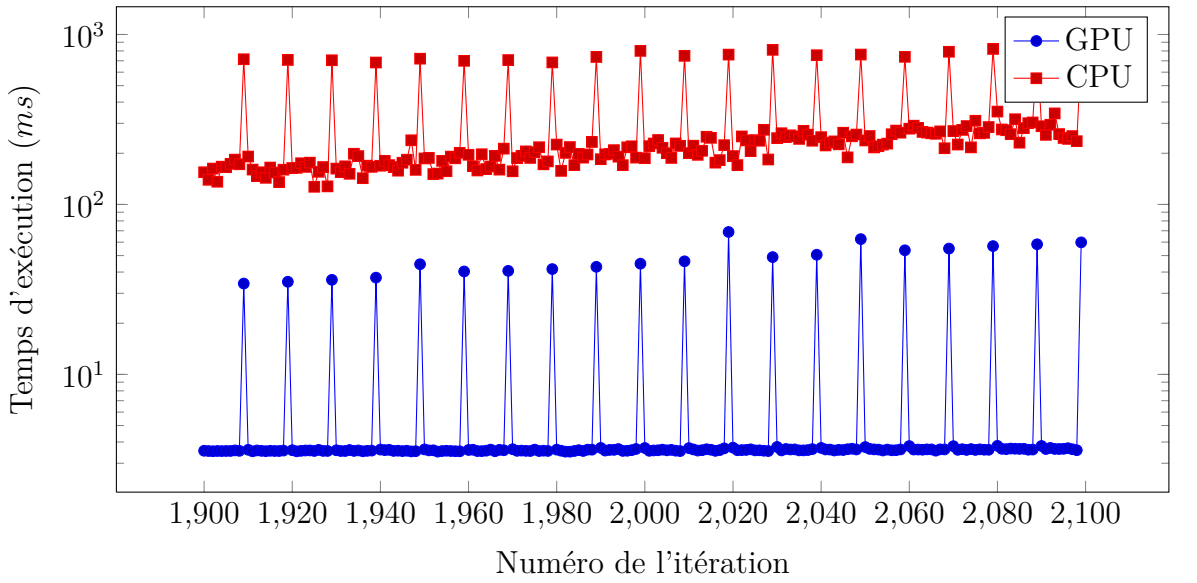
Par rapport à une exécution séquentielle sur 1 coeur CPU, l'accélération de la version GPU est de l'ordre de 15 lorsque la détection de collision n'est pas exécutée, de l'ordre de 30 sinon.

Les résultats en phase d'impact sont de même présentés pour décrire l'évolution de la simulation (figure 2.12(b)). Le temps d'exécution moyen des pas de temps sans détection de collision augmente sur la version CPU. Dans la zone de l'impact, les particules sont plus nombreuses, générant de nombreux contacts supplémentaires. Aussi le calcul des forces est-il plus long. Avec la version GPU, le temps d'exécution augmente de 10% seulement. Sur le GPU, les structures de données contenant les liens de cohésion ne sont pas modifiées, seuls les liens cassés sont invalidés dans la structure ce qui n'apporte pas de modification fondamentale. En revanche, des contacts peuvent être créés, ce qui implique des échanges de données entre le CPU et le GPU pour l'allocation de mémoire supplémentaire et la mise à jour des contacts. C'est pour cela que le temps d'exécution des iterations avec détection de collision augmente progressivement. En début d'impact les contacts créés ne sont pas assez nombreux pour influencer le temps d'exécution du calcul des forces, d'où le fait que le temps d'exécution des itérations sans détection de collision reste constant.

Le temps total de simulation de 2 310 itérations est de 320 ± 5s sur le CPU et de 16.5 ± 0.5s avec la version GPU. L'accélération moyenne mesurée est de l'ordre de 20. Pour apprécier le potentiel du GPU par rapport aux capacités des processeurs multi-cœurs, nous pouvons faire l'estimation suivante. En misant sur une accélération de 4 à 10 sur de 8 à 12 cœurs, par exemple, l'accélération du GPU serait comprise entre 5 et 2.



(a) Après 100 pas de temps, avant l'impact.



(b) Après 2 000 pas de temps, l'impacteur a commencé à s'écraser contre la dalle.

FIGURE 2.12 – Temps d'exécution par itération avec le GPU et le CPU. Les pics réguliers correspondent aux iterations où l'étape de détection de collision est réalisée.

Étapes	CPU (ms)	% CPU	GPU (ms)	% GPU
Réinitialisation forces internes	.	.	0,11	3,1%
Calcul des forces	36,5	76,7%	0,92	26 %
Loi de transfert du moment	1,15	2,4 %	0,15	4,15 %
Accumulation des forces	0,38	0,8 %	0,7	20,2%
Réinitialisation des raideurs	0,67	1,4%	0,13	3,7 %
Calcul des raideurs critiques	7,1	14,9 %	1,25	35,5%
Calcul du nouveau pas de temps	1,8	3,71%	0,26	7,5%

TABLE 2.1 – Temps d'exécution (ms) et pourcentage correspondant des principales étapes au début de la simulation, sans détection de collision.

Étapes	CPU (ms)	% CPU	GPU (ms)	% GPU
Réinitialisation forces internes	.	.	0,11	3%
Calcul des forces	300,6	96%	1,14	29,7 %
Loi de transfert du moment	1,54	0,5%	0,3	7,8 %
Accumulation des forces	0,37	0,12 %	0,74	19,3%
Réinitialisation des raideurs	0,67	0,22%	0,13	3,5 %
Calcul des raideurs critiques	8,1	2,58 %	1,15	29,8%
Calcul du nouveau pas de temps	1,8	0,57%	0,27	7%

TABLE 2.2 – Temps d'exécution (ms) et pourcentage correspondant des principales étapes de la simulation au moment de l'impact, sans détection de collision.

2.4.3 Dans les entrailles de la version GPU

La comparaison du temps passé dans les différentes étapes au début de la simulation (tableau 2.1) puis au moment de l'impact (tableau 2.2) confirment l'augmentation du temps de calcul des forces pour la version CPU.

Avec la version GPU, les étapes les plus coûteuses sont le calcul des forces et l'accumulation des forces, d'une part et le calcul des raideurs critiques d'autre part, lorsque la détection de collision n'est pas considérée. Une métrique importante pour évaluer la performance d'une implantation GPU est le taux d'occupation des kernels (tableau 2.3). Il s'agit du rapport entre le nombre de threads pouvant être exécutés simultanément pour le calcul d'un kernel par rapport au nombre maximum de threads sur l'architecture. Ce taux peut être estimé au préalable en fonction du nombre de threads par bloc, de la taille demandée en mémoire partagée, du nombre de registres par thread. Par exemple, les kernels qui traitent le calcul des forces par type d'iteration sont limités par la taille de la mémoire disponible par thread. Le nombre de registres maximum est atteint. Cela impacte le nombre de blocs pouvant être ordonnancés simultanément et au final, le temps d'exécution.

D'autres données de profiling comme l'efficacité des chargements (load %) et/ou des écritures (store %) en mémoire permettent de montrer l'intérêt d'utiliser des structures organisées de manière à favoriser les accès fusionnés à la mémoire globale. Le taux d'efficacité des branchements conditionnels (branch %) diminue lorsque le nombre de branchements divergents augmente. Ainsi pouvons-nous confirmer l'intérêt du découpage

Kernel	GPU (ms)	*occ.* (%)	*reg.*	*branch* (%)	*store* (%)	*load* (%)
`computeForceCoheSteel`	0,04	15	63	97,78	102,6	47,6
`computeForceCoheConcrete`	0,585	24,6	63	99,2	102,4	25,5
`computeForceBima`	0,197	23,3	63	96,8	99,4	44,8
`ltmDetermination`	0,03	14,7	63	96,884	39,7	20,6
`addForceSteel`	0,095	25,9	25	99,993	99,7	65,8
`addForceConcrete`	0,354	32,9	25	99,996	100	9,2
`addForceBima`	0,18	25,9	25	99,996	100	15,9
`computeStiffnessesSteel`	0,062	27,1	37	99,988	100	46,4
`computeStiffnessesConcrete`	0,899	32,7	37	99,996	100	8,16
`computeStiffnessesBima`	0,172	24,6	37	99,995	100	16,7
`computeCriticalTime`	0,012	30,4	34	99,963	99,9	5,
`findLocaliseMin`	0,025	10,7	12	100	100	0,4
`reduceMin`	0,005	3,8	16	94.737	78,8	5,1
`accFromF`	0,019	30,3	26	99,944	100	100
`findLocaliseMinMax`	0,022	7,3	26	100	25	21
`reduceMinMax`	0,01	3,2	32	90,9	25	26,3
`storeEldi`	0,02	30,6	34	99,989	99,96	41,2
`validCells`	0,005	28,1	8	74,958	13,9	66,3
`compactCells`	0,01	13,7	10	99,9	30,3	38,9
`computeCellMap`	0,072	6,1	18	71,9	25	96,8
`reorderData`	0,035	31	14	99,888	37,5	7,8
`checkForContacts`	4,7	7,7	46	59,91	0	23,4

TABLE 2.3 – Principaux kernels avec leur temps d'exécution (ms), le taux d'occupation atteint (*occ.* %), le nombre de registres par thread (*reg.*), l'efficacité des branchements (*branch* %), l'efficacité des opérations d'écriture en mémoire globale (*store* %) et l'efficacité des opérations de lecture en mémoire globale (*load* %).

en kernels adaptés aux types de traitement à effectuer et des structures de données utilisées qui permettent extraire un maximum de parallélisme pour quasiment tous les kernels.

Le kernel le plus coûteux pour ce qui est du temps d'exécution est `checkForContacts`. Les lectures et écritures en mémoire sont peu efficaces et de nombreux branchements divergents pénalisent l'exécution. Dans la cellule courante, seules les particules situées après la particule courante sont testées. Autrement dit, pour une même cellule, tous les threads ne feront pas le même nombre de tests. De plus, les accès pendant cette phase ne sont pas alignés. Lors du parcours des cellules voisines, les particules lues par les threads d'une même cellule sont les mêmes à chaque itération. En revanche, les particules d'un même bloc peuvent appartenir à plusieurs cellules ou les particules d'une même cellule peuvent être traitées par différents bloc contigüs. Ainsi le traitement est très irrégulier, dégradant les performances.

2.4.4 Grille de détection sans limites

Dans le cadre de la simulation présentée dans cette étude, des éléments sont éjectés par la violence de l'impact. Sans traitement particulier, ces éléments font s'accroître les dimensions de la grille de détection. Par exemple, pour la simulation de la dalle de 14 274 éléments, au début de la simulation 1 365 cellules sont occupées par des particules alors que la grille totale contient 2 205 cellules. Après l'impact, la grille peut contenir plusieurs centaines de milliers de cellules, voire plusieurs millions. Il est alors avantageux de ne pas stocker toutes les cellules comme dans les méthodes classiques même s'il ne s'agit que d'une ou de deux valeurs par cellule de la grille complète. Avec la technique que nous avons proposée (section 2.3.7), les structures ont la taille du nombre d'éléments quelque soit le domaine de simulation.

Les étapes spécifiques à l'utilisation de cette technique sont le compactage des cellules et la création de la carte de cellules voisines pour chaque cellule. Ces étapes pèsent pour quelques pourcents seulement de la phase de détection. En outre, la préparation de la carte des cellules voisines permet de factoriser le coût de la recherche des cellules voisines, augmentant la rapidité d'exécution de la recherche. Cette technique pourrait être évaluée également dans le cadre des méthodes classiques sans compactage des cellules valides.

2.5 Discussion

Nous avons réalisé l'implantation sur GPU d'une simulation physique basée sur des éléments discrets. L'utilisation des GPUs était alors nettement moins développée dans ce domaine que dans celui, proche, de la dynamique moléculaire. Nous avons également proposé une technique pour limiter l'espace mémoire nécessaire lors de la détection de collision.

L'effort de développement nécessaire pour tirer partie du GPU est conséquent. La mémoire est limitée et les transferts doivent être gérés explicitement, voire réorganisés

pour ne pas pénaliser le temps d'exécution. La pression sur la taille de la mémoire partagée et le nombre de registres disponibles affecte l'efficacité des programmes. Nous avons porté une attention spécifique aux structures de données afin d'extraire le maximum de parallélisme dans les différentes étapes, notamment en limitant le nombre de branchements divergents.

Nous avons évalué notre mise en œuvre et affiché des accélérations moyennes de l'ordre de 20. Les performances obtenues sont plus modérées que sur des démonstrations jouets à cause de la gestion de différents types d'objets et de la quantité de données à manipuler. Les gains obtenus peuvent être concurrencés par une programmation parallèle plus classique sur multi-cœurs. Considérez l'effort à fournir pour chacune de ces approches : l'intérêt du GPU semble alors limité.

Nous avons mis en pratique notre code en exécutant une scène de près de quinze mille éléments. Nous avions à notre disposition une scène constituée de 261 323 éléments et de plus d'un million de liens cohésifs. Mais de l'ordre de 5Go de mémoire sont nécessaires pour une telle simulation. Ainsi n'avons-nous pas pu l'exploiter, le GPU utilisé présentant seulement $2,6$Go. Il aurait été intéressant de disposer d'une simulation de taille intermédiaire. L'utilisation du GPU pour des simulations d'envergure industrielle pourrait néanmoins être évaluée en utilisant des cartes ayant plus de mémoire comme certaines de la série Kepler.

Cependant une meilleure option serait de tirer partie des architectures multi-GPUs en distribuant les calculs sur plusieurs cartes. Cela suppose de supporter une décomposition du domaine de simulation mais permet de bénéficier de plus de mémoire. De meilleures accélérations pourraient encore être obtenues sur des plateformes hétérogènes multi-CPU/multi-GPU grâce à une exploitation simultanée des deux types de coeurs [Hermann et al., 2010]. L'idée est de combiner les deux architectures pour effectuer sur le GPU uniquement les étapes pour lesquelles il est le plus efficace.

S'intéresser à l'organisation des données en mémoire pourrait également permettre d'améliorer les performances. Il conviendrait d'évaluer l'apport potentiel du tri des particules en mémoire en utilisant une courbe en Z ou une courbe de Hilbert [Anderson et al., 2008]. Une étude serait nécessaire pour déterminer si toutes les données doivent être triées et à quelle fréquence. L'intérêt est de limiter les accès aux données lointaines afin d'améliorer l'efficacité des lectures et écritures en mémoire, notamment lors de la détection de collision.

Malgré l'accélération obtenue sur GPU, la détection de nouvelles interactions est onéreuse par rapport à l'ensemble de la simulation : près d'un quart du temps total alors qu'elle n'est exécutée que tous les 10 pas de temps. L'idéal serait de limiter le nombre de paires testées, d'éviter la reconstruction complète des structures nécessaires et de maintenir une bonne localité des éléments en mémoire.

Nous avons décidé de ne pas pousser plus loin les développements GPU pour nous focaliser sur des problématiques plus fondamentales visant à répondre à ce besoin. Nous nous sommes intéressés à des structures de données *Cache-Oblivious* utilisables pour maintenir triées des particules en mouvement.

III

Vers des structures adaptatives

Parfois je rêve, je divague, je vois des vagues
Et dans la brume au bout du quai
J'vois un bateau qui vient m'chercher

Pour m'sortir de ce trou, où je fais des trous
Des p'tits trous, des p'tits trous, toujours des p'tits trous

S. Gainsbourg, *Le Poinçonneur des Lilas*.

Jeux avec des trous 3

La partie précédente nous a plongé dans un univers de particules en mouvement. Nous avons cherché à tirer parti d'une architecture particulière pour diminuer le temps d'exécution des simulations physiques. Nous avons détaillé les différentes phases de calcul et expliqué les méthodes utilisées pour les rendre plus efficaces. La détection de collision est l'un des plus gros centres de coût dans le cas des simulations particulaires. Les méthodes proposées utilisent généralement une décomposition en grille régulière. En raison du coût de leur mise à jour, les structures de données sont recréées à chaque étape. Pourtant, du fait des contraintes sur la stabilité numérique, les pas de temps sont petits et les changements à appliquer peu nombreux.

Comment tenir compte de cette cohérence temporelle ? Sans pénaliser la localité spatiale nécessaire à l'efficacité des parcours des structures, naturellement !

Regrouper en mémoire les particules proches dans l'espace permet de diminuer le nombre d'accès mémoire. Les trier en utilisant une courbe de recouvrement de l'espace de type courbe en Z favorise encore la proximité spatiale (figure 3.1). Pour suivre le déplacement des éléments, l'ordre des données doit cependant être maintenu tout au long de l'exécution.

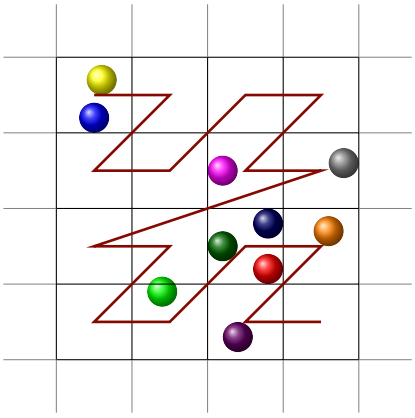


FIGURE 3.1 – Particules plongées dans une grille de détection. Les segments représentent la fonction de recouvrement de l'espace dite courbe en Z.

Nous nous intéressons à la dynamisation des structures de données en s'appuyant sur des schémas amortis afin de maintenir les éléments triés. Après une présentation des travaux liés au tri de listes presque triées (section 3.1), nous étudions l'introduction de trous dans cette situation du maintien de l'ordre entre les éléments (sections 3.2 et 3.3). Nous présentons PaVo, un algorithme original et efficace pour traiter ce problème (section 3.4).

3.1 Un problème de tri

La complexité algorithmique optimale du tri par comparaison de N éléments est $\Omega(N \log N)$. Cette borne inférieure est valable lorsqu'aucune hypothèse n'est faite sur les éléments à trier.

Dans les applications pratiques, il est fréquent que les données en entrée soient presque triées, justifiant le développement d'algorithmes adaptés (section 3.1.1). Si les algorithmes proposés sont efficaces du point de vue du nombre de comparaisons, ils ne le sont pas tous vis-à-vis du nombre d'accès mémoire. Différents modèles de mémoire peuvent être utiles pour analyser les algorithmes et les structures de données (section 3.1.2). Par exemple, modifier un algorithme pour mieux utiliser les caches permet généralement d'améliorer la performance pratique.

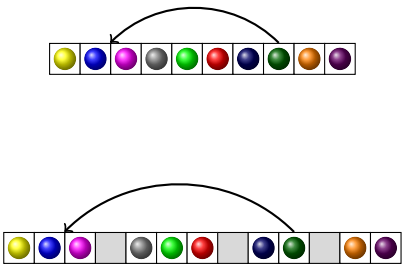


FIGURE 3.2 – Déplacer une particule coûte cher dans un tableau dense : et si nous ajoutions des trous ?

Nous nous intéressons dans ce chapitre à une structure nommée *Packed-Memory Array* (PMA), une solution au dilemne entre l'optimisation de la complexité algorithmique et celle du nombre d'accès mémoire. L'idée de base est simple : un tableau dense est efficace d'un point de vue des défauts de cache mais les déplacements d'éléments sont coûteux. En effet, quelle que soit la rapidité avec laquelle la nouvelle position d'un élément est déterminée, tous les éléments situés entre la position d'origine et la position cible devront être décalés (figure 3.2). En ajoutant suffisamment de trous dans le tableau, le nombre moyen de déplacements en mémoire peut être réduit (sections 3.2, 3.3 et 3.4).

3.1.1 Tri de listes presques triées

De nombreuses études autour du tri de séquences d'éléments presque triées sont menées. Plusieurs variantes d'algorithmes utilisant le principe de l'insertion ont été proposées à la fin des années 70. En introduisant le *Local Insertion Sort*, [Mehlhorn, 1979] a apporté les premières briques nécessaires à la formalisation de ces études. Les algorithmes de tri sont dits *adaptatifs* lorsque le temps d'exécution est une fonction croissante du nombre d'éléments total et du désordre dans la séquence [Mannila, 1985]. En pratique, plus la liste d'éléments est désordonnée, plus la complexité de ces algorithmes tendra vers $O(N \log N)$. Lorsque les éléments sont quasiment correctement pré-triés, la complexité s'approche de $O(N)$.

Certains algorithmes de tri sont naturellement adaptatifs pour un ou plusieurs types de désordre. Par exemple, le principe de l'algorithme *Straight Insertion Sort* est de parcourir chaque élément et de l'insérer à sa nouvelle position. Si les éléments sont pré-triés alors l'algorithme effectue $O(N)$ comparaisons mais aucun échange de données. Le *Natural Merge Sort*, une variante de l'algorithme de tri par fusion, constitue un autre exemple d'algorithme adaptatif. Les sous-séquences déjà triées en ordre ascendant sont repérées puis fusionnées deux à deux. L'algorithme *SmoothSort*, introduit en 1982, effectue $O(N)$ comparaisons et aucun échange de données si le tableau est déjà trié [Dijkstra, 1982]. Il s'agit d'une nouvelle version de l'algorithme du tri par tas sachant tirer parti de l'ordre existant. De nombreux autres algorithmes ont été proposés : par exemple une adaptation de l'algorithme *Quicksort* [Dromey, 1984], mais aussi plus récemment *Splitsort* [Levcopoulos and Petersson, 1991].

Le concept de *presortedness*, caractère pré-trié d'une séquence, a été formalisé par [Mannila, 1985] ainsi que la notion d'optimalité d'un tri adaptatif vis-à-vis d'une mesure de désordre donnée. L'auteur a introduit 4 mesures du caractère pré-trié d'une séquence X d'éléments. $Inv(X)$ compte le nombre de paires d'éléments inversés. Par exemple, l'algorithme *Straight Insertion Sort* s'exécute en $Inv(X)+n-1$ comparaisons et $Inv(X)+2n-1$ mouvements de données. $Run(X)$ établit le nombre de sous-séquences déjà triés en ordre croissant. Typiquement, cette mesure donne directement le nombre de sous-séquences à fusionner par un *Natural Merge Sort*. Peuvent également être considérés la plus longue séquence triée $Las(X)$ ou encore $Exc(X)$, le plus petit nombre d'échanges à effectuer pour retrouver une séquence triée. D'autres métriques ont été proposées [Carlsson and Chen, 1992 ; Estivill-Castro and Wood, 1992]. Chaque mesure a son sens, généralement intuitif, et un algorithme adaptatif optimal par rapport à l'une des métriques ne l'est pas forcément par rapport à une autre. Des auteurs se sont également intéressés aux relations entre elles [Estivill-Castro and Wood, 1992 ; Petersson and Moffat, 1995]. Par exemple, $Rem(X)$ est plus contraignante que $Exc(X)$, ce qui signifie qu'un algorithme optimal vis-à-vis de *Rem* le sera aussi pour *Exc*.

[Cook and Kim, 1980] ont mené une étude expérimentale du comportement des algorithmes de tri classiques vis-à-vis de séquences presques triées. Ils introduisent aussi un algorithme adaptatif, *CKSort*, qui consiste simplement à extraire le plus petit ensemble possible rendant la liste principale triée, à trier la liste extraite avec Quicksort puis à la fusionner à la liste principale. Cet algorithme présente de bonnes performances mais n'est toutefois pas *Rem*-optimal à cause du pire cas de l'algorithme Quicksort, comme l'ont montré [Petersson and Moffat, 1995].

3.1.2 Influence de la hiérarchie mémoire

En 1988, [Aggarwal and Vitter, 1988] ont établi le coût des transferts mémoires des algorithmes de tri dans le cadre du modèle *Cache-Aware*. La complexité optimale d'un algorithme de tri en cache est $\Omega\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$, avec N le nombre d'éléments, B la taille d'un bloc de cache pouvant être lu en un accès et M le nombre de blocs. Une relation entre le nombre de comparaisons effectuées et le nombre d'accès mémoire a été prouvée pour tout algorithme travaillant sur un nombre d'éléments N qui permet de retrouver cette borne inférieure [Arge et al., 1993].

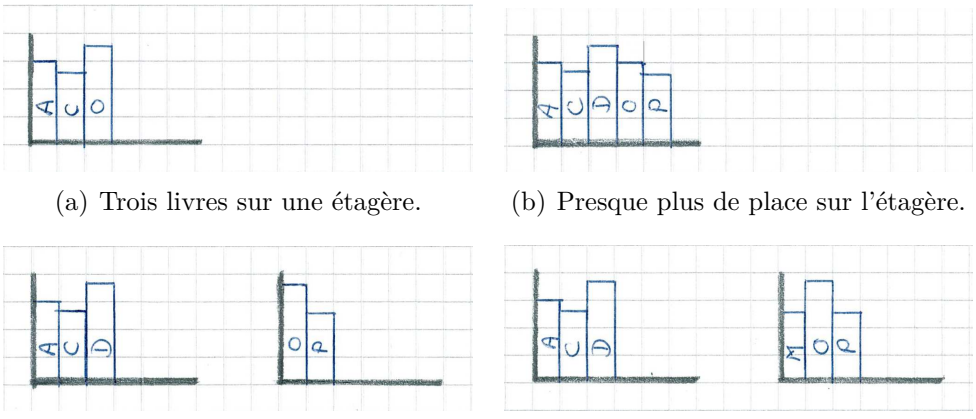
[LaMarca and Ladner, 1997] se sont intéressés à l'influence du cache sur la performance d'algorithmes de tri classiques : le tri par tas, le tri par fusion, le *Quicksort* et le *Radix Sort*. Ils ont implanté des optimisations favorables au cache et ont mesuré le gain obtenu dans les différentes versions. À titre d'exemple, l'algorithme du tri par tas, généralement basé sur un 2-tas, est modifié par un d-tas avec d le nombre de clés résidant dans une ligne de cache. L'alignement des données est également pris en compte.

D'autres travaux ont présenté l'amélioration des performances d'algorithmes de tri par diverses optimisations comme le *padding* des données de manière à limiter le nombre de conflits en cache ainsi que le nombre de TLB *misses* [Xiao et al., 2000]. Ces optimisations sont intrinsèquement liées à la hiérarchie mémoire et aux caractéristiques des caches spécifiques à une architecture donnée.

Le modèle *Cache-Oblivious* proposé par [Frigo et al., 1999] permet de s'affranchir de toute connaissance de ces paramètres liés au matériel. En utilisant ce modèle, les auteurs ont conçu *Funnelsort* et *Distribution sort*, deux algorithmes de tris basés sur le tri par fusion et le tri par distribution respectivement, tous deux optimaux en cache. Ces résultats ont été repris dans une vue d'ensemble des algorithmes *Cache-Oblivious* par [Demaine, 2002] qui s'est également intéressé aux structures de données.

En ce qui concerne les tris adaptatifs, [Brodal et al., 2005] ont introduit la notion d'I/O optimalité pour un tri adaptatif par rapport à une mesure de désordre donné. Ils ont proposé des algorithmes optimaux en cache pour le modèle *Cache-Aware* comme pour le modèle *Cache-Oblivious*. Ces algorithmes théoriques qui étendent et combinent plusieurs algorithmes sont complexes et semblent difficiles à mettre en œuvre.

3.2 Tri de bibliothèque



(a) Trois livres sur une étagère. (b) Presque plus de place sur l'étagère.

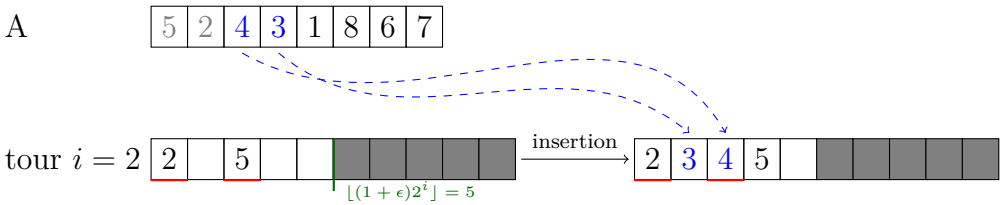
(c) Répartition des livres sur deux étagères. (d) Insertion de nouveaux ouvrages.

FIGURE 3.3 – Illustration du principe de rangement des livres dans une bibliothèque.

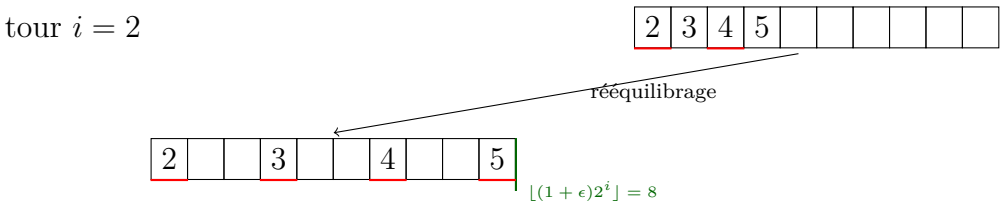
En guise d'apéritif, nous présentons dans cette section une étude préliminaire à l'intérêt des trous. Nous nous intéressons au tri de N éléments par insertion. La complexité de ce tri est $O(N^2)$ dans sa version standard puisqu'en moyenne, il faut

comparer chaque nouvel élément i avec $(i-1)/2$ éléments. Ainsi, l'algorithme est peu performant pour de grandes séquences. Il reste intéressant pour de très petites séquences grâce à son très faible surcoût et est de ce fait généralement utilisé comme terminaison d'autres algorithmes de tri. De plus, comme nous l'avons vu, c'est algorithme adaptatif bien que non optimal. [Bender et al., 2004] ont proposé une variante appelée tri de bibliothèque. Ce tri utilise un schéma permettant d'amortir les coûts algorithmiques de maintenance sur les éléments insérés et un stockage des données dans une structure à trous et atteint une complexité asymptotique en $O(N \log N)$.

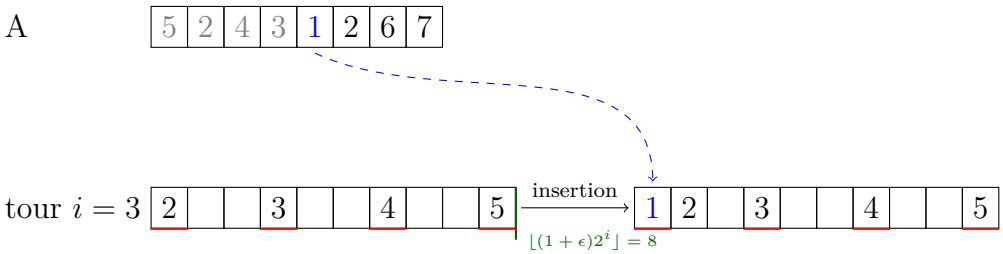
Soit un bibliothécaire dont la tâche est de ranger des ouvrages par ordre alphabétique sur des étagères. Les livres se présentent les uns après les autres. Les premiers livres sont placés sur la première étagère disponible (figure 3.3(a)). Lorque l'étagère n'a plus assez de place, alors les livres sont répartis sur deux étagères (figure 3.3(c)). En gardant de la marge sur chaque étagère il est facile d'ajouter un nouveau livre sans devoir déplacer tous les ouvrages (figure 3.3(d)). De temps en temps, il faut redistribuer les livres sur de nouvelles étagères ce qui peut prendre un peu de temps mais ce fastidieux travail n'est pas nécessaire souvent, son coût est alors amorti.



(a) Début du tour 2, insertion des éléments de valeur 4 et 3.



(b) Répartition des 2^2 éléments triés.



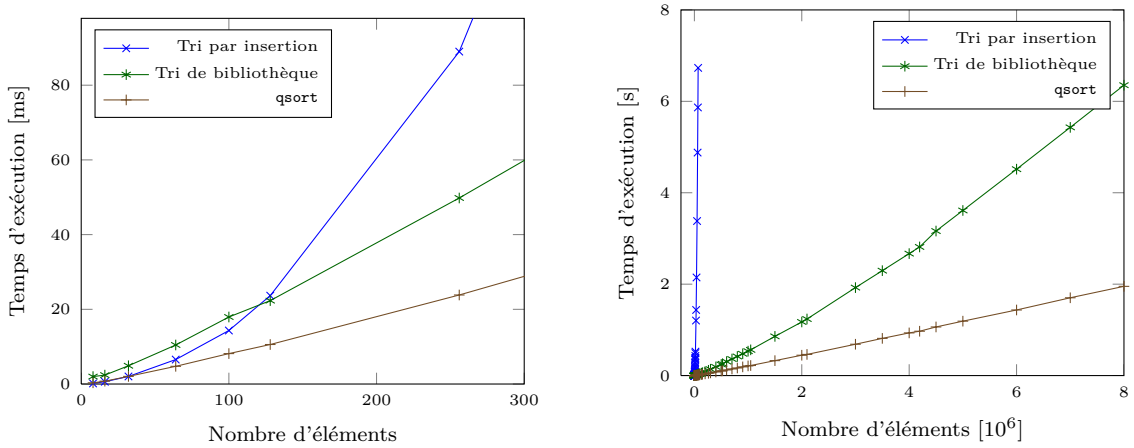
(c) Début du tour 3.

FIGURE 3.4 – Illustration de l'algorithme du tri de bibliothèque pour $N = 8$ éléments et $\epsilon = 1/3$.

[Bender et al., 2004] ont effectivement prouvé qu'en utilisant cette méthode et en redistribuant les éléments avant que l'espace ne vienne à manquer sur chaque ensemble, trier N éléments par insertion se fait en $O(N \log N)$ déplacements. Voyons comment il faut procéder en pratique. L'algorithme nécessite un tableau de tri de taille $(1+\epsilon)N$ avec $\epsilon > 0$. Le coefficient ϵ représente le taux d'emplacements laissés libres sur chacune des étagères virtuelles. Le principe est d'insérer les éléments par paquets en $\log N$ tours. À chaque tour, le nombre d'éléments insérés en même temps est doublé. Au début du i-ème tour, 2^{i-1} éléments ont déjà été insérés et répartis régulièrement dans les $(1+\epsilon)2^i$ premières positions du tableau (figure 3.4(a)). Dans la première phase du tour i, les 2^{i-1} éléments supplémentaires sont insérés dans ce sous-tableau. Pour chaque insertion, la position de destination est déterminée par recherche dichotomique puis l'élément est inséré en décalant les éléments de rang supérieur jusqu'au premier trou rencontré. Les éléments sont ensuite rééquilibrés, c'est-à-dire distribués régulièrement dans le sous-tableau de taille $(2+2\epsilon)2^i$ (figure 3.4(b)). Au tour suivant, les éléments sont de nouveau insérés un à un dans le tableau (figure 3.4(c)).

3.2.1 Comportement du tri de bibliothèque

Nous présentons dans cette section les résultats de notre implantation de l'algorithme de tri de bibliothèque. Aucune évaluation expérimentale de ce tri n'a été publiée à notre connaissance. Plus spécifiquement, ce qui nous intéresse dans cette étude est le comportement de l'algorithme selon le nombre de trous ajoutés.



(a) Temps d'exécution pour de petites séquences.

(b) Temps d'exécution avec jusqu'à 8 000 000 éléments.

FIGURE 3.5 – Temps d'exécution du tri par insertion, du quicksort (qsort libc) et du tri de bibliothèque avec $\epsilon = 0.5$ pour différentes tailles de problème.

Nous avons comparé le tri de bibliothèque au tri par insertion classique et au Quicksort très optimisé fourni par la **libC** par la fonction `qsort`. L'expérience consiste à mesurer le temps d'exécution de ces trois codes pour le tri de N éléments tirés uniformément au hasard (figure 3.5). Le tri par insertion classique et le Quicksort procèdent au tri sur des tableaux d'éléments denses. Le tri par insertion classique est, comme

attendu, extrêmement coûteux au-delà de quelques dizaines d'éléments (figure 3.5(a)). Le surcoût de traitement de l'algorithme du tri de bibliothèque devient très rapidement négligeable par rapport au tri par insertion classique (au-dessus de 120 éléments).

L'algorithme `qsort` est une référence pour le tri de tableaux denses. Bien que notre version du tri de bibliothèque ne soit pas aussi performante (figure 3.5(b)), le résultat obtenu est très intéressant. En effet, la complexité algorithmique de `qsort` est $O(N \log N)$. Notre implantation non optimisée du tri de bibliothèque est proche de cette complexité à un facteur près (figure 3.6). Or, si nous avions inséré un à un les éléments dans un tableau puis trié à chaque étape avec `qsort`, nous nous serions retrouvé dans la situation du pire cas de l'algorithme Quicksort. Cet algorithme trie un tableau presque trié à un élément près, avec une complexité en $O(N^2)$. Le temps d'exécution est alors comparable à celui du tri par insertion classique. Autrement dit, ces résultats valident le principe de l'ajout de trous dans un tableau dense dans la situation de l'insertion de nouveaux éléments au cours du temps.

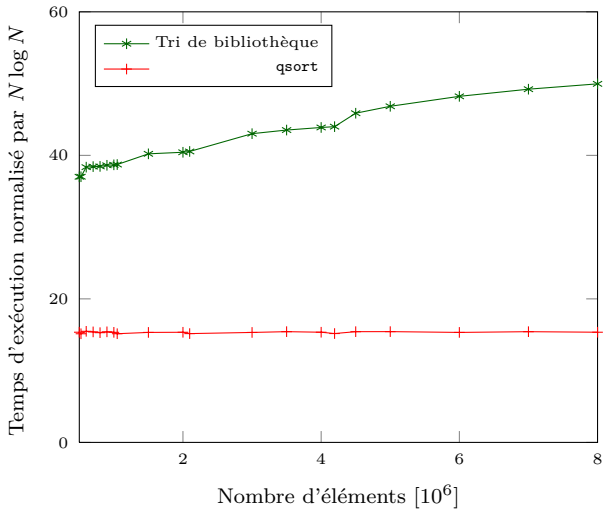


FIGURE 3.6 – Comparaison du comportement asymptotique du `qsort` et du tri de bibliothèque : le temps d'exécution du tri pour N éléments est divisé par $N \log N$. Le comportement de l'algorithme `qsort` est parfait pour le tri de N éléments aléatoires uniformément répartis.

La valeur choisie pour le taux de trous, $\epsilon = 0.5$ est un bon compromis entre la taille supplémentaire nécessaire et le temps d'exécution. En effet, pour un nombre d'éléments donné, les valeurs de ϵ pour lesquelles le temps d'exécution est le plus faible sont $\epsilon = 0.5$ (50% de trous) et $\epsilon = 1.0$ (double la taille du tableau) (figure 3.7). Des irrégularités importantes sont observées pour $0.1 \leq \epsilon \leq 1.0$. La zone entre 0.4 et 0.6 a été discrétisée plus précisément pour déterminer s'il s'agit d'un effet particulier de la valeur 0.5. Les points tracés correspondent à des moyennes sur 31 exécutions et ne sont donc pas le fruit d'une irrévérentieuse pirouette de la machine utilisée. Il y a un bien un effet particulier pour $\epsilon = 0.5$ et $\epsilon = 1.0$. Nous soupçonnons des effets de simplification des calculs arithmétiques, les points 0.25 et 0.75 étant également des minima locaux.

Pour chaque taux de trous, nous avons compté le nombre de déplacements en mémoire effectués (figure 3.7). La mesure inclut les déplacements lors des phases

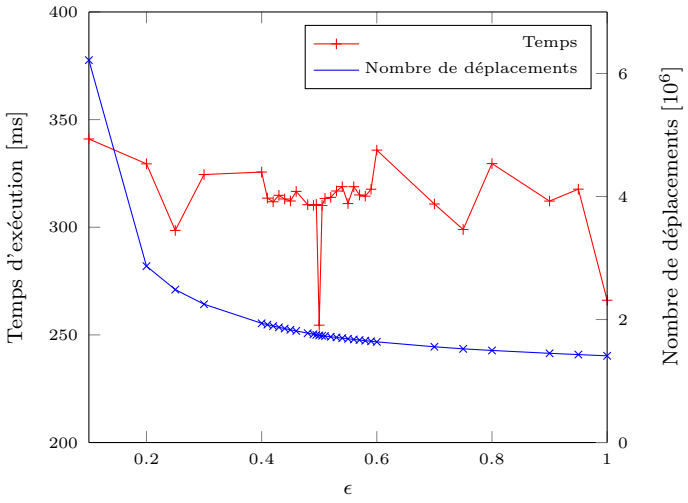


FIGURE 3.7 – Influence de la densité de trous sur la performance du tri de bibliothèque et sur le nombre de déplacements en mémoire pour $N = 50\,000$.

d'insertions et des phases de rééquilibrages. Comme espéré, avec l'augmentation de ϵ moins de mouvements en mémoire sont nécessaires. En effet, pour un même nombre d'éléments à trier, quelque soit le taux de trous le nombre d'étapes de rééquilibrage est constant et le nombre d'éléments à redistribuer également. Nous observons alors la diminution de l'effort à fournir pour insérer les éléments dans la phase d'insertion. Les comportements présentés ici pour $N = 50\,000$ éléments sont observables pour d'autres tailles de séquences.

Si l'algorithme de la bibliothèque fonctionne aussi bien, c'est parce qu'intrinsèquement, le coût des opérations de rééquilibrages est amorti sur le nombre d'éléments insérés. Plus la structure contient d'éléments moins souvent sont déclenchés les rééquilibrages. En maintenant suffisamment de trous, les insertions génèrent peu de déplacements mémoire. Cette étude préliminaire, utile pour démontrer l'intérêt pratique des trous atteint cependant ses limites. En effet, pour maintenir un ordre dans une liste d'éléments, nous avons besoin d'une opération de suppression, non supportée par le tri de bibliothèque.

3.3 PMA

3.3.1 La structure

L'idée de la dissémination de trous parmi des éléments en ordre a été utilisée dans une structure de queues de priorité proposée par [Itai et al., 1981]. La difficulté est là aussi de maintenir à faible coût la répartition des trous. La structure appelée *Packed Memory Array* (PMA) a été présentée ultérieurement par [Bender et al., 2000, 2005] et intègre le support de la suppression des éléments.

Le concept est de distribuer des trous dans un tableau de manière à avoir, quelque soit l'endroit où un nouvel élément s'insère, peu de déplacements à effectuer. Pour cela, le tableau est vu comme s'il était composé de plusieurs morceaux appelés *segments* mis

bout à bout. En gardant des trous dans chacun des segments, nous garantissons des insertions nécessitant peu de déplacements.

Le PMA est ainsi un tableau virtuellement découpé en une multitude S de segments de taille s. Choisir un nombre de segments égal à une puissance de 2 permet d'édifier un arbre binaire complet dont les feuilles sont les segments du tableau. Chaque nœud de l'arbre correspond à une *fenêtre* de segments (figure 3.8). La racine de l'arbre est la fenêtre contenant tous les segments du PMA. La *capacité* C d'une fenêtre est sa taille. Pour un niveau l donné, la capacité des fenêtres est donnée par le nombre de segments de la fenêtre multiplié par la taille d'un segment : $C_l = 2^l s$. Pour maintenir au niveau des feuilles, c.-à-d. dans les segments, un nombre de trous suffisant, des limites sur le rapport entre le nombre d'éléments valides et la capacité des fenêtres sont définies à chaque niveau de l'arbre. Ces densités d'éléments minimum ρ_l et maximum τ_l doivent être respectées par toutes les fenêtres d'un niveau l.

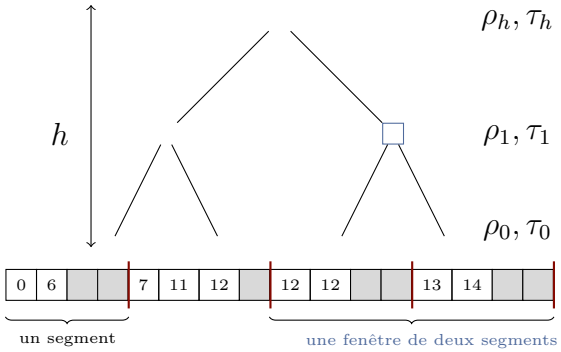


FIGURE 3.8 – Structure de type PMA. Le tableau est divisé en segments qui peuvent contenir des trous (cellules grisées). La densité de chaque fenêtre de niveau i est bornée par les valeurs ρ_i et τ_i.

3.3.2 L'algorithme

Nous expliquons maintenant comment les insertions et suppressions d'éléments sont réalisées dans le PMA tout en maintenant la densité souhaitée des fenêtres à chaque niveau [Bender et al., 2000].

Soit un PMA valide dans lequel toutes les densités d'éléments sont respectées pour toutes les fenêtres de la structure. Lors de l'ajout d'un nouvel élément, une recherche dichotomique permet de déterminer efficacement le segment dans lequel l'insertion doit être effectuée. S'il reste suffisamment de place dans le segment pour ajouter un nouvel élément tout en observant les limites sur les densités, l'insertion peut être réalisée directement. Dans le cas contraire, il faut *rééquilibrer* une partie de la structure : redistribuer les éléments de plusieurs segments pour répartir les trous.

Le principe est de chercher la plus petite fenêtre de segments dans laquelle il y a suffisamment de place pour l'ajout de l'élément. L'arbre contenant le nombre d'éléments par fenêtre n'a pas besoin d'être stocké. La remontée dans l'arbre parmi les ancètres du nœud fautif s'arrête dès que le rapport entre le nombre d'éléments plus un et la

capacité de la fenêtre satisfait la densité maximale du niveau courant. Si la recherche de la fenêtre adéquate atteint la racine de l'arbre et que la densité totale serait dépassée par l'ajout d'un élément, il faut ajouter de l'espace dans le tableau dont la taille est alors augmentée, comme cela est fait pour le tri de bibliothèque (section 3.2). Lorsque la fenêtre de rééquilibrage a été déterminée, l'opération suivante consiste à répartir uniformément les éléments entre les différents segments et à finalement insérer le nouvel élément.

Les suppressions d'éléments à l'intérieur de la structure sont gérées de manière symétrique. L'élément à supprimer et son segment sont localisés. Le nombre d'éléments du segment doit respecter sa densité minimum malgré la suppression. Si ce n'est pas le cas, nous recherchons la fenêtre de segments dans laquelle le nombre d'éléments est suffisant avant de procéder à un rééquilibrage. A l'instar des insertions, s'il ne reste plus assez d'éléments dans le PMA, sa taille est réduite et les éléments sont regroupés dans l'espace disponible.

3.3.3 Le cadre

Nous exposons ici quelques relations qui encadrent le bon fonctionnement du PMA et garantissent sa performance.

Soit un PMA de taille N. La densité d'un segment est comprise entre ρ_0 et τ_0, la densité du PMA total entre ρ_h et τ_h. Ces densités vérifient :

$$0 \leq \rho_0 < ... < \rho_h < \tau_h < ... < \tau_0 \leq 1 \tag{3.1}$$

Les feuilles de l'arbre peuvent être localement plus vides ou plus remplies que la structure totale. Cela permet de réduire le nombre de rééquilibrages de bas niveau, favorisant leur amortissement. Après un rééquilibrage à un niveau l, toutes les sous-fenêtres respectent leurs densités.

Lorsque la taille du PMA contient trop d'éléments, l'ajout d'un élément déclenche une augmentation de sa taille. Il faut que la nouvelle densité après l'ajout de l'élément respecte la densité inférieure, imposant la contrainte supplémentaire :

$$2\rho_h < \tau_h \tag{3.2}$$

Les densités d'un niveau intermédiaire l contenant 2^l segments sont interpolées par :

$$\tau_l = \tau_h + (\tau_0 - \tau_h)\frac{h-l}{h} \tag{3.3}$$

$$\rho_l = \rho_h + (\rho_0 - \tau_h)\frac{h-l}{h} \tag{3.4}$$

Toutes valeurs de ρ_0, ρ_h, τ_h et τ_0 satisfaisant les équations (3.1 - 3.2) peuvent être choisies. La différence entre les densités maximum de deux niveaux consécutifs est :

$$\tau_{l-1} - \tau_l = \frac{\tau_0 - \tau_h}{h} \tag{3.5}$$

3.3.4 La preuve

Le coût amorti de l'insertion ou de la suppression d'un élément dans le PMA est $O(\log^2 N)$ déplacements, le tout générant $O(1 + \frac{\log^2 N}{B})$ transferts mémoire amortis avec B la taille d'une ligne de cache [Bender et al., 2000, 2005]. Pour certains schémas particuliers comme les insertions et les suppressions uniformément réparties, cette complexité est réduite à $O(\log N)$. Cette complexité encore plus intéressante peut également être atteinte grâce une amélioration du PMA introduite par [Bender and Hu, 2007]. Les rééquilibrages sont adaptés en tenant compte de la fréquence des insertions dans les différentes parties ce qui permet à l'algorithme de bien se comporter dans tous les cas, même lorsque les motifs d'insertion sont répétitifs comme l'insertion toujours après un même élément ce qui est le cas le plus défavorable du PMA classique.

Nous donnons ici le principe de la preuve de complexité proposée par [Bender et al., 2000]. Grâce aux trous, l'insertion triée dans un segment du PMA où il y a la place se fait en général en $O(\log N)$ déplacements mémoire. Les opérations coûteuses sont les rééquilibrages des fenêtres du PMA. Un rééquilibrage est déclenché lorsque la densité d'un segment est dépassée. L'opération est réalisée sur une fenêtre u de niveau l dont la densité $d(u)$ est telle que $\rho_l \leq d(u) \leq \tau_l$ et dont l'un des nœuds fils v présente une densité $d(v)$ ne respectant pas les densités ρ_{l-1} ou τ_{l-1}. Après le rééquilibrage, la densité du fils vérifie $\rho_l \leq d(v) \leq \tau_l$. Au plus $\tau_l C_{l-1}$ éléments ont été distribués dans la fenêtre du nœud fils. La densité de ce nœud sera de nouveau dépassée lorsqu'au moins $(\tau_{l-1} - \tau_l)C_{l-1}$ nouveaux éléments auront été insérés.

Le rééquilibrage provoquant au maximum C_l déplacements mémoire, son coût amorti par insertion est :

$$\frac{C_l}{(\tau_{l-1} - \tau_l)C_{l-1}} = \frac{2}{(\tau_{l-1} - \tau_l)} \tag{3.6}$$

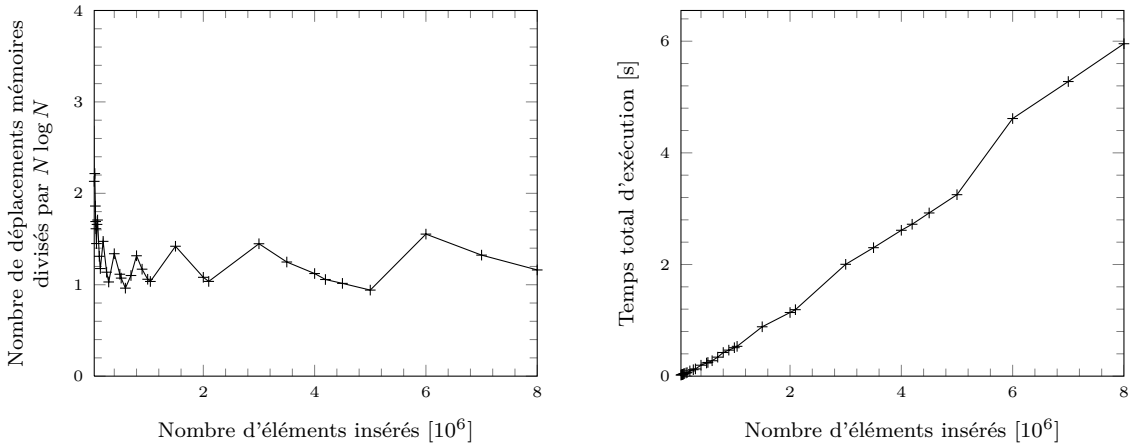
La taille des segments est $C_0 = \Theta(\log N)$ alors la hauteur h de l'arbre est $h = \Theta(\log\left(\frac{N}{\log N}\right)) = \Theta(\log N)$. Or d'après l'équation (3.5), nous obtenons une limite sur le coût amorti par insertion en $O(\log N)$. Lorsqu'un élément est inséré dans le PMA, il contribue à l'augmentation du nombre d'éléments de chacun des $h = \Theta(\log N)$ niveaux de l'arbre. Le taille amortie totale des opérations de rééquilibrage par insertion est alors $O(\log^2 N)$.

3.3.5 Résultats expérimentaux

Nous analysons notre implantation du PMA et comparons son comportement aux résultats publiés dans [Bender and Hu, 2007].

La première étude réalisée est celle de l'évolution du nombre de déplacements en mémoire pour une distribution aléatoire uniforme d'insertions (figure 3.9(a)). Les densités choisies pour le PMA sont les valeurs de référence dans [Bender and Hu, 2007] : $\tau_0 = 0.92$, $\tau_h = 0.7$, $\rho_h = 0.3$ et $\rho_0 = 0.08$. Le nombre de déplacements en mémoire moyen par insertion normalisé par $\log N$ est globalement stable pour les deux codes comparés [voir Bender and Hu, 2007, figure 15]. Notre version semble générer relativement plus de déplacements pour de petites tailles de séquences. En revanche,

nous améliorons les performances en obtenant un facteur compris entre 1 et 1.5 entre 200 000 et 1 500 000 au lieu du facteur entre 1.5 et 2 pour les mêmes tailles de séquences qu'obtient [Bender and Hu, 2007]. Nous observons de petites augmentations relatives du nombre moyen de déplacements mémoires. Ces petits pics sont aussi visibles dans l'implantation de Bender and Hu et sont liés à l'évolution de la taille du PMA. Lorsque l'augmentation de la taille de la structure se produit alors que le nombre total d'éléments à insérer est presque atteint, le dernier rééquilibrage n'a pas le temps d'être amorti par les insertions supplémentaires.



(a) Déplacements mémoires normalisés par $N \log N$.

(b) Temps d'exécution sur Houle.

FIGURE 3.9 – Insertion de N éléments aléatoires dans un PMA.

Nous présentons également l'évolution du temps d'exécution avec le nombre d'éléments insérés dans un PMA (figure 3.9(b)). Nous insérons 1 500 000 éléments en moins d'une seconde sur la machine de test Houle à 2.27GHz (section 4.4.1). Pour 1 400 000 éléments, les auteurs obtenaient en 2007 des temps de l'ordre de 11 secondes sur un Pentium 4 à 3GHz [voir Bender and Hu, 2007, figure 15]. Les architectures ont tellement évolué depuis cette époque que les comparaisons quantitatives sont difficiles. L'ordre de grandeur entre les performances des deux versions peut s'expliquer par la différence entre les bandes passantes mémoire, l'architecture du processeur, l'amélioration des compilateurs ou des optimisations différentes dans la structure. [Bender and Hu, 2007] ne détaillent pas leur implantation. La structure que nous avons adoptée et ses optimisations seront discutées dans le chapitre 4.

Notre PMA a le comportement asymptotique souhaité, $O(N \log N)$. De plus, il est au moins aussi performant que la version de référence. Nous pourrons ainsi l'utiliser sans complexe pour des comparaisons dans la suite de ce document.

3.4 Voyages au sein d'un PMA

Les opérations d'insertion et de suppression sont supportées efficacement par le PMA. Cependant le mouvement d'éléments au sein de la structure n'a pas été étudié. Le PMA ayant été conçu pour le domaine des bases de données, cette opération avait moins d'intérêt que pour la simulation numérique. Un élément est qualifié de *voyageur* si sa position par rapport aux autres éléments n'est plus correcte. Nous comptons par la suite le nombre de voyageurs, c'est-à-dire Rem, le nombre d'éléments qu'il faut enlever pour retrouver une liste triée (section 3.1.1).

En l'état, cette modification de l'ordre dans un PMA peut être traitée par le déplacement des éléments voyageurs un à un : suppression de l'élément puis ré-insertion dans la structure. Cependant, le parcours de la structure pour supprimer les éléments devant se déplacer risque d'engendrer des rééquilibrages inutiles. En effet, d'un côté du PMA les segments se dépeuplent pendant que d'autres segments se remplissent. Lorsque l'autre côté sera atteint des éléments devront peut–être réintégrer la première partie. Si des rééquilibrages ont été réalisés dans une fenêtre à cause des insertions, ils seront peut-être de nouveau nécessaires après les suppressions.

Pour éviter ces surcoûts nous proposons PaVo, l'algorithme des Paquets de Voyageurs permettant de maintenir l'ordre au sein d'un PMA lorsqu'une fraction des éléments ne sont plus à leur place et doivent voyager dans la structure. L'idée est d'agréger les requêtes de voyages afin de covoiturer pour améliorer le débit et éviter la pollution, des caches, par exemple.

3.4.1 L'algorithme des paquets de voyageurs, PaVo

Nous supposons qu'il est possible de retrouver en un parcours de la structure les cN éléments voyageurs, avec c tel que $0 \leq c \leq 1$. Le PMA est retrié en deux étapes : identification, puis réintroduction combinée des voyageurs. La conformité du PMA (respect des densités) est seulement garantie de nouveau à la fin de la réintroduction.

Lors de la phase d'identification, les éléments voyageurs sont retirés du PMA et assemblés dans un tableau dense de voyageurs. Les densités des segments ne sont pas vérifiées, aussi le PMA se trouve-t-il dans un état non correct du point de vue de la répartition des éléments. Lorsque tous les voyageurs d'une session ont été retrouvés, le tableau de voyageurs est trié par Quicksort.

La réintroduction des éléments s'effectue récursivement par paquets en parcourant de la racine jusqu'aux feuilles l'arbre des fenêtres. Si uniquement des mouvements d'éléments sont effectués (pas d'ajout ni de retrait), nous sommes assurés que la taille du PMA n'a pas à être modifiée.

Par une recherche dichotomique au sein du tableau trié des voyageurs nous déterminons le nombre d'éléments devant être placés respectivement dans le fils de gauche et dans celui de droite du nœud courant de niveau l. La valeur de pivot qui sert à délimiter les deux ensembles est le plus petit élément du fils droit de la fenêtre de niveau l. Si pour chacune deux sous-fenêtres la somme des voyageurs et des éléments déjà en place respecte les densités minimum et maximum du niveau $l-1$ alors cette étape

est répétée pour tous les enfants du nœud en utilisant le sous-ensemble des voyageurs correspondant (figure 3.10).

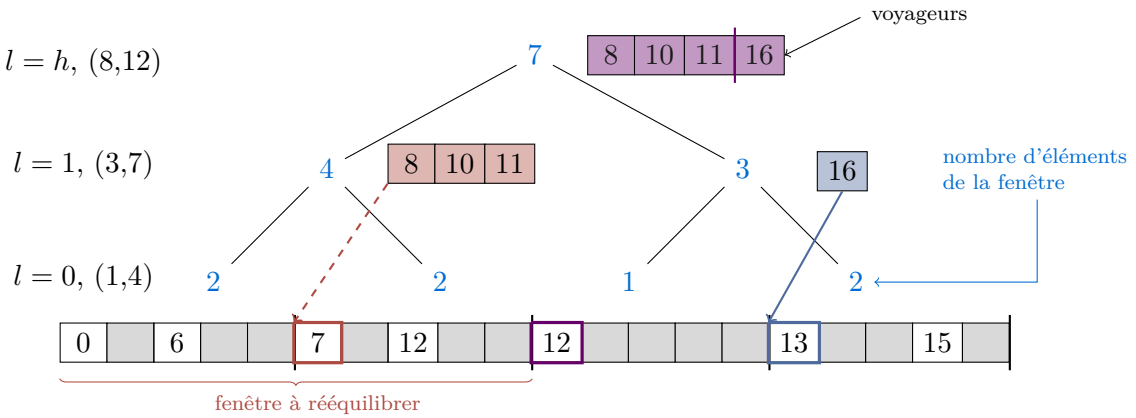


FIGURE 3.10 – Fonctionnement de l'algorithme PaVo. Pour chaque niveau sont données les limites minimum et maximum exprimées en nombre d'éléments. Les éléments voyageurs en violet ont déjà été séparés en deux groupes rouge et bleu car les densités étaient respectées ($3 \leq 4+3 \leq 7$ à gauche, $3 \leq 3+1 \leq 7$ à droite). L'élément du groupe bleu peut être contenu dans son segment cible. Les éléments du groupe rouge devraient s'insérer dans le deuxième segment qui atteindrait alors $2+3 = 5 > 4$ éléments. La fenêtre de gauche doit être rééquilibrée.

Si l'une des densités de l'un des fils est rendue caduque par l'insertion des éléments prévus alors un rééquilibrage spécial de la fenêtre est nécessaire. L'opération consiste à effectuer la fusion triée des éléments voyageurs et des éléments du PMA tout en les distribuant uniformément dans les segments concernés. Même si les voyageurs ont tous été placés, les densités des segments du nœud courant doivent être contrôlées, car des éléments ont pu être retirés de ces segments lors de la première phase et les densités minimum peuvent ne plus être respectées. Si tel est le cas, un rééquilibrage de la fenêtre est déclenché. Au contraire, s'il reste des voyageurs à insérer alors que la fenêtre concernée ne contient plus d'éléments, les voyageurs sont aisément distribués équitablement entre les segments de cette fenêtre.

3.4.1.1 Modification du nombre d'éléments

L'algorithme présenté ci-dessus permet également de gérer les insertions et les suppressions d'éléments. En effet, pour des insertions il suffit d'intégrer les nouveaux éléments dans le tableau de voyageurs et pour des suppressions, il suffit de ne pas transférer les éléments à supprimer dans ce tableau de voyageurs.

Pour gérer cette possibilité, nous vérifions avant d'amorcer la descente de l'arbre si le nombre d'éléments total respecte les densités globales du PMA. Selon le cas, la structure est réduite ou augmentée ce qui permet de dérouler l'algorithme de réintroduction des voyageurs par paquets en étant assuré que le PMA pourra contenir tous les éléments et qu'il ne sera pas trop vide.

3.4.1.2 Partition des voyageurs

Le tri des voyageurs peut être fait au cours de la descente. Les éléments d'une fenêtre sont simplement échangés dans le tableau de manière à ce que tous les éléments inférieurs au pivot soient à gauche et tous les éléments supérieurs à droite. La valeur du pivot est donnée par l'élément médian de la fenêtre considérée. Ce choix peut avoir un impact principalement lorsque les distributions de voyages ne sont pas bien équilibrées. Dans l'implantation de l'algorithme Quicksort, le pivot est l'élément médian entre le premier des éléments à trier, le dernier et celui du milieu. Pour le cas de distribution uniforme, cette valeur sera proche de celle donnée par les éléments du PMA.

Si le nombre d'éléments de chaque côté ne respectent pas l'une ou l'autre des limites de densités, alors les éléments voyageurs sont triés avant la fusion avec les éléments du PMA.

3.4.2 Études

Nous étudions la complexité de l'algorithme PaVo dans différentes situations. Nous ne proposons pas de preuve formelle mais des intuitions sur le comportement de l'algorithme.

3.4.2.1 Dans le pire cas

Considérons comme état initial un PMA équilibré de manière homogène sur toute la structure : la densité de chaque segment est $\tau_h < \rho < \rho_h$. Dans la situation où les éléments voyagent d'un segment donné de la seconde moitié du PMA vers un segment donné de la première moitié, les éléments sont supprimés de la partie droite pour être insérés dans la partie gauche. Il s'agit d'un pire cas pour PaVo. En effet, en insérant systématiquement sur le même segment, le nombre de rééquilibrages est maximisé par rapport au nombre d'insertions. De plus, nous ne tirons pas partie du fait d'avoir rassemblé les requêtes de voyages, puisque le mécanisme de sélection du pivot sera dégénéré. Les R voyageurs de valeur égale seront triés en $O(R \log R)$ ou $O(R^2)$ comparaisons selon l'algorithme de tri utilisé.

Les insertions d'un côté et les suppressions de l'autre vont générer le même nombre de rééquilibrages. Au fur et à mesure des insertions par vagues de R voyageurs dans le même segment, celui-ci sera trop plein et devra être rééquilibré avec ses voisins. Si nous poursuivons cette opération suffisamment longtemps, petit à petit la hauteur des rééquilibrages va augmenter jusqu'à atteindre un rééquilibrage complet de la structure. En fait, quelle que soit la taille des paquets de voyageurs, la borne de complexité du PMA s'applique puisque les seules opérations que nous effectuons sur le PMA sont des insertions. Ainsi le coût amorti par voyage est dans le cas général $O(\log^2 N)$.

Il est probable que cette borne puisse être améliorée pour des grandes tailles de paquets. Le cas extrême étant un paquet de taille suffisante pour directement déclencher un équilibrage de plus haut niveau. Le nombre amorti de mouvements de données par voyage est alors en $O(\log(N))$.

3.4.2.2 Presque rien à faire

Dans le meilleur des cas, un tout petit nombre d'éléments se déplacent et ne génèrent pas de rééquilibrages, ni lors de leur suppression ni lors de leur réinsertion.

Soit R le nombre de voyageurs. Le tri des voyageurs se fait en $O(R \log R)$. Si tout est bien équilibré, la position des pivots dans le tableau des voyageurs est déterminée par une recherche dichotomique en :

$$\begin{aligned} \sum_{i=0}^{h-1} 2^i \log(\frac{R}{2^i}) &= 2^h \log R - \sum_{i=0}^{h-1} i2^i, & (3.7) \\ &= 2^h \log R - 2^h(h-2) - 2, & (3.8) \\ &= \frac{N}{\log N}\left(\log R - \log \frac{N}{\log N} - 2\right) - 2. & (3.9) \end{aligned}$$

opérations. Il faut ensuite rechercher linéairement la position d'insertion dans les segments de taille $\Theta(\log N)$, ce qui pour les R éléments voyageurs est borné par $R \log N$.

La complexité algorithmique totale est donnée par $O(R \log R + \frac{N \log R}{\log N} + R \log N)$ soit $O(2R \log N + \frac{N \log R}{\log N})$.

3.4.2.3 Un PMA rend-il le tri adaptatif ?

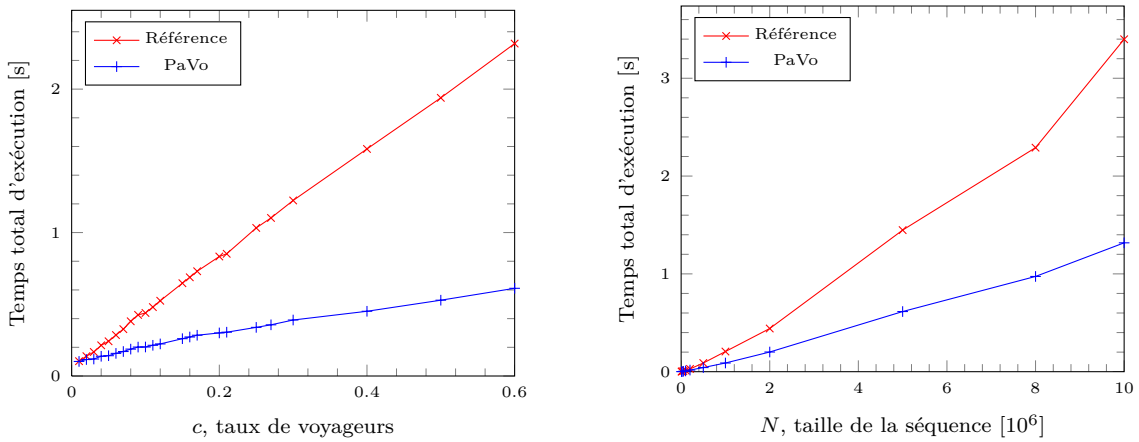
Le PMA n'a pas été positionné comme un tri adaptatif par ses auteurs, et à ce titre sa complexité vis-à-vis de mesures de désordre n'a pas été étudiée. Les auteurs étudient plutôt le comportement du PMA pour divers schémas d'insertion ou suppression classiques dans les bases de données (répartition uniforme, insertions répétées en début de PMA, etc.), premier domaine d'application cible du PMA.

L'algorithme PaVo est clairement un tri adaptatif dans le principe puisque le travail effectué dépend uniquement du désordre. Plus formellement, l'algorithme PV peut être vu comme une variante de l'algorithme de [Cook and Kim, 1980] avec un schéma particulier pour éviter la fusion des deux listes, coûteuse en mouvements de données. PaVo serait *Rem* optimal à condition d'utiliser un tri des voyageurs en $R \log(R)$ en pire cas [Petersson and Moffat, 1995].

3.4.3 Comportement des voyageurs

L'implantation de l'algorithme des voyages par paquets au sein d'un PMA sera discutée et analysée dans le chapitre 4.

Nous nous intéressons ici au comportement de l'algorithme PaVo par rapport à la méthode dite de référence en effectuant une suppression puis une insertion de chaque élément voyageur dans le PMA. L'expérience réalisée consiste à parcourir les éléments pour repérer ceux dont la valeur a changé. Ces éléments une fois tous identifiés, avec la méthode de référence ils sont cherchés, supprimés puis insérés. L'algorithme PaVo les supprime sans rééquilibrage, les copie dans un tableau auxiliaire puis les réinsère. Nous



(a) Influence du taux c de voyageurs pour une séquence de $K = 2\,000\,000$ éléments.

(b) Influence de la taille de la séquence pour $c = 0,1$ soit 10% de voyageurs.

FIGURE 3.11 – Comparaison des temps d'exécution de la méthode de référence et de l'algorithme PV pour le tri de $R = cK$ voyageurs dans un PMA de taille N.

mesurons le temps d'exécution total de toutes ces étapes que nous répétons plusieurs fois pour en extraire des temps moyens représentatifs (figure 3.11). La séquence d'éléments est déterminée de manière uniformément aléatoire. De même les éléments voyageurs sont sélectionnés aléatoirement et une nouvelle clé choisie au hasard leur est attribuée.

La première étude est réalisée à nombre d'éléments constant, $N = 2\,000\,000$, en faisant varier le taux de voyageurs (figure 3.11(a)). Pour des taux de 1% ou 2%, la méthode de référence est plus performante, au-dessus, l'algorithme PaVo est nettement plus intéressant. Les deux méthodes présentent des comportements linéaires avec le taux de voyageurs, avec un facteur de l'ordre de 3 pour la méthode de référence, seulement de 0.5 pour PaVo.

L'étude à toux de déplacements constant, ici 10%, permet d'apprécier l'influence de la taille de la séquence sur le temps d'exécution de chacune de ces méthodes jusqu'à 10 000 000 éléments (figure 3.11(b)). Le temps d'exécution de PaVo croît linéairement avec le nombre total d'éléments. Ce résultat est remarquable : nous sommes loin du comportement asymptotique des tris classiques sur des tableaux denses.

3.5 Discussion

Dans ce chapitre, nous avons illustré sur un cas simple l'intérêt des trous pour améliorer l'efficacité d'algorithmes de tri basés sur le principe d'insertion. L'algorithme PaVo permet d'appliquer des mises à jour d'éléments efficaces dans un tableau presque trié (section 3.4.1). Les résultats préliminaires prouvent l'intérêt de cette approche originale.

Il serait intéressant de travailler la complexité théorique de PaVo vis-à-vis d'autres mesures de désordre ou par rapport à l'optimalité en cache en s'appuyant sur la définition de [Brodal et al., 2005]. Pour cette étude, il faudrait identifier précisément des générateurs de désordre pertinents ce qui est une tâche délicate : les études expérimentales sont dans le meilleur des cas conduites en testant plusieurs types de distributions [Xiao et al., 2000].

L'évaluation précise et l'utilité applicative de l'algorithme de déplacements par paquets au sein d'un PMA sont discutées dans le chapitre suivant (chapitre 4). Nous apprendrons notamment que l'espace mémoire nécessaire pour la maintenance de cette structure est linéaire en fonction du nombre d'éléments.

Dans le dernier chapitre de cette partie (chapitre 5) nous évaluons la version parallèle de l'algorithme. D'après l'intuition qui nous pousse à avancer, l'intérêt de PaVo devrait pleinement s'exprimer par rapport aux algorithmes de tri parallèles sur des tableaux denses. À suivre...

Les dessous de PaVo 4

Au chapitre précédent, nous avons présenté la structure de données appelée PMA. Cette structure permet de maintenir un taux de trous dans un tableau d'éléments (triés). Nous avons introduit un algorithme pour retrier un paquet d'éléments appelés voyageurs dont la position a été modifiée. Nous avons montré que cet algorithme, PaVo, permettait d'appliquer plus efficacement des déplacements dans un PMA.

Nous avons introduit l'intérêt des trous dans le cas simple du tri de bibliothèque. Mais, qu'apportent les trous au tri de séquences presque triées ?

En voici un avant-goût prometteur : pour 2% de voyageurs, PaVo s'illustre par un impressionnant gain de performance, même par rapport à des tris très optimisés comme `qsort` et `stdsort` (figure 4.1). Le profil de ces courbes est intéressant : l'accélération par rapport à `qsort` dépend peu de la taille des éléments en mémoire. En revanche, les performances de la version de l'algorithme `stdsort` que nous avons utilisée se dégradent fortement avec l'augmentation en mémoire.

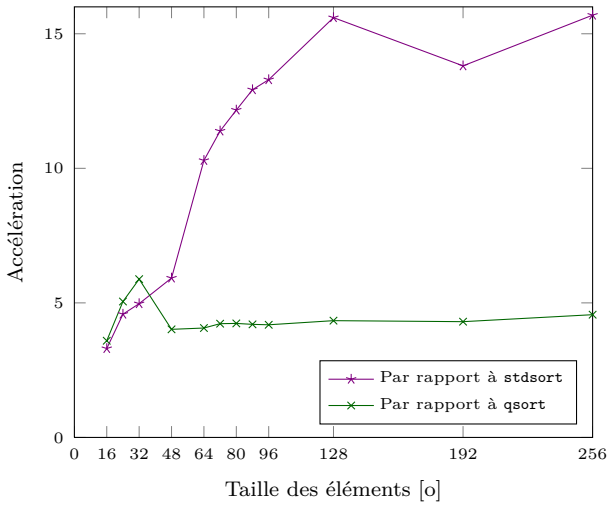


FIGURE 4.1 – Tri d'une séquence presque triée comportant $K = 2\,000\,000$ éléments dont 2% de voyageurs, accélération de PaVo par rapport à `stdsort` et `qsort` selon la taille des éléments en mémoire.

Dans ce chapitre, nous nous proposons de répondre en détail à cette question. Nous expliquons la structure (sections 4.1 et 4.2) et les algorithmes (section 4.3) utilisés. Nous analysons l'apport de notre solution (section 4.4) puis nous l'appliquons à des mouvements de particules SPH (section 4.5).

4.1 Interface

Nous avons implanté l'algorithme des Paquets de Voyageurs (PaVo) dans la librairie **Storm** écrite en C qui définit et maintient un PMA. Nous commençons par décliner rapidement les différentes opérations pouvant être appliquées sur la structure. Les structures utilisées sont détaillées dans la section suivante (section 4.2), les algorithmes décrits ensuite (section 4.3).

L'interface d'accès à la structure permet de construire un PMA (`pma_build`), d'insérer des éléments (`pma_add_elt`), de supprimer des éléments (`pma_remove_elt`) et bien sûr de supprimer la structure (`pma_destroy`). Il est également possible d'initialiser le PMA en utilisant un tableau d'éléments déjà triés (`pma_init`). La recherche d'éléments est naturellement réalisable (`pma_search_elt`).

L'interface propose un appel pour demander le compactage des éléments valides du PMA dans un tableau auxiliaire (`pma_compact`). **Storm** permet aussi de retrouver l'ensemble des positions contenant des éléments valides (`pma_valid_pos`) ou une combinaison de l'ensemble des positions valides et des pointeurs vers le contenu des éléments correspondants (`pma_valid_pos_ptr`).

Des itérateurs ont été prévus pour parcourir les éléments valides (`pma_iter`), ou les éléments valides ayant la même clé (`pma_range_iter`). Cependant il est généralement plus efficace d'effectuer un `foreach` directement sur la structure.

Des fonctions supplémentaires renseignent sur la taille du PMA (`pma_get_size`) ou encore le nombre de segments utilisés (`pma_get_segments`), affichent son contenu (`pma_info_print`) et retournent des statistiques d'utilisation de chacune de ses fonctions (`pma_stats_print`).

En ce qui concerne l'algorithme PaVo proprement dit, les éléments peuvent être supprimés sans générer de rééquilibrage de la structure (`pma_remove_elt_nobalance`). Le PMA est alors momentanément non cohérent. Le programme utilisateur se charge de stocker les éléments dans un tableau temporaire et demande ensuite leur ré-intégration par la fonction (`pma_add_array_elts`). Cette étape est nécessaire pour reconstruire la cohérence de la structure.

4.2 Storm à la loupe

Nous décrivons dans cette section les structures embrassées pour l'implantation de l'algorithme PaVo.

4.2.1 Brique de base

Nous avons choisi de représenter l'ordre entre les éléments par une clé associée à chaque élément. Aucune garantie n'est donnée sur la position relative des éléments de même clé.

La clé permettant d'ordonner les éléments est codée sur 64bits. Le contenu des éléments fourni par l'utilisateur est copié à côté de la clé et la brique de base ainsi

constituée est alignée sur 8o. Au minimum, un PMA ayant une taille N utilisera donc $N \times 16o$ (figure 4.2).

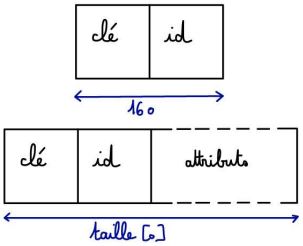


FIGURE 4.2 – Brique de base du PMA.

4.2.2 Taille du PMA

La taille N du PMA est calculée en fonction des densités τ_0, τ_h, ρ_0 et ρ_h (section 3.3.3). Les valeurs par défaut sont celles utilisées par Bender and Hu, 2007 (section 3.3.5) : $\tau_0 = 0,92$, $\tau_h = 0,7$, $\rho_0 = 0,08$ et $\tau_h = 0,3$. Une étude de l'influence des densités choisies est présentée plus loin dans ce document (section 4.6.1). Le PMA est initialisé pour contenir jusqu'à K éléments mais peut par la suite être réduit ou augmenté en fonction des insertions et des suppressions effectuées.

Soit N la taille totale du PMA, S le nombre de segments. D'après l'algorithme originel d'insertion des éléments, le PMA est doublé lorsque les densités maximum ne sont plus respectées. Le problème est que cela contraint N à des puissance de deux. Par exemple, pour un tableau contenant $K = 1\,000\,000$ nous obtenons $N = 2\,097\,152$ soit une densité effective de l'ordre de $2,1$. Utiliser plus de deux fois la taille occupée par les K éléments pénalise excessivement les performances lors des parcours du tableau d'éléments, par exemple.

Voici comment nous procédons. La taille t d'une structure pouvant contenir K éléments à la densité τ_h est $t = \lceil \frac{K}{\tau_h} \rceil$. Le nombre S de segments nécessaires est donné par :

$$S = \llbracket \frac{t}{\log_2 t} \rrbracket \tag{4.1}$$

$\llbracket x \rrbracket$ est la plus petite puissance de 2 supérieure ou égale à x. La capacité C des segments est arrondie à $\lceil \frac{t}{S} \rceil$. La taille totale N du PMA est finalement obtenue par $N = \text{cap} \times S$ où seul le nombre de segments est une puissance de 2. La densité effective du PMA est alors légèrement inférieure à la densité maximum demandée (tableau 4.1).

Lorsque le PMA est initialisé avec `pma_init`, l'utilisateur fournit un tableau d'éléments associés à leur clé déjà triés entre eux. Les éléments sont aisément répartis uniformément dans les segments. En revanche, lors de l'insertion d'éléments, à chaque augmentation de la structure, le nombre de segments doit toujours être une puissance de deux. Le facteur d'accroissement est déterminé par le rapport entre la nouvelle taille, calculée d'après la méthode ci-dessus, et l'ancienne taille. Dans certains cas, la taille augmentée s'obtient avec moins de segments (et une capacité des segments plus importante) ou avec plus de segments mais une capacité plus faible. Ces situations

K	S	cap	N	ρ
100	32	5	160	0,625
1 000	256	6	1 536	0,651
10 000	2 048	7	14 336	0,698
100 000	16 384	9	147 456	0,678
1 000 000	131 072	11	1 441 792	0,694
10 000 000	1 048 576	14	14 680 064	0,681
100 000 000	8 388 608	18	150 994 944	0,662

TABLE 4.1 – Nombre S de segments, capacité C, taille N du PMA en fonction du nombre d'éléments K pour une densité maximum du tableau complet souhaitée $\tau_h = 0,7$. La densité effective ρ pour chaque taille est également indiquée.

conduisent à différencier les algorithmes pouvant être utilisés pour redistribuer les éléments.

4.2.3 Gestion du PMA

La structure générale du PMA est constituée d'un tableau `array` contenant N briques de base (clé plus contenu de l'élément). Nous maintenons également les clés `keys` du premier élément de chaque segment non vide dans un tableau de taille S. Cette structure permet d'accélerer la recherche d'éléments en diminuant le nombre d'accès mémoire nécessaires pour trouver le segment dans lequel l'élément cherché devrait être situé.

4.2.4 Arbre du nombre d'éléments par fenêtre

Nous conservons le nombre d'éléments de chaque segment dans un tableau nommé `count`. Les densités minimum et maximum de chaque niveau (`level_min` et `level_max`) sont calculées à chaque modification de la taille du PMA. En pratique, les densités sont exprimées en nombre d'éléments pour éviter la répétition des calculs flottants lors du fonctionnement.

Le tableau `count` sert également à maintenir le nombre d'éléments de chaque fenêtre. La structure de type tas utilisée est ordonnée des feuilles jusqu'à la racine. Cette structure a un grand intérêt pour l'algorithme PaVo car elle permet d'éviter d'avoir à parcourir tout le PMA dès la première étape de réintroduction des voyageurs pour compter le nombre d'éléments. En effet, nous parcourons le PMA de la racine contenant tous les segments aux feuilles représentées par les différents segments (section 3.4.1 page 69). Au total, sans cette structure, $\Theta(N \log N)$ lectures supplémentaires seraient nécessaires pour calculer la densité de chaque fenêtre à traiter lors de l'insertion des nouveaux éléments.

Cette structure n'est pas absolument nécessaire dans le cadre de l'insertion et de la suppression [Bender and Hu, 2007]. En effet, à chaque insertion ou suppression, les limites de densités sont vérifiées au niveau du segment traité en premier lieu. Comme

la fenêtre à rééquilibrer est espérée petite, le nombre de segments à parcourir ne doit pas être trop grand. En pratique nous avons tout de même relevé une amélioration de performance lorsqu'une structure de ce type est utilisée, même pour les insertions et suppressions classiques.

4.2.5 À l'intérieur des segments

Nous ne compactons pas les éléments en début de chaque segment, contrairement à ce que pourrait suggérer l'appellation *Packed-Memory Array*. Après des suppressions, les éléments peuvent être un peu dispersés dans les segments. Pour ne pas perdre les éléments, chaque case doit être marquée occupée ou non. Nous utilisons un champ de bites pour repérer les trous de la structure. Les bits représentant une case occupée par un élément valide sont assignés à 1 et les autres à 0.

Les éléments pourraient être systématiquement compactés dans les segments. Cela permettrait de diminuer globalement le nombre d'accès mémoire. Cependant, cela augmenterait le nombre de déplacements en mémoire lors de l'insertion d'éléments. De plus, il se trouve que les opérations de rééquilibrage redistribuent les éléments dans les segments de manière compactée. Aussi le nombre de segments dont les éléments sont dispersés est-il faible en pratique lors de l'utilisation de PaVo.

4.3 Les algorithmes

Les algorithmes de maintenance du PMA et certaines subtilités peuvent avoir un impact significatif sur les performance.

4.3.1 Insertion et suppression d'un élément

`pma_add_elt` permet l'insertion unitaire d'un nouvel élément dans le PMA. Les paramètres à fournir sont le contenu de l'élément et sa clé. L'ajout se déroule dans les grandes lignes selon l'algorithme proposé par Bender et al., 2000 et décrit dans le chapitre précédent (section 3.3.2).

Le segment cible du nouvel élément est déterminé par une recherche dichotomique utilisant le tableau des clés `keys`. Si le nombre d'éléments du segment est supérieur à la limite autorisée, alors le nombre d'éléments de chaque fenêtre ancêtre est lu dans l'arbre `count`. La remontée s'arrête au niveau l si le nombre d'éléments dans la fenêtre plus un respecte la limite du niveau (`level_max[l]`).

L'opération de rééquilibrage redistribue les éléments en place parmi les segments de la fenêtre et met à jour en conséquence les structures `keys` et `count`. Puisque les éléments peuvent s'être déplacés entre les segments, nous recherchons le nouveau segment dans lequel l'élément doit s'insérer. Si nous n'avons pas de chance (ça arrive de temps en temps !), ce segment est encore trop plein. Dans ce cas, nous savons qu'il y a cependant suffisamment de place dans la fenêtre où le rééquilibrage a eu lieu. Une

opération supplémentaire permet de décaler les éléments jusqu'à atteindre un segment moins plein.

Nous aurions pu insérer l'élément pendant la phase de rééquilibrage. Cependant, cela nécessite des tests supplémentaires systématiques pour ajouter l'élément à sa place. En pratique cette situation se produit suffisamment peu souvent pour privilégier l'efficacité dans le cas le plus courant. Une autre solution serait d'insérer l'élément dans le segment cible avant de faire le rééquilibrage. Mais cette solution interdit le choix de $\tau_0 = 1$ comme densité maximum d'un segment.

L'arbre `count` du nombre d'éléments de chaque fenêtre est maintenu lors des opérations d'insertion et de suppression. Lorsqu'il n'y a pas eu de rééquilibrage, la mise à jour consiste simplement en un incrément (ou un décrément pour une suppression) du nombre d'éléments du segment modifié et de chacun de ces nœuds parents. Après un rééquilibrage, tous les parents des segments de la fenêtre concernée doivent être modifiés.

Pour supprimer un élément, `pma_remove_elt` utilise la clé fournie pour rechercher tous les éléments correspondant. Une fonction d'égalité est transmise et est appliquée aux éléments de même clé pour retrouver celui qui doit être supprimé. Le retrait suit en miroir les principes de l'algorithme d'insertion et peut générer des rééquilibrages de la structure si la densité du segment se retrouve en dessous de la densité attendue.

4.3.2 PaVo

Pour préparer l'exécution de l'algorithme PaVo, une phase d'identification des éléments voyageurs est nécessaire. Cette étape est réalisée par l'utilisateur via un parcours des éléments valides contenus dans le PMA. Pour chaque élément, si sa position par rapport aux autres change, c'est-à-dire dans notre implantation si sa clé est modifiée, alors l'élément doit être copié dans le tableau des voyageurs puis retiré de la structure avec `pma_remove_elt_nobalance`. Comme nous l'avons indiqué ci-dessus, cette opération permet d'invalider l'élément dans le PMA sans générer de rééquilibrage, même si la densité du segment concerné chûte en deçà des limites autorisées.

Le PMA est alors (momentanément) incohérent, les densités d'éléments n'étant pas respectées. Seuls le nombre d'éléments et la clé de chaque segment modifié sont mis à jour. À la fin de l'étape de retrait des voyageurs, l'arbre du nombre d'éléments de chaque fenêtre est calculé pour être utilisé par la suite.

Les éléments à ré-insérer (`pma_add_array_elts`) dans le PMA ne sont pas triés initialement. Dans la version qui n'utilise pas le mode *partition* (voir section 3.4.1.2), tous les voyageurs sont triés via un appel à la fonction de tri `qsort` de la **libC**.

Ensuite l'algorithme décrit dans le chapitre précédent (section 3.4.1) peut être appliqué (algorithme 1). La descente dans les nœuds de l'arbre est répétée jusqu'à ce que l'on atteigne un segment (ligne 2) ou jusqu'à ce que les densités ne soient plus respectées (ligne 13). Dans les deux cas, une opération de rééquilibrage avec insertion est utilisée pour insérer les éléments restant.

Algorithm 1 Algorithme PaVo

Require : La fenêtre correspondant aux s segments à partir de $s0$ peut contenir v voyageurs.

Require : Le sous-ensemble des v voyageurs est trié.

$node$: nœud de l'arbre correspondant à la fenêtre
l : niveau courant dans l'arbre
$s0$: premier segment de la fenêtre courante
s : nombre de segments
$v0$: premier voyageur
v : nombre de voyageurs

```
1 : function ALGO_PV(node, l, s0, s, v0, v)
2 :     if s = 1 then
3 :         if v > 0 then
4 :             BALANCE(s0, s, v0, v)
5 :         end if
6 :     else
7 :         pivot ← keys[s0 + s/2]
8 :         v_g ← le nombre de voyageurs < pivot
9 :         v_d ← v − v_g
10 :        n_g ← v_g + count[node_g]
11 :        n_d ← v_d + count[node_d]
12 :        if n_g > level_max[l/2] ou n_d > level_max[l/2]
13 :          ou n_g < level_min[l/2] ou n_d < level_min[l/2] then
14 :            BALANCE(s0, s, v0, v)
15 :        else
16 :            if v_g > 0 then
17 :                ALGO_PV(node_g, l − 1, s0, s/2, v0, v_g)
18 :            else
19 :                if la densité d'un segment de la sous-fenêtre est insuffisante then
20 :                    BALANCE(s0, s/2, v0, v/2)
21 :                end if
22 :            end if
23 :            if v_d > 0 then
24 :                ALGO_PV(node_d, l − 1, s0 + s/2, s/2, v0 + v_g, v_d)
25 :            else
26 :                if la densité d'un segment de la sous-fenêtre est insuffisante then
27 :                    BALANCE(s0, s/2, v0, v/2)
28 :                end if
29 :            end if
30 :        end if
31 :    end if
32 : end function
```

La valeur servant de pivot est la clé associée au segment du milieu de la fenêtre courante (ligne 7). Le nombre d'éléments de chaque sous-fenêtre est facilement obtenu grâce à l'arbre du nombre d'éléments par nœud `count` (ligne 10).

Si les densités sont respectées mais qu'il n'y a plus d'éléments voyageurs à insérer dans l'un des fils du nœud courant (ligne 16), il faut tout de même vérifier que le nombre d'éléments de chaque segment est suffisant (ligne 19). En effet, puisque lors de l'étape d'identification des voyageurs aucune opération de rééquilibrage n'a été menée, les segments peuvent présenter des densités inférieures aux limites attendues.

Pour passer en mode *partition*, il faut rajouter une étape. Au lieu de simplement chercher parmi des voyageurs triés où se situe la limite donnée par la valeur du pivot, les éléments sont échangés dans le tableau de voyageurs jusqu'à ce que les éléments inférieurs d'une part et supérieurs d'autre part au pivot soit séparés. Lorsqu'il est nécessaire de rééquilibrer une partie de la structure, alors les éléments voyageurs à intégrer sont triés.

4.3.3 Rééquilibrage avec insertion

Comment rééquilibrer une fenêtre de segments tout en insérant les éléments voyageurs ? Les éléments placés dans le PMA, d'une part et les éléments à insérer d'autre part sont déjà triés aussi l'opération relève-t-elle d'une (simple ?) fusion.

Le nombre total d'éléments de la fenêtre permet de calculer le nombre qui doivent être écrits dans chaque segment. Les segments à rééquilibrer sont parcourus de gauche à droite. Pour chaque position à occuper, il faut vérifier si un élément est déjà présent. Si c'est le cas et que cet élément vient avant le premier voyageur à placer, alors l'algorithme avance à la position suivante. Si le premier voyageur doit être placé avant, l'élément est copié dans une queue tampon stockée dans un tableau annexe et le voyageur est inséré. Si la case n'est pas occupée, l'élément écrit est le premier voyageur, un élément de la queue tampon ou, si elle est vide, le prochain élément de la structure, selon leur ordre respectif.

Lorsque toutes les cases à occuper ont été traitées dans un segment, alors les éventuels éléments situés dans les emplacements suivants sont ajoutés à la suite de la queue tampon.

4.4 Évaluations expérimentales

Nous présentons dans cette section l'évaluation expérimentale de l'algorithme PaVo. La machine de test, Houle (section 4.4.1), utilise le compilateur `g++-4.7.2` avec les optimisations de niveau `-O3`. L'algorithme de tri par insertion classique a été ré-implanté. L'algorithme Quicksort est celui hautement optimisé de la fonction `qsort` de la **libC**. Dans la librairie C++ **std**, `stdsort` est une version aux petits oignons du tri *IntroSort*. Nous utilisons également `mergesort` et `heapsort` les implantations du tri par fusion et du tri par tas fournies par la librairie **BSD**.

K	N	N/K	R(`init`)	R(`add`)
100 000	163 840	1,64	1,84	1,85
200 000	327 680	1,64	1,92	2,61
500 000	786 432	1,57	2,02	2,41
1 000 000	1 572 864	1,57	1,96	2,31
2 900 000	4 456 448	1,54	1,97	2,77
5 000 000	7 864 320	1,57	2,14	2,8
10 000 000	15 728 640	1,57	2,14	2,75

TABLE 4.2 – Somme de K éléments. Comparaison R du temps d'exécution dans un PMA de taille N avec $\tau_h = 0,67$, $\tau_0 = 0,92$, $\rho_0 = 0,08$ et $\rho_h = 0,33$ et dans un tableau dense de taille K.

4.4.1 Houle

La machine *houle* a 2 nœuds NUMA constitués chacun de 4 cœurs Intel Xeon 5520 de fréquence 2.27Go. Les cœurs d'un même nœud partagent un cache L3 de 8Mo et tous les cœurs accèdent à une mémoire de 18Go.

4.4.2 L'inconvénient des trous

Commençons l'évaluation de la solution en nous focalisant sur le désavantage des trous : parcourir les éléments valides est plus coûteux au sein d'un PMA que dans un tableau dense. Premièrement, la présence de trous peut augmenter le nombre d'accès mémoire nécessaires. Deuxièmement, il faut un peu de gestion pour repérer les cases où sont stockés des éléments intéressants et passer de l'une à l'autre.

Nous conduisons une expérience consistant à remplir un PMA avec des éléments choisis au hasard. Nous mesurons le temps d'exécution d'un parcours permettant de calculer la somme des éléments valides et nous le comparons à celui du calcul de la même somme dans un tableau dense. Deux répartitions des éléments au sein du PMA sont étudiées : lorsque le PMA est initialisé avec `pma_init` ou en ajoutant les éléments un par un avec `pma_add_elt` (tableau 4.2). Dans le premier cas, le nombre d'éléments par segment est K/S à un près, les éléments étant simplement répartis dans la structure. Dans le deuxième cas, le PMA est créé par des insertions successives qui peuvent générer des rééquilibrages locaux. Le nombre d'éléments dans chacun des segments varie alors entre les limites minimum et maximum admises pour un segment. Nous détaillons cet aspect section 4.4.2.1.

Le rapport R des temps d'exécution entre le PMA et le tableau dense est supérieur au rapport des tailles du tableau (tableau 4.2). Dans le cas du tableau dense, il suffit en effet de parcourir tous les éléments et de les ajouter. Dans le cas du PMA, il faut ne parcourir que les éléments valides. Cela nécessite un test supplémentaire. Parce que l'opération effectuée, une somme, est très simple, toute opération supplémentaire se

traduit par un surcoût pouvant être important. Dans les premières versions testées de la structure, les champs de bits de validité étaient stockés regroupés par 64 et nous utilisions des itérateurs sur la structure permettant de retourner le prochain élément valide. La gestion des itérateurs conduisait à des surcoûts compris entre 3 et 4. Sans itérateur, le surcoût est plutôt de l'ordre de 2 pour des rapports de taille N/K autour de 1.6. Après une série d'expériences, nous avons compris qu'il faut la combinaison du test et de l'accès mémoire supplémentaire pour expliquer ce temps de parcours tout de même supérieur à l'augmentation de la taille du tableau.

Une solution est de coder la validité d'un élément par une valeur spéciale de sa clé. En effet, l'élément devant être accédé, le test supplémentaire ne génère pas d'accès mémoire supplémentaire. Nous n'avons pas souhaité suivre cette voie pour ne pas contraindre les valeurs des clés. Si les segments étaient systématiquement compactés il serait également possible de ne stocker que le nombre d'éléments valides de chacun, facilitant le travail du *prefetcher*. Pour améliorer les performances, nous avons choisi de remplacer l'utilisation des itérateurs par le simple parcours d'un tableau de booléens stockant la validité de chaque élément sur un entier. Par exemple, pour $K = 1\,000\,000$ le rapport R des temps d'exécution vaut $1,96$ alors que le rapport des tailles $N/K = 1,57$ (tableau 4.2). Pour $K = 100\,000$, le temps d'exécution relatif du PMA est plus faible car la structure tient dans le cache et le test est effectué juste après l'initialisation du PMA.

Il est possible d'être encore plus efficace en codant la validité de 64 éléments sur un même mot de 8o. Le parcours du PMA s'effectue par paquets de 64. L'accès mémoire supplémentaire est factorisé sur 64 éléments. Les surcoûts sont alors de l'ordre du rapport entre la taille N du PMA utilisé et celle K du tableau dense. Par exemple pour $K = 100\,000$, le rapport R des temps d'exécution est $R = 1,54$ dans la situation de l'initialisation par `pma_init`. Pour $K = 1\,000\,000$, le rapport des tailles est $N/K = 1,57$ et le ratio R vaut $1,78$ dans le cas `init` mais $2,16$ lorsque les éléments ne sont pas répartis uniformément dans les segments. Une solution à tester pour limiter l'impact de la dispersion non homogène pourrait être de stocker un mot par segment. Nous n'utilisons pas encore ces versions dans **Storm**, car des optimisations sont encore nécessaires dans les autres parties des algorithmes.

En fait, l'importance des surcoûts mesurés est modérée car l'expérience menée ici est le cas le plus défavorable au PMA. En effet, les éléments utilisés ne contiennent qu'une clé et un identifiant, et l'opération effectuée est une simple somme. Avec des calculs plus complexes sur chaque élément, par exemple la mise à jour de forces ou de nouvelles positions d'un ensemble de particules, le temps des accès mémoire et des tests supplémentaires est masqué par les nombreuses opérations flottantes.

4.4.2.1 Étalement dans les segments

Selon le mode d'initialisation de la structure, ainsi qu'après des insertions, des suppressions ou des voyages, le nombre d'éléments par segments évolue. Par exemple, nous avons mentionné ci-dessus des différences selon le mode d'initialisation choisi. L'impact sur le PMA peut être visualisé par le nombre de segments par nombre d'éléments contenus (figure 4.3).

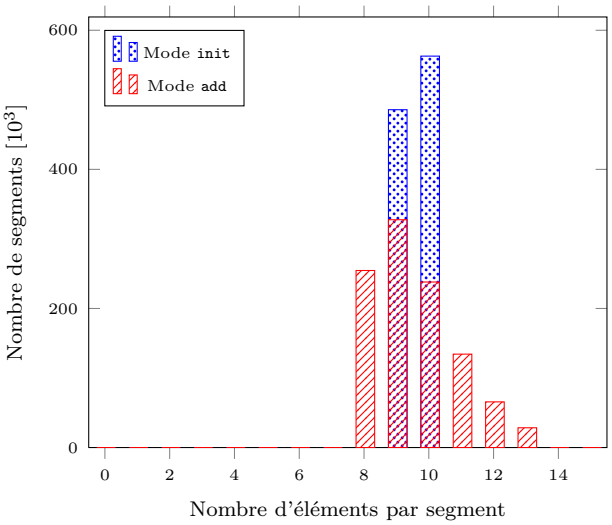


FIGURE 4.3 – Nombre de segments en fonction du nombre d'éléments qu'ils contiennent. Le PMA contient $K = 10\,000\,000$ éléments insérés en mode `init` ou `add`.

Avec $K = 10\,000\,000$ éléments, la taille du PMA est $N = 15\,728\,640$ pour $\tau_h = 0,67$, il y a $S = 1\,048\,576$ segments. En mode dit `init`, les éléments d'une séquence prétriée sont simplement répartis dans les segments qui contiennent alors $K/S = 9,54$ éléments, c'est-à-dire 9 ou 10 éléments. Lorsque les éléments non triés sont ajoutés un à un, leur répartition dans les segments dépend de leur ordre d'arrivée. La taille utilisée par le PMA est augmentée lorsque la place n'est plus suffisante, occasionnant des rééquilibrages globaux. Au-dessus de 8 388 608 éléments, le nombre maximal de segments est atteint et leur capacité est cap = 15. Le dernier rééquilibrage distribue ainsi 8 éléments dans chaque segment. Les éléments ajoutés ensuite viennent incrémenter le compteur d'éléments de certains segments selon leur ordre, d'où une dispersion plus grande du nombre d'éléments par segment.

4.4.3 Protocole d'évaluation de PaVo

Nous nous intéressons dorénavant à la mise à jour d'éléments au sein d'un PMA. Nous décrivons dans cette section le protocole expérimental utilisé par la suite.

L'initialisation, notée INIT, consiste à préparer K éléments constitués d'une clé et d'un identifiant unique compris entre 0 et $K - 1$. La clé de chaque élément est déterminée selon le type de distribution sélectionnée. Les éléments sont insérés et triés dans différentes structures à tester, par exemple un PMA ou un tableau dense et également conservés dans un tableau témoin.

Lorsque la séquence de K éléments est prête, $R = cK$ éléments avec $0 \leq c \leq 1$ sont sélectionnés (étape SELECTION) dans le tableau témoin et une nouvelle clé leur est attribuée. L'identifiant et la nouvelle clé sont stockés dans une *map*, `map_moves` pour être retrouvés efficacement par la suite. Après cette étape commune, pour chacune des structures à tester, nous effectuons d'abord la mise à jour, MAJ, qui consiste en un parcours des éléments de la structure. Si l'identifiant de l'élément est trouvé dans

`map_moves` alors sa clé est changée. Dans le cas du PMA avec PaVo, l'élément est copié dans le tableau des voyageurs puis la fonction `pma_remove_elt_nobalance` est appelée. Dans la dernière étape, TRI, nous retrions les structures. Il peut s'agir de l'appel à une fonction de tri sur un tableau dense ou de l'appel à `pma_add_array_elts` pour le PMA.

L'expérience peut être reproduite plusieurs fois en répétant l'enchainement des étapes SELECTION, MAJ et TRI.

Ce protocole reproduit un schéma correspondant à la détection de plusieurs changements d'ordre dans une séquence d'éléments triés conduisant à un nouveau tri de la séquence. La détection des changements de l'étape MAJ pourrait être combinée par exemple avec le parcours d'un ensemble de particules et le calcul de leurs nouvelles positions. La phase commune SELECTION est construite pour représenter des distributions de voyages au sein d'une séquence d'éléments.

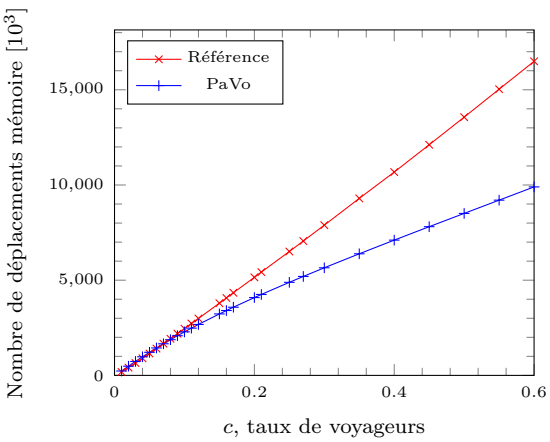
Avec une structure de type PMA non munie de l'algo PaVo, il ne serait pas possible d'effectuer la suppression et la réinsertion des éléments au fur et à mesure du parcours car ces opérations peuvent modifier l'ordre et la position des éléments. Or nous souhaitons parcourir une et une seule fois chaque élément. Dans ce cas, le parcours de la phase MAJ sert à repérer les éléments voyageurs et à stocker dans une autre *map* leur identifiant et leur ancienne clé. À l'étape TRI, chaque élément de cette nouvelle *map* est cherché pour être supprimé dans le PMA et réinséré avec sa nouvelle clé.

4.4.4 Nombre de déplacements

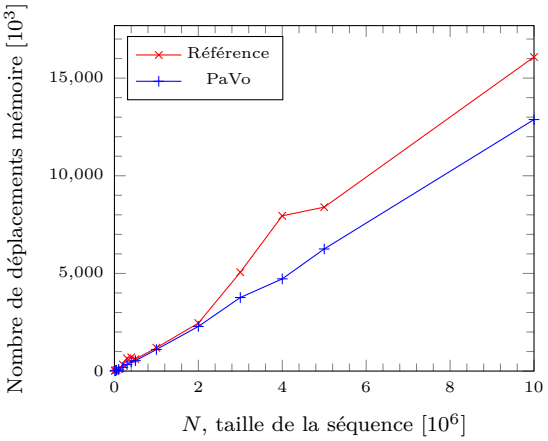
Au chapitre précédent (chap. 3, page 72), nous avons exhibé des courbes de temps d'exécution correspondant à des voyages au sein du PMA en comparant l'algorithme PaVo à la méthode dite de référence par suppression et insertion. Nous présentons ici les mesures correspondantes en ce qui concerne le nombre de copies en mémoire (figure 4.4). Ces expériences ont été faites en suivant le protocole décrit ci-dessus avec des distributions de nombre aléatoires obtenues par un appel au générateur `rand`. Nous relevons la moyenne sur une trentaine de vagues de $R = cN$ voyageurs appliquées consécutivement.

Nous retrouvons des comportements comparables à ceux constatés pour la performance globale. Cependant, le facteur d'écart entre les deux courbes est moins important en ce qui concerne le nombre de déplacements mémoire que pour le temps d'exécution. La complexité purement algorithmique de l'algorithme PaVo est plus faible. De plus, les déplacements mémoire effectués par l'algorithme PaVo sont plus locaux grâce au traitement aggrégé des éléments.

Pour $K = 2\,000\,000$ éléments et $c = 0,1$, soit des paquets de $R = 200\,000$ éléments voyageurs, l'accélération obtenue par l'algo PaVo est de l'ordre de 2. Le rapport du nombre d'instructions exécutées par la méthode de référence et par PaVo vaut $1,63$ et le nombre moyen de cycles par instruction est $1,34$ fois plus important pour la méthode de référence. En multipliant les deux, nous retrouvons le facteur 2 global. Cela confirme la combinaison d'une meilleure complexité algorithmique (diminution du nombre d'instructions) avec des accès mémoire moins coûteux ou moins d'accès mémoire (diminution du nombre de cycles par instruction). En observant d'autres informations



(a) Influence du taux c de voyageurs pour une séquence de $K = 2\,000\,000$ éléments.



(b) Influence de la taille de la séquence pour $c = 0,1$ soit 10% de voyageurs.

FIGURE 4.4 – Nombre de déplacements mémoire effectués par la méthode de référence et par l'algorithme PV pour le tri de $R = cK$ voyageurs dans un PMA de taille N.

fournies par des compteurs de performance, nous constatons que le rapport du nombre total de lectures et d'écritures en mémoire entre les deux versions est de seulement $1,09$, proche du rapport du nombre de déplacements en mémoire $1,07$ mesuré. Il s'agit donc plus d'efficacité des accès en mémoire que de diminution du nombre d'accès. En effet, la méthode de référence génère $3,9$ fois plus de requêtes que PaVo au niveau du cache L3. Rapporté au nombre total d'accès, près d'un tiers des accès atteignent le niveau de cache L3 pour la méthode de référence et seulement 9% pour PaVo dont les accès sont donc plus locaux.

4.4.5 Comparaison avec le tri par insertion et le Quicksort

Nous présentons dans cette section certains résultats publiés dans l'article *A Packed Memory Array to Keep Moving Particles Sorted* [Durand et al., 2012a]. Nous nous intéressons aux tests de tri d'éléments représentés par des entiers, l'application aux particules sera détaillée dans une autre partie (section 4.5).

Nous avions comparé PaVo au tri par insertion et au Quicksort. Par rapport au protocole expérimental décrit ci-dessus, les résultats présentés ont été mesurés en tenant compte uniquement de la partie TRI, ce qui peut introduire de légères variations par rapport aux autres parties de ce chapitre. Les clés des éléments sont, comme précédemment, choisies aléatoirement et les expériences répétées 32 fois pour avoir des moyennes significatives.

Pour 5% de voyageurs, l'algorithme PaVo permet de réordonner une séquence de 10 000 000 éléments en moyenne en moins de 400ms alors que `qsort` utilise de l'ordre de 2 100ms (tableau 4.3). Le tri par insertion est trop coûteux pour être utilisé sur des tableaux de grande taille. En effet, avec seulement 100 000 éléments, l'algorithme requiert de l'ordre de 1s.

Conformément aux attentes, la comparaison entre PaVo et `qsort` montre que PaVo

K	qsort	isort	PaVo
100 000	13.4±0.06	1247±5	3.86±0.02
1 000 000	169±0.6	xx	38.9±0.06
10 000 000	2088±8	xx	372.2±0.5

TABLE 4.3 – Temps d'exécution moyens (ms) pour la mise à jour de 5% d'éléments avec `qsort`, un tri par insertion `isort` et l'algorithme PaVo. Comparaison des étapes TRI uniquement.

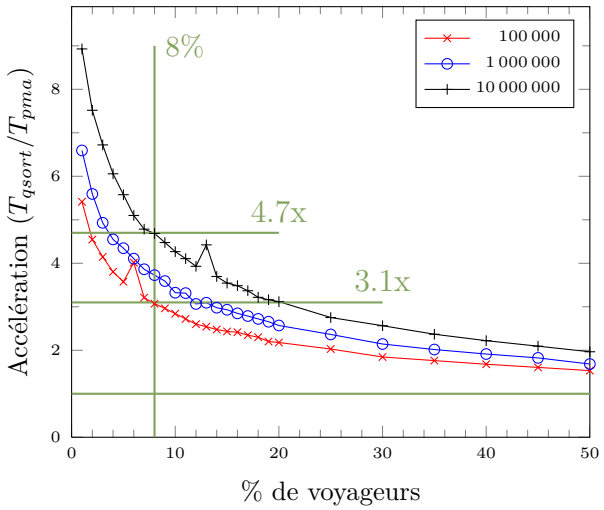


FIGURE 4.5 – Accélération de l'algorithme des paquets de voyageurs dans le PMA par rapport au tri du tableau dense par un `qsort` en fonction du pourcentage de voyageurs, pour différentes tailles de séquences.

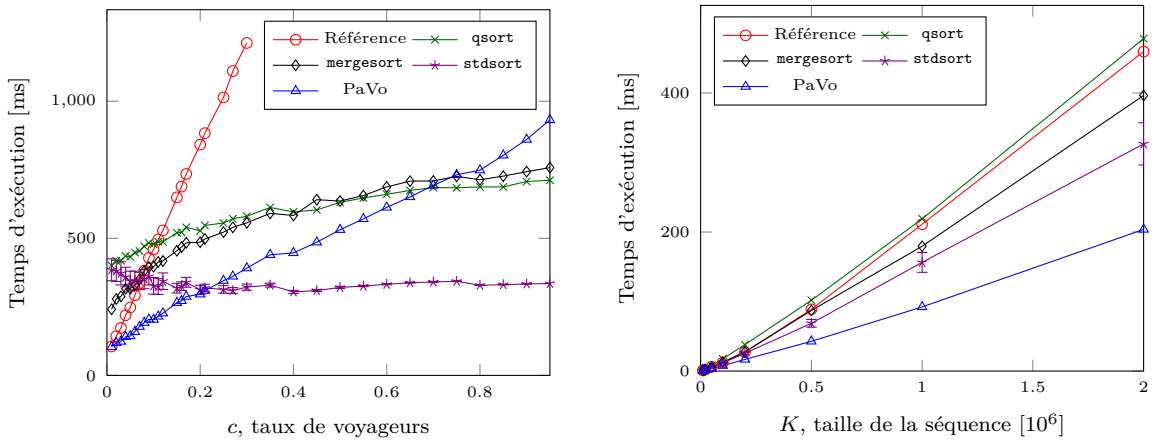
est plus efficace pour de faibles taux de déplacements (figure 4.5). Avec 8% de voyages, l'accélération par rapport au quicksort est de $3,1$ pour $K = 100\,000$ et de $4,7$ pour $K = 10\,000\,000$. Ainsi, l'algorithme PaVo dont l'efficacité repose sur la présence de trous régulièrement répartis est performant par rapport au tri d'un tableau dense dusse-t-il être effectué par le très réputé `qsort`.

Dans les sections suivantes, nous étoffons ces résultats : nous allons nous comparer à d'autres algorithmes, tester l'influence de la taille des éléments puis donner des résultats pour d'autres types de distributions.

4.4.6 Comparaison avec d'autres algorithmes

D'autres algorithmes de tri peuvent être testés sur des tableaux denses. Nous appliquons le protocole décrit à la section 4.4.3 et nous mesurons le temps d'exécution de l'ensemble des parties MAJ et TRI pour chacun des algorithmes testés. Toutes les méthodes testées travaillent sur des tableaux denses sauf la méthode appelée Référence qui représente encore la suppression et insertion dans un PMA et notre algorithme PaVo qui se base également sur un PMA.

Nous représentons la performance des algorithmes à taille K de séquence donnée



(a) Influence du taux c de voyageurs pour une séquence de $K = 2\,000\,000$ éléments.

(b) Influence de la taille de la séquence pour $c = 0,1$ soit 10% de voyageurs.

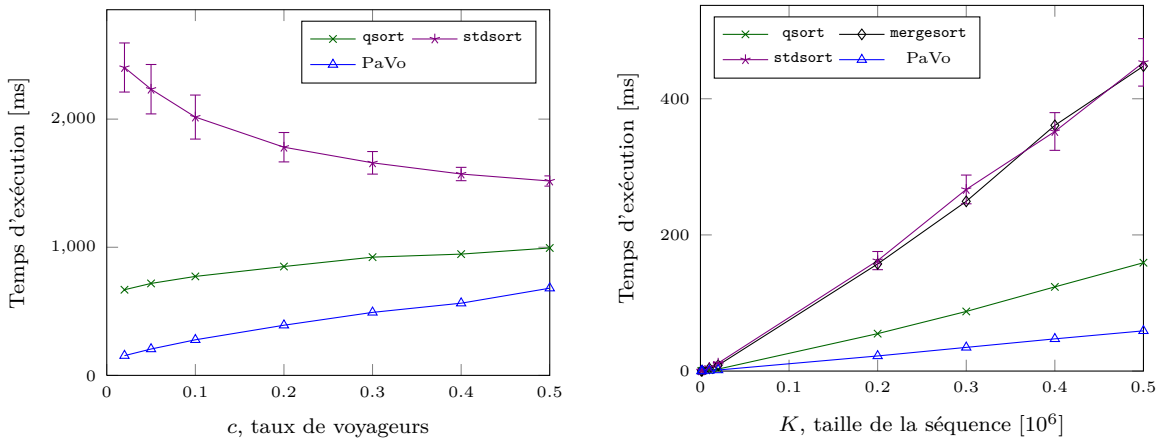
FIGURE 4.6 – Temps d'exécution de différents algorithmes de tris pour $R = cK$ voyageurs.

pour différents taux de voyageurs (figure 4.6(a)) ainsi que l'évolution en fonction du nombre d'éléments pour un taux de voyages fixe (figure 4.6(b)). Nous n'avons pas reporté les résultats de l'algorithme de tri par tas de la version de la librairie **BSD**, `heapsort`, car il peu performant, étant 3 fois plus lent que `qsort`. Les valeurs sont moyennées et, lorsqu'ils ne sont pas insignifiants, les intervalles de confiance à 95% sont indiqués.

Nous pouvons établir plusieurs remarques intéressantes :

- `mergesort` est plus performant que `qsort` jusqu'à 40% de voyageurs.
- la méthode de Référence sur PMA est plus lente que les tris sur tableaux denses au-dessus de 10%.
- PaVo est plus performant que `qsort` et `mergesort` jusqu'à 65% de voyageurs.
- le temps d'exécution de `stdsort` ne dépend pas du taux de voyageurs.
- `stdsort` est le plus rapide de tous à partir de 20% de voyageurs.

La méthode `stdsort` implante le tri IntroSort [Musser, 1997]. Ce tri commence par effectuer une sorte de Quicksort. L'algorithme s'analyse et s'il considère passer trop de temps à avancer dans le déroulement du Quicksort, il peut décider de changer de méthode et d'utiliser une implantation de tri par tas. Ce dernier est sensé être favorable aux situations de séquences presque triées dans sa version adaptative SmoothSort (section 3.1.1). Consignons en outre l'intervalle de confiance à 95% très large de `stdsort` jusqu'à 30% qui pourrait attester d'un changement de méthode en cours de déroulement pouvant se révéler plus ou moins pertinent selon les distributions. Les bons résultats de cette méthode peuvent ainsi s'expliquer lorsqu'il y a peu de changements mais nous pourrions nous attendre à retrouver des résultats quantitatifs similaires à ceux de `qsort` dès qu'il y a trop de voyageurs. Première différence, les opérations de comparaisons de `stdsort` sont *templatées*, là où `qsort` fait appel à une fonction de comparaison fournie par l'utilisateur. Cependant, nous avons comparé l'exécution de l'algorithme PaVo avec un appel à une fonction de comparaison ou via des comparaisons directes de clés et nous n'avons noté que quelques pourcents d'amélioration avec la comparaison



(a) Influence du taux c de voyageurs pour une séquence de $K = 2\,000\,000$ éléments.

(b) Influence de la taille de la séquence pour $c = 0,1$ soit 10% de voyageurs.

FIGURE 4.7 – Temps d'exécution de différents algorithmes de tris pour $R = cK$ voyageurs dans le cas où la taille des éléments correspond à 2 lignes de cache.

directe, n'expliquant pas l'ampleur de la différence relevée ici entre `qsort` et `stdsort`. En fait l'explication réside probablement dans des différences d'optimisations de ces deux fonctions. Il faudrait une analyse poussée des deux programmes pour donner une explication complète ce qui sort de notre cadre d'étude, d'autant que nous allons voir par la suite que `stdsort` ne se comporte pas aussi bien dans toutes les situations (rappel : figure 4.1) !

4.4.7 Impact de la taille des éléments

Jusqu'à présent, nous avons travaillé avec des briques élémentaires ne comportant qu'une clé et un identifiant, soit de taille en mémoire 16o. Selon les applications envisagées, les éléments peuvent transporter avec eux de nombreuses informations supplémentaires.

Nous répétons l'expérience précédente avec plusieurs méthodes mais en travaillant avec des éléments *paddés* de manière à peser chacun pour 2 lignes de cache en mémoire soit 128o. Les résultats ont été produits avec `qsort`, `mergesort`, `stdsort` et PaVo (figure 4.7). Les performances de `stdsort` se dégradent fortement avec l'augmentation de la taille en mémoire. L'accélération de PaVo par rapport à `qsort` passe de 2.3 pour des petits éléments à 2.8 pour $K = 2\,000\,000$ et $c = 0.1$.

Nous présentons également l'impact de l'augmentation de la taille des éléments sur le temps d'exécution de `stdsort` et `qsort` pour une séquence de $K = 2\,000\,000$ éléments avec 10% de voyageurs (figure 4.8). Comme nous l'avions indiqué en introduction de ce chapitre pour une séquence de 2% (figure 4.1), les performances de `stdsort` se dégradent avec la taille des éléments à trier. Les temps d'exécution de PaVo et `qsort` augmentent beaucoup plus légèrement.

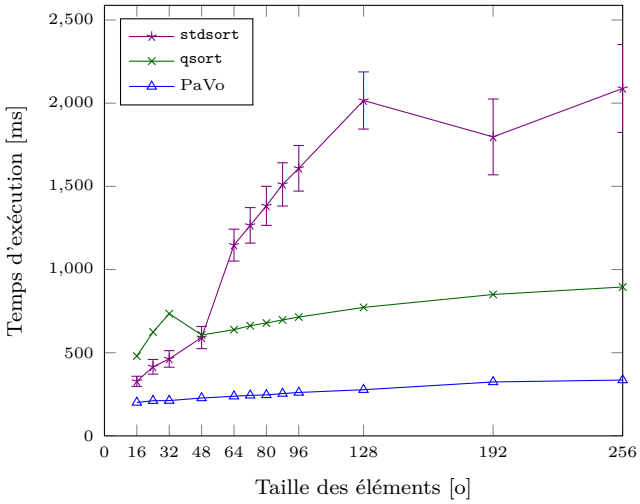


FIGURE 4.8 – Influence de la taille des éléments pour une séquence de $K = 2\,000\,000$ éléments avec 10% de voyageurs.

4.5 Application

L'algorithme PaVo est appliqué à une simulation de fluides basée sur des SPH (section 1.2.2). Nous avons pris le parti d'utiliser la librairie **Storm** dans le logiciel libre **Fluids** (section 1.7.3). Ce programme, destiné à des applications temps-réel ne s'est pas révélé être le meilleur démonstrateur pour notre solution. Cependant le fait même d'obtenir des accélérations dans un contexte difficile montre l'intérêt de l'approche.

Comme ceux de la section 4.4.5, les résultats ont été présentés dans l'article *A Packed Memory Arary to Keeep Moving Particles Sorted* [Durand et al., 2012a].

La simulation utilisée dans les expériences relatées est le *Corner Breaking Dam* : un fluide est relâché alors qu'il se trouve dans un coin supérieur d'une boîte de taille fixe. Les éléments de fluide sont soumis à la gravité et vont se répartir dans la boîte.

4.5.1 Travaux préliminaires

Fluids utilise initialement une taille de cellule qui correspond à deux fois le rayon d'influence des particules SPH. Dans ce cas, seulement 8 cellules doivent être testées à condition de prendre en considération pour les choisir la position de la particule à l'intérieur de la cellule. Cependant les cellules étant plus grosses, plus de particules sont testées inutilement (voir section 1.6.1) : seulement de l'ordre de $\frac{4\pi}{3*8*2^3} \approx 7\%$ des paires testées sont suffisamment proches.

Nous nous sommes ramenés à une grille plus classique avec pour taille de cellule le rayon d'influence des particules. L'étape de détection de collision doit alors vérifier les particules des 27 cellules voisines de chaque particule. Nous avons même divisé par 2 le nombre de tests effectués en tenant compte de la réciprocité des interactions. Pour le même nombre d'interactions, le nombre de paires testées ainsi que le temps total de la détection de collision sont nettement diminués en utilisant ces modifications (4.4).

taille, #nb	paires testées	interactions	ratio	temps
$2h, 8$	36878962	2691708	13.7	298
$h, 27$	16472058	2691708	6.1	232
$h, 14$	8236029	2691708	3.1	145

TABLE 4.4 – Influence de la taille (taille) des cellules utilisées dans la grille de détection sur le nombre de paires testées et le temps d'exécution de l'étape de détection de collision pour 130 000 particules avec Fluids.

De plus, nous avons modifié le logiciel pour qu'il supporte plus de 65 536 particules, limite qui était celle de la version **Fluids v.2**.

Enfin, nous avons ajouté à Fluids la possibilité de trier le tableau de particules et une fréquence, optionnelle, à laquelle le tri est exécuté. Nous nous sommes dotés de ces capacités car le tri des données en mémoire est nécessaire pour augmenter leur localité spatiale. Il est généralement coûteux et il est alors effectué périodiquement (par exemple tous les 100 pas de temps) ce qui permet d'amortir son coût [Ihmsen et al., 2011].

La réorganisation du tableau des particules se fait en 4 étapes dans Fluids. Un premier parcours permet de construire un tableau de paires contenant pour chaque élément sa position en mémoire et la clé de la cellule à laquelle il appartient. Les cellules peuvent être ordonnées selon une courbe en Z ou non. La méthode `stdsort` est utilisée pour trier les paires par ordre croissant en fonction de leurs clés. Les données des particules sont copiées dans un tableau temporaire en utilisant le résultat du tri des paires. Le tableau temporaire est finalement recopié dans le tableau des particules. Dans d'autres situations, un échange de pointeurs pourrait être fait pour éviter la copie mais cela n'était pas immédiat à mettre en place dans **Fluids**. En tous cas, cette méthode rend le tri des particules plus efficace car l'algorithme de tri `stdsort` est alors appliqué sur des objets plus légers (section 4.4.7). Cependant, il faut disposer du double de la taille du tableau des particules pour le tableau temporaire.

Nous avons mesuré le temps respectif des principales étapes du schéma d'intégration des SPH de Fluids (tableau 4.5). En fait, dans le logiciel Fluids, le temps passé pour la détection de collision est largement supérieur au coût du tri qui ne compte finalement que pour quelques pourcents. Ces mesures ont été réalisées en triant le tableau dense des particules avec un `stdsort`. Ces résultats sont différents de résultats classiques en simulation physique de fluides, puisque le tri est ici suffisamment peu coûteux relativement aux autres étapes pour pouvoir être répété à chaque pas de temps sans pénaliser le temps d'exécution d'une itération [Kunaseth et al., 2012].

Nous avons relevé un nombre moyen de paires d'interaction par élément plus élevé que dans la littérature [Ihmsen et al., 2011]. Ce qui explique que l'étape de détection de collision prenne plus de temps puisque nous avons plus de travail à faire. Ces différences peuvent être la conséquence de changements dans le rapport entre le rayon d'influence des particules SPH et leur densité. Il est possible également que les structures utilisées dans les autres simulateurs soient plus efficaces pour l'étape de détection de collision.

étape	%
tri	4.5
détection des voisins	49.1
calcul des pressions	6.3
calcul des forces	36.8
mise à jour des positions	3.2

TABLE 4.5 – Pourcentage de temps passé dans les différentes étapes d'une simulation de type *Corner Breaking Dam* avec 180 000 particules dans Fluids. Les valeurs présentées sont des moyennes mesurées sur la simulation de 4 000 pas de temps dans la version où les particules sont triées à chaque pas de temps.

4.5.2 Mise en place

Le logiciel maintient les paramètres des particules SPH dans un grand tableau dense. Nous avons remplacé le tableau dense de particules par un PMA pour les maintenir triéés selon l'indice de la cellule dans laquelle elles se trouvent. Nous avons également implanté une relation d'ordre optionnelle sur les cellules qui suit une courbe en Z.

Avec le simple remplacement du tableau de particules par un PMA et l'utilisation des itérateurs pour parcourir la structure lors des différentes phases, nous avons noté un surcoût important, principalement visible lors de l'étape de détection de collision. En effet, dans cette étape, les structures sont parcourues de nombreuses fois et les calculs effectués pour chaque élément sont simples ne permettant pas de masquer le surcoût d'un PMA parcouru avec des itérateurs (section 4.4.2).

Nous avons alors utilisé pour cette étape la fonction `pma_valid_pos_ptr` qui permet de compacter rapidement les positions des éléments valides dans le PMA. Dans les situations où les particules sont triées, il est facile de déterminer, en un seul parcours des particules, la liste des particules de chaque cellule. Il suffit pour cela de conserver pour chaque cellule la position de sa première et de sa dernière particule, à la manière dont c'est fait pour des simulations sur GPU (section 2.3.7). Cette amélioration a également été ajoutée dans Fluids uniquement dans le cas où les particules sont triées en mémoire. Le gain mesuré dans Fluids est de 12%. Combiner le compactage des positions valides et l'identification rapide des particules de chaque cellule a permis de gagner 35% dans la version avec le PMA.

4.5.3 Résultats

Nous avons évalué l'impact du tri des particules sur le temps total de la simulation en mesurant le temps d'exécution de chaque pas de temps pour les différentes versions (figure 4.9). Les versions comparées sont Fluids sans tri, avec un tri tous les 100 pas de temps, en triant à chaque iteration ou alors avec le PMA. La simulation étudiée observe le mouvement d'un fluide constitué de 180 000 particules dans une boîte de 132 963 cellules ($141 \times 41 \times 23$).

Le constat évident est que le tri des particules améliore la performance de la simulation. Les particules se déplacent dans la boîte et finissent par se mélanger. Des

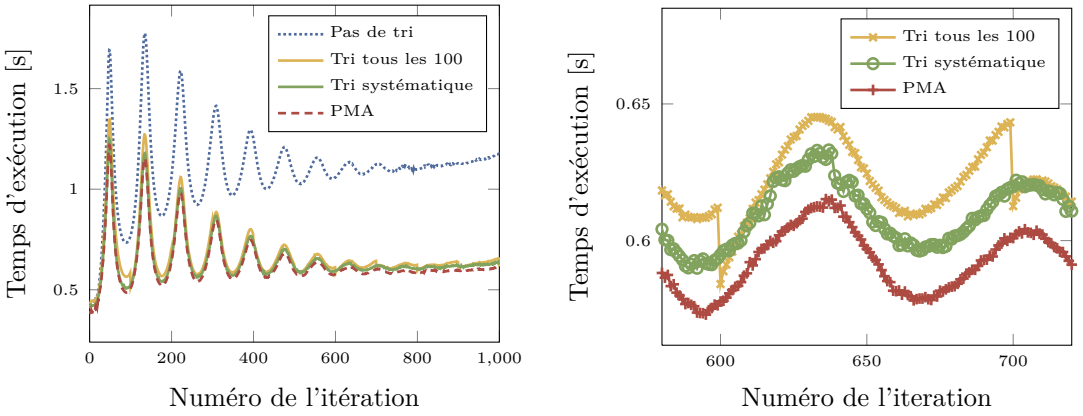


FIGURE 4.9 – Temps d'exécution total en seconde pour une simulation de type *Corner Breaking Dam* avec 180 000 particules. Les différentes versions comparées incluent l'exécution sans tri, le tri tous les 100 pas de temps, le tri tous les pas de temps et en utilisant le PMA.

particules proches en mémoire sont alors loin dans l'espace aussi les interactions entre les particules génèrent-elles plus de défauts de cache. Le temps d'exécution de la version non triée est, à la fin de la simulation, de l'ordre du double du temps d'exécution des versions triées.

Puisque le tri est suffisamment léger, se limiter à ne trier que tous les 100 pas de temps n'est pas nécessaire. Par rapport au cas non trié, nous observons que le tri périodique permet toutefois d'éviter le mélange des particules et l'explosion du temps d'exécution.

Enfin, la version utilisant le PMA se révèle autour de 2% meilleure. Étant donné le temps passé dans les différentes étapes de la simulation (section 4.5.1) et le faible poids de l'étape de tri, ce résultat est intéressant et dévoile le potentiel de la structure.

Finalement, comment se comporte l'algorithme PaVo lorsqu'il est confronté à de véritables voyageurs ? La figure 4.10 illustre la comparaison entre le tri des particules par `stdsort` et leur mise à jour au sein d'un PMA avec notre algorithme fétiche. Attention, les résultats ne peuvent pas être directement comparés à ceux de la figure précédente car il y a 2 900 000 particules dans cette simulation. Au début de la simulation, lorsque le fluide est laissé libre en haut de la boîte, les particules SPH sont soumises à la gravité et jusqu'à plus de 30% d'entre elles changent de cellule à chaque pas de temps. Par la suite, le pourcentage de voyageurs est compris entre 5 et 10%. Le temps d'exécution de l'algorithme est bien lié au pourcentage de voyages à chaque iteration.

Le temps d'exécution de PaVo est supérieur au temps de `stdsort` lorsque le taux de voyageurs dépasse les 30%. Les mesures de temps présentées ici sont réalisées avec, pour le tri via `stdsort`, un tri de paires position-clés comme expliqué plus haut (section 4.5.1). Comme nous l'avons dit, cette technique avantage `stdsort` par rapport à notre algorithme puisque les éléments triés sont légers.

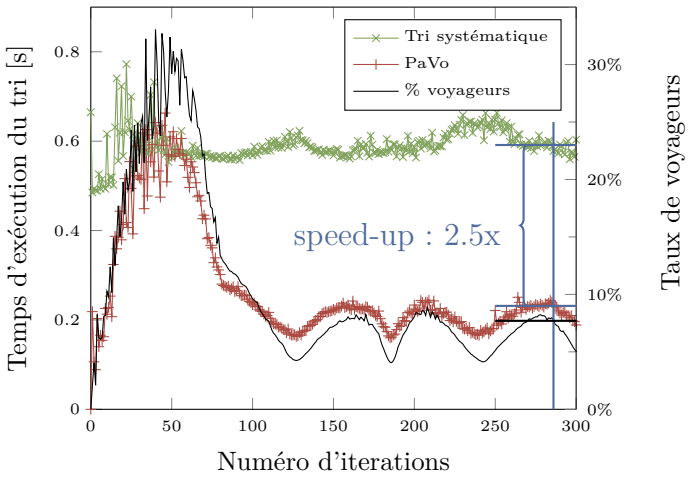


FIGURE 4.10 – Temps du tri du tableau des particules avec `stdsort` et PaVo dans une simulation de type *Corner Breaking Dam* avec 2 900 000 particules. Le taux de voyageurs est également présenté pour chaque iteration.

4.5.4 Extraction de distributions

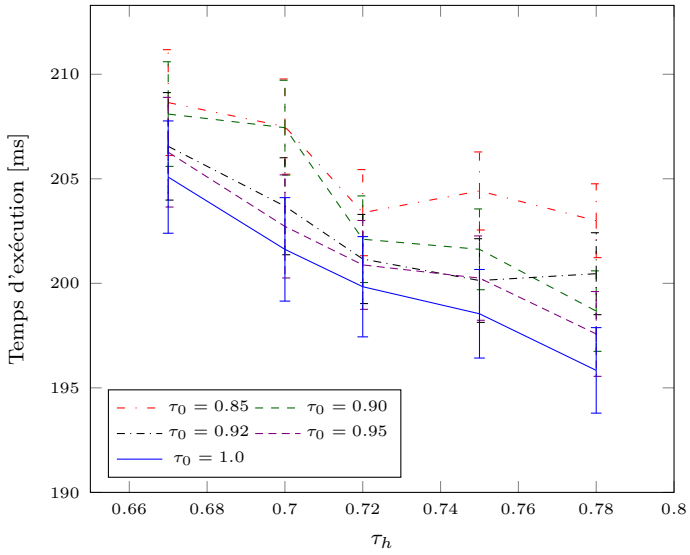
Nous avons extrait de cette dernière simulation des données permettant de reconstruire le mouvements des particules entre les pas de temps 190 et 214 correspondant à des taux de déplacements situés entre 6% et 10%. Nous avons pour cela conservé pour chaque particule l'identifiant géométrique ou selon une courbe en Z de la cellule dans laquelle elle se trouve.

4.6 Analyse avancée de PaVo

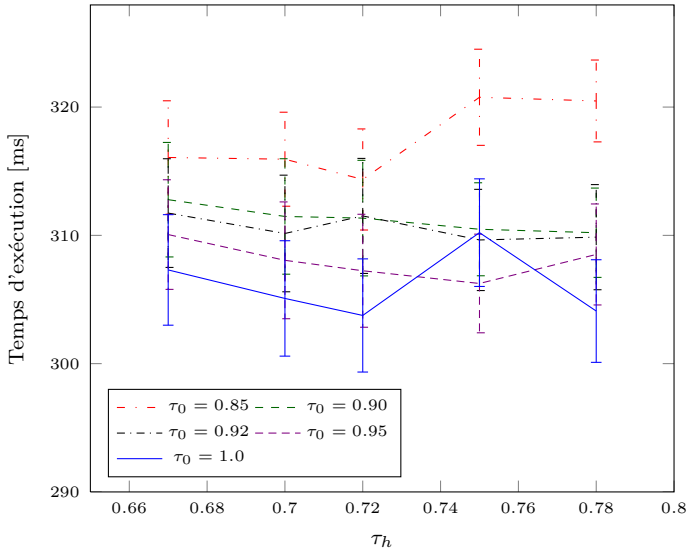
4.6.1 Influence des densités

Quelle est l'influence des densités choisies pour le PMA sur le temps d'exécution du tri d'éléments ? Le spectre des possibilités semble vaste mais toutes les combinaisons ne sont pas possibles. En pratique, les densités minimum au niveau du segment ρ_0 et du PMA complet ρ_h sont choisies par symétrie autour de $0,5$ par rapport aux densités maximum τ_0 et τ_h. Ainsi, et pour vérifier $2\rho_h < \tau_h$ (équation 3.2), la densité minimale pouvant être choisie comme maximum pour le PMA total est $\tau_h = 0,67$, donnant $\rho_h = 0,33$. Nous avons sélectionné quelques valeurs pour τ_0 et τ_h et reproduit notre expérience en mesurant le temps d'exécution pour des éléments de taille 128o (figure 4.11).

Avec une distribution aléatoire sélectionnant 5% de voyageurs sur $K = 2\,000\,000$ éléments (figure 4.11(a)), la tendance est assez claire. Les densités les plus élevées au niveau du segment sont les plus favorables. Pour des distributions de voyages uniformes, puisque les échanges sont globalement répartis, être souple au niveau du nombre maximum et minimum d'éléments par segment permet de réduire le nombre de rééquilibrages. D'autre part, plus la densité maximum au niveau du PMA augmente, plus le temps d'exécution diminue. En fait, pour 2 000 000 éléments, le calcul de la taille



(a) Distribution uniforme, $K = 2\,000\,000$ avec 5% de voyageurs.



(b) Distribution extraite de la simulation dans Fluids, $K = 2\,900\,000$ élements.

FIGURE 4.11 – Temps d'exécution pour différentes configurations des densités maximum du segment τ_0 et du PMA total τ_h. Les éléments ont pour taille 128o.

du PMA pour des densités maximum de $0,67$ et $0,70$ donne le nombre de segments $S = 262\,144$ et des capacités par segment de 12 et 11 éléments respectivement. Pour les densités supérieures, le nombre de segments est divisé par 2 et les capacités se trouvent être comprises entre 22 et 20. Ce qui assure plus de souplesse tout en réduisant l'espace supplémentaire utilisé.

Avec la distribution extraite de la simulation de particules dans Fluids, le nombre d'éléments est de $2\,900\,000$. Quelle que soit la densité du PMA total le nombre de segments est $S = 262\,144$. La densité choisie n'influe que sur la capacité des segments : 17, 16, 16, 15 et 15 pour $\tau_h = 0,67, 0,70, 0,72, 0,75$ et 0.78. Ce qui explique le peu de différences constatées sur les différentes courbes avec l'augmentation de τ_h (figure 4.11(b)).

Nous retrouvons toutefois la tendance à la diminution du temps d'exécution avec l'augmentation de la densité maximum au niveau du segment. Sans être extrême, le couple de valeur $\tau_h = 0,75$ et $\tau_0 = 0,95$ semble constituer un bon compromis dans ces différentes situations.

4.6.2 Diminution de l'empreinte mémoire

L'étape de rééquilibrage telle que décrite plus tôt dans ce chapitre (section 4.3.3) utilise en pratique une queue tampon dont la taille est difficile à calibrer. Pour éviter des réallocations en mémoire, nous utilisons un tableau de la taille totale du PMA. Bien que l'espace mémoire utilisé reste linéaire avec le nombre d'éléments, doubler la mémoire du PMA est un peu excessif.

Nous avons finalement mis au point un algorithme pour effectuer le rééquilibrage de manière plus économe en mémoire. Nous utilisons d'une part le tableau des voyageurs lorsqu'il s'agit de faire de la place pour l'insertion d'un voyageur, et d'autre part une petite queue dont la taille se limite au nombre d'éléments maximum par segment moins le nombre d'éléments minimum plus 1. Cet espace permet de gérer la diminution du nombre d'éléments d'un segment suite à un rééquilibrage d'une fenêtre. Les éléments supplémentaire doivent être stockés en respectant leur ordre avant de passer au traitement du segment suivant.

L'algorithme précis, un peu plus complexe, génère plus de comparaisons ce qui occasionne une légère diminution des performances. En pratique, il s'agit seulement de quelques pourcents pour passer d'un espace mémoire supplémentaire en $\Theta(N)$ à un espace borné par $O(\log N)$. Les expériences conduites dans les sections précédentes l'ont été avec l'ancien algorithme mais au chapitre suivant nous utiliserons le nouveau. La diminution de l'empreinte mémoire est encore plus intéressante sur des architectures parallèles car cela permet de réduire la quantité de transferts mémoire.

4.7 Discussion

Nous avons montré l'intérêt d'une structure de type PMA munie de l'algorithme PaVo pour accélérer le tri de séquences presque triées. Nous avons étudié le comportement de l'algorithme dans de nombreuses situations et donné les clés pour comprendre la structure et ses bonnes performances. Le travail dans une structure non dense, avec des trous comme le PMA, permet d'accélérer le maintien de l'ordre entre les éléments. Les accélérations obtenues augmentent avec la taille des éléments mis à jour en limitant le nombre de déplacements mémoires et en favorisant la localité spatiale des accès.

Trier des particules dans une simulation physique augmente les performances générales de l'ensemble des étapes de calculs. De plus, utiliser PaVo nous permet de tirer partie de la cohérence temporelle de ces simulations puisque nous ne traitons que les éléments le nécessitant. Grâce à PaVo, le surcoût due à la conservation de l'ordre entre les particules est limité même si de nombreux paramètres sont associées aux éléments. Le choix de cet algorithme est à privilégier dans ce contexte par rapport par exemple à `stdsort` dont l'efficacité décroît fortement avec l'augmentation de la taille des éléments.

Nous avons indiqué au fil des sections des optimisations qui pourraient être réalisées pour augmenter les performances de la structure. Par exemple en modifiant la manière de stocker en mémoire l'arbre du nombre d'éléments ou en améliorant encore les performances de l'implantation grâce à l'utilisation d'opérations efficaces sur les champs de bits représentant la validité des emplacements du PMA. Ces aspects feront l'objet de travaux ultérieurs visant à finaliser la librairie **Storm**.

Dans le cas où seules des modifications de l'ordre sont appliquées (pas d'insertion de nouvel élément, ni suppression), nous avons commencé à développer une méthode "en-place" de PaVo. L'idée est d'éviter la copie des voyageurs dans un tableau annexe. Puisque tous les éléments sont contenus dans le PMA, les voyageurs pourraient être simplement étiquetés comme tels et déplacés dans la structure pour rétablir l'ordre. L'enjeu est de limiter l'espace total nécessaire, pour le moment en $O(R + K)$ avec R le nombre d'éléments se déplaçant et K le nombre d'éléments de la séquence totale. La difficulté est d'effectuer la partition des éléments en minimisant le nombre d'échanges en mémoire. Ces travaux seront poursuivis par la suite.

Nous ne nous sommes pas préoccupés dans la méthode proposée de la stabilité du tri. Cette question pourrait faire l'objet d'études intéressantes. En ce qui concerne les applications comme la simulation physique ce point n'est pas primordial car typiquement, les éléments d'une même case, ou clé, n'ont pas de raison spéciale d'être organisés selon un ordre particulier.

Les voyageurs tirent partie de l'infrastructure 5

Les chapitres précédents ont présenté puis décrit PaVo, un algorithme de tri de séquences presque triées. L'efficacité de l'algorithme repose sur une structure à trous permettant de limiter le nombre de déplacements mémoire nécessaires ainsi que sur l'aggrégation des requêtes de voyages au sein de la structure. Puisqu'avec des trous moins de mouvements de données sont nécessaires, ces trous devraient également participer à la réduction des dépendances de données en parallèle. C'est l'idée que nous allons tester dans ce chapitre. Nous verrons que PaVo, l'algorithme des Paquets de Voyageurs profite pleinement de l'infrastructure mis à sa disposition...

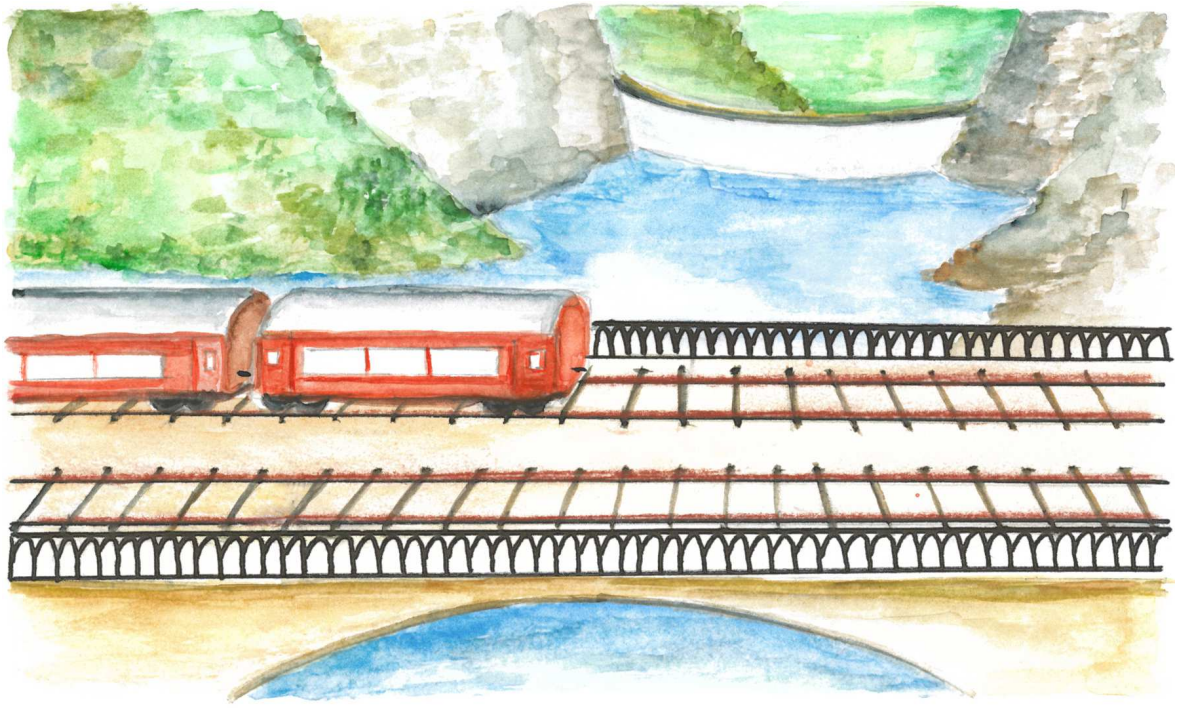


FIGURE 5.1 – Les voyageurs sont transportés dans des wagons sur les lignes parallèles du chemin de fer.

Au programme de ce chapitre : la description de l'algorithme parallèle (section 5.1), un zoom sur l'utilisation de la mémoire (section 5.2), une présentation de l'adaptation de l'ordonnanceur de vol et de la prédistribution des données (section 5.3.1). Nous évaluons ensuite les performances de l'algorithme PaVo parallèle (section 5.4).

5.1 PaVo sur des rails parallèles

Nous avons décrit en détail les différentes étapes de l'algorithme dans le chapitre précédent (section 4.2). Nous présentons dans cette section l'algorithme parallèle correspondant.

5.1.1 Approche douce

PaVo est appliqué en deux étapes principales : la détection des voyageurs effectuée par un parcours de la structure, étape MAJ, et la réinsertion des éléments dans le PMA, étape TRI (section 4.3.2).

MAJ se parallélise facilement par une découpe de la structure en morceaux contenant un ou plusieurs segments du PMA. Le travail à effectuer pour chaque segment est indépendant. Le nombre important de segments (voir tableau 4.1, page 78) suffit à nourrir les différents travailleurs. Une parallélisation plus fine nécessiterait des communications car le nombre d'éléments du segment doit être mis à jour ainsi que la clé du premier élément du segment.

Pour éviter des coûts de communication entre les threads lors de la partie MAJ, chacun copie les voyageurs qu'il détecte dans un espace personnel. Avant de passer à l'étape suivante, il faut rassembler les éléments. Par rapport à l'algorithme séquentiel, cela rajoute une étape appelée RED pour réduction.

L'algorithme séquentiel de la partie TRI est une descente récursive dans l'arbre représentant virtuellement le PMA. Chaque appel récursif est indépendant puisque s'opérant sur des parties différentes de la structure et du tableau des voyageurs. Nous créons des tâches pour chacun de ces appels. L'efficacité de l'algorithme séquentiel repose sur le fait que peu d'étapes de rééquilibrage sont nécessaires et que ces dernières sont effectuées la plupart du temps sur de petites parties de la structure. Ainsi cette étape est-elle actuellement traitée séquentiellement, de la même manière que l'insertion des voyageurs au niveau d'un segment.

5.1.2 Passage à l'échelle

L'algorithme parallèle décrit ci-dessus a deux principaux défauts qui ne permettent pas une utilisation efficace de plusieurs cœurs. En un, l'étape RED est séquentielle. Un calcul de préfixe pourrait être fait pour que chacun recopie ses éléments à leur emplacement de destination. Cependant, l'étape resterait coûteuse, en particulier sur des architectures à mémoire hiérarchique.

Le deuxième point est le manque de parallélisme des premières étapes de la descente dans l'arbre du PMA. Avant de créer les deux tâches indépendantes correspondant à la division du problème, il faut effectuer la partition de tous les éléments voyageurs, c'est-à-dire regrouper contigüment tous les éléments inférieurs au pivot. Lors des premières étapes, cette opération est effectuée par 1 puis 2 puis 4 (puis ...) cœurs seulement. La quantité de parallélisme exhibée dans les premières phases est ainsi limitée.

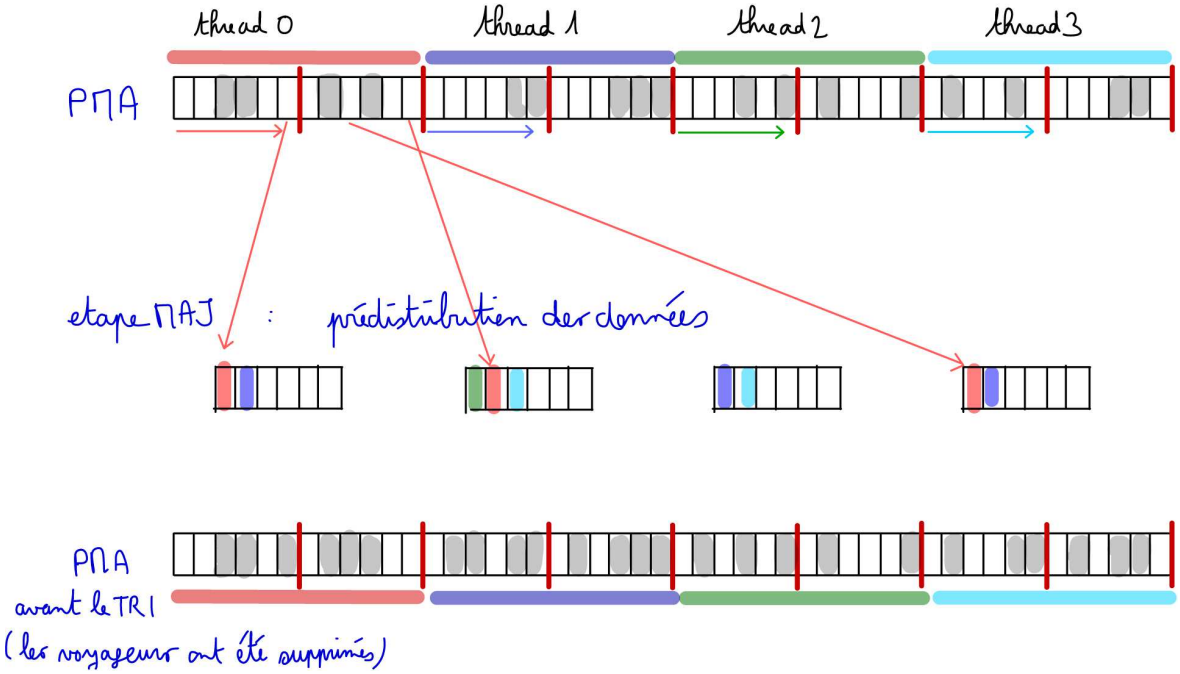


FIGURE 5.2 – Principe de l'algorithme parallèle avancé.

5.1.3 Augmenter le parallélisme

En fait, nous pouvons tirer partie de la structure de données pour nettement augmenter le parallélisme de ces étapes. Comment ? À la manière d'un *Bucket Sort* : les éléments voyageurs sont pré-partitionnés en utilisant les valeurs des différentes sous-parties de la structure (figure 5.2).

Le PMA est divisé en autant de régions que le nombre de travailleurs pour permettre d'exécuter la partie TRI de l'algorithme parallèle indépendemment sur chacun des sous-arbres. Les valeurs des pivots des différentes régions ainsi définies sont utilisées dans l'étape MAJ pour distribuer chacun des voyageurs vers son sous-arbre de destination. L'étape TRI commence alors par la simple vérification de la consistance du nombre d'éléments avec les densités des fenêtres. Seulement si elles ne sont pas respectées, une réduction est effectuée sur l'ensemble des régions correspondantes et un rééquilibrage séquentiel est déclenché sur cette fenêtre. Lorsque le nombre de voyageurs à insérer dans chacune des fenêtres respecte les seuils de densités et que la fenêtre correspondant à une région est atteinte, alors l'algorithme parallèle décrit précédemment est déroulé.

5.1.4 Précisions

Soit p le nombre d'entités de calcul indépendantes. La structure du PMA est virtuellement séparée en $\llbracket p \rrbracket$ [1] régions. Les pivots correspondant à chaque premier segment des fenêtres correspondant aux régions sont consultés pour pré-répartir les voyageurs directement dans leur espace de destination.

1. $\llbracket x \rrbracket$ représente la plus petite puissance de 2 supérieure ou égale à x.

Cela nécessite une communication entre les travailleurs pour savoir à quelle position insérer le voyageur en cours de traitement. En pratique, nous utilisons un compteur du nombre d'éléments déjà insérés dans chaque région. Un nouvel élément peut être ajouté après la réservation d'un emplacement via un `atomic_add`. Pour éviter le faux partage de lignes de cache, les compteurs doivent être paddés de manière à être chacun distribué sur une ligne de cache différente. Nous avons relevé une dégradation très importante des performance lors d'une mauvaise allocation de ces compteurs. Pour améliorer encore les performances, nous combinons les requêtes : chaque thread attend d'avoir plusieurs voyageurs à insérer dans une même région avant de réserver les emplacements correspondants et de les recopier.

Lors de l'étape TRI, le nombre de voyageurs à insérer dans chaque région est connu, et les voyageurs rassemblés par région. De manière similaire à ce qui est fait pour le nombre d'éléments de la structure (section 4.2.4), nous construisons l'arbre du nombre de voyageurs, pour éviter de les recompter à chaque niveau.

5.2 Mémoire

Nous présentons dans cette section des premières mesures de performances autour de l'utilisation de la mémoire. Nous commençons par une brève présentation des deux architectures utilisées, IdFreeze (section 5.2.1) et IdRouille (section 5.2.2).

Les expériences réalisées dans ce chapitre reprennent le protocole expérimental présenté au chapitre précédent (section 4.4.3, page 4.4.3). Une map est construite avant l'étape MAJ qui stocke les identifiants des voyageurs et leur nouvelle clé. La map est consultée pour la mise à jour pendant un parcours de la structure.

5.2.1 IdFreeze

La machine *IdFreeze* est une architecture AMD 48 cœurs (nom de code Magny-Cours - AMD 6174) avec 256Go de mémoire. Chaque cœur est cadencé à 2.2GHz. La machine a 8 nœuds NUMA, chacun d'entre eux est lié à 6 cœurs partageant un cache L3 de 5Mo. Les caches L1 et L2 sont privés à chaque cœur et sont de taille 64ko et 512ko respectivement.

5.2.2 IdRouille

IdRouille est une machine Intel 32 cœurs qui possède 4 nœuds NUMA avec chacun 8 cœurs Intel Xeon X7560 à 2.27GHz accèdant une mémoire de 64Go. Les cœurs d'un même nœud NUMA se partagent un cache L3 de 24Mo et disposent de caches privés L1 et L2 (32ko et 256ko).

5.2.3 Sur un banc

Si la mémoire des machines NUMA peut être accédée par tous les cœurs, elle est organisée en différents *bancs*, associés à chacun des nœuds NUMA. Les cœurs d'un même nœud accèdent plus rapidement à la mémoire qui leur est associée qu'à celle des autres nœuds. Nous commençons par étudier le comportement de PaVo sur un seul banc mémoire.

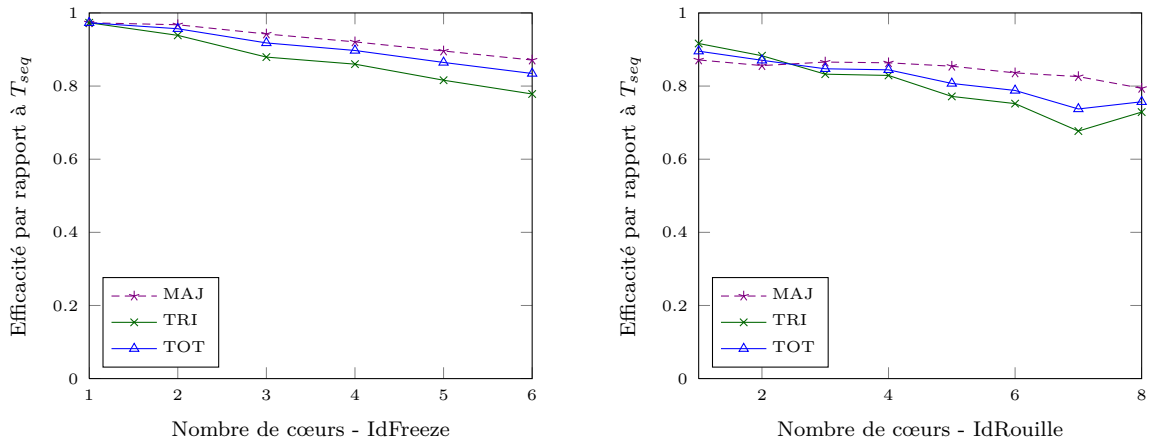


FIGURE 5.3 – Efficacité de l'algorithme parallèle simple utilisant les cœurs d'un même nœud NUMA. Séquence de $K = 2\,000\,000$ éléments avec 10% de voyageurs.

Nous évaluons la performance des étapes MAJ et TRI dans la situation de l'algorithme parallèle basique (section 5.1.1). Nous comparons leur efficacité parallèle, c'est-à-dire le rapport entre le temps d'exécution séquentiel T_{seq} et le temps d'exécution parallèle divisé par le nombre de cœurs utilisés (figure 5.3). Les temps de l'étape RED ont été ajoutés à l'étape MAJ.

Le surcoût parallèle donné par le rapport entre T_{seq} et T_1 est plus faible sur la machine IdFreeze ($2,5\%$) que sur IdRouille ($11,7\%$). Sur cette dernière, c'est la phase MAJ qui présente la plus grande dégradation de performance ($14,7\%$). L'efficacité parallèle sur un même banc est de $83,4\%$ pour $p = 6$ sur IdFreeze et de $78,8\%$ pour $p = 6$ et $75,7\%$ pour $p = 8$ sur IdRouille.

La diminution de l'efficacité de l'étape TRI est plus importante que celle de l'étape MAJ. L'étude des mesures données par les compteurs de performance sur cette partie montre que le nombre d'instructions du premier cœur, celui qui traite les premières étapes, est presque constant à partir de 4 cœurs utilisés. Les threads supplémentaires se partagent les tâches à exécuter une fois qu'un niveau de parallélisme suffisant est atteint, limitant l'efficacité de la méthode.

Les tests avec l'algorithme avancé (section 5.1.3) montrent une légère amélioration de l'efficacité parallèle (figure 5.4). Sur les deux architectures nous observons une inversion des courbes d'efficacité entre l'étape MAJ et l'étape TRI. En fait, avec cet algorithme, l'étape MAJ effectue le pré-partitionnement afin que tous les threads puissent commencer l'étape TRI simultanément. L'objectif est d'améliorer l'efficacité parallèle en découpant la partie peu parallèle du début de la descente de l'arbre du PMA. Ainsi, la partie MAJ

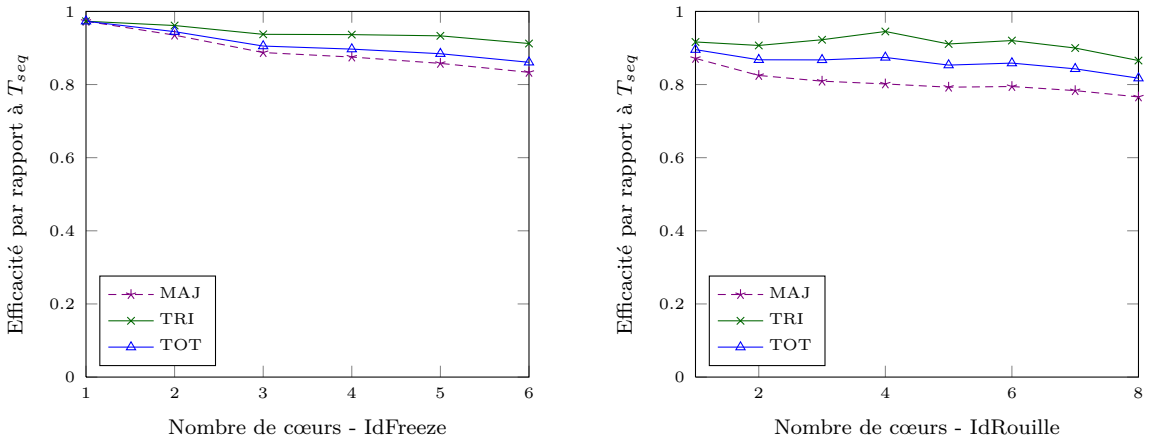


FIGURE 5.4 – Efficacité de l'algorithme parallèle avancé avec pré-distribution des voyageurs dans les différentes sous-fenêtres, pour $K = 2\,000\,000$ et 10% de voyageurs et sur un seul banc NUMA.

est plus longue avec plus de communications alors que le point d'entrée de la partie TRI est constitué de plusieurs sous-ensembles indépendants. Lors de l'étape TRI, le nombre d'instructions exécutées est mieux réparti entre eux : les cœurs se partagent mieux le travail. Même avec un petit nombre de cœurs d'un même nœud NUMA, l'algorithme avancé est plus efficace : 86% pour $p = 6$ sur IdFreeze, $85,9\%$ et $81,7\%$ pour 6 et 8 cœurs avec IdRouille.

5.2.4 Impact du placement mémoire

Quel est l'impact du placement des données en mémoire ? Pour le savoir, nous pouvons allouer les pages mémoire correspondant aux données sur un nœud différent sur lequel sont placés les cœurs effectuant les calculs (tableau 5.1). Par exemple, sur la machine IdFreeze, nous plaçons 6 cœurs sur le nœud NUMA 5, puis nous associons toutes les données sur d'autres bancs.

Nœud	IdFreeze	IdRouille
0	58.80±0.06	17.23±0.08
1	59.23±0.06	17.23±0.08
2	60.20±0.05	15.71±0.05
3	59.93±0.05	17.02±0.08
4	55.94±0.06	
5	43.99±0.04	
6	75.84±0.06	
7	76.01±0.05	

TABLE 5.1 – Temps d'exécution (ms) de l'étape MAJ de l'algorithme avancé pour $K = 2\,000\,000$ et 10% de voyageurs. Les données sont allouées sur un banc différent pour chaque expérience. Les 6 threads d'IdFreeze sont alloués sur le nœud 5 et les 8 threads d'IdRouille sur le nœud 2 de cette machine.

Sur la machine IdFreeze à l'architecture très particulière, nous observons plusieurs niveaux de performance. En fait, le nombre limité de contrôleurs mémoire (ou *Bridge HyperTransport*) fait que certains nœuds n'ont pas d'accès direct entre eux. Il leur faut alors "rebondir" par un autre nœud. Rapportés au temps de l'étape lorsque la mémoire est associée au banc courant, nous retrouvons un niveau d'échange $1,27$ fois plus lent pour le nœud 4 sur la même socket que le nœud 5. Les nœuds 0, 1, 2 et 3 présentent un facteur de l'ordre de $1,34$. Les nœuds les plus distants sont 6 et 7 avec un facteur s'élevant à $1,72$. Sur la machine IdRouille, il n'y a que deux niveaux : accès au nœud sur lequel les threads sont associés (nœud 2) ou accès à un nœud distant avec un facteur de perte de $1,09$.

5.2.5 Augmentation du nombre de processeurs

Nous étudions le comportement de l'algorithme parallèle amélioré PaVo lors de l'augmentation du nombre de cœurs et de l'utilisation de différents nœuds NUMA. Nous mesurons l'accélération parallèle de PaVo, c'est-à-dire le temps d'exécution séquentiel T_{seq} divisé par le temps d'exécution de p cœurs T_p (figure 5.5). Dans cette expérience, la mémoire n'est pas contrôlée : en pratique, elle est allouée avec une stratégie *first-touch* sur le premier nœud utilisé (souvent 0).

Nous constatons sur les deux architectures un tassement de l'accélération obtenue, quelle que soit la taille de la séquence. À titre de comparaison, pour une séquence de $K = 10\,000\,000$ éléments sur la machine IdRouille, l'algorithme parallèle basique atteint une accélération sur 32 cœurs de $9,9$ au lieu de $12,7$ pour l'algorithme parallèle amélioré.

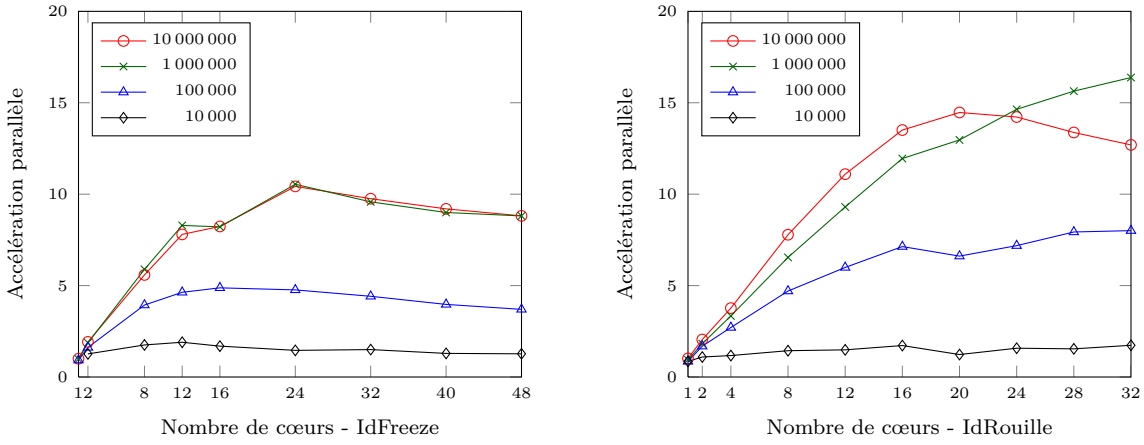


FIGURE 5.5 – Accélération de PaVo en fonction du nombre de cœurs utilisés pour différentes tailles de séquence avec 10% de voyageurs.

C'est l'étape MAJ qui se comporte le moins bien avec l'augmentation du nombre de cœurs. En fait, cette étape est celle qui fait le plus d'accès mémoire. Comme nous l'avons dit, les accès mémoire distants sont plus lents. Or dans cette expérience, la plupart des threads font des accès distans du fait de la stratégie d'allocation par défaut. De plus, la bande passante mémoire des différents contrôleurs mémoire est limitée.

Notamment, sur la machine IdRouille avec $K = 10\,000\,000$ augmenter le nombre de cœurs au-delà de 20 se traduit par une augmentation du temps de calcul. Pourtant, la séquence 10 fois plus petite atteint de meilleurs accélérations. Il ne s'agit donc pas d'un manque de parallélisme mais bien d'une limitation due aux accès mémoire. Nous allons nous intéresser de plus près à ces aspects.

5.2.6 Stratégies d'allocation

Pour éviter les écueils de l'expérience précédente, il existe d'autres politiques d'allocations mémoire, comme la politique `interleave` fournie par `numactl` (ou `hwloc`) qui permet de répartir la mémoire allouée dynamiquement sur les différents bancs mémoire utilisés à raison d'une page par banc de manière cyclique. Nous verrons qu'il est également possible d'associer spécifiquement la mémoire uniquement à l'un ou l'autre des nœuds NUMA avec `membind`.

Nous reprenons l'expérience de la section précédente avec la politique d'allocation `interleave` (figure 5.6). Nous observons une forte amélioration de l'accélération parallèle dès 8 cœurs. Les séquences de grande taille passent mieux à l'échelle, affichant des efficacités parallèles de 55% sur 48 cœurs (IdFreeze), et de 87, 8% sur les 32 cœurs de la machine IdRouille. Les éléments utilisés dans cette expérience sont les briques de base de seulement 16o.

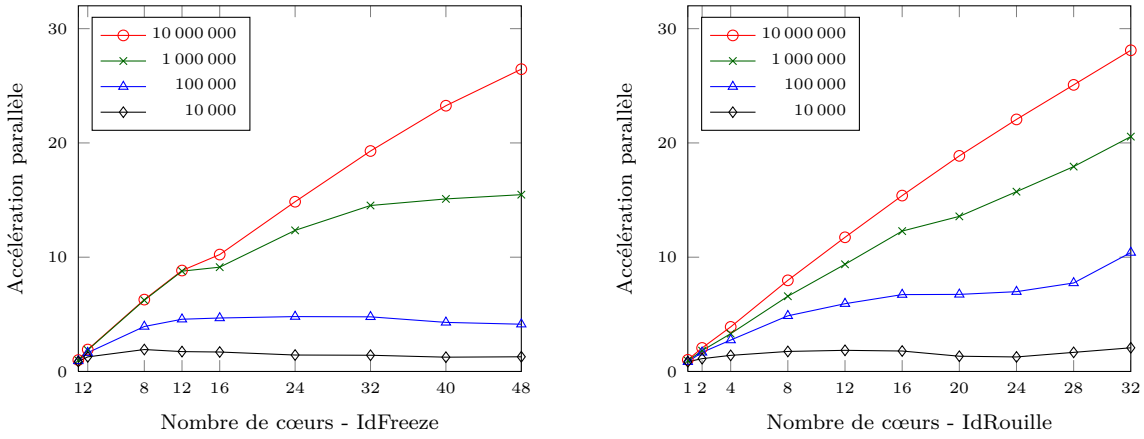


FIGURE 5.6 – Accélération de PaVo en fonction du nombre de cœurs utilisés pour différentes tailles de séquence avec 10% de voyageurs lorsque la stratégie `interleave` est utilisée.

5.2.7 Allocation par bloc

L'efficacité parallèle est meilleure sur la machine IdRouille dont l'architecture est plus symétrique que celle d'IdFreeze. Les accès à des bancs mémoire distants sont plus coûteux que sur cette dernière. Ainsi, la politique `interleave` qui permet des accès en moyenne meilleurs car de temps en temps proches ne suffit pas sur les machines aux architectures irrégulières. De plus, IdRouille a plus de cache L3 par cœur, et les

éléments utilisés dans l'expérience précédente sont les briques de base de seulement 16o. En effet, lorsque la taille des éléments augmente, ici 2 lignes de cache par élément (128o), l'accélération parallèle semble limitée même sur cette machine avec la stratégie d'allocation `interleave` (figure 5.7).

Plutôt que de distribuer les pages mémoires de manière circulaire sur les différents nœuds, nous pouvons essayer de tirer partie de la localité spatiale dont notre structure permet de bénéficier en allouant la mémoire par blocs de données. Avec PaVo, les accès à la structure sont limités dans l'espace : chaque tâche travaille sur une plage de données contigüe et indépendante des autres. Nous utilisons une stratégie d'allocation *Par bloc* qui consiste à allouer la mémoire nécessaire par morceaux contigus sur les différents bancs mémoire utilisés. Ainsi les tâches ont plus de chance de travailler sur des données situées sur le même banc mémoire.

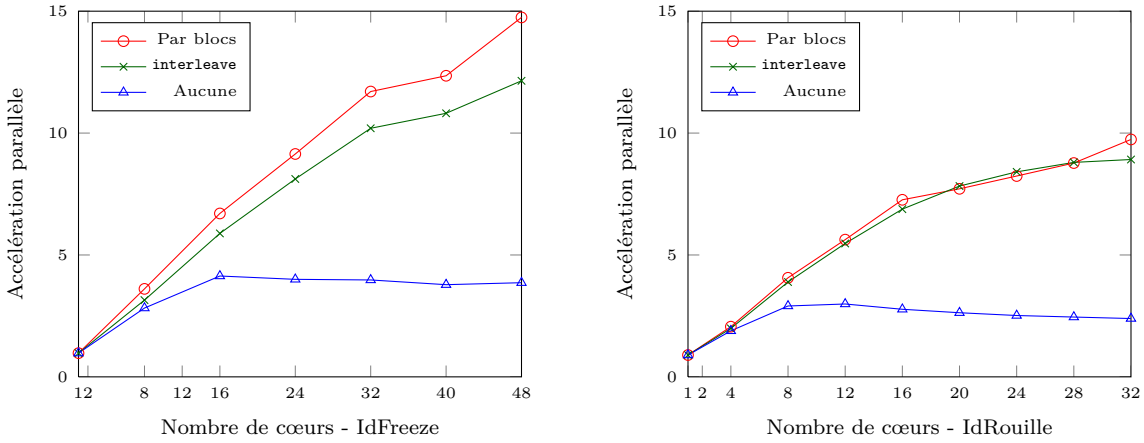


FIGURE 5.7 – Accélération de PaVo en fonction du nombre de cœurs utilisés pour différentes stratégies de placement en mémoire. Les éléments ont pour taille 2 lignes de cache, $K = 2\,000\,000$, $c = 10\%$.

Avec cette stratégie, nous observons sur la machine IdFreeze une amélioration de l'accélération parallèle par rapport à la stratégie Page cyclique (figure 5.7). Avec des structures de taille plus importante que précédemment et bien que seule la taille des éléments aient changé et ni le nombre de déplacements ni la taille de la séquence, l'accélération parallèle sur la machine IdRouille est limitée par la bande passante mémoire. La stratégie Par blocs est légèrement meilleure que la stratégie Page cyclique sur cette machine mais uniquement pour 16 et 32 cœurs.

5.3 Répartition du travail

Cette fois-ci, il ne s'agit pas d'un moyen de transport : nous utilisons le vol, mais le vol de tâches uniquement, une action socialement peu recommandable... et qui pourtant favorise l'utilisation des ressources.

5.3.1 Adaptation du vol

En effet, pour s'adapter aux irrégularités de l'algorithme, les tâches de l'étape TRI sont réparties entre les cœurs via un mécanisme de vol de travail. De nombreux supports exécutifs implantent ce mécanisme : OpenMP [Board, 2008], TBB [Reinders, 2007], Cilk [Blumofe et al., 1996] ainsi que XKaapi. Nous utilisons ce dernier car le très faible surcoût de création de ses tâches [Broquedis et al., 2012] permet de ne pas se préoccuper du choix du grain parallèle dans l'étape TRI.

Un petit bilan : nous avons réparti les données par bloc contigus sur les différents bancs mémoire des nœuds utilisés et nous utilisons un support exécutif de vol de tâches efficace. Nous avons montré que les accès à des bancs mémoire distants sont plus coûteux que les accès au banc associé au nœud. Pour améliorer tout ceci, nous avons testé une adaptation du vol : un thread qui a terminé tout son travail commence par chercher des tâches prêtes dans son environnement proche, c'est-à-dire au sein du nœud NUMA auquel il appartient. Cette stratégie de vol local peut être stricte ou non. Dans sa déclinaison stricte, un thread ne peut pas choisir une victime à voler hors de son nœud NUMA. Dans le cas contraire, non strict, nous autorisons un thread à voler en dehors de son nœud NUMA lorsqu'il a déjà effectué plusieurs tentatives de vols.

Il ne manque plus qu'une chose : prédistribuer les tâches sur les différents nœuds pour tenir compte de la répartition des données. Pour cela, nous utilisons les informations remontées via le support exécutif. Nous déterminons le nombre de nœuds utilisés par l'application. Muni de ses informations, le thread qui commence la descente récursive de l'étape TRI pousse des tâches spéciales sur les différents nœuds NUMA auxquelles elles sont destinées. Ces tâches permettent à un thread du nœud cible de poursuivre la descente dans l'arbre, c'est-à-dire dans l'algorithme avancé, la simple vérification des densités. Petit à petit, tous les nœuds reçoivent du travail. De même que précédemment, lorsque le niveau des régions est atteint, alors les fenêtres sont traitées classiquement par création de tâche à chaque nouvel appel récursif.

En utilisant cette adaptation du vol standard, nous évitons ou nous interdisons (dans la version stricte) les vols de tâches concernant des données distantes. Nous avons étudié le tri de $K = 2\,000\,000$ avec 10% de voyageurs (figure 5.8). Nous représentons le gain des différentes stratégie par rapport à la politique Page-cyclique utilisée seule. Comme nous l'avions vu précédemment (figure 5.7), dans certaines configurations, la distribution des données par bloc n'est pas bénéfique sur la machine IdRouille. Cependant, l'association de l'allocation des données par bloc et de la pré-distribution des tâches permet de gagner jusqu'à 28% sur IdFreeze et plus de 10% sur IdRouille.

La stratégie de vol stricte donne de meilleures performances sur les distributions uniformes testées ici car la quantité de travail est globalement bien répartie. Or éviter de voler un nœud distant permet d'éviter de rapatrier des données lointaines qui ne seront probablement pas réutilisées à l'itération suivante. Cependant, si la quantité de travail était très irrégulière, il faudrait permettre aux threads d'aller aider leurs camarades quitte à ce que cela leur coûte un peu plus cher. En pratique, dans les expérience de la partie évaluation (section 5.4), nous utilisons la stratégie non stricte.

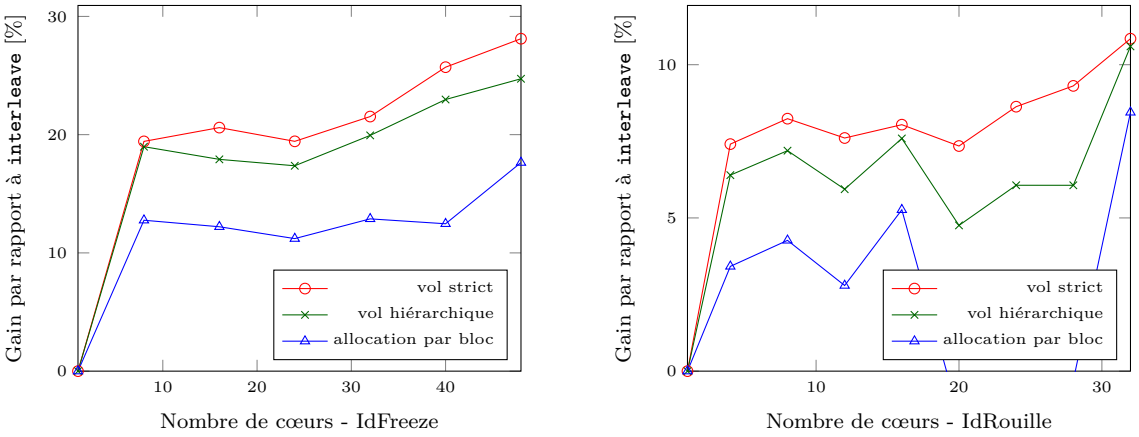


FIGURE 5.8 – Impact des différentes stratégies de vol sur la somme des temps d'exécution des étapes MAJ et TRI pour une séquence de $K = 2\,000\,000$ éléments avec 10% de voyageurs.

5.3.2 Boucles parallèles adaptatives

Nous venons de détailler notre stratégie en ce qui concerne l'étape TRI. Comment l'étape MAJ est-elle gérée ?

Le travail pouvant être non-homogène entre les différentes parties, nous utilisons une boucle adaptative qui a l'avantage d'éviter la création de tâches non utiles tout en permettant de répartir la quantité de travail entre tous. Dans ce cadre, nous avons étudié l'intérêt d'un ordonnanceur de boucle parallèle tenant compte de la hiérarchie mémoire. Cette stratégie de vol a été mise en place dans la librairie libKomp [Broquedis et al., 2012]. Les résultats ont été comparés aux autres ordonnanceurs de boucles fournis par OpenMP, travaux publiés dans *An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines* [Durand et al., 2013].

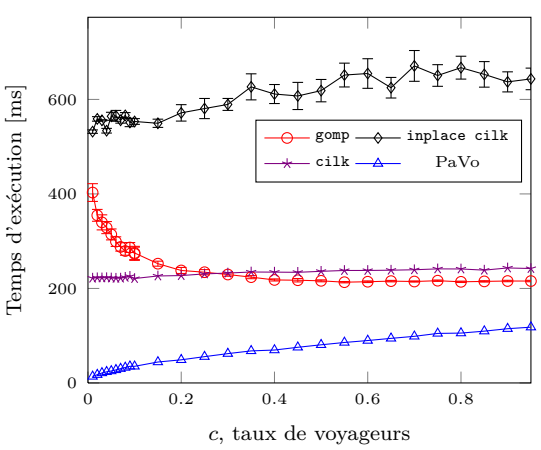
Dans ce chapitre, nous avons simplement utilisé la version contenue dans XKaapi sur laquelle se base libKomp.

5.4 Évaluation

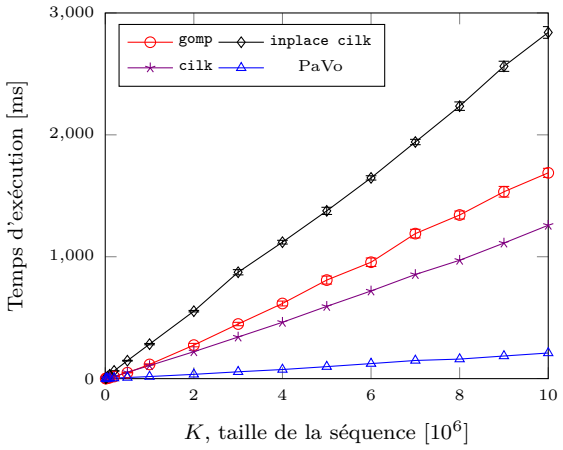
5.4.1 Comparaison

Nous avons testé les tris sur tableaux denses proposées par plusieurs librairies parallèles standard.

Le tri standard de la librairie STL, `std::sort`, possède une version parallèle utilisant OpenMP, via `libGomp`, identifiée par `gomp` par la suite. TBB propose `tbb::parallel_sort`, ici `tbb`. L'extension CilkPlus de la librairie Cilk fournit deux tris parallèles : `cilkpub::cilk_sort_in_place`, abrégé en `inplace_cilk` qui implante un Merge Sort parallèle et `cilkpub::cilk_sort`, repéré par `cilk`, basé sur un Sample Sort [Software, 2013].



(a) Influence du taux c de voyageurs pour une séquence de $k = 2\,000\,000$ éléments.



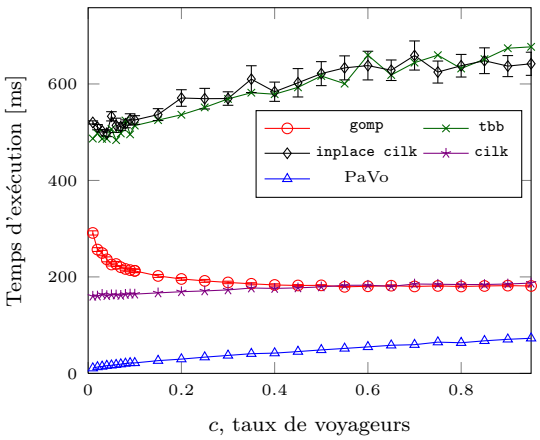
(b) Influence de la taille de la séquence pour $c = 0,1$ soit 10% de voyageurs.

FIGURE 5.9 – Temps d'exécution de différents algorithmes de tris parallèles pour $R = cK$ voyageurs sur la machine idfreeze avec $p = 24$ cœurs. La taille des éléments est de 2 lignes de cache.

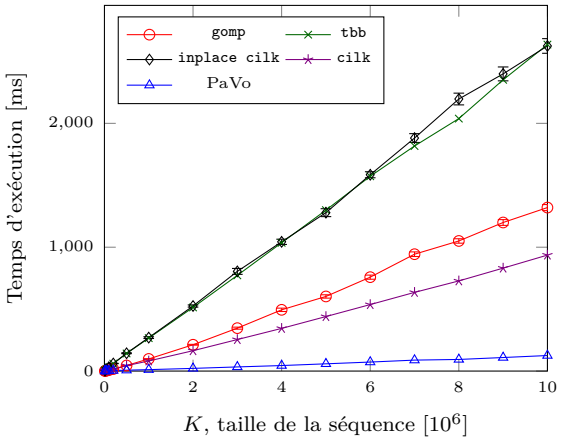
Nous utilisons ici encore le protocole expérimental décrit au chapitre précédent (section 4.4.3, page 85). Nous mesurons le temps d'exécution total (étapes MAJ et TRI). L'étape MAJ est parallélisée pour chacun des tests des algorithmes de tris parallèles en utilisant les boucles parallèles disponibles dans chacun de ces environnements.

Sur la machine IdFreeze, nous nous intéressons dans un premier temps au comportement des algorithmes en utilisant 24 cœurs répartis sur les 8 nœuds NUMA (figure 5.9). Nous représentons la performance des différents algorithmes pour une taille de séquence $K = 2\,000\,000$ fixée en faisant varier le taux de voyageurs (figure 5.9(a)). L'algorithme de tri le plus performant sur des tableaux denses dans cette configuration est `cilk`. Même lorsque le pourcentage de voyageurs atteint 95%, `cilk` est tout de même moins efficace que PaVo. Le gain de PaVo par rapport à `cilk` est compris entre 16 pour 1% de voyageurs et 2 pour 95%. À taux de voyageurs fixé, 10%, l'augmentation de la taille de la séquence accroît la différence entre les différents algorithmes et PaVo (figure 5.9(b)). Ce dernier s'exécute près de 6 fois plus rapidement que `cilk` pour $K = 10\,000\,000$ et de l'ordre de 4 fois pour $K = 10\,000$.

La comportement des différents algorithmes en utilisant tout le potentiel de la machine, 48 cœurs, est similaire (figure 5.10). `tbb` est très proche de la version `inplace_cilk`, ce qui est cohérent car il s'agit également d'une variante parallèle de l'algorithme Merge Sort. L'accélération de PaVo par rapport à `cilk` augmente un peu atteignant 2,6 pour $K = 2\,000\,000$ et 95% de voyages et plus de 7,4 pour $K = 10\,000\,000$ et 10%.



(a) Influence du taux c de voyageurs pour une séquence de $K = 2\,000\,000$ éléments.



(b) Influence de la taille de la séquence pour $c = 0,1$ soit 10% de voyageurs.

FIGURE 5.10 – Temps d'exécution de différents algorithmes de tris parallèles pour $R = cK$ voyageurs sur la machine idfreeze avec $p = 48$ cœurs. La taille des éléments est de 2 lignes de cache.

5.4.2 Des voyages de particules

Nous appliquons les voyages issus de l'exécution d'une simulation SPH avec 2 900 000 particules (section 4.5.4, page 95) effectuée avec Fluids (section 1.7.3).

Nous étudions avec cette distribution l'influence de la répartition des threads sur les différents nœuds NUMA (tableau 5.2). Nous utilisons l'allocation par bloc, la prédistribution des tâches et le vol local non strict. Les résultats montrent que pour les deux architectures testées, les configurations les meilleurs sont celles qui utilisent le plus de nœuds NUMA. Cela permet en effet d'augmenter l'efficacité des échanges mémoires puisqu'ils sont alors distribués sur plusieurs bancs mémoire.

L'accélération totale pour cette distribution atteint 19, 87 sur 48 cœurs sur IdFreeze, soit une efficacité de 41, 38%. Avec IdRouille, l'accélération maximum est 13, 25, soit une efficacité de 41, 4%. À titre de comparaison, l'efficacité avec 32 cœurs sur IdFreeze s'élève à 51%.

Terminons cette série d'évaluation sur la distribution de particules par une étude de l'impact de la taille des particules entre `cilk` et PaVo (figure 5.11). Entre 16o et 256o, la taille des éléments est multipliée par 16. Le temps d'exécution moyen de PaVo n'augmente que d'un facteur 2, 4 alors que l'augmentation de `cilk` est de l'ordre de 13. L'accélération de PaVo par rapport à `cilk` est de 11, 8 pour les éléments de 256o et de 7, 8 pour des éléments de taille 2 lignes de cache. Nous expliquons ces résultats par un nombre de copies mémoire beaucoup plus grandes dans le cas de `cilk`.

Nœuds x Cœurs	IdFreeze	IdRouille
8 cœurs		
2 x 4	103±3	71±2
4 x 2		60.4±1.2
8 x 1	99±2	
12 cœurs		
2 x 6	76±2	62±2
4 x 3	70±1.5	46±1
16 cœurs		
2 x 8		57.5±1.5
4 x 4	75.8±23	38±1
8 x 2	56±1.5	
24 cœurs		
4 x 6	64±25	
8 x 3	44±2	

TABLE 5.2 – Temps d'exécution total (ms) de l'étape MAJ pour la distribution extraite de déplacements de particules dans fluids ($K = 2\,900\,000$ et 10% de voyageurs maximum).

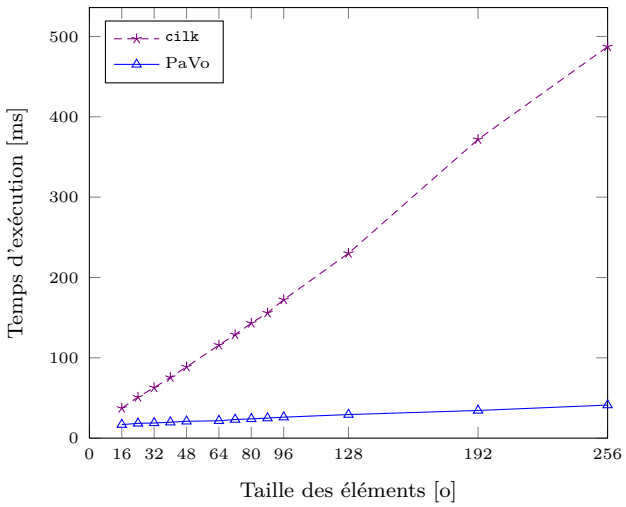


FIGURE 5.11 – Influence de la taille des éléments sur la machine IdFreeze avec 48 cœurs. Les $K = 2\,900\,000$ éléments sont soumis à la distribution extraite de Fluids.

5.5 Discussion

Nous avons proposé un algorithme parallèle pour traiter le tri de séquences presques triées. PaVo est basé sur une structure Cache-Oblivious qui maintient des trous entre les éléments valides pour faciliter l'insertion. L'idée est de limiter le nombre de déplacements mémoire à effectuer pour maintenir l'ordre dans une séquence.

Nous avons utilisé le mécanisme de vol de tâches pour dérouler l'étape de tri de notre algorithme et proposé une adaptation de la stratégie de vol pour tenir compte du placement des données. Nous avons à cette occasion testé le principe du vol local dans le cadre de boucles parallèles adaptatives.

L'algorithme parallèle PaVo a atteint son objectif principal : démontrer l'intérêt du trou. En effet, les gains par rapport aux tris parallèles sur des tableaux denses sont encore plus élevés que les gains séquentiels. Les gains obtenus sont très importants : de 6 à 11 par rapport au meilleur algorithme que nous avons pu tester (`sort` de CilkPlus).

L'algorithme s'adapte à des distributions non uniformes et même des éléments de grande tailles peuvent être maintenus en ordre efficacement. Le protocole expérimental décrit et expérimenté ici pourrait facilement être intégré dans un simulateur physique. La boucle parallèle adaptative de l'étape que nous avons appelée MAJ correspondra au parcours des éléments pour mettre à jour leurs positions en fonctions des forces calculées au préalable. Les éléments dont le changement de coordonnées induit un changement de clé de cellule seront considérés comme voyageurs. À la fin de la mise à jour des positions, l'étape de TRI pourra être rapidement exécutée pour réintégrer ces éléments en ordre.

IV

Croisée des chemins

Synthèse 6

Les travaux présentés dans ce document se sont scindés en deux parties très différentes : dans la première nous nous sommes frottés aux aspérités de la programmation sur GPU. Nous avons proposé la première implantation complète de simulation basée sur des éléments discrets. Nous avons relevé les difficultés inhérentes à l'irrégularité du problème. Nous avons néanmoins présenté de bonnes accélérations tout en notant l'important effort à fournir.

Puis nous nous sommes intéressés à un algorithme de tri de séquences presque triées utilisant une structure à trous pour réduire les déplacements en mémoire et ainsi les dépendances de données. PaVo est adaptatif car le nombre d'opérations réalisées dépend du travail à effectuer. Sa complexité est $O(\log^2 n)$ dans le cas moyen et l'espace mémoire nécessaire est borné par $O(n)$. Finalement, nous avons proposé un algorithme parallèle, détaillé une politique de placement des données en mémoire et étudié une adaptation de l'ordonnanceur de répartition du travail tenant compte de la hiérarchie mémoire de la machine utilisée. En utilisant une machine parallèle NUMA de 48 cœurs, nous avons relevé une efficacité parallèle de PaVo de l'ordre de 40% pour des distributions de voyages représentant des mouvements de particules. Par rapport à l'algorithme de tri parallèle sur tableau dense le plus performant, parmi ceux que nous avons testé, le gain de performance de PaVo est de presque 8.

L'une sur GPU, l'autre sur CPU, l'une liée à une architecture particulière, l'autre destinée à tout un spectre de machines... sous ces différents points de vue, nous nous sommes intéressés à la parallélisation de problèmes irréguliers générant d'importants échanges mémoire. Dans ces deux facettes de la problématique, nous avons détaillé des solutions pratiques et expliqué les concepts sous-jacent.

À la croisée des chemins de tous ces travaux, il serait intéressant de tester l'apport d'un PMA pour améliorer la performance des simulations basées sur des EDs. Nous avons établi la dégradation des performances au cours de la simulation avec l'étalement des voisins. Nous avons montré que l'algorithme PaVo permet de maintenir efficacement des éléments triés en mémoire. En utilisant une représentation de l'espace de type courbe en Z ou courbe de Hilbert, il est possible d'augmenter la localité spatiale des données en mémoire. Combiner ces différentes approches devrait permettre d'améliorer simplement la performance de ces simulations.

L'algorithme que nous avons proposé pourra être utilisé dans de nombreuses situations. Par exemple, la méthode de détection de collision *Sweep and Prune* ordonne les projections des volumes englobant les différents objets de la scène simulée selon les 3 axes [Cohen et al., 1995]. Cela permet d'éliminer rapidement les paires ne pouvant pas être en contact et ainsi limiter le nombre de tests. Le coût du tri augmente avec le

nombre d'objets des scènes, or les mouvements relatifs sont faibles d'un pas de temps à l'autre, faisant de cet algorithme un bon candidat pour tirer avantage de PaVo et de son PMA.

Toujours pour la détection de collision, l'efficacité de la méthode utilisant une division de l'espace en grille régulière repose sur les structures de données utilisées. Si les cellules de la grille sont toutes stockées en mémoire, l'espace nécessaire est dépendant du domaine de simulation. Lorsque les particules n'occupent qu'une partie de l'espace de simulation, des méthodes comme le *Compact Hashing* ont été proposées pour optimiser l'espace mémoire nécessaire tout en limitant le surcoût de la gestion des structures supplémentaires [Ihmsen et al., 2011]. Nous suggérons d'utiliser un PMA pour stocker les cellules pleines de la grille de détection avec des liens vers le PMA permettant de conserver les particules triées. La mise à jour de la grille de détection sera rapide et les données seront conservées avec de bonnes propriétés spatiales. Un peu de travail de programmation est encore nécessaire pour passer de cette bonne idée à la réalisation efficace, mais cela pourrait bien rencontrer un franc succès !

Nous nous sommes également intéressés au développement d'une version GPU de l'algorithme PaVo. Dans le cadre d'un stage de Master 2 Recherche au printemps 2013, Alexis Martin a travaillé au sein de l'équipe MOAIS sur la mise en œuvre de la structure sur ce support. Encore une fois, la programmation GPU s'est révélée complexe et les résultats ne sont pas suffisamment complets pour être présentés ici.

Depuis plusieurs années, une nouvelle architecture était annoncée par le constructeur Intel. Le fameux projet *Larrabee* sensé être capable de concurrencer le GPU, s'est finalement converti en *Intel MIC*, pour *Intel Many Integrated Core architecture* puis a pris le doux non commercial de *Xéon Phi*. PaVo et son PMA seront facilement portés vers cette architecture grâce à l'aspect classique de sa programmation. Les performances devraient être au rendez-vous, l'architecture offrant un grand nombre d'unités indépendantes : actuellement 60 bénéficiant de capacités d'hyper-threading pouvant tirer partie du parallélisme exhibé par PaVo.

Finalement, PaVo permet de supporter la dynamicité avec un contrôle fin et structuré de l'usage de la mémoire à l'opposé des techniques à base de listes chaînées. La consommation d'énergie est un enjeu crucial des futures architectures, et les mouvements de données l'un des postes le plus gourmand dans ce registre. PaVo optimise justement ces déplacements. À ce titre, cet algorithme ainsi que ses extensions vers d'autres structures de données accélératrices parallèles ont de beaux jours devant eux !

Bibliographie

Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9) :1116–1127, September 1988. 3.1.2

J. Alder and T. E. Wainwright. Phase transitions for a hard sphere system. *Journal of Chemical Physics*, 27 :1208–1209, 1957. 1.4

Jérémie Allard, Hadrien Courtecuisse, and François Faure. Implicit fem and fluid coupling on gpu for interactive multiphysics simulation. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, Vancouver, Canada, August 2011. ACM. 1.6

Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *J. Comput. Phys.*, 227 :5342–5359, May 2008. 2.3.1.1, 2.5

Lars Arge, Mikael Knudsen, and Kirsten Larsen. A general lower bound on the i/o-complexity of comparison-based algorithms. In *Proceedings of the Third Workshop on Algorithms and Data Structures*, WADS '93, pages 83–94, London, UK, UK, 1993. Springer-Verlag. 3.1.2

Josh Barnes and Piet Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(6096) :446–449, 1986. 1.4.1.2

Serkan Bayraktar, Ugur Güdükbay, and Bülent Özgüç. Gpu-based neighbor-search algorithm for particle simulations. *J. Graphics, GPU, & Game Tools*, 14(1) :31–42, 2009. 2.3.1.3

M.A. Bender, E.D. Demaine, and M. Farach-Colton. Cache-oblivious b-trees. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 399 –409, 2000. 3.3.1, 3.3.2, 3.3.4, 4.3.1

Michael A. Bender and Haodong Hu. An adaptive packed-memory array. *ACM Trans. Database Syst.*, 32, November 2007. 3.3.4, 3.3.5, 3.3.5, 4.2.2, 4.2.4

Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is o(n log n). *CoRR*, cs.DS/0407003, 2004. 3.2, 3.2

Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. *SIAM J. Comput.*, 35(2) :341–358, 2005. 3.3.1, 3.3.4

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk : An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1) :55–69, August 25 1996. 5.3.1

OpenMP Architecture Review Board. Openmp api, version 3.0, May 2008. 5.3.1

Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz. Cache-aware and cache-oblivious adaptive sorting. In *Proceedings of the 32nd international conference on Automata, Languages and Programming*, ICALP'05, pages 576–588, Berlin, Heidelberg, 2005. Springer-Verlag. 3.1.2, 3.5

François Broquedis, Thierry Gautier, and Vincent Danjean. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 102–115, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-30960-1. 5.3.1, 5.3.2

W. Michael Brown, Peng Wang, Steven J. Plimpton, and Arnold N. Tharrington. Implementing molecular dynamics on hybrid high performance computers – short range forces. *Computer Physics Communications*, 182(4) :898 – 911, 2011. 2.3.1.1

Arthur R. Butz. Convergence with hilbert's space filling curve. *Journal of Computer and System Sciences*, 3(2) :128 – 146, 1969. 2.3.1.1

Svante Carlsson and Jingsen Chen. On partitions and presortedness of sequences. *Acta Informatica*, 29 :267–280, 1992. 3.1.1

Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. I-collide : an interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, I3D '95, pages 189–ff., New York, NY, USA, 1995. ACM. 6

Curtis R. Cook and Do Jin Kim. Best sorting algorithm for nearly sorted lists. *Commun. ACM*, 23(11) :620–624, November 1980. 3.1.1, 3.4.2.3

R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen differenzengleichungen der mathematischen physik. *Mathematische Annalen*, 100(1) :32–74, 1928. 1.7.1, 2.1.1

P. A. Cundall and O. D. L. Strack. A discrete numerical model for granular assemblies. *Geotechnique*, 29(1) :47–65, 1979. 1.5

H.K. Dai and H.C. Su. On the locality properties of space-filling curves. In Toshihide Ibaraki, Naoki Katoh, and Hirotaka Ono, editors, *Algorithms and Computation*, volume 2906 of *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin Heidelberg, 2003. 2.3.1.1

Erik D. Demaine. Cache-oblivious algorithms and data structures. In *IN LECTURE NOTES FROM THE EEF SUMMER SCHOOL ON MASSIVE DATA SETS*, 2002. 3.1.2

Edsger W. Dijkstra. Smoothsort, an alternative for sorting in situ. *Science of Computer Programming*, 1(3) :223 – 233, 1982. 3.1.1

R. Geoff Dromey. Exploiting partial order with quicksort. *Software : Practice and Experience*, 14(6) :509–518, 1984. 3.1.1

Marie Durand, François Faure, and Bruno Raffin. A Packed Memory Array to Keep Moving Particles Sorted. In *9th Workshop on Virtual Reality Interaction and Physical Simulation (VRIPHYS)*, December 2012a. (document), 4.4.5, 4.5

Marie Durand, Philippe Marin, François Faure, and Bruno Raffin. DEM-based simulation of concrete structures on GPU. *European Journal of Environmental and Civil Engineering*, pages 1–13, August 2012b. (document)

Marie Durand, François Broquedis, Thierry Gautier, and Bruno Raffin. An efficient openmp loop scheduler for irregular applications on large-scale numa machines. In *IWOMP*, pages 141–155, 2013. (document), 5.3.2

Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Comput. Surv.*, 24(4) :441–476, December 1992. 3.1.1

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society. 3.1.2

Prashant Goswami, Philipp Schlegel, Barbara Solenthaler, and Renato Pajarola. Interactive sph simulation and rendering on the gpu. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 55–64, Aire-la-Ville, Switzerland, 2010. Eurographics Association. 2.3.1.3

Simon Green. Particle simulation using cuda. CUDA SDK, May 2010. 2.1.2, 2.3.1.3, 2.3.7

L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2) :325–348, December 1987. 1.4.1.2

Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed particle hydrodynamics on gpus. *Structure*, pages 1–8, 2007. 2.3.1.3

Francis H. Harlow. Hydrodynamic problems involving large fluid distortions. *Journal of the ACM*, 4(2) :137–142, april 1957. 1.2.1

S. Hentz, L. Daudeville, and F. V. Donzé. Identification and validation of a discrete element model for concrete. *Journal of Engineering Mechanics*, 130(6) :709–719, 2004. 1.5

Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, and Jérémie Allard. Multi-gpu and multi-cpu parallelization for interactive physics simulations. In *Euro-Par (2)*, pages 235–246, 2010. 1.7.2, 2.5

Rama C. Hoetzlein. Fluids v.3 - a large-scale, open source fluid simulator., december 2012. URL `http://fluids3.com`. 1.2.2

Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. A parallel sph implementation on multi-core cpus. *Computer Graphics Forum*, 30(1) :99–112, 2011. 4.5.1, 4.5.1, 6

Alon Itai, Alan Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin / Heidelberg, 1981. 3.3.1

Manaschai Kunaseth, Ken-ichi Nomura, Hikmet Dursun, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. Memory-access optimization of parallel molecular dynamics simulation via dynamic data reordering. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pages 781–792, Berlin, Heidelberg, 2012. Springer-Verlag. 4.5.1

L3SR. Laboratoire sols solides structures risques, CNRS, Grenoble. URL `http://3sr.hmg.inpg.fr/3sr/`. 1

Anthony LaMarca and Richard E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, SODA '97, pages 370–379, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics. 3.1.2

Christos Levcopoulos and Ola Petersson. Splitsort—an adaptive sorting algorithm. *Information Processing Letters*, 39(4) :205 – 211, 1991. 3.1.1

Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Trans. Comput.*, 34(4) :318–325, April 1985. 3.1.1

Kurt Mehlhorn. Sorting presorted files. In K. Weihrauch, editor, *Theoretical Computer Science 4th GI Conference*, volume 67 of *Lecture Notes in Computer Science*, pages 199–212. Springer Berlin Heidelberg, 1979. 3.1.1

J. J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30 :543–574, 1992. 1.2.2

B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1) :124–141, 2001. 2.3.1.1

David R. Musser. Introspective sorting and selection algorithms. *Softw. Pract. Exper.*, 27(8) :983–993, August 1997. 4.4.6

NVIDIA. NVIDIA CUDA c programming guide version 4.0. CUDA Documentation, June 2011. URL `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`. 2.3.4

Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2) :153 – 179, 1995. 3.1.1, 3.4.2.3

D.C. Rapaport. Enhanced molecular dynamics performance with a programmable graphics processor. *Computer Physics Communications*, 182(4) :926 – 934, 2011. 2.3.1.1

James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808. 5.3.1

V Rokhlin. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics*, 60(2) :187 – 207, 1985. 1.4.1.2

J. Rousseau, P. Marin, L. Daudeville, and S. Potapov. Couplage Eléments Finis/Eléments Discrets : application à la simulation d'impact localisé sur un ouvrage en béton armé. In *9ème Colloque National en calcul des Structures.*, Giens, France, May 2009. HAL. 1.7.1

Jessica Rousseau. *Modélisation numérique du comportement dynamique de structures sous impact sévère avec un couplage éléments discrets / éléments finis.* These, Université Joseph-Fourier - Grenoble I, September 2009. 2, 2.1.1

N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –10. IEEE Computer Society, may 2009. 2.3.7

Yusuke Shigeto and Mikio Sakai. Parallel computing of discrete element method on multi-core processors. *Particuology*, 9(4) :398–405, 2011. 2.3.1.4, 2.3.5

Intel Software. Parallel sorts for cilk plus, March 2013. 5.4.1

SPHysics. Sphysics, 2011. URL `https://wiki.manchester.ac.uk/sphysics/index.php/SPHYSICS_Home_Page`. 1.2.2

Alfeus Sunarso, Tomohiro Tsuji, and Shigeomi Chono. Gpu-accelerated molecular dynamics simulation for study of liquid crystalline flows. *J. Comput. Physics*, 229 (15) :5486–5497, 2010. 2.3.1.1

Jacobus Antoon van Meel, Axel Arnold, Daan Frenkel, Simon F. Portegies Zwart, and Robert G. Belleman. Harvesting graphics power for MD simulations. *Molecular Simulation*, 34(03) :259–266, May 2008. 2.3.1.1

L. Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159 :98–103, July 1967. 1.6.1

Li Xiao, Xiaodong Zhang, and Stefan A. Kubricht. Improving memory performance of sorting algorithms. *J. Exp. Algorithmics*, 5, December 2000. 3.1.2, 3.5

Fengquan Zhang, Lei Hu, Jiawen Wu, and Xukun Shen. A sph-based method for interactive fluids simulation on the multi-gpu. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '11, pages 423–426, New York, NY, USA, 2011. ACM. 2.3.1.3

Xiangkun Zhao, Fengxia Li, and Shouyi Zhan. A new gpu-based neighbor search algorithm for fluid simulations. In Zhengbing Hu and Ping Ma, editors, *Second International Workshop on Database Technology and Applications, DBTA 2010, Wuhan, Hubei, China, November 27-28, 2010, Proceedings*, pages 1–4. IEEE, 2010. 2.3.1.3

Les joueurs exigeants acquièrent dès que possible une carte graphique capable de satisfaire leur soif d'immersion dans des jeux dont la précision, le réalisme et l'interactivité redoublent d'intensité au fil du temps. Depuis l'avènement des cartes graphiques dédiées au calcul généraliste, ils n'en sont plus les seuls clients. Dans un premier temps, nous analysons l'apport de ces architectures parallèles spécifiques pour des simulations physiques à grande échelle.

Cette étude nous permet de mettre en avant un goulot d'étranglement en particulier limitant la performance des simulations. Partons d'un cas typique : les fissures d'une structure complexe de type barrage en béton armé peuvent être modélisées par un ensemble de particules. La cohésion de la matière ainsi simulée est assurée par les interactions entre elles. Chaque particule est représentée en mémoire par un ensemble de paramètres physiques à consulter systématiquement pour tout calcul de forces entre deux particules. Ainsi, pour que les calculs soient rapides, les données de particules proches dans l'espace doivent être proches en mémoire. Dans le cas contraire, le nombre de défauts de cache augmente et la limite de bande passante de la mémoire peut être atteinte, particulièrement en parallèle, bornant les performances. L'enjeu est de maintenir l'organisation des données en mémoire tout au long de la simulation malgré les mouvements des particules. Les algorithmes de tri standard ne sont pas adaptés car ils trient systématiquement tous les éléments. De plus, ils travaillent sur des structures denses ce qui implique de nombreux déplacements de données en mémoire.

Nous proposons PaVo, un algorithme de tri dit adaptatif, c'est-à-dire qu'il sait tirer parti de l'ordre pré-existant dans une séquence. De plus, PaVo maintient des trous dans la structure, répartis de manière à réduire le nombre de déplacements mémoires nécessaires. Nous présentons une généreuse étude expérimentale et comparons les résultats obtenus à plusieurs tris renommés. La diminution des accès à la mémoire a encore plus d'importance pour des simulations à grande échelles sur des architectures parallèles. Nous détaillons une version parallèle de PaVo et évaluons son intérêt. Pour tenir compte de l'irrégularité des applications, la charge de travail est équilibrée dynamiquement par vol de travail. Nous proposons de distribuer automatiquement les données en mémoire de manière à profiter des architectures hiérarchiques. Les tâches sont pré-assignées aux cœurs pour utiliser cette distribution et nous adaptons le moteur de vol pour favoriser des vols de tâches concernant des données proches en mémoire.

Mots-clés : Simulation physique, Calcul parallèle, Architectures NUMA, Algorithmes de tri adaptatif, Structure de données à trou.

Gamers are used to throw onto the latest graphics cards to play immersive games which precision, realism and interactivity keep increasing over time. With general-propose processing on graphics processing units, scientists now participate in graphics card use too. First, we examine these architectures interest for large-scale physics simulations.

Drawing on this experience, we highlight in particular a bottleneck in simulations performance. Let us consider a typical situation: cracks in complex reinforced concrete structures such as dams are modelised by many particles. Interactions between particles simulate the matter cohesion. In computer memory, each particle is represented by a set of physical parameters used for every force calculations between two particles. Then, to speed up computations, data from particles close in space should be close in memory. Otherwise, the number of cache misses raises up and memory bandwidth may be reached, specially in parallel environments, limiting global performance. The challenge is to maintain data organization during the simulation despite particle movements. Classical sorting algorithms do not suit such situations because they consistently sort all the elements. Besides, they work upon dense structures leading to a lot of memory transfers.

We propose PaVo, an adaptive sort which means it benefits from sequence presortedness. Moreover, to reduce the number of necessary memory transfers, PaVo spreads some gaps inside the data structure. We present a large experimental study and confront results to reputed sorting algorithms. Reducing memory requests is again more important for large scale simulations with parallel architectures. We detail a parallel version of PaVo and evaluate its interest. To deal with application irregularities, we do load balancing with work-stealing. We take advantage of hierarchical architectures by automatically distributing data in memory. Thus, tasks are pre-assigned to cores with respect to this organization and we adapt the scheduler to favor steals of tasks working on data close in memory.

Keywords: Physics Simulation, High Performance Computing, NUMA Architectures, Adaptive Sorting Algorithm, Data Structure with Gaps.