N° ordre : 5026

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX 1

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

Par Cyril BORDAGE

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : informatique

Ordonnancement dynamique, adapté aux architectures hétérogènes, de la méthode multipôle pour les équations de Maxwell, en électromagnétisme

Directeurs de thèse : M. GOUDIN David et M. NAMYST Raymond

Soutenue le : 20 décembre 2013

Devant la commission d'examen formée de :

M. AMESTOY Patrick	Professeur des Universités	ENSEEIHT	Rapporteur
M. COLLINO Francis	Ingénieur de Recherche		
M. DARRIGRAND Éric	Maître de Conférences	Université de Rennes 1	
M. GOUDIN David	Ingénieur de Recherche	CEA/CESTA	Encadrant de thèse
M. NAMYST Raymond	Professeur des Universités	Labri	Encadrant de thèse
M. PELLEGRINI François	Professeur des Universités	Université de Bordeaux	Président du jury
M. PEREZ Christian	Directeur de Recherche	LIP	Rapporteur

Remerciements

Je vais procéder ici aux traditionnels remerciements. Non pas parce qu'ils sont traditionnels, mais parce qu'il est important de montrer qu'une thèse ça ne se fait pas tout seul, mais en étant entouré. C'est comme ça qu'on est motivé, qu'on garde la bonne humeur, qu'on est ~~moins~~ davantage productif. Je tiens à préciser que l'ordre de citation (alphabétique) n'a aucune corrélation avec l'ordre d'importance.

Tout d'abord je tiens à remercier mes encadrants « officiels » (*dixit* David) : David et Raymond. Leur grande force a été de subir mon humour (si on peut le qualifier ainsi). David a réussi à me transmettre sa bonne humeur (ce qui est d'autant plus respectable si on prend en compte ses origines bergeracoises). Raymond n'est également pas le dernier pour les plaisanteries. Ensuite, moins officiels mais non moins importants : Agnès et Francis. Grâce à eux, j'ai eu l'illusion de maitriser la composante numérique de ma thèse. Merci pour leur investissement. Je remercie également Éric, Christian, Patrick et Xavier pour leurs corrections.

Pendant ma thèse, j'ai eu la chance d'avoir plein de collègues sympathiques au CEA puis à l'INRIA. Ces remerciements se veulent synthétiques. Pour des détails, ou des réclamations (erreurs ou oublis), n'hésitez pas à me contacter.

Tout d'abord, globalement, je tiens à remercier mon service d'accueil au CEA, qui m'a fourni un cadre agréable et chaleureux (je ne parle pas de la température l'été) : Anne-Pascale, qui a su faire preuve d'une grande patience malgré ma gentillesse ; Bernard, qui m'a prouvé chaque matin sa poigne ; Céline, pour sa bonne humeur et ses « blagues » ; David H., pour ses conversations intéressantes ; Florian, qui aimait tant prendre le car ; François C. ; François G., pour nos débats si passionnés en voiture ; François V., pour notre travail d'équipe lors de l'âne rouge (malgré l'alligator de 15m) ; Fred M. ; Fred O. ; Gaby, pour ses jambes poilues et ses carottes râpées ; Gérard, qui adore la pizza 3 fromages surtout sans fromage ; Guilhem, qui a su me faire découvrir le Gaillac ; Isabelle, avec qui il est toujours agréable de parler ; Jean, mon ancien co-bureau qui est de très bonne compagnie et de bon conseil (musique, films) « Malgré ton âge avancé et ta passion pour la soupe, je ne pense pas que tu sois un vieux croûton. Que ton émeoaerosagophilie te mène loin. » ; Jean-Jacques ; Jean-Marc, que j'ai dû laisser gagner au Judo ; Johanne, toujours souriante et sympathique ; Julien, qui doit dormir actuellement ; Laurent, pour son histoire avec D. Voynet ; Manu, qui a su nous apporter les bonnes boissons ; Marc, pour ses résumés de la matinale de Skyrock ; Matthieu C., qui m'a donné de bonnes astuces ; Matthieu L., qui adore nos blagues fines et délicates, et innovantes (notamment sur les gens du Nord) ; Matthieu J., que j'ai dû laisser gagner au badminton ; Michel, toujours de bonne humeur et capable de quasiment supporter mes blagues ; Mickaël, qui rentrait parfaitement entre les deux armoires ; Muriel, qui ne doit plus avoir peur que je rate le bus ; Nicolas G., qui a découvert (testé ?) un nouvel usage des carambars ; Nicolas H., qui voit que j'ai enfin fini ma thèse ; Olivier, avec qui j'ai eu des conversations intéressantes mais qui donnaient, aux autres, mal à la tête ; Onesim, qui me remplace si fidèlement, « Veille sur Jean. » ;

Pascal A., de très bon conseil en cinéma ; Pascal J., qui m'a permis de penser au-delà de la thèse ; Patrick, pour ses discussions politiques ; Pierre B., qui ne doit pas savoir qui je suis ; Pierre M., ami des zombies ; PH, pour son documentaire (que j'attends encore) sur l'évolution de ma thèse ; Thibault, qui a su obtenir toutes les faveurs de Virginie ; Sébastien C., qui n'est pas parvenu à briller au Basket ; Sébastien D., qui a su concilier thèse et soirées astronomie (bien sûr) ; Tahn-Ha, enjouée, sûrement grâce à son super gâteau vert ; Titus, le berger allemand si fidèle ; Xavier, toujours là pour faire un discours ; Yohan, pour ses salutations si courtoises ; Zaza, pour ses cours et tortures ; les personnes avec qui je partageais un moment de détente à la cafétéria, et qui ne mangeaient pas de chocolat ; les différents chauffeurs de bus qui ont su garder leur patience malgré l'heure tardive ; les cuisiniers de la cantine.

Pour ma dernière année, à l'INRIA, j'ai également eu le plaisir de rencontrer les personnes suivantes : Alexandre ; Andra, pour ses grilles de mots fléchés ; Antoine, qui utilisent les meilleurs outils (les mêmes que moi) ; Arista, à l'origine d'une choré fantastique ; Aude, qui a réussi à ne pas craquer à nos blagues ; Bertrand, qui aime la délicatesse et la grande poésie ; Brice, qui est plus présent comme collègue que comme prof ; Cédric, qui a su être un bon compagnon de galère ; Christopher, qui aime gagner des points Godwin ; Cyril, qui doit s'acheter une imprimante ; Denis ; Emmanuel, avec qui j'ai eu le plaisir de faire des pauses jeux ; François R., qui doit être en pause (ou chez lui) ; François T., qui a vu tout Internet ; Géraldine, pour ses conversations intéressantes ; Guillaume, qui m'a fait découvrir Sharknado et compagnie ; Manu, pour ses discours passionnés et sa poésie ; Marc, toujours prêt à aider ; Marie-Chistine, toujours très sympathique ; Matthieu, pour les pauses café ; Nathalie, qui m'a aidé ; Olivier, pour sa réactivité aux « bip bip café » ; l'Open Space en général, qui a contribué grandement à ma bonne humeur ; Pierre-André, qui fait du Pierre-André ; Paulo, fervent défenseur de Latex et Vim ; Samuel, qui corrige les bogues avant que j'ai le temps de retourner à mon bureau ; Sébastien, « Que la crasse-mitraillette soit avec toi ! » ; Siutumoro, à demain ; Sylvain, qui trouve que tout est bien fait ; les macarons et cannelés ; le Samsung Home Clean.

Je tiens aussi à remercier mes parents et ma sœur Céline, qui m'ont fait le plaisir d'assister à ma soutenance. Merci également à Florence, et à Richard qui a su m'apporter son expertise anglaise, Pour terminer, je me dois de remercier Sarah, la femme de ma vie (sauf si elle me quitte), qui a su m'accompagner et me supporter durant ce périple. Elle m'a permis de garder le moral pendant les moments difficiles, grâce au bonheur qu'elle m'a apporté.

Naturellement, je remercie également tout ceux que j'ai oubliés. Sachez qu'actuellement, je pense à vous !

Non-remerciements Je pense qu'il est également important de pointer du doigt les différents obstacles au bon déroulement de ma thèse, même si j'ai réussi à les contourner avec intelligence et sagesse. Tout d'abord, la plus grande épreuve pour moi fut de survivre à l'instant 10h15 : les mauvaises odeurs de Jean (les plus gentils prétexteront la soupe). La deuxième, qui fut présente à la fois au CEA et à l'INRIA, a été principalement orchestrée par Gaby et Paulo : le thé à la menthe. Pour le reste, je citerai en vrac : David qui pensait être Mike Brant, les liens sur IRC, les gens du CEA qui mangeaient du chocolat avec leur café, le monsieur qui met un demi-litre de parfum qui ne sent pas bon, le monsieur qui ramasse les poubelles et qui s'amuse avec feu notre balle, les « Ne pas tenir compte de ce message », Guilhem qui a su me faire découvrir certains Gaillac, les architectes du bâtiment de l'INRIA, la climatisation au CEA, Olivier qui a arrêté inexplicablement de manger avec nous, la rapidité de Paulo à table, l'attention de François T. quand on lui parle ou qu'il attend une réponse. Si j'ai oublié des détails, estimez-vous heureux !

Table des matières

Introduction

Le cadre de ce travail repose sur le besoin de connaître la réponse d'un objet soumis à une onde électromagnétique. Pour simuler le comportement électromagnétique d'objets 3D dans le domaine fréquentiel, nous devons résoudre les équations de Maxwell. Pour cela, nous utilisons des méthodes numériques standards basées sur une approximation par éléments finis d'équations intégrales de surface comme les formulations EFIE ou CFIE. Ces formulations conduisent à un système linéaire avec une matrice pleine complexe non hermitienne. Pour résoudre ce système, nous pouvons utiliser une méthode directe qui consistera à inverser la matrice du système. Les avantages de cette méthode sont d'une part la précision et d'autre part que l'inversion de la matrice est valable pour tous les seconds membres que nous aurons à traiter, Cependant, cette opération est très coûteuse : sa complexité en temps est en $\mathrm{O}(N^3)$, N étant le nombre d'inconnues. L'inversion peut se faire par différentes méthodes de factorisation telles que `LU` ou encore $\mathtt{LDL}^{\mathtt{T}}$. L'autre approche est l'utilisation d'une méthode itérative qui va permettre de converger vers la solution sans procéder à l'inversion de la matrice. Nous pouvons par exemple citer GMRES ou le gradient conjugué. Ces méthodes ont pour point commun d'effectuer un produit matrice-vecteur pour chaque itération jusqu'à la convergence finale. Elles présentent le grand avantage d'avoir une complexité en $\mathrm{O}(N^2)$. Cette complexité bien que réduite par rapport aux méthodes directes, reste toutefois encore bien trop élevée si nous voulons traiter des problèmes comportant des millions d'inconnues.

La complexité des méthodes itératives provient des produits matrice-vecteur effectués à chaque itération. Par conséquent, si nous voulons réduire le coût des solveurs itératifs, il est nécessaire d'accélérer les produits matrice-vecteur. C'est dans cette tâche qu'intervient la méthode multipôle. Celle-ci est capable d'accélérer le type de produits que nous avons à réaliser. Elle a été développée et adaptée à l'électromagnétisme dans les années 90.

Elle permet de diminuer la complexité d'un produit en $\mathrm{O}(N^2)$ à $\mathrm{O}(N \log(N))$. C'est cette méthode qui sera au cœur de notre étude. Comme notre but final concerne l'électromagnétique, nous ne traiterons pas la méthode multipôle dans un cadre général mais pour des noyaux oscillants de la forme $\frac{e^{\mathrm{i}kr}}{r}$. On retrouve ce noyau dans d'autres domaines tels que l'acoustique. Une autre application courante de la méthode multipôle est l'équation de Laplace, mais son noyau est de la forme $\frac{1}{r}$ et l'algorithme en est modifié. Nous ne nous intéresserons pas aux noyaux de cette forme pour notre étude.

Malgré les efforts sur la complexité de calculs, il reste difficile de traiter des objets comportant des millions de mailles sur un seul processeur. Pour y parvenir, il est indispensable d'avoir un algorithme parallèle pouvant utiliser simultanément plusieurs processeurs. Le stockage aussi peut être assez conséquent, et nécessiter beaucoup de mémoire. Quand nous augmentons la fréquence de l'onde incidente, il est nécessaire d'augmenter le nombre de mailles pour des problèmes de précision. Ainsi, le nombre de mailles et donc d'inconnues croît comme le carré de la fréquence. Si nous voulons alors résoudre notre problème sur un objet volumineux tel un avion à une fréquence

élevée (1 GHz), l'utilisation d'un supercalculateur est inévitable. Malheureusement, l'algorithme de la méthode multipôle ne se prête pas immédiatement à la parallélisation, les calculs ayant de nombreuses dépendances. Cela peut être problématique en mémoire distribuée du point de vue des performances. De plus, les architectures des supercalculateurs modernes sont de plus en plus complexes. Aux centaines de milliers de processeurs, il faut maintenant ajouter des accélérateurs, comme par exemple des GPU (processeurs de cartes graphiques). Avec ces nouvelles architectures, l'équilibre de charge devient plus difficile et les calculs doivent être adaptés suivant la machine cible.

Cadre de la thèse et contribution

Notre travail est l'étude de la parallélisation de la méthode multipôle pour profiter au maximum des capacités des machines de calcul actuelles comportant de la mémoire partagée au sein d'un nœud, de la mémoire distribuée entre les nœuds, et des accélérateurs au sein des nœuds. C'est dans cette optique que nous orientons cette recherche.

Pour cela, nous avons réalisé l'implémentation de la méthode multipôle sur une machine de calcul hétérogène – mémoires partagée et distribuée, et GPU – en utilisant un ordonnanceur de tâches et des bibliothèques de calcul haute performance. L'algorithme a dû être adapté pour convenir à ce type d'approche et nous avons dû réarranger nos données pour créer des tâches ordonnançables aisément et de manière performante. Nous nous sommes attaché à implémenter de manière efficace les tâches, le tout en utilisant des bibliothèques pérennes qui, nous espérons, sauront suivre l'évolution des architectures.

Organisation du document

Pour exposer nos travaux, nous avons choisi de découper ce document en quatre chapitres : une présentation de la méthode multipôle et des supercalculateurs, une présentation des travaux sur la méthode multipôle nous aidant à exploiter au mieux les ressources des supercalculateurs ainsi que les outils informatiques, notre contribution en utilisant l'ordonnanceur dynamique StarPU, et enfin les résultats que nous avons obtenus.

Le premier chapitre intitulé *Présentation* est découpé en deux parties. La première concerne la méthode multipôle. Nous montrerons d'abord comment intervient la méthode multipôle en nous servant d'un exemple académique tiré de l'acoustique pour des soucis de simplicité, l'important étant que le noyau soit de type oscillant. Ensuite, nous détaillerons le fonctionnement de la méthode pour en comprendre chaque étape. Dans la deuxième partie, nous nous attacherons à présenter les architectures des machines de calcul des plus petites en passant par les plus grosses, en montrant leur évolution

Le deuxième chapitre intitulé *État de l'art*, est découpé en deux parties. Dans la première, nous ferons un tour d'horizon des travaux concernant la méthode multipôle. Ceux-ci nous seront utiles pour effectuer notre implémentation. Dans la deuxième partie, nous présenterons les différentes techniques et les différents outils visant à obtenir des implémentations efficaces sur les supercalculateurs.

Dans le troisième chapitre, intitulé *Contribution*, nous attaquerons le cœur de notre travail. Nous y présenterons les travaux effectués. Nous expliquerons pourquoi nous avons choisi d'utiliser un ordonnanceur dynamique de tâches. Nous détaillerons alors quelles modifications ont été nécessaires dans l'algorithme multipôle pour y parvenir. En particulier, nous décrirons de manière approfondie toutes les tâches à réaliser et comment elles ont dû être adaptées. Ensuite nous verrons l'adaptation que nous avons faite pour utiliser la mémoire distribuée. Puis, nous verrons comment s'est produite l'intégration de l'ordonnanceur que nous avons choisi, StarPU. Enfin, nous exposerons les différentes optimisations au sein des tâches, à la fois pour leur implémentation sur CPU et sur GPU.

Le quatrième et dernier chapitre *Résultats* est l'évaluation de nos travaux. En premier lieu, nous allons donner quelques grandeurs caractéristiques de la méthodes multipôle pour nous rendre compte des valeurs types, pour établir un cadre de tests. Suite à cela, nous évaluerons nos tâches en comparant les différentes implémentations sur CPU et GPU en regardant l'influence des tailles d'entrée. Une fois que nous aurons vu quelles sont les implémentations qui se comportent le plus

efficacement en termes de performance, nous pourrons aborder et même tester l'exécution sur un nœud en mémoire partagée. Ces tests permettront de voir l'impact de l'ordonnanceur sur l'algorithme et d'évaluer l'intérêt de l'utilisation des GPU. Enfin, nous pourrons tester notre travail sur une machine complète comportant plusieurs nœuds et faire des tests de scalabilité.

Sommaire

Chapitre

1

Présentation

It's hardware that makes a machine fast. It's software that makes a fast machine slow.

Craig Bruce.

Nous souhaitons calculer plusieurs produits matrices-vecteurs ayant une forme bien particulière. Ce calcul est fait par une méthode ayant une complexité moindre qu'une méthode directe : la méthode multipôle rapide *Plane Wave Fast Multipole Method*. Pour résoudre des problèmes de grande taille, nous devons exécuter notre algorithme sur des supercalculateurs, permettant davantage de stockage et une puissance de calcul accrue.

1.1 Description de la problématique

Nous allons voir quelle est la particularité de ce produit matrice-vecteur à accélérer, et dans quel cas nous pouvons le rencontrer.

1.1.1 Position du problème

Le problème au cœur de nos travaux est de calculer de manière rapide

$$V_i = \sum_{j \in I,\, j \neq i} \mathbb{G}_{i,j}\, U_j, \quad i \in I$$

où I est un ensemble fini d'indices, $(U_i)_{i \in I}$ est un vecteur complexe (ou réel) donné, et $(V_i)_{i \in I}$ est le vecteur dont nous voulons calculer les composantes. Enfin, la matrice $(\mathbb{G}_{i,j})_{i,j \in I \times I}$ a une structure bien particulière : ses éléments se calculent à l'aide de la formule

$$\mathbb{G}_{i,j} = G(x_i, x_j) = \frac{e^{\mathrm{i}k|x_i - x_j|}}{4\pi |x_i - x_j|}.$$

$G(x, y)$ est le *noyau de Green* sortant de l'équation d'Helmholtz 3D. Le nombre $k = \frac{2\pi}{\lambda}$, que nous supposons réel, positif et non nul, est le nombre d'onde ; λ est la longueur d'onde ; les x_i, $i \in I$ sont des points de l'espace tridimensionnel formant un nuage de points. Le noyau est la solution élémentaire sortante de l'équation d'Helmholtz

$$k^2 G(x, y) + \Delta_x G(x, y) = -\delta(x - y).$$

Le qualificatif « sortant » signifiant que G vérifie la condition de Sommerfield à l'infini

$$\lim_{|x|\to\infty} |x| \left(\partial_{|x|} G(x,y) - ikG(x,y)\right) = 0.$$

Ce noyau (et ses dérivées) intervient de manière systématique dans tous les problèmes d'ondes stationnaires (i.e. à fréquence fixée) en milieu homogène ou, au moins, homogène par morceaux. Nous pouvons citer l'acoustique, l'électromagnétisme ou encore l'élastodynamique. Une bonne introduction à tous ces problèmes d'ondes est la monographie de Colton Kress [25] ou encore les livres de Nédelec [63] et de Chen et Zhou [22], pour les aspects mathématiques, ou encore le livre de Jones [48] pour les applications en électromagnétisme.

Avant de présenter la méthode multipôle qui permet d'obtenir, à moindre coût, le produit matrice-vecteur qui nous intéresse, nous proposons d'illustrer et de motiver l'intérêt de ce calcul rapide par un exemple académique.

1.1.2 Exemple d'application

Dans cette section, nous montrons, sur un exemple assez simple, où intervient le produit matrice vecteur présenté précédemment. L'idée est de donner une motivation un peu concrète à travers une application possible de notre travail. Nous avons opté pour un exemple tiré de l'acoustique car il conduit à des notations moins complexes.

1.1.2.1 Un problème modèle

Prenons une surface fermée régulière Γ délimitant un domaine fermé borné Ω_i. Nous nous intéressons à la diffraction d'une onde incidente u_{inc} (une onde plane par exemple) sur l'obstacle tridimensionnel Ω_i que nous supposons rigide, voir la figure 1.1. En régime stationnaire, le problème peut se modéliser par le système d'équations aux dérivées partielles

$$\begin{cases} k^2 u + \Delta u = 0 & \text{dans l'extérieur de } \Omega_i \\ u = -u_{inc} & \text{sur } \Gamma \\ \lim\limits_{|x|\to\infty} |x| \left(\partial_{|x|} u(x) - iku(x)\right) = 0 & \text{à l'infini.} \end{cases}$$

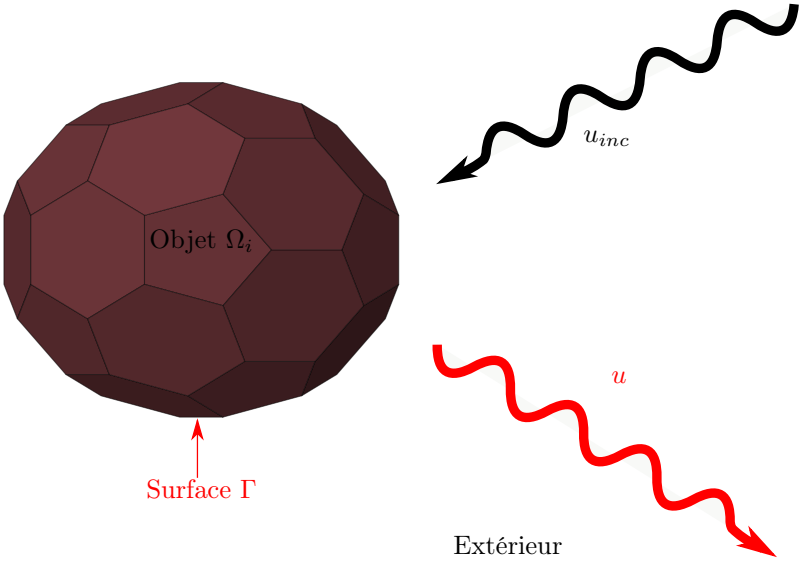


FIGURE 1.1 – Objet soumis à une onde plane incidente

La solution peut être recherchée sous la forme du potentiel vérifiant la condition à l'infini

$$u(x) = \int_\Gamma G(x,y)\lambda(y)\, d\Gamma(y),$$

avec

$$G(x,y) = \frac{e^{ik|x-y|}}{4\pi|x-y|},$$

l'inconnue $\lambda(y)$ s'interprétant comme la dérivée normale de $u + u_i$ sur Γ, [25]. Pour déterminer l'inconnue $\lambda(y)$, on tient compte de la condition à la limite sur Γ pour aboutir à l'équation intégrale

$$-u_i(x) = \int_\Gamma G(x,y)\lambda(y)\, d\Gamma(y),\ x \in \Gamma. \tag{1.1}$$

Nous pouvons maintenant passer à la discrétisation de celle-ci.

1.1.2.2 Problème modèle discrétisé

Nous proposons de calculer une approximation de la solution de l'équation intégrale (1.1) en nous appuyant sur une triangulation de Γ et sur l'utilisation de la *méthode de Galerkin*. Pour cela, choisissons un ensemble de triangles $\mathcal{T}_h$, h étant une taille caractéristique du diamètre des triangles et nous approchons Γ par

$$\Gamma \simeq \Gamma_h = \bigcup_{T \in \mathcal{T}_h} T.$$

Ensuite, nous considérons l'ensemble V_h des fonctions dont la restriction à chaque triangle de $\mathcal{T}_h$ est constante. La solution est ainsi recherchée dans V_h, c'est-à-dire sous la forme

$$\lambda(x) \simeq \lambda_h(x) = \sum_{T \in \mathcal{T}_h} \lambda_T\, \chi_T(x),$$

χ_T est la fonction indicatrice du triangle T : 1 sur T, 0 ailleurs.

La solution numérique est alors complètement déterminée si l'on connaît les $N_T = \text{cardinal}(\mathcal{T}_h)$ valeurs λ_T. Nous avons donc besoin de N_T équations pour les obtenir. L'idée est de moyenner l'équation intégrale sur chaque triangle T' de la triangulation. Le problème discrétisé se réécrit en conséquence :

$$\textit{trouver } \lambda_T,\ T \in \mathcal{T}_h,\ \textit{tel que pour tout } T' \textit{ de } \mathcal{T}_h,$$

$$\sum_{T \in \mathcal{T}_h} \left(\int_{T'} \int_T G(x,y) dT(y)\, dT'(x) \right) \lambda_T = -\int_{T'} u_{inc}\, dT'(x)$$

soit un système linéaire de la forme

$$\mathbb{Z}\, I = V,$$

avec

$$\mathbb{Z}_{T,T'} = \int_{T'} \int_T G(x,y) dT(y)\, dT'(x), \quad V_{T'} = -\int_{T'} u_{inc}(x)\, dT'(x), \quad I_T = \lambda_T.$$

Les coefficients $\mathbb{Z}_{T,T'}$ de la matrice doivent être calculés numériquement. Pour cela nous utilisons des formules de quadrature standards :

$$\mathbb{Z}_{T,T'} \simeq \mathbb{Z}^q_{T,T'} = \sum_g \sum_g G(x_g^T, x_{g'}^{T'})\, \varpi_g^T \varpi_{g'}^{T'}, \quad T \text{ et } T' \text{ loins.}$$

Les x_g^T, sont les points de Gauss sur T et les ϖ_g^T sont les poids de Gauss sur T.

Cependant, cette formule de quadrature n'est applicable que quand T et T' sont suffisamment éloignés, à cause de la singularité du noyau $G(x,y)$ quand $x = y$. Dans le cas de paires de triangles proches, nous utilisons des techniques plus sophistiquées.

Ainsi, la matrice se décompose en une somme de deux matrices

$$\mathbb{Z} = \mathbb{Z}^q + \mathbb{Z}^s$$

avec s pour sophistiqué et q pour quadrature, et

$$\mathbb{Z}^s_{T,T'} = \begin{bmatrix} \mathbb{Z}^s_{T,T'} & \text{si } T, T' \text{ proches} \\ 0 & \text{sinon} \end{bmatrix}, \quad \mathbb{Z}^q_{T,T'} = \begin{bmatrix} \mathbb{Z}^q_{T,T'} & \text{si } T, T' \text{ loins} \\ 0 & \text{sinon} \end{bmatrix}.$$

Le système linéaire étant assemblé, il ne reste plus qu'à le résoudre. Cependant, avant d'aborder ce point, nous allons discuter de l'erreur induite par ce procédé.

1.1.2.3 Caractéristique de l'erreur

L'erreur sur la solution numérique provient de plusieurs sources, nous pouvons citer cinq types d'erreurs.

L'erreur géométrique Nous avons approché Γ par Γ_h. D'une certaine façon, nous résolvons plutôt le problème de diffraction sur l'ouvert complémentaire du domaine polyédrique extérieur à Γ_h.

L'erreur d'approximation La solution est approchée dans V_h. Ainsi, h doit être assez petit pour que cette erreur soit faible.

L'erreur sur l'équation L'équation n'est satisfaite qu'en moyenne sur chaque triangle, et non point par point.

L'erreur de quadrature Nous ne calculons pas de façon exacte les éléments matriciels.

L'erreur de résolution du système linéaire Si nous utilisons une méthode directe, méthode LU par exemple, cette erreur est due à l'arithmétique finie de l'ordinateur. Si nous employons une méthode itérative, nous n'obtiendrons qu'une solution approchée avec contrôle uniquement du résidu sur l'équation.

À cette liste, nous pourrions ajouter une erreur supplémentaire, qui est l'erreur induite par le calcul multipôle. Nous verrons que celui-ci consiste à ne pas calculer de manière exacte la fonction de Green qui intervient dans $\mathbb{Z}^q$, la partie formée par les interactions lointaines de la matrice du système. D'une certaine façon, cette erreur est de même nature que l'erreur de quadrature. Cette remarque nous semble intéressante car elle permet de relativiser l'impact de l'utilisation de la FMM sur la qualité de la solution : l'erreur FMM est une erreur parmi d'autres.

En ce qui concerne l'erreur d'approximation, de nombreuses études ont été réalisées, [23]. Dit rapidement, la méthode est précise lorsque le produit kh ou encore le rapport $\frac{h}{\lambda}$ est suffisamment petit (h est le plus grand diamètre des triangles). Si nous définissons $N_\lambda = \frac{\lambda}{h}$, le nombre de points par longueur d'onde pour le plus grand triangle, il est d'usage de prendre de l'ordre de 10 points. Regardons la taille du système linéaire de notre problème, qui est le nombre de triangles. Nous avons, si S est la surface de Γ, et si nous estimons la surface de chaque triangle par $\frac{h^2}{2}$ (un carré divisé en deux triangles) :

$$N_T \sim \frac{2S}{h^2} = \frac{2\lambda^2}{h^2}\frac{S}{\lambda^2} = 2N_\lambda^2 \frac{S}{\lambda^2} = 8\pi^2 N_\lambda^2 k^2 S.$$

Ainsi, à précision fixée, le nombre de triangles croît comme l'inverse du carré de la longueur d'onde. Le nombre d'inconnues va donc augmenter comme le carré du nombre d'onde k du problème. À géométrie fixée, le nombre d'inconnues va ainsi être important pour k grand : on parlera alors de problème haute fréquence. Dans certaines applications, il n'est pas rare d'atteindre plusieurs millions d'inconnues, [70, 69]. Pour cette taille de problème, une mise en œuvre efficace d'assemblage et de résolution du système est indispensable.

1.1.2.4 Résolution du système linéaire

Pour obtenir la solution numérique, nous devons résoudre le système linéaire $\mathbb{Z}I = V$ de taille N_T. Celui-ci peut être résolu de manière directe, en inversant la matrice. Par exemple, nous pouvons utiliser une factorisation `LU` ou `LDL`$^{\mathtt{T}}$. Ces méthodes ont une complexité de calcul qui est en $O(N_T^3)$. Les avantages de la méthode directe sont la précision du résultat, et qu'une fois la factorisation faite, le calcul de la solution est en $O(N_T^2)$ pour n'importe quel second membre.

Le nombre d'inconnues pouvant être grand, il est intéressant d'employer une méthode itérative, qui fournit une solution approchée dont la précision est inférieure à l'erreur de géométrie ou d'approximation. Le coût de ces méthodes varie comme le produit du nombre N_i d'itérations nécessaires à la convergence, par le coût d'un produit matrice-vecteur. Choisir un solveur efficace est primordial pour que N_i soit faible. Pour le type de matrices que nous manipulons (symétriques mais non hermitiennes), il existe plusieurs méthodes itératives permettant d'obtenir la solution approchée notamment *GMRES*, le gradient bi-conjugué, *QMR*,... Chacune d'elles peut être préconditionnée (à gauche ou/et à droite), le préconditionneur permettant de diminuer encore N_i. Chacune de ces méthodes a la particularité d'effectuer des produits matrice-vecteur à chaque itération. C'est le coût de ces produits qui donnera la complexité de la méthode. Notre matrice $\mathbb{Z}$ étant pleine, une approche directe donnera une complexité en $N_i\, N_T^2$, ce qui est encore trop important pour les problèmes à plus de 100000 inconnues. C'est là qu'intervient la méthode multipôle en fournissant le résultat, avec une erreur contrôlée, pour une complexité bien moindre : en $O(N_T \log N_T)$ d'après Darve, et en $O(N_T)$ pour Chew (ce dernier utilisant des interpolateurs non spectraux plus efficaces mais moins précis).

1.1.2.5 Le produit matrice vecteur

Si nous revenons au produit $U = \mathbb{Z}I$ qui nous intéresse, nous proposons ici une transformation qui permet de s'affranchir du maillage, du choix particulier des fonctions de base, des poids de quadrature, ..., en nous ramenant à un produit qui ne fait intervenir qu'un nuage de points (en l'occurrence le nuage constitué par l'ensemble de tous les points de Gauss de tous les triangles). Pour ce faire, nous remarquons que la décomposition

$$\mathbb{Z} = \mathbb{Z}^q + \mathbb{Z}^s$$

avec

$$\mathbb{Z}^s_{T,T'} = \left[\begin{array}{ll} \mathbb{Z}^s_{T,T'} & \text{si } T, T' \in \tilde{\mathcal{P}} \\ 0 & \text{sinon} \end{array} \right], \quad \mathbb{Z}^q_{T,T'} = \left[\begin{array}{ll} \mathbb{Z}^q_{T,T'} & \text{si } T, T' \notin \tilde{\mathcal{P}} \\ 0 & \text{sinon} \end{array} \right]$$

et si G_T et $G_{T'}$ sont les centres de gravité de T et T',

$$\tilde{\mathcal{P}} = \{(T, T'),\, |G_T - G'_T| \leq 3 \max(\text{diam T}, \text{diam T'})\}$$

peut se réécrire

$$\mathbb{Z} = \tilde{\mathbb{Z}}^q + \tilde{\mathbb{Z}}^s$$

avec

$$\tilde{\mathbb{Z}}^s_{T,T'} = \left[\begin{array}{ll} \mathbb{Z}^s_{T,T'} - \tilde{\mathbb{Z}}^q_{T,T'} & \text{si } T, T' \in \tilde{\mathcal{P}} \\ 0 & \text{sinon} \end{array} \right]$$

et

$$\tilde{\mathbb{Z}}^q_{T,T'} = \sum_{g=1,3} \sum_{g'=1,3} G(x_g^T, x_{g'}^{T'})\, \varpi_g^T \varpi_{g'}^{T'}, \quad T \neq T'$$

$$\tilde{\mathbb{Z}}^q_{T,T} = \sum_{g=1,3} \sum_{g'=1,3, g' \neq g} G(x_g^T, x_{g'}^T)\, \varpi_g^T \varpi_{g'}^{T'}, \quad T = T'.$$

Le but de cette transformation est d'ôter la dépendance en $\mathcal{P}$ sur $\tilde{\mathbb{Z}}^q$ en reportant celle-ci sur la matrice des interactions proches $\tilde{\mathbb{Z}}^s$. L'assemblage de $\tilde{\mathbb{Z}}^s$ a des complexités en stockage et en calcul en $O(N_T)$. Le produit creux avec un vecteur, $\tilde{\mathbb{Z}}^s I$, peut se faire en N_T opérations. La complexité du produit matrice-vecteur global n'est donc induite que par la partie en $\tilde{\mathbb{Z}}^q I$, et c'est sur elle que doit porter l'effort.

Introduisons le nuage de points formés par tous les points de Gauss :

$$X = \left\{ x_i \in \mathbb{R}^3, \setminus\ x_i = x_g^T,\, T \in \mathcal{T}_h,\, g,\, i = 1 \cdots N \right\}.$$

Décomposons le produit $\tilde{\mathbb{Z}}^q I$ en trois étapes

1. Nous formons le vecteur $\tilde{I}$ au point de Gauss, pour $i = 1, ..., n$

$$\tilde{I}_i = \varpi_g^T\, I_T, \text{ si } x_i = x_g^T.$$

2. Nous formons le vecteur $\tilde{V}$ au point de Gauss

$$\tilde{V}_j = \sum_{i=1, i\neq j}^{n} G(x_i, x_j)\tilde{I}_i,\ j = 1 \cdots n.$$

3. Nous formons le vecteur V

$$V_T = \sum_g \tilde{V}_{j(g,T)} \varpi_g^T$$

où $j(g,T)$ est l'indice du point de X tel que $x_j = x_g^T$.

Les étapes 1 et 3 correspondant à $O(N_T)$ opérations, le coût du produit matrice-vecteur provient de l'étape 2. Son optimisation via l'utilisation de la méthode multipôle est l'objet de notre travail.

Pour davantage de clarté, dans la suite, nous ne parlerons plus de ces points x_i comme points de quadrature mais comme simples points, le but étant d'éviter la confusion avec une nouvelle quadrature que nous aborderons plus tard.

1.2 Méthode multipôle

Nous présentons dans cette section, la méthode multipôle pour l'évaluation rapide du produit matrice vecteur

$$V_i = \sum_{j=1, j\neq i}^{n} \mathbb{G}_{i,j}\, U_j, \quad i \in I$$

avec

$$\mathbb{G}_{i,j} = G(x_i, x_j) = \frac{e^{ik|x_i - x_j|}}{4\pi|x_i - x_j|}.$$

La méthode multipôle a été introduite en 1987 par Rokhlin et Greengard [41]. Dans les années 90, elle a été adaptée à l'électromagnétisme par Rokhlin [24]. Song et Chew [71], Darve [28], et Rahola [67] en ont proposé une version diagonale. C'est cette version diagonale que nous avons retenue. Une description pédagogique de cette méthode est donnée dans [9], ou [73] avec une version algébrique de la méthode.

1.2.1 Décomposition par blocs

L'idée sur laquelle se fondent toutes les méthodes qui conduisent à une accélération du produit matrice-vecteur repose sur une compression de certains blocs de la matrice $\mathbb{G}$. Ces blocs sont construits en exploitant le fait que les indices de colonnes et de lignes de cette matrice correspondent de fait à des points de l'espace. En regroupant ces points par paquets de points (géométriquement regroupés), on peut associer à toute paire de paquets de points, un bloc de la matrice $\mathbb{G}$. Par exemple, soient deux paquets de points $\mathcal{B}_t$ et $\mathcal{B}_s$ [1], le bloc correspondant sera

$$\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s} := (\mathbb{G}_{i,j})_{x_i\in\mathcal{B}_t,\, x_j\in\mathcal{B}_s}.$$

Lorsque les paquets de points sont éloignés, nous observons que les variations des coefficients matriciels du bloc sont régulières lorsque nous bougeons les points x_i et x_j dans leurs paquets respectifs. C'est cette régularité qui peut être exploitée pour faire de la compression.

1.2.1.1 Compression de blocs matriciels

Il existe de nombreuses factorisations conduisant à des algorithmes efficaces de compression. Nous pouvons déjà citer les factorisations permettant d'accélérer les algorithmes `LU` ou $\texttt{LDL}^{\texttt{T}}$ pour l'inversion des matrices. Pour notre part, nous les utilisons pour accélérer le produit matrice-vecteur.

Une première approche est d'effectuer une décomposition en valeurs singulières (`SVD`). Elle consiste à factoriser $\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s}$ de taille $m\times n$ sous la forme

$$\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s} = {}^t\mathbb{U}\Lambda\mathbb{V}$$

où $\mathbb{U}$, $\mathbb{V}$ sont des matrices orthogonales de tailles respectives $m\times p$ et $n\times p$ avec $p=\min(n,m)$. La matrice Λ de taille $p\times p$ est diagonale, ces termes diagonaux contenant les *valeurs singulières* du bloc $\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s}$. Lorsque les blocs sont éloignés, nous pouvons observer que certaines valeurs singulières sont très petites. En les négligeant, nous obtenons une version compressée de la forme

$$\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s} \simeq {}^t\mathbb{U}\Lambda'\mathbb{V} \tag{1.2}$$

avec Λ' diagonale mais avec $q<p$ éléments non nuls. Une fois la décomposition obtenue, le produit matrice vecteur correspondant au bloc factorisé s'obtient en $Cq(n+m)$ opérations élémentaires, ce qui se compare favorablement au coût d'un calcul direct qui est en Cnm. Le problème est que la `SVD` est une opération coûteuse et qui demande d'assembler la totalité des éléments de la matrice. La méthode *ACA* – pour *Adaptative Cross Approximation* – est une méthode purement algébrique développée par Bedendorf, [10] en 2000. En bref, elle permet de construire une approximation de type (1.2) en n'utilisant que certaines lignes et colonnes du bloc $\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s}$. Nous avons ainsi

$$\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s} \simeq {}^t\mathbb{U}^{\mathcal{B}_t,\mathcal{B}_s}\Lambda^{\mathcal{B}_t,\mathcal{B}_s}\mathbb{V}^{\mathcal{B}_t,\mathcal{B}_s}. \tag{1.3}$$

C'est une méthode très intéressante car très générale d'application, et elle connaît un grand retentissement ces dernières années. Malheureusement, elle perd de son efficacité lors d'une montée en fréquence, c'est-à-dire pour k grand, voir [88].

La méthode multipôle quant à elle, utilise pleinement l'expression de G pour construire des approximations de type

$$\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s} \simeq {}^*\mathbb{A}^{\mathcal{B}_t}\mathbb{T}^{\mathcal{B}_t-\mathcal{B}_s}\mathbb{A}^{\mathcal{B}_s} \tag{1.4}$$

(le signe $*$ signifiant transposée conjuguée). Nous nous rendons compte que les matrices $\mathbb{A}^{\mathcal{B}_t}$ et $\mathbb{A}^{\mathcal{B}_s}$ ne dépendent pas de la paire de paquets mais seulement d'un unique paquet. Cela conduit à des factorisations communes des calculs pour la prise en compte de blocs distincts de type $\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s}$ et $\mathbb{G}^{\mathcal{B}'_t,\mathcal{B}'_s}$, lorsque $\mathcal{B}_t=\mathcal{B}'_t$ ou encore $\mathcal{B}_s=\mathcal{B}'_s$.

1. Les notations $\mathcal{B}_t$, $\mathcal{B}_s$ correspondent à boîte cible (*target*) et boîte source ; nous anticipons ici le fait que les paquets de points correspondront par la suite à des boîtes cubiques.

1.2.1.2 Forme spectrale du noyau de Green

Toute la difficulté réside maintenant dans la factorisation elle-même. Pour y parvenir, nous allons utiliser une forme spectrale du noyau d'Helmholtz qui repose sur un théorème d'addition de Gegenbauer et la formule de Funk-Hecke [25, 85]. Nous avons

$$G(x_i, x_j) = \lim_{L\to\infty} \int_{S^2} e^{\mathrm{i}k(x_i-c_t)\cdot\hat{s}}\, T^L(\hat{s}; c_t - c_s)\, e^{-ik(x_j-c_s)\cdot\hat{s}}\, d\sigma(\hat{s}) \tag{1.5}$$

où

- S^2 est la sphère unité, les vecteurs unitaires $\hat{s}$ sont des directions sur S^2, $d\sigma(\hat{s})$ est la mesure de surface sur S^2 ;
- c_s et c_t sont deux points suffisamment éloignés l'un de l'autre tandis que x_i (resp. x_j) est relativement proche de c_s (resp. c_t) ; soient c_s et c_t, qui satisfont à l'inégalité :

$$|x_i - c_t| + |x_j - c_s| \leq \eta |c_t - c_s|. \tag{1.6}$$

  pour un certain η choisi plus petit que 1 ;
- T^L est la fonction de translation entre c_t et c_s :

$$T^L(\hat{s}; c_t - c_s) = \frac{\mathrm{i}k}{16\pi^2} \sum_{l=0}^{L} \mathrm{i}^l (2l+1) h_l^{(1)}(k|c_t - c_s|) P_l\left(\hat{s}\cdot\frac{c_t - c_s}{|c_t - c_s|}\right) \tag{1.7}$$

— $h_l^{(1)}(t)$ est la fonction de Hankel sphérique d'argument t d'ordre l
— $P_l(x)$ est le polynôme de Legendre d'ordre l et d'argument x.

Les fonctions spéciales qui interviennent ici sont bien connues, voir [25] et [1], et sont calculables par des algorithmes répertoriés.

Maintenant, l'idée est de construire une approximation du noyau en utilisant cette formule avec un L fixé. Ce nombre L est par définition *le nombre de multipôles*. Prendre L grand accroît la précision mais augmente les variations de la fonction intégrante. Une fois L choisi, nous allons utiliser une quadrature sur S^2 pour transformer l'intégrale en une somme finie. Si D est le nombre de points de quadratures sur S^2, $\hat{s}_p$ et ϖ_p les points et poids de quadrature, notre approximation est de la forme

$$G(x_i, x_j) \simeq G^{L,D}(x_i, x_j) = \sum_{p=1}^{D} \varpi_p\, e^{ik(x-c_t)\cdot\hat{s}_p}\, T^L(\hat{s}_p; c_t - c_s)\, e^{-ik(y-c_s)\cdot\hat{s}_p}. \tag{1.8}$$

Dans cette formule, nous prenons en compte D directions d'ondes planes. Nous remarquons que le noyau approché se présente comme une somme de fonctions à variables séparées (ondes planes en x_i et en x_j). Cette propriété est essentielle pour établir le lien avec la factorisation recherchée.

1.2.1.3 Une factorisation « géométrique »

Nous allons nous servir de notre formule approchée pour construire notre factorisation pour des blocs éloignés. Supposons que x_i soit dans un paquet $\mathcal{B}^t$ et x_j dans un paquet $\mathcal{B}^s$. Prenons comme points $c_{\mathcal{B}^s}$ et $c_{\mathcal{B}^t}$ les centres de gravité des deux paquets, voir la figure 1.2. La décomposition géométrique

$$x_i - x_j = (x_i - c_{\mathcal{B}^t}) + (c_{\mathcal{B}^t} - c_{\mathcal{B}^s}) - (x_j - c_{\mathcal{B}^s}),$$

couplée à l'approximation (1.8), nous permet de factoriser le bloc $\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s}$ sous la forme (1.4), en prenant

$$\mathbb{A}^{\mathcal{B}^t}_{p,i} = \sqrt{\varpi_p}\, e^{ik(x_i - c_{\mathcal{B}^t})\cdot\hat{s}_p}, \quad \mathbb{A}^{\mathcal{B}^s}_{p,j} = \sqrt{\varpi_p}\, e^{ik(x_j - c_{\mathcal{B}^s})\cdot\hat{s}_p}, \quad \mathbb{T}^{\mathcal{B}_t-\mathcal{B}_s}_{p,p} = T^L(c_{\mathcal{B}^t} - c_{\mathcal{B}^s}; \hat{s}_p). \tag{1.9}$$

L'éloignement imposé par le forme spectrale est vérifié par la création de paquets permettant de séparer les interactions proches des interactions lointaines. En effet, deux points sont considérés comme proches s'ils sont dans le même paquet, mais également s'ils sont dans des paquets voisins (les deux points peuvent être proches de la frontière commune). La séparation des variables vue précédemment ne peut donc se produire que dans des zones éloignées.

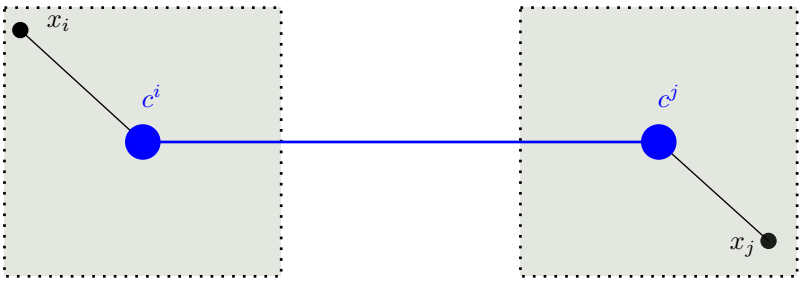


FIGURE 1.2 – Séparation des variables

1.2.1.4 Troncature de la série

D'un point de vue numérique, il est nécessaire de tronquer la série (1.7). Le choix du nombre de multipôles joue un rôle fondamental dans la qualité et la précision de l'approximation. Si L est trop petit, la troncature de la série ne pourra fournir un résultat satisfaisant. À l'inverse, s'il est choisi trop grand, nous rencontrons des problèmes de représentation machine car T^L diverge rapidement lorsque L tend vers l'infini. Plusieurs propositions ont été faites pour le choix de L. Elles ont toutes la propriété essentielle que le bon L ne dépend que de la taille des paquets rapportée au nombre d'ondes, et non pas à la distance entre les paquets, pour peu que le critère d'éloignement soit uniformément satisfait. Si d est le plus grand diamètre de tous les paquets, le nombre de multipôles ne dépend que du produit kd et de la précision à atteindre.

Il n'existe pas de formule exacte donnant le nombre de multipôles, mais simplement des formules d'estimation. Coifman, [24], a été le premier à proposer une formule

$$L = kd + D\,\log(kd+\pi), \quad D = \log_{10}\frac{1}{\epsilon}.$$

, où ϵ est la précision recherchée.

Chew dans [23] a établi le critère semi-empirique

$$L = kd + 1.85\,D^{\frac{2}{3}}(kd)^{\frac{1}{3}}, \quad D = \log_{10}\frac{1}{\epsilon}.$$

Carayol et al., dans [18, 19], ont établi la formule asymptotique (i.e. pour kd grand)

$$L = kd + (\tilde{D}(\epsilon))^{\frac{2}{3}}\,(kd)^{\frac{1}{3}} - \frac{1}{2}$$

avec

$$\tilde{D}(\epsilon) = \frac{1}{4\sqrt{2}} W\left(\left(\frac{\eta+1}{\eta-1}\right)^{\frac{3}{2}}\frac{kd}{4\epsilon^6}\right),$$

où W est la *Fonction de Lambert* $W(x)e^{W(x)} = x$ et η est le nombre défini dans le critère d'éloignement (1.6) et ϵ est une précision à atteindre.

Comme nous l'avons dit, toutes ces formules sont de fait utilisables lorsque kd est assez grand. Pour kd petit, nous rencontrons des problèmes de représentation des grands nombres en machine et les formules multipôles se trouvent altérées par les erreurs d'arrondis, [64]. En règle générale, si nous voulons atteindre une précision ϵ, le produit kd doit être supérieur à une constante dépendant d'ϵ. Cette difficulté n'empêche nullement la méthode multipôle de fournir des résultats en 10^{-4} avec une très bonne complexité, en prenant une constante égale à 0.3 (par exemple sur des maillages en $N_\lambda = \frac{\lambda}{10}$ pour revenir à notre exemple en acoustique présenté précédemment).

1.2.1.5 Discrétisation sur la sphère unité

Pour effectuer le calcul, nous devons maintenant discrétiser l'intégration sur la sphère unité. Pour cela, nous devons choisir une quadrature sur la sphère unité adaptée, pour intégrer correctement le produit des deux ondes planes et de la fonction de translation intervenant dans (1.5). Pour ce faire, nous montrons tout d'abord que le produit des deux ondes planes

$$\hat{s} \mapsto e^{ik[(x-c_{\mathcal{B}^t})-(y-c_{\mathcal{B}^s})]\cdot\hat{s}}$$

est très bien approché par sa projection sur les harmoniques sphériques de degré $L^e(kd)$ avec $d > |x - c_{\mathcal{B}^t}| + |y - c_{\mathcal{B}^s}|$ (cf. [25] pour la définition des harmoniques sphériques).

D'autre part, la fonction de translation est une fonction harmonique de degré égal à L. Le produit de fonctions harmoniques de degré respectif L^e et L étant harmonique de degré $L + L^e$, nous aboutissons à la conclusion que notre loi de quadrature sera précise si elle intègre correctement toutes les fonctions harmoniques de degrés $L + L^e \simeq 2L$.

Une loi simple, qui intègre exactement toutes les harmoniques sphériques de degré inférieur à $2L$, consiste à choisir $L + 1$ points de Gauss Legendre en $\cos\theta$ dans $[-1, 1]$ et $M > 2L + 1$ points équidistribués sur $[0, 2\pi]$ en φ, (θ, φ) étant les angles classiques du système de coordonnées sphériques.

Précisons la quadrature retenue. Pour $L > 0$ fixé, nous considérons les $L + 1$ zéros, x_i^L du polynôme de degré $L + 1$, $P_{L+1}(x)$. Nous posons

$$\theta_i^L = \arccos(x_i^L), i = 1, ..., L + 1, \quad \varphi_j^L = \frac{2\pi(j-1)}{2L+1},\ j = 1, ..., 2L + 1$$

et nous prenons

$$\hat{s}_p := \hat{s}_{i,j} = (\sin\theta_i^L \cos\varphi_j^L, \sin\theta_i^L \sin\varphi_j^L, \cos\theta_i^L)$$

$$\text{avec les poids } \varpi_p := \varpi_i = \frac{2\pi}{2L+1} \frac{2(1-(x_i^L)^2)}{[(L+1)P_L(x_i^L)]^2}.$$

Nous retiendrons que, dès que le nombre de multipôles L, est choisi, la quadrature et donc le nombre D de directions sont alors complètement déterminés. Si nous comptons le nombre de directions impliquées, nous trouvons approximativement $D \simeq L^2$ points de quadrature. Comme L croît avec la taille d du paquet, nous voyons que plus d est grand, plus le nombre de directions à prendre en compte est important. Cette croissance du nombre D avec le diamètre des paquets d est une spécificité du noyau d'Helmholtz. Elle n'existe pas lorsque nous traitons du noyau statique homogène $\frac{1}{4\pi[x-y]}$, qui correspond à $k = 0$.

1.2.2 Méthode multipôle simple-niveau

En utilisant tous ces ingrédients, grâce à un choix pratique des paquets, la méthode multipôle parvient à accélérer le produit.

1.2.2.1 Principe

La FMM est basée sur un partitionnement de l'espace. Premièrement, une partition de points, $\mathcal{P}$, est créée, basée sur des critères géométriques. Le partitionnement est constitué de boîtes. Si $\mathcal{B}$ est une boîte de la partition, nous définissons :

$$u^{\mathcal{B}} = (U_i)_{x_i \in \mathcal{B}} \qquad v^{\mathcal{B}} = (V_i)_{x_i \in \mathcal{B}} \qquad \mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s} = (\mathbb{G}_{i,j})_{(x_i,x_j)\in(\mathcal{B}_t\times\mathcal{B}_s)}. \tag{1.10}$$

Grâce aux étapes vues précédemment, nous avons une factorisation approchée de $\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s}$ si $\mathcal{B}_t$ et $\mathcal{B}_s$ ne sont pas voisins, ce qui signifie qu'ils n'ont aucun sommet en commun.

$$\mathbb{G}^{\mathcal{B}_t,\mathcal{B}_s} \simeq \left(\mathbb{A}^{\mathcal{B}_t}\right)^\star\ \mathbb{T}^{\mathcal{B}_t-\mathcal{B}_s}\ \mathbb{A}^{\mathcal{B}_s}, \qquad \text{avec } \mathcal{B}_s \notin \mathcal{V}(\mathcal{B}_t) \tag{1.11}$$

où :

— $\mathcal{V}(\mathcal{B}_t)$, l'ensemble des voisins de $\mathcal{B}_t$;
— $\mathbb{A}^{\mathcal{B}}$ est une matrice $D \times N^{\mathcal{B}}$, appelée matrice d'agrégation ;
— $\mathbb{T}^{\mathcal{B}_t-\mathcal{B}_s}$ est une matrice diagonale $D \times D$, appelée matrice de translation ;
— $\mathbb{A}^\star$ est la transposée conjuguée de $\mathbb{A}$;
— $N^{\mathcal{B}}$ est le nombre d'éléments dans $\mathcal{B}$ choisi proportionnel à $\sqrt{n}$;

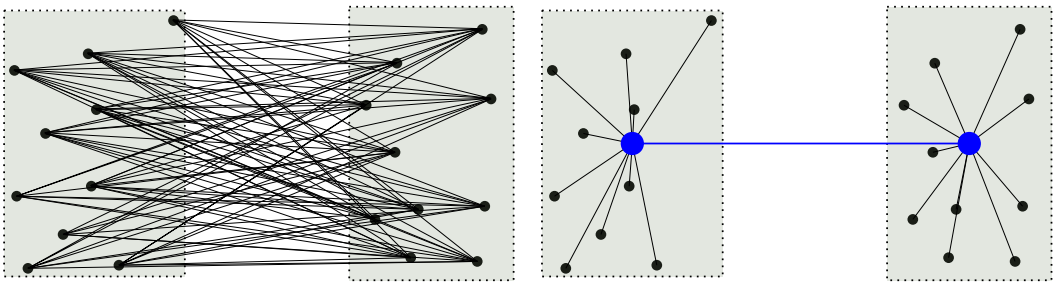


FIGURE 1.3 – Direct versus méthode multipôle

— D est le nombre de points pris sur la sphère unité (1.8), il est appelé le nombre de directions (c'est-à-dire d'ondes planes de quadrature).

La factorisation est valide seulement si $\mathcal{B}_t$ et $\mathcal{B}_s$ ne sont pas voisins. Nous devons donc séparer les calculs de $v^{\mathcal{B}_t}$ en deux parties :

$$v^{\mathcal{B}_t} = v^{\mathcal{B}_t}_{\text{far}} + v^{\mathcal{B}_t}_{\text{near}}, \tag{1.12}$$

$v^{\mathcal{B}_t}_{\text{near}}$ est calculé directement, mais pour $v^{\mathcal{B}_t}_{\text{far}}$, la factorisation donne

$$v^{\mathcal{B}_t}_{\text{far}} \simeq \left(\mathbb{A}^{\mathcal{B}_t}\right)^{\star} \sum_{\mathcal{B}_s \notin \mathcal{V}(\mathcal{B}_t)} \mathbb{T}^{\mathcal{B}_t - \mathcal{B}_s}\ \mathbb{A}^{\mathcal{B}_s} u^{\mathcal{B}_s} \tag{1.13}$$

$\mathbb{A}^{\mathcal{B}_s} u^{\mathcal{B}_s}$ est calculé une seule fois par boîte $\mathcal{B}_s$, pour chaque boîte $\mathcal{B}_t$. Alors que sans la factorisation, la multiplication est calculée pour chaque couple de boîtes (ou points). C'est la source de l'accélération de la méthode multipôle simple.

Avec tous ses éléments, nous avons les bases pour définir une méthode pour effectuer un calcul rapide du produit.

1.2.2.2 Point de vue géométrique

D'un point de vue géométrique, la réduction de la complexité de calcul produite par la méthode multipôle se voit aisément. La figure 1.3 nous montre deux ensembles de points qui interagissent. À gauche, les interactions sont effectuées directement, leur nombre est donc égal au produit du nombre de points du premier ensemble avec le nombre de points du deuxième ensemble. À droite, nous voyons le principe de la méthode multipôle : pour chaque ensemble de points une interaction équivalente est calculée à partir des points de l'ensemble. L'interaction d'un point avec un autre appartenant à un ensemble distant se fera par l'intermédiaire de l'interaction équivalente. Ainsi, le nombre d'interactions à calculer n'est plus le produit du nombre de points des paquets mais la somme (à laquelle il faut rajouter l'interaction équivalente entre les deux paquets).

1.2.2.3 Fonctionnement

Comme nous l'avons vu précédemment, la méthode multipôle repose sur un partitionnement de l'espace. Pour cela, nous allons découper notre objet en cubes de taille d dans lesquels nous calculerons des champs équivalents. Nous pouvons voir un tel découpage à la figure 1.4. Pour améliorer la visualisation, nous avons dessiné les faces et non les points de quadratures. Par la suite, tous nos schémas seront en 2D pour des soucis de lecture, mais nous travaillons bien sur des objets 3D.

Comme nous ne considérons que la surface de l'objet, seuls les cubes contenant des points de quadrature des faces de l'objet vont être conservés. Une fois cette opération faite, le calcul s'effectue en deux parties : le calcul des interactions proches et le calcul des interactions éloignées, comme montré à la figure 1.5. La boîte rouge est la boîte pour laquelle les interactions sont calculées, et les boîtes grises sont celles impliquées dans le calcul.

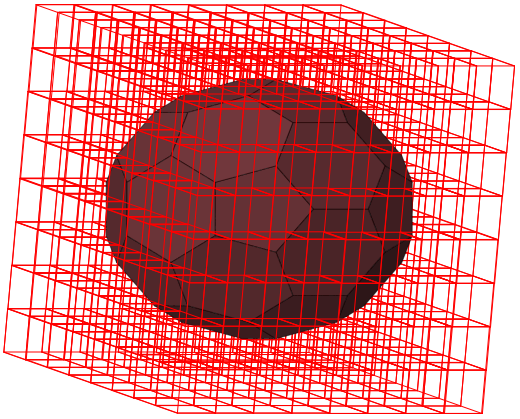
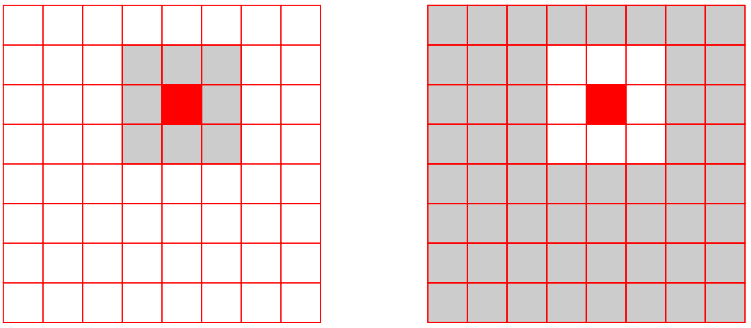


FIGURE 1.4 – Découpage de l'objet en cubes



(a) Les interactions proches (b) Les interactions lointaines

FIGURE 1.5 – Les deux parties de la méthode multipôle

Interactions proches Nous ne nous attarderons pas sur le calcul des champs proches qui est simplement un calcul direct, comme nous pouvons le voir à la figure 1.6. Il s'agit simplement du calcul direct entre tous les points de la boîte considérée (boîte rouge) et tous les points de ses voisines (boîte rouge et boîtes grises).

Interactions lointaines Les interactions lointaines représentent le cœur de la méthode multipôle. Leur calcul s'effectue en trois étapes qui sont : l'agrégation, la translation et la désagrégation.

Agrégation Elle consiste à calculer le champ équivalent produit par l'ensemble des points contenus dans une boîte.

$$F^{\mathcal{B}_s} = \mathbb{A}^{\mathcal{B}_s} u^{\mathcal{B}_s}, \quad \forall \mathcal{B}_s \in \mathcal{P}. \tag{1.14}$$

Le vecteur $u^{\mathcal{B}_s}$ est le projeté sur la boîte $\mathcal{B}_s$ du vecteur u d'entrée. La matrice $\mathbb{A}^{\mathcal{B}_s}$ permet de calculer le paquet d'ondes planes engendré par les charges contenues dans la boîte $\mathcal{B}_s$. Le centre de phase du paquet est centré au centre de la boîte. Elle a autant de colonnes qu'il y a de points dans la boîte et autant de lignes qu'il y a de directions (D). $F^{\mathcal{B}_s}$ est un D vecteur, appelé le vecteur champ de la boîte $\mathcal{B}_s$.

Cette opération est schématisée à la figure 1.7a pour la boîte rouge. Les flèches représentent la translation des champs des points, suite à l'action de la matrice $\mathbb{A}$.

La matrice $\mathbb{A}^{\mathcal{B}_s}$ est calculée avec la formule suivante :

$$\mathbb{A}^{\mathcal{B}_s}_{i,j} = \mathrm{e}^{\mathrm{i}k\overrightarrow{x_i c_{\mathcal{B}^s}}\cdot\vec{s_j}} \sqrt{\varpi_j} \tag{1.15}$$

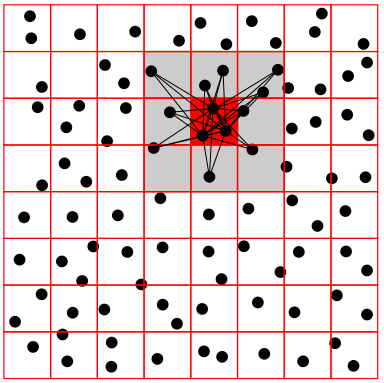


FIGURE 1.6 – Les interactions proches sont calculées directement

Translation Elle correspond au calcul de la contribution sur la boîte $\mathcal{B}_t$ provenant des boîtes non voisines. Elle est la somme des champs des boîtes éloignées F, pondérées par une matrice de translation $\mathbb{T}$ dépendant de la distance entre les boîtes concernées.

$$N^{\mathcal{B}_t} = \sum_{\mathcal{B}_s \notin \mathcal{V}(\mathcal{B}_t)} \mathbb{T}^{\mathcal{B}_s - \mathcal{B}_t} \; F^{\mathcal{B}_s} \tag{1.16}$$

$\mathbb{T}^{\mathcal{B}_s - \mathcal{B}_t}$, la matrice de translation de la boîte $\mathcal{B}_s$ vers la boîte $\mathcal{B}_t$, est une matrice diagonale de taille le nombre de multipôles. Elle ne dépend que du vecteur $\overrightarrow{c_{\mathcal{B}^s} c_{\mathcal{B}^t}}$ où $c_{\mathcal{B}^i}$ est le centre de la boîte $\mathcal{B}_i$. $N^{\mathcal{B}_t}$ est un vecteur de taille D.

La figure 1.7b représente le calcul de $N^{\mathcal{B}_t}$ pour la boîte rouge. Les boîtes grises sont les boîtes non voisines de la boîte rouge et participent donc à la translation. Les flèches représentent l'action de T sur le champ équivalent d'une boîte non voisine.

La matrice de translation est une matrice diagonale définie par la formule 1.7 ce qui donne :

$$\mathbb{T}^{\mathcal{B}_s - \mathcal{B}_t}_{i,i} = \frac{\mathrm{i}k}{16\pi^2} \sum_{l=0}^{L} \mathrm{i}^l (2l+1) h_l^{(1)}(k|c_{\mathcal{B}^t} - c_{\mathcal{B}^s}|) P_l \left(\hat{s}_i \cdot \frac{c_{\mathcal{B}^t} - c_{\mathcal{B}^s}}{|c_{\mathcal{B}^t} - c_{\mathcal{B}^s}|} \right). \tag{1.17}$$

Désagrégation Elle correspond à l'opération inverse de l'agrégation. Maintenant que le champ équivalent provenant des boîtes non voisines a été calculé, il est nécessaire de se ramener à la valeur du vecteur V_j en tout point x_j de chaque boîte de la partition $\mathcal{P}$. Voici comment nous retrouvons le champ au niveau du projeté du vecteur :

$$v^{\mathcal{B}_t}_{\text{far}} = \left(\mathbb{A}^{\mathcal{B}_t} \right)^{\star} \; N^{\mathcal{B}_t}. \tag{1.18}$$

La matrice $\mathbb{A}^{\mathcal{B}_t}$ est la même que celle utilisée pour l'agrégation, étant donné qu'elle est l'opération inverse.

Nous pouvons voir sur la figure 1.7c, la désagrégation de la boîte rouge.

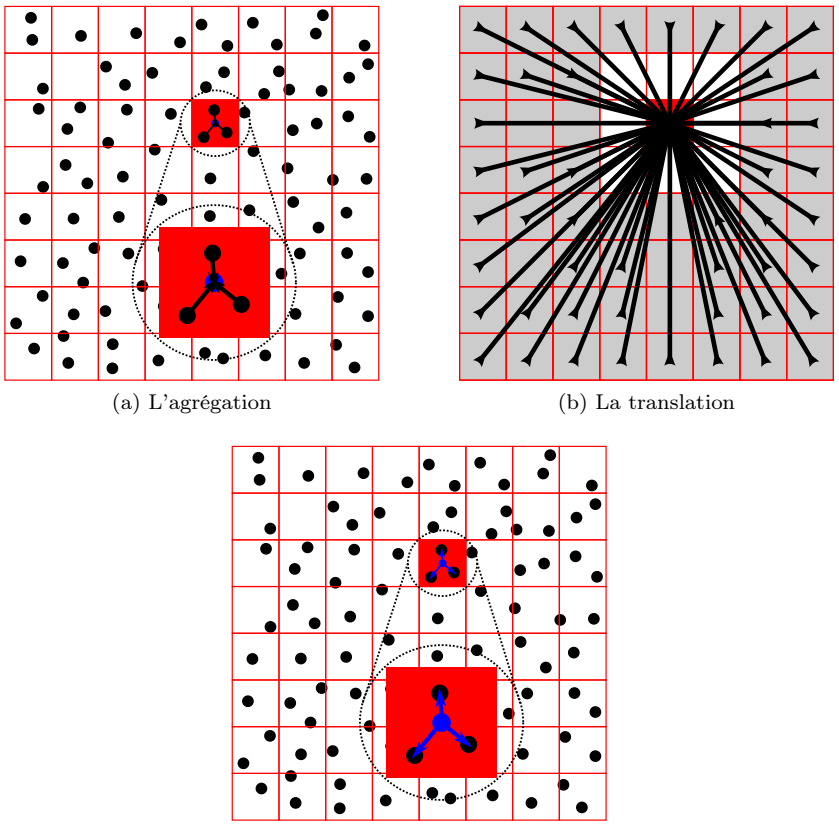
Finalement, pour récupérer le vecteur sur tous les points de la partition, d'après (1.10) et (1.12), nous faisons un relèvement (opération inverse de la projection) et nous obtenons v :

$$v = \sum_{\mathcal{B}_s \in \mathcal{P}} v^{\mathcal{B}_s}. \tag{1.19}$$

1.2.2.4 Performances

La translation peut être une étape très consommatrice en temps. La raison est que le nombre de translations est important. En effet, pour chaque boîte, il faut effectuer une translation avec chacun de ses non-voisins. Cela représente environ le carré du nombre de boîtes.

Comme nous pouvons le voir à la figure 1.8, pour diminuer le nombre de translations, il suffit d'augmenter la taille des boîtes. En contrepartie, le nombre de calculs directs augmente. Il faut donc trouver un équilibre entre le nombre de translations et le nombre de calculs directs.



(a) L'agrégation (b) La translation

(c) La désagrégation

FIGURE 1.7 – Les étapes de la méthode multipôle

Si les n points du nuage sont bien répartis sur une surface et si ces points sont distants en moyenne d'une fraction de la longueur d'onde, nous sommes conduits à prendre le nombre de boîtes, $N^{\mathcal{B}}$, proportionnel à $\sqrt{n}$, voir [75]. Dans ce cas, le nombre de directions est proportionnel à n et chaque étape (agrégation, translation et désagrégation, et calculs directs) a une complexité équivalente en $O(n\sqrt{n})$.

La complexité de la méthode multipôle simple-niveau est très intéressante par rapport à la complexité du calcul direct, comme le montre la figure 1.9.

1.2.3 Méthode multipôle multi-niveau

L'algorithme multi-niveau a été développé par Song et Chew [71] en 1995. C'est une amélioration de la méthode simple-niveau que nous venons de voir.

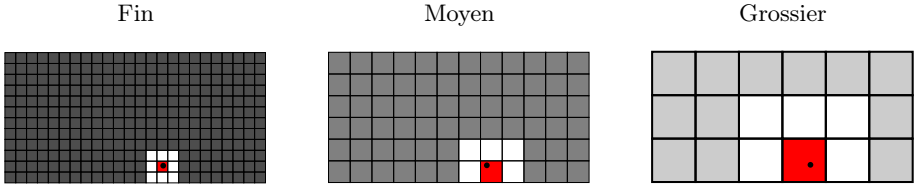


FIGURE 1.8 – Translations vers les boîtes grises, de la boîte avec le point noir, suivant trois partitionnement. Les boîtes blanches sont les calculs directs

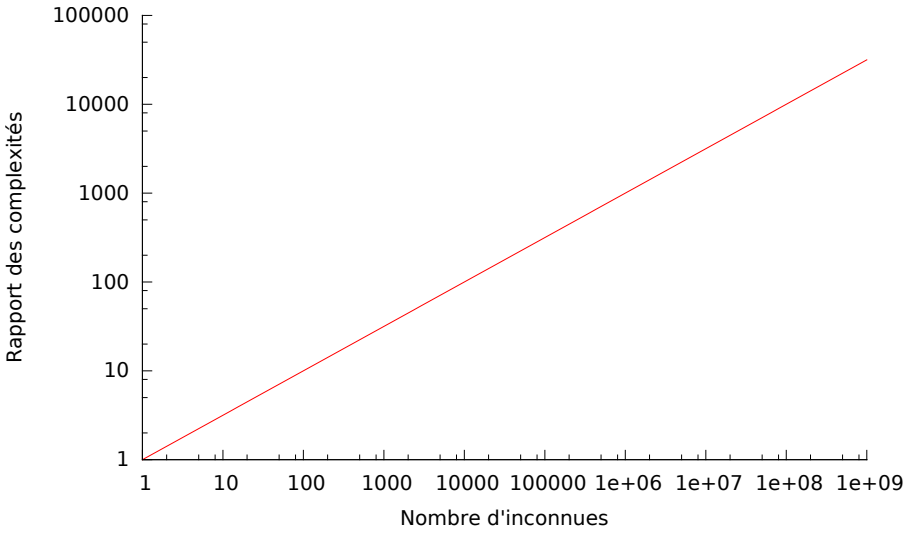


FIGURE 1.9 – Rapport de la complexité de la méthode directe par rapport à la complexité de la méthode simple-niveau

1.2.3.1 Principe

L'algorithme multi-niveau permet de limiter le nombre de translations tout en minimisant le nombre de calculs directs.

Pour cela, il faut considérer une interaction entre deux boîtes au niveau le plus grossier possible – les deux boîtes ne doivent pas être voisines. Cela implique qu'il n'y ait pas un seul partitionnement mais plusieurs partitionnements imbriqués. Ainsi, les translations sont effectuées là où elles concernent le nombre minimal de boîtes. Sur la figure 1.10, les translations concernant le point noir sont faites sur 3 niveaux. Les boîtes sont translatées seulement vers les boîtes voisines éloignées, c'est-à-dire les boîtes qui sont enfants des voisins du parent de la boîte rouge, mais non voisines de cette boîte rouge (par enfant, nous signifions les boîtes contenues dans la boîte considérée). L'explication est simple : les boîtes non-voisines et donc également les enfants, sont traitées par le parent. Finalement, le point reçoit la contribution de chaque boîte exceptée des voisins de la boîte rouge, comme dans la méthode multipôle simple niveau.

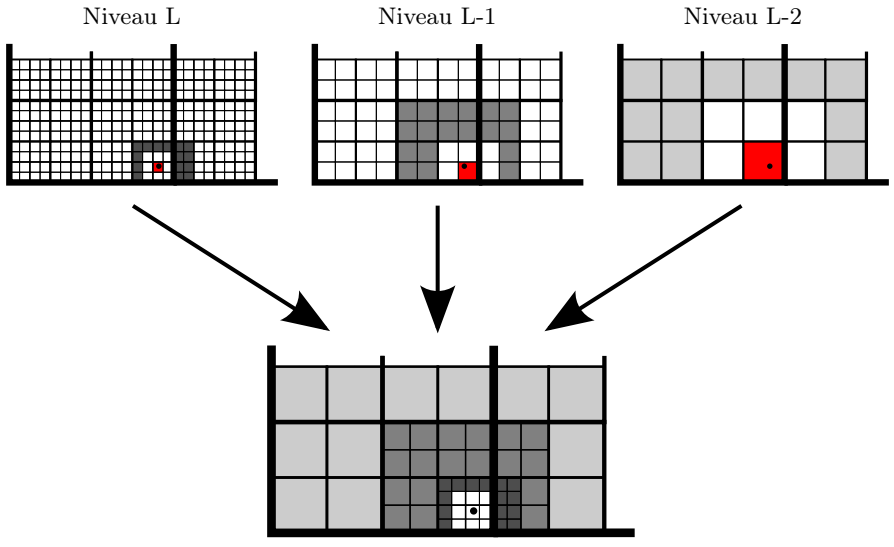


FIGURE 1.10 – Les translations dans la ML-FMM

1.2.3.2 Point de vue géométrique

L'apport de l'algorithme multi-niveau peut être abordé géométriquement comme montré sur la figure 1.11.

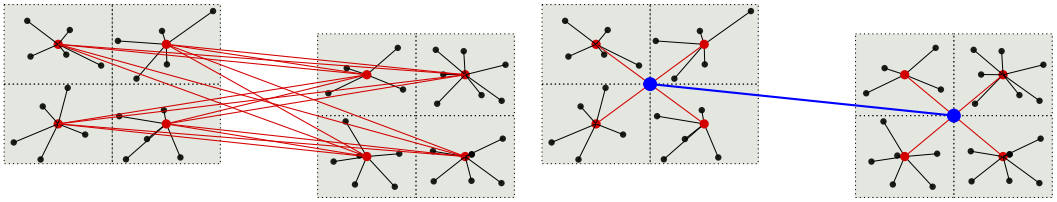


FIGURE 1.11 – Simple-niveau versus multi-niveau

Ce principe d'interactions est très courant dans les systèmes de télécommunications que nous connaissons. Prenons l'exemple d'Internet. Quand nous voulons accéder à un site web, nous ne faisons pas la connexion directement, nous passons par des intermédiaires. Il serait impensable d'avoir un lien direct vers chaque site web existant. Pour limiter les connexions, nous passons d'abord par notre fournisseur d'accès à Internet qui connecte plusieurs de ses abonnés d'une zone ensemble. Puis, nous passons par des routeurs qui connectent plusieurs zones. Ensuite, par exemple nous pouvons arriver sur le réseau de l'université qui héberge le site que nous voulons consulter et descendre dans les sous-réseaux de l'université jusqu'à atteindre la machine fournissant le site web. Nous aurions aussi pu prendre comme analogie n'importe quel système de réseau de communication tel le réseau routier, ou la distribution du courrier.

1.2.3.3 Partitionnement

Pour réduire le nombre total de translations, la méthode multipôle multi-niveau (MLFMM) utilise plusieurs niveaux de partitions imbriquées. La première étape de l'algorithme est donc de former des boîtes dans lesquelles nous calculerons des champs équivalents. Pour cela, nous allons placer l'objet à étudier dans un cube, plus précisément les points de quadrature des faces de l'objet.

L'étape suivante est une succession de découpages : le cube va être scindé en huit cubes de manière répétée, jusqu'à ce que la taille des cubes ait atteint une certaine limite. Nous pouvons voir à la figure 1.12 comment le découpage a été effectué. Seuls les cubes possédant au moins un point de quadrature sont conservés.

Le premier niveau, appelé niveau le plus haut et noté 1, est la partition faite à partir de la boîte englobant l'objet. Le dernier niveau est appelé niveau le plus bas et est noté $\pounds$. Nous appelons *parent* d'une boîte d'un niveau, la boîte dans le niveau supérieur qui inclut cette boîte. Nous définissons de manière réciproque les *enfants* d'une boîte.

Par conséquent, nous avons la relation sur la taille des boîtes d'un niveau :

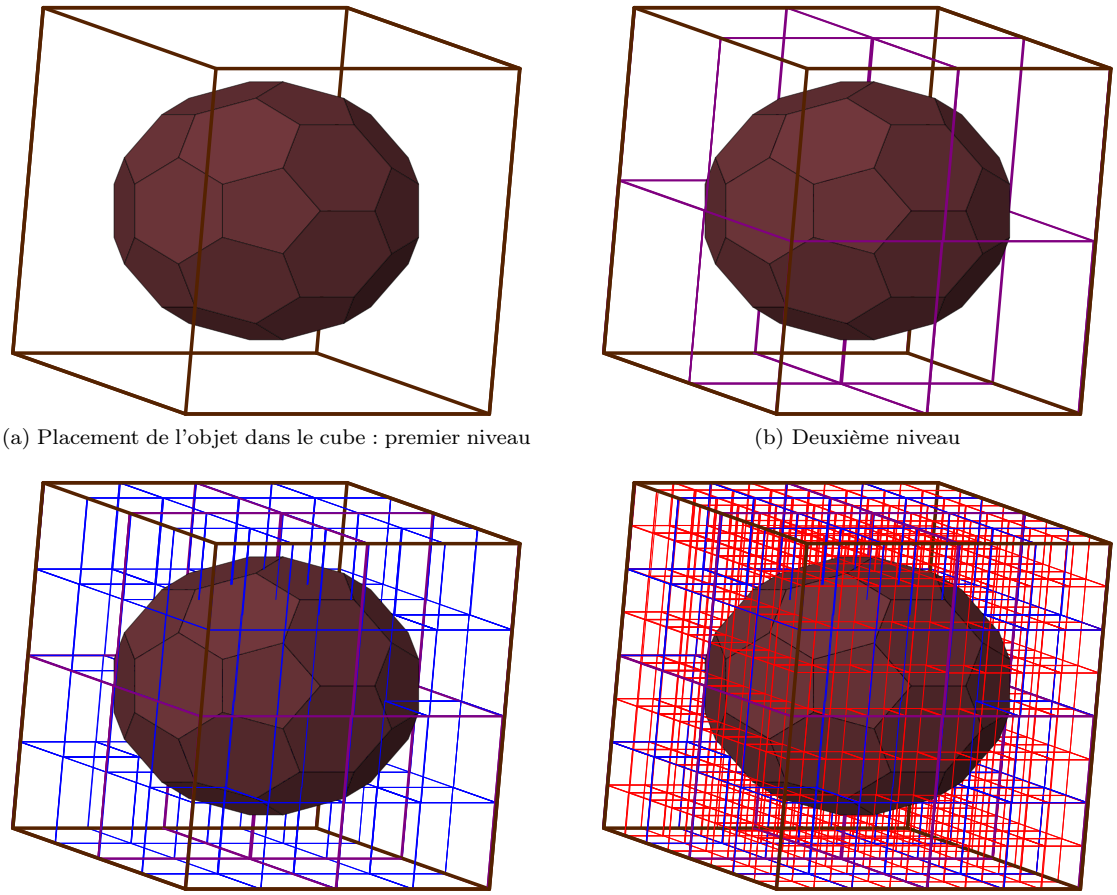
$$d_\ell = \frac{d_1}{2^\ell},$$

avec d_1 la taille du cube de départ et $\ell = 1, ..., \pounds$ le niveau de la boîte.

1.2.3.4 Interpolateur de la sphère unité

Comme nous l'avons vu à la section 1.2.1.4, le choix du nombre de multipôles dépend de la taille de la boîte. Or, dans le cas de la méthode multi-niveau, la taille de la boîte d_ℓ et donc le nombre de multipôles L_ℓ sont fonctions du niveau. Par conséquent, la taille des champs $F^{\mathcal{B}}$ dépend également du niveau. Nous devons alors avoir recours à des interpolateurs de la sphère unité pour passer d'une quadrature d'un niveau à celle du niveau supérieur ou inférieur.

L'algorithme d'interpolation le plus utilisé est l'algorithme d'Alpert [47]. Il consiste à interpoler en θ les modes de Fourier en φ. Nous ne justifierons pas ici cette méthode, nous contentant d'en donner l'algorithme, en considérant le passage entre un niveau (1) et un niveau (2).



(a) Placement de l'objet dans le cube : premier niveau

(b) Deuxième niveau

(c) Troisième niveau

(d) Dernier niveau

FIGURE 1.12 – Découpage successif de l'objet en cubes

1. Pour $i = 1, .., L^{(1)} + 1$, nous évaluons les coefficients de Fourier $\hat{F}_m(\theta_i^{(1)})$, $|m| \leq L^{(1)} + 1$ via une transformée de Fourier rapide, FFT, de longueur $2L^{(1)} + 1$ sur $F(\theta_i^{(1)}, \varphi_j^{(1)})$.
2. Pour $m = -L^{(1)}, L^{(1)}$, nous calculons les interpolées en θ des modes via

$$\hat{F}'_m(\theta_i^{(2)}) = \sum_{k=1}^{L^{(1)}+1} \mathbb{M}_m^{L^{(1)}+1}(\theta_i^{(2)}, \theta_k^{(1)})\, F_m(\theta_k^{(1)}),\ i = 1, \ldots, L^{(2)} + 1$$

3. Pour $i = 1, \ldots, L^{(2)} + 1$, nous complétons $\hat{F}'_m(\theta_i^{(2)})$ par des 0 pour les modes m tels que $L^{(1)} < |m| \leq L^{(2)}$.
4. Pour tout $i = 1, \ldots, L^{(2)}$, calcul de $F(\theta_i^{(2]}, \varphi_j^{(2)})$ par une `FFT` inverse de longueur $2L^{(2)} + 1$ à partir des $\hat{F}'_m(\theta_i^{(2)})$.

Les matrices $\mathbb{M}_m^L(\theta_i^{(2)}, \theta_k^{(1)})$ ont une forme assez particulière qui permet un stockage avantageux

$$\mathbb{M}_m^L(\theta_i^{(2)}, \theta_k^{(1)}) = \sqrt{\frac{(L+1)^2 - m^2}{4(L+1)^2 - 1}}\, \frac{P_{L+1}^m(\theta_i^{(2)}) P_L^m(\theta_k^{(1)}) - P_{L+1}^m(\theta_k^{(1)}) P_L^m(\theta_i^{(2)})}{\cos(\theta_i^{(2)}) - \cos(\theta_k^{(1)})}$$

où les $P_L^m(x)$ sont les *Fonctions de Legendre associées* convenablement normalisées [25, 47].

Le coût de cette méthode est celui induit par $L^{(1)}+1$ transformées de Fourier de taille $2L^{(1)}+1$, $L^{(2)}+1$ transformées de Fourier de taille $2L^{(2)}+1$, et $2L^{(1)}+1$ produits matrice vecteur de taille $L^{(1)}+1 \times L^{(2)}+1$. Cette dernière opération peut encore être accélérée en appliquant un algorithme

de compression ou une méthode type multipôle sur la matrice des inverses des différences de cosinus [4].

Comme nous l'avons dit, ces formules d'interpolation sont utiles pour factoriser encore les calculs. Supposons que nous manipulons des boîtes de petite taille (d_1) et de grande taille ($d_2 = 2d_1$, pour fixer les idées). Soit $\mathcal{B}$ une boîte de grande taille, composée de boîtes plus petites, par exemple $b_1, b_2, .., b_4$. Si c_q est le centre des b_q, et c le centre de $\mathcal{B}$, nous observons que nous avons la formule élémentaire :

$$e^{ik(x_i-c)\cdot\hat{s}} = e^{ik(c_q-c)\cdot\hat{s}}\, e^{ik(x_i-c_q)\cdot\hat{s}}.$$

Maintenant, si les points de quadratures (les $\hat{s}_p^{(1)}$ et les $\hat{s}_p^{(2)}$) étaient les mêmes, nous pourrions ainsi écrire (avec un petit abus de notation) :

$$F^{\mathcal{B}} = \sum_{q=1}^{4} \mathbb{E}^{\mathcal{B}-b_q}\, F^{b_q}$$

avec $\mathbb{E}^{\mathcal{B}-b_q}$ diagonale :

$$\mathbb{E}^{\mathcal{B}-b_q}_{p,q} = \delta_p^q\, e^{ik(c_q-c)\cdot\hat{s}_p}. \tag{1.20}$$

Calculer $\mathbb{A}^{\mathcal{B}}U$ reviendrait à récupérer les champs associés aux petits boîtes contenues dans $\mathcal{B}$, de les multiplier par un facteur de décalage, direction par direction, puis de sommer sur les quatre contributions.

Mais, comme nous l'avons vu, le nombre de directions D dépend de la taille d de la boîte. Les $\hat{s}_p^{(1)}$ et $\hat{s}_p^{(2)}$ ne sont donc pas les mêmes et nous ne pourrons utiliser notre formule que si nous l'interpolons. Si nous notons $\mathbb{I}^{(2,1)}$ l'interpolateur décrit ci-dessus [2], cela revient à nous servir de

$$F^{\mathcal{B}} \simeq \mathbb{I}^{(2,1)} \left(\sum_{q=1}^{4} \mathbb{E}^{\mathcal{B}-b_q}\, F^{b_q} \right)$$

ou

$$F^{\mathcal{B}} \simeq \sum_{q=1}^{4} \mathbb{E}^{\mathcal{B}-b_q} \left(\mathbb{I}^{(2,1)} F^{b_q} \right),$$

selon que nous interpolons avant de décaler ou l'inverse. Dans un cas $\mathbb{E}^{\mathcal{B}-b_q}$ utilise la quadrature la plus pauvre, dans l'autre, la plus riche. Par ce tour de passe-passe, on évite de calculer toutes les interactions directement en utilisant les champs des grosses boîtes et en leur appliquant des translations adaptées. C'est dans cette factorisation supplémentaire des calculs qu'on obtient un gain sur la complexité lorsque l'on passe de l'algorithme multipôle dit simple niveau (toutes les boîtes de même taille) à l'algorithme multi-niveau (plusieurs tailles de boîtes).

Remarque Lorsque nous transposons-conjuguons la première approximation de $\mathbb{A}$

$$\left(F^{\mathcal{B}}\right)^* \simeq \left(\sum_{q=1}^{4} \left(F^{b_q}\right)^* \overline{\mathbb{E}^{\mathcal{B}-b_q}} \right) {}^t\mathbb{I}^{(2,1)},$$

nous voyons apparaître la transposée de l'interpolateur : c'est ce que nous dénommerons un *anterpolateur*. Celui-ci intervient de la même manière pour éviter de sommer directement les contributions des grosses boîtes.

2. normalisé en ligne par l'inverse de la racine carré du poids de la première quadrature et en colonne par la racine carrée du poids de la seconde, dans le but d'enlever puis de remettre le poids dans la définition (1.9).

1.2.3.5 Algorithme

L'algorithme multi-niveau diffère de l'algorithme simple-niveau de part les changements de niveaux. Nous avons donc deux étapes supplémentaires que sont la montée et la descente. Pour l'algorithme multi-niveau, l'étape de translation est remplacée par une boucle sur les niveaux de trois étapes : la montée, la translation et la descente. Nous définissons $\mathcal{B}^l$, une boîte du niveau l, et $\mathcal{P}^l$, la partition du niveau l. Le plus haut niveau est le niveau 1 et le plus bas, le niveau L. L'algorithme a cinq types d'opérations.

Agrégation Elle est identique à l'agrégation présentée pour la version simple-niveau. Nous rappelons la formule pour simplifier la lecture :

$$F^{\mathcal{B}_s^{(\pounds)}} = \mathbb{A}^{\mathcal{B}_s^{(\pounds)}} u^{\mathcal{B}_s^{(\pounds)}}, \quad \forall \mathcal{B}_s^{(\pounds)} \in \mathcal{P}^{(\pounds)} \tag{1.21}$$

Le vecteur $u^{\mathcal{B}_s^{(\pounds)}}$ est le projeté sur la boîte $\mathcal{B}_s^{(\pounds)}$ du vecteur u d'entrée. La matrice $\mathbb{A}^{\mathcal{B}_s^{(\pounds)}}$ a autant de colonnes qu'il y a de points dans la boîte, et autant de lignes qu'il y a de directions ($D^{(\pounds)}$). $F^{\mathcal{B}_s^{(\pounds)}}$ est un $D^{(\pounds)}$ vecteur, appelé le vecteur associé à la boîte $\mathcal{B}_s^{(\pounds)}$.

La matrice $\mathbb{A}^{\mathcal{B}_s^{(\pounds)}}$ est calculée avec la formule suivante :

$$\mathbb{A}_{i,j}^{\mathcal{B}_s^{(\pounds)}} = \mathrm{e}^{ik\overrightarrow{x_i c_{\mathcal{B}_s^{(\pounds)}}}\cdot\vec{s_j}}. \tag{1.22}$$

Montée La méthode multi-niveau implique de connaître les vecteurs des boîtes de chaque niveau.

Nous rappelons que D, le nombre de directions et donc la taille du vecteur agrégé F, dépend de la taille des boîtes et par conséquent du niveau. Il en résulte que la taille $D^{(\ell)}$ du vecteur d'un enfant est plus petite que $D^{(\ell-1)}$, la taille du vecteur de son parent. Pour passer à un niveau parent, il est alors nécessaire d'interpoler les vecteurs des enfants pour avoir la bonne dimension. Cette étape est réalisée grâce à une multiplication par la matrice d'interpolation $\mathbb{I}^{\ell,\ell-1}$.

Le vecteur d'une boîte est calculé en agrégeant les vecteurs de ses boîtes enfants. Cela correspond à la somme de tous les vecteurs enfants interpolés, multipliés par une matrice de décalage vers le centre de la boîte parent notée $\mathbb{E}$. Pour des soucis de performance, nous faisons d'abord le décalage et la somme pour n'avoir plus qu'une seule interpolation à réaliser. Il y a une légère perte de précision mais tout à fait acceptable [74].

Pour tout $\mathcal{B}_s^{(\ell)} \in \mathcal{P}^{(\ell)},\ N-1 \leq \ell \leq 3$,

$$F^{\mathcal{B}_s^n} = \mathbb{I}^{\ell,\ell-1} \sum_{\mathcal{B}_s^{(\ell-1)} \subset \mathcal{B}_s^{(\ell)}} \mathbb{E}^{\mathcal{B}_s^{(\ell)} - \mathcal{B}_s^{(\ell-1)}} F^{\mathcal{B}_s^{(\ell-1)}} \tag{1.23}$$

où :

— $\mathbb{E}^{\mathcal{B}^{(\ell)}-\mathcal{B}^{(\ell-1)}}$ est une matrice diagonale de taille $D^{(\ell-1)} \times D^{(\ell-1)}$, appelée matrice de décalage, est définie par

$$\mathbb{E}_{i,i}^{\mathcal{B}_s^{(\ell)} - \mathcal{B}^{(\ell-1)}} = \mathrm{e}^{\mathrm{i}k\vec{s_i}\cdot\overrightarrow{M_{\mathcal{B}^{(\ell)}} M_{\mathcal{B}^{(\ell-1)}}}} \sqrt{\varpi_i}; \tag{1.24}$$

— $\mathbb{I}^{\ell,\ell-1}$ est une $D^{(\ell)} \times D^{(\ell-1)}$ matrice définie comme nous l'avons vu à la section 1.2.3.4.

Remarque : il n'est pas utile de monter plus haut que le niveau 3 car au dessus, toutes les boîtes sont voisines entre elles et donc il n'y aura pas de translation à calculer.

Translation Ce qui change dans la translation de la méthode multi-niveau ce sont les participants. La somme ne se fait plus sur l'ensemble des non-voisins mais simplement sur l'ensemble des voisins éloignés de $\mathcal{B}_t^{(\ell)}$ notés $\mathcal{V}_{\text{far}}(\mathcal{B}_t^{(\ell)})$.

Pour tout $\mathcal{B}_s^{(\ell)} \in \mathcal{P}^{(\ell)},\ \pounds \leq \ell \leq 3$,

$$N_T^{\mathcal{B}_t^{(\ell)}} = \sum_{\mathcal{B}_s^{(\ell)} \in \mathcal{V}_{\text{far}}(\mathcal{B}_t^{(\ell)})} \mathbb{T}^{\mathcal{B}_t^{(\ell)} - \mathcal{B}_s^{(\ell)}} F^{\mathcal{B}_s^{(\ell)}}. \tag{1.25}$$

Descente La descente est l'opération inverse de la montée. Pour chaque enfant, Il s'agit de décaler le champ depuis le centre du parent vers le centre de l'enfant grâce à la transposée de la matrice conjuguée de $\mathbb{E}^{\mathcal{B}_t^{(\ell)},\mathcal{B}_t^{(\ell+1)}}$. Ensuite, il faut se ramener dans chaque enfant à un vecteur de taille $D^{(\ell+1)}$ depuis le vecteur décalé de taille $D^{(\ell)}$. Pour cela, nous avons l'étape d'anterpolation qui consiste à multiplier le champ décalé par la transposée du conjugué de la matrice $\mathbb{I}^{\ell+1,\ell}$.

$\mathcal{L}-1 \leq \ell \leq 3$,

$$N^{\mathcal{B}_t^{(\ell+1)}} = \begin{cases} N_T^{\mathcal{B}_t^{(\ell+1)}} & \text{if } l = 3 \\ N_T^{\mathcal{B}_t^{(\ell+1)}} + \left(\mathbb{E}^{\mathcal{B}_t^{(\ell+1)},\mathcal{B}_t^{(\ell)}}\right)^\star \left(\mathbb{I}^{\ell+1,\ell}\right)^\star N^{\mathcal{B}_t^\ell} & \text{if } \ell \in [\![4,\mathcal{L}]\!] \end{cases}. \tag{1.26}$$

Désagrégation Elle est la copie de la désagrégation de la version simple-niveau. Pour rappel :

$\forall \mathcal{B}^{(\mathcal{L})} \in \mathcal{P}^{(\mathcal{L})}$,

$$v_{\text{far}}^{\mathcal{B}_t^{(\mathcal{L})}} = \left(\mathbb{A}^{\mathcal{B}_t^{(\mathcal{L})}}\right)^\star \mathbb{N}^{\mathcal{B}_t^{(\mathcal{L})}}. \tag{1.27}$$

1.2.3.6 Performances

La méthode multi-niveau a une complexité en temps et en espace de $\mathrm{O}(n \log n)$. Vous trouverez des détails sur le calcul de ces complexités dans [71].

Pour nous rendre compte du gain en complexité de la méthode multi-niveau, nous avons tracé la courbe du ratio des deux complexités à la figure 1.13.

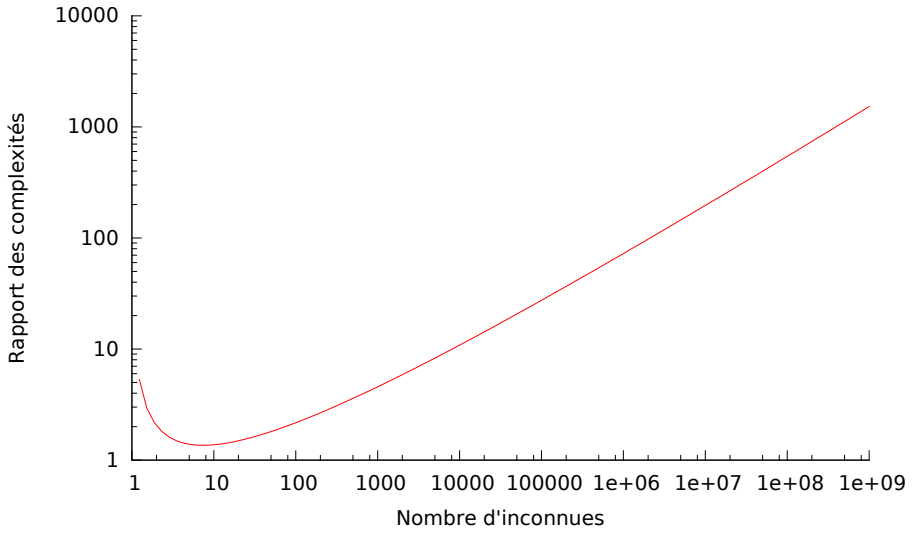


FIGURE 1.13 – Rapport de la complexité de l'algorithme simple-niveau par rapport à la complexité de l'algorithme multi-niveau

1.2.4 L'algorithme en pratique

L'aspect hiérarchique du découpage conduit naturellement à la représentation des boîtes à l'aide d'un arbre.

1.2.4.1 La construction de l'arbre

Chaque nœud est une boîte, et les enfants d'un nœud sont des sous-boîtes. Une boîte comportant huit sous-boîtes directes, cet arbre est appelé un *octree*.

Afin d'avoir des accès mémoire au maximum contigus, il est indispensable de choisir une numérotation la plus adaptée. Nous avons suivi le choix de Sylvand qui a étudié ce problème [74], et avons donc utilisé la numérotation hiérarchique, comme nous le verrons à la section 2.1.2.

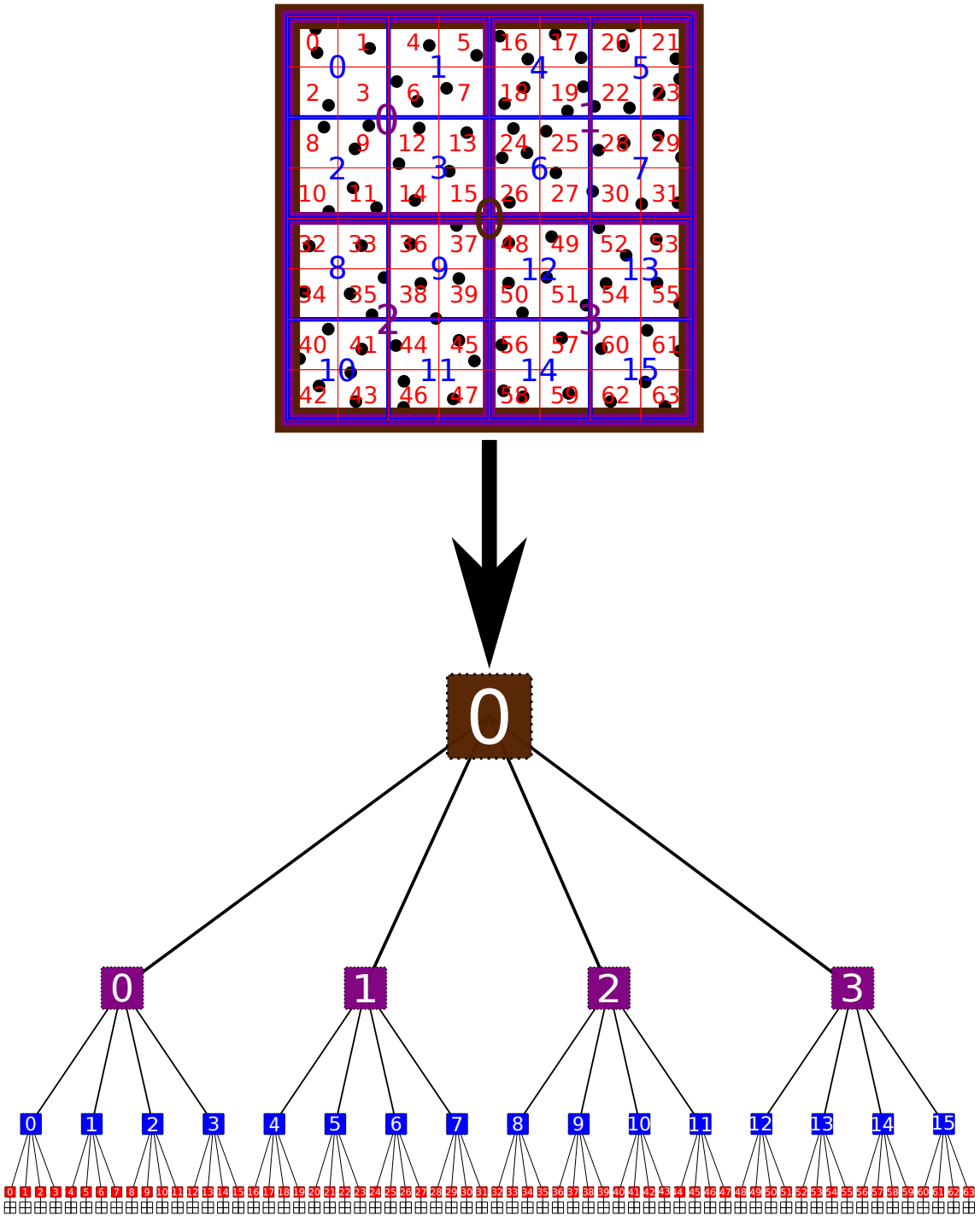


FIGURE 1.14 – La construction de l'octree

Sur la figure 1.14, nous pouvons voir comment depuis le partitionnement de notre espace, nous construisons notre octree. Dans les schémas que nous présentons, par souci de clarté, nous nous limiterons à nouveau à deux dimensions. Les arbres représentés seront donc des *quadtrees*.

À chaque nœud de l'arbre sont associés des vecteurs qui correspondent aux différents champs de la boîte. Les différentes opérations constituant la FMM, agissent sur les vecteurs associés aux boîtes.

Pour choisir la hauteur de l'arbre nous donnant les meilleures performances, nous prenons la plus petite taille de boîte d pour un nombre d'onde k telle que $kd > 0.3$.

1.2.4.2 Les opérations

Agrégation Elle consiste à initialiser les vecteurs F des feuilles de l'arbre. L'initialisation consiste en la multiplication du projeté du vecteur d'entrée sur la boîte, par la matrice $\mathbb{A}^{\mathcal{B}_s^L}$ dépendant de la boîte.

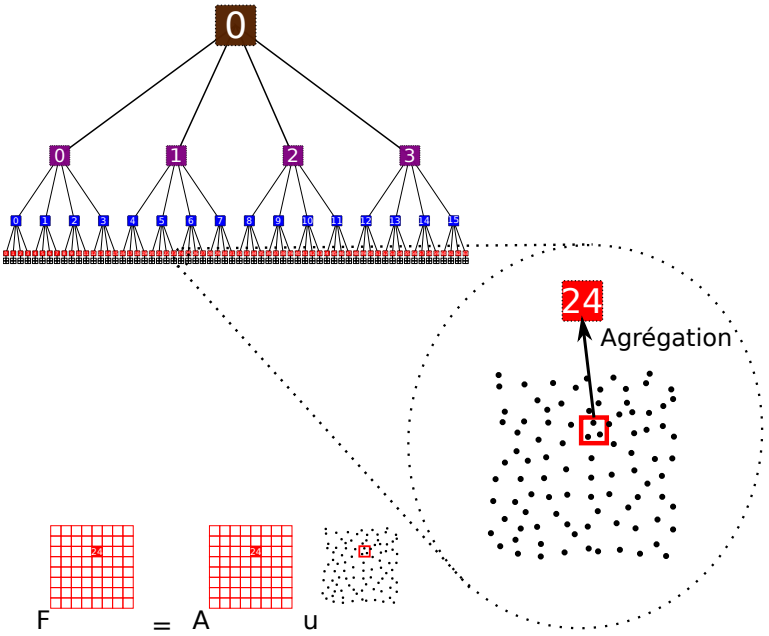


FIGURE 1.15 – L'agrégation

Montée Elle consiste à sommer les champs F des fils d'un nœud, puis à interpoler la somme, voir la figure 1.16. La numérotation choisie permet d'avoir les numéros des fils d'un nœud contigus, ce qui est intéressant pour les accès mémoire. Elle a lieu du niveau feuille jusqu'au niveau tout en haut. Le nombre de matrices de décalage $\mathbb{E}$ par niveau est égal au nombre possible d'enfants, c'est-à-dire 8.

Translation Elle a lieu pour chaque boîte de chaque niveau. Elle est la translation des vecteurs des voisins éloignés d'une boîte considérée. Pour le niveau au sommet, les boîtes translatées sont toutes celles qui ne sont pas voisines, voir la figure 1.17. La matrice de translation dépend du vecteur séparant une boîte de son voisin éloigné. Par conséquent, elle ne dépend simplement que du niveau considéré. Le nombre de matrices $\mathbb{T}$ correspond aux nombres de vecteurs différents possibles entre une boîte et un voisin éloigné. Avec le découpage en octree cubique, un voisin (éloigné ou proche) peut au maximum être à 3 d'écart de la boîte, pour chaque coordonnée. Nous avons donc 7^3 vecteurs possibles vers un voisin. À ce nombre, il faut enlever le nombre de vecteurs vers les voisins proches, pour obtenir le nombre de vecteurs vers un voisin éloigné. Le nombre de matrices différentes est alors de $7^3 - 3^3 = 316$ par niveau.

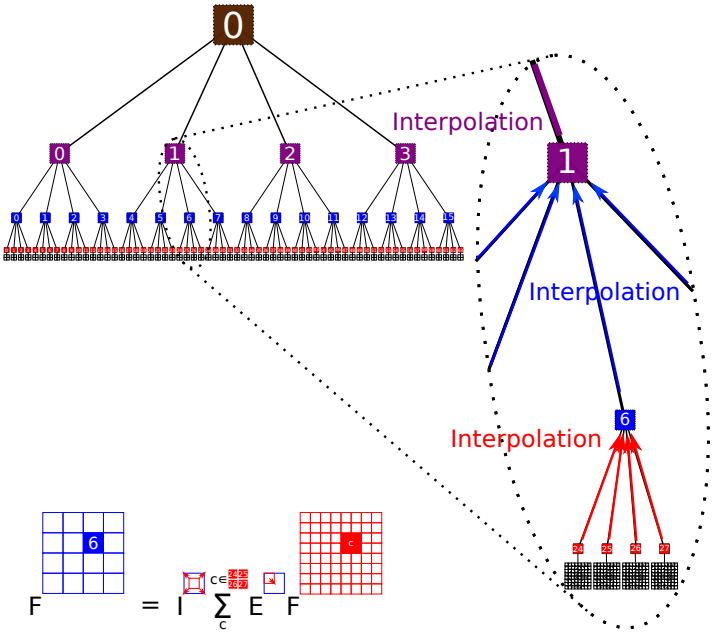


FIGURE 1.16 – Le décalage et l'interpolation

Descente Elle est l'inverse de la montée. Elle se produit depuis le niveau au sommet jusqu'au niveau feuille. Elle est représentée à la figure 1.18.

Désagrégation Comme pour la méthode simple-niveau, elle est l'opération inverse de l'agrégation, *cf* figure 1.19.

1.2.4.3 Un algorithme en deux phases

Pour résumer, la MLFMM est en deux phases. Une première phase de montée globale, jusqu'au niveau 3, composée d'agrégations et de montées locales (décalage et interpolation), voir la figure 1.20a. La deuxième est la descente globale qui consiste à partir du niveau haut et à descendre jusqu'en bas en faisant des descentes locales (anterpolation et décalage) auxquelles les contributions de autres boîtes sont ajoutées grâce à la translation, figure 1.20b.

Après avoir effectué une présentation de la méthode multipôle, nous allons étudier les évolutions des architectures des supercalculateurs.

1.3 Architectures actuelles des supercalculateurs

Un supercalculateur est un ordinateur qui a une puissance élevée par rapport aux simples ordinateurs grand public, et qui a pour seule vocation d'effectuer des tâches précises telles que le calcul scientifique.

Depuis leur apparition, les supercalculateurs ont connu de grandes évolutions. Du premier supercalculateur composé d'un seul processeur aux supercalculateurs actuels comportant plusieurs centaines de milliers de processeurs divers, il y a eu de nombreuses étapes et de grandes (r)évolutions dans leur conception.

1.3.1 La machine mono processeur

Les supercalculateurs sont apparus dans les années 60. Le premier supercalculateur reconnu est le CDC 6600, sorti en 1964. C'est un ordinateur composé d'un seul processeur effectuant les calculs - même s'il y a dix processeurs périphériques dédiés aux tâches d'entrées/sorties. La grande nouveauté de ce processeur est de pouvoir réaliser plusieurs instructions simultanées grâce à ces dix unités fonctionnelles parallèles [79]. C'est le premier processeur superscalaire.

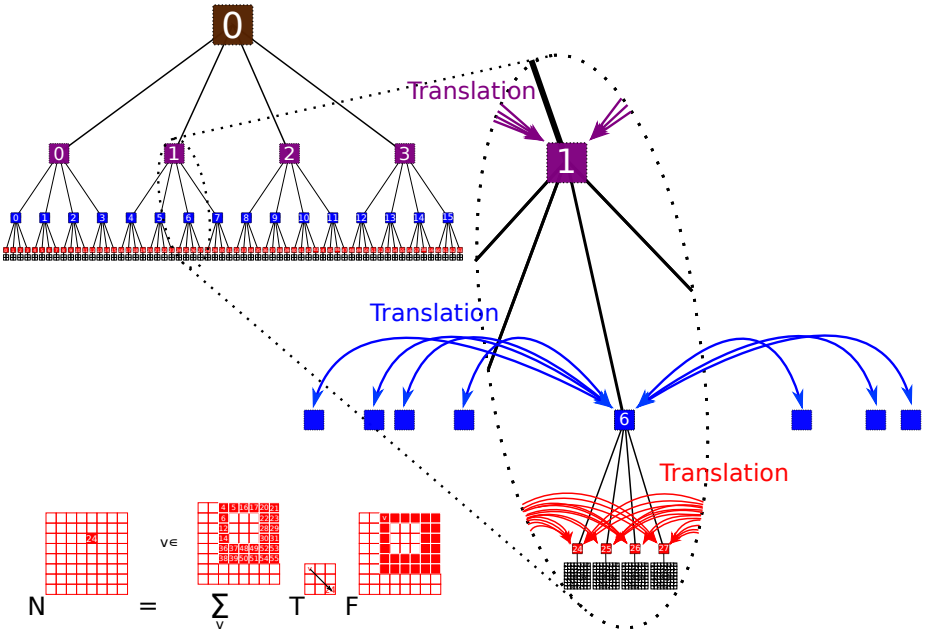


FIGURE 1.17 – La translation

1.3.1.1 Structure

Le processeur est le composant d'un ordinateur qui exécute les instructions des programmes. Il contient de nombreux transistors et est séparé en plusieurs unités ayant chacune un rôle précis. Dans cette présentation nous ne nous attarderons pas sur le fonctionnement des processeurs, pour davantage de détails voir [45].

Au départ, les processeurs sont directement reliés à la mémoire centrale, appelée RAM (Random Access Memory) pour mémoire à accès direct, à laquelle ils accèdent par l'intermédiaire de registres (petite mémoire interne très rapide), voir la figure 1.21a.

Dans les années 70, sont apparus les processeurs vectoriels. Ces processeurs disposent de registres vectoriels, permettant de charger plusieurs valeurs en une seule fois. Il est ensuite possible d'appliquer une seule et même instruction vectorielle sur les données chargées, ce qui permet d'accélérer le traitement des données. Ces processeurs sont dits processeurs SIMD pour *Single Instruction Multiple Data*. Nous retrouvons ces mécanismes dans les processeurs actuels, par exemple, avec les extensions vectorielles aux jeux d'instructions AVX et SSE4.

1.3.1.2 Mémoire cache

À l'époque des premiers supercalculateurs, la mémoire était bien plus rapide que le processeur mais c'était sans compter sur la loi de Moore énoncée en 1965 [54] et revue en 1975 [55]. Cette loi indique que le nombre de transistors sur une puce double tous les deux ans. Cette loi se vérifie quasiment depuis 1973, et elle se manifeste alors par le doublement de la fréquence. Au début des années 1990, l'augmentation continue de la fréquence a donc permis au processeur de devenir plus rapide que la mémoire qui n'a pas connu d'augmentation semblable à celle des processeurs, concernant sa bande passante. Cette dernière devient donc un frein à l'envolée des performances.

Heureusement, nous savons construire des mémoires bien plus rapides que celles constituant la mémoire centrale. Ces mémoires sont appelées SRAM pour *Static RAM* par opposition à DRAM, *Dynamic RAM*, composant la mémoire globale. Mais le gain de performances ne se fait pas sans sacrifice : le coût est élevé et la capacité de stockage est très réduite. Pour avoir un ordre d'idée, la capacité actuelle maximale pour les plus rapides de ces mémoires, est environ cent fois moins importante que celle de la DRAM. À cause de leur faible capacité, il est évident que ces mémoires ne peuvent remplacer la traditionnelle mémoire DRAM.

L'idée est donc de précharger les données sur la SRAM, appelée mémoire cache, pour pouvoir

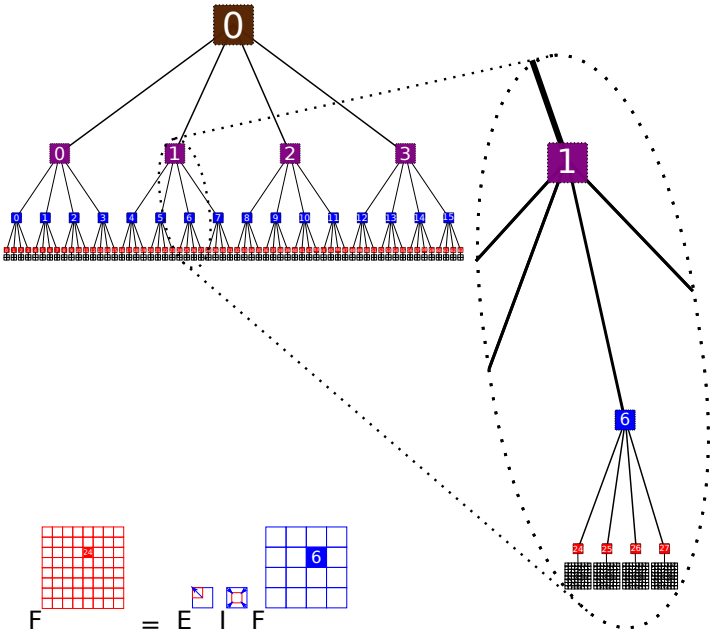


FIGURE 1.18 – L'anterpolation et le décalage

les fournir rapidement au processeur quand il en aura besoin. Le processeur ne devra plus attendre le temps de transfert depuis la mémoire centrale mais simplement depuis le cache, plus rapide. Ces mémoires sont donc placées entre le processeur et la mémoire centrale, figure 1.21b.

Il y a plusieurs niveaux de cache dans un processeur pour lisser la grande différence de vitesse entre le processeur et la mémoire centrale. Les niveaux de cache réalisent des compromis différents entre la latence et la capacité : les caches plus proches du processeur seront plus rapides et les caches plus proches de la mémoire centrale seront plus grands. De nos jours, les processeurs comportent 2 (L1, L2) ou 3 (L1, L2, L3) niveaux de cache de taille allant jusqu'à 512 ko pour le cache L1, 2 Mo pour le cache L2 et 20 Mo pour le cache L3.

Comme nous le verrons à la section 2.2.1.3, la difficulté dans la gestion de cache est de savoir quelles données vont être utilisées prochainement par le processeur.

Fonctionnement Les mémoires caches sont divisées en lignes mémoire toutes accessibles directement par le processeur. Contrairement aux échanges avec le processeur, les lectures/écritures depuis/sur la mémoire centrale ne peuvent se faire à l'unité. Ces échanges sont groupés par blocs de taille fixe, nommés lignes de cache. Une ligne de cache fait généralement entre 64 et 256 octets.

Pour l'unité de calcul demandant une donnée, l'utilisation du cache est transparente. Le cache intercepte les accès mémoire et agit suivant la configuration. Soit la donnée demandée est déjà présente dans la mémoire cache – on appelle cela un *cache hit* ou succès de cache – et il n'a plus qu'à la fournir. Soit la donnée est absente du cache – on appelle cela un *cache miss* ou défaut de cache – et il est alors nécessaire de lire/écrire depuis/vers la RAM.

La lecture de données est réalisée assez trivialement, le plus dur étant de savoir quelles lignes garder dans le cache quand celui-ci est plein. Nous ne détaillerons pas ici les différentes stratégies, pour plus de détails voir [32]. Mais pour des raisons de performances, il existe des mécanismes plus complexes pour l'écriture. Nous allons simplement et très brièvement rappeler qu'il existe le mécanisme de *write back* qui ne répercute une écriture dans la mémoire centrale qu'au moment où la ligne concernée est vidée du cache, ou encore le mécanisme dit de *write through* qui permet à une donnée d'être écrite directement dans la mémoire centrale. Il peut aussi y avoir un tampon permettant de stocker les écritures avant leur réalisation effective dans la mémoire. Comme il y a plusieurs niveaux de cache, se pose aussi le problème de cohérence de cache lors de l'écriture. En effet, une modification sur une donnée d'un niveau de cache doit être considérée dans les autres niveaux contenant également cette donnée. Nous arrêtons ici notre présentation succincte

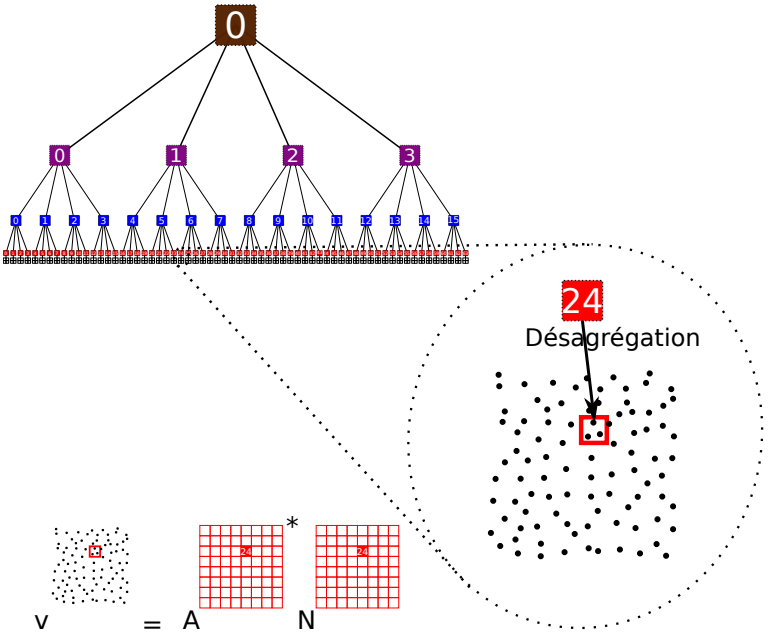


FIGURE 1.19 – La désagrégation

du fonctionnement, les caches n'étant pas le sujet de notre étude.

Localités Le cache prend tout son intérêt lors d'un succès de cache et n'apporte rien lors d'un défaut de cache. Il est donc important de minimiser les défauts de cache. Pour cela, les caches sont dépendants de deux principes : la localité temporelle et la localité spatiale.

La localité temporelle est le fait qu'une donnée risque d'être utilisée plusieurs fois dans un temps très court. Ce phénomène se produit souvent dans les programmes, typiquement dans des boucles. Mettre en cache les données d'une boucle peut donc être bénéfique. Le problème est que le cache a une taille limitée et ne peut pas garder beaucoup de données. Quand il est plein, des lignes doivent être supprimées et donc un nouvel accès aux données des lignes provoquera un défaut de cache. Les seules solutions sont d'augmenter la taille du cache, ou d'augmenter la localité temporelle de notre programme, voir la section 2.2.1.1.

De son côté, la localité spatiale concerne les données et leurs voisins. Quand une donnée est chargée dans le cache, c'est toute une ligne de cache qui est chargée. Si une donnée voisine est maintenant demandée par le processeur, elle risque fort de se trouver dans la même ligne de cache et donc de provoquer un succès de cache. C'est le cas par exemple lors d'un parcours séquentiel d'un tableau. Nous verrons à la section 2.2.1.3 comment améliorer la localité spatiale.

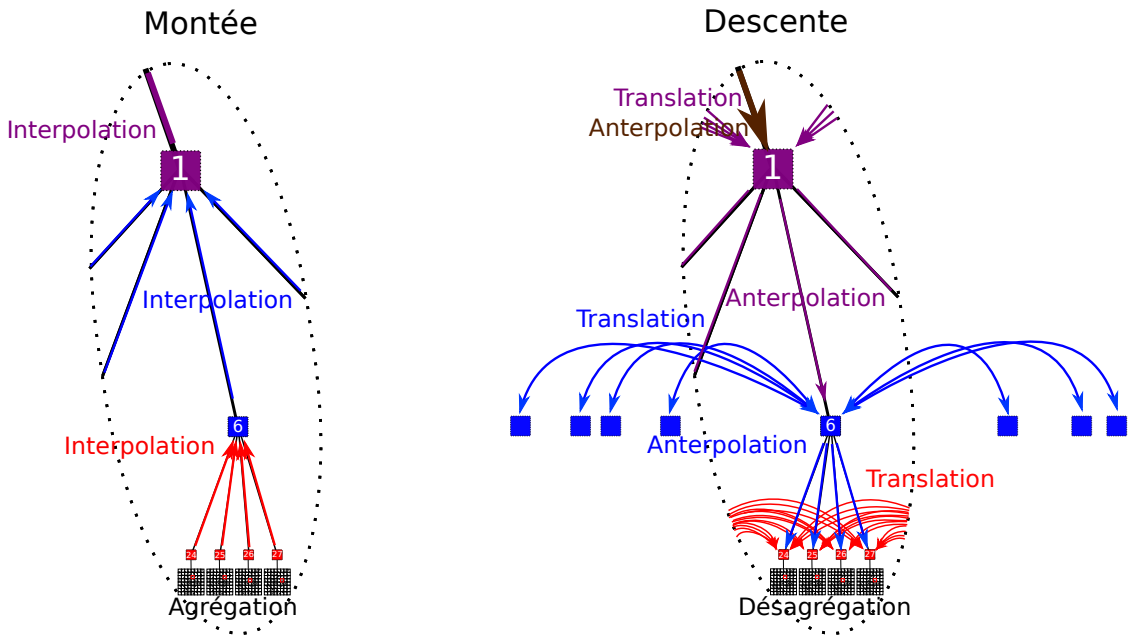
1.3.2 Les multiprocesseurs

La puissance d'un processeur est limitée par les capacités de gravure sur la puce. La seule solution pour augmenter la puissance des machines est d'agréger plusieurs processeurs ensemble.

1.3.2.1 Les multiprocesseurs symétriques

Pour de meilleure performances, l'agrégation est faite sur la même mémoire centrale, la distance étant une barrière à l'efficacité. Ce type d'architecture s'appelle architecture à multiprocesseur symétrique, pour *symmetric multiprocessing* abrégé en SMP. Le terme symétrique indique que les processeurs ont le même accès à la mémoire. En 1962, Burroughs introduit le D825, premier processeur SMP à voir le jour [6].

Le fait que plusieurs processeurs partagent la mémoire centrale impose certaines modifications dans la gestion des caches. En effet, comme c'était le cas entre deux niveaux de cache, il faut garder une cohérence des données entre les caches des différents processeurs. Si une donnée est dans le



(a) La montée globale

(b) La descente globale

FIGURE 1.20 – L'algorithme multipôle

cache de deux processeurs et qu'un des deux la modifie, le cache de l'autre processeur ne contiendra pas la nouvelle valeur de la donnée. Ce mécanisme de cohérence implique un surcoût.

Mais le principal problème avec cette approche c'est le partage de la bande passante de la mémoire centrale par les processeurs. L'accès à la mémoire devient alors un important goulet d'étranglement, ce qui est préjudiciable pour bon nombre d'applications. Il n'est donc pas possible d'agréger trop de processeurs sur la même mémoire sans agir au détriment des performances.

1.3.2.2 Les grappes

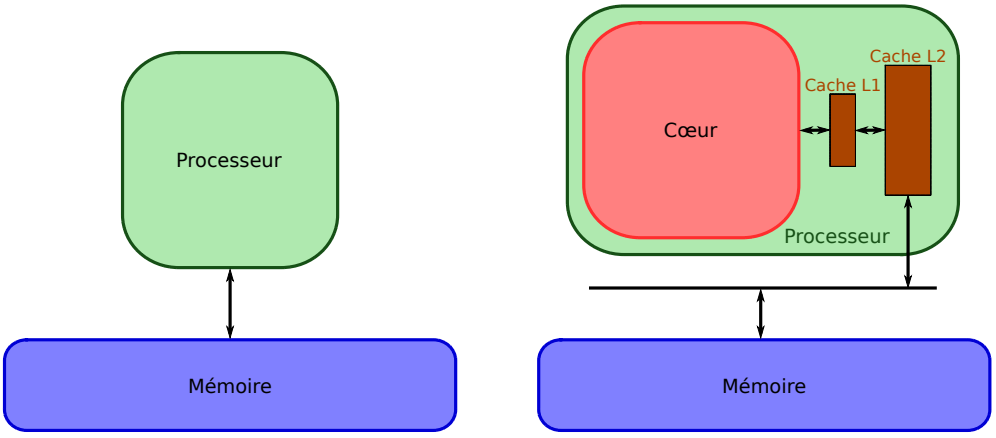
Pour pouvoir agréger davantage de processeurs, des machines sont reliées entre elles par des réseaux. Pour obtenir un système performant, les machines, appelées nœuds, doivent être physiquement proches et reliées entre elles par des réseaux à très haut débit et à faible latence, tels que Myrinet, Quadrics ou Infiniband. Ces ensembles de machines n'en formant plus qu'une seule sont appelés des grappes d'ordinateurs, en anglais *computer cluster*, ou grappes de calcul, voir la figure 1.23. Ces nœuds peuvent être composés de processeurs SMP pour obtenir de meilleures performances.

Les grappes de calcul s'opposent aux grilles de calcul par la distances entre les nœuds. En effet, les grilles sont l'agrégation de plusieurs ordinateurs hétérogènes, à l'échelle d'un territoire. Les latences entre les nœuds sont alors plus importantes mais les grilles sont moins onéreuses. En France, nous trouvons Grid5000, qui contient des ordinateurs répartis sur tout le territoire.

Les nœuds ont leur propre mémoire et le partage des données nécessite, de ce fait, le transfert de données de la mémoire d'un nœud vers celle d'un autre.

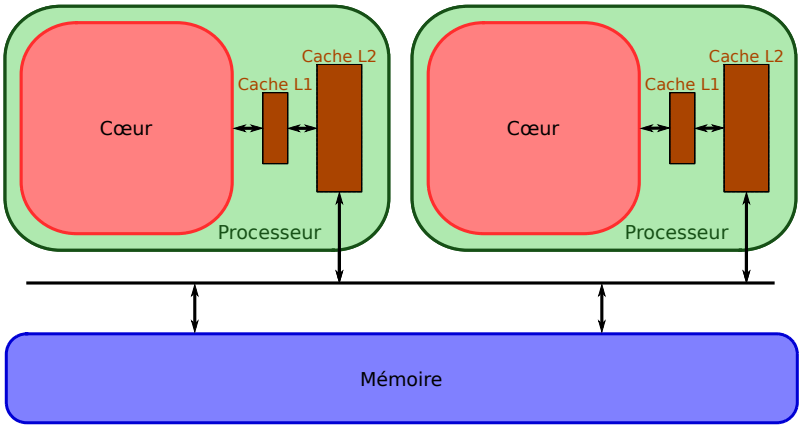
1.3.3 Les processeurs multicœurs

La loi de Moore, que nous avons vue précédemment, prévoit le doublement du nombre de transistors tous les 2 ans, et s'est manifestée par le doublement de la fréquence des processeurs jusqu'en 2005. À cette date, la finesse de gravure des processeurs permet de placer 250 millions de transistors sur une même puce et permet alors d'atteindre des fréquences de l'ordre de 3 GHz. Une telle fréquence provoque une très grande dissipation thermique – elle est proportionnelle au carré

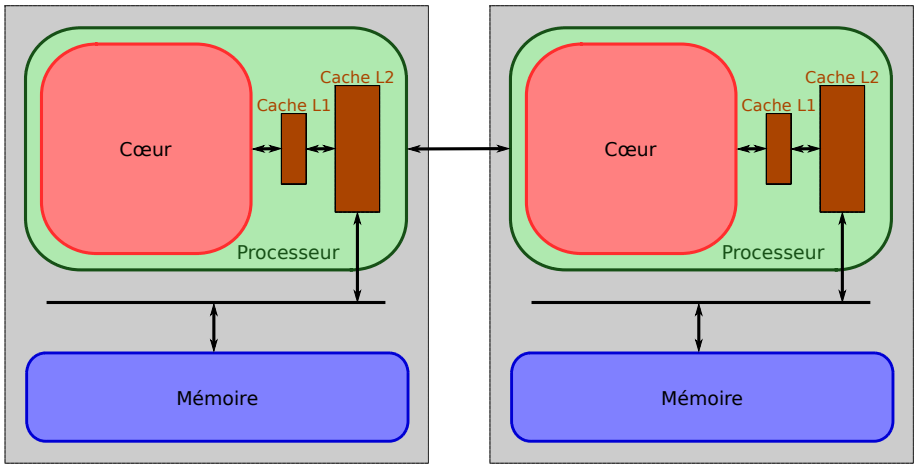


(a) Connexion directe à la mémoire (b) Utilisation de caches

FIGURE 1.21 – Processeur



(a) Multiprocesseur symétrique (SMP)



(b) Multiprocesseur avec mémoires séparées

FIGURE 1.22 – Multiprocesseurs

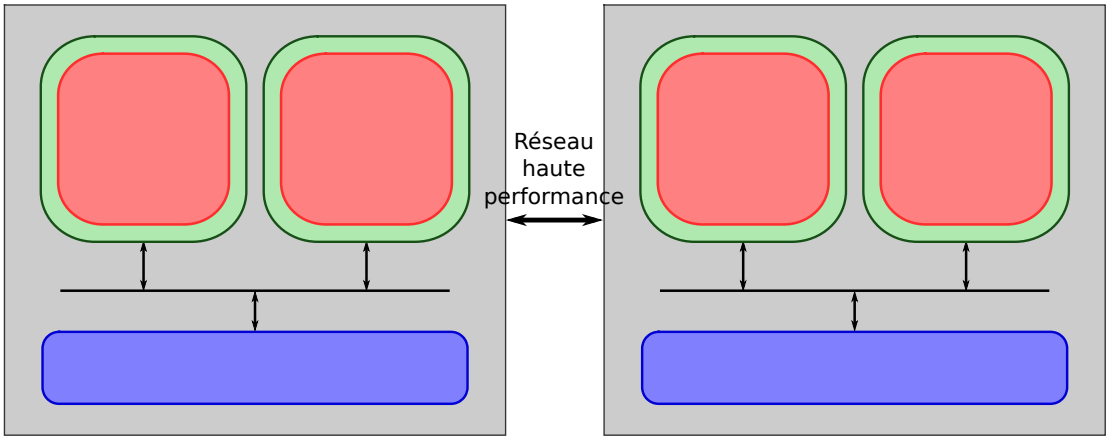


FIGURE 1.23 – Structure d'une grappe de calcul comportant deux nœuds de deux processeurs

de la fréquence – et donc des problèmes de surchauffe. Ce problème de surchauffe empêche d'utiliser l'espace gagné par l'amélioration constante de la finesse de gravure, pour la montée en fréquence. L'espace supplémentaire pourrait être utilisé pour augmenter la complexité du processeur : unité de calcul supplémentaire, unité de prédictions plus évoluée. Mais les processeurs sont déjà très complexes, et le gain apporté ne serait que faible, surtout en le comparant aux efforts fournis et aux coûts de production.

Pour se servir des transistors disponibles, les fabricants ont décidé de placer un cœur de calcul supplémentaire sur la puce du processeur. C'est le POWER4, fabriqué par IBM à partir de 2001, bien avant la fin de la montée en fréquence, qui est le premier processeur à comporter deux cœurs sur la même puce [77]. En 2005, Intel et AMD sortent leur premier processeur à deux cœurs, respectivement le Pentium D et l'Athlon 64 X2. Depuis, le nombre de cœurs augmente progressivement. Ce type de processeur est appelé processeur multicœur.

1.3.3.1 Structure

Comme pour les processeurs ne possédant qu'un seul cœur, les processeurs multicœurs sont reliés à la mémoire centrale par l'intermédiaire de caches.

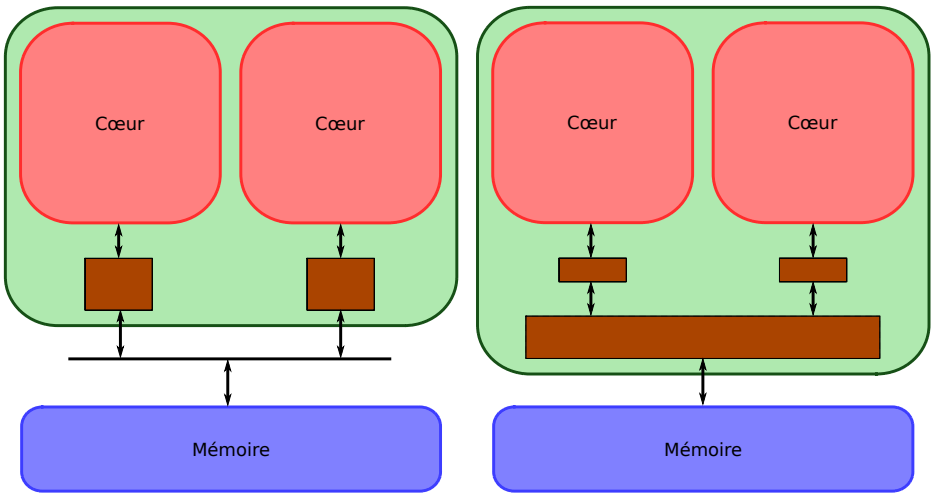
La première approche est identique à celle des multiprocesseurs symétriques, vus à la figure 1.22a, c'est-à-dire relier les caches de chacun des cœurs à la mémoire centrale, comme le montre la figure 1.24a, avec les mêmes problèmes de cohérence de cache et de limitation de bande passante.

Un inconvénient supplémentaire avec cette approche réside dans le fait que les caches sont plus petits, la place sur la puce étant limitée (le cache prend environ la moitié de la puce). Une deuxième approche est de partager un ou plusieurs niveaux de cache entre les cœurs, voir figure 1.24b. Cette approche peut conduire à une concurrence de place entre les cœurs au niveau du cache.

Départager ces deux approches dépend des besoins des programmes exécutés. En général, l'approche à partage de cache est privilégiée.

1.3.3.2 Vers les architectures hybrides

En construisant une grappe de machines multiprocesseurs multicœurs, nous obtenons ce que nous appelons une machine à architecture hybride. Cette dénomination fait référence à la présence de mémoire partagée entre des processeurs et de mémoire distribuée entre les nœuds. La figure 1.25 nous montre un exemple de machine hybride. Cette machine est composée de trois nœuds de quatre processeurs quadri-cœurs. Si le cœur A désire accéder à des données détenues dans B, il devra passer par le réseau. Pour des données de C, il aura besoin de passer par l'intermédiaire de deux autres processeurs. Pour des données de D, il utilisera un intermédiaire. Enfin pour des données situées en E, il aura un accès direct. Nous venons de voir les différents types d'accès du plus coûteux au moins coûteux. Cela nous montre que les coûts d'accès ne sont pas les mêmes suivant la position des



(a) Sans partage de cache (b) Avec partage de cache

FIGURE 1.24 – Processeurs multicœurs

données. C'est pour cette raison que ce type de machine est appelé machine NUMA (Non Uniform Memory Access).

D'autres inégalités d'accès à la mémoire peuvent s'ajouter dans les connexions réseau qui peuvent, par exemple, avoir une topologie en arbre.

1.3.3.3 Évaluer la puissance d'une machine

Pour évaluer un cœur de calcul, prendre la fréquence seule comme critère n'est pas valable. Il y a beaucoup d'autres mécanismes comme par exemple le jeu d'instructions ou le nombre d'étages de *pipeline*. Les caches entrent également en jeu. De plus, un processeur ne va pas toujours être le plus efficace sur tous les programmes suivant le type d'opération qu'il demande.

Néanmoins, pour avoir une idée assez précise, le nombre d'opérations flottantes par seconde est communément utilisé pour évaluer la puissance d'une ressource de calcul, et par extension, d'une machine. Il est noté flops, pour *Floating Point Operations Per Second*. Ce nombre est bien entendu théorique, et il est difficile, voire même impossible, à atteindre par une application. Pour le calculer, il faut effectuer le rapport entre le nombre d'opérations nécessaires et le temps écoulé.

Pour avoir un ordre d'idée, la figure 1.26 nous montre l'évolution au TOP500 [80] de la puissance des supercalculateurs. Le TOP500 est un classement semestriel référençant les 500 ordinateurs les plus puissants au monde.

1.3.4 L'utilisation des GPU dans le calcul scientifique

Les cartes graphiques sont responsables de l'affichage, elles produisent les images qui seront affichées à l'écran. Leur deuxième rôle est de produire le rendu d'images en 3D, très gourmand en calculs. Pour cela, elles ont besoin de processeurs spécifiques et rapides, notamment pour les jeux video. Ces processeurs sont appelés processeurs graphiques, GPU pour Graphics Processing Unit.

Avec le développement du marché du jeu vidéo, de grandes avancées sont faites dans les GPU, afin de parvenir à des rendus de plus en plus complexes qui nécessitent d'effectuer de lourds calculs vectoriels. Au début des années 2000, les GPU deviennent donc des processeurs très puissants spécialisés dans le calcul intensif et massivement parallèles. Leurs coûts de production sont réduits grâce à une forte demande. Ils offrent ainsi un grand nombre de flops par dollar [35]. Il faut ajouter à cela une faible consommation en Watts par flops comparés aux CPU.

Toutes ces caractéristiques font du GPU un bon candidat pour le calcul haute performance. En 2002, la communauté scientifique commence à s'y intéresser [78, 14, 44, 51] et le terme GPGPU apparaît pour *General-Purpose Processing on Graphics Processing Units* [40].

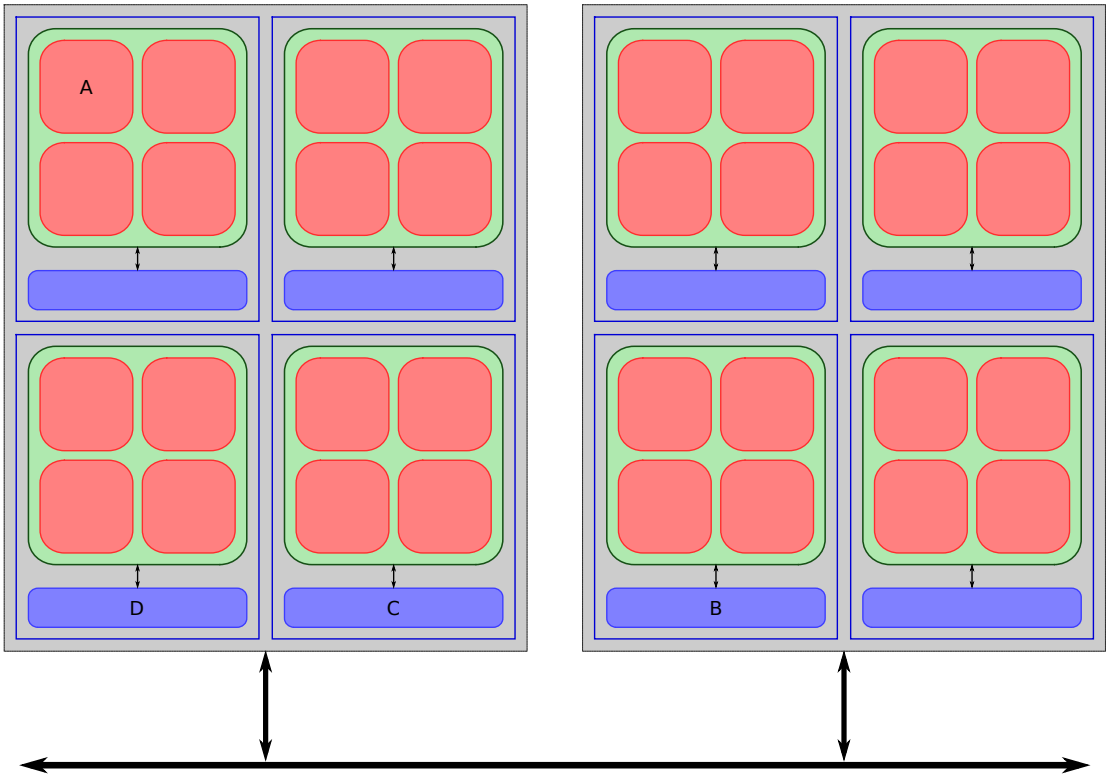


FIGURE 1.25 – Structure d'une machine hybride comportant deux nœuds de quatre processeurs quadri-cœurs

Dans notre document, nous ne parlerons que des GPU CUDA, car ils composent les machines cibles que nous utiliserons.

1.3.4.1 Comparaison avec les CPU

Aux figures 1.27 et 1.28, nous avons un comparatif du nombre d'opérations flottantes par seconde et de la bande passante de la mémoire entre des GPU et des CPU.

Comme nous l'avons évoqué, les GPU sont des processeurs massivement parallèles. Par exemple, nous pouvons citer les dernières cartes NVidia Tesla K20 qui possèdent 2500 cœurs. Cela s'explique par le fait que davantage de transistors sont réservés aux calculs plutôt qu'à la mise en cache des données et au contrôle des instructions [61]. Cette plus grande simplicité dans les circuits de contrôle provient du fait que les GPU sont des processeurs vectoriels, il est donc plus simple de gérer une multitude de cœurs qui réalisent les mêmes instructions. Ces propos sont illustrés par la figure 1.29.

1.3.4.2 Structure

Les GPU sont composés de plusieurs multiprocesseurs comportant plusieurs cœurs. Chaque multiprocesseur comporte une mémoire partagée, voir figure 1.30. Le GPU est relié à une mémoire globale, qui peut communiquer avec la mémoire du CPU, appelée hôte, afin de permettre des transferts. Cette connexion se fait par bus PCI express.

Le GPU est un processeur vectoriel, c'est-à-dire qu'il est capable de réaliser une instruction sur plusieurs données en même temps, en anglais Single Instruction Multiple Data (SIMD).

L'exécution d'une tâche se fait en ses données en une grille de blocs de threads. La parallélisation en threads correspond à une parallélisation de données. Chaque bloc est exécuté par un multiprocesseur. Cette exécution se fait en simultanée par groupe de threads dans le bloc, nommé *warp*. Comme nous sommes dans le cadre du SIMD, toutes les instructions réalisées en même temps par les cœurs des multiprocesseurs sont identiques. Le multiprocesseur exécute tous les warps de

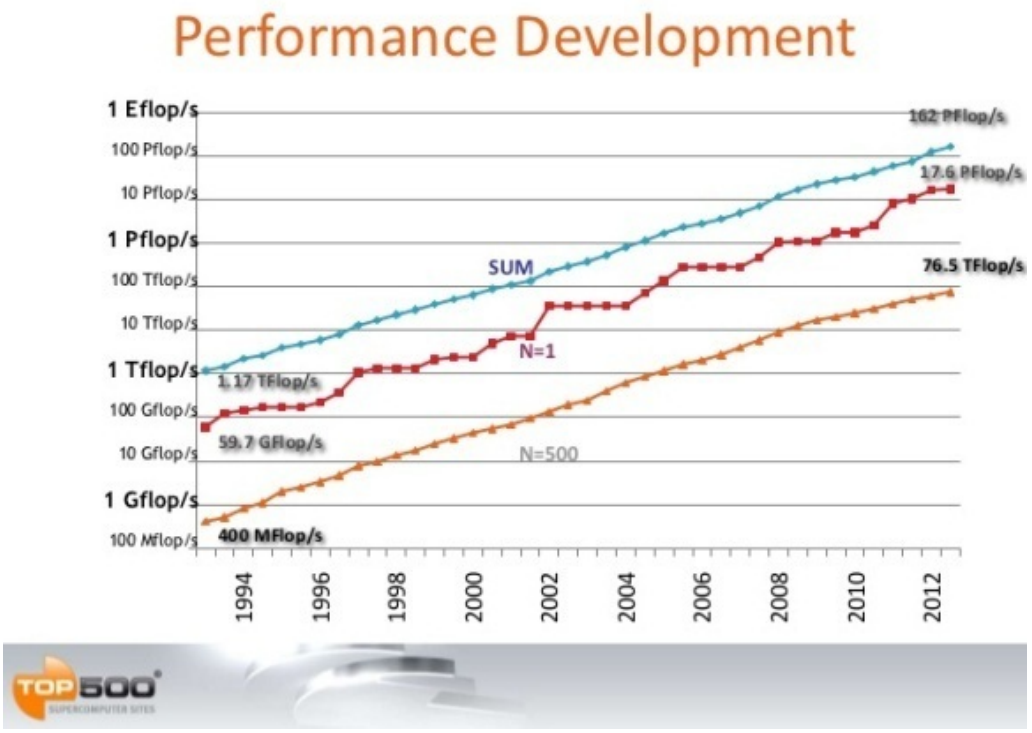


FIGURE 1.26 – Évolution de la puissance des supercalculateurs depuis 1994

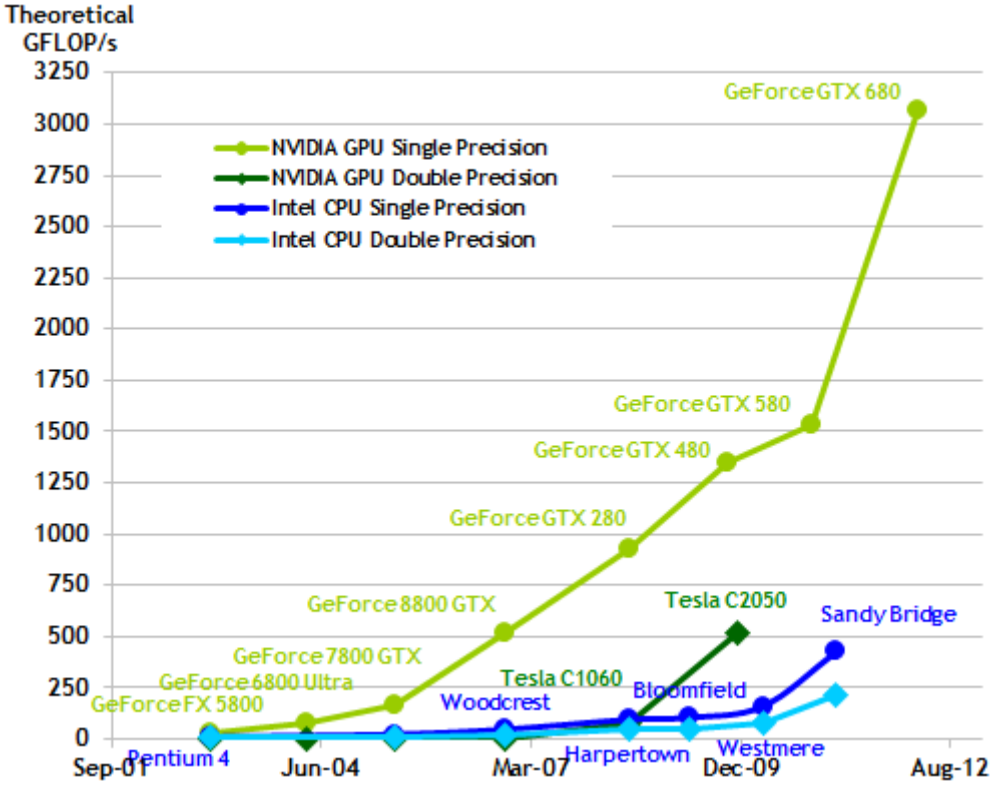


FIGURE 1.27 – Comparaison des flops théoriques entre CPU et GPU

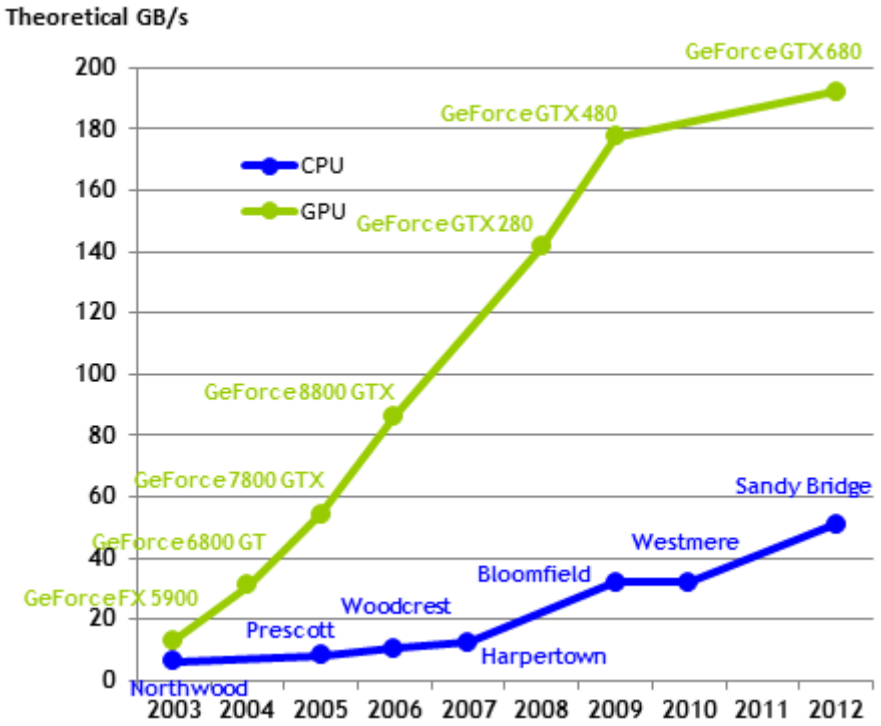


FIGURE 1.28 – Comparaison des bandes passantes entre CPU et GPU

threads dans un bloc de manière intriquée, c'est-à-dire avec des enchevêtrements, permettant ainsi de recouvrir les éventuels temps d'accès mémoire, comme montré sur la figure 1.31.

Les différentes générations des cartes CUDA sont désignées par leur fonctionnalités (appelées *compute capability* pour capacité de calcul). Suivant la version de *compute capability*, nous aurons des caractéristiques de fonctionnement légèrement différentes.

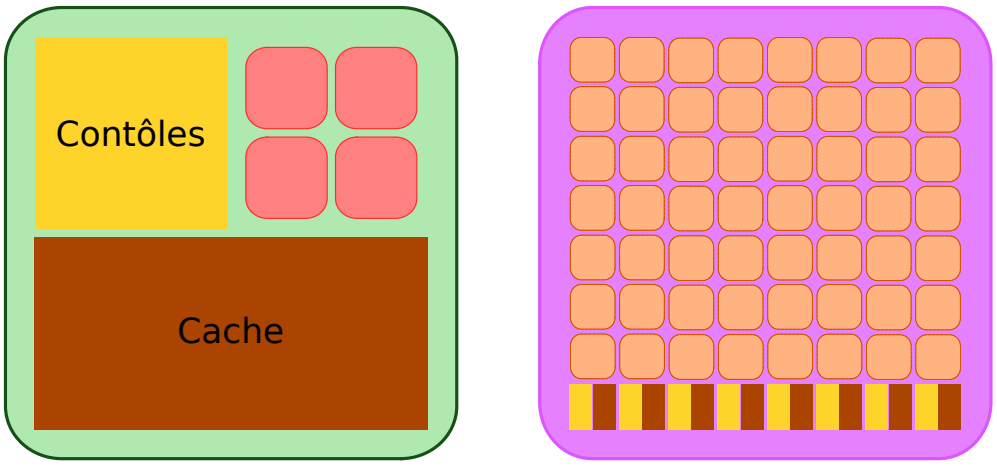
1.3.4.3 Mémoire

Comme nous l'avons vu, contrairement aux processeurs classiques, les processeurs graphiques possèdent leur propre mémoire, embarquée sur la carte graphique. Les processeurs graphiques sont reliés à la mémoire centrale par l'intermédiaire du bus PCI. Il est donc inévitable de transférer les données depuis la mémoire centrale vers la mémoire du GPU avant de pouvoir les exploiter. Pour les cartes de *compute capability* supérieure à 2.2, il existe des mécanismes de zéro copie. Sur la carte graphique, il existe plusieurs types de mémoire que nous allons détailler.

Mémoire globale Cette mémoire est partagée par tous les threads et par l'hôte. Elle permet de récupérer les données depuis la mémoire centrale de l'hôte. Cette mémoire permet d'effectuer des accès coalescents suivant certaines conditions, dépendant de l'architecture de la carte.

Pour les cartes 1.0 et 1.1, les accès mémoire d'un warp sont séparés en deux demandes, par demi-warp (16 threads). Les mots lus peuvent être de taille 4, 8 ou 16 octets. Pour des mots de 4 octets, les 16 mots accédés doivent se situer dans le même segment de 64 octets. Pour les mots de 8 octets, ils doivent être dans le même segment de 128 octets. Et enfin, pour les mots de 16 octets, les 8 premiers mots doivent être dans le même segment de 128 octets, tout comme les 8 derniers mots. Les accès doivent se produire dans l'ordre, le i^e thread doit accéder au i^e mot pour avoir un accès coalescent. Si ces conditions ne sont pas remplies, les accès d'un demi-warp seront séparés en 16 transactions.

Pour les cartes 1.2 et 1.3, le support des mots d'1 et 2 octets est ajouté. Pour les mots de 4, 8 et 16 octets, les accès peuvent se faire par segment de 128 octets si besoin (par exemple des accès non alignés sur un segment de 64 octets). Les accès n'ont plus besoin d'être séquentiels.



(a) Puce d'un processeur (b) Puce d'un GPU

FIGURE 1.29 – Processeur versus GPU

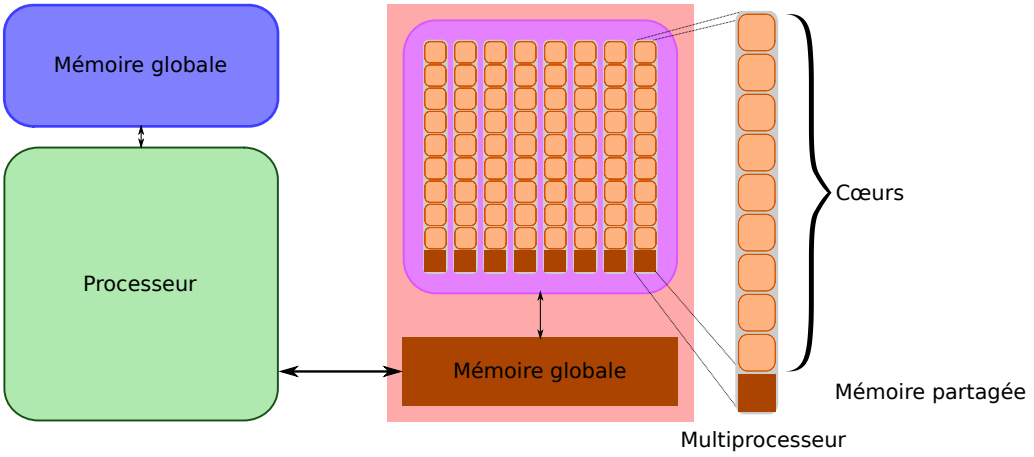


FIGURE 1.30 – Carte graphique

Avec les cartes 2.x et 3.0 apparaissent les accès en utilisant des caches, impliquant des accès coalescents pour tous les types d'accès concernant un seul segment. De plus les accès se font par warp et non plus par demi-warp. Pour cela, il faut que les données accédées soient alignées, c'est-à-dire sur la même ligne de cache, sinon il y aura autant de transactions que de lignes de cache impliquées.

Ces mécanismes sont résumés à la figure 1.32.

Actuellement, la capacité de cette mémoire peut atteindre jusqu'à 6 Go sur les dernières cartes.

Mémoire partagée Elle est partagée par tous les threads d'un même warp. Le temps d'accès est réduit par rapport à la mémoire globale, entre 20 et 40 fois moins [61]. Elle peut être accédée en même temps par autant de threads d'un warp, qu'elle possède de bancs mémoire. Il est possible aussi de réaliser une lecture multiple (*broadcast*) en même temps par tous les threads. Les adresses mémoire sont réparties cycliquement sur les différents bancs. Pour les cartes de *compute capability* 1.x, chaque banc mémoire est capable de fournir jusqu'à 4 octets en 2 cycles, et les mots de 4 octets successifs sont assignés à des bancs successifs. Malgré des warps de taille 32, il n'y a que 16 bancs mémoire. Les accès concurrents ne se font donc que par demi-warp. Pour les cartes de *compute capability* 2.x, la nouveauté est que les 32 threads d'un warp peuvent accéder en même temps à la mémoire grâce à la présence de 32 bancs mémoire. Enfin, pour les cartes 3.x, la différence par rapport à la 2.x est que les mots peuvent être de 4 ou 8 octets.

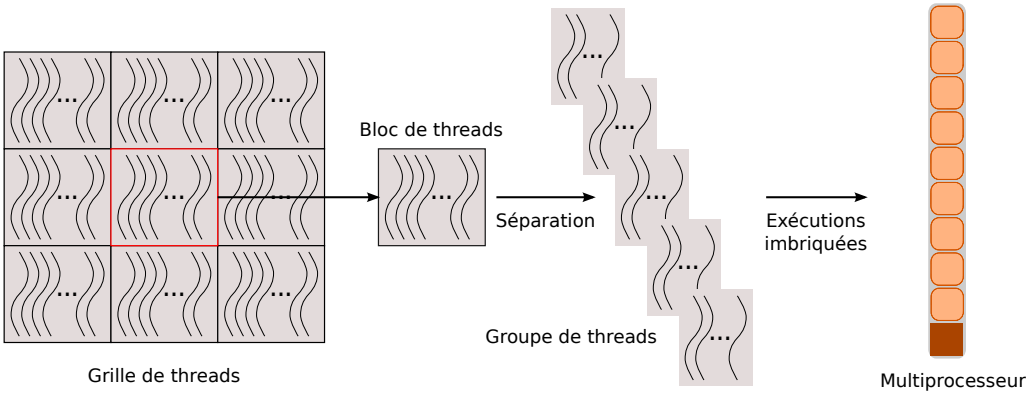


FIGURE 1.31 – Exécution sur un GPU

Pour éviter les conflits de bancs, il faut donc que les threads d'un warp (ou d'un demi-warp pour les 1.x) accèdent à des bancs différents. Les accès peuvent donc s'effectuer de manière totalement concurrente tant qu'il existe une bijection entre les threads et les bancs accédés. C'est également le cas lors d'une lecture globale (*broadcast*).

Mémoire locale C'est la mémoire la plus rapide. C'est un ensemble de registres, propres à chaque thread.

Il existe également la mémoire constante et la mémoire de texture que nous n'aborderons pas dans ce document. Pour plus de précisions, le lecteur pourra se référer à la *CUDA toolkit documentation*, disponible sur le site internet de NVidia, qui comporte toutes les informations détaillées.

1.3.4.4 Vers les machines hétérogènes

Pour profiter de la puissance des GPU, il est essentiel d'avoir un parallélisme de données très important. Malheureusement, toutes les applications et surtout tous les calculs ne s'y prêtent pas forcément, il reste donc indispensable d'utiliser encore des CPU performants. C'est pourquoi les GPU ont été ajoutés aux machines hybrides pour composer ce qu'on appelle des machines hétérogènes. À la figure 1.33, nous avons une machine composée de 3 nœuds, chacun comportant deux processeurs quadri-cœurs et deux cartes graphiques.

1.3.5 Discussion

Nous avons vu que les architectures ont rapidement évolué, nous pouvons alors nous demander si les architectures actuelles vont perdurer. C'est évidemment une question à laquelle nous ne pouvons répondre tant les inconnues sont nombreuses. Peut-être la barrière de la fréquence sera-t-elle levée et la course au nombre de cœurs par processeur ralentira ? Nous pouvons seulement étudier les tendances actuelles pour prévoir ce qui nous attend.

Quand nous détaillons l'évolution du TOP500 depuis quelques années, la tendance est à la grande multiplication de cœurs moins puissants (aux alentours de 2 GHz) avec des processeurs comportant 16 cœurs. Intel confirme la multiplication du nombre de cœurs avec la sortie de son processeur MIC en 2012. Le MIC, pour *Many Integrated Core Architecture*, maintenant appelé le Xeon Phi, intègre 61 cœurs.

L'autre tendance est comme nous l'avons vu, l'utilisation d'accélérateurs tels les GPU et même le rapprochement des CPU et des GPU. Nous pouvons citer AMD qui avec son HSA pour *Heterogeneous Systems Architecture* (anciennement Fusion) [36], intègre sur la même puce un CPU et un GPU. Du côté d'Intel, il y a les derniers Sandy Bridge qui intègrent un circuit décodant la vidéo. Chez NVidia, il y a des rumeurs sur le projet Denver qui vise à intégrer un GPU à un processeur ARM [31] pour tout type d'ordinateur, dont les supercalculateurs, ou encore le projet Boulder.

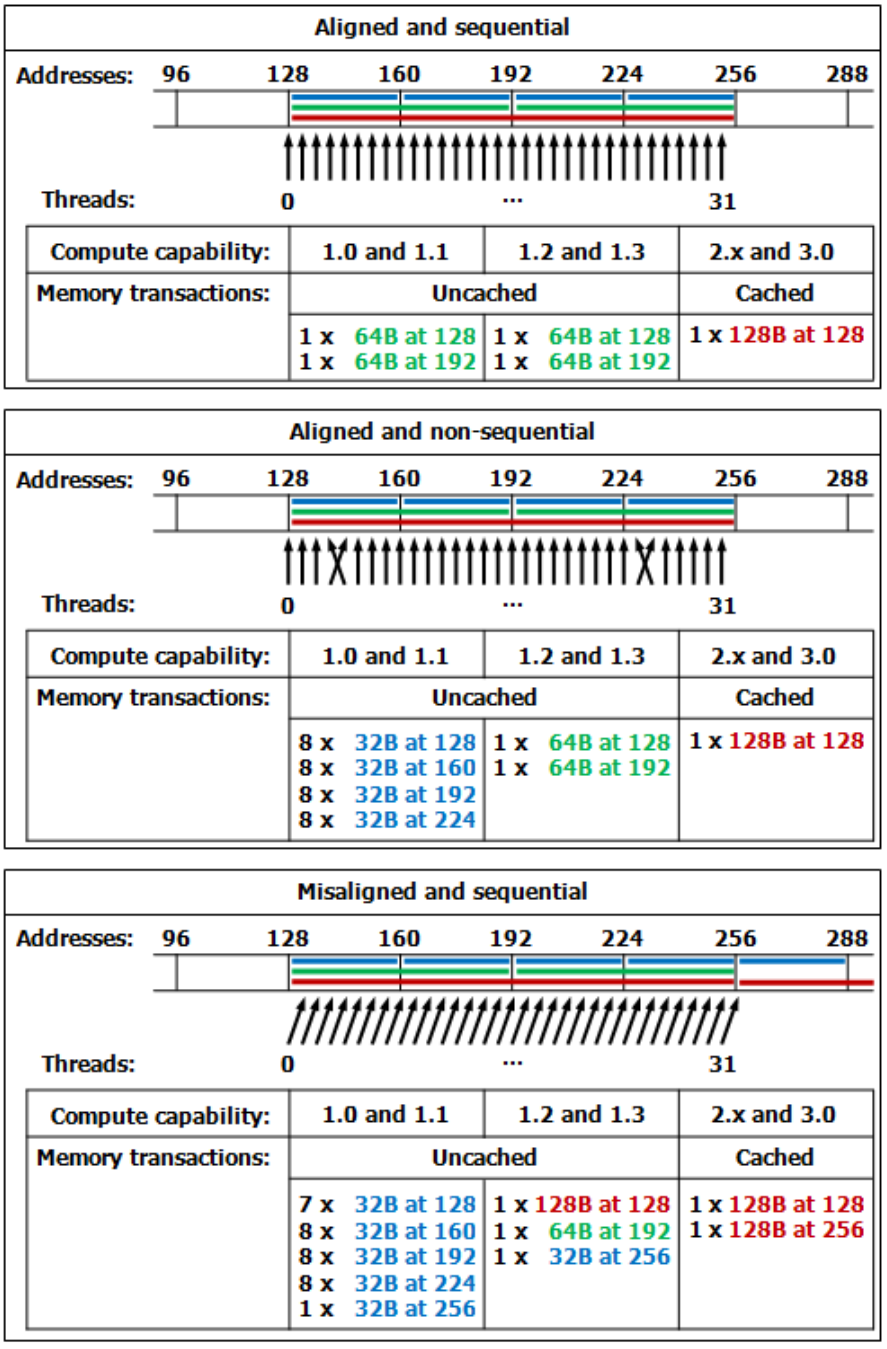


FIGURE 1.32 – Accès à la mémoire globale

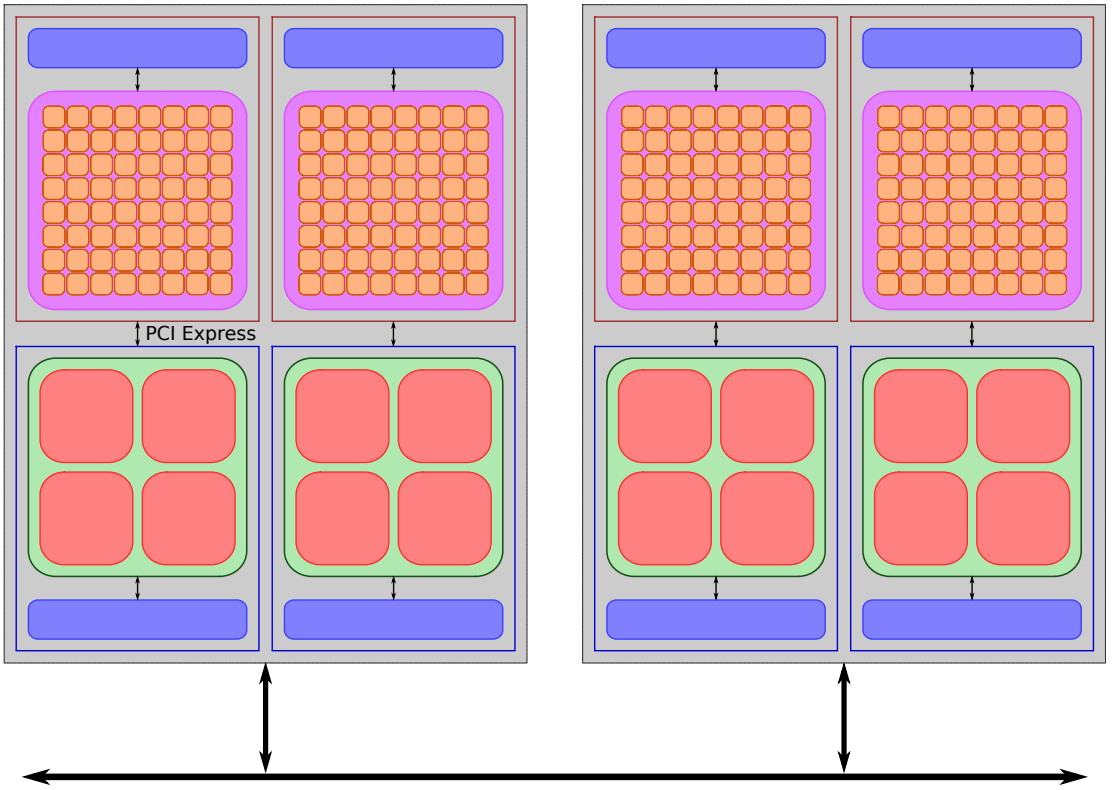


FIGURE 1.33 – Structure d'une machine hétérogène comportant deux nœuds composés de deux processeurs quadri-cœurs et deux GPU

1.4 Conclusion

Dans ce chapitre, nous avons montré en détails l'algorithme de la méthode multipôle multi-niveau. Il est basée sur la décomposition de l'espace en boîtes imbriquées qui forment un arbre, appelé *octree*. À chaque boîte est associé un vecteur complexe appelé champ, sur lequel vont s'effectuer les opérations. Celles-ci sont des opérations matricielles ou vectorielles, ainsi que des transformées de Fourier, qui s'organisent en deux phases pour les interactions lointaines : une remontée de la valeur du champ vers le sommet de l'arbre et une redescente qui prend en compte les interactions provenant des voisins éloignés. Il est indispensable d'ajouter à cela le calcul des interactions proches. Chaque étape de l'algorithme agit sur une ou plusieurs boîtes bien déterminées, en fonction de la géométrie de l'arbre.

Nous avons également étudié les évolutions des machines de calcul, qui deviennent de plus en plus complexes. Elles se composent de milliers de nœuds, comprenant des dizaines de cœurs. Nous avons donc à la fois de la mémoire partagée et de la mémoire distribuée. S'ajoutent à cela des accélérateurs, comme par exemple des GPU. L'algorithme de base de la méthode multipôle doit donc être adapté pour prendre en charge ces nouvelles architectures.

Sommaire

Chapitre

2

État de l'art

Together we stand... divided we fall !

Roger Waters, Pink Floyd, album The Wall (30 novembre 1979).

Nous souhaitons implémenter la méthode multipôle sur une machine hétérogène. Dans ce but, nous nous intéressons aux nombreux travaux existants, permettant des exécutions parallèles, afin de juger de leur efficacité dans ce contexte. Nous devons maintenant replacer la méthode multipôle dans le contexte du parallélisme. Pour cela, nous allons d'abord exposer les divers travaux sur la méthode multipôle, ayant un lien avec le parallélisme. Ensuite, nous montrerons les différents outils et techniques pour utiliser efficacement les machines hétérogènes, afin de guider nos choix dans l'implémentation la plus adaptée.

2.1 Parallélisation de la méthode multipôle

La méthode multipôle est une méthode qui permet d'accélérer des résolutions itératives, comme nous l'avons montré. Ses complexités en temps et en espace sont en $\mathrm{O}(n \log n)$. En électromagnétisme, il est intéressant de travailler sur des objets qui comportent un très grand nombre de mailles (plusieurs millions) et donc beaucoup de points de quadrature, pour gagner en précision et travailler sur des objets plus volumineux. Malgré l'augmentation de la fréquence des processeurs, nous restons limités. De plus, d'après ce que nous avons vu, la tendance depuis une décennie est plutôt à l'augmentation du nombre de processeurs et de cœurs. Par conséquent, une parallélisation est intéressante afin d'obtenir un résultat plus rapidement, mais elle l'est également afin de disposer de davantage de mémoire, par l'utilisation de plusieurs nœuds, et ainsi traiter des cas comportant un plus grand nombre de mailles.

C'est pourquoi de nombreux travaux ont été effectués pour permettre l'exécution de cette méthode sur des machines parallèles. Nous présenterons d'abord chronologiquement les évolutions dans les choix de distribution des données. Ensuite, nous évoquerons les travaux sur GPU. Puis, nous montrerons l'intérêt de choisir une bonne numérotation des boîtes. Enfin, nous présenterons des travaux sur la parallélisation de la méthode multipôle au moyen d'un liste de tâches.

2.1.1 Techniques de parallélisation

Nous allons réaliser une brève chronologie de différentes techniques de parallélisation de la méthode multipôle multi-niveau.

2.1.1.1 Parallélisation naïve

La méthode multipôle effectue des opérations sur un nombre important de boîtes organisées en octree. Par conséquent, la première parallélisation intuitive repose sur la distribution des boîtes entre les processeurs, comme montré sur la figure 2.1. Étant donnée une certaine séquentialité entre les niveaux de la méthode multipôle, la distribution est faite par niveau. Dans le but de limiter les communications entre les processeurs (ne partageant pas leur mémoire), la distribution doit permettre au maximum d'attribuer les enfants et les parents au même processeur.

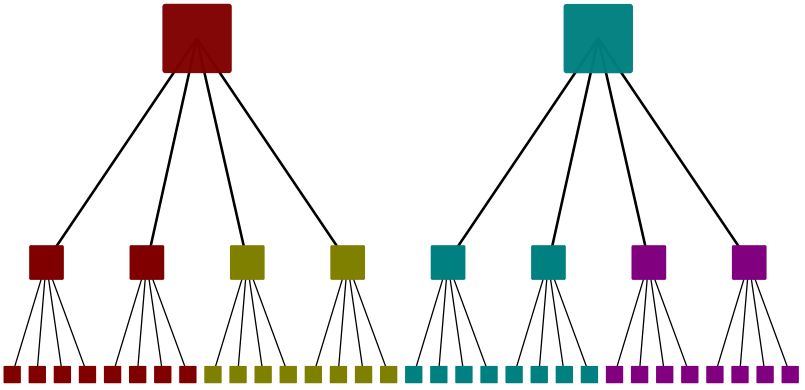


FIGURE 2.1 – Distribution naïve entre quatre nœuds représentés par des couleurs différentes

Plus on descend dans l'octree, plus il y a de boîtes. Le nombre de boîtes est donc suffisant dans le cas d'une distribution par boîte dans le bas de l'arbre. Pour le haut de l'arbre, le nombre de boîtes peut être inférieur au nombre de processeurs, ce qui laisse des processeurs sans boîte, et ainsi un déséquilibre de charge.

Pour la méthode multipôle utilisée dans le cadre de Laplace, le nombre de calculs est constant par boîte car le nombre de directions est constant par niveau. Les niveaux supérieurs, où le nombre de boîtes est faible par rapport au bas de l'arbre, ne génèrent pas beaucoup de calculs. La faible efficacité parallèle de ces niveaux n'a alors qu'une répercussion faible sur l'efficacité parallèle générale de la méthode multipôle.

Concernant notre cadre d'étude, c'est-à-dire avec Helmholtz, les vecteurs calculés pour chaque boîte ont des tailles qui correspondent au carré du nombre de directions. Ainsi, en descendant dans l'arbre, leur taille diminue. De ce fait, le nombre de calculs est quasiment le même à chaque niveau, le nombre de boîtes qui diminue en montant dans l'arbre étant compensé par le nombre de directions qui augmente. Une mauvaise répartition dans les niveaux hauts a forcément une conséquence importante sur l'efficacité de la parallélisation. Par conséquent, une telle parallélisation n'est pas envisageable avec un nombre élevé de processeurs, c'est-à-dire supérieur au nombre de boîtes du niveau le plus haut.

2.1.1.2 Parallélisation hybride

Afin de palier le problème des niveaux hauts qui n'ont pas assez de boîtes par rapport au nombre de processeurs, une nouvelle distribution de ces niveaux est envisagée par Velemparambil et Chew [82]. Elle se fait selon les directions pour les niveaux hauts, et selon les boîtes pour les niveaux qui comportent assez de boîtes par rapport au nombre de processeurs. Nous pouvons voir le principe de cette distribution à la figure 2.2.

Nous avons montré dans la section 1.2.1.4 que le nombre de directions dépendait du niveau dans l'arbre. Plus nous nous plaçons dans le haut de l'arbre plus le nombre de directions est élevé. Dans ce cas, ce nombre peut atteindre plusieurs centaines. Cette distribution permet ainsi d'assurer une meilleure distribution des calculs que précédemment et, Velemparambil et Chew, montrent que le surcoût de communication n'est pas important pour passer du niveau distribué au niveau partagé.

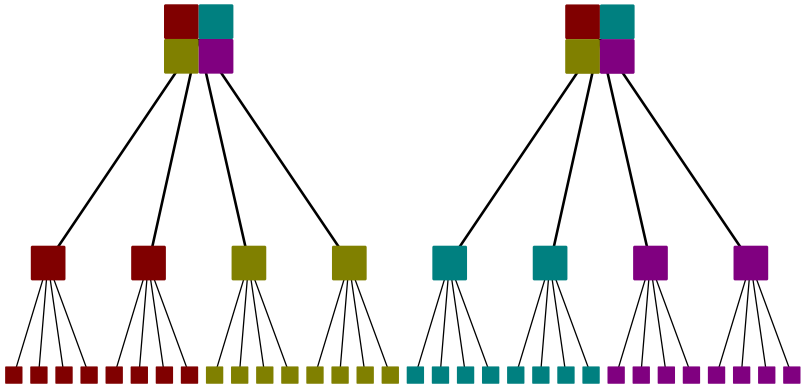


FIGURE 2.2 – Distribution hybride entre quatre nœuds représentés par des couleurs différentes

2.1.1.3 Parallélisation hiérarchique

La parallélisation précédente conduit à une amélioration de la distribution des niveaux hauts. Néanmoins, il subsiste un problème pour les niveaux médians. En effet, ces niveaux présentent le désavantage de ne pas comporter beaucoup de boîtes ni beaucoup de directions. Si le nombre de boîtes est proche du nombre de processeurs, une distribution suivant les boîtes va conduire à un grand nombre de communications par de nombreux processeurs lors d'une translation. De même, une distribution par directions tandis que leur nombre est faible va conduire à des communications entre de nombreux processeurs lors de l'interpolation.

C'est pourquoi Ergül et Gürel ont proposé dans [33] de distribuer chaque niveau à la fois en directions et en boîtes. Les niveaux hauts seront davantage distribués par multipôle et les niveaux bas davantage par boîte, voir la figure 2.3.

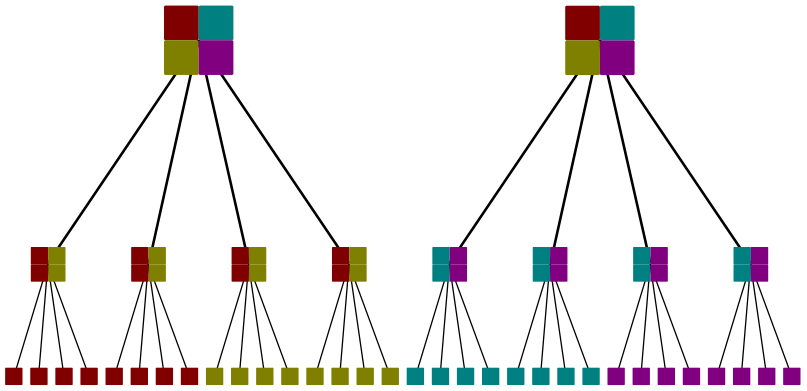


FIGURE 2.3 – Distribution hiérarchique entre quatre nœuds représentés par des couleurs différentes

Avec cette distribution hiérarchique, ils obtiennent dans [34] une efficacité pour 128 processeurs de 60% alors qu'ils avaient obtenu 30% à l'aide d'une distribution hybride. Même si cette distribution apporte un gain, les résultats obtenus ne sont pas à la hauteur des machines actuelles en termes d'efficacité ou de passage à l'échelle.

2.1.1.4 Parallélisation simple-niveau

Les machines comportent de plus en plus de processeurs. Or, malgré les nombreux travaux existants, les méthodes précédentes ne passent pas à l'échelle sur ces machines. La méthode multipôle simple-niveau (SLFMM) présente l'avantage de passer parfaitement à l'échelle, simplement en effectuant une distribution suivant les directions. En effet, en l'absence des étapes de montée et descente, comme c'est le cas dans la méthode multipôle simple niveau, les calculs entre différentes directions sont indépendants entre eux. La seule communication nécessaire est une réduction à la

fin de l'algorithme. Le défaut de la méthode simple niveau est bien évidemment sa complexité en $O(n^{\frac{3}{2}})$ au lieu de $O(n \log n)$ pour la méthode multipôle multi-niveau (MLFMM).

2.1.1.5 Parallélisation FFT-FMM

En 2007, Waltz et *et al.*, se sont intéressés à nouveau à la méthode multipôle simple-niveau dans [84]. Ils ont apporté une amélioration en termes de complexité nommée la FMM-FFT. Grâce à l'introduction de transformées de Fourier (FFT) dans la translation, ils sont arrivés à une complexité en temps en $O(n^{\frac{4}{3}} \log^{\frac{2}{3}} n)$. Ainsi, ils ont voulu démontrer que la FMM-FFT était une alternative viable à la MLFMM, utilisée avec une machine à mémoire distribuée comportant 64 processeurs ou plus. Bien que leur algorithme soit moins efficace que la MLFMM, il présente l'avantage d'avoir une meilleure scalabilité. Par conséquent, il peut se révéler intéressant sur des machines à mémoire distribuée.

2.1.1.6 Parallélisation MLFMA-FFT

Suite aux travaux réalisés sur la FMM-FFT, Mouriño *et al.* [56] ont réussi à obtenir des scalabilités quasiment parfaites avec 512 cœurs sur une machine hybride (mémoire partagée et mémoire distribuée).

Taboada *et al.* dans [76] évoquent le fait que la parallélisation de la MLFMM est plutôt intéressante en mémoire partagée du fait de l'absence de communications. Ils proposent alors de coupler les avantages de la MLFMM en mémoire partagée et de la FMM-FFT en mémoire distribuée. Pour cela, ils bloquent la remontée globale au niveau à partir duquel il n'y a plus assez de boîtes, par rapport au nombre de nœuds (mémoire distribuée). De cette manière, ils réalisent une FMM-FFT sur ce niveau. Pour les niveaux en dessous, ils font une distribution par boîtes entre les nœuds et réalisent une MLFMM classique. Les seules communications inter-nœuds proviendront de l'étape de translation. C'est pourquoi, ils répliquent les données utiles des autres nœuds. Leur algorithme s'appelle la MLFMA-FFT.

Grâce à cette technique de parallélisation, ils ont résolu un problème à 150 millions de points en utilisant 1024 processeurs, avec une efficacité d'environ 75%. Ils ont également traité un problème de taille 500 millions, avec une version modifiée de leur algorithme utilisant moins de mémoire, en obtenant une efficacité de 50%.

2.1.1.7 Du côté des GPU

Il existe de nombreux travaux concernant l'implémentation de la méthode multipôle en utilisant des GPU. Malheureusement ceux-ci ne concernent pas les noyaux oscillants et donc les équations de Maxwell. Cependant, à titre purement indicatif, nous pouvons citer Gumerov [43] qui parvient à une accélération de 70 entre un processeur monocœur Intel Core 2 extreme QX de 2.67 Ghz et un GPU Nvidia Geforce 8800 GTX. Cette accélération s'explique par l'utilisation de noyaux optimisés tels que les CUBLAS que nous verrons à la section 2.2.3.2.

À nouveau pour les noyaux non-oscillants, Lashuk *et al.* dans [53], montrent un gain de 30 en utilisant 256 GPU en plus des 1024 CPU. Ils présentent également une exécution sur 65 000 cœurs avec une efficacité de 50 %. Dans [66], nous retrouvons les travaux précédents, utilisés dans un couplage de code pour la simulation du flux sanguin, avec une exécution sur 200 000 cœurs !

Ces résultats sont impressionnants, mais malheureusement ne sont pas transposables à des noyaux oscillants, car, comme nous le découvrirons dans la suite, le nombre de directions variable joue un rôle très important dans les performances.

2.1.2 Numérotation des données

Afin d'obtenir une parallélisation efficace, il est important de chercher à réduire au maximum le nombre de communications. C'est pourquoi, il est indispensable de tenir compte des liens de parenté des boîtes, et de leur proximité. Notamment quand une boîte est affectée à un nœud, toute sa descendance doit l'être aussi, pour ne pas ajouter des communications lors des montées et descentes. De la même façon, mettre les boîtes proches le plus possible sur le même nœud, permet

de réduire les communications lors des translations car elles ont une forte probabilité d'être voisins éloignés et, de ce fait, de communiquer.

Pour simplifier la distribution, il est intéressant de choisir une numérotation qui tient compte de ces critères, afin de seulement avoir à distribuer des intervalle de boîtes. Sylvand [74] a réalisé un comparatif poussé sur différentes numérotations : cartésienne, hiérarchique (ou numérotation de Morton), de Peano. La conclusion de cette étude est que la numérotation de Morton est la meilleure et est simple à mettre en place.

C'est la numérotation que nous avons utilisée sur la figure 1.14 et que nous utiliserons dans notre implémentation. Cette numérotation offre la possibilité d'avoir une numérotation continue pour toute une fratrie, et une grande partie des voisins éloignés aura un numéro proche.

2.1.3 Utilisation d'une liste de tâches

Une bonne façon d'aborder une parallélisation est de penser l'algorithme sous forme de tâches, ce qui la rend généralement plus aisée à réaliser. Dans sa thèse, Sylvand [74] présente son implémentation de la méthode multipôle basée sur une liste de tâches. À noter, que cette implémentation n'a pas comme cible une architecture hétérogène (nous somme en 2002 !). Il y décrit l'algorithme multipôle comme une suite de tâches élémentaires possédant des contraintes sur leur ordre d'exécution. Les tâches définies sont l'agrégation, la désagrégation, la montée, la descente et la translation. Chaque processeur sera responsable d'une boîte, et gèrera toutes les tâches élémentaires qui la concernent.

Un des avantages de cette méthode est que la liste est construite au préalable. Cela donne la possibilité par exemple, de procéder à des optimisations sur cette liste en faisant des réordonnancement pour limiter les cas où les processeurs sont en attente de résultats et n'ont pas de travail disponible. Cependant, c'est un travail très complexe à réaliser qui doit être adapté au cas par cas. Nous verrons à la section 2.2.4, des outils offrant la possibilité de réaliser cette tâche.

Afin d'utiliser au mieux une architecture hétérogène, il n'est pas efficace d'employer des tâches élémentaires. En effet, celles-ci ont besoin de beaucoup de calculs pour réaliser des opérations vectorielles optimales. Il est alors préférable de se servir d'une liste de tâches qui contiennent plusieurs calculs. De plus, il faut bien distinguer au sein d'une même tâche les parties qui diffèrent pour avoir un maximum de régularité. Par exemple, il peut être intéressant de séparer la montée, en décalage, transformée de Fourier et multiplication de la méthode d'Alpert, comme nous le verrons à la section 3.2.1.

2.2 Outils pour la parallélisation

Afin de pouvoir programmer sur des machines parallèles, il est nécessaire d'employer des bibliothèques offrant la possibilité d'exploiter les différentes ressources. La création des tâches parallèles peut être faite explicitement par le programmeur, ou celui-ci peut utiliser des outils automatiques. Au moment de l'exécution, les tâches peuvent être assignées statiquement à des ressources, ou nous pouvons avoir un système dynamique qui agira en fonction de la disponibilité des ressources. Mais avant de détailler ces points, nous allons regarder comment optimiser simplement un programme sur un seul cœur.

Le but de cette section n'est pas de dresser la liste exhaustive de tous les outils et toutes les techniques existants. Ce travail a déjà été réalisé parfaitement par de nombreux auteurs que nous référencerons directement dans la suite du document. De notre côté, nous cherchons simplement à poser les bases qui serviront à justifier nos choix présentés au chapitre suivant, servant la méthode multipôle.

2.2.1 Programmer sur un seul cœur

Pour aboutir au calcul haute performance, avant de nous concentrer sur le parallélisme, il est primordial d'avoir un code efficace sur un seul cœur. Nous ne parlons pas d'algorithme, pour lequel une parallélisation n'est pas forcément la plus efficace lorsqu'elle est effectuée sur la meilleure version séquentielle (non parallèle).

Nous ne présenterons pas ici de techniques évoluées d'optimisation mais simplement des techniques simples, réalisables par tous et qui n'apportent pas ou peu de surcoût au codage.

2.2.1.1 Le compilateur

Afin de passer du langage de programmation, compréhensible par le programmeur, au langage compréhensible par le processeur, nous utilisons un compilateur. Son travail ne se limite pas à une simple traduction. Il réalise bien plus, il procède à de nombreuses optimisations qui augmenteront nettement les performances de notre programme. Nous pouvons nous en rendre compte en les activant, ou non, lors de la compilation (avec souvent plusieurs niveaux d'optimisation disponibles). Par exemple, avec le compilateur GNU pour le langage C, il existe les options *-O0*, *-O1*, *-O2*, *-O3*, pour un niveau incrémental d'optimisations. Nous n'allons pas citer les différentes optimisations existantes, qui servent surtout à réordonnancer les instructions afin d'éviter les attentes du processeur, ou alors à profiter au maximum des caches, en augmentant la localité temporelle par exemple. Pour davantage de détails, le lecteur pourra se référer à [2].

Malheureusement, le compilateur n'a pas un grand pouvoir d'action. Il ne peut effectuer que des modifications locales, et ne pourra pas corriger un programme mal conçu (en termes de performances). C'est pourquoi, le programmeur doit porter une attention particulière aux performances quand il réalise son implémentation.

2.2.1.2 Les performances au cœur de la conception

Comme nous l'avons montré à la section 1.3.1, les accès à la mémoire centrale ralentissent le processeur. Par conséquent, il faut maximiser les succès de cache. C'est pourquoi, nous devons augmenter les localités temporelle et spatiale. Le compilateur fait également ce travail, mais concernant la localité spatiale, c'est toute la conception de l'application qui entre en jeu.

En effet, pour stocker nos données et réaliser des calculs, nous devons choisir des types de données. Ceux-ci vont conditionner la localité spatiale. Afin d'être plus concret, prenons un exemple. Nous avons une structure décrivant une boîte, qui contient les champs suivants : le numéro de la boîte, son nombre d'éléments, sa taille et sa position. Maintenant, nous voulons connaître parmi un ensemble de boîtes le nombre moyen d'éléments qu'elles contiennent. Pour ce faire, nous allons parcourir nos structures parmi l'ensemble et lire le champ concernant le nombre d'éléments. Quand nous lisons le champ qui nous intéresse dans la structure de boîte, les données proches sont chargées dans une ligne de cache avec d'autres données proches. Une première remarque est qu'il serait intéressant de disposer d'un ensemble de structures de boîtes contigües en mémoire. Mais ce qui pose problème c'est que des données dont nous n'aurons pas l'utilité dans notre opération, que sont le numéro, la taille et la position, seront aussi chargée dans le cache. Le cache sera alors rempli avec des données inutiles et il y aura davantage de défauts de cache. Nous comprenons bien que dans ce genre de problème, le compilateur à moins qu'il ne soit capable de repenser notre application, n'a pas un grand pouvoir d'intervention.

Deux règles simples se déduisent de ce que nous avons étudié. La première est qu'il faut au maximum garder les données contigües en mémoire, surtout si elles doivent être parcourues. La deuxième est qu'il faut préférer, pour des besoins de performances, les structures d'ensembles aux ensembles de structures, quand tous les éléments de la structure ne sont pas tous utilisés en même temps.

2.2.1.3 Aider le compilateur

Comme nous l'avons expliqué, le compilateur n'est pas capable de remanier tout le code. Il n'est pas capable non plus de réaliser des optimisations entre une fonction appelée et sa fonction appelante (par exemple pour une bibliothèque ou quand le contexte d'appel a une influence). Pour que cela soit envisageable, il faut que la fonction appelée soit remplacée par son code dans la fonction appelante. Cette opération appelé *inlining* n'est pas effectuée automatiquement par le compilateur car elle augmente la taille du code et peut, dans certains cas, conduire à une baisse de performance. Si le programmeur pense qu'il va y avoir des optimisations possibles ou si une fonction est appelée un grand nombre de fois mais n'est pas très coûteuse (et donc les simples

appels de cette fonction apportent un surcoût non négligeable), il a la possibilité d'indiquer au compilateur d'effectuer l'*inlining* de cette fonction.

Pour constater les optimisations qu'il est possible de réaliser concernant les caches, étudions leur impact sur le produit de deux matrices. Dans cet exemple nous disposons de deux matrices $\mathbb{A}$ et $\mathbb{B}$ de taille $n \times n$ et nous voulons calculer $\mathbb{C} = \mathbb{A}\mathbb{B}$. Nous avons alors :

$$\mathbb{C}_{i,j} = \sum_{k=0}^{n} \mathbb{A}_{i,k}\mathbb{B}_{k,j}$$

Nos matrices sont stockées dans l'ordre des lignes. En prenant comme algorithme basique de ce produit l'imbrication des trois boucles en i, j et k, où k est la boucle la plus interne, le parcours de $\mathbb{A}$ et de $\mathbb{C}$ va s'effectuer dans l'ordre des lignes (respectivement j puis i, et k puis i) mais pour $\mathbb{B}$ il se fera dans l'ordre des colonnes (k puis j). Sur la figure 2.4, les flèches en pointillés représentent l'ordre de parcours de l'algorithme.

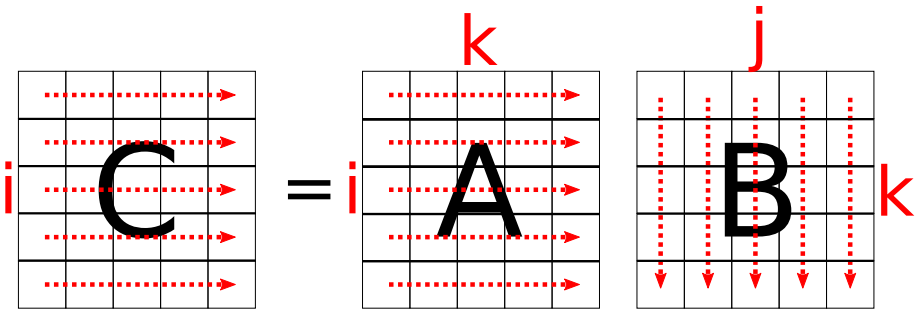


FIGURE 2.4 – Sens de stockage et de parcours des matrices

Les sauts dans les accès mémoire provoqués par les nombreux accès par colonne de la matrice $\mathbb{B}$, provoquent des défauts de caches. Cela est d'autant plus important que chaque colonne de $\mathbb{B}$ est parcourue n fois. Une solution simple est de transposer la matrice $\mathbb{B}$ avant d'effectuer le produit, le parcours lors de la transposition n'ayant lieu qu'une seule fois. Avec cette simple astuce, en prenant une valeur de n à 1000, nous obtenons sur un Intel Core 2 à 2.6 *Ghz*, un gain supérieur à 4 [32] !

Nous pouvons aller plus loin dans les optimisations. Le problème avec l'algorithme de base est que les colonnes de la matrice $\mathbb{B}$ sont parcourues plusieurs fois mais à des intervalles trop distants en temps. Les lignes de cache contenant les valeurs des colonnes ont déjà été vidées quand un nouvel accès à ces données a lieu. Afin de pouvoir utiliser les données dans le cache avant qu'elles soient vidées, l'ordre du produit doit être modifié. Nous procédons à un produit par blocs, adaptés à la taille du cache. Avec ce nouvel algorithme nous obtenons un gain quasiment égal à 6 par rapport à l'algorithme de base. Il existe une dernière amélioration qui consiste à tirer profit des instructions vectorielles du processeur (SSE2). En l'ajoutant, l'algorithme final peut afficher un gain de 10 par rapport à l'algorithme naïf. Pour davantage de détails sur cet exemple, vous pouvez consulter le très complet [32].

Cet exemple nous a permis de voir qu'il ne faut pas négliger le cache lors de la conception des algorithmes. Heureusement, en général, les compilateurs sont capables de détecter automatiquement ce genre d'optimisations, et il est même possible de les y aider. De plus, comme nous allons le voir à la section 2.2.1.5, des outils permettent, avec certains types de calcul, de nous affranchir de ce travail.

2.2.1.4 Alignement mémoire

Certains processeurs chargent les données par blocs, de la taille d'un mot, alignés en mémoire. Si nous voulons lire un entier, mais que celui-ci se trouve dans deux blocs différents, nous aurons alors besoin de deux chargements. Cette situation peut être encore plus handicapante quand elle se produit sur une instruction vectorielle de type SSE. Heureusement, le compilateur sait bien gérer l'alignement des données sur les blocs, quitte à rajouter du bourrage (ou *padding* en anglais). Il est aussi capable de réordonner les champs dans une structure pour des besoins d'alignement.

2.2.1.5 Les bibliothèques de calcul scientifique

Comme nous venons de le voir, il est primordial de prendre en compte l'architecture sur lequel le programme est exécuté, et ses types de calcul.

La majorité des calculs s'adapte simplement aux différentes architectures. Mais dans le cas d'optimisations avancées, comme nous l'avons exposé avec la multiplication, il y a une forte dépendance à la machine sur laquelle sera exécuté le programme. Nous obtenons un code qui ne sera pas portable (en termes de performances) et non pérenne. Heureusement, il existe des bibliothèques qui incluent des opérations mathématiques optimisées pour les différents processeurs.

La plus connue de ces bibliothèques est sans aucun doute la bibliothèque BLAS [12] pour *Basic Linear Algebra Subprograms*. Il existe de multiples implémentations développées par les constructeurs de processeurs comme Intel ou AMD, avec respectivement MKL [46] et ACML [5] (qui font bien plus), afin de fournir des opérations optimisées sur leurs processeurs, mais également par d'autres équipes telles que Goto Blas [20] ou Atlas [86]. Comme son nom l'indique, elle permet de réaliser les opérations d'algèbre linéaire de base, sur des vecteurs et des matrices, telles que des copies, des échanges, des multiplications, des produits scalaires. Les BLAS sont classés en trois niveaux suivant la complexité des calculs. Les BLAS 1 concernent les calculs linéaire de type par exemple produit scalaire, les BLAS 2 les calculs quadratiques de type produit matrice/vecteur et les BLAS 3 les calculs cubiques de type produit matrice/matrice.

Les BLAS 3 sont des opérations qui utilisent, avec des matrices de taille $n \times n$, $2n^2$ données pour n^3 opérations. Elles permettent donc d'exploiter davantage la puissance de calcul d'un processeur que les BLAS 1 ou 2, qui ont un rapport entre données utiles et calculs nécessaires, plus importants.

Il est important d'évoquer la bibliothèque FFTW [37], qui nous sera utile lors des interpolations. Elle implémente de manière optimisée les transformées de Fourier et dans ce but elle impose d'agir sur des espaces mémoire correctement alignés, voir 2.2.1.4.

Pour la pérennité des codes, il est indispensable d'utiliser ce genre de bibliothèques, qui évolueront avec les architectures, et qui sauront garder le code de notre programme intact et toujours performant.

2.2.2 En parallèle

Maintenant que nous avons rapidement vu comment nous devons programmer sur un cœur, nous allons pouvoir aborder comment exploiter davantage de ressources de calcul avec le multicœur ou le multi-processeur.

C'est pourquoi, nous aborderons les différentes approches suivant comment les données sont échangées et les types de ressources de calcul, les outils pour y parvenir étant très différents.

2.2.2.1 Programmation en mémoire partagée

L'intérêt évident de la mémoire partagée réside dans l'échange des données qui se fait de façon plus ou moins transparente par l'intermédiaire des threads, ou processus légers. Les threads s'opposent aux processus, plus gourmands et ne partageant pas leurs données entre eux.

Pthreads La bibliothèque Pthreads [57, 16] (pour POSIX threads) est une bibliothèque portable qui implémente les threads. Elle met à disposition une interface complète permettant de manipuler les threads : création, synchronisation, attente, destruction... Elle offre aussi des fonctions qui permettent le partage de données sans conflits par l'intermédiaire de mutex. Ce cas de figure se retrouve par exemple quand nous devons empêcher la modification de données qui peuvent être lues par un autre thread.

Cette bibliothèque offre la possibilité de lancer facilement plusieurs calculs en parallèle mais reste assez lourde à utiliser quand il s'agit de paralléliser de nombreuses parties d'un code.

OpenMP La complexité de la création et de la gestion des threads peut devenir importante dans l'écriture d'un code. En 1997, le standard OpenMP [29, 21], pour *Open Multi-processing*, est défini. C'est une interface de programmation qui offre une utilisation simplifiée des threads. OpenMP, moyennant quelques instructions sous la forme de directives de compilation, s'occupe de

la création et de la destruction des threads. Il offre davantage de possibilités, comme par exemple, distribuer les différentes itérations (indépendantes) d'une boucle entre différents threads, grâce à son support exécutif. Nous avons aussi la possibilité de déclarer des tâches depuis la version de 2008. La gestion des variables est nettement simplifiée : il suffit de déclarer quelles variables vont être accédées par plusieurs threads et comment, et OpenMP va générer à la compilation le code nécessaire.

OpenMP est beaucoup utilisé notamment sur les architectures hybrides/hétérogènes grâce à sa grande simplicité. Malheureusement, cette grande simplicité ne permet pas d'avoir un contrôle total sur les placements, et il est donc très difficile d'exploiter pleinement une architecture fortement NUMA (accès mémoire non uniformes).

Cilk Le langage Cilk développé en 1994 [13] est une extension au langage C. Le programmeur spécifie par des mots-clefs quelles sont les fonctions à effectuer en parallèle. Au moment de l'exécution, l'ordonnanceur de Cilk s'occupe de distribuer les tâches entre les processeurs. Pour cela, au moment de l'exécution, des threads seront créés avec chacun sa file d'exécution de tâches. Il est également possible de procéder à du vol de tâches par un thread inactif. Malheureusement, Cilk ne tient pas compte de l'aspect NUMA.

TBB Intel Thread Building Blocks est une bibliothèque pour le C++ apparue en 2006 [87]. Cette bibliothèque a pour vocation à être de plus haut niveau possible, laissant seulement l'utilisateur indiquer les portions de code parallélisables. Elle reste cependant plus intrusive qu'OpenMP. Comme avec Cilk, il y a des files d'exécution de tâches pour les threads. De plus, le mécanisme de vol de travail prend en compte la topologie de la machine (notamment la distance des données et le partage de caches).

2.2.2.2 Programmation en mémoire distribuée

Afin de partager des données en mémoire distribuée, il est nécessaire d'effectuer des transferts, il y a donc un surcoût par rapport à une architecture à mémoire partagée. Deux approches coexistent dans la gestion de la mémoire distribuée. La première est appelée programmation par passage de messages. Les transferts sont réalisés explicitement par le programmeur. La deuxième, la programmation par espaces d'adressage globaux partagés, offre une gestion implicite des transferts entre les mémoires.

Nous ne parlerons pas des machines à mémoire distribuée partagée (DSM pour *Distributed Shared Memory*) où la mémoire distribuée est vue comme une seule mémoire. Ce type de mémoire offre difficilement de hautes performances.

Programmation par passage de messages Ce modèle de programmation implique une gestion des données faite exclusivement par le programmeur.

PVM Parallel Virtual Machine est apparu en 1989 [39]. Elle fournit des fonctions permettant la création de tâches, les communications entre elles et les synchronisations. Elle a eu un grand succès auprès des utilisateurs de systèmes distribuées mais son manque d'optimisation a fait que les utilisateurs se sont tournés vers MPI.

MPI Message Passing Interface est un standard sorti en 1993, qui apporte comme PVM. Cependant il est conçu initialement pour la performance. Ainsi, les opérations telles que les communications non bloquantes ont été considérées dès la conception et sont gérées efficacement.

Nous trouvons de nombreuses implémentations dans différents langages, citons les deux plus célèbres : MPICH2, OpenMPI. Il existe aussi des dérivées de ces implémentations faites par des constructeurs ou des assembleurs. MPI est très utilisé dans la communauté scientifique qui travaille sur des architectures hybrides.

Programmation par espaces d'adressage globaux partagés Ce type de programmation présente l'avantage d'être facile à utiliser. En effet, la mémoire distribuée est vue comme une mémoire partagée.

UPC Unified Parallel C est une extension du langage C apparue en 2001 [3]. Dans UPC, les données sont par défaut privées à un thread, et sont stockées sur la mémoire du nœud rattaché au thread. Le programmeur a la possibilité de définir des variables comme partagées. Elles seront alors stockées sur une seule mémoire. Un tableau partagé sera stocké sur plusieurs nœuds par exemple en blocs cycliques, le programmeur pouvant spécifier le type de stockage. Quand une tâche est déclarée comme parallèle, elle est exécutée par les threads qui possèdent les morceaux de données concernés.

HPF High Performance Fortran est une extension du langage Fortran 90 sortie en 1993 [50]. Les directives de compilation de cette extension offrent des outils d'alignement de données, de distribution de données, et de parallélisation de boucles. HPF a été majoritairement abandonné par la communauté, notamment à cause de la grande complexité pour son implémentation dans les compilateurs.

CAF Co-array Fortran [59], est également une extension du Fortran. Elle se base sur la réplication de programmes exécutés de manière asynchrone. Ils ont chacun leur propre jeu de données, les accès partagés étant faits explicitement. CAF fait partie de la norme Fortran 2008, et est implémentée dans le compilateur g95.

XcalableMP Comme OpenMP, c'est une interface basée sur des directives, compatible avec les langages Fortran et C. Elle est apparue en 2000 et s'inspire de HPF. Des directives permettent de partager le travail et de distribuer les données utiles.

Il existe bien d'autres outils de parallélisation de ce type que nous ne détaillerons pas (Titanium, X10, Fortress, Chapel ...).

2.2.3 Programmation des GPU

Quand un programme est lancé sur une machine ayant des GPU, il est exécuté par le CPU, seules certaines fonctions sont exécutées par le GPU. Ceux-ci sont utilisés comme accélérateurs. Pour cela, il faut que notre programme contienne des parties de code adaptées aux processeurs graphiques.

2.2.3.1 Programmation explicite

Différents langages ont été créés afin de prendre en charge les architectures particulières de nos processeurs graphiques.

Cuda Signifiant *Compute Unified Device Architecture*, CUDA est le langage de programmation de NVidia permettant d'exécuter des programmes sur ses GPU. Sa première version sort en 2007 [62, 61, 60].

Pour exécuter une fonction sur un GPU, il est nécessaire d'écrire un noyau (*kernel*) et de l'appeler depuis notre programme exécuté sur l'hôte c'est-à-dire le CPU. CUDA est une extension du C et permet d'écrire le noyau qui contient les instructions à réaliser par le GPU. Afin de mieux comprendre son utilisation nous allons prendre un exemple de produit de matrices, voir le code 2.1.

```
1    // Definition du noyau
2    __global__ void MatMul(float *a, float *b, float *c, int N)
3    {
4        int row = blockIdx.y * blockDim.y + threadIdx.y;
5        int col = blockIdx.x * blockDim.x + threadIdx.x;
6
```

```
7           float sum = 0;
8
9           for (int i = 0; i < TILE_DIM; i++)
10            c[row*N+col] += a[row*TILE_DIM+i] * b[i*N+col];
11          c[row*N+col] = sum;
12      }
13
14      int main()
15      {
16          ...
17          // Invocation du noyau
18          dim3 threadsPerBlock(16, 16);
19          dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
20          MatMul<<<numBlocks, threadsPerBlock>>>(A, b, C, N);
21          ...
22      }
```

Code 2.1 – Produit de deux matrices avec CUDA

Pour les cartes de *compute capability* inférieure à 2.2, il est nécessaire de procéder à la copie entre la mémoire hôte et la mémoire globale. Il est possible de réaliser cette opération de manière synchrone ou, pour de meilleures performances, en recouvrant la copie par des calculs de manière asynchrone.

La figure 1.31 nous montre que les différents threads doivent être partitionnés en blocs composant une grille. Cette opération est effectuée par le programmeur au moment du lancement du noyau (code 2.1 ligne 19). Nous définissons alors une grille 3D de blocs de threads. Dans l'exemple, la grille ne comporte que deux dimensions, la première étant égale à la largeur de la matrice divisée par la largeur du bloc, et la deuxième de la même manière avec les hauteurs. Ainsi, chaque valeur de la matrice est associée à un thread. Les blocs ont été choisis de taille carrée 16×16, c'est un choix couramment fait. Ce découpage est résumé sur la figure 2.5.

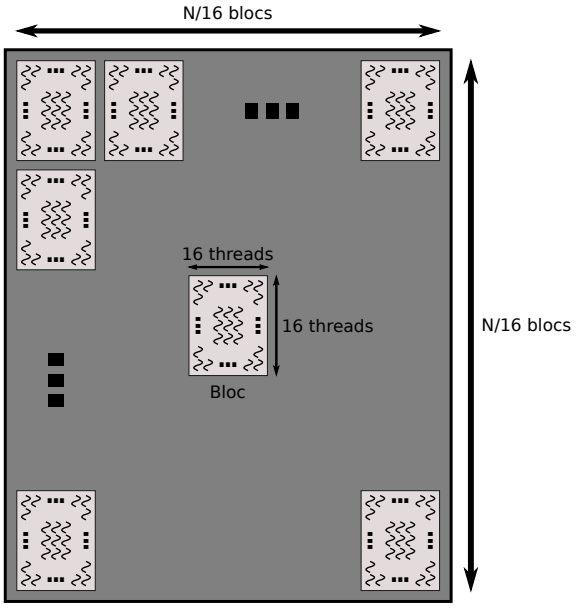


FIGURE 2.5 – Création de la grille

Le GPU a accès à plusieurs types de mémoire présentant chacune ses particularités, voir la section 1.3.4.3. Afin de nous rendre compte de l'impact de la bonne utilisation de la mémoire, nous allons reprendre l'exemple de la multiplication matricielle tiré de [60]. Dans le code 2.1, nous avons présenté l'implémentation naïve. Un des problèmes avec cette implémentation est la lecture de la

matrice A. En effet, à chaque itération i, tous les threads d'un warp lisent la même valeur depuis la mémoire globale. Cette lecture est effectuée en une seule transaction mémoire. En revanche, elle n'utilise pas toute la bande passante qui permet de lire davantage de mots en une transaction. C'est pourquoi, il est préférable que chaque thread copie les données de A, par des accès coalescents, vers la mémoire partagée. Les lectures se feront alors depuis la mémoire partagée, où les lectures moins adaptées pourront se faire en ayant un impact plus réduit sur les performances, voir le code 2.2.

```
__global__ void coalescentMatMul(float *a, float* b, float *c,
                                 int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM];

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i]* b[i*N+col];
    }
    c[row*N+col] = sum;
}
```

Code 2.2 – Produit de deux matrices en profitant de la coalescence

L'autre problème majeur est que la matrice B est lue plusieurs fois, sans précaution particulière. Il est intéressant une nouvelle fois de s'aider de la mémoire partagée pour effectuer les lectures répétées, voir le code 2.3.

```
__global__ void sharedABMultiply(float *a, float* b, float *c,
                                 int N)
{
    __shared__ float aTile[TILE_DIM][TILE_DIM],
                     bTile[TILE_DIM][TILE_DIM];
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;
    aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
    bTile[threadIdx.y][threadIdx.x] = b[threadIdx.y*N+col];
    __syncthreads();
    for (int i = 0; i < TILE_DIM; i++) {
        sum += aTile[threadIdx.y][i]* bTile[i][threadIdx.x];
    }
    c[row*N+col] = sum;
}
```

Code 2.3 – Produit de deux matrices en utilisant toute la bande passante

Maintenant que nous avons présenté des optimisations simples, il est important de mesurer leur impact sur les performances. Le test sur une Tesla M2090, nous donne une multiplication de la bande passante de 2.3 pour la première amélioration et 2.7 en ajoutant la deuxième.

OpenCL Il est proposé en 2008 comme standard de programmation pour architectures hétérogènes [58]. Il gère à la fois les CPU (IBM, Intel, AMD, VIA), les GPU (NVidia, AMD, S3), les DSP. Contrairement à CUDA qui ne gère que les cartes CUDA de NVidia, il présente le grand avantage d'être compatible avec plusieurs types d'accélérateurs.

Quand nous avons commencé nos travaux, les performances apportées par OpenCL sur cartes CUDA étaient moins importantes que celles offertes par le langage CUDA. Nous n'avons donc pas poussé l'étude sur ce langage, même si sa polyvalence lui donne un intérêt certain.

2.2.3.2 Bibliothèques de calcul

Nous avons vu combien il pouvait être difficile de programmer sur des GPU de façon optimisée. Heureusement, comme avec les CPU, nous trouvons des bibliothèques regroupant des opérations optimisées, facilitant le travail du programmeur. Il existe l'équivalent des BLAS pour CUDA, nommé CUBLAS. De même, nous disposons de la bibliothèque CUFFT qui correspond à la FFTW des GPU.

2.2.3.3 Génération de code

S'aider d'une bibliothèque de calcul est un choix très intéressant pour le programmeur lui évitant un travail fastidieux. Malheureusement, tous les traitements spécifiques à un programme ne s'écrivent pas obligatoirement comme un ensemble d'opérations connues et gérées. Cependant, certains outils sont capables de générer automatiquement les noyaux pour des accélérateurs, par exemple en utilisant des annotations. Nous pouvons citer HMPP [17], PGI Accelerators [42], ou encore OpenACC [65], la future extension d'OpenMP.

OpenACC est intéressant car c'est un standard de programmation parallèle développé par de nombreux industriels tels que CAPS, Cray, NVidia, PGI. Il se compose de directives de compilation pour le C, le C++ et le Fortran, à la manière d'OpenMP. Il est possible de définir des régions de code qui peuvent être exécutées sur un accélérateur au lieu d'utiliser le CPU, hôte de l'application. Son principal intérêt est de permettre une programmation haut-niveau en utilisant des accélérateurs, sans avoir explicitement à les initialiser, à transférer les données entre l'hôte et les accélérateurs. Il fonctionne avec les accélérateurs AMD, Intel et NVidia.

Le point noir avec l'approche par génération automatique est qu'il est difficile d'obtenir des noyaux performants, surtout avec les GPU n'étant pas de la dernière génération. Néanmoins, elle est très prometteuse, étant donnée la considérable diminution de travail à réaliser par le programmeur, et surtout grâce aux évolutions des architectures des accélérateurs qui la rend plus efficace.

2.2.4 Ordonnancement des tâches

Nous avons exposé les différentes méthodes pour programmer sur une machine hétérogène. Nous pouvons avoir un code qui comporte des portions exécutées sur des cœurs simples, d'autres sur des accélérateurs, et encore d'autres sur les deux. L'idéal serait d'occuper toutes nos ressources au maximum afin d'exploiter pleinement notre machine. L'ordonnanceur du système sur laquelle notre application tourne sait distribuer les tâches entre les cœurs – même si ce n'est pas toujours fait de façon optimisée – mais ne prend pas en charge les accélérateurs. Par conséquent, le programmeur doit effectuer la répartition optimale des tâches entre les ressources disponibles lui-même, et ceci avec tous les jeux de données d'entrée permis par son programme et sur toutes les machines. C'est là qu'interviennent nos ordonnanceurs de tâches, nous libérant de ce fastidieux travail. Nous nous intéressons seulement aux ordonnanceurs gérant les accélérateurs.

HMPP, en plus de la génération de code, nous donne la possibilité de gérer l'ordonnancement dynamique. Charm++ [49, 52] est une bibliothèque qui offre un ordonnancement basé sur la programmation objet (C++), grâce à des objets appelés *chares*, qui sont contrôlés par passage de messages. Kaapi [38], pour *Kernel for Adaptative Asynchronous Parallel and Interactive programming*, est une bibliothèque C++ permettant également l'ordonnancement de tâches grâces à des mécanismes de vol de travail et de partitionnement de graphes. Nous allons dans la suite présenter un autre ordonnanceur nommé StarPU, et voir ce qui le différencie des autres.

2.2.5 StarPU

C'est une bibliothèque développée par l'équipe Runtime d'Inria Bordeaux Sud-Ouest [8]. Elle fonctionne sur le modèle de programmation par tâche. En effet, la définition des multiples tâches du programme est laissée à la charge du programmeur. À chaque tâche doivent être associées une ou plusieurs implémentations pour des ressources de calcul. Le travail de StarPU a lieu au moment de l'exécution : il choisit quelle tâche doit être exécutée à quel moment et sur quelle ressource.

2.2.5.1 Fonctionnalités

Lorsque le programmeur soumet ses tâches à StarPU, celui-ci construit un graphe de dépendances automatiquement. Pour cela, il se base sur les données d'entrée et de sortie des tâches. Par exemple deux tâches qui prendront une seule et même entrée sans la modifier, et qui seront soumises à la suite, pourront être exécutées dans n'importe quel ordre indépendamment. Les dépendances peuvent également être spécifiées manuellement.

Grâce au graphe de tâches, StarPU, en suivant un algorithme d'ordonnancement, distribue les tâches entre les ressources de calcul. StarPU propose plusieurs algorithmes d'ordonnancement. Le plus simple utilise une liste globale de tâches avec ou non la gestion de priorités, ou le vol de travail. Mais il existe également des algorithmes avancés comme par exemple HEFT, pour *Heterogeneous Earliest Finish Time* [81]. Cette stratégie essaie de minimiser le temps de fin des tâches. Dans ce but, elle va estimer le temps d'exécution en incluant, si besoin, le temps de transfert des données.

Le temps de transfert est facile à prévoir. Pour le temps d'exécution, StarPU va utiliser soit un modèle de comportement de la fonction associée à la tâche, donné par le programmeur, ou alors il va le construire lors d'une étape de calibration. Une fois toutes les fonctions de l'application calibrées sur toutes les ressources de calcul, StarPU est capable de prévoir les temps de fin des tâches et prendre les meilleures décisions.

Afin de recouvrir les communications par des calculs, StarPU procède au transfert des données avant que la tâche les utilise. De plus, il sait gérer efficacement des tâches qui vont sommer des résultats en faisant un arbre de réduction.

StarPU s'exécute sur un seul nœud à la fois. Par conséquent, il ne permet pas l'ordonnancement de tâches entre plusieurs nœuds. Cependant, il offre des mécanismes permettant d'interagir avec MPI. Par exemple, il est capable de créer la requête MPI de réception d'une donnée, qu'il sait présente sur un autre nœud, ou alors débloquer une tâche lors de la réception ou l'envoi d'une donnée.

2.2.5.2 Utilisation

Afin de comprendre le travail à réaliser, nous allons entrevoir comment StarPU s'utilise. Dans ce but, reprenons l'exemple de la multiplication de deux matrices en étudiant le code 1.

Il est d'abord nécessaire de définir ce qui est appelé une *codelet* qui est une structure regroupant plusieurs types d'informations. Le premier concerne les implémentations grâce au champ *.where* (ligne 5, code 1) qui indique sur quelle ressource s'applique cette codelet, et aux champs *.cpu_funcs* et *cuda_funcs* qui donnent la liste des fonctions implémentées respectivement sur CPU et sur GPU. La deuxième information contenue dans la codelet concerne les données. Tout d'abord le nombre d'entrées/sorties de notre fonction est donné par le champ *nbuffers* à la ligne 6. Le champ *modes* permet quant à lui de savoir si nous manipulons une entrée ($STARPU\_R$ pour une lecture), une sortie ($STARPU\_W$ pour une écriture et $STARPU\_REDUX$ pour une réduction), ou une entrée-sortie ($STARPU\_RW$ pour une lecture-écriture).

Nous avons défini à la ligne 10, l'implémentation sur CPU comme un simple appel à la fonction *sgemm* définie dans les BLAS. De même, l'implémentation sur GPU n'est qu'un appel à CUBLAS (ligne 25). Nous remarquons que les deux implémentations ont le même prototype, standardisé. Le premier paramètre est un tableau d'interfaces servant à accéder à nos données manipulées dans la fonction, et récupérées grâce à $STARPU\_BLOCK\_GET\_PTR$. Le nombre de données manipulées est donné par le champ *nbuffers*. Le deuxième paramètre est un pointeur vers une liste d'arguments fixes.

Afin de manipuler des données, celles-ci doivent être enregistrées auprès de StarPU. Cette opération, faite aux lignes 33 à 35, a besoin de l'adresse de la zone à enregistrer et initialise un gestionnaire de données nommé *handle*. Il permet d'identifier une donnée, et de construire automatiquement les dépendances entre les tâches agissant sur cette même donnée.

La dernière étape est la création et la soumission de la tâche (ligne 37). Celle-ci a besoin de notre codelet et de nos handles. La soumission de la tâche est faite de manière asynchrone et le programme va pouvoir passer à la ligne 43 avant que la tâche ne soit terminée.

L'intégration de StarPU est simple, le plus dur est de déterminer les tâches qui composent notre application et d'écrire les différentes implémentations, plus couramment appelées noyaux.

2.3 Conclusion

Les travaux réalisés sur la méthode multipôle, en électromagnétisme, ne se sont pas encore focalisés sur son exécution utilisant toutes les ressources disponibles sur les machines de calcul actuelles, hétérogène, mais donnent de bonnes bases.

Nous avons montré que pour obtenir de bonnes performances, nous devons toujours utiliser des bibliothèques de calcul optimisées et adaptées aux processeurs cibles (CPU ou GPU) telles que les BLAS. Pour exploiter une machine hétérogène, un ordonnanceur semble être inévitable, à moins de parfaitement connaître son application et ceci quelque soit la machine cible. Il est donc utopique de créer l'application parfaite, qui s'adapte à toutes les machines sans employer des outils qui eux sauront s'adapter à des architectures variées.

Par conséquent, nous devons adapter l'algorithme de la méthode multipôle pour qu'il intègre l'utilisation d'un ordonnanceur dynamique avec des tâches optimisées.

```
static struct starpu_codelet matmul_codelet =
{
  .cpu_funcs = {matmul_cpu},
  .cuda_funcs = {matmul_gpu},
  .where = STARPU_CPU | STARPU_CUDA,
  .nbuffers = 3,
  .modes = {STARPU_W, STARPU_R, STARPU_R},
} ;

void matmul_cpu(void *buffers[], void *args)
{
  float *C = (float *)STARPU_MATRIX_GET_PTR(buffers[0]) ;
  float *A = (float *)STARPU_MATRIX_GET_PTR(buffers[1]) ;
  float *B = (float *)STARPU_MATRIX_GET_PTR(buffers[2]) ;
  int N = (int)STARPU_MATRIX_GET_NX(buffers[0]) ;

  sgemm(N, ..., A, ..., B, ..., C) ;
}

void matmul_gpu(void *buffers[], void *args)
{
  float *C = (complex float *)STARPU_MATRIX_GET_PTR(buffers[0]) ;
  float *A = (complex float *)STARPU_MATRIX_GET_PTR(buffers[1]) ;
  float *B = (complex float *)STARPU_MATRIX_GET_PTR(buffers[2]) ;
  int N = (int)STARPU_MATRIX_GET_NX(buffers[0]) ;

  cublasSgemm(N, ... A, ..., B, ..., C) ;
}
int main()
{
  ...

  starpu_matrix_data_register(&A_handle, ..., A, N, N ...) ;
  starpu_matrix_data_register(&B_handle, ..., B, N, N ...) ;
  starpu_matrix_data_register(&C_handle, ..., C, N, N ...) ;

  starpu_insert_task(&matmul_codelet, ...,
      C_handle, ..., // C
      A_handle, ..., // A
      B_handle, ..., // B
      ...) ;

  ...
}
```

Code 1 – Produit de deux matrices avec StarPU

Sommaire

Chapitre

3

Contribution

An algorithm must be seen to be believed.
Vol. I, Fundamental Algorithms, Section 1.1 (1968).

Notre objectif est d'implémenter la méthode multipôle sur une machine hétérogène le plus efficacement possible. Nous venons de voir les différents outils et techniques dont nous disposons. Nous allons dans ce chapitre donner et justifier nos choix techniques. Puis, nous montrerons comment nous avons adapté la méthode multipôle afin qu'elle réponde à nos spécifications, au niveau de l'organisation des données, de la gestion des communications et des étapes de calcul.

3.1 Choix d'implémentation

Les choix d'implémentation vont conditionner tout le fonctionnement de notre application. Ils concernent en premier lieu le langage de programmation utilisé, mais également les outils exploités.

3.1.1 Choix de langage

La première décision à prendre lors de l'implémentation d'un algorithme, est bien entendu le choix du langage de programmation. Dans notre cas, la méthode multipôle doit être intégrée à un solveur existant qui est écrit en Fortran. Or, la version de CUDA pour le Fortran est développée par PGI et elle est payante. De plus, les outils tels que les ordonnanceurs dynamiques que nous avons présentés ne sont pas disponibles en Fortran. Nous avons donc décidé d'utiliser un autre langage que le Fortran.

Afin de pouvoir s'interfacer assez aisément avec le Fortran, nous avons restreint notre choix aux C et C++. Le C++ a l'avantage d'être un langage objet et donc de permettre une conception plus organisée des entités à manipuler. Cependant, il est nécessaire d'utiliser les mécanismes offerts par le C++ avec parcimonie, de sorte à ne pas dégrader les performances de notre application. La méthode multipôle ne nécessite pas de structures de données complexes et multiples. Les mécanismes proposés par le C nous ont semblé suffisants, et ce langage est plus compréhensible, et donc maintenable, par des habitués du Fortran comparé au C++. Nous avons donc retenu le langage C pour nos travaux.

3.1.2 Utilisation d'outils pour la parallélisation

La méthode multipôle intègre de nombreux calculs de types différents. De plus, le nombre de boîtes et de directions est différent à chaque niveau, et le nombre de voisins proches et éloignés

varie suivant la boîte considérée. Les temps d'exécution sont donc différents suivant les niveaux, et suivant les boîtes. Une autre problématique réside dans le fait que les tâches se comporteront différemment selon le type de cœur qui l'exécute. Pour toutes ces raisons, nous pensons qu'un ordonnancement statique de la méthode multipôle performant est irréalisable sur une machine hétérogène.

Nous nous sommes alors orientés vers l'utilisation d'un ordonnanceur de tâches dynamique. Écrire un ordonnanceur de tâches dynamique pour la méthode multipôle, aurait quasiment été équivalent à écrire un ordonnanceur de tâches générique, pour les mêmes raisons qui font qu'un ordonnanceur statique serait inefficace. C'est pourquoi nous avons choisi d'utiliser un produit existant et performant.

L'ordonnanceur retenu a été StarPU, développé par Inria Bordeaux Sud-Ouest. Il fournit toutes les possibilités que nous recherchons à savoir la gestion des CPU et GPU, de multiples stratégies d'ordonnancement, le préchargement des données afin de recouvrir les communications, ainsi que l'exécution en mémoire distribuée par l'intermédiaire de la bibliothèque MPI.

Concernant la génération de code, nous avons préféré la mettre de côté pour le moment. En effet, nos noyaux de calcul sont assez simples et permettent d'être écrits à la main, et nous pensons que la génération automatique de code pour GPU ne donne pas (encore !) de résultats aussi performants. Cependant, afin de rester quand même le plus maintenable possible, nous utilisons au maximum des bibliothèques de calcul : BLAS, FFTW, CUBLAS, CUFFT.

3.2 Gestion des tâches

Un ordonnanceur dynamique manipule des tâches. C'est pourquoi, il est nécessaire d'écrire l'algorithme multipôle comme une succession de créations et exécutions de tâches. Pour le moment, nous nous situons dans le cadre d'un seul nœud en mémoire partagée, StarPU n'ordonnançant les tâches qu'en intra-nœud. Nous aborderons l'exécution en mémoire distribuée à la section 3.4.

Une tâche est une action réalisée sur des données, elle est donc composée d'une fonction et de données sur lesquelles appliquer cette fonction. Il nous faut alors découper l'algorithme de la méthode multipôle, en fonctions et en données associées. Nous n'allons pas aborder le problème de l'initialisation, les différentes étapes étant faciles à paralléliser, mais nous concentrer sur l'algorithme multipôle.

3.2.1 Choix des fonctions

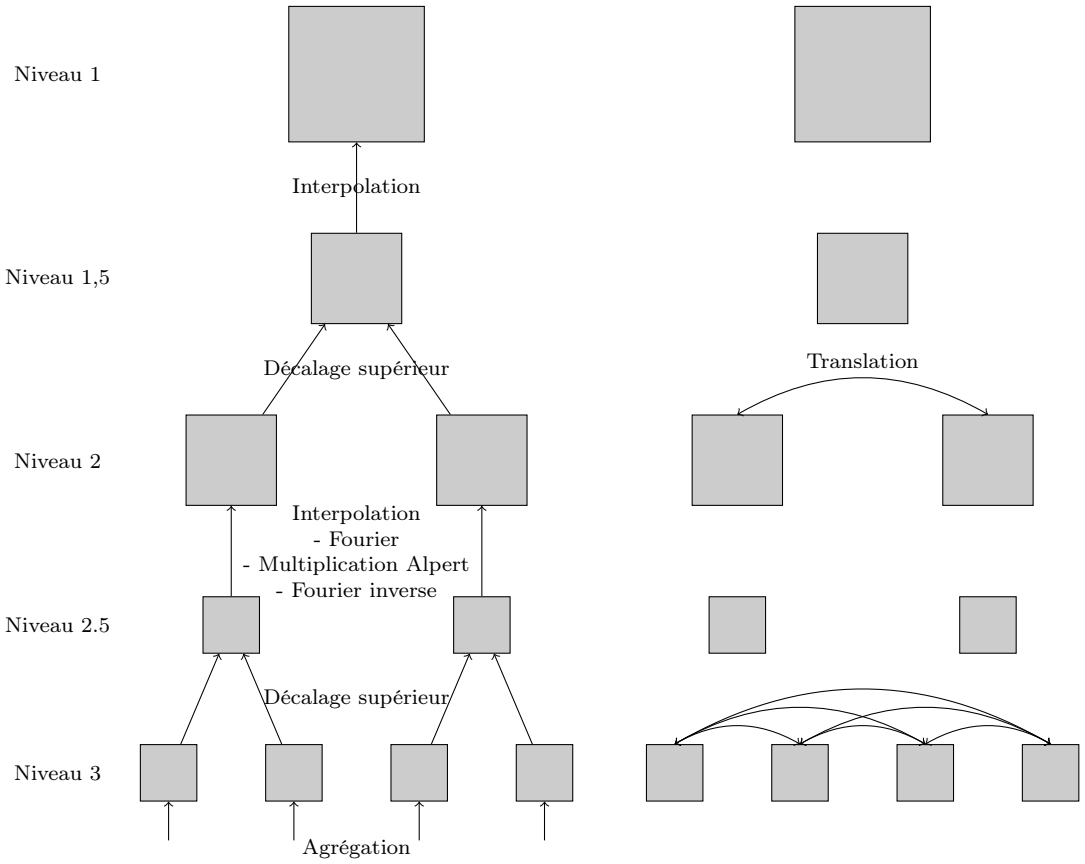
Afin de permettre à l'ordonnanceur de choisir au mieux le cœur le plus adapté à une tâche, il est nécessaire que le travail de la fonction soit ciblé, et ne concerne qu'un même type de calcul. En effet, il est plus intéressant de confier un produit de matrices à un GPU, qu'un simple produit scalaire. Il vaut mieux alors séparer ces deux types d'opérations en deux tâches afin de permettre éventuellement de réaliser les produits là où ils sont le plus performants.

Le découpage de l'algorithme multipôle en fonctions est assez aisé, et est issu naturellement des différentes étapes de la méthode multipôle : agrégation, translation, interpolation, décalage supérieur, anterpolation, décalage inférieur, désagrégation, et calcul des interactions proches. Les interpolations et anterpolations sont séparées en transformées de Fourier et la multiplication matricielle de la méthode d'Alpert.

Nous avons distingué l'interpolation du décalage supérieur (et l'anterpolation du décalage inférieur), car les calculs sont bien distincts. Pour marquer cette distinction, nous rajoutons des demi-niveaux où les boîtes sont issues du décalage supérieur, mais n'ont pas encore été interpolées. Nous avons synthétisé les tâches qui composent l'algorithme multipôle sur la figure 3.1. Elle représente l'ensemble des tâches lors du calcul des interactions lointaines de la méthode multipôle. Afin d'avoir l'algorithme complet, il est nécessaire d'y ajouter les interactions proches.

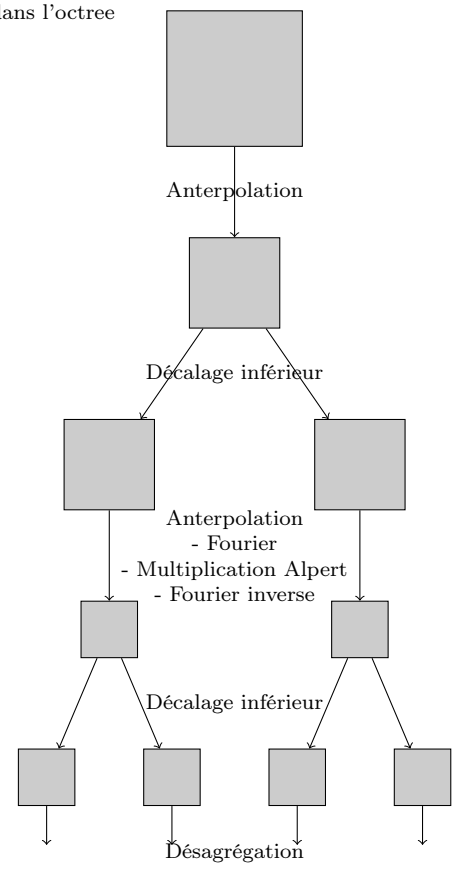
3.2.2 Découpage des données en blocs

Maintenant que nous avons choisi les fonctions, il s'agit d'y associer des données en créant des blocs. Le découpage des données est équivalent à la distribution des données. Comme nous l'avons



(a) Remontée dans l'octree

(b) Translations



(c) Descente dans l'octree

FIGURE 3.1 – Tâches de l'algorithme multipôle

vu à la section 2.1.1.3, la distribution la plus efficace est une distribution hiérarchique, à la fois suivant les boîtes, et suivant les directions. Nous faisons donc le même choix concernant la création des blocs.

3.2.2.1 Organisation des données

Pour obtenir de bonnes performances, il est indispensable d'organiser les données en mémoire de manière à profiter au maximum des caches, comme nous l'avons vu dans 2.2.1.2. Le premier facteur est la numérotation de Morton des boîtes, voir la section 2.1.2. L'autre paramètre consiste à grouper les données de nature similaire, nous devons donc préférer les structures de tableaux aux tableaux de structures, qui sont la plupart du temps à proscrire.

Comme nous répartissons nos données entre différentes tâches, il n'est pas nécessaire que toutes les données soient contigües, mais seulement les données d'une même tâche, c'est-à-dire dans un bloc de boîtes et de directions.

3.2.2.2 Création des blocs

Les blocs de données sont créés en groupant des boîtes et des directions. Plus nous descendons dans l'arbre, moins les blocs comportent de directions et plus ils comportent de boîtes.

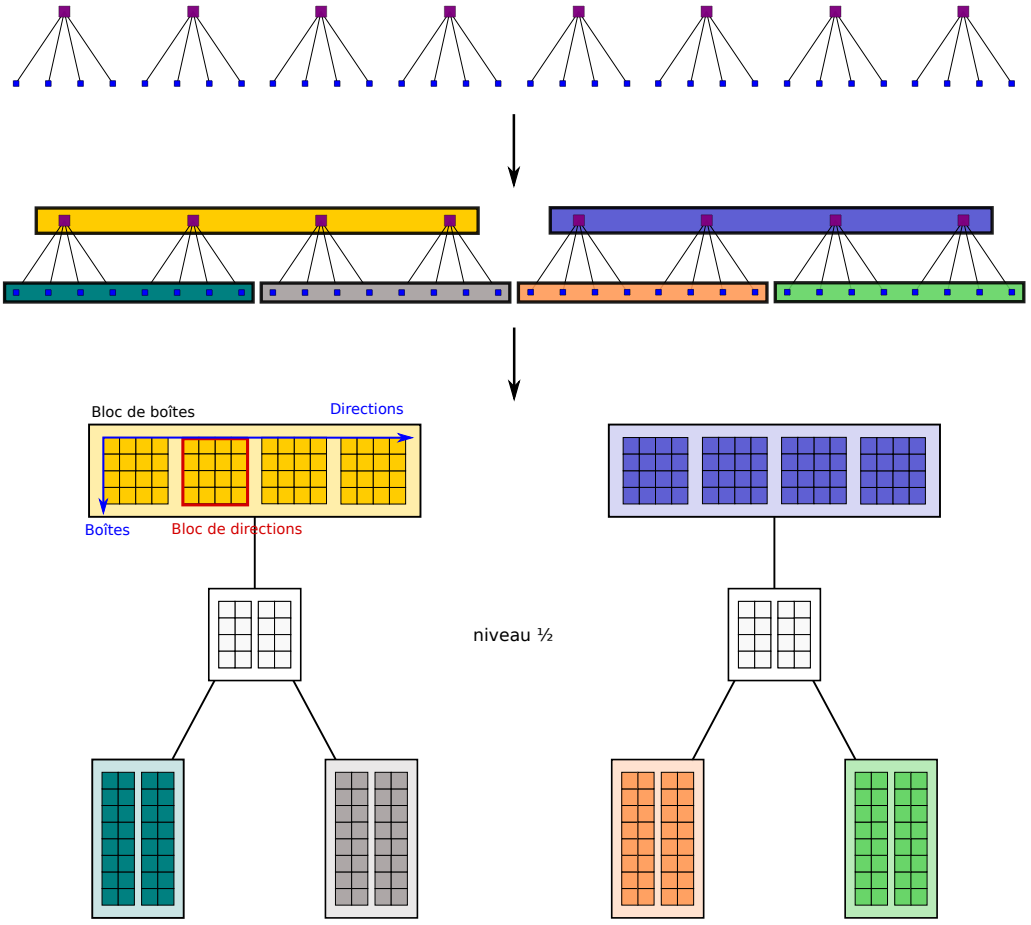


FIGURE 3.2 – Répartition des données en blocs

Nous allons détailler notre procédure de création des blocs. D'abord, nous partons du haut de l'arbre. Nous y groupons nos boîtes en blocs en veillant à ce que les fratries ne soient pas séparées en imposant un nombre minimal de boîtes par bloc. Nous poursuivons au niveau inférieur avec une stratégie identique, en imposant la contrainte que toutes les boîtes d'un bloc doivent avoir la même boîte parent. Ensuite, nous créons les blocs de directions dans les blocs de boîtes, en veillant

à garder toutes les directions ayant le même θ mais différents ϕ ensemble. Cette précaution nous permettra une gestion des interpolations (anterpolations) simplifiée. Nous avons également une limite minimale sur le nombre de directions par bloc de directions. Nous pouvons aussi fixer un nombre total minimal de blocs à avoir par niveau. La figure 3.2 nous montre ce découpage sur un arbre à deux niveaux.

Pour des raison d'efficacité, une tâche doit engendrer un nombre suffisant de calculs afin que le surcoût de son ordonnancement soit négligeable devant son temps d'exécution. Le nombre de données associées à une tâche doit alors être assez conséquent. Nous fixons un nombre de tâches par niveau en y adjoignant une contrainte sur les nombres minimaux de boîtes et de directions pour ne pas créer de tâches trop petites. Les tâches des niveaux plus bas contiennent davantage de boîtes que celles des niveaux plus hauts qui contiennent plus de directions.

L'ordonnanceur va confier l'exécution de ces tâches aux ressources de calcul disponibles qui semblent convenir le mieux selon la stratégie choisie et dans l'ordre qu'il aura déterminé. Il faut cependant qu'il respecte certaines dépendances entre les tâches.

3.2.3 Dépendance entre les tâches

La méthode multipôle est un algorithme basé sur un octree, il y a donc des dépendances entre les niveaux mais également au sein d'un niveau. Comme nous venons de le voir, nous travaillons sur des blocs de plusieurs boîtes et plusieurs directions. Regardons les différentes dépendances entre les blocs pour chaque étape de l'algorithme. Dans ce but, nous avons défini quelques notations. Dans les figures suivantes (de 3.3 à 3.10), les tâches sont représentées par des boîtes de la forme $\text{Tâche}(\text{bloc\_de\_boîtes}^{(\text{niveau})}, \text{bloc\_de\_directions}^{(\text{niveau})})$. Les blocs désignés sont les entrées des tâches spécifiées. Les dépendances sont représentées par des flèches, qui sont pleines quand les dépendances de leur destination sont toutes présentes, et en pointillés quand les dépendances de leur destination sont exprimées partiellement.

Nous tenons à rappeler que nous avons pris des notations qui peuvent, au premier abord, induire en erreur. En effet, la racine de l'arbre est le haut de l'arbre et nous notons ce niveau 1. À l'inverse, le bas de l'arbre, où se trouvent les feuilles, est noté $\pounds$. Donc, quand nous montons, nous passons du niveau l au niveau $l-1$ et inversement, la descente nous fait passer du niveau l au niveau $l+1$. Ces notations ont été choisies pour des soucis de généricité quand l'octree n'est pas équilibré.

3.2.3.1 Agrégation

L'étape d'agrégation n'a besoin que du vecteur d'entrée et des coordonnées des points. L'agrégation d'un bloc de boîtes pour un bloc de directions n'a par conséquent pas de dépendance, voir la figure 3.3.

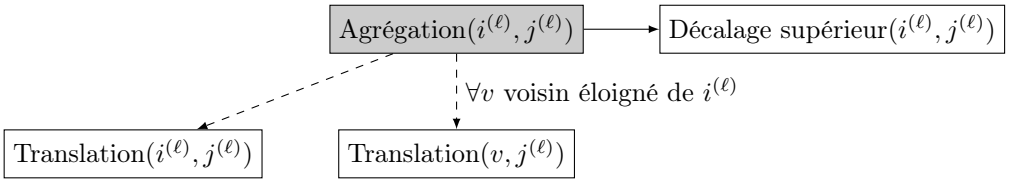


FIGURE 3.3 – Dépendances de l'agrégation

Il permet de débloquer un décalage supérieur et apporte une partie pour les translations de ses voisins éloignés (et lui-même).

3.2.3.2 Interpolation

Quand nous réalisons l'interpolation, nous cherchons à calculer le champ dans le niveau supérieur. Or, dans ce niveau, nous avons également des blocs de directions. Quand nous faisons une interpolation, nous avons alors besoin de connaître le bloc de directions supérieur qui sera la destination. L'interpolation est donc spécifique au bloc de directions de destination. Sur la figure 3.4,

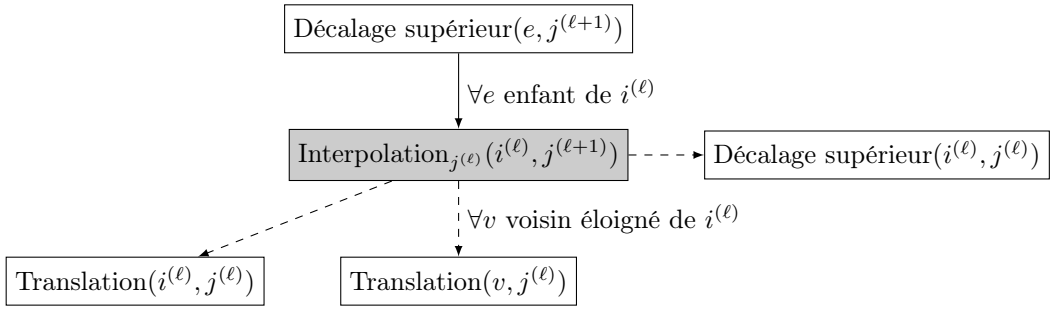


FIGURE 3.4 – Dépendances de l'interpolation

nous avons indicé l'interpolation par le bloc de destination cible, pour identifier l'interpolation concernée.

L'interpolation permet d'apporter une partie du champ nécessaire aux translations et au décalage supérieur.

3.2.3.3 Translation

La translation est l'opération qui a le plus de dépendances. Nous avons besoin du champ de tous les blocs contenant des boîtes voisines éloignées des boîtes de notre bloc. Nous nommons ces blocs, blocs voisins éloignés d'un bloc. Il est important de noter qu'un bloc fait partie de ses voisins éloignés. Ensuite, pour avoir le champ d'un voisin éloigné, il faut sommer les interpolations provenant de tous les blocs de directions du niveau précédent. Ces dépendances sont montrées sur la figure 3.5.

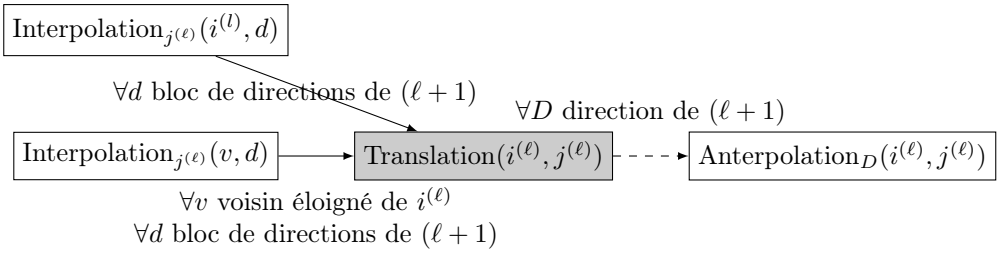


FIGURE 3.5 – Dépendances de la translation

La translation permet d'apporter une contribution qui sera utilisée par l'anterpolation lors de la descente.

3.2.3.4 Décalage supérieur

Le décalage supérieur nécessite d'avoir les champs des boîtes à décaler déjà calculés, c'est-à-dire, comme pour la translation, l'interpolation de tous les blocs de directions du niveau précédent. Le décalage supérieur apporte une contribution afin d'effectuer l'interpolation et obtenir le champ au niveau supérieur, voir la figure 3.6.

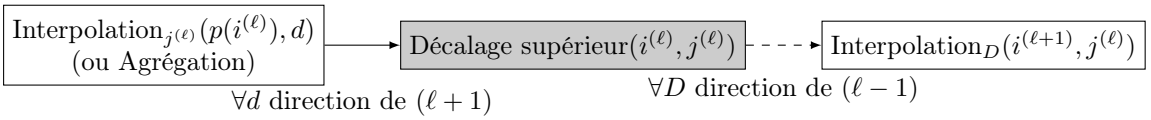


FIGURE 3.6 – Dépendances du décalage supérieur

3.2.3.5 Anterpolation

De la même manière que pour l'interpolation, l'anterpolation d'un bloc de directions ne donne qu'une partie du champ du bloc inférieur. Pour disposer du champ complet d'un bloc inférieur, nous avons besoin de sommer tous les blocs de directions du niveau supérieur, voir la figure 3.7.

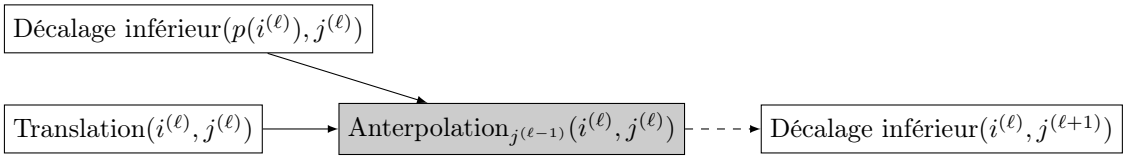


FIGURE 3.7 – Dépendances de l'anterpolation

3.2.3.6 Décalage inférieur

Le décalage inférieur utilise le résultat des anterpolations. Comme nous le montrons sur la figure 3.8, nous avons choisi de séparer les différents décalages inférieurs suivant les blocs fils permettant une plus grande souplesse dans l'ordonnancement.

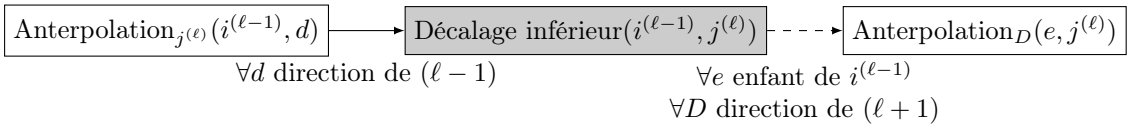


FIGURE 3.8 – Dépendances du décalage inférieur

3.2.3.7 Désagrégation

La désagrégation d'un bloc apporte une des contributions sommées pour obtenir le bloc résultat. Elle possède les mêmes dépendances que l'anterpolation, voir la figure 3.9.

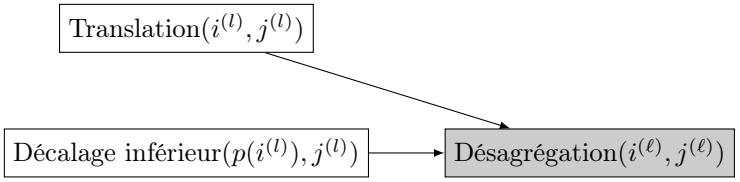


FIGURE 3.9 – Dépendances de la désagrégation

3.2.3.8 Interactions proches

Les interactions proches sont totalement indépendantes des autres opérations, voir la figure 3.10. Afin de calculer toutes les interactions proches des boîtes au sein d'un bloc, de la même manière que pour la translation, il est nécessaire d'avoir les valeurs des blocs voisins, ces valeurs étant disponibles en entrée de l'algorithme. Une tâche d'interactions proches implique un couple de blocs composé du bloc à calculer, et d'un de ses voisins. En conséquence, pour avoir toutes les interactions prises en compte, il faudra autant de tâches qu'il y a de voisins.

3.2.3.9 Ensemble des tâches

Nous avons représenté sur la figure 3.11 les dépendances entre les blocs de directions pour chaque type de tâche.

FIGURE 3.10 – Aucune dépendance pour les interactions proches

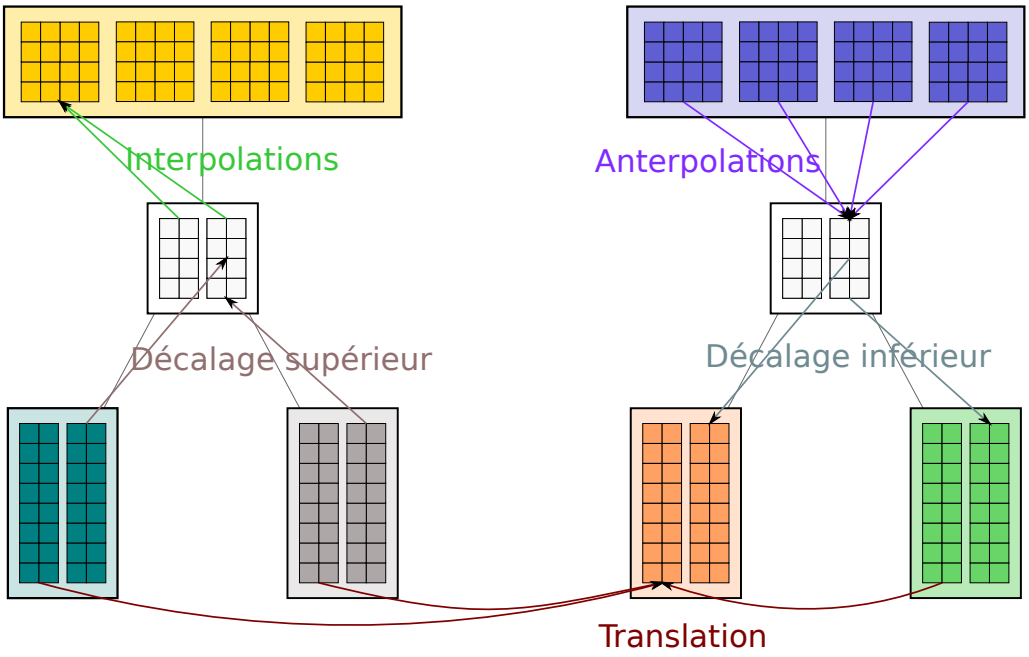


FIGURE 3.11 – Dépendance entre les blocs

3.3 Adaptation des tâches

Les tâches créées doivent avoir le moins de dépendances entre elles, pour avoir le plus de parallélisme possible. Les dépendances des tâches proviennent des données qu'elles manipulent, il est donc nécessaire d'avoir des données bien adaptées aux tâches.

3.3.1 Minimisation des dépendances de données

Nous n'allons pas ici développer les dépendances entre les blocs de données qui ont déjà été vues au travers des dépendances entre les tâches. Nous allons simplement étudier comment nous pouvons minimiser leur impact.

3.3.1.1 Les dépendances simples

Les tâches qui ne dépendent que d'un bloc et qui ne produisent qu'un bloc ne posent pas de problèmes et ne sont donc pas modifiées. C'est le cas de l'agrégation, des décalages inférieur et supérieur, et de la désagrégation.

La dépendance simple de ces tâches est triviale sauf concernant le décalage supérieur. Dans ce cas, la dépendance simple vient du fait que nous avons choisi de ne pas séparer les fratries entre plusieurs blocs. Un seul bloc enfant va donc contribuer au calcul du bloc parent. Comme notre tâche ne concerne qu'une partie (contigüe) du bloc de sortie, nous pouvons le partitionner. Ainsi, les tâches de la même fratrie agiront sur des partitions distinctes du bloc et seront alors indépendantes entre elles, voir la figure 3.12.

Avec le décalage inférieur, pour limiter les données concernées aux simples données utiles, nous partitionnons également le bloc d'entrée.

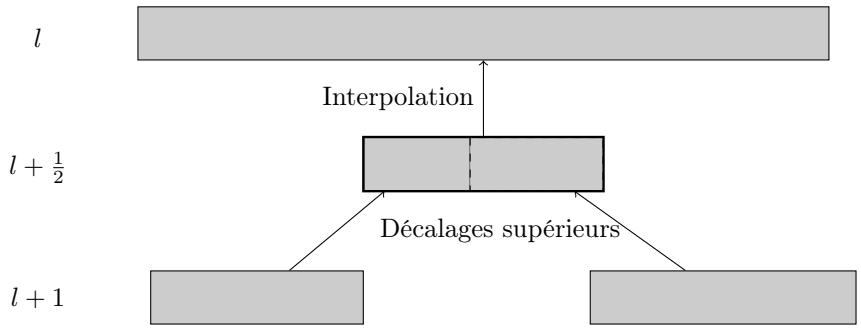


FIGURE 3.12 – Dépendances entre les blocs de boîtes lors du décalage supérieur

3.3.1.2 Les dépendances par réduction

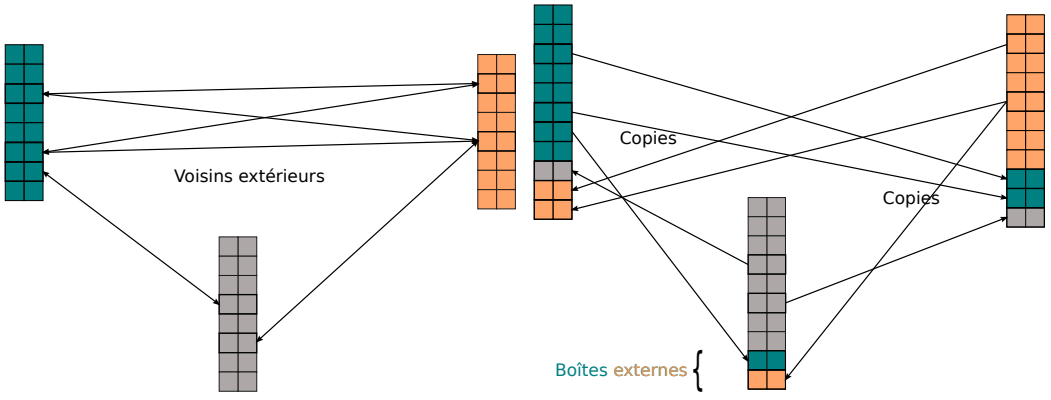
Certaines tâches ne permettent pas directement de calculer la valeur d'un bloc mais y participent, dans le sens où il est nécessaire de sommer le résultat de toutes les tâches qui agissent sur le bloc de sortie afin d'avoir sa valeur. Les tâches qui se trouvent dans cette configuration sont l'interpolation et l'anterpolation. La réduction se fait suivant les blocs de directions, car ces étapes nécessitent d'avoir toutes les directions.

Cependant, cette dépendance ne pose pas de réel problème. En effet, l'ordonnanceur que nous avons choisi sait traiter les réductions sur les données. Ainsi, l'impact sur les performances sera assez faible.

3.3.1.3 Les dépendances provenant de plusieurs entrées

Dans ce que nous venons de voir, nous avions toujours un bloc (ou une somme de blocs) en entrée qui contribuait à un bloc en sortie. Ce n'est plus le cas lors du calcul d'interactions entre les boîtes que ce soit pour le calcul des interactions proches ou pour les translations.

Translations La translation est l'opération qui concerne le plus de protagonistes. Afin de calculer tous les champs d'un bloc, il faut tous les voisins éloignés de toutes les boîtes du bloc. Grâce à la numérotation de Morton, la plupart des voisins éloignés sont contenus dans le bloc. Dans la figure 3.13a, nous pouvons voir un exemple de dépendances entre les blocs d'un même niveau lors de la translation.



(a) Besoin des voisins éloignés externes au bloc (b) Copies des voisins externes au bloc

FIGURE 3.13 – Une translation impliquant trois blocs de boîtes

Cette dépendance va être problématique surtout lors du calcul sur GPU où il faudra copier tous les blocs dont dépend un bloc au moment de la translation pour réaliser cette opération. Afin de résoudre ce problème, nous avons décidé de copier les voisins éloignés des boîtes d'un

bloc, externes à ce bloc. La figure 3.13b montre l'étape de copie supplémentaire à réaliser avant d'effectuer la translation, qui aura alors simplement besoin des données sur le bloc concerné. Le surcoût mémoire de cette copie ne doit pas représenter beaucoup par rapport à la taille des blocs, grâce à la numérotation choisie et si les blocs sont de taille suffisante.

Notre nouveau bloc est donc composé de plusieurs parties : la première contient les champs locaux, et les suivantes sont les copies des champs nécessaires provenant des blocs voisins, séparées par bloc source.

L'autre avantage de cette méthode est que l'algorithme de translation ne fait plus de distinction entre les données externes et internes, puisque tous les voisins éloignés sont maintenant internes. Cette régularité dans l'algorithme est encore plus intéressante avec les GPU.

Interactions proches Le calcul des interactions proches nécessite d'avoir en entrée tous les blocs voisins du bloc cible. Nous faisons alors comme avec la translation, et copions tous les voisins des boîtes du bloc, externes au bloc. Cette opération permet de s'affranchir de gérer plusieurs blocs dont le nombre de données utiles serait assez faible.

3.3.2 Synthèse des dépendances

Suite aux modifications apportées, les dépendances sont désormais efficacement prises en compte par l'ordonnanceur, voir la figure 3.14. L'intégration avec l'ordonnanceur sera donc aisée et celui-ci aura une grande liberté d'ordonnancement.

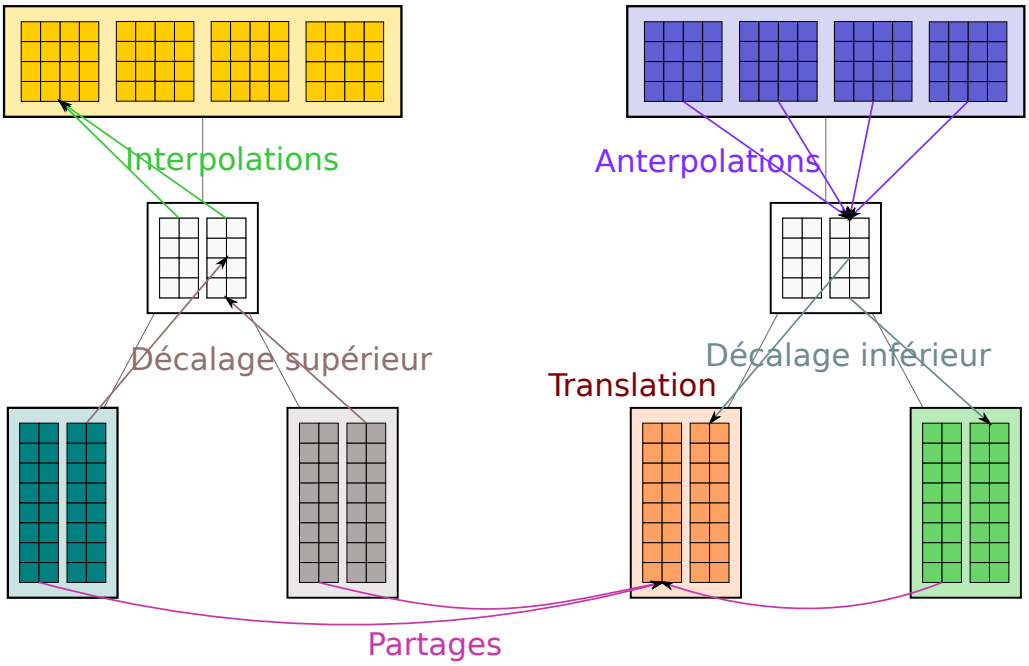


FIGURE 3.14 – Nouveau schéma de dépendance entre les blocs

3.4 Utilisation en mémoire distribuée

Notre machine cible est une machine hétérogène, dont la mémoire globale est distribuée entre les nœuds. StarPU ne gère pas implicitement et directement la mémoire distribuée entre les nœuds contrairement au cas intra-nœud où les données sont automatiquement copiées vers le GPU par exemple.

C'est pourquoi, même si StarPU prend en compte les communications réalisées à l'aide de la bibliothèque MPI lors de sa gestion des dépendances, comme nous l'avons vu à la section 2.2.5, nous devons distribuer les données manuellement et gérer explicitement les communications. Les

communications entre nœuds sont lentes et donc à éviter, ou à recouvrir par des calculs. Nous allons voir comment nous limitons les transferts nécessaires et comment les opérations ont été adaptées.

3.4.1 Distribution des données

Les données doivent être réparties manuellement entre les différents nœuds de notre machine. De ce fait, nous avons appliqué la distribution MLFMA-FFT [56]. Une distribution statique permet de minimiser le nombre de communications car les blocs qui s'échangent des messages ont des numéros la plupart du temps proche.

Nous partons du haut de l'arbre, et nous recherchons le premier niveau qui comporte « assez » de boîtes pour les distribuer de manière équilibrée entre les nœuds. À ce niveau, nous découpons l'arbre en sous-arbres que nous distribuons entre les nœuds. Pour le niveau supérieur, qui sera appelé le niveau 1, nous faisons une distribution simplement par blocs de directions.

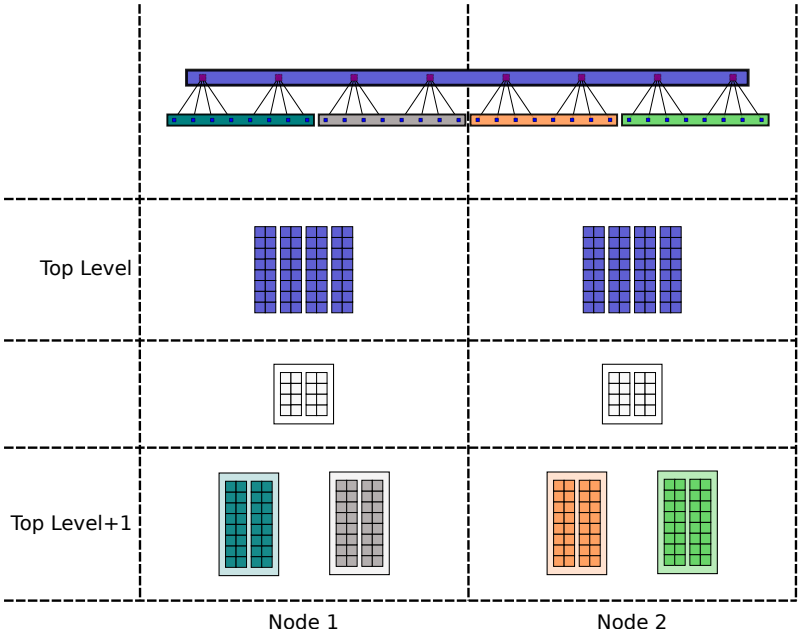


FIGURE 3.15 – Distribution des calculs

En ne distribuant que les boîtes entre les nœuds et non les directions, nous supprimons les communications entre les nœuds lors des interpolations (anterpolations) au détriment des communications lors des translations, qui augmentent alors. Nous avons fait ce choix car les interpolations sont des tâches qui permettent de créer du parallélisme, en débloquant le niveau supérieur de l'arbre. Au contraire, les translations sont disponibles rapidement, au moment de la phase de montée, mais ne sont nécessaires que lors de la phase de descente. Il y a donc plus de flexibilité pour leur exécution et les communications ont le temps d'être recouvertes. Par conséquent, il est important de favoriser les interpolations, et ne pas distribuer les directions entre les nœuds implique d'avoir exclusivement des dépendances locales.

3.4.2 Équilibrage de charge

Le calcul du premier niveau nécessitera les contributions de tous les nœuds, ce qui entraine une synchronisation. Par conséquent, il est important que les nœuds aient un nombre équivalent de calculs pendant la remontée avant d'accéder au premier niveau. Nous allons donc voir comment nous pouvons réduire l'impact de cette synchronisation.

3.4.2.1 Masquer les attentes

Nous avons à notre disposition les interactions proches, qui n'ont aucune dépendance et peuvent ainsi être exécutées à n'importe quel moment de l'algorithme. En imposant une priorité basse sur l'exécution de ces tâches, elles seront exécutées seulement quand il n'y a pas de tâche disponible. Cette situation se produit quand un nœud est en attente de communications, ou à la fin de la descente quand il n'y a plus de tâches autres à exécuter. Elle peut aussi avoir lieu simplement en cas d'attente intra-nœud, quand il n'y a plus assez de parallélisme et que des unités de calcul attendent le résultat d'autres.

Afin d'augmenter la capacité de ces interactions proches à masquer les déséquilibres entre les nœuds, nous ne procédons pas à leur distribution de manière statique. Chaque nœud a à sa disposition une réserve d'interactions proches à calculer, et quand cette réserve est faible, il doit en demander d'autres, à un processus qui sera chargé de la distribution. Le coût supplémentaire de la demande et de la réponse est faible, seul un intervalle d'interactions proches à calculer est transmis. Ainsi, les nœuds qui auront le moins de calculs à effectuer, au lieu d'attendre, effectueront davantage de calculs d'interactions proches.

Avec cette stratégie, nous espérons minimiser les défauts des déséquilibres dus à la distribution. Cependant, les interactions proches ne sont pas une ressource inépuisable, il est donc primordial d'équilibrer au maximum le nombre total de calculs entre les nœuds.

3.4.2.2 Coûts des sous-arbres

Une première idée pourrait être d'associer un poids à chaque boîte représentant le coût de calcul de son champ. Cependant, comme nous le verrons aux sections 4.2 et 4.3.2, le temps d'exécution des opérations sur une tâche dépend de plusieurs paramètres, notamment le type d'unité de calcul et la taille du bloc. Même le coût en nombre d'opérations flottantes donne une estimation très peu précise, car elle ne prend pas en compte l'efficacité de la tâche sur la ressource de calcul concernée.

Cependant, il existe une solution facile qui nous donne une estimation assez représentative du temps de chaque sous-arbre et triviale à mettre en œuvre : compter le nombre de feuilles de chaque sous-arbre. Cette solution n'est pas exacte comme le nombre de voisins éloignés peut être différents selon les boîtes, mais elle nous donne une assez bonne approximation malgré tout, surtout quand l'objet à étudier a une géométrie régulière. Il est à noter que cette solution n'est pas applicable à la méthode multipôle dite adaptative, dans laquelle la profondeur de l'arbre dépend du nombre de points par boîte. Dans ce cas, une meilleure solution serait la résolution d'un système linéaire construit à partir des estimations des temps des tâches, pour lequel nous chercherions à minimiser le temps total d'exécution.

3.4.2.3 Application de l'équilibrage de charge

Comme nous ne distribuons pas les directions entre les nœuds – à part pour le premier niveau – il nous suffit de répartir équitablement les boîtes du niveau à distribuer entre les nœuds. Dans ce but, nous calculons d'abord, le coût total du niveau, en sommant tous les coûts de ses boîtes. Puis, nous calculons le coût moyen à attribuer à un nœud, et nous distribuons les boîtes vers ce nœud jusqu'à ce que nous soyons au dessus du nombre moyen. Nous choisissons alors de garder ou non la dernière boîte donnée au nœud selon que le coût est plus proche de la moyenne ou non. Nous pouvons alors passer au nœud suivant pour lequel nous appliquons le même algorithme en recalculant la moyenne sur le coût restant. Nous procédons de cette manière en recalculant la moyenne à chaque nœud, afin de réduire les écarts entre les nœuds, comme sur l'exemple suivant 3.1.

Cet exemple nous montre qu'avec l'algorithme naïf, le déséquilibre de charge est plus important. Avec l'algorithme avancé, l'équilibrage de charge, bien que meilleur, reste assez médiocre. Cependant, c'est le mieux que nous puissions faire en gardant une distribution contigüe qui permet de limiter les communications.

3.4.2.4 Tenir compte de la topologie

Pour limiter l'impact négatif des communications, il est important de favoriser les communications sur des nœuds proches. Nous n'utilisons pas d'outil spécifique pour gérer les communications

Boîte	1	2	3	4	5	6	7
Coût	20	15	10	10	10	13	24
Méthode naïve							
Nœud	1		2		3		4
Charge	20		25		20		37
Méthode avancée							
Nœud	1		2		3		4
Charge	20		25		33		24

TABLE 3.1 – Impact sur la distribution suivant l'algorithme choisi

MPI en mémoire distribuée. Or, MPI ne connait pas la topologie de la machine sur laquelle sont exécutés les processus. Il n'est donc pas possible d'avoir des affinités particulières entre processus pour lesquels nous savons qu'il y a beaucoup d'échanges.

Cependant, grâce au fichier contenant la liste des nœuds, passé à *mpirun*, il est possible de choisir sur quel nœud sera exécuté un processus d'un rang donné. Il nous reste alors à connaître la topologie de la machine. Pour cela, nous utilisons *Hwloc* [15], qui réalise cette tâche parfaitement.

Ce qui nous intéresse, c'est d'avoir les processus de rang proche qui ont les coûts de communication les plus faibles, car ils sont susceptibles de communiquer entre eux. Nous associons alors les nœuds topologiquement proches à des rangs consécutifs. Ce travail est réalisé avant le lancement du programme, par le script de lancement.

3.4.3 Gestion des communications

Une bonne gestion des communications est le point critique d'une bonne parallélisation. Elles sont lentes – comparées aux accès en mémoire partagée – et il est important de limiter leur nombre, afin de ne pas surcharger le réseau pour ne pas les ralentir encore plus. Les seules opérations qui impliquent plusieurs nœuds sont les copies partielles nécessaires aux translations, ainsi que la réunion des sous-arbres au sommet.

3.4.3.1 Échanges au sommet de l'arbre

Afin de limiter les échanges à ce niveau, la distribution n'est faite que suivant les directions. Quand nous passons du niveau intermédiaire au premier niveau, voir la figure 3.16, les interpolations nécessitent des données de tous les nœuds. De plus, les données distribuées sont utilisées plusieurs fois par les nœuds.

Pour éviter une redondance des communications et une modification de la fonction d'interpolation, nous avons décidé d'utiliser un niveau fictif qui servira de relai vers le niveau sommet, voir la figure 3.17. Ce niveau supplémentaire représente le niveau sommet, mais distribué par blocs de boîtes conformément au sous-arbre attribué au nœud. Cela nous permet d'avoir la même fonction d'interpolation que pour les autres niveaux. Nous devons simplement ajouter une étape de copie qui permettra la redistribution du niveau.

3.4.3.2 Translation

L'opération critique au niveau des communications entre les nœuds est la translation. En effet, elle peut impliquer des blocs ne se trouvant pas localement et donc entraîner des communications entre les nœuds. Comme nous pouvons le voir sur la figure 3.18a, l'étape de translation implique beaucoup de communications entre les nœuds. Afin de s'affranchir de ce problème, nous groupons toutes les communications entre deux nœuds en une seule. Pour cela, nous créons un bloc supplémentaire qui servira de relai pour les communications MPI. Sur la figure 3.18b, les blocs rouges contiennent toutes les données qui doivent être échangées, et donc à la fois les données reçues et les données envoyées. Un bloc de communication est rajouté pour chaque nœud avec lequel il y a une communication.

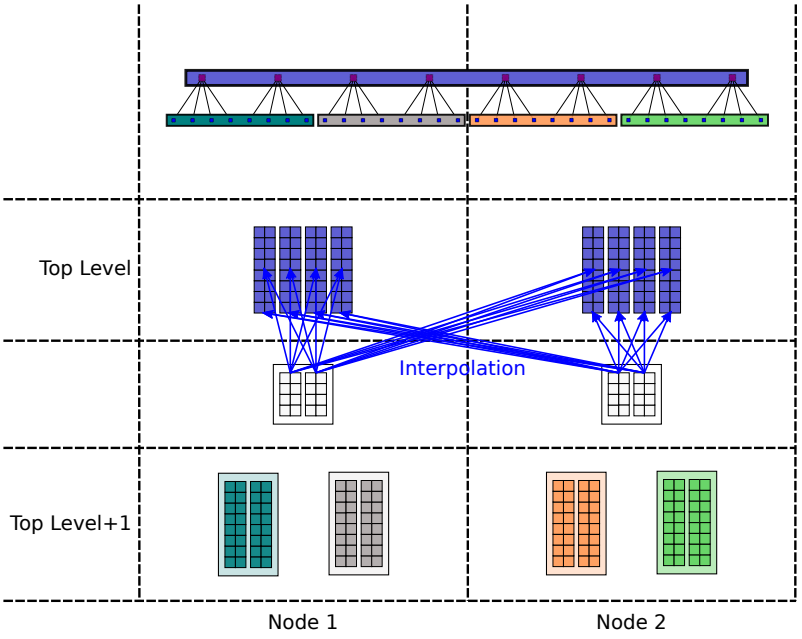


FIGURE 3.16 – Échanges lors de l'interpolation pour le premier niveau

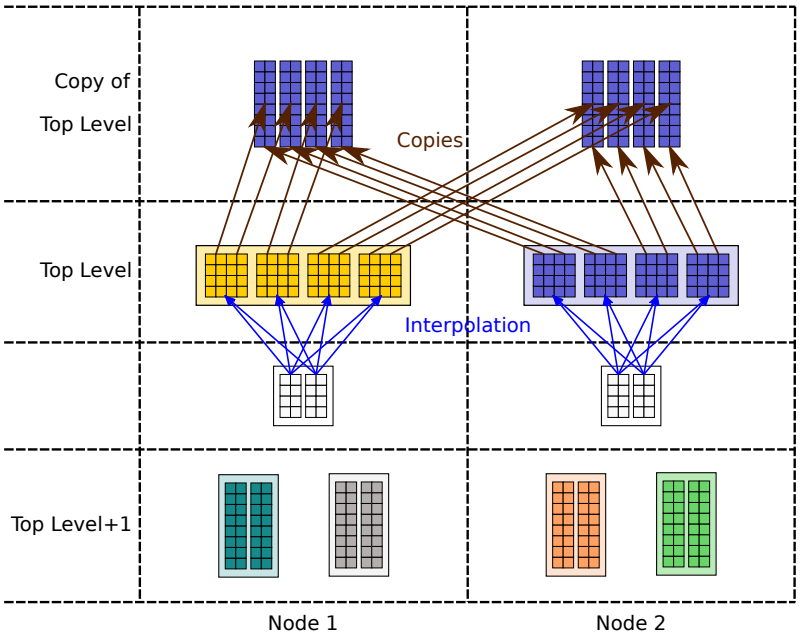
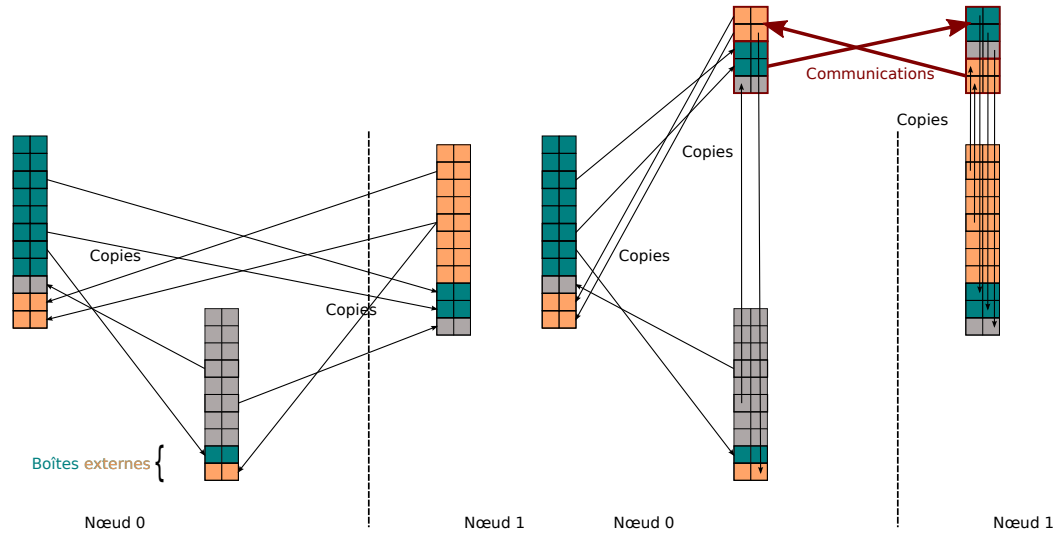


FIGURE 3.17 – Échanges avec le niveau supplémentaire



(a) Étape de translation entre les blocs d'un niveau distribué sur deux nœuds

(b) Copies des voisins externes au bloc

FIGURE 3.18 – Les translations en mémoire distribuée entre 3 blocs de boîtes

À l'étape des partages, nous avons alors à copier des données vers les blocs MPI, mais également à copier depuis les blocs MPI vers les blocs voisins éloignés. Comme pour les autres blocs, les blocs MPI sont séparés en plusieurs parties : les champs locaux au bloc et les différentes copies des blocs voisins. La partie concernant les champs locaux est la copie des boîtes d'un nœud voisin nécessaires au nœud courant, elle est donc issue d'une communication avec un autre nœud. Les parties suivantes sont des copies de champs locaux, et vont être communiquées au nœud distant.

3.4.3.3 Synthèse des communications entre les nœuds

Finalement, il y a deux types de communications, voir la figure 3.19. La première se produit avec tous les niveaux excepté le niveau sommet. C'est l'échange de données pour l'étape de translation. L'avantage de cette communication est qu'elle peut être recouverte facilement par des calculs. En effet, les données sont prêtes lors de la phase de montée mais peuvent être utilisées plus tard lors de la phase de descente lorsque qu'elles sont additionnées au résultat de l'anterpolation. La deuxième étape, qui concerne exclusivement le niveau sommet, est l'échange entre ce niveau et le niveau relai qui a été ajouté.

3.5 Utilisation de StarPU

Maintenant que nous avons vu comment nous créons les tâches de la méthode multipôle, nous allons détailler leur intégration avec la bibliothèque StarPU. Pour cela, nous allons étudier l'enregistrement des données auprès de StarPU, puis, comment sont gérées les dépendances.

3.5.1 Gestion des données

StarPU a besoin de connaître les données sur lesquelles les tâches vont agir. Les données doivent être enregistrées auprès de StarPU en créant un gestionnaire qui permettra d'identifier une donnée sur laquelle les tâches vont agir.

Pour la méthode multipôle, nous avons besoin d'agir principalement sur trois types de données : les champs successifs lors de la phase de montée, ceux lors de la phase de descente, et les interactions proches. Nous n'allons pas évoquer la gestion des données auxiliaires telles que les matrices de translation ou d'interpolation car celles-ci sont gérées « trivialement ».

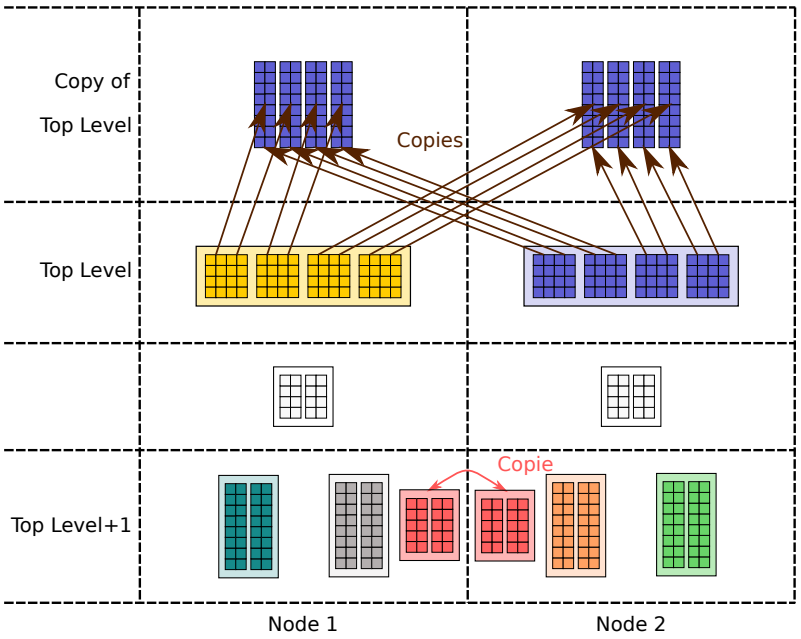


FIGURE 3.19 – Communications entre les nœuds par l'algorithme multipôle

3.5.1.1 Les décalages

Nous avons vu à la section 3.3.1.1 que les décalages nécessitaient de partitionner les données. Nous donnons alors à StarPU un filtre de partitionnement correspondant à la relation de parenté afin que chaque sous-bloc corresponde à son bloc enfant.

3.5.1.2 Le champ de montée F

Le champ F d'un bloc est la sortie de l'interpolation avec le niveau enfant, et sert d'entrée pour le décalage supérieur vers le niveau parent. Nous avons vu précédemment que nous couplons le champ F d'un bloc avec les champs F partagés par les voisins éloignés externes, afin de faciliter la translation. Ces relations sont résumées sur la figure 3.20 sur laquelle nous voyons le champ Fe qui est le couplage de F et des partages. Les flèches étiquetées par un W désignent les tâches qui écrivent la donnée représentée dans un cadre. Celles étiquetées par un R indique une lecture de la donnée par la tâche.

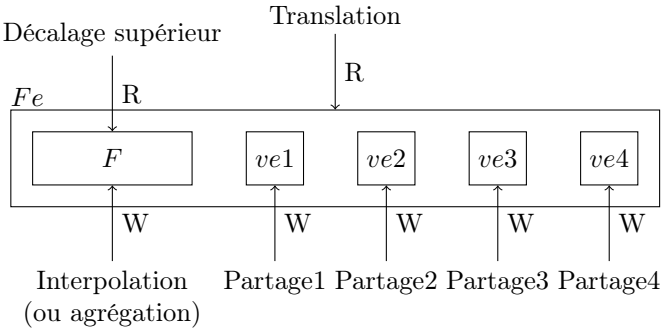


FIGURE 3.20 – Gestion des données lors de la montée

Pour avoir facilement Fe comme une réunion contigüe directement sans copie, nous le déclarons explicitement. Les données F, $ve1$, $ve2$, $ve3$, $ve4$ deviennent alors des sous-partitions de la donnée Fe. Le partitionnement de données est géré par StarPU. Pour cela, nous indiquons à StarPU un partitionnement pour obtenir nos sous-données, au moyen de ce qui est appelé un filtre de

partitionnement.

3.5.1.3 Le champ de descente N

Le champ N, calculé lors de la descente, est la somme des contributions provenant des voisins externes et donc de la translation, ainsi que celle provenant du parent du niveau supérieur issue du décalage inférieur. La lecture de cette donnée permet l'anterpolation. La figure 3.21 montre ces différents accès à la donnée.

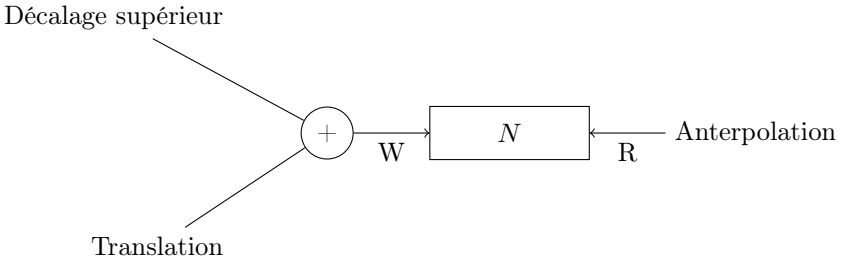


FIGURE 3.21 – Gestion des données lors de la descente

Le changement dans l'accès à N réside dans la somme entre la translation et le décalage inférieur. Ce mode d'accès est facile à prendre en compte dans StarPU qui possède un mode d'accès réservé à la réduction. Les tâches vont donc pouvoir accéder à la donnée de manière transparente et StarPU se chargera d'ajouter les données écrites dans N.

3.5.2 Gestion des dépendances

StarPU déduit les dépendances entre les tâches automatiquement grâce à deux critères : l'ordre de soumission des tâches et leurs modes d'accès aux données. Ces critères permettent d'ordonnancer correctement toutes nos tâches, excepté quand un départitionnement entre en jeu. L'autre point que nous allons étudier est la dépendance d'une tâche en lien avec une communication MPI.

3.5.2.1 Cas particulier du partitionnement

Dans StarPU, il est impossible de soumettre une tâche sur une donnée partitionnée, mais seulement sur ses partitions. De plus, le départitionnement se fait de manière synchrone. C'est pourquoi, pour ne pas introduire des barrières de synchronisation, nous ne pouvons soumettre le partitionnement lors de la phase de soumission globale des tâches. Nous allons étudier comment il est possible de contourner ce problème concernant trois données.

Les champs inter-niveaux Pour des soucis d'efficacité, nous avons vu qu'il était nécessaire de recourir au partitionnement lors des décalages. Lors de la phase de montée, notre bloc de l'inter-niveau est partitionné, et ensuite, les tâches écrivant les partitions sont soumises. La suite logique serait de soumettre l'interpolation du bloc complet, mais le problème est que ce bloc est encore partitionné. Il nous faut alors le départitionner, mais cette action nécessite la terminaison des tâches sur les partitions. Cela a comme conséquence de limiter l'intérêt de l'ordonnancement, voir la figure 3.22.

Nous avons alors décidé de ne pas départitionner ni de soumettre l'interpolation après avoir soumis le décalage supérieur, retardant cette opération lors de la fin des tâches sur les partitions. Dans ce but, nous avons ajouté une tâche qui dépend explicitement des tâches sur les partitions et qui s'exécute seulement lorsque la donnée peut être départitionnée. La tâche ajoutée ne peut faire directement le départitionnement ou la soumission, mais c'est possible en retour de tâche (*callback*). La figure 3.23 nous montre comment l'algorithme est modifié.

Afin que les tâches s'exécutent dans le bon ordre avec les dépendances correctes, il est nécessaire de préciser des dépendances manuellement. En effet, il ne faut pas que le décalage supérieur du niveau l puisse être effectué, si l'interpolation soumise en retour de l'interpolation modifiée n'est

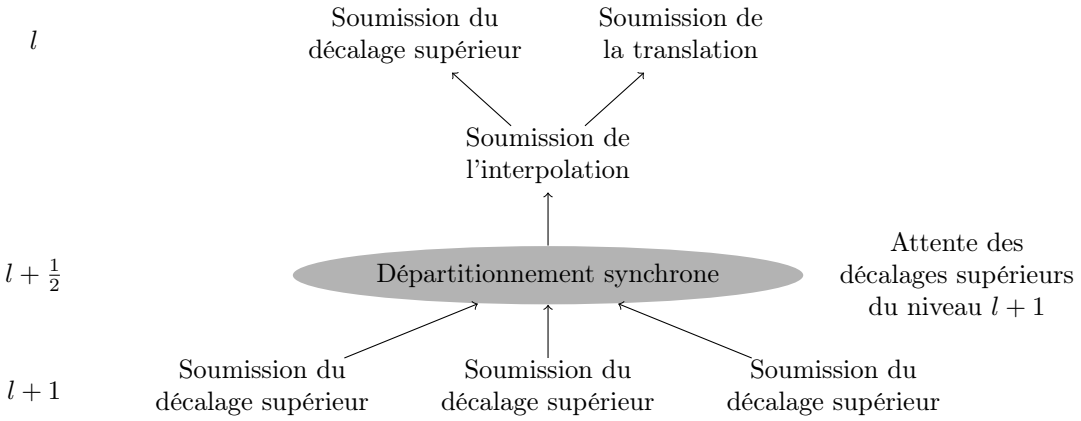


FIGURE 3.22 – Soumission des tâches lors de la montée locale

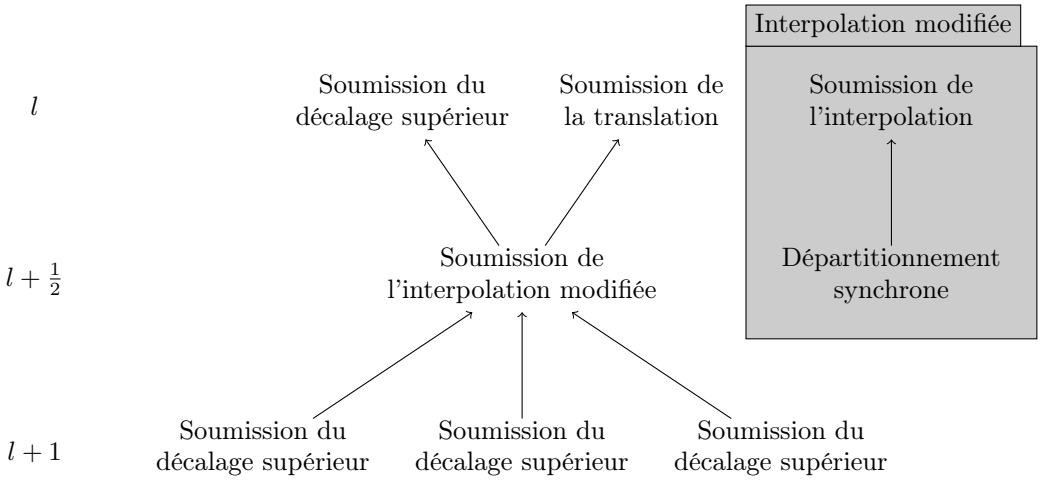


FIGURE 3.23 – Intégration de la tâche modifiée lors de la montée locale

pas terminée. Nous ajoutons alors une étiquette (*tag*) à l'interpolation et créons manuellement une dépendance sur cette étiquette, pour le décalage supérieur du niveau l. De même, l'interpolation modifiée ne doit pas s'exécuter avant la fin des décalages supérieurs, et il nous est difficile d'ajouter toutes les partitions comme données d'entrée. Comme précédemment, nous ajoutons alors les dépendances en utilisant des étiquettes. Il est alors nécessaire de supprimer les dépendances implicites sur les données, car il est possible que la tâche ne soit soumise en *callback* qu'après la soumission du décalage supérieur et de la translation qui en dépendent. Le nouveau schéma de dépendances se trouve sur la figure 3.24.

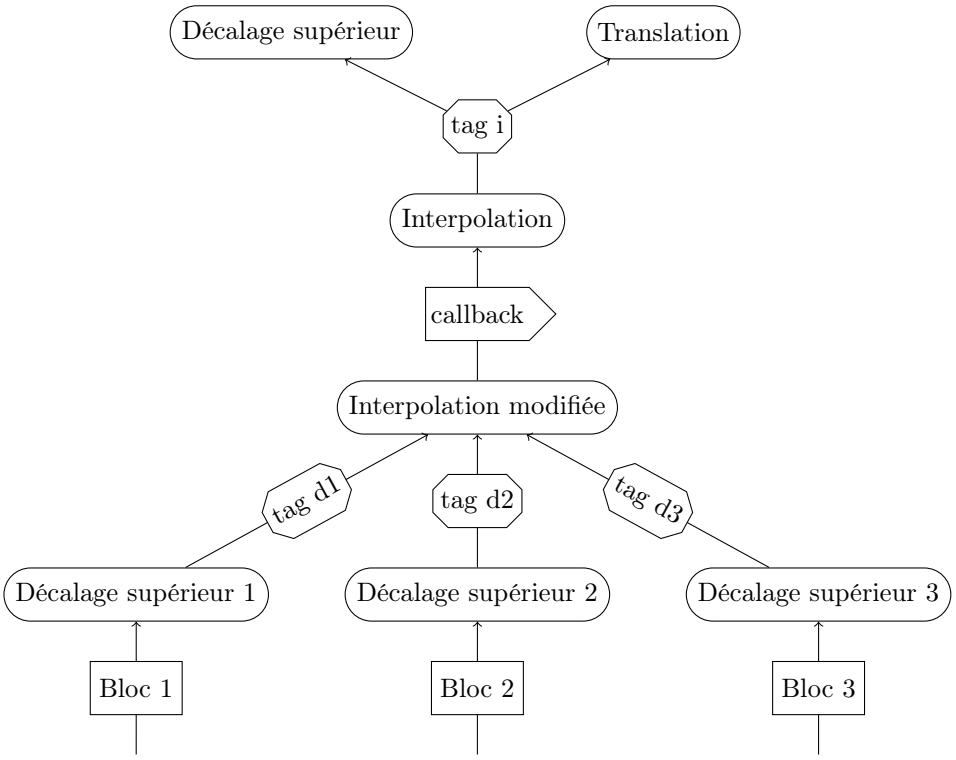


FIGURE 3.24 – Dépendances dans StarPU pour la montée locale

Le champ F et les voisins extérieurs Il s'agit de la donnée Fe vue sur la figure 3.20. Nous voyons que ses partitions proviennent de différentes tâches. Pour soumettre la translation, nous devons départitionner Fe, voir la figure 3.25, mais cela implique d'attendre la terminaison de l'interpolation et des partages, ce qui n'est pas bon pour l'ordonnancement.

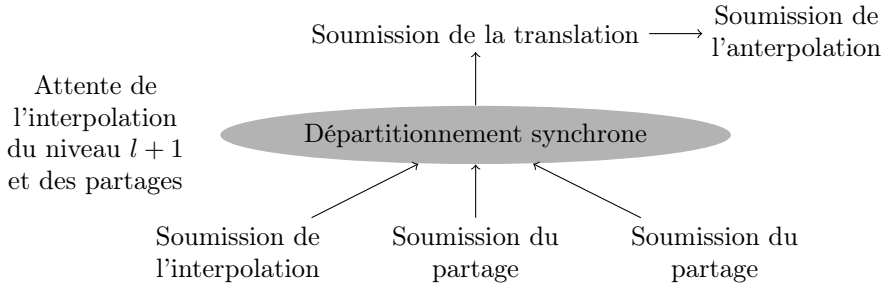


FIGURE 3.25 – Soumission des tâches lors de la translation

Afin de pallier ce problème, nous procédons comme pour les champs inter-niveaux. Nous ajoutons une tâche qui se chargera de faire le départitionnement et de soumettre la translation, quand

l'interpolation et les partages seront terminés. La figure 3.26 montre l'algorithme modifié.

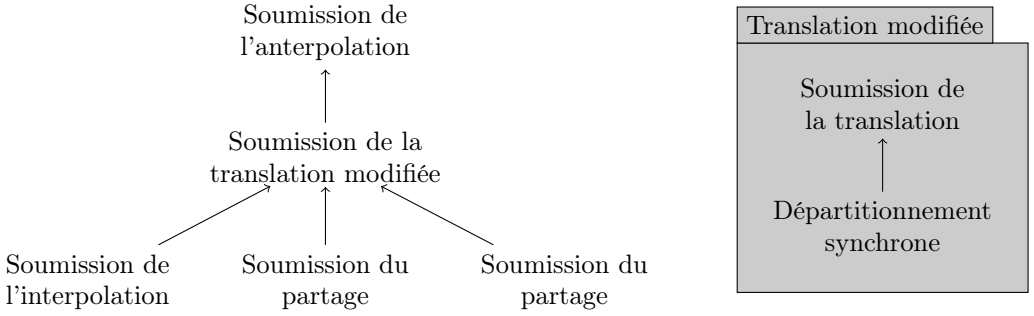


FIGURE 3.26 – Intégration de la translation modifiée

Nous devons alors imposer manuellement des dépendances au moyen d'étiquettes, voir la figure 3.27.

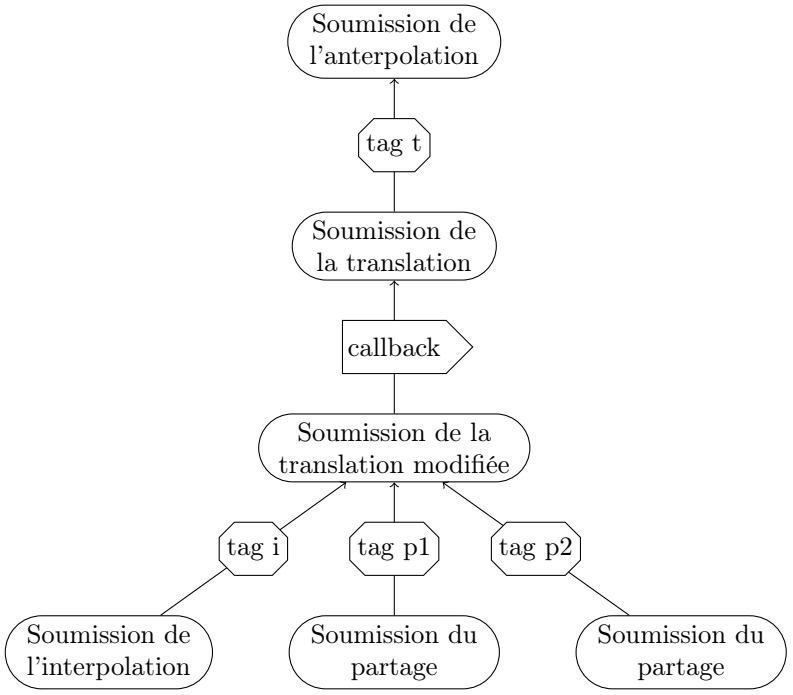


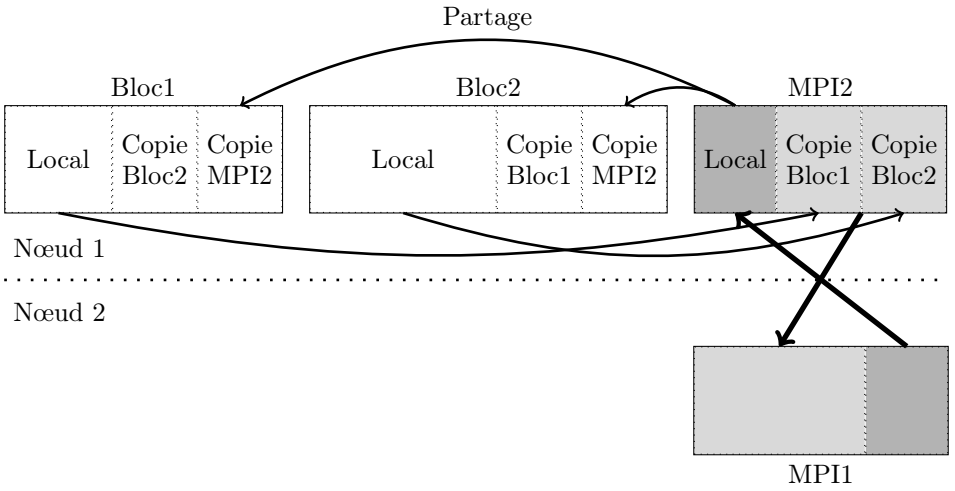
FIGURE 3.27 – Intégration de la translation modifiée

3.5.2.2 MPI

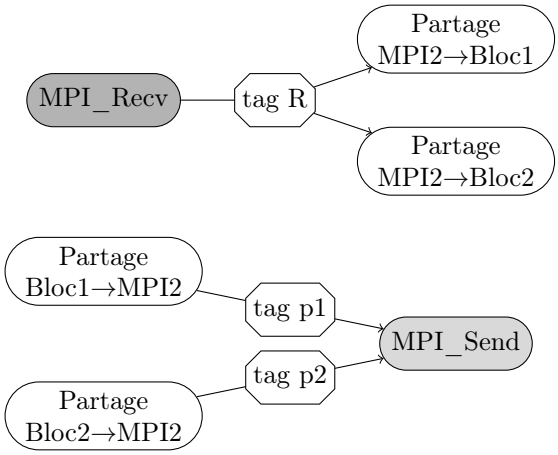
Comme nous l'avons montré à la figure 3.19, les tâches d'un nœud ont des dépendances provenant de communications MPI d'autres nœuds. Même si StarPU n'est pas capable d'ordonnancer les tâches sur plusieurs nœuds, il sait prendre en compte les communications MPI lors de ses décisions d'ordonnancement. Pour cela, il suffit d'utiliser les encapsulages des fonction de la bibliothèque MPI. Par exemple, au lieu d'utiliser la fonction *MPI_Irecv*, nous utilisons la fonction *starpu_mpi_irecv_detached*. Nous associons une étiquette aux communications MPI, qui sera activée à la terminaison de la communication MPI. Ainsi, nous ajoutons les bonnes dépendances aux tâches. Les communications se produisent pour les translations et pour le premier niveau.

Les Translations Nous devons attendre de recevoir les données dans le bloc MPI avant de les partager avec les blocs locaux. Inversement, nous devons attendre les partages des blocs locaux vers

le bloc MPI, avant de les envoyer au nœud destinataire. Ce mécanisme est résumé par la figure 3.28 sur laquelle nous avons représenté un nœud local $n1$ comportant deux blocs $B1$ et $B2$ et un bloc MPI *MPI2* correspondant au nœud $n2$.



(a) Échanges concernant le bloc MPI2



(b) Dépendances des tâches de communication MPI pour le nœud 1

FIGURE 3.28 – Gestion des communications lors des partages

3.5.3 Gestion de l'ordonnancement

Pour maximiser le recouvrement des communications par les calculs et éviter les attentes dues aux dépendances entre certaines tâches, il est indispensable de créer le maximum de parallélisme. D'après l'analyse des dépendances, nous avons constaté qu'il fallait favoriser les montées car elles vont permettre de débloquer des tâches et donc de masquer les attentes dues aux communications pour les translations. Les calculs des interactions proches sont des tâches totalement indépendantes, elles peuvent donc permettre de combler des temps d'attente de fin des tâches de montée, ou de réception de données. Il est important de tout mettre en œuvre pour que les cœurs n'aient pas à attendre les données entre deux calculs. C'est pourquoi, il faut que l'ordonnancement favorise l'exécution des tâches de la montée. Dans StarPU, il est possible d'intégrer ce mécanisme en utilisant les priorités et la bonne politique d'ordonnancement.

3.5.3.1 Priorités

Pour favoriser ou non des tâches avec StarPU, il faut mettre des priorités aux tâches. Celle-ci est définie comme un entier relatif, pour lequel la valeur 0 correspond à la priorité par défaut. Les priorités doivent gérer plusieurs points.

Bien gérer les communications MPI Les priorités doivent permettre de faire les communications MPI le plus tôt possible. Il est alors nécessaire de mettre une forte priorité sur les tâches de partage qui font des copies vers les blocs MPI, et ainsi permettent la résolution des dépendances des *MPI_Send*. Ainsi, nous limiterons la pénalité subie par un nœud ayant moins de calculs à réaliser, et ayant besoin des données de ses voisins afin de réaliser les translations ou les échanges du le premier niveau.

Il n'est pas important de finir la phase de montée le plus rapidement. En effet, nous pourrions penser qu'il serait intéressant, pour un nœud ayant moins de calculs, de commencer la phase de descente plus tôt au lieu d'attendre. Par cette technique, nous décalons simplement l'attente à la fin.

Avoir assez de parallélisme Comme nous soumettons les interpolations et les translations au moyen de *callbacks*, nous n'avons pas toutes les tâches soumises à la suite. Il faut donc que nous fassions attention à avoir assez de tâches disponibles pour fournir du travail à toutes nos ressources. Un autre point essentiel, est que les tâches disponibles doivent être de nature différente afin de convenir à toutes nos ressources de calcul.

Masquer les attentes Le point noir d'un ordonnancement est d'avoir des ressources inactives, en attente de tâches disponibles. L'avantage de la méthode multipôle est que nous avons les interactions proches qui sont totalement indépendantes et qui peuvent donc permettre de combler les attentes. Pour que ces tâches jouent ce rôle, nous devons mettre leur priorité la plus basse possible. Ainsi, elles ne seront ordonnancées qu'en dernier recours.

Le danger de mettre des priorités sur les tâches est que l'ordonnanceur perd sa liberté pour faire les meilleurs choix. Si nous favorisons trop une tâche, les ressources de calcul n'auront que cette tâche à réaliser au début, et cela peut être problématique. Prenons par exemple, un cas de figure où les étapes de montée locale ne sont pas efficaces sur GPU alors que la translation l'est. Si nous favorisons la montée locale par rapport à la translation, nous aurons alors des GPU exécutant les tâches de montée, alors qu'il aurait pu être préférable qu'ils fissent les translations. Nous devrons donc ajuster les priorités en prenant en compte tous ces éléments, et en réalisant des tests sur le comportement des tâches sur les différentes ressources de calcul. Nous détaillerons ce travail dans la section 4.2.

3.5.3.2 Politique d'ordonnancement

StarPU permet de choisir une politique d'exécution au lancement du programme. La politique la plus intéressante et qui prend en charge la gestion des priorités est nommée *dmdas* pour *deque model data aware sorted*. Elle se base sur un modèle de performances en considérant les temps de transfert des données, construit à l'exécution ou fourni, afin de prendre des décisions. Elle place les tâches là où leur temps de terminaison est minimal.

3.6 Optimisation des tâches

Maintenant que nous avons vu comment nous allons intégrer l'ordonnanceur à l'algorithme multipôle, il est temps de nous intéresser à l'implémentation de l'algorithme lui-même et plus particulièrement aux tâches.

3.6.1 Agrégation et désagrégation

Nous traitons l'agrégation et la désagrégation de la même manière, la seule différence étant d'utiliser la transposée du conjugué de la matrice pour la désagrégation.

L'agrégation est un ensemble de produits matrice-vecteur, un pour chaque boîte de notre bloc. Malheureusement, ces produits ne peuvent pas être groupés, nous avons donc autant d'appels à la fonction réalisant le produit que de boîtes dans le bloc. Si les produits sont petits, le surcoût des appels peut ne pas être négligeable.

3.6.1.1 CPU

Pour réaliser ces produits, nous utilisons la fonction de type *gemv* des BLAS. Elle réalise de manière optimisée ces produits. De plus, elle est capable de transposer conjuguer à la volée la matrice, ce qui nous sert pour la désagrégation. Nous avons choisi de faire nos opérations en simple précision, en conformité avec la précision de la méthode multipôle, et comme nous manipulons des complexes, nous utilisons plus particulièrement la fonction *cgemv*.

3.6.1.2 GPU

Pour les GPU, nous utilisons la fonction équivalente de CUBLAS, qui s'appelle *cublasCgemv*. Elle gère également les transpositions-conjugaisons de matrices. Les produits successifs sont exécutés en utilisant différentes *streams*, permettant au GPU d'en exécuter plusieurs en même temps et ainsi de recouvrir les latences mémoire.

3.6.2 Translation

En entrée de la translation, nous avons la matrice des champs des boîtes voisines éloignées du bloc, composée des champs locaux et externes au bloc. Nous avons également la matrice composée de vecteurs représentant les matrices diagonales des translations à appliquer. À cela, nous devons ajouter les informations portant sur les voisins éloignés des boîtes du bloc et leur type, et de quel type sont-ils, afin de savoir quelle matrice de translation devra être appliquée. Ces informations sont stockées par trois listes : la liste de tous les indices des voisins extérieurs à la suite dans l'ordre des boîtes, la liste des types de voisins extérieurs, et la liste des indices marquant le changement de boite.

3.6.2.1 CPU

La translation n'est pas un simple enchaînement d'opérations matricielles, elle nécessite de lire des tableaux pour connaître les indices des voisins éloignés et des matrices diagonales à utiliser. Par conséquent, il nous faut avant chaque produit matrice-vecteur, trouver quels en sont les acteurs, et ensuite réaliser le produit.

Ce produit matrice-vecteur n'est pas un produit classique, car la matrice est diagonale. Il correspond à un produit terme à terme entre la diagonale et le vecteur. Malheureusement, la fonction qui réalise ce type de calcul n'existe pas dans les BLAS. Nous avons alors codé cette opération de la façon la plus simple, voir l'algorithme 1.

foreach *boîte* $b \in$ *bloc* **do**
  foreach *boîte* $v \in$ *Voisin(b)* **do**
    foreach *direction* d **do**
      $N_{b,d} = N_{b,d} + T_{type(b,v),d} * F_{v,d}$
    end
  end
end

Algorithme 1 : Algorithme naïf de la translation d'un bloc

Cet algorithme n'est pas optimisé pour favoriser la localité temporelle de la donnée N. En effet, à chaque voisin différent de b, nous ajoutons la contribution dans N. Afin d'augmenter la localité

temporelle et ainsi profiter davantage des caches, il serait donc intéressant de grouper les écritures sur les mêmes indices de N. Nous avons réalisé ce changement sur l'algorithme 2.

foreach *boîte* $b \in$ *bloc* **do**
  foreach *direction* d **do**
    foreach *boîte* $v \in$ *Voisin(b)* **do**
      $N_{b,d} = N_{b,d} + T_{type(b,v),d} * F_{v,d}$
    end
  end
end

Algorithme 2 : Algorithme de la translation d'un bloc, utilisant le principe de la localité

Le problème avec ce nouvel algorithme est que la double boucle sur les directions ne peut plus être faite en utilisant des instructions vectorielles. Mais, en faisant ce qu'on appelle du tuilage de boucle (*loop tiling*), il est tout à fait possible de cumuler la localité et les instructions vectorielles. Cette technique est montré par l'algorithme 3.

foreach *boîte* $b \in$ *bloc* **do**
  foreach *direction* D *multiple de* $TUILE$ **do**
    foreach *boîte* $v \in$ *Voisin(b)* **do**
      foreach $d \in [1, TUILE[$ **do**
        $N_{b,D*TUILE+d} = N_{b,D*TUILE+d} + T_{type(b,v),D*TUILE+d} * F_{v,D*TUILE+d}$
      end
    end
  end
end

Algorithme 3 : Algorithme de la translation d'un bloc, utilisant le tuilage de boucle

3.6.2.2 GPU

Afin d'utiliser les CUBLAS avec la translation, nous allons devoir la décomposer en un ensemble de fonctions matricielles. Cependant, il ne faut pas que les opérations soient trop petites et que la majorité du temps soit passée à réaliser des appels.

Avec CUBLAS Contrairement aux BLAS sur CPU, dans les CUBLAS, il existe la fonction *cublasCdgmm* qui permet de réaliser un produit avec une matrice diagonale. Nous pouvons alors implémenter la translation comme une série d'appels à cette fonction en utilisant plusieurs *streams* comme nous l'avons vu avec l'agrégation. Le problème est que la fonction qui calcule le produit diagonal n'est pas capable de faire la réduction additive pour sommer les résultats. Afin que la translation soit complète, il faudrait alors utiliser un vecteur intermédiaire qui serait sommé avec le vecteur résultat.

Avec notre implémentation Face aux problèmes posés par la version CUBLAS, nous avons décidé d'écrire le noyau nous-mêmes. La première décision à prendre est de choisir comment seront créés les blocs. Une première approche pourrait être de faire un bloc pour chaque couple de boîte-voisin. Un bloc serait alors en charge de calculer la contribution d'un voisin sur une boîte.

Comme pour la version CPU, en faisant ainsi, nous ne profitons pas du fait que le résultat est la somme de plusieurs contributions, et écrivons en mémoire globale pour chaque couple boîte-voisin. Afin de profiter de cette particularité, nous pouvons alors créer un bloc pour chaque boîte du bloc de données, laissant ainsi chaque bloc sommer les contributions de tous les voisins éloignés de la boîte pour laquelle ils sont en charge. Le code CUDA correspondant à ce découpage est détaillé dans le code 3.1.

```
1 static __global__ void translation_cuda_ker3
```

```
  (cuComplex *multipoleFields, cuComplex *transferMatrix,
   int *farNeighbourIdx, int *farNeighbourList, int *farNeighbourType,
   cuComplex *localFields, int firstBox, int dirNum)
{
  extern __shared__ cuComplex localMF[] ; // 1st shared variable [
     dirNum]
  __shared__ int *localFNIdx ; // 2nd shared variable [2]
  __shared__ int *localFNL ; // 3rd shared variable [fNNum]
  __shared__ int *localFNT ;  // 4th shared variable [fNNum]
  int fNNum ;

  int box = blockIdx.x+firstBox ;

  // Copy of the second shared variable
  ...

  // Copy of the third shared variable
  ...

  // Copy of the fourth shared variable
  ...

  __syncthreads() ;

  int dir = blockIdx.y*blockDim.x+threadIdx.x ;
  if (dir <  dirNum)
  {
    int ldir = threadIdx.x ;
    localMF[ldir] = make_cuFloatComplex(0, 0) ;

    for (int fNIdx = 0 ; fNIdx < fNNum ; fNIdx++)
    {
      int type = localFNT[fNIdx] ;
      int fN = localFNL[fNIdx] ;
      localMF[ldir] =
        cuCaddf(localMF[ldir],
                cuCmulf(transferMatrix[type*dirNum+dir], // TODO
                   localTM ?
                        localFields[fN*dirNum+dir])) ;
    }
    multipoleFields[box*dirNum+dir] =
      cuCaddf(multipoleFields[box*dirNum+dir], localMF[ldir]) ;
  }
}
```

Code 3.1 – Calcul principal de la translation en CUDA

Nous devons copier en mémoire partagée le champ de sortie, qui sera utilisé plusieurs fois par un thread. Nous ne copions pas en mémoire partagée le champ d'entrée bien qu'utilisé plusieurs fois, car sa taille est trop importante.

Nous copions également les listes afin de réduire le nombre de lectures en mémoire globale en profitant des accès coalescents. Ainsi, nous avons le nombre de lectures en mémoire globale qui vaut le nombre de voisins divisé par le nombre de threads. Puis nous faisons un nombre de lectures en mémoire partagée qui est égal au nombre de threads, au lieu d'avoir nombre de threads lectures en mémoire globale, moins rapide d'accès. Pour ajouter et multiplier des complexes stockés en simple précision, nous devons utiliser respectivement *cuCaddf* et *cuCmulf*.

Concernant le nombre de threads par bloc, il est conseillé de prendre 64 dans la documentation

de CUDA, mais cela pose un problème quand le nombre de directions est très faible. À ce moment-là, s'il est inférieur à 32, nous pouvons utiliser des blocs de 32 threads. Cela peut aussi être avantageux quand le modulo du nombre de directions par le nombre de threads est élevé.

3.6.3 Décalages

Les décalages sont des opérations de multiplication du champ d'entrée par une matrice diagonale. Pour le décalage supérieur, le champ de sortie va être la somme de plusieurs produits, et pour le décalage inférieur, c'est le champ d'entrée qui sera utilisé plusieurs fois pour donner les champs de sortie.

Cette opération est similaire à la translation entre deux boîtes, en remplaçant la relation de voisin éloigné par la relation de parenté.

3.6.3.1 CPU

L'algorithme implémenté est le même que pour la translation : afin d'optimiser l'utilisation des caches, nous mettons en œuvre du tuilage de boucle, voir l'algorithme 3.

3.6.3.2 GPU

Comme avec le CPU, nous avons le même algorithme que pour la translation avec CUDA. Un bloc se charge de calculer, soit le champ du parent issu de tous les enfants, pour le décalage supérieur, soit le champ des enfants issu de celui du parent, pour le décalage inférieur.

3.6.4 Interpolation et anterpolation

Les anterpolations sont équivalentes aux interpolations en inversant les deux parties de l'algorithme. Par conséquent nous n'allons étudier que l'interpolation.

Afin de réaliser l'interpolation, nous utilisons l'algorithme d'Alpert vu à la section 1.2.3.4, en appliquant un algorithme de compression. Nous avons alors quatre étapes : la transformée de Fourier (point 1 de la section 1.2.3.4), la multiplication compressée, la recopie en ajoutant ou supprimant une partie (suivant si transformée de Fourier normale ou inverse), et la transformée inverse.

3.6.4.1 Étude

Nous allons examiner l'algorithme de multiplication ainsi que la recopie.

Multiplication compressée L'algorithme compressé nous a été fourni par Francis Collino. La somme des produits par la matrice $\mathbb{M}$ est remplacée par l'algorithme 4.

Pour implémenter cet algorithme de manière efficace, nous devons grouper les calculs de type identique en séparant les boucles. Nous avons agrandi le tableau $tempV$ pour qu'il serve à tous les modes et toutes les boîtes. De la même manière, nous avons transformé le scalaire v en un tableau à trois dimensions pour intégrer les boîtes, les modes et les θ de sortie. Nous avons intégré la multiplication par $Poids\theta$ au polynôme P. Nous obtenons alors l'algorithme 5.

Cet algorithme peut se réécrire comme des opérations matricielles. Nous avons alors les trois calculs suivants pour chaque boîte b :

$$\begin{cases} diag(tempV(b)) = diag(In(b)) * diag(P) \\ V(b) = tempV(b) * M \\ diag(Out(b)) = diag(V(b)) * diag(P') \end{cases}$$

où :

- $diag(A)$ est une matrice diagonale composée des éléments de A ;
- $tempV$ est une matrice de taille $nbBoite \times nbMode \times nbIn\theta$;
- In est une matrice de taille $nbBoite \times nbMode \times nbIn\theta$;
- P est une matrice de taille $nbMode \times nbIn\theta$;
- V est une matrice de taille $nbBoite \times nbMode \times nbOut\theta$;
- M est une matrice de taille $nbIn\theta \times nbOut\theta$;

foreach *boîte* $b \in bloc$ **do**
  Transposer($In(b)$)
  foreach *mode* m **do**
    foreach *theta d'entrée* $in\theta$ **do**
      $tempV(in\theta) \leftarrow In(b, m, in\Theta) * Poids\theta(in\theta) * P(m, in\theta)$
    end
    foreach *theta de sortie* $out\theta$ **do**
      $V \leftarrow 0$
      foreach *theta d'entrée* $in\theta$ **do**
        $V \leftarrow V + M(out\theta, in\theta) * tempV(in\theta)$
      end
      $Out(b, m, out\theta) \leftarrow v * P'(m, out\theta)$
    end
  end
  Transposer($Out(b)$)
end

Algorithme 4 : Algorithme du produit par M compressé

foreach *boîte* $b \in bloc$ **do**
  Transposer($In(b)$)
end
foreach *boîte* $b \in bloc$ **do**
  foreach *mode* m **do**
    foreach *theta d'entrée* $in\theta$ **do**
      $tempV(b, m, in\theta) \leftarrow In(b, m, in\Theta) * P(m, in\theta)$
    end
  end
end
foreach *boîte* $b \in bloc$ **do**
  foreach *mode* m **do**
    foreach *theta de sortie* $out\theta$ **do**
      $V(b, m, out\theta) \leftarrow 0$
      foreach *theta d'entrée* $in\theta$ **do**
        $V(b, m, out\theta) \leftarrow V(b, m, out\theta) + M(out\theta, in\theta) * tempV(b, m, min\theta)$
      end
    end
  end
end
foreach *boîte* $b \in bloc$ **do**
  foreach *mode* m **do**
    foreach *theta de sortie* $out\theta$ **do**
      $Out(b, m, out\theta) \leftarrow V(b, m, out\theta) * P'(m, out\theta)$
    end
  end
end
foreach *boîte* $b \in bloc$ **do**
  Transposer($Out(b)$)
end

Algorithme 5 : Algorithme du produit par M compressé après la séparation des calculs

— Out est une matrice de taille $nbBoite \times nbMode \times nbOut\theta$;
— P' est une matrice de taille $nbMode \times nbOut\theta$.

Nous constatons qu'il est possible d'intégrer les calculs sur les boîtes b dans les opérations matricielles. Pour cela, comme nous avons des matrices qui ont trois dimensions, nous devons grouper deux dimensions qui se suivent, afin de n'en faire qu'une. Pour la première équation, nous devons grouper la dimension sur les modes et celle sur les $in\theta$. Pour le deuxième calcul, nous groupons les dimensions sur les boîtes et sur les modes. Le dernier calcul nécessite de fusionner les dimensions sur les modes et sur les $out\theta$. Nous obtenons alors le système :

$$\begin{cases} tempV = In * diag(P) \\ V = tempV * M \\ Out = V * diag(P') \end{cases} \tag{3.1}$$

où :
— $tempV$ est une matrice de taille :
  — $nbBoite \times (nbMode * nbIn\theta)$ pour la première équation,
  — $(nbBoite * nbMode) \times nbIn\theta$ pour la deuxième équation ;
— In est une matrice de taille $nbBoite \times (nbMode * nbIn\theta)$;
— V est une matrice de taille :
  — $(nbBoite * nbMode) \times nbOut\theta$ pour la deuxième équation,
  — $nbBoite \times (nbMode * nbOut\theta)$ pour la troisième équation ;
— Out est une matrice de taille $nbBoite \times (nbMode * nbOut\theta)$;

Dans l'algorithme, nous avons une transposition au début, ainsi qu'à la fin. En les supprimant, nous obtenons le système suivant :

$$\begin{cases} tempV = In * diag(P) \\ V(b) = M * tempV(b) \\ Out = V * diag(P') \end{cases} \tag{3.2}$$

où :
— $tempV$ est une matrice de taille :
  — $nbBoite \times (nbIn\theta * nbMode)$ pour la première équation,
  — $nbBoite \times nbIn\theta \times nbMode$ pour la deuxième équation ;
— In est une matrice de taille $nbBoite \times (nbIn\theta * nbMode)$;
— P est une matrice de taille $nbIn\theta \times nbMode$;
— V est une matrice de taille :
  — $nbBoite \times nbOut\theta \times nbMode$ pour la deuxième équation,
  — $nbBoite \times (nbOut\theta * nbMode)$ pour la troisième équation ;
— M est une matrice de taille $nbOut\theta \times nbIn\theta$;
— P' est une matrice de taille $nbOut\theta \times nbMode$.

L'avantage de ce système est qu'il est obtenu sans avoir à faire des transpositions de In et de Out. L'inconvénient est que la multiplication par M ne peut plus se faire en un seul produit. Nous comparerons à la section 4.2.4.1 les performances des calculs sans transposition, avec ceux impliquant des transpositions.

Recopie partielle de colonnes Après la transformée de Fourier, il est nécessaire de recopier seulement certains blocs colonnes de la matrice : les $l + 1$ premières et les l dernières. Nous avons schématisé cette opération à la figure 3.29.

Pour copier ces blocs, nous pouvons utiliser deux copies de matrices, en utilisant du bourrage (*padding*). Pour le premier bloc, le bourrage est de $k + l$ en entrée et l en sortie. Pour le deuxième bloc, nous avons un bourrage de $l + 1 + k$ en entrée et $l + 1$ en sortie. Ces deux copies de matrices peuvent être réunies en une seule, si nous nous affranchissons de problèmes causés par les première et dernière lignes. Comme nous le constatons sur la figure 3.30, nous obtenons alors un bourrage de k en entrée.

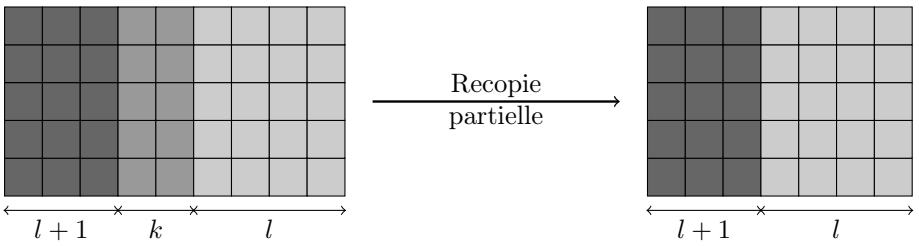


FIGURE 3.29 – Recopie d'une partie des colonnes de la matrice

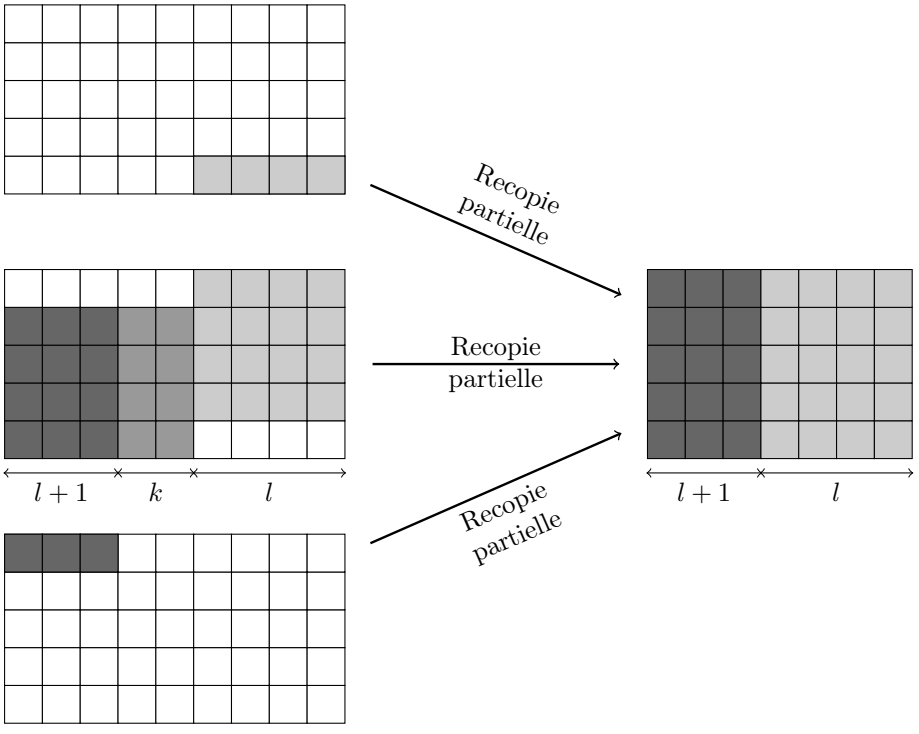


FIGURE 3.30 – Recopie améliorée d'une partie des colonnes de la matrice

3.6.4.2 CPU

Les transformées de Fourier sont réalisées en utilisant la bibliothèque FFTW, grâce à l'appel *fftwf_ execute_ dft*. Les opérations sur les matrices avec les BLAS. Malheureusement, le produit diagonal n'est pas présent dans les BLAS. Il est donc nécessaire de l'implémenter, en utilisant comme nous l'avons déjà vu le tuilage de boucle.

3.6.4.3 GPU

Pour les transformées de Fourier, nous utilisons la fonction *cufftExecC2C* des CUBLAS. Pour les opérations sur les matrices, nous utilisons les équivalents des BLAS avec quelques petites différences. La première est qu'il existe une fonction permettant de faire le produit diagonal ! Cette fonction se nomme *cublasSdgmm*. La deuxième différence implique le produit principal avec M, qui est une matrice réelle, alors que l'autre terme est complexe. Or, sur CPU nous avions la fonction *scgemm* qui est malheureusement absente sur GPU. Afin de contourner ce problème, nous décomposons alors notre matrice complexe en deux matrices réelles, puis nous la rassemblons, ce qui implique un surcoût non négligeable.

3.6.5 Interactions proches

Le calcul des interactions proches est fait directement contrairement aux interactions lointaines. Pour le bloc de boîtes, nous calculons toutes les interactions de tous les points voisins entre eux. Un calcul d'interaction est la multiplication entre l'exponentielle d'une valeur proportionnelle à la norme du vecteur séparant les deux points, et la valeur d'entrée au point qui interagit avec l'autre.

3.6.5.1 CPU

Nous n'utilisons pas d'algorithme particulier, nous avons implémenté le plus simplement l'algorithme, comme présenté à l'algorithme 6.

foreach *boîte* $b \in bloc$ **do**
  foreach *voisin* $v \in Voisin(b)$ **do**
    foreach *point* $pi \in b$ **do**
      foreach *point* $qi \in b$ **do**
        if $p \neq q$ **then**
          $Out[p] \leftarrow Out(p) + exp(i * k * norm(\vec{pq}) * In(q)/(4 * \pi * norm(\vec{pq})$
        end
      end
    end
  end
end

Algorithme 6 : Algorithme du produit par M compressé

Pour calculer l'exponentielle, nous utilisons la fonction *cexpf* de la bibliothèque mathématique.

3.6.5.2 GPU

Nous avons décidé de lancer un noyau CUDA par couple boîte-voisin. Dans notre noyau, chaque thread d'un bloc s'occupera d'un point destination, en calculant toutes les contributions des points de la boîte voisine, appelés points source.

Les threads ont trois coordonnées à lire pour chaque point, ce qui correspond à 12 octets. Le problème est qu'il est impossible de lire des mots de 12 octets ou même que chaque thread lise une coordonnée de son point, le pas (*stride*) étant trop grand. Par conséquent, nous copions les coordonnées des points à la suite en mémoire partagée, et ensuite les lectures pourront se faire sans problème. Le code 3.2 nous montre l'implémentation en CUDA.

```
static __global__ void near_cuda_ker(...)
{
  extern __shared__ float sharedInCoords[] ;
  __shared__ float *sharedOutCoords ;
  __shared__ cuComplex *sharedIn ;
  ...
  // Lecture des coordonnees des points destination pour copie en
  // memoire partagee : sharedOutCoords
  ...

  globalOutPoint = blockIdx.x*blockDim.x+threadIdx.x+firstOutPoint ;
  if (globalOutPoint < outPointMax)
  {
    outCoord[0] = sharedOutCoords[threadIdx.x*3+0] ;
    outCoord[1] = sharedOutCoords[threadIdx.x*3+1] ;
    outCoord[2] = sharedOutCoords[threadIdx.x*3+2] ;
  }

  // Iteration permettant de traiter tous les points source
  for (int i = 0 ; i < (inPointNum+blockDim.x-1)/blockDim.x ; i++)
  {
    int inPoint = i*blockDim.x+threadIdx.x ;

    // Copie des coordonnees des points source vers sharedInCoords
    ...

    if (inPoint < inPointNum)
      sharedIn[threadIdx.x] = in[inPoint+firstInPoint] ;
    __syncthreads() ;

    if (globalOutPoint < outPointMax)
    {
      // Parcours des points source
      for (int j = 0 ; j < blockDim.x ; j++)
      {
        // Verification que le thread doit travailler
        if (i*blockDim.x+j < inPointNum)
        {
          float x, y, z, div, norm ;
          cuComplex temp ;

          x = outCoord[0]-sharedInCoords[j*3+0] ;
          y = outCoord[1]-sharedInCoords[j*3+1] ;
          z = outCoord[2]-sharedInCoords[j*3+2] ;
          norm = sqrtf(x*x+y*y+z*z) ;
          if (norm > norm_crit)
          {
            temp = cuCmulf(cuCexpif(k*norm), sharedIn[j]) ;
            div = 4.*M_PI*norm ;
            temp = make_cuFloatComplex(cuCrealf(temp)/div, cuCimagf(
                temp)/div) ;
            localOut = cuCaddf(localOut, temp) ;
          }
        }
        else
          break ;
```

```
56          }
57        }
58      }
59      if (globalOutPoint < outPointMax)
60        out[globalOutPoint] = cuCaddf(out[globalOutPoint], localOut) ;
61  }
```

Code 3.2 – Calcul des interactions proches

Pour régler le problème des points, il est possible de modifier le stockage des points dans tout notre algorithme, en plaçant toutes les coordonnées d'un même axe à la suite, c'est-à-dire tous les x, puis tous les y, et tous les z. Ainsi, nous n'avons plus besoin d'utiliser des copies en mémoire partagée, qui n'étaient lues qu'une seule fois. Le calcul de l'exponentielle est réalisé par la fonction cuCexpif.

3.7 Conclusion

Nous proposons un algorithme qui utilise à la fois les CPU et les GPU d'un nœud, grâce à un ordonnanceur dynamique, et qui s'exécute également sur plusieurs nœuds grâce à MPI. C'est pourquoi, nous avons exprimé l'algorithme multipôle en une série de tâches agissant sur des blocs de boîtes et de directions. Celles-ci sont l'agrégation, la translation, les décalages supérieur et inférieur, la transformée de Fourier, la multiplication d'Alpert, la désagrégation, et le calcul des interactions proches. À cela, nous avons ajouté des tâches permettant de réaliser les communications entre les blocs. La communications entre les nœuds se fait grâce à des blocs fictifs dédiés. Enfin, pour recouvrir les communications et les éventuels déséquilibres de la distribution statique des boîtes, nous distribuons dynamiquement les interactions proches à calculer.

Sommaire

Chapitre

4

Résultats

The best way to predict the future is to implement it.
David Heinemeier Hansson.

Nous venons de présenter notre adaptation de la méthode multipôle. Dans ce chapitre, nous allons nous attacher à justifier expérimentalement nos choix. Dans un premier temps, nous allons expliciter les conditions dans lesquelles les tests sont effectués. Nous verrons les grandeurs caractéristiques de la méthode multipôle qui nous permettront de réaliser des tests adaptés. Une fois ce travail terminé, nous évaluerons les différents noyaux pour toutes nos tâches, à la fois pour le CPU et le GPU. Ensuite, d'après les résultats obtenus individuellement sur les noyaux, nous pourrons tester l'exécution de notre implémentation en séquentiel puis en parallèle, sur un nœud hétérogène comportant des GPU. La dernière étape est l'utilisation de MPI, afin de réaliser des tests sur des machines à mémoire distribuée.

4.1 Présentation du contexte

Nous allons d'abord détailler les exemples qui seront utilisés. Puis, nous montrerons les grandeurs caractéristiques de la méthode multipôle concernant les exemples choisis. Enfin, nous présenterons les machines nous ayant permis d'effectuer nos tests.

4.1.1 Cas tests

Nos tests préliminaires vont permettre de valider (ou d'invalider) nos choix. Les cas tests utilisés doivent donc être « génériques », dans le sens où les résultats ne doivent pas être le reflet simplement du cas test choisi. Pour cela, ils doivent avoir un maillage régulier et ne pas privilégier une direction de l'espace.

C'est pourquoi, nous avons choisi des sphères, dont les points sont répartis uniformément sur la surface, en trois déclinaisons : 1 million de points, 5 millions de points, et 8 millions de points. Pour chacune, la taille est adaptée au nombre de points et à la fréquence, de telle sorte qu'il y ait 10 points par longueur d'onde. Nous avons fait ce choix car, dans notre cadre d'étude, nous avons couramment un tel nombre de points.

Nous tenons à préciser que nous réalisons ces tests dans un but de justification de nos choix, nous ne cherchons pas à établir des records sur des tailles de problèmes. De plus, la préparation des cas tests est longue, à cause de tous les paramètres possibles à régler, et il a fallu exécuter les tests de nombreuses fois pour confirmer le résultat et pour réaliser toutes les mesures nécessaires.

Cas test	Nombre de points	Taille (m)	Fréquence (Mhz)
Sphère 1M	1 000 000	52.7	716
Sphère 5M	5 000 000	52.8	716
Sphère 8M	8 000 000	52.8	907

TABLE 4.1 – Caractéristiques des cas tests

C'est pourquoi nos différents cas tests ont des tailles assez faibles, mais représentatives de ce que nous cherchons à vérifier.

Les caractéristiques de chacune de nos sphères sont synthétisées dans le tableau 4.1.

4.1.2 Caractéristiques de l'octree

Afin de comprendre ce qui se cache derrière le mot *octree*, nous allons étudier ses caractéristiques pour nos trois sphères. Le tableau 4.2 permet d'avoir un ordre d'idée sur les nombres de directions et de boîtes, qui le composent. Nous avons également ajouté les nombres moyens par boîte de voisins éloignés, de voisins et de points.

Ce qui est important de retenir de ces valeurs, c'est surtout le nombre de voisins éloignés qui vaut en moyenne 45, et le nombre de voisins au dernier niveau qui est 10. Ces valeurs nous serviront de références dans nos futurs tests.

4.1.3 Configuration des machines de tests

Nous avons utilisé deux plateformes de tests afin de réaliser nos différentes exécutions, suivant nos besoins.

4.1.3.1 Averell

Cette machine possède 2 processeurs, 2 GPU et 64 Go de mémoire. Les processeurs sont des Intel Xeon E5-2650 de 8 cœurs chacun. Ce processeur est cadencé à 2 Ghz et possède un cache L1 de 32 ko par cœur, un cache L2 de 256 ko par cœur, et un cache L3 partagé de 20 Mo. La puissance d'un cœur est de 16 Gflops.

En plus des processeurs, nous trouvons 2 GPU, des Tesla M2070. Ils ont une CUDA capability valant 2.0 et comportent 6 Go de mémoire. Leur puissance est de 1030 Gflops en simple précision et 515 Gflops en double précision.

Cette machine servira pour tous les tests préliminaires mais également pour les tests concernant l'exécution sur un seul nœud.

4.1.3.2 Mirage

C'est une machine composée de 8 nœuds de 2 Hexa-cœurs, 3 GPU et 36 Go de mémoire. Les nœuds sont reliés entre eux par un réseau Infiniband. Les hexa-cœurs sont des Westmere Intel® Xeon® X5650, cadencés à 2.67 Ghz, ont 6 caches de données L1 de 32 ko, 6 caches L2 de 256 ko et un cache L3 de 12 Mo. La puissance d'un cœur est de 10.664 Gflops. Les GPU sont des Tesla M2070 comme sur Averell.

Cette machine va être utilisée pour les tests en mémoire distribuée.

4.2 Évaluation des noyaux

Dans le chapitre précédent, nous avons étudié chaque tâche et évoqué pour celles-ci des optimisations éventuelles. Ici, nous allons confronter les différentes implémentations entre elles pour chaque noyau, et ce pour les CPU et les GPU. Nous étudierons seulement les tâches qui ont un comportement bien distinct. De ce fait, la désagrégation très semblable à l'agrégation, et le décalage inférieur semblable au décalage supérieur, ne seront pas étudiés. Pour les autres tâches, une fois

Niveau	Boîtes	θ	Directions	Voisins éloignés	Voisins
3	56	116	27840	44	
4	272	63	8064	44	
5	1160	36	2880	46	
6	4720	22	1056	47	
7	18270	13	416	45	
8	69742	10	240	43	
9	241658	7	112	37	12

(a) Tailles de l'octree pour une sphère à 1 million de points

Niveau	Boîtes	θ	Directions	Voisins éloignés	Voisins
3	56	241	120500	44	
4	272	128	36864	44	
5	1160	69	9936	46	
6	4760	40	3840	47	
7	18948	23	1104	45	
8	72832	14	448	45	
9	281456	10	240	43	
10	1001940	10	240	43	
11	2631670	6	96	26	8

(b) Tailles de l'octree pour une sphère à 5 millions de points

Niveau	Boîtes	θ	Directions	Voisins éloignés	Voisins
3	56	301	192640	44	
4	272	158	50560	44	
5	1160	85	15300	46	
6	4760	48	4800	47	
7	19040	27	1620	47	
8	74652	18	720	46	
9	287384	11	264	44	
10	1057524	8	160	40	
11	3213748	6	96	31	10

(c) Tailles de l'octree pour une sphère à 8 millions

TABLE 4.2 – Grandeurs de l'octree

le comportement sur CPU et GPU identifié, nous établirons une comparaison entre les meilleurs implémentations.

Pour chacun de ces tests, nous avons accordé une grande importance à la précision, en réitérant de nombreuses fois les exécutions. Sur GPU, nous avons obtenu des mesures très précises, moins de 0.5%. Cela n'a pas été le cas sur CPU, où parfois nous obtenions des valeurs de temps aberrantes valant 10 fois ce que nous avions déjà trouvé, même pour des gros calculs. Nous avons alors éliminé de nos résultats les mesures trop extrêmes. Ainsi, même si notre précision reste faible, elle reste représentative de réelles exécutions.

Les tests consistent principalement en une variation des paramètres sur lesquels nous avons la capacité d'influer, notamment le nombre de directions et le nombre de boîtes dans le bloc. Nous rappelons que le découpage selon les boîtes peut s'effectuer avec toutes les valeurs possibles, que le nombre de directions est égal à $\theta\phi$, et que les blocs de directions ne peuvent être construits qu'en séparant les θ. Nous pouvons alors avoir au maximum θ blocs de directions. La valeur de θ est égale à L, le nombre de multipôles.

4.2.1 Agrégation

Les tailles des matrices impliquées dans l'agrégation d'un bloc dépendent de trois paramètres : le nombre de points pour toutes les boîtes du bloc, le nombre de directions et le nombres de boîtes. Le nombre de points ne dépend que de la géométrie de l'objet étudié, et de la profondeur de l'arbre. Nous n'avons donc qu'un contrôle très limité sur cette valeur. Il n'en est pas de même pour le nombre de directions ni pour le nombre de boîtes, pour lesquels nous pouvons adapter leur valeur suivant la manière dont nous construisons nos blocs.

L'agrégation d'un bloc correspond aux multiplications à gauche des vecteurs d'entrée par la matrice d'agrégation. Le nombre de multiplications correspond au nombre de boîtes dans le bloc, chaque multiplication étant différente. Pour une boîte donnée, la matrice d'agrégation associée possède autant de lignes qu'il y a de directions et autant de colonnes qu'il y a de points contenus dans la boîte.

4.2.1.1 CPU

Nous avons étudié l'influence du nombre de points par boîte, qui dépend de la profondeur maximale de l'arbre. Ensuite, nous avons étudié l'influence du choix de la taille des blocs.

Influence du nombre de points Nous avons mesuré l'influence du nombre de points en faisant varier également le nombre de boîtes. Celui-ci détermine le nombre de produits matrice-vecteur qui sont calculés.

Nous avons décidé de faire varier le nombre de boîtes plutôt que le nombre de directions car l'agrégation se produit au dernier niveau, pour lequel nous avons un large choix de tailles pour les blocs de boîtes. Nous avons donc gardé un nombre de directions fixe, égal à 96.

Les temps sont présentés par la figure 4.1. Dans les BLAS que nous utilisons, la MKL de Intel, nous avons plusieurs stratégies suivant les tailles de la matrice et des vecteurs. Ceci peut expliquer les sauts. Nous n'avons pas de justification plus complète concernant l'influence, bien que légère, du nombre de boîtes. Concernant le nombre de points, nous voyons logiquement une augmentation de la performance quand celui-ci augmente. Le palier est atteint à partir de 512 points.

Influence de la taille des blocs Les blocs sont définis par deux tailles : le nombre de boîtes et le nombre de directions. Ce dernier représente le nombre de lignes de la matrice et donc la dimension du vecteur de sortie. Afin de mesurer l'impact du choix des tailles de blocs, nous avons fait varier le nombre de directions entre 8 et 2048, et le nombre de boîtes entre 8 et 2048. Nous avons pris un nombre de points par boîte égal à 30. Les résultats sont présentés sur la figure 4.2.

La courbe a la même allure que pour la variation du nombre de points, avec un palier atteint pour 128 directions.

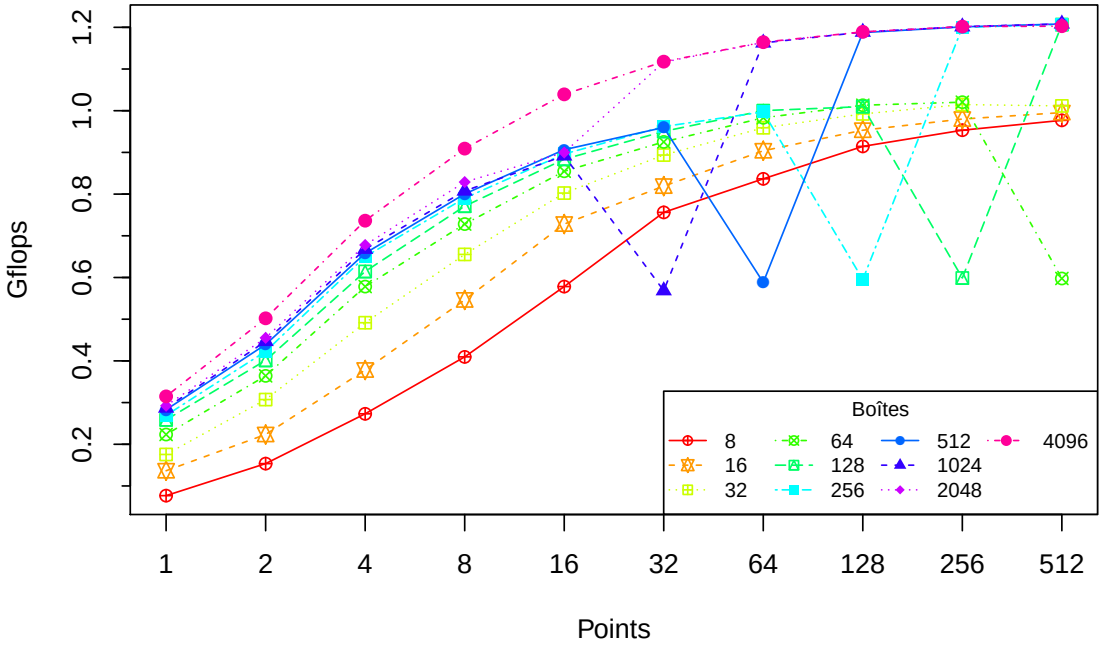


FIGURE 4.1 – Flops de l'agrégation en fonction du nombre de points sur CPU

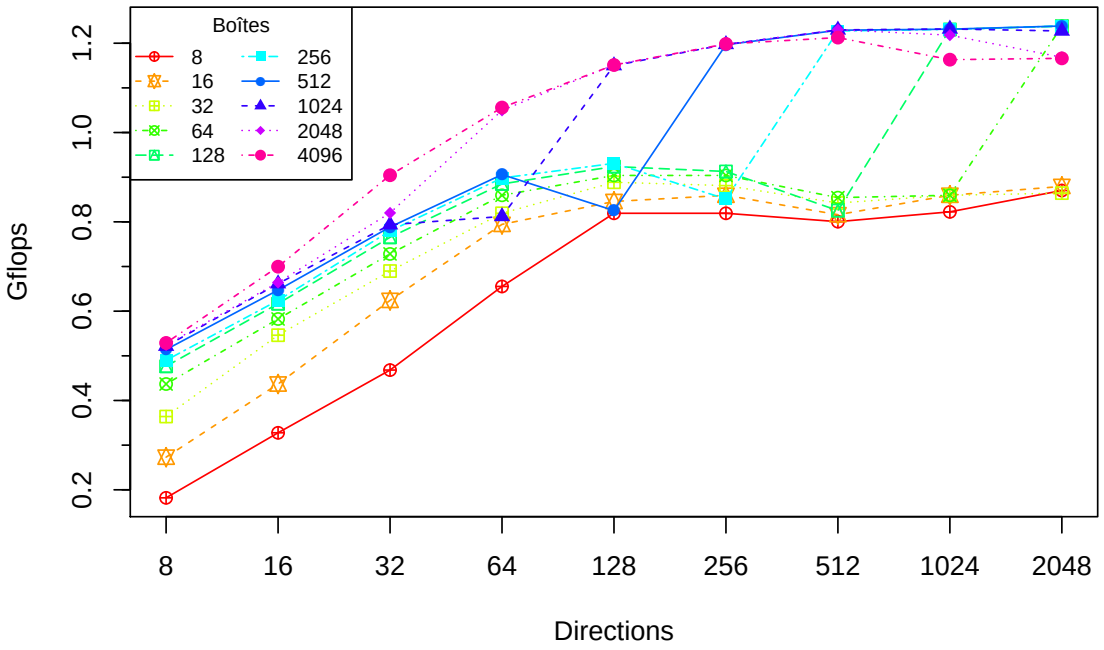


FIGURE 4.2 – Agrégation : flops en fonction de la taille du bloc (pour 30 points) sur CPU

4.2.1.2 GPU

Nous avons procédé aux mêmes tests que pour le CPU, à savoir l'impact du nombre de points et de la taille des blocs.

Influence du nombre de points L'impact du nombre de points est mis en avant par la figure 4.3. Nous voyons un comportement totalement différent de celui sur CPU. Ici, l'augmentation se fait linéairement en deux parties. En effet, pour 256 points, la pente diminue. Ce ralentissement apparait car nous atteignons la puissance maximale pour ce type de calcul.

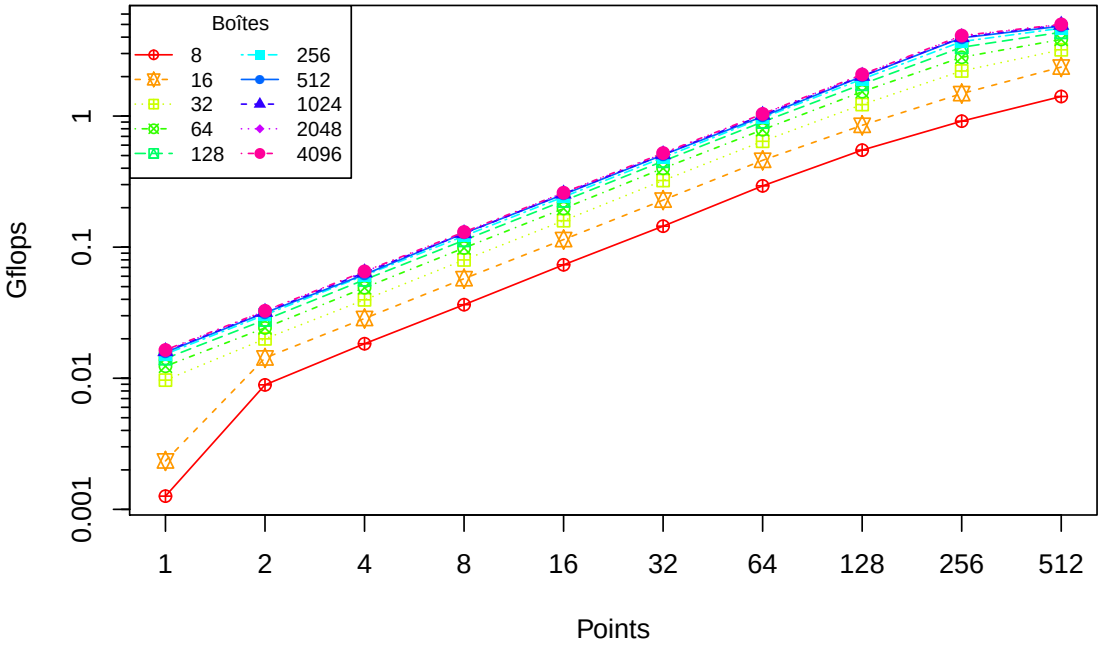


FIGURE 4.3 – Agrégation : flops en fonction du nombre de points sur GPU

Influence de la taille des blocs Notre noyau réalise autant de produits matrice-vecteur qu'il y a de boîtes dans le bloc. Afin d'occuper le GPU pleinement, nous avons assigné les différents produits à différents *streams*, pour les effectuer en parallèle sur le GPU. Ainsi, il est possible de masquer les latences des accès mémoire d'un calcul en basculant sur un autre. La matrice comporte autant de lignes qu'il y a de directions,

La figure 4.4 expose l'évolution du temps face à la modification des tailles des blocs. Nous y voyons clairement que le nombre de directions influence grandement la performance, ainsi que le nombre de boîtes.

Nous constatons que l'augmentation du nombre de boîtes permet de masquer les latences. À partir de 1024 boîtes il n'y a plus de gain supplémentaire, et l'augmentation est même négligeable dès 256 boîtes.

La performance augmente linéairement avec le nombre de directions sans ralentir. Cependant, comme l'agrégation ne se produit qu'au dernier niveau, il est difficile d'atteindre un grand nombre de directions comme le tableau 4.2 le démontre.

4.2.1.3 Comparaison CPU-GPU

Maintenant que nous avons vu comment nous pouvons influer sur les performances de nos noyaux sur CPU et sur GPU, il nous reste à les comparer. Les figures 4.5a et 4.5b synthétisent les résultats sur CPU et GPU en faisant un comparatif grâce au ratio du temps sur CPU par celui sur GPU. La première figure fait la comparaison en fonction de la taille des blocs et la deuxième en fonction du nombre de points.

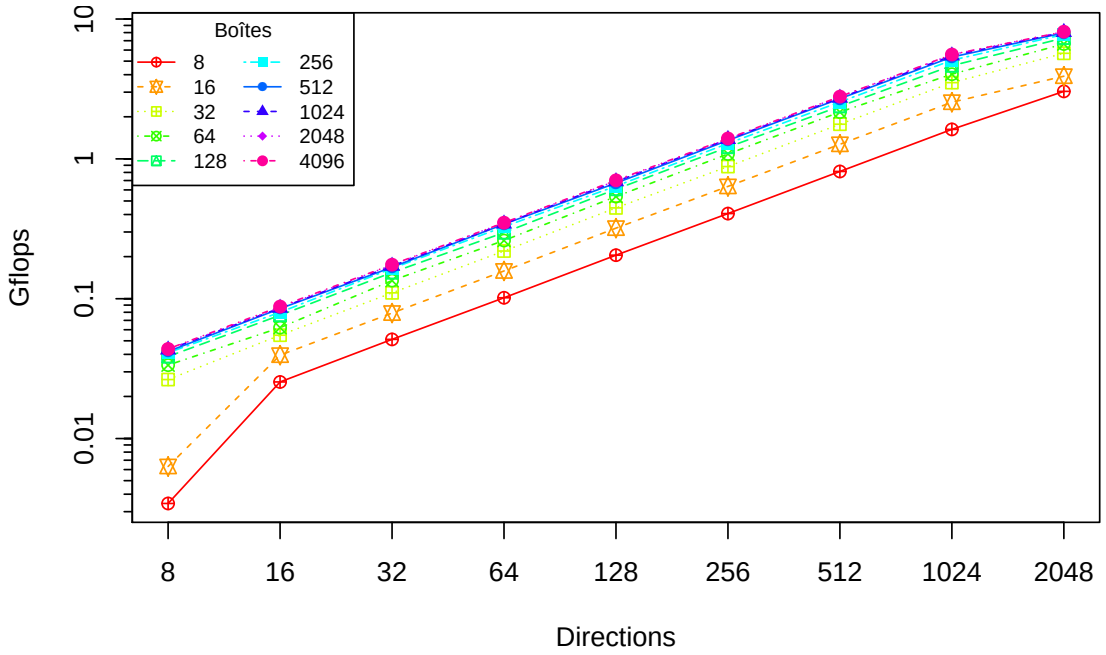


FIGURE 4.4 – Agrégation : flops en fonction de la taille du bloc sur GPU

Pour que le GPU exécute le calcul plus rapidement que le CPU, il faut qu'il y ait au moins 64 points pour 256 boîtes avec 96 directions, ou au moins 256 directions pour 32 avec 30 points.

Pour l'agrégation, dans des conditions réelles (avec des tailles de blocs réalistes), le CPU sera plus avantageux, car le nombre de points sera inférieur à ce que nous imposons.

4.2.2 Translation

La translation consiste à parcourir toutes les boîtes d'un bloc et, pour chaque boîte, à multiplier terme à terme son vecteur par celui de chacun de ses voisins. C'est une opération qui sera très limitée par la vitesse de la mémoire. Pour ces tests, nous avons pris un nombre de voisins extérieurs par boîte de 45, valeur moyenne de nos cas tests. Pour les CPU et les GPU, nous recherchons l'évolution des performances suivant la taille des blocs.

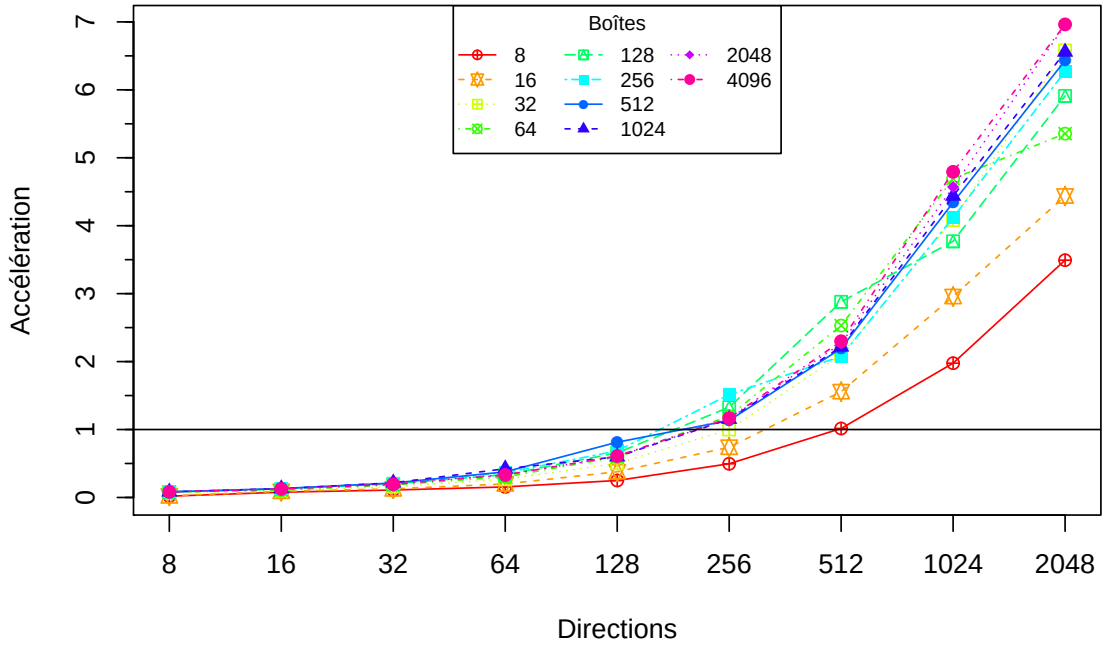
4.2.2.1 CPU

Nous avons présenté plusieurs algorithmes concernant l'implémentation de la translation sur CPU dans la section 3.6.2.1. Nous ne ferons pas ici la comparaison des implémentations mais avons choisi la meilleure. Cependant, ce comparatif est réalisé dans la section 4.2.3.1 avec le décalage qui a les mêmes variations d'implémentations. Les résultats montrant l'impact de la variation de la taille du bloc sont présentés par la figure 4.6.

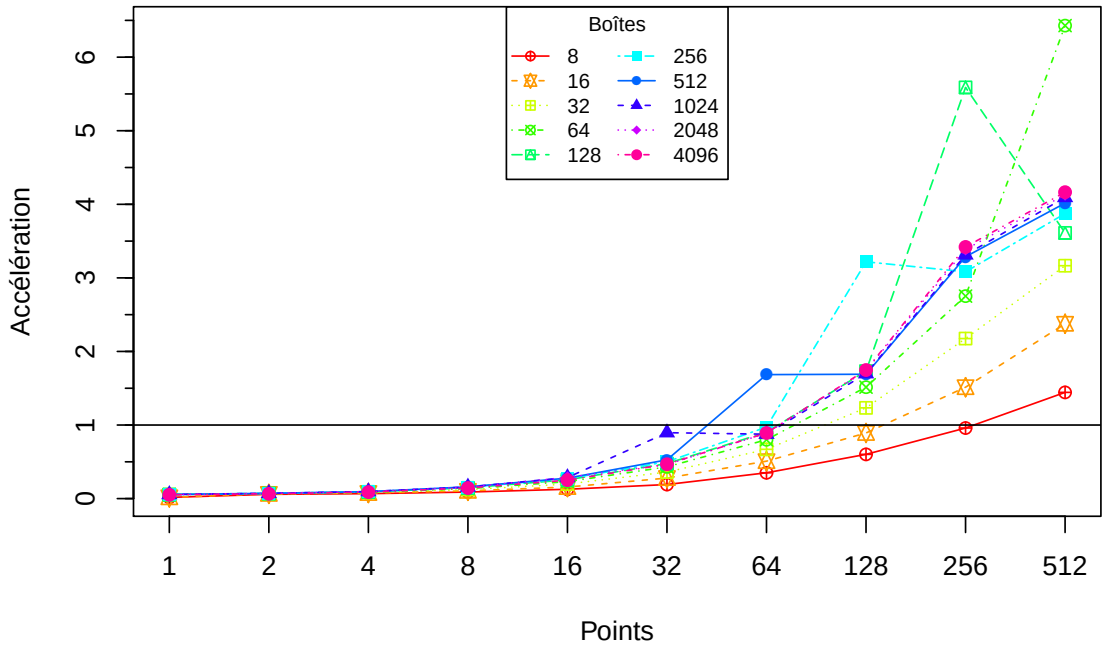
Nous voyons que le changement de la taille des blocs n'a aucune incidence sur les performances constatées. Par ailleurs, les performances sont très éloignées de la puissance crête d'un cœur, car ce type de calcul dépend de la vitesse de la mémoire (*memory bound*).

4.2.2.2 GPU

La figure 4.7 montre les résultats, pour les mêmes tests, sur GPU. Ici, le comportement est totalement différent de ce que nous avions obtenu sur CPU. Le nombre de flops augmente avec l'augmentation de la taille du bloc, jusqu'à ce que le bloc fasse 256 ko. Au-delà, la performance diminue légèrement et reste stable. Ce palier correspond au débordement du cache qui ne peut plus stocker tout le bloc, dont les données sont parcourues plusieurs fois. Notons que, comme avec le CPU, notre maximum de 6 Gflops, est bien peu comparé, au téraflops théorique de notre GPU.



(a) En fonction de la taille des blocs (30 points)



(b) En fonction du nombre de points et de boîtes (pour 96 directions)

FIGURE 4.5 – Agrégation : rapport du temps sur CPU par rapport au temps sur GPU

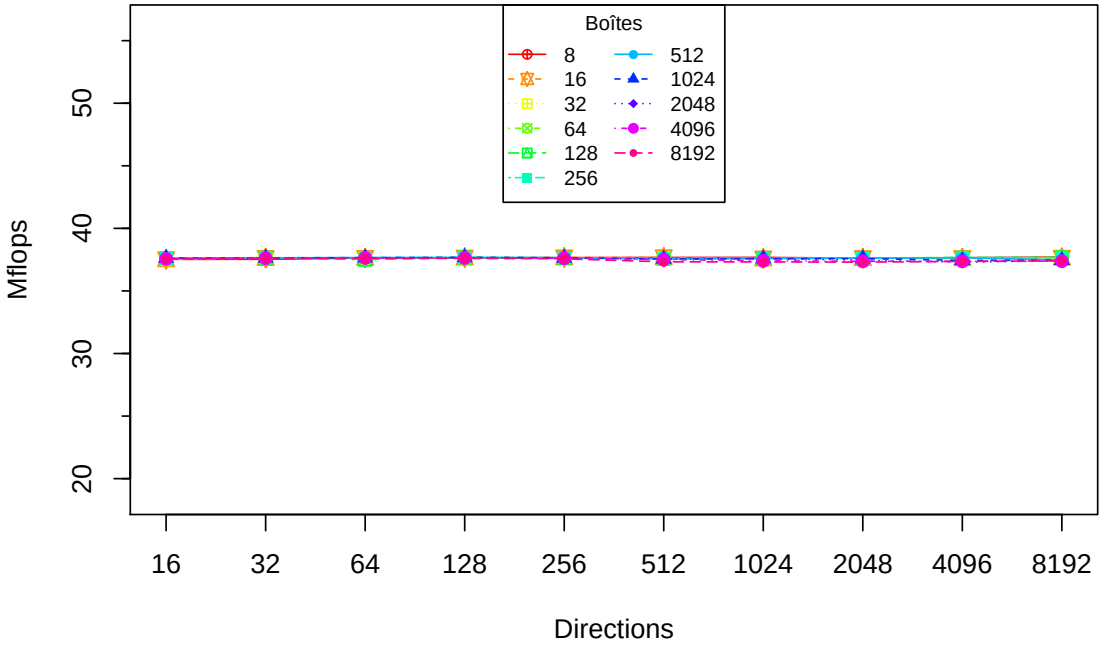


FIGURE 4.6 – Translation : influence de la taille des blocs sur CPU

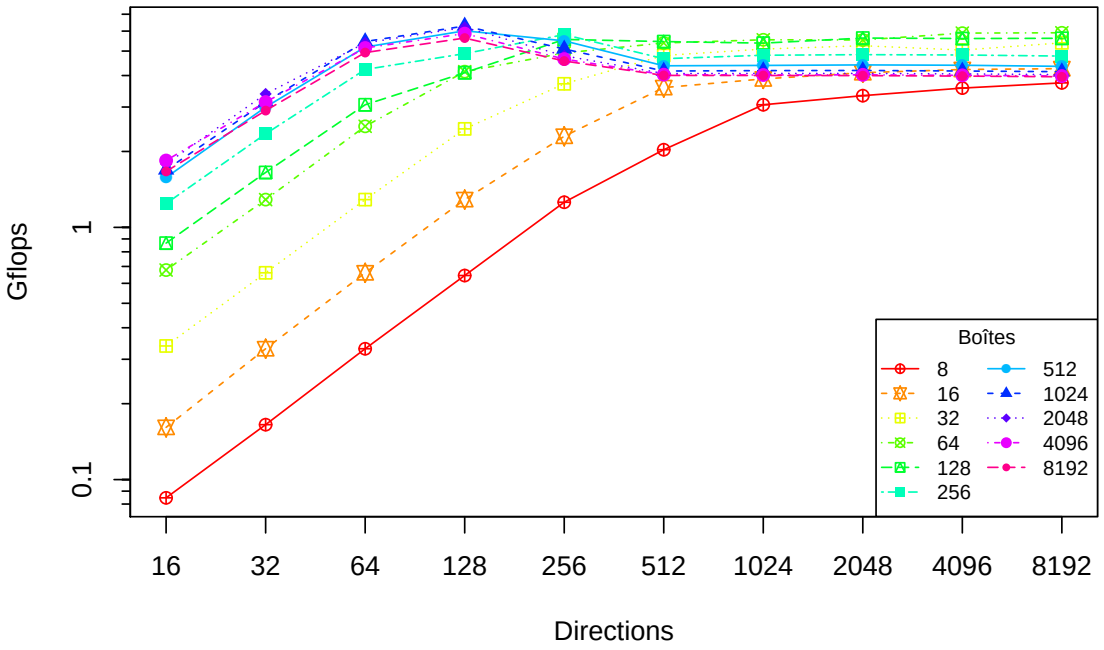


FIGURE 4.7 – Translation : influence de la taille des blocs sur GPU

4.2.2.3 Comparaison CPU/GPU

La performance de la translation est dépendante de la vitesse de la mémoire. En y ajoutant la régularité des calculs, cette situation est très favorable au GPU. La figure 4.8 confirme cette hypothèse. En effet, nous arrivons à obtenir un gain supérieur à 150.

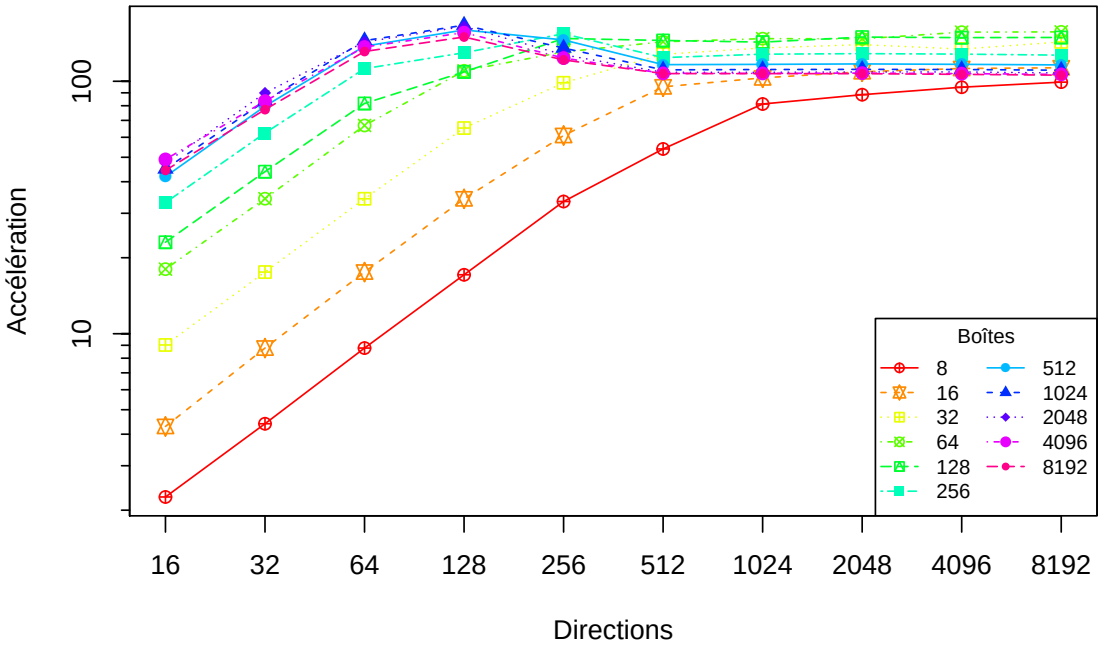


FIGURE 4.8 – Translation : rapport du temps sur CPU par rapport au temps sur GPU

Le choix de la taille du bloc n'est imposé que si nous cherchons les meilleures performances sur GPU. Idéalement, il faut choisir des blocs de taille 256 ko, mais une valeur supérieure peut tout à fait convenir. Ce choix est possible à tous les niveaux de l'octree.

4.2.3 Décalage

Cette opération consiste, comme pour la translation, à effectuer des multiplications terme à terme de vecteurs. La différence importante est que les parcours se font de manière contigüe, ou presque, pour les vecteurs de décalage. Mais, si le nombre de directions le permet, les huit vecteurs de décalage pourront rester dans le cache tout au long du calcul.

4.2.3.1 CPU

Nous avons réalisé trois implémentations du décalage sur CPU. Nous allons étudier l'influence des tailles du bloc, et effectuer une comparaison entre elles. La première implémentation utilise des instructions vectorielles, pour les multiplications. La deuxième effectue un parcours dans l'ordre du vecteur de sortie. Enfin, la troisième et dernière implémentation utilise le même parcours que l'implémentation précédente mais en faisant du tuilage de boucle afin de profiter des avantages de la première, voir la section 3.6.2.1 pour le détail des algorithmes.

Influence de la taille des blocs Nous avons fait varier notre taille de bloc, pour nos trois implémentations. Leur comportement est identique, et la figure 4.9 montre plus en détails celui de la première implémentation. Nous remarquons que le changement de la taille du bloc n'a aucune répercussion sur les performances obtenues.

Comparaison des implémentations Regardons maintenant comment se positionnent les différentes implémentations entre elles. La figure 4.10 nous montre l'efficacité de chaque implémentation

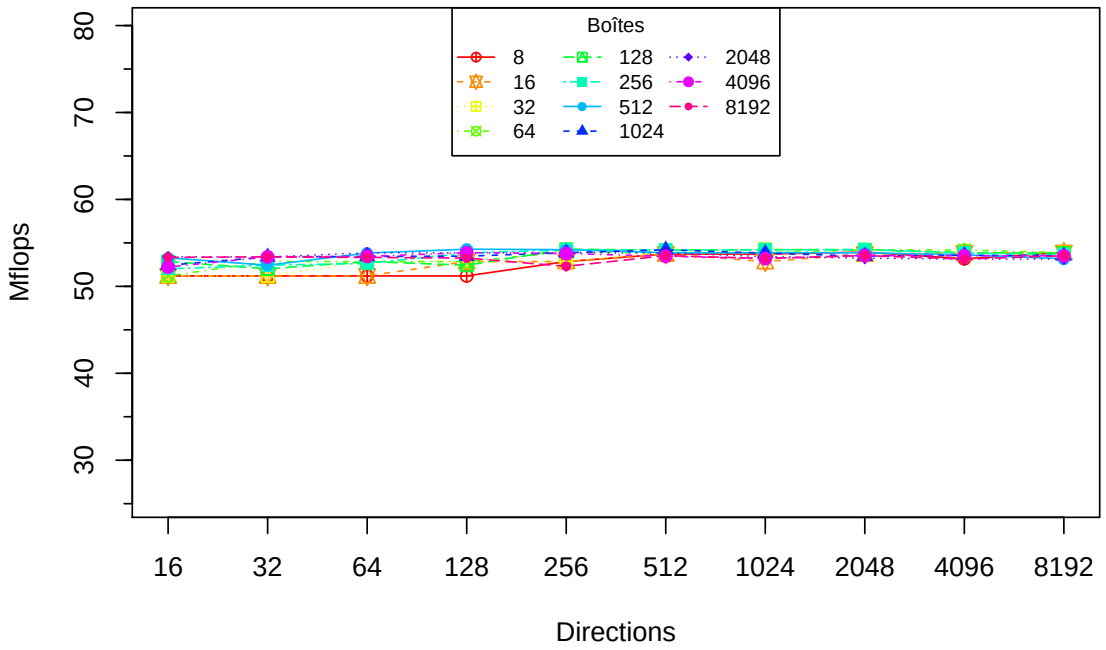


FIGURE 4.9 – Décalage : flops en fonction de la taille du bloc sur CPU, pour la première implémentation

pour 512 boîtes. L'implémentation qui parcourt le vecteur de sortie dans le bon ordre est moins performante que celle qui utilise des opérations vectorielles. L'implémentation la plus efficace est la troisième, qui cumule les avantages des deux premières. Le gain apporté est de 25%. Il est à noter que le compilateur gcc est capable de réaliser lui-même le tuilage de boucle, en activant manuellement l'option *floop-block*.

4.2.3.2 GPU

Pour le GPU, le seul test que nous avons à réaliser est celui mettant en avant l'impact des tailles sur la performance de notre tâche. Les résultats visibles sur la figure 4.11 montrent une augmentation jusqu'à un seuil de 5 Gflops dû au côté *memory bound* de notre calcul (comme avec la translation).

Afin d'avoir une performance optimale, nous devons choisir des blocs d'au moins 4096 boîtes si nous avons moins de 256 directions, c'est-à-dire pour le ou les derniers niveaux suivant le cas. Pour les niveaux au dessus, 1024 boîtes suffisent, et même 512 si nous dépassons 1024 directions.

4.2.3.3 Comparaison CPU-GPU

Comme nous le voyons sur la figure 4.12, le GPU est 100 fois plus rapide. Le gain de performance apporté par le GPU est moins important qu'avec la translation car la régularité des accès et la présence en cache des vecteurs de décalage profitent au CPU.

À nouveau, la taille des blocs n'est dictée que par le GPU. Il est avisé de prendre des blocs de taille 4096 pour le bas de l'arbre, sinon 1024 ou même 512 pour le haut de l'arbre.

4.2.4 Alpert

La méthode d'Alpert est assimilable à un produit matrice-matrice dont la première matrice possède *nombre de boîtes* $*$ *nombre de modes* lignes, et *nombre de* θ *du niveau d'entrée* colonnes, et la deuxième matrice *nombre de* θ *du niveau d'entrée* lignes, et *nombre de* θ *du niveau de sortie* colonnes, le nombre de modes correspondant à $2 * L + 1$ du niveau le plus bas des deux. Cependant, nous n'aurons pas la performance d'un produit matrice-matrice car il y a aussi d'autres opérations telles que des transpositions.

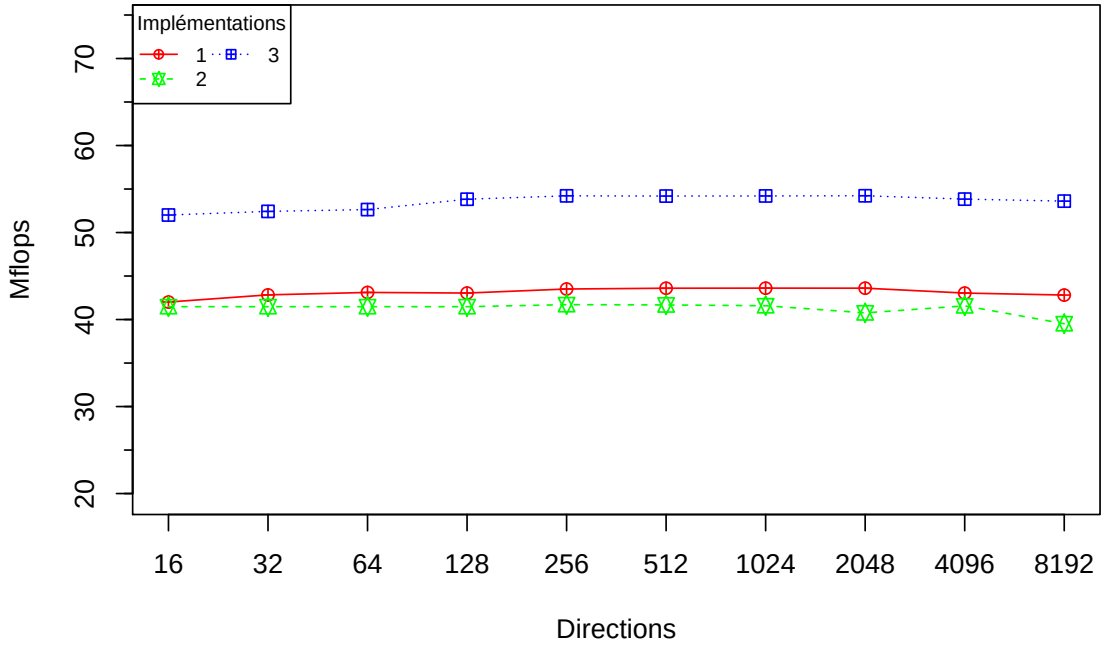


FIGURE 4.10 – Décalage : comparaison des performances des implémentations pour 512 boîtes

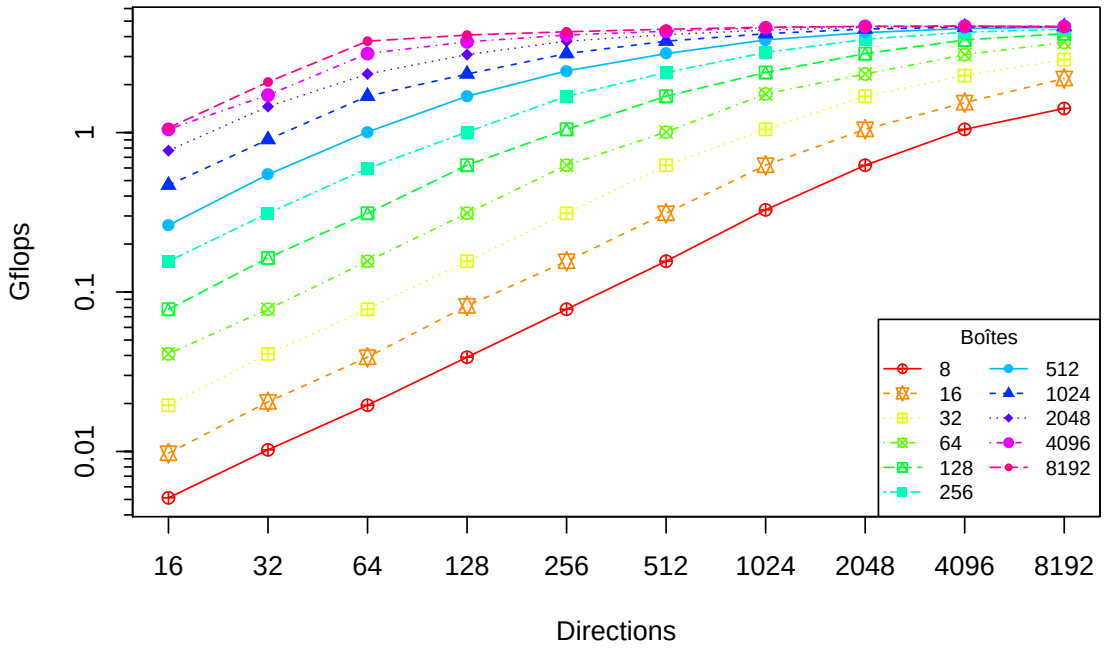


FIGURE 4.11 – Décalage : flops en fonction de la taille du bloc sur GPU

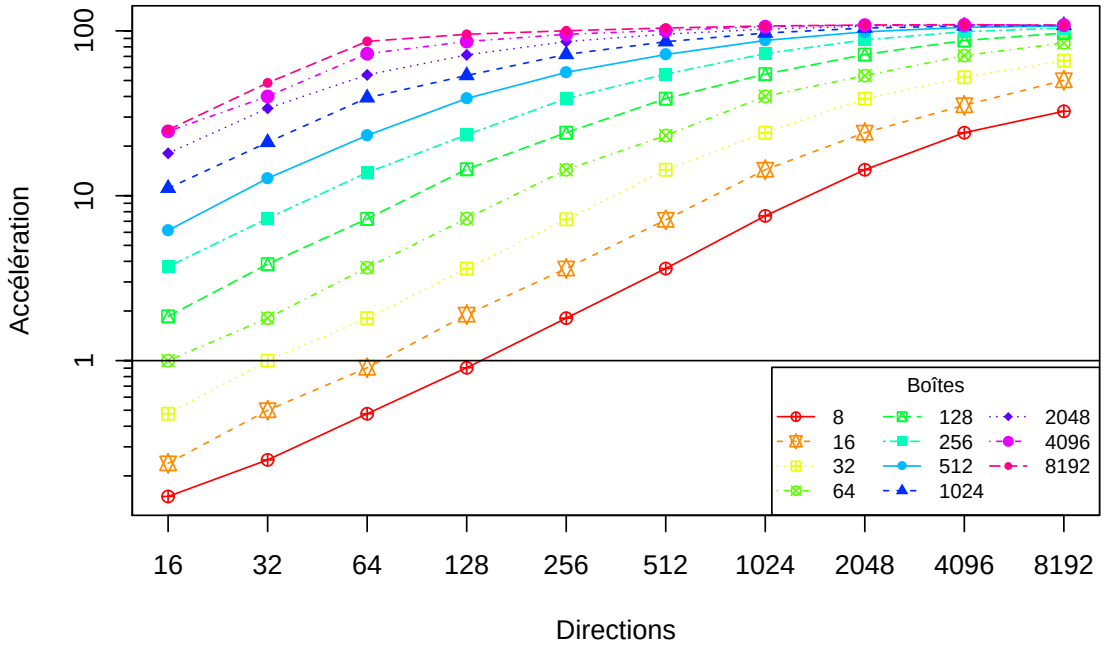


FIGURE 4.12 – Décalage : temps sur GPU par rapport au temps sur CPU

D'après les tailles des matrices, nous avons quatre paramètres. Cependant, dans un but de simplification des tests, nous allons prendre la même valeur pour θ du niveau d'entrée et du niveau de sortie. De plus, nous allons étudier notre tâche pour deux nombres de modes : un petit avec $L = 10$, et un grand avec $L = 128$. Il est à noter que les valeurs prises par θ ne peuvent être supérieures à L (pour le niveau d'entrée).

4.2.4.1 CPU

Nous avons réalisé 5 implémentations de cette tâche, voir la section 3.6.4.1. La première est l'implémentation de base avec des boucles imbriquées. La deuxième est le résultat de la séparation des boucles. La troisième est comme la deuxième version, mais en utilisant des BLAS, notamment pour le produit matrice-vecteur.

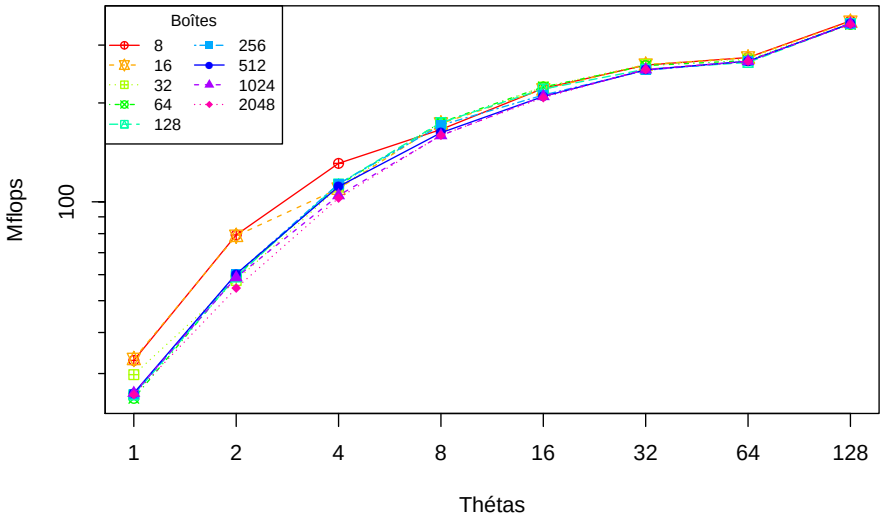
Les quatrième et cinquième versions sont des versions sans transposition préalable mais elles utilisent plusieurs produits matrice-matrice au lieu d'un. Pour la quatrième implémentation, l'appel BLAS *scgemm* est utilisé. Le problème est que le coefficient multiplicateur α qui vaut 1, est déclaré comme complexe, il risque donc d'y avoir des calculs inutiles si la valeur n'est pas testée dans l'appel BLAS. C'est pourquoi nous avons ajouté une cinquième implémentation séparant les parties réelle et imaginaire pour le produit.

Influence de la taille des blocs Cette fois-ci, le calcul dépend du nombre de θ et non simplement du nombre de directions. En effet, nous avons à réaliser différents produits de matrices ayant comme dimension fixe le nombre de modes ($2l + 1$ du niveau bas). Nous avons fait varier la taille des blocs de manière réaliste, dans le sens où la valeur de θ ne peut dépasser la valeur de L.

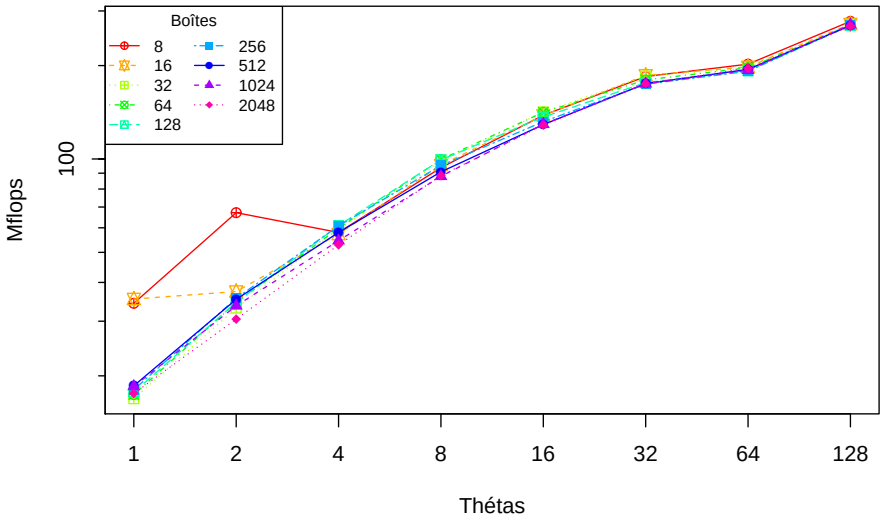
Comme les algorithmes sont différents entre les implémentations, nous avons réalisé le test pour chacune.

Pour un niveau haut La figure 4.13 synthétise ces tests pour $L = 128$, ce qui représente un niveau en haut de l'arbre.

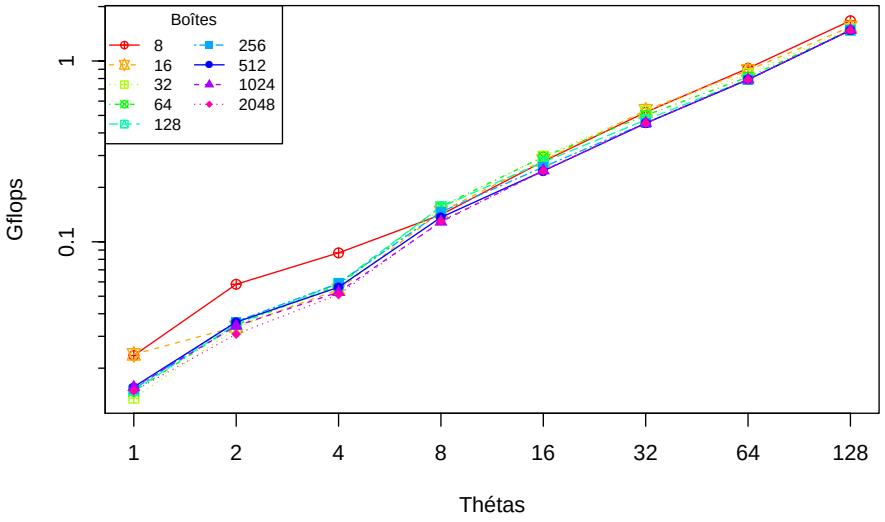
La première remarque est que les deux premières implémentations qui n'utilisent pas les BLAS convergent rapidement, vers une asymptote horizontale. Ce n'est pas le cas des implémentations appelant des BLAS, qui ont une augmentation quasi-linéaire suivant le nombre de θ, et pour lesquels les intervalles de tests ne permettent pas de montrer leur convergence.



(a) Première implémentation

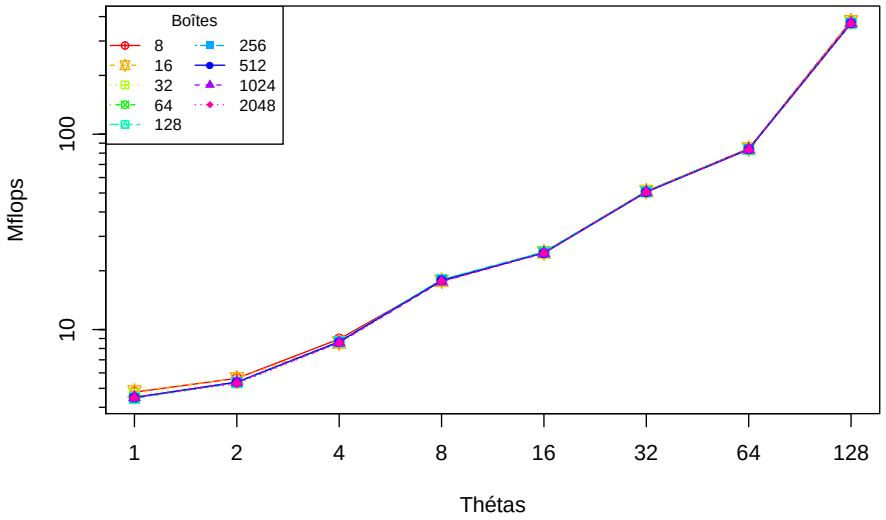


(b) Deuxième implémentation

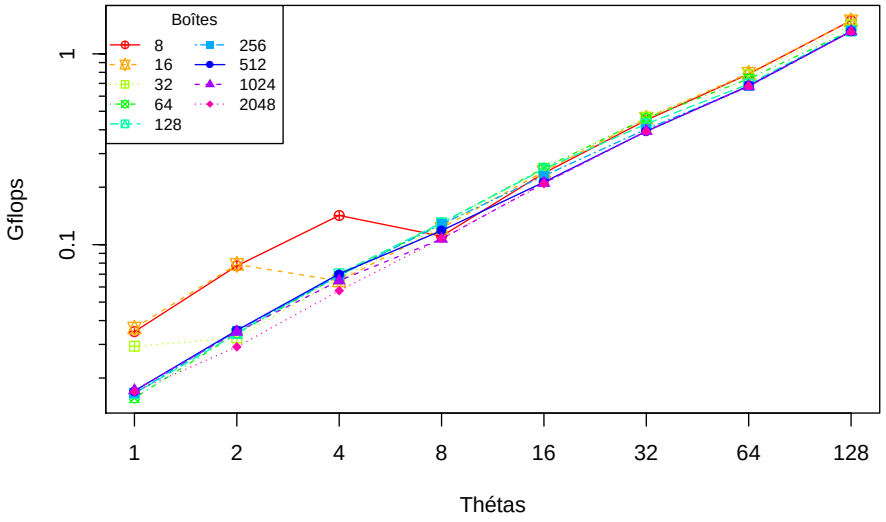


(c) Troisième implémentation

FIGURE 4.13 – Alpert : flops en fonction de la taille des blocs sur CPU pour $L = 128$



(d) Quatrième implémentation



(e) Cinquième implémentation

FIGURE 4.13 – Alpert : flops en fonction de la taille des blocs sur CPU pour $L = 128$

Pour chacune de ses implémentations, seule la valeur de θ modifie la performance du calcul, le nombre de boîtes représentant le nombre de fois où les produits sont effectués.

Pour un niveau bas Nous avons également effectué ces tests pour un niveau situé dans le bas de l'arbre en prenant L valant 10. Les conclusions de ces tests sont équivalentes à celles pour L valant 128.

Comparaison des implémentations Les résultats de la comparaison des implémentations pour L valant 128, sont visibles sur la figure 4.14. Le comportement reste le même pour L valant 10.

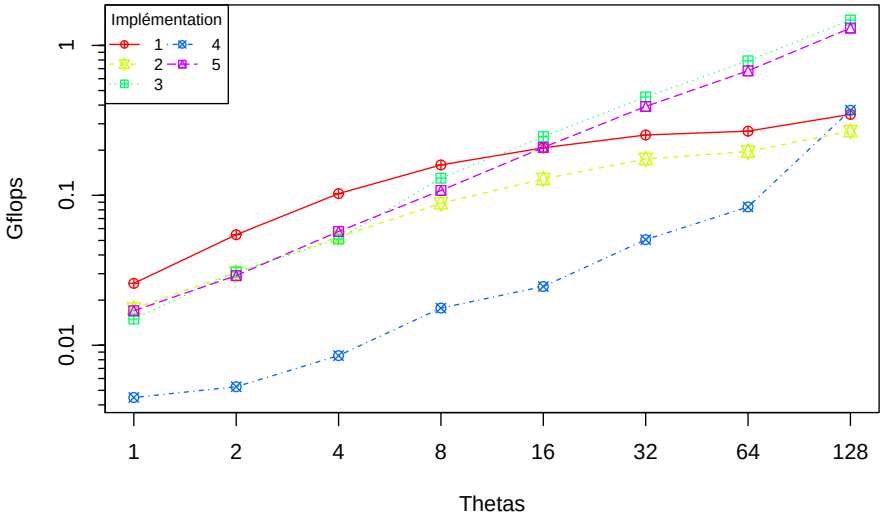


FIGURE 4.14 – Alpert : comparaison des flops pour différentes implémentations sur CPU

La première implémentation est la plus performante jusqu'à une certaine taille, $\theta = 32$. En effet, cette implémentation regroupe tous les calculs dans une seule et même boucle, et comme ceux-ci sont dépendants, cela profite à la localité des caches et explique cette performance. La deuxième implémentation qui réalise des calculs en boucles séparées sans utiliser des BLAS, est naturellement très mauvaise. Les implémentations utilisant un ou plusieurs produits matrice-matrice ont des performances équivalentes, malgré la séparation des parties imaginaires et réelles pour l'une, et sont les plus performantes à partir de $\theta = 432$.

Les résultats de la quatrième implémentation, qui utilise plusieurs produits matriciels mixtes (complexes et réels), sont surprenants. Les mauvais résultats proviennent du fait que la valeur du paramètre α n'est pas testée, ce qui apporte une diminution de la performance d'un facteur valant un peu moins de 6, ce qui correspond au coût d'une multiplication complexe.

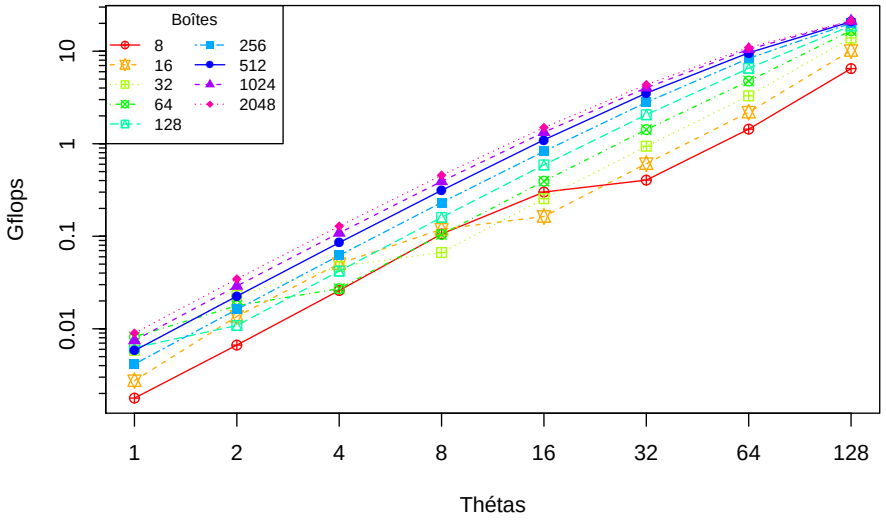
4.2.4.2 GPU

Nous avons réalisé deux implémentations, chacune utilisant des produits matrice-matrice, effectués en séparant les partie réelle et imaginaire, les CUBLAS ne prenant pas en charge les multiplications mixtes réels/complexes. La première implémentation est l'algorithme avec transposition permettant d'effectuer une seule multiplication matricielle. La deuxième implémentation ne comporte pas de transposition et réalise autant de multiplications qu'il y a de boîtes dans le bloc.

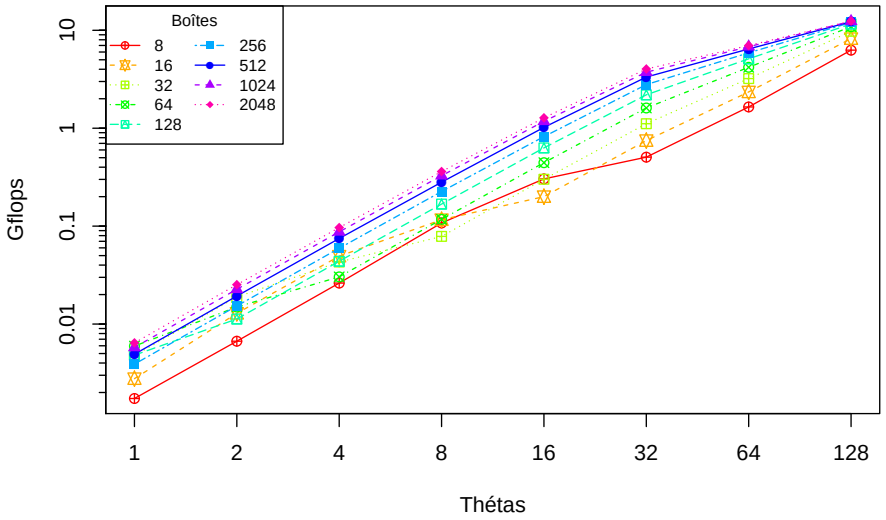
Influence de la taille des blocs Comme avec le CPU, nous avons étudié cette influence dans deux conditions différentes : pour un niveau haut, et pour un niveau bas.

Niveau haut La figure 4.15 montre le comportement des deux implémentations face aux modifications apportées à la taille du bloc. Les deux implémentations ont le même comportement : l'augmentation de performance diffère seulement à partir de θ valant 32, où la deuxième implémentation subit un ralentissement plus important. Concernant le nombre de boîtes, son augmentation

reste bénéfique même si elle ralentit au-delà de 1024 boîtes. Concernant le nombre de θ il serait optimal d'augmenter sa valeur au delà de 128, mais ce n'est pas possible.



(a) Première implémentation



(b) Deuxième implémentation

FIGURE 4.15 – Alpert : flops en fonction de la taille des blocs sur GPU

Niveau bas Les résultats donnent des allures de courbes identiques même si le ralentissement dans le gain de vitesse se déclare dès 128 boîtes.

Comparaison des implémentations Afin de confronter les deux implémentations, nous avons choisi de montrer les résultats pour des blocs de 512 boîtes en faisant varier θ. D'après la figure 4.16, les deux implémentations ont des performances quasiment similaires jusqu'à $\theta = 32$ où elles commencent à diverger, en faveur de l'implémentation ne réalisant qu'un seul produit matriciel.

4.2.4.3 Comparaison CPU-GPU

En utilisant les implémentations les plus performantes pour le CPU et le GPU, nous avons établi une comparaison en fonction de la taille des blocs pour $L = 128$, voir la figure 4.17. Sur GPU, nous avons un gain quand le nombre de boîtes est supérieur à 512 à partir de θ valant 4, et quel que soit le nombre de boîtes pour θ supérieur à 32.

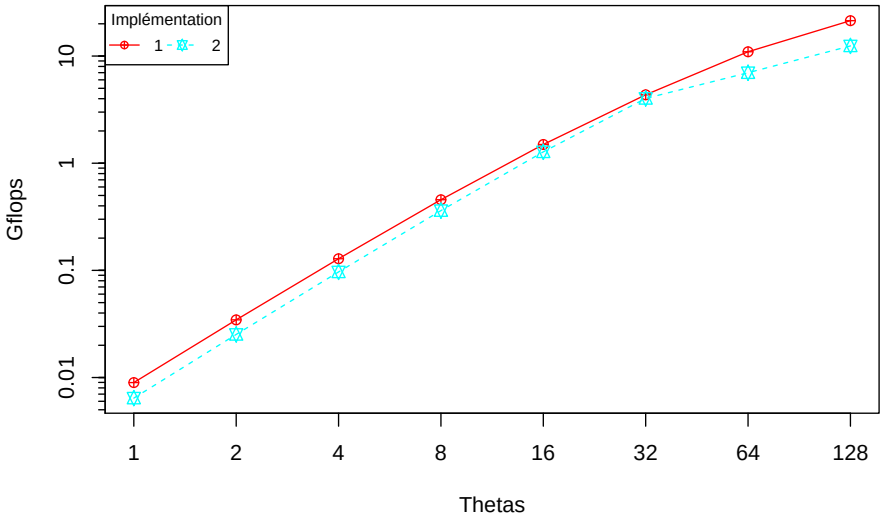


FIGURE 4.16 – Alpert : Comparaison des temps pour différentes implémentations sur GPU

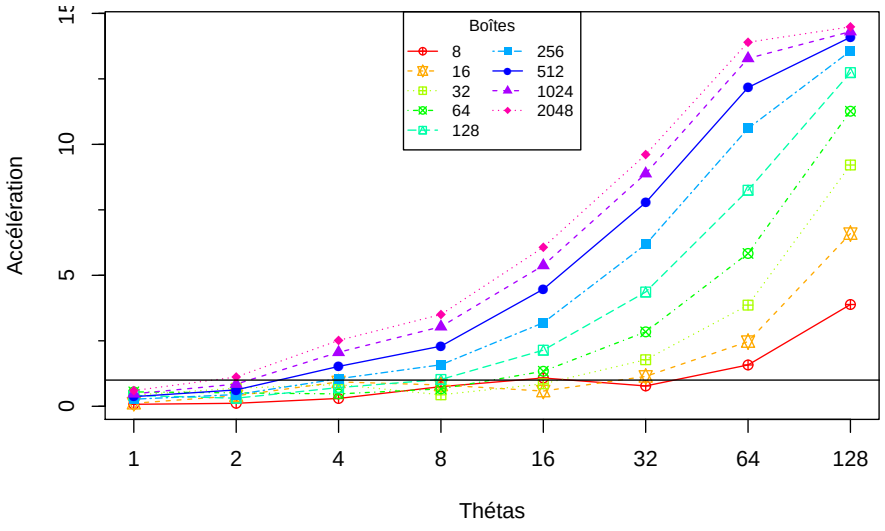


FIGURE 4.17 – Alpert : rapport du temps sur CPU par rapport au temps sur GPU pour $L = 128$

Pour $L = 10$, nous avons les mêmes résultats que quand nous prenons des valeurs de θ inférieures à 10, c'est-à-dire des résultats incontestablement en faveur du CPU, voir la figure 4.18.

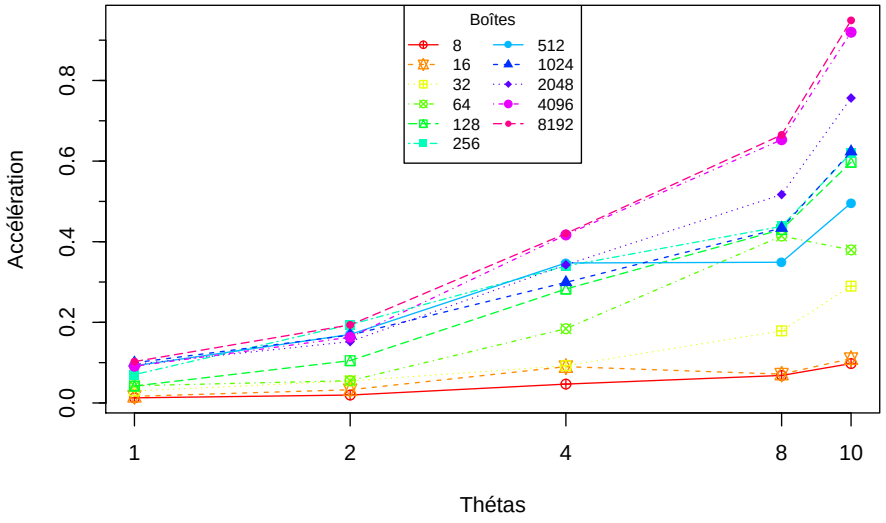


FIGURE 4.18 – Alpert : rapport du temps sur CPU par rapport au temps sur GPU pour $L = 10$

Quand nous sommes en bas de l'arbre, le CPU est plus avantageux car le nombre de θ est faible. Quand nous sommes en haut de l'arbre, le GPU est plus avantageux mais avec un impact moindre, du fait du nombre de boîtes qui est limité.

Il est intéressant d'avoir le maximum de θ afin d'avoir les meilleures performances sur GPU et sur CPU, avec dans ce cas un léger avantage pour le GPU. Ce dernier sera alors plus performant que le CPU sur cette tâche seulement dans les niveaux situés dans le haut de l'arbre.

4.2.5 Transformées de Fourier

Cette étape est une boucle sur les boîtes et sur les θ réalisant des transformées de Fourier de taille ϕ. Ainsi, il nous est nécessaire d'étudier l'influence d'un seul paramètre : le nombre de transformées de Fourier à réaliser.

Cependant, il y a un autre paramètre qui rentre en jeu. En effet, à la fois sur CPU et sur GPU, les bibliothèques FFTW et CuFFT permettent de réaliser plusieurs FFT en un seul appel en se passant d'une boucle. L'impact sur la mémoire de ce regroupement est connu : la taille de la zone de travail servant à calculer la FFT est égale à la taille de la transformée multipliée par le nombre à réaliser.

Or, comme la zone de travail est allouée lors de l'initialisation, l'impact peut se révéler important. C'est pourquoi nous avons choisi de tester l'agrégation de tous les appels provenant de la boucle en θ, avec autant d'appels qu'il y a de boîtes. Cette solution permettrait de ne pas utiliser trop de mémoire, sachant que la zone mémoire peut être utilisée pour plusieurs blocs de directions. Une agrégation par bloc de boîtes permettrait une réutilisation difficile de la zone mémoire car d'un bloc de boîtes à l'autre, le nombre de boîtes varie. En effet, même imposé, comme les blocs ne contiennent que des fratries entières, il y a des différences entre les tailles de blocs de boîtes. Par conséquent, nous allons mesurer l'impact sur les performances, afin de savoir si l'agrégation selon θ est intéressante.

Outre la transformée de Fourier, le calcul comporte une étape postérieure de recopies partielles, avec multiplication par un scalaire, du vecteur issu de la transformée.

4.2.5.1 CPU

Nous avons trois types de résultats à présenter. D'abord, l'évolution des performances face à l'augmentation du nombre de FFT à réaliser. Ensuite, nous analyserons l'intérêt d'intégrer plusieurs transformées dans un seul appel. Et enfin, nous confronterons nos implémentations.

Influence de la taille des blocs Nous avons réalisé trois implémentations de notre étape de transformée de Fourier, une par modification de la méthode de recopie des blocs après la transformée de Fourier. La première est une recopie manuelle, la deuxième est une recopie en deux blocs discontinus, et la troisième et dernière est une recopie en trois étapes qui sont le début, le milieu et la fin.

Dans ce but, nous nous sommes placés dans deux cas, $L = 10$ et $L = 128$ pour voir le comportement lorsque la taille de la transformée est petite et quand elle est assez grande (par rapport à notre situation).

Les résultats du premier cas sont présentés sur la figure 4.20. L'influence de la taille du bloc n'est présente que pour des valeurs faibles, inférieures à 32. Cela s'explique par l'étape de recopie. De plus, θ et le nombre de boîtes ont une influence équivalente. Pour le cas $L = 128$, voir la figure 4.19, le comportement est le même mais le palier est atteint plus rapidement.

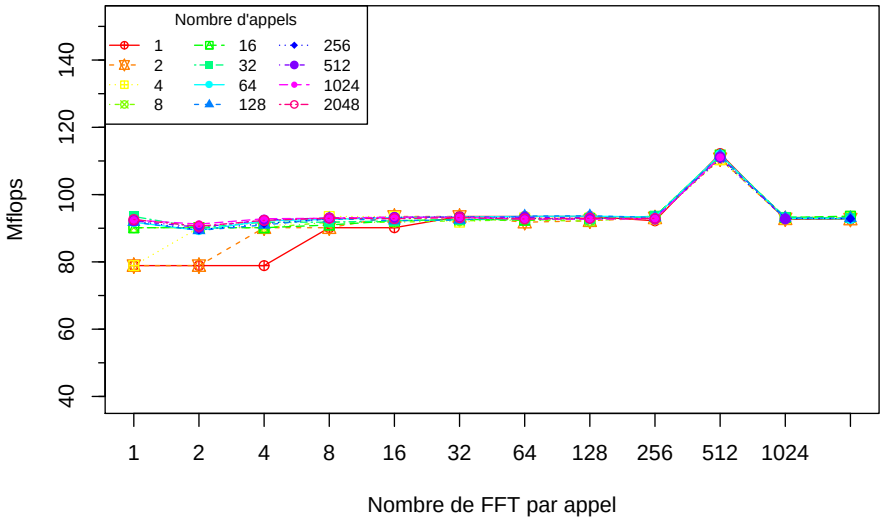


FIGURE 4.19 – Fourier : flops en fonction de la taille du bloc sur CPU avec $L = 10$ pour la première implémentation

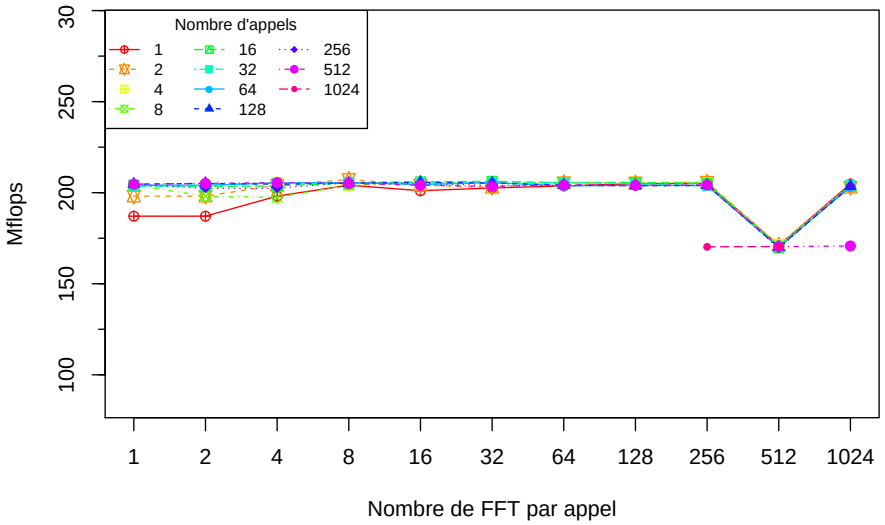


FIGURE 4.20 – Fourier : flops en fonction de la taille du bloc sur CPU avec $L = 128$ pour la première implémentation

Influence de l'agrégation des transformées Comme nous l'avons déjà précisé, la bibliothèque FFTW permet d'agréger plusieurs FFT en un seul appel de fonction. L'avantage est la suppression du temps d'appel de la fonction, ce qui devrait représenter un gain négligeable. Afin de nous en rendre compte, nous avons réalisé un test dans lequel nous avons fait varier le nombre de FFT par appel mais également le nombre d'appels. Ainsi, nous pouvons comparer l'influence des deux variables. Pour cela, nous avons représenté sur la figure 4.21, pour $L = 128$, le rapport entre le temps du la version privilégiant le nombre de transformées par appel, et le temps de celle privilégiant le nombre d'appels. Le gain obtenu est nul ce qui va dans le sens de notre hypothèse. Cela se vérifie également pour $L = 10$.

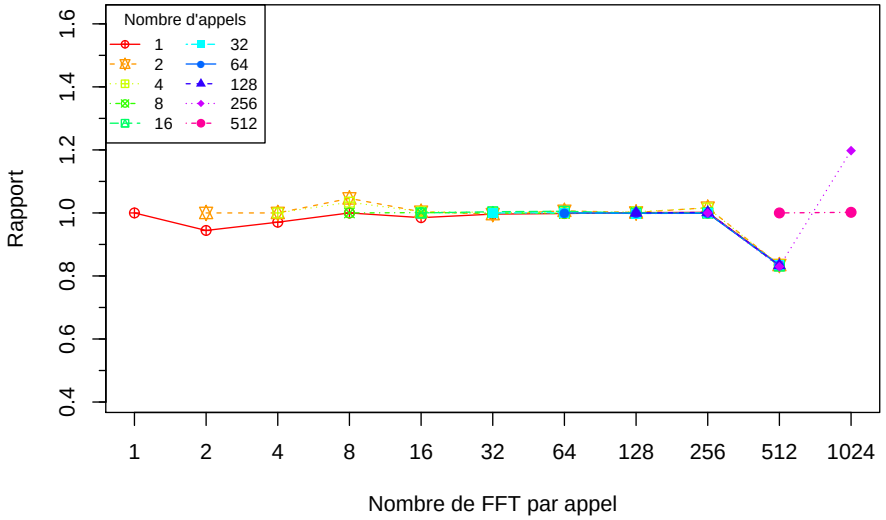


FIGURE 4.21 – Fourier : rapport entre le temps de la version privilégiant le nombre de transformées par appel, et le temps de celle privilégiant le nombre d'appels, avec $L = 128$ pour la première implémentation

Comparaison des implémentations La différence entre les implémentations concerne uniquement la partie de recopie, voir la section 3.6.4.1. Afin de les comparer, nous avons choisi deux valeurs de L, 10 et 128, et nous avons à nouveau fait varier la taille des blocs. Notre première implémentation est une copie faite « à la main », c'est-à-dire sans utilisation de bibliothèque. Les deuxième et troisième implémentations utilisent *mkl _somatcopy*, en faisant respectivement deux copies, et une grosse copie.

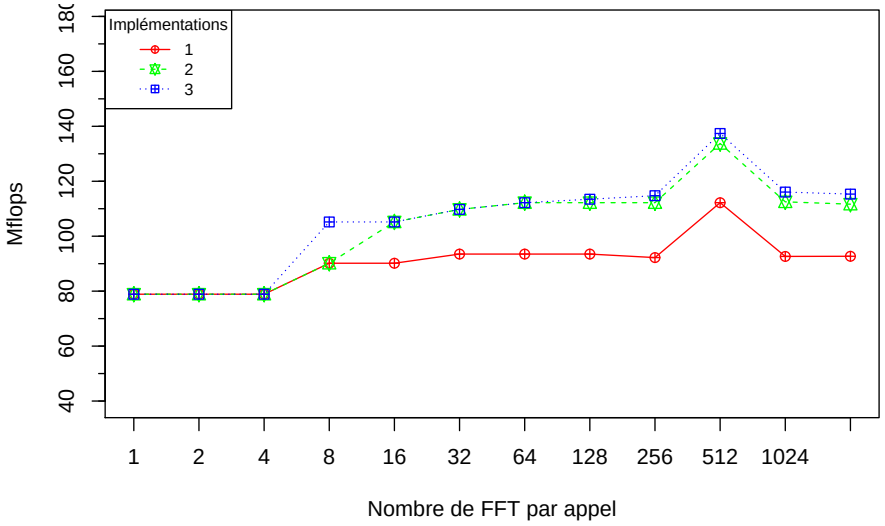
Les résultats visibles à la figure 4.22 donnent un léger avantage aux deux copies utilisant la MKL, sans grande différence entre les deux.

4.2.5.2 GPU

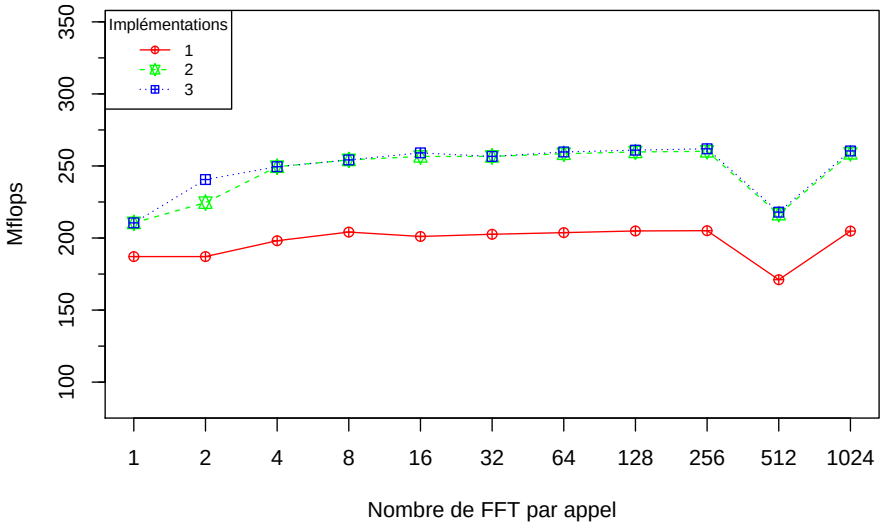
Comme avec le CPU, nous allons étudier l'impact du changement de la taille du bloc, et approfondir l'impact de l'agrégation des transformées dans un seul appel. Nous aurons également à comparer nos deux implémentations.

Influence des tailles de boucles Nous avons, comme avec le CPU, effectué des tests pour deux valeurs de L : 10 pour un niveau en bas de l'arbre et 128 pour un niveau assez en haut. Les résultats du niveau bas sont présentés à la figure 4.23, pour la première implémentation mais le comportement est le même pour la deuxième.

L'augmentation du nombre de directions ou du nombre de boîtes permet d'améliorer l'efficacité de notre tâche jusqu'à un seuil, qui est atteint à partir de 256 θ et 128 boîtes. Augmenter le nombre de directions au-delà de 128 n'a aucune répercussion.



(a) $L = 10$



(b) $m = 128$

FIGURE 4.22 – Fourier : comparaison des différentes implémentations sur CPU

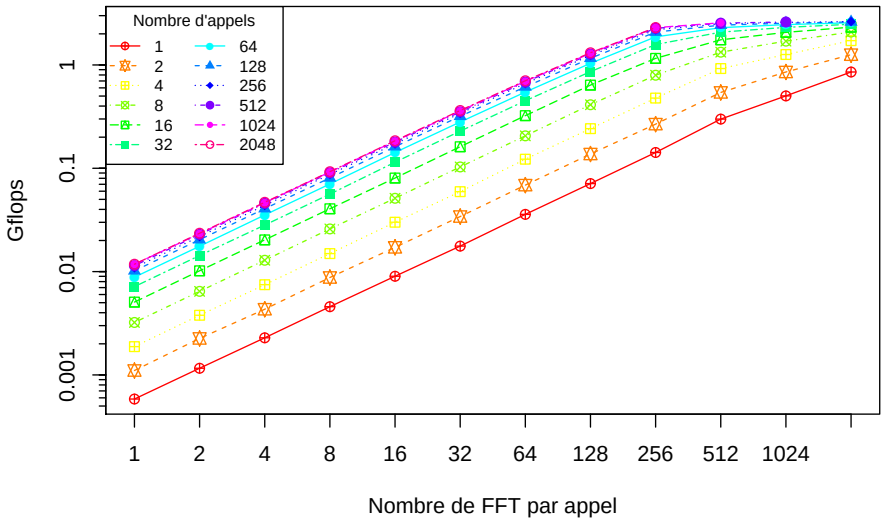


FIGURE 4.23 – Flops de l'étape de la transformée de Fourier en fonction de la taille des blocs sur GPU pour $L = 10$, pour la première implémentation

Pour le niveau en haut de l'arbre, voir la figure 4.24, nous avons le même comportement mais le ralentissement de l'augmentation se retrouve plus tôt : 64 θ et 32 boîtes. Il est donc possible de construire des blocs permettant d'obtenir de bonnes performances.

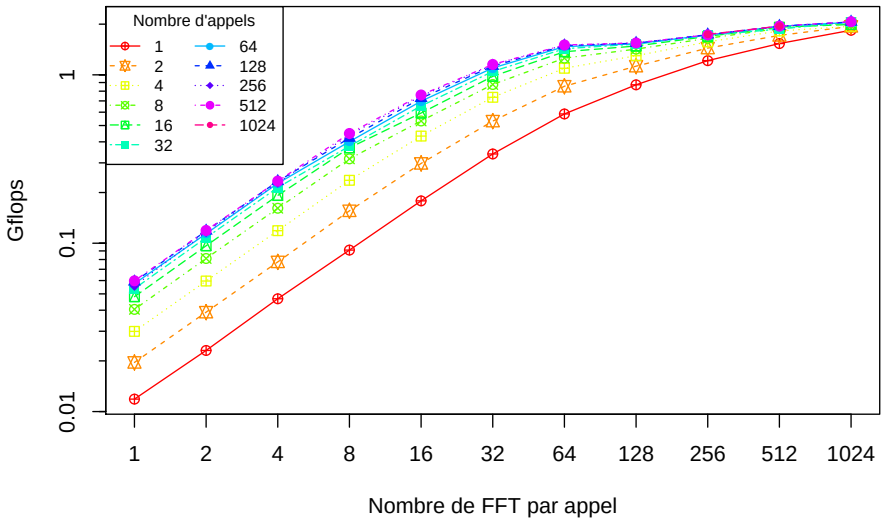


FIGURE 4.24 – Flops de l'étape de la transformée de Fourier en fonction du nombre de boucles sur GPU avec $L = 128$ pour la première implémentation

Influence de l'agrégation des transformées Avec la bibliothèque CUFFT, il est également possible de réaliser plusieurs FFT en un seul appel. L'avantage par rapport au CPU, est que le coût du lancement du noyau sur le GPU coûte plus cher, et que même s'il est possible de lancer plusieurs noyaux en même temps (à l'aide de *streams* afin de masquer les latences des accès mémoire), agréger les appels semble plus favorable à réduire les latences.

Les figures 4.25 et 4.26 présentent les gains obtenus en choisissant d'agréger les transformées plutôt qu'en multipliant les appels. Nous voyons qu'il est toujours préférable de choisir l'agrégation. Cependant, ces résultats montrent également que le gain de l'agrégation diminue avec le nombre d'appels. Notre choix d'agréger les transformées pour la boucle en θ semble donc être un bon compromis.

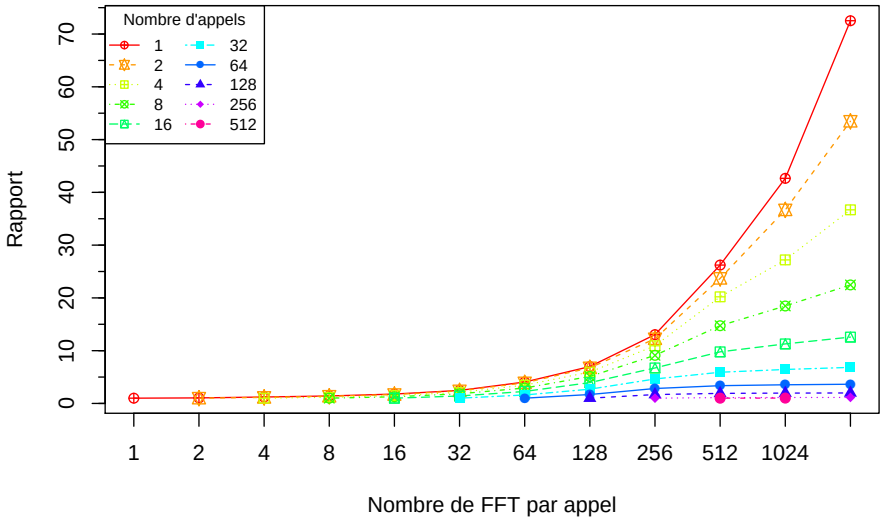


FIGURE 4.25 – Gains obtenus en privilégiant le nombre de FFT par appel plutôt que le nombre d'appels, sur GPU avec $L = 10$

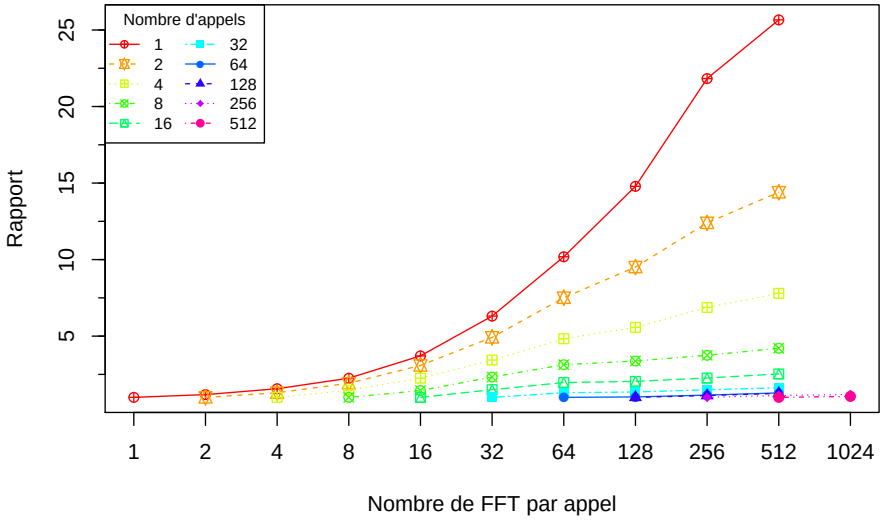
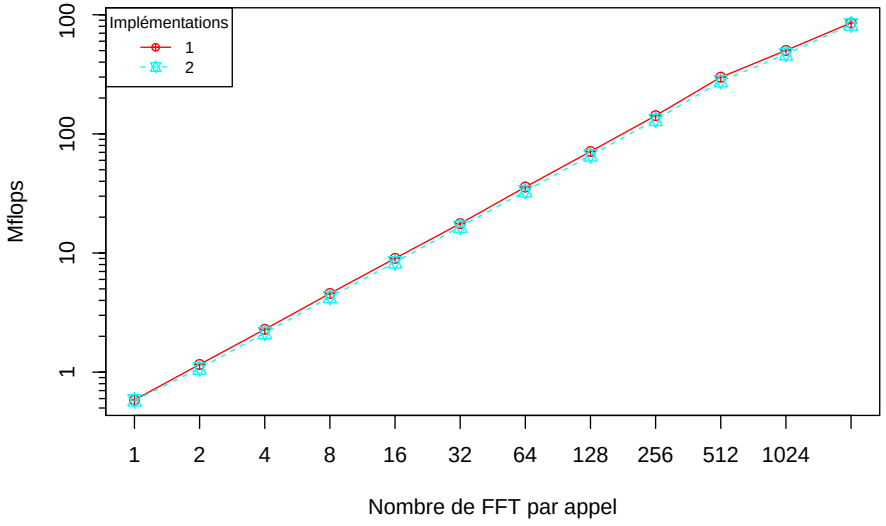


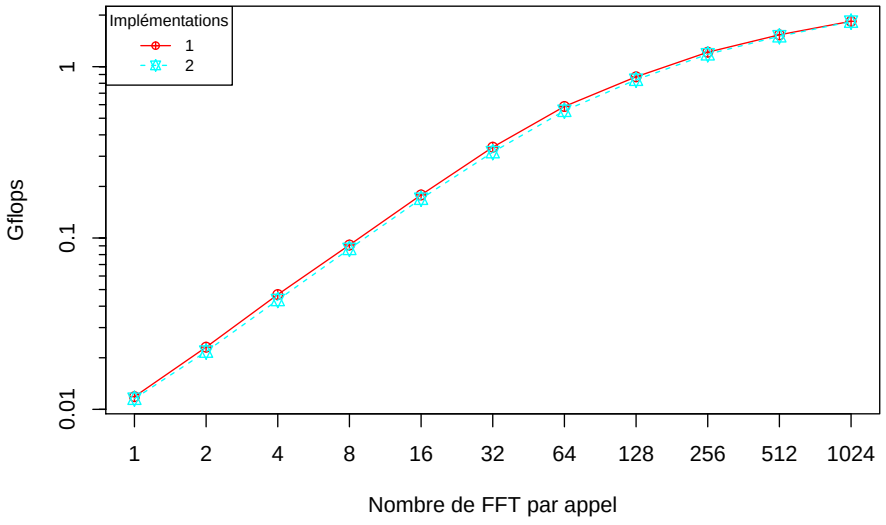
FIGURE 4.26 – Gains obtenus en privilégiant le nombre de FFT par appel plutôt que le nombre d'appels, sur GPU avec $L = 128$

Comparaison des implémentations Nous avons réalisé deux implémentations de notre étape de transformée. La première est une recopie en deux blocs discontinus, et la deuxième est une recopie en trois étapes qui sont le début, le milieu et la fin.

Afin de les comparer, nous avons choisi comme précédemment deux valeurs de L, 10 et 128, et avons procédé à des mesures de temps en fonction du nombre de transformées de Fourier à effectuer en un seul appel pour voir davantage les effets de la recopie. Les résultats sont visibles à la figure 4.27,



(a) $L = 10$



(b) $L = 128$

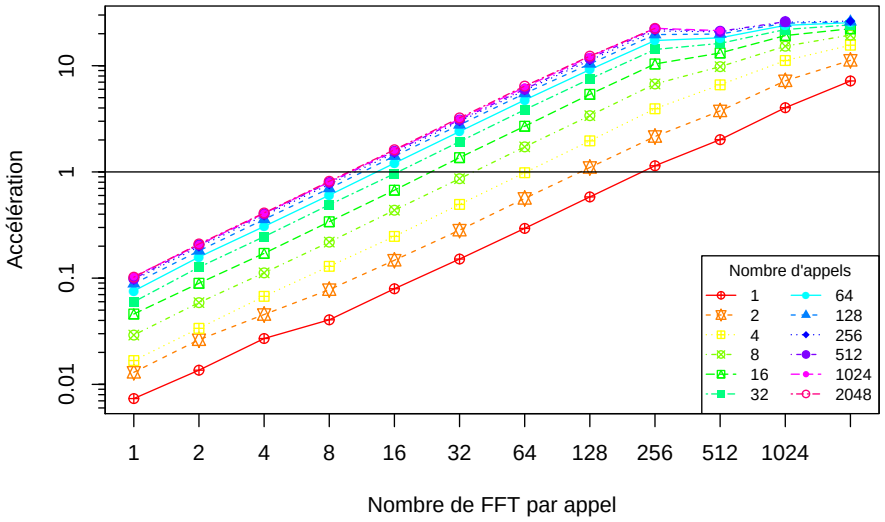
FIGURE 4.27 – Fourier : comparaison des différentes implémentations sur GPU

La deuxième implémentation, qui utilise une grosse recopie et deux petites, est très légèrement plus performante.

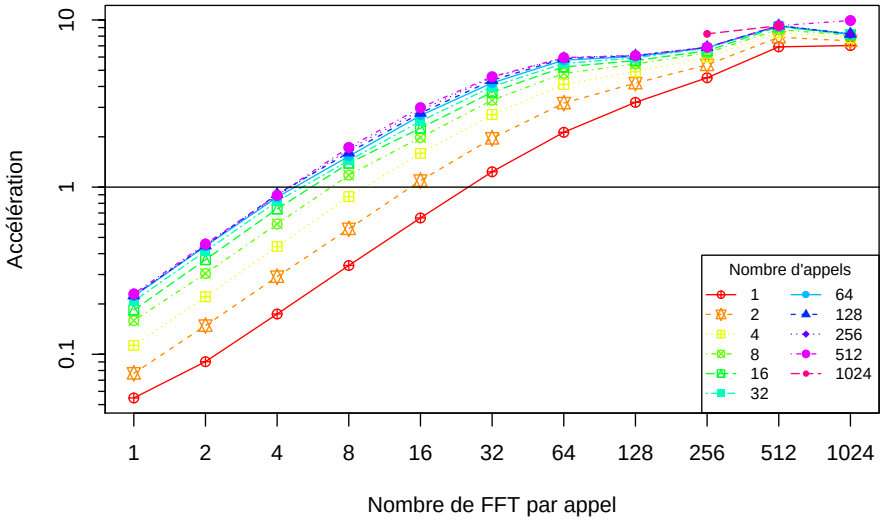
4.2.5.3 Comparaison CPU-GPU

La figure 4.28 montre l'accélération apportée par le GPU en fonction de la valeur de L. Pour $L = 10$, comme nous avons choisi d'agréger suivant la boucle en θ nous ne pourrons dépasser 10 FFT par appel, ce qui est insuffisant pour permettre d'avoir un calcul plus efficace sur GPU.

Pour $L = 128$, nous pouvons obtenir, sans trop de contraintes, un gain de 7 sur GPU, le maximum étant 10, ce qui reste assez faible.



(a) $L = 10$



(b) $L = 128$

FIGURE 4.28 – Fourier : comparaison du temps CPU par rapport au temps GPU

L'agrégation des transformées de Fourier de la boucle en θ en un seul appel est intéressante pour le haut de l'arbre, même si l'exécution sur GPU apporte un gain assez faible.

4.2.6 Interactions proches

Le calcul des interactions proches peut se faire, comme avec les agrégations, en précalculant les matrices des interactions. Dans ce cas, le calcul est un produit matrice-vecteur. Avec ce précalcul, le temps consacré aux interactions proches est réduit, il est donc intéressant de diminuer la profondeur de l'arbre. Cependant, le coût de stockage des matrices augmente et peut devenir important. En effet, il faut, pour chaque paire de boîtes voisines, calculer une matrice de taille le nombre de points d'une boîte. La taille totale représentée par les matrices est alors égale au nombre de points, multiplié par le nombre de voisins par boîte, multiplié par le nombre de points par boîte. En général, cela représente plus que pour les matrices d'agrégation. De plus, nous avons choisi d'utiliser les interactions proches comme tampons afin de masquer les attentes. Nous les distribuons dynamiquement entre les nœuds, mais il faudrait que chaque nœud possède ces matrices. C'est pourquoi nous avons choisi de calculer ces interactions directement, sans précalcul.

Pour la méthode directe, le calcul consiste à déterminer l'exponentielle complexe d'un nombre dépendant de leur distance, pour chaque paire de points de deux boîtes voisines.

Afin de réaliser ces tests, nous avons fait varier le nombre de points par boîte et le nombre de boîtes, en prenant un nombre de voisins par boîte valant 10.

4.2.6.1 CPU

La figure 4.29 montre l'efficacité en fonction du nombre de points par boîte et du nombre de boîtes par bloc. Le nombre de boîtes servant à réitérer les opérations, il est évident qu'il n'a pas d'influence sur la performance des calculs. Concernant le nombre de points, en dessous de 16, nous n'atteignons pas le palier à cause de la surcharge de la mesure de temps, de chargement des données non amorti et de l'incapacité d'utiliser au maximum des instructions vectorielles. Au-delà, nous atteignons un palier de 320 Mflops.

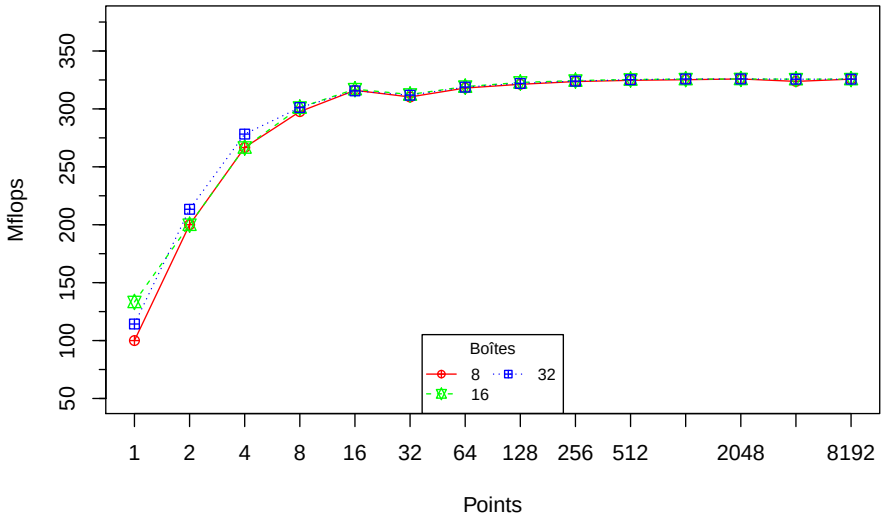


FIGURE 4.29 – Interactions proches : flops en fonction du nombre de points par boîte et du nombre de boîtes par bloc sur CPU

4.2.6.2 GPU

Un calcul d'une interaction proche représente 25 flops pour 2 points (source et destination) chargés en mémoire. De plus, le point de destination est réutilisé pour chaque point source voisin. Cette opération est donc la plus intéressante de la méthode multipôle concernant le nombre d'opérations par nombre de chargements en mémoire. Du coup, les accès mémoire vont facilement être

recouverts. D'ailleurs, le faible gain offert par l'utilisation de *streams* différents par boîte (comme pour l'agrégation), visible sur la figure 4.30, confirme cela. Dans notre noyau, chaque thread travaille sur des points sources différents, il est donc normal qu'il y ait une augmentation importante jusqu'à un certain seuil. Celui-ci se produit pour 1024 points.

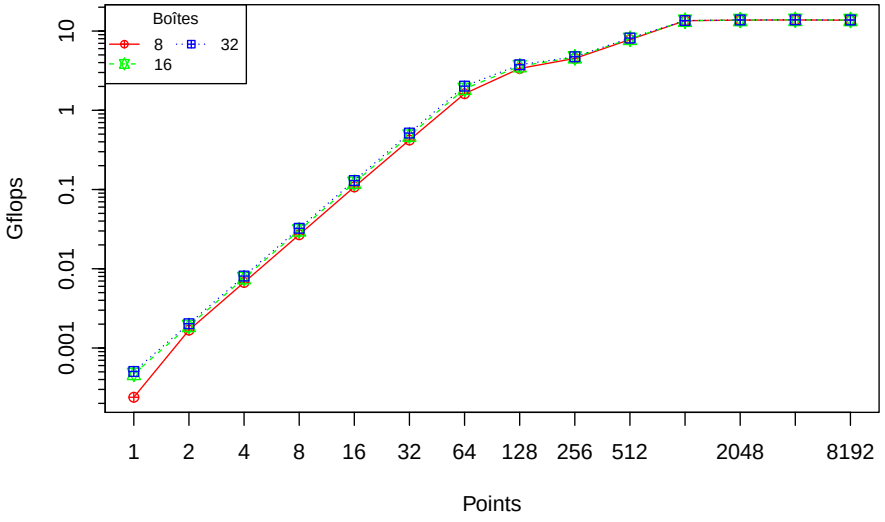


FIGURE 4.30 – Interactions proches : flops en fonction du nombre de points par boîte et du nombre de boîtes par bloc sur GPU

4.2.6.3 Comparaison CPU-GPU

La figure 4.31 effectue une comparaison du temps d'exécution sur CPU par rapport à celui sur GPU. Quand le nombre de points est inférieur à 25, l'exécution sur CPU est bien plus performante d'autant plus que le nombre de points est faible. C'est seulement au-delà que le GPU devient plus avantageux permettant obtenir un gain maximal de 40 pour 1024 points.

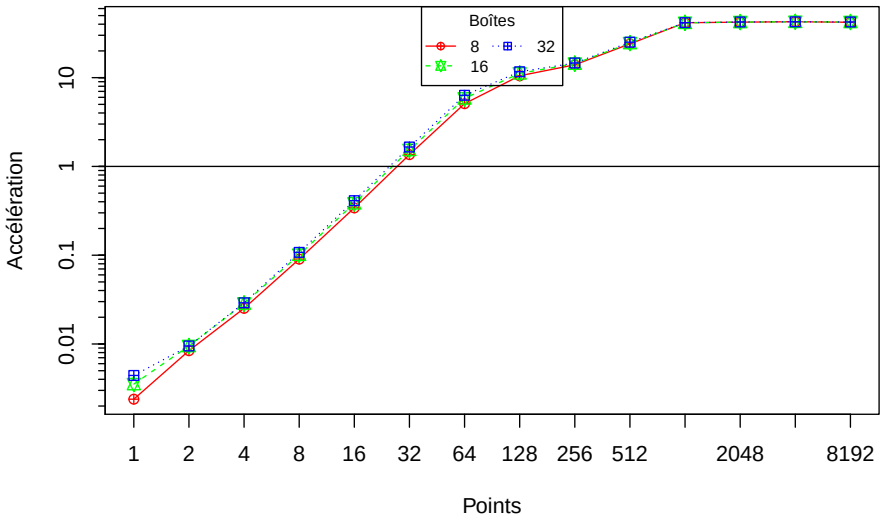


FIGURE 4.31 – Interactions proches : comparaison du temps CPU par rapport au temps GPU

Un si grand nombre de points par boîte est accessible seulement quand l'arbre n'est pas trop profond. Dans ce cas, nous pourrions utiliser les GPU, mais nous perdrions les avantages de la méthode multipôle. Par conséquent, pour cette tâche, l'exécution sur GPU sera peu ou pas avantageuse. Le choix d'exécution sera bien entendu réalisé par l'ordonnanceur. Ceci est seulement vrai

avec le calcul des interactions proches sans précalcul.

4.2.7 Conclusion

Il y a plusieurs éléments importants à retenir de l'étude des noyaux composant la méthode multipôle. Tout d'abord, l'augmentation de la taille des blocs n'est profitable qu'au GPU, sauf avec la méthode d'Alpert qui a besoin d'un grand nombre de θ pour être performante même sur CPU.

Les tâches sont généralement favorables au GPU mais deux tâches sont à retenir : la translation et le décalage. Elles peuvent respectivement trouver un gain entre 100 et 150, et 80 sur GPU. Pour cela, la translation n'a pas besoin de conditions extrêmes : 512 boîtes et si possible 64 directions. Ces conditions sont acceptables pratiquement dans tout l'octree, sauf en haut de l'octree où nous pourrons prendre davantage de directions pour compenser le faible nombre de boîtes. De même, les besoins du décalage sur GPU s'obtiennent aisément.

De son côté, le CPU est très avantageux par rapport au GPU pour l'agrégation, le calcul des interactions proches pour un faible nombre de points, et les transformées de Fourier surtout dans le bas de l'arbre.

4.3 Exécution séquentielle

Maintenant que nous connaissons le comportement de chacune de nos tâches, nous pouvons étudier le comportement de l'algorithme séquentiel, sur CPU et sur GPU

4.3.1 Poids des tâches

Nous avons mesuré l'impact des différents paramètres sur les performances de nos tâches à la section précédente. Cependant, ce qui nous intéresse est la performance de l'algorithme tout entier. Nous devons alors mesurer l'importance des tâches dans l'algorithme multipôle.

Si, nous prenons une taille de bloc importante afin de rendre une tâche performante, nous aurons alors peu de blocs. Cependant, même si cette tâche va pouvoir s'exécuter de manière optimale, il ne faut pas oublier que notre application est parallèle, et augmenter la taille des blocs entraîne une réduction de leur nombre. Ainsi, avec des gros blocs, nous manquerons de tâches pour satisfaire toutes nos unités de calcul, qui auraient pu se contenter de blocs plus réduits pour d'autres tâches plus importantes. Notre choix semble alors mauvais, d'autant plus si cette tâche ne représente qu'une partie minime du temps de l'algorithme. C'est pourquoi, il est important de regarder le poids de chaque tâche dans l'algorithme, afin de conditionner nos choix.

Pour connaître ce poids, nous pourrions réaliser une étude théorique sur le nombre d'opérations produites par chaque tâche. Cependant, comme nous l'avons vu précédemment, l'efficacité dépend de la taille des données et du type de GPU ou CPU utilisé. En conséquence, nous allons étudier le poids des tâches grâce à des mesures expérimentales, qui sont elles aussi sensibles aux différents paramètres, mais qui nous donnerons quand même un ordre d'idée qui pourra si besoin être complété par les résultats de nos autres tests.

Nos tests ont consisté à mesurer à chaque niveau, le temps représenté par chaque étape de l'algorithme multipôle. Ces mesures ont été réalisées sur CPU et sur GPU à l'aide de la sphère d'un million de points. Pour chaque niveau, nous n'avons pas fait de découpage par blocs, pour que les mesures ne soient pas influencées par cette opération.

CPU Regardons en premier lieu la répartition du temps entre les tâches pour le CPU, sur la figure 4.3. Le résultat est sans appel : la translation se démarque nettement avec à elle seule 80% du temps de l'algorithme. En deuxième position viennent les interactions proches. Notons que si nous avions une profondeur de l'octree plus faible, le poids des interactions proches serait plus élevé.

	Agrégation		Translation		Décalage		Alpert		Fourier		Proche	
Niveau	Temps	Ratio	Temps	Ratio	Temps	Ratio	Temps	Ratio	Temps	Ratio	Temps	Ratio
0	0.0000	0.00	1.3067	82.16	0.0000	0.00	0.2600	16.35	0.0237	1.49	0.0000	0.00
1	0.0000	0.00	1.8377	83.59	0.0779	3.54	0.2502	11.38	0.0328	1.49	0.0000	0.00
2	0.0000	0.00	2.8985	88.00	0.1186	3.60	0.2201	6.68	0.0567	1.72	0.0000	0.00
3	0.0000	0.00	4.4070	89.06	0.1773	3.58	0.2751	5.56	0.0892	1.80	0.0000	0.00
4	0.0000	0.00	6.4241	88.20	0.2703	3.71	0.4293	5.89	0.1600	2.20	0.0000	0.00
5	0.0000	0.00	13.4853	88.97	0.5954	3.93	0.7087	4.68	0.3682	2.43	0.0000	0.00
6	0.4269	1.57	18.9300	69.74	0.9655	3.56	0.0000	0.00	0.0000	0.00	6.8209	25.13
Octree	0.4269	0.69	49.2893	79.99	2.2049	3.58	2.1433	3.48	0.7306	1.19	6.8209	11.07

TABLE 4.3 – Répartition des temps pour chacune des tâches (secondes et pourcentages) sur CPU

	Agrégation		Translation		Décalage		Alpert		Fourier		Proche	
Niveau	Temps	Ratio	Temps	Ratio	Temps	Ratio	Temps	Ratio	Temps	Ratio	Temps	Ratio
0	0.0000	0.00	0.0057	7.09	0.0000	0.00	0.0131	16.46	0.0610	76.45	0.0000	0.00
1	0.0000	0.00	0.0081	4.24	0.0009	0.45	0.0203	10.56	0.1626	84.74	0.0000	0.00
2	0.0000	0.00	0.0127	2.25	0.0014	0.24	0.0445	7.87	0.5065	89.64	0.0000	0.00
3	0.0000	0.00	0.0199	1.42	0.0020	0.14	0.1296	9.25	1.2493	89.19	0.0000	0.00
4	0.0000	0.00	0.0280	0.71	0.0030	0.08	0.4602	11.75	3.4264	87.46	0.0000	0.00
5	0.0000	0.00	0.0581	0.48	0.0065	0.05	1.6261	13.51	10.3484	85.96	0.0000	0.00
6	2.5493	4.70	0.1063	0.20	0.0116	0.02	0.0000	0.00	0.0000	0.00	51.5228	95.08
Octree	2.5493	3.52	0.2388	0.33	0.0253	0.03	2.2938	3.17	15.7542	21.76	51.5228	71.18

TABLE 4.4 – Répartition des temps pour chacune des tâches (secondes et pourcentages) sur GPU

GPU Le tableau 4.4 présente les résultats vus précédemment mais pour le GPU. Ces résultats n'ont aucun lien avec les précédents. Cette fois-ci, c'est la méthode de Fourier qui est en tête pour les premiers niveaux, avec entre 75% et 90% du temps. Pour le dernier niveau, les interactions proches totalisent quasiment l'ensemble des calculs, ce qui représente plus de 50% de l'algorithme global. Ce résultat est dû aux efficacités moindres de ces deux tâches sur GPU.

Conclusion Les poids sont très différents selon le type d'unité de calcul qui exécute la tâche. L'utilisation du GPU semble montrer un grand intérêt car les translations largement majoritaires sur CPU représentent un temps très réduit sur GPU. À l'inverse, les interactions proches et les transformées de Fourier qui prédominent sur GPU représentent peu sur CPU. Cette situation nous est favorable car CPU et GPU sont alors complémentaires.

4.3.2 Influence de la taille des blocs

Comme nous l'avons vu à la section 4.2, la taille des blocs a un impact important sur la performance de nos noyaux, d'autant plus sur les GPU. Afin d'évaluer l'impact global sur la méthode multipôle, nous allons mesurer les temps pris par chaque tâche pour quatre tailles de blocs différentes.

Pour ce faire, nous avons utilisé la sphère comportant un million de points. La première taille de blocs, dite petite est définie de telle sorte que les blocs soient les plus petits possible pour le cas étudié, et donc qu'il y ait le plus de blocs à chaque niveau (tout en veillant à ne pas utiliser plus de mémoire que disponible, voir la section 4.3.3). La taille de blocs dite grande impose un seul bloc à chaque niveau. Enfin, pour la taille moyenne, nous avons pris des blocs assez grands mais pas trop afin qu'il y en ait assez (pour le parallélisme). Le tableau 4.5 montre en détails les caractéristiques des blocs pour nos quatre tests. Le bloc grand a été construit pour qu'il y ait quasiment un seul bloc par niveau. Le bloc désigné comme petit, a été construit en prenant des petites tailles, volontairement inférieures à ce que nous avons recommandé dans notre étude sur les noyaux. Le bloc moyen suit à peu près les recommandations que nous avons faites. Enfin, le bloc intermédiaire essaie de suivre les recommandations tout en imposant un minimum de 30 blocs par niveau. Ces tests ont été réalisés sur la machine Averell et sont synthétisés par le tableau 4.6.

La première remarque que nous pouvons émettre, est en rapport avec la section 4.3.1 précédente, elle concerne le poids des différentes tâches. En effet, en réduisant la taille des blocs, la multiplication d'Alpert devient bien moins efficace, à la fois sur GPU et sur CPU, et prend une part importante

	Petit		Intermédiaire		Moyen		Grand	
Niveau	Boîtes	Directions	Boîtes	Directions	Boîtes	Directions	Boîtes	Directions
0	56	240	56	960	56	960	56	27840
1	34	128	272	260	272	896	272	8064
2	145	80	580	160	580	480	1160	2880
3	257	48	590	264	590	352	4720	1056
4	515	32	515	416	1024	416	18270	416
5	514	24	1964	240	2051	240	69742	240
6	514	16	2049	112	7097	112	241658	112

TABLE 4.5 – Caractéristiques du découpage en blocs pour 4 tailles différentes pour la sphère à 1 million de points

dans l'algorithme total, ce qui change la répartition que nous avions trouvée. Dans ce cas, il pourrait être intéressant de bloquer la remontée de l'arbre à un certain niveau, et ainsi privilégier des translations, voir la section 2.1.1.6.

Le bloc moyen est quasiment aussi performant que le bloc grand, à l'exception de la méthode d'Alpert qui, rappelons le, a besoin d'un maximum de θ. Sans surprise, le bloc petit subit une grande perte d'efficacité notamment pour la méthode d'Alpert à la fois sur CPU et GPU. Cette perte est également visible sur GPU pour les transformées de Fourier et l'agrégation.

Le découpage en blocs doit donc être fait avec précaution afin de garder un minimum d'efficacité.

4.3.3 Surcoût du partage

Le point-clé de notre parallélisation est l'ajout de l'étape de partage. Celle-ci, qui permet à chaque bloc d'être indépendant au niveau des données, ne fait pas partie de l'algorithme multipôle séquentiel, nous l'avons ajoutée pour l'algorithme parallèle. Les copies des boîtes externes au bloc engendrent des surcoûts en mémoire et en temps.

Afin d'obtenir ces deux mesures, nous avons repris notre sphère d'un million de points sur la machine Averell, voir le tableau 4.7.

Impact sur la mémoire Le pourcentage de voisins éloignés externes diminue quand le nombre de boîtes dans le bloc augmente. C'est tout à fait logique, car aux voisins absents près, le nombre de boîtes d'un bloc est proportionnel à son volume, alors que le nombre de voisins éloignés externes est proportionnel à sa surface. Si notre priorité est l'occupation mémoire, nous pourrons alors choisir des blocs de grande taille pour diminuer l'impact du partage.

Impact sur le temps de calcul Comme nous pouvons le voir sur le tableau 4.8, le temps de calcul représenté par l'étape de partage est négligeable, d'autant plus, qu'il permet d'optimiser l'étape de translation et donc réduire le temps global nécessaire.

4.3.4 Conclusion

D'après les tests préliminaires, les GPU vont avoir leur rôle à jouer dans l'algorithme multipôle. Notre choix de découpage en tâches avec des étapes de partage semble judicieux, même s'il est difficile de choisir des blocs de taille optimale.

4.4 Exécution sur un nœud hétérogène

Maintenant que nous connaissons le comportement de notre algorithme multipôle en séquentiel, nous pouvons nous intéresser à l'exécution des tâches en parallèle, sur plusieurs CPU et plusieurs GPU.

Niveau	Tâche	Temps CPU				Temps GPU			
		Petit	Inter	Moyen	Grand	Petit	Inter	Moyen	Grand
0	Translation	1.1130	1.3367	1.3367	1.3412	0.0085	0.0111	0.0066	0.0057
	Alpert	7.1631	3.6321	3.6320	0.2704	24.5467	9.2386	2.9762	0.0134
	Fourier	0.0214	0.0479	0.0490	0.0304	0.5791	0.2039	0.2538	0.0730
1	Translation	1.5761	1.8853	1.8844	1.8867	0.0258	0.0095	0.0084	0.0082
	Décalage	0.0664	0.0873	0.0798	0.0798	0.0228	0.0036	0.0036	0.0008
	Alpert	14.8468	3.9973	2.2071	0.2606	40.3201	11.4650	1.1337	0.0208
	Fourier	0.0324	0.0841	0.0568	0.0393	0.8703	0.4558	0.2666	0.1946
2	Translation	2.4792	2.9750	2.9751	2.9761	0.0282	0.0151	0.0134	0.0129
	Décalage	0.1014	0.1365	0.1235	0.1215	0.0123	0.0059	0.0018	0.0014
	Alpert	10.0026	2.1639	0.9544	0.2279	29.4988	4.0100	1.0184	0.0463
	Fourier	0.0511	0.1314	0.0966	0.0686	2.9606	1.3397	0.8827	0.5625
3	Translation	3.7774	4.5236	4.5255	4.5247	0.0583	0.0204	0.0204	0.0201
	Décalage	0.1515	0.1907	0.1846	0.1817	0.0164	0.0121	0.0030	0.0020
	Alpert	3.1459	0.5963	0.4312	0.2822	36.8131	0.7638	0.7023	0.1343
	Fourier	0.0737	0.1543	0.1512	0.1102	8.2803	1.9091	2.2246	1.5765
4	Translation	5.5050	6.5949	6.5959	6.5954	0.0659	0.0292	0.0294	0.0286
	Décalage	0.2328	0.2813	0.2782	0.2769	0.0195	0.0094	0.0044	0.0029
	Alpert	3.5643	0.6945	0.6425	0.4473	55.7285	0.7892	0.6728	0.4649
	Fourier	0.1058	0.2419	0.2085	0.1638	10.4819	3.4352	3.3037	3.3081
5	Translation	11.5751	13.8523	13.8488	13.8447	0.1870	0.0589	0.0607	0.0582
	Décalage	0.5132	0.6186	0.6137	0.6100	0.0579	0.0104	0.0105	0.0065
	Alpert	6.6666	1.2353	1.0628	0.7316	114.8399	1.9516	2.0902	1.6653
	Fourier	0.3275	0.4901	0.5153	0.4505	56.1238	10.7774	12.9772	12.9159
6	Agrégation	0.8550	0.7493	0.6302	0.4285	17.3263	2.6999	2.7053	2.5453
	Translation	16.2614	19.4365	19.4413	19.4340	0.3808	0.1106	0.1080	0.1061
	Décalage	0.8454	0.9916	0.9954	0.9888	0.1365	0.0265	0.0154	0.0114
	Proche	5.9081	6.9513	7.0942	7.0958	51.5053	51.5133	51.5649	51.5454
Octree	Agrégation	0.8550	0.7493	0.6302	0.4285	17.3263	2.6999	2.7053	2.5453
	Translation	42.2871	50.6044	50.6076	50.6029	0.7547	0.2547	0.2469	0.2399
	Décalage	1.9107	2.3061	2.2752	2.2587	0.2654	0.0679	0.0387	0.0250
	Alpert	45.3893	12.3195	9.8290	2.2199	301.7470	28.2182	8.5937	2.3449
	Fourier	0.6120	1.1498	1.0837	0.8628	79.2960	18.1211	19.9087	18.6305
	Proche	5.9081	6.9513	7.0942	7.0958	51.5053	51.5133	51.5649	51.5454
	Total	97.1589	74.2171	71.3975	63.3273	447.4763	101.6138	80.3512	72.6211

TABLE 4.6 – Temps (en secondes) des tâches suivant la taille des blocs pour la sphère à 1 million de points

Niveau	Blocs	Boîtes	Boîtes externes	Surcoût mémoire (%)
0	1	56	0	0.00
1	1	272	0	0.00
2	2	580	128	22.07
3	8	590	198	33.56
4	16	1024	419	40.92
5	32	2051	718	35.01
6	40	7097	1113	15.68

TABLE 4.7 – Impact de l'introduction de l'étape de partage sur la mémoire

Niveau	Blocs	Temps CPU			Temps GPU		
		Total	Partage	Surcoût (%)	Total	Partage	Surcoût (%)
2	2	4.1347	0.0041	0.10	1.8089	0.0083	0.46
3	8	5.2705	0.0131	0.25	2.4091	0.0258	1.07
4	16	7.7188	0.0089	0.11	4.0955	0.0346	0.85
5	32	15.9630	0.0186	0.12	12.8676	0.1186	0.92
6	40	28.1815	0.0205	0.07	54.6220	0.2284	0.42
Octree	-	71.3975	0.0652	0.09	80.3512	0.4158	0.52

TABLE 4.8 – Impact de l'introduction de l'étape de partage sur le temps de calcul, pour la sphère d'un million de points

Découpage	Type de blocs	Nombre de blocs						
		Niv 0	Niv 1	Niv 2	Niv 3	Niv 4	Niv 5	Niv 6
Petit	Bloc boîtes	1	8	8	16	32	124	423
	Bloc directions	116	63	36	22	13	10	7
	Total	116	504	288	352	416	1240	2961
Intermédiaire	Bloc boîtes	1	1	2	8	32	32	106
	Bloc directions	29	31	18	4	1	1	1
	Total	29	31	36	32	32	32	106
Moyen	Bloc boîtes	1	1	2	8	16	32	40
	Bloc directions	29	9	6	3	1	1	1
	Total	29	9	12	24	16	32	40
Grand	Bloc boîtes	1	1	1	1	1	1	1
	Bloc directions	1	1	1	1	1	1	1
	Total	1	1	1	1	1	1	1

TABLE 4.9 – Nombre de blocs de directions et de blocs de boîtes suivant le découpage choisi, pour la sphère à un million de points

Nous devons étudier plusieurs aspects. Tout d'abord, nous devons confirmer l'impact de la taille des blocs. Ensuite, nous pourrons procéder à des tests de scalabilité, et mesurer les bénéfices de l'utilisation d'un ordonnanceur dynamique.

4.4.1 Influence de la taille des blocs

Comme nous l'avons vu à la section 4.2, augmenter la taille des blocs permet en général d'augmenter les performances du calcul. Le problème est qu'en augmentant la taille des blocs, nous diminuons le nombre de blocs par niveau, et nous perdons alors du parallélisme. De plus, avoir une granularité trop importante rend l'ordonnancement plus critique : des problèmes de bulles – attente d'une tâche – peuvent apparaitre, et chaque décision d'ordonnancement a des répercussions plus critiques.

Pour mieux comprendre l'impact de ce phénomène, nous avons repris nos quatre découpages du tableau 4.5 avec notre sphère d'un million de points, et pour chacun, nous avons compté le nombre de blocs par niveau, voir le tableau 4.9.

Maintenant que nous connaissons toutes les données, nous pouvons passer aux tests de l'exécution sur notre machine Averell avec l'ensemble des cœurs CPU et GPU. Le tableau 4.10 est une retranscription des mesures des temps pour chaque découpage. Le résultat important est que le découpage intermédiaire est le plus rapide car même si certaines de ses tâches sont plus longues, il fournit davantage de parallélisme que les découpage moyen et grand. Quant au découpage petit, il fournit plus de blocs que nécessaires pour un bon parallélisme, au détriment des performances des tâches.

Découpage	Temps (s)
Petit	4.32
Intermédiaire	2.60
Moyen	2.84
Grand	6.67

TABLE 4.10 – Temps pour une exécution parallèle suivant le découpage

Nombre	0 GPU			1 GPU			2 GPU		
de CPU	Temps	Gain	Efficacité	Temps	Gain	Efficacité	Temps	Gain	Efficacité
1	458.93	1.00	100.00	513.65	0.89	1.37	-	-	-
2	232.78	1.97	98.57	65.80	6.97	10.51	297.47	1.54	1.18
4	118.11	3.89	97.14	32.48	14.13	20.66	37.45	12.25	9.23
6	79.82	5.75	95.83	26.85	17.09	24.29	25.25	18.18	13.49
8	60.61	7.57	94.65	22.61	20.30	28.04	20.74	22.13	16.18
10	49.16	9.34	93.36	19.94	23.01	30.94	17.78	25.82	18.61
12	41.74	10.99	91.62	18.64	24.61	32.23	16.74	27.42	19.48
14	36.49	12.58	89.84	17.33	26.48	33.79	15.56	29.49	20.66
16	32.26	14.23	88.91	16.87	27.20	33.84	14.57	31.49	21.76

TABLE 4.11 – Efficacité, en pourcentage, de la résolution de la sphère à 5 millions de points sur machine hétérogène

Nous ne proposons pas de méthode systématique qui permettrait de trouver la taille idéale des blocs. Elle dépend de la configuration de la machine. Il s'agit de trouver un équilibre entre optimalité des noyaux et parallélisme.

4.4.2 Scalabilité sur architecture hétérogène

Afin de juger de l'efficacité de notre parallélisation, nous allons étudier la scalabilité de notre programme. Pour cela, nous avons pris la sphère à 5 millions de points, résolue sur la machine Averell. Pour le découpage en blocs, nous avons respecté les critères vus précédemment, en imposant en général un minimum de blocs adapté au nombre de CPU et GPU. Les résultats des exécutions sont présentés sur le tableau 4.11.

Regardons d'abord l'évolution du temps simplement en faisant varier le nombre de CPU, sans GPU. Nous obtenons une efficacité de 88.9% pour 16 processeurs, ce qui indique que notre parallélisation a un coût assez faible. Cela confirme le fait que le temps de l'étape de partage entre les blocs est négligeable.

Étudions maintenant le cas où il y a des GPU. Nous rappelons que StarPU se réserve un CPU par GPU à contrôler. Par conséquent, quand nous utilisons un GPU et un CPU, cela signifie qu'il y a un GPU qui réalise les tâches de la méthode multipôle, mais aucun CPU. Ce cas-là obtient alors naturellement un gain très mauvais à cause des interactions proches très coûteuses sur GPU. Pour les cas où nous avons au moins un CPU qui travaille, nous obtenons des gains assez importants. Comme il est difficile de juger de l'efficacité de l'exécution simplement en regardant les gains, nous avons calculé l'efficacité, en nous basant sur le temps pour un seul CPU, pondéré par le quotient de la puissance théorique (en flops) du cas avec un CPU sur celle du cas concerné :

$$\text{Efficacité}_{cas} = \frac{\text{Temps}_{1CPU}\text{Flops}_{1CPU}}{\text{Temps}_{cas}\text{Flops}_{cas}}.$$

Ainsi, si nous trouvons que le doublement du gain en ajoutant 1 GPU aux 16 CPU est important, nous relativisons en regardant l'efficacité qui n'est même pas de 30%. Une autre mesure qui aurait du sens serait de mesurer l'efficacité énergétique (nombre d'opérations flottantes par Watt). Tout dépend ce que nous recherchons.

Nombre	Temps (s)	
de CPU	1 GPU	2 GPU
2	76.78956	
4	30.52096	42.31946
6	19.84575	25.87443
8	17.17306	22.17561
10	16.07873	18.05570
12	16.87641	18.64803
14	16.26643	18.57163
16	16.16069	18.36083

TABLE 4.12 – Temps (en secondes) de la résolution de la sphère à 5 millions de points sur machine hétérogène, avec ordonnancement statique

En présence de GPU, l'augmentation du nombre de CPU permet d'augmenter l'efficacité. Cela provient du fait que nous avons un CPU réservé par GPU. Nous en déduisons qu'il y a trop de GPU par rapport au nombre de CPU, pour exploiter au mieux les ressources. Malgré tout, le deuxième GPU apporte un gain.

Ce test permet de valider nos choix de parallélisation, et montre l'intérêt réel de l'utilisation de GPU dans la méthode multipôle.

4.4.3 Intérêt de l'ordonnanceur

Nous venons de défendre notre approche, mais n'avons pas encore justifié l'utilisation d'un ordonnancement dynamique. Nous avons vu à la section 4.2 que les GPU étaient intéressants principalement pour les translations. Nous pouvons alors nous demander si l'approche consistant à exécuter toutes les translations sur GPU, n'aurait pas donné des résultats équivalents ou même meilleurs. Le tableau 4.12 montre les temps d'exécution obtenus, en forçant l'ordonnanceur à réaliser toutes les translations, et seulement les translations, sur GPU. Ces résultats sont à rapprocher de ceux du tableau 4.11.

En présence d'un seul GPU, l'ordonnancement statique est légèrement meilleur que l'ordonnancement dynamique. En revanche, quand nous ajoutons un GPU, il devient moins bon, et même moins bon que la version n'utilisant qu'un seul GPU. Cela s'explique par le fait que les GPU entrent en concurrence pour les accès à la mémoire centrale. De plus, il n'y a pas assez de translations pour satisfaire les deux GPU pendant toute l'exécution, qui n'auront alors plus de tâches à traiter. C'est là où l'ordonnancement dynamique prend tout son sens.

Afin de mieux comprendre ce phénomène, nous avons visualisé la répartition des tâches en présence de 16 CPU (donc 14 qui participent à l'algorithme) et 2 GPU, et l'avons synthétisée dans le tableau 4.13.

Nous remarquons bien que les GPU ne traitent pas seulement des translations, mais aussi des Alpert, Fourier et décalage. De plus, les CPU traitent quelques translations, quand ils sont en attente de tâches disponibles.

4.4.4 Conclusion

Grâce aux différents tests effectués, nous avons validé l'intérêt pour la parallélisation de notre découpage en blocs avec une étape de partage. Nous avons justifié à la fois l'utilisation de GPU et celle d'un ordonnanceur dynamique.

4.5 Exécution en mémoire distribuée

Pour utiliser plusieurs nœuds, notre approche repose sur une distribution statique des sous-arbres conjuguée avec une distribution dynamique des interactions proches de façon à atténuer les

Processeur	Nombre d'exécutions					
	Agrégation	Alpert	Décalage	Fourier	Proche	Translation
CPU 0	8	245	21	75	3	1
CPU 1	10	184	28	117	6	1
CPU 2	6	201	39	99	4	1
CPU 3	4	245	28	104	6	1
CPU 4	2	225	20	64	4	2
CPU 5	5	213	17	57	4	2
CPU 6	4	212	12	78	5	1
CPU 7	5	195	10	61	5	1
CPU 8	3	167	9	60	8	2
CPU 9	1	208	13	46	6	1
CPU 10	2	170	8	35	4	2
CPU 11	15	204	48	146	6	1
CPU 12	2	194	10	41	4	2
CPU 13	3	171	11	39	5	2
GPU 0	0	217	64	61	0	92
GPU 1	0	181	0	43	0	286

TABLE 4.13 – Répartition des tâches sur chaque unité de calcul

Sous-arbre	1	2	3	4
Nombre de feuilles	657953	657959	657876	657882

TABLE 4.14 – Nombre de feuilles des 4 sous-arbres de la sphère à 5 millions de points

déséquilibres. Nous allons, dans un premier temps, valider ces choix, puis, nous procéderons à un test de scalabilité.

4.5.1 Équilibrage de charge

Nous devons vérifier que nos distributions sont bien adaptées. Puis, nous nous assurerons de l'intérêt de la distribution dynamique des interactions proches.

4.5.1.1 Distribution

Notre distribution des sous-arbres se fait en attribuant un poids à chaque sous-arbre. Ce poids correspond au nombre de feuilles qu'il possèdent. Afin de vérifier que nos distributions sont correctes, nous allons réaliser deux tests sur la sphère à 5 millions de points, pour laquelle nous aurons coupé le premier niveau en 4 blocs de boîtes. Vue la nature régulière et symétrique de la sphère, nous devrions avoir 4 sous-arbres de poids similaires. Pour le premier test, nous laisserons notre algorithme choisir la distribution statique, et pour le deuxième nous forcerons un déséquilibre.

La première information, que nous vérifions grâce au tableau 4.14, est que les poids sont bien similaires entre les 4 sous-arbres.

Nous avons réalisé ces deux tests en utilisant 2 nœuds de Mirage. Le tableau 4.15 présente le nombre d'interactions proches qui ont été distribuées pour chaque nœud et pour les deux tests. Pour la distribution automatique, nous avons bien 2 blocs par nœud, et pour la distribution déséquilibrée, nous avons imposé 1 seul bloc pour le premier nœud. La distribution dynamique est bien révélatrice du déséquilibre : le nœud qui a le moins de boîtes réalise 9 fois plus d'interactions proches en compensation.

La distribution dynamique tient bien compte des déséquilibres de la distribution statique.

Distribution	Noeud	Nombre de blocs	Interactions proches
Automatique	0	2	36
	1	2	36
Déséquilibrée	0	1	65
	1	3	7

TABLE 4.15 – Nombre d'interactions proches calculées par chaque nœud, en fonction de la distribution des boîtes adoptée

Distribution des boîtes	Distribution des interactions proches	Temps (s)
Équilibrée	Statique	16.45
	Dynamique	16.45
Déséquilibrée	Statique	24.55
	Dynamique	21.02

TABLE 4.16 – Temps d'exécution (en secondes) des distributions statique et dynamique des interactions proches dans des cas équilibré et déséquilibré

4.5.1.2 Recouvrement des déséquilibres

Afin de montrer l'intérêt de la distribution dynamique des interactions proches, nous l'avons confrontée à une distribution statique dans les cas équilibré et déséquilibré. Le tableau 4.16 montre les différents temps obtenus.

Le résultat est sans appel, la distribution dynamique souffre moins du déséquilibre. Quand nous regardons les temps en détails, le calcul de 3 sous-arbres seuls prend environ 21 secondes et le calcul complet des interactions proches 6.5 secondes. Comme nous l'avons vu dans le tableau 4.15, le nœud en surcharge de travail traite également quelques interactions proches, et met également 21 secondes ! Cela provient du fait que celles-ci servent également à masquer les attentes intra-nœuds. Sur cet exemple, la distribution dynamique a permis de minimiser le déséquilibre au maximum.

4.5.2 Test de scalabilité

Nous allons maintenant soumettre notre algorithme au test de scalabilité. Malheureusement, la plus grosse machine dont nous disposons ne dispose que de 8 nœuds, ce qui est peu si nous comparons avec les supercalculateurs actuels dotés de dizaines ou même centaines de milliers de nœuds. Cependant, ces tests donneront au moins un *a priori* et une validation sur des petites configurations. Nous avons utilisé la sphère de 8 millions, pour laquelle nous avons pris le niveau racine comme niveau de partage. Aussi, nous n'avons pas changé les tailles de blocs en fonction du nombre de nœuds. Nous cherchons à démontrer ici l'efficacité de notre algorithme et non les effets du choix du niveau de partage, une étude plus poussée faisant varier le niveau de partage montrerait réellement l'impact de la distribution des boîtes entre les nœuds sur l'algorithme multipôle.

Le tableau 4.17 synthétise les mesures de temps effectuées suivant le nombre de nœuds. Jusqu'au maximum de nos tests, la scalabilité est quasiment parfaite.

Nombre de nœuds	1	2	4	8
Temps (s)	97.78	48.91	24.46	12.25
Efficacité (%)		99.96	99.94	99.78

TABLE 4.17 – Temps d'exécution (en secondes) et efficacité (en pourcentage) en fonction du nombre de nœuds pour la sphère à 8 millions de points, sur la machine Mirage

4.6 Conclusion

Nos tests nous ont d'abord permis de montrer que les GPU permettent d'accélérer certaines tâches de la méthode multipôle, en particulier les translations et les décalages, avec un gain qui peut atteindre la centaine. L'utilisation de GPU sur une machine hétérogène à 16 cœurs, nous a permis d'accélérer la résolution par 2 avec 1 GPU et de gagner 10% supplémentaires avec 2. De plus, l'utilisation d'un ordonnanceur dynamique avec l'appui de notre algorithme permet un léger gain par rapport à un ordonnancement statique.

Concernant la version distribuée, nous avons établi l'utilité de la distribution dynamique des interactions proches à propos du recouvrement des déséquilibres entre les nœuds, et des communications. Notre étude de scalabilité obtient un score quasiment parfait, bien qu'effectuée avec peu de nœuds, est très encourageante pour des tests plus importants.

Conclusion

Pour résoudre les équations de Maxwell et calculer le comportement électromagnétique d'un objet soumis à une onde incidente, la méthode multipôle est un atout indéniable réduisant la complexité du calcul, en se basant sur une décomposition spatiale à l'aide d'un octree formé de boîtes. Cependant, ses besoins en calcul et en mémoire restent en constante augmentation. Pour simuler plus finement des objets complexes, il est nécessaire de traiter des cas comportant davantage de points. S'ajoute à cela une volonté d'obtenir le résultat plus rapidement. Parallèlement, les machines de calcul continuent sans cesse d'évoluer, avec leur nombre de processeurs qui augmente considérablement, mais également avec la généralisation des machines hétérogènes qui intègrent des accélérateurs, type GPU ou autres.

De nombreux travaux ont été menés dans le but d'adapter la méthode multipôle à l'augmentation du nombre de processeurs. Ils ont permis de faire évoluer l'algorithme pour l'adapter aux architectures hybrides, notamment avec la distribution par boîtes et directions adaptée au niveau. Cependant, les résultats montrent de bonnes efficacités qui semblent limitées à des machines disposant d'un nombre de cœurs borné au millier. Il est donc incontournable d'utiliser les accélérateurs qui se multiplient sur les machines d'aujourd'hui et de demain. Dans ce domaine, les recherches sont principalement limitées aux autres applications, ne concernant pas l'électromagnétisme, de la méthode multipôle. Les résultats sont impressionnants, cependant l'algorithme diffère suffisamment pour que ces travaux ne soient pas applicables directement pour l'électromagnétisme.

Avec l'évolution des machines, les outils pour les exploiter ont également évolué. Nous trouvons tout d'abord des bibliothèques de calcul qui donnent accès à des opérations fortement optimisées sur CPU et GPU. Nous pouvons citer les BLAS pour CPU, et CUBLAS pour GPU, qui fournissent une grande quantité d'opérations d'algèbre linéaire, ou encore la FFTW pour CPU ou CUFFT pour GPU, qui permettent d'effectuer des transformées de Fourier, indispensables pour la méthode multipôle. Ces bibliothèques permettent d'avoir certaines tâches de notre algorithme performantes, mais elles ne vont pas permettre d'avoir un algorithme performant sur architecture hétérogène. D'autres bibliothèques s'intéressent à la génération de code pour les accélérateurs quand les opérations ne peuvent être décomposées en BLAS. Mais la réelle difficulté dans l'utilisation des machines hétérogènes, du moins pour la méthode multipôle, reste l'adaptation de l'algorithme pour profiter au mieux des ressources. Dans cette optique, il existe des ordonnanceurs dynamiques de tâches, tel StarPU, qui répartissent les différents calculs entre les différentes ressources.

C'est pour mieux faire correspondre l'offre proposée par les nouvelles architectures, et principalement par le développement des accélérateurs, avec les demandes de résolutions en quête de davantage de vitesse, que nous avons réalisé ces travaux. Ceux-ci s'inscrivent dans une démarche de recherche d'efficacité sur les architectures hétérogènes, en insistant sur une volonté de pérennité, et ce pour chaque étape de notre démarche.

Contribution

L'originalité de nos travaux réside dans l'utilisation d'un ordonnanceur dynamique pour répartir les tâches sur des machines hétérogènes disposant de CPU et GPU. Pour y parvenir, la première étape a été la recherche de noyaux de calcul efficaces pour les différentes tâches sur CPU et GPU, en nous efforçant d'employer des techniques optimisées. Pour appréhender la méthode multipôle sur machine hétérogène, nous avons étudié le comportement de chaque tâche qui la compose. À partir de mesures, nous avons pu établir des critères sur le choix des tailles pour adapter les noyaux aux ressources de calcul et ainsi obtenir les meilleures performances. Ces tests ont également permis de motiver l'utilité de GPU dans l'algorithme de la méthode multipôle car ils ont montré un gain de performance important par rapport au CPU, principalement pour les tâches de translation et de décalage.

Afin d'intégrer l'ordonnancement dynamique à la méthode multipôle, nous avons proposé une nouvelle stratégie fondée sur la construction de tâches agissant sur des blocs de données. Nos tâches sont issues des étapes de la méthode multipôle que nous avons décomposées pour qu'elles effectuent un seul type de calcul. Nous avons groupé les données en blocs de boîtes et de directions. Nous avons pris la décision de forcer chaque tâche de calcul à utiliser au maximum deux blocs à la fois, pour limiter les transferts mémoire en particulier sur GPU. Pour cela, nous avons extrait les échanges de données des tâches de calcul en ajoutant une tâche dite de partage, et qui va rendre les blocs indépendants notamment lors de la tâche de translation. Les blocs de données ont également dû être modifiés pour intégrer tous les voisins éloignés. Nos tests ont montré que cette tâche supplémentaire apporte un surcoût en temps de calcul qui est négligeable. Les exécutions en mémoire partagée nous ont permis de confirmer la validité de notre approche, en exposant les bénéfices de l'intégration d'un ordonnanceur dynamique. L'avantage le plus évident est un usage accru des ressources de la machine car l'ordonnanceur dynamique cherche à éviter de laisser une unité de calcul sans travail. Nous avons vu qu'il pourra lui confier une tâche qui n'est pas forcément adaptée, mais qui permettra de ne pas gaspiller les ressources par des attentes sans travail. Ce type d'optimisation n'est pas possible avec un ordonnancement statique car il doit être adapté à la machine et également à l'objet étudié. Un autre avantage est l'évolutivité du code, sans effort important de programmation, en cas d'exécution avec des types d'architectures autres que CPU/GPU.

Nous avons complété notre version en mémoire partagée de l'algorithme multipôle, afin de prendre en charge la mémoire distribuée pour utiliser plusieurs nœuds de calcul. Notre distribution des calculs pour les interactions lointaines est classique : nous attribuons à chaque nœud, statiquement, des sous-arbres à traiter, définis à partir d'un niveau racine. Comme en mémoire partagée avec l'emploi de tâches de partage, nous avons modifié l'algorithme pour bien séparer les tâches de calcul des tâches de communication. Pour cela, nous avons introduit des blocs de boîtes supplémentaires, un à chaque niveau et pour chaque nœud en communication, sur lesquels aucun calcul n'aura lieu, mais seulement des communications. Avec cette distribution des boîtes, et en choisissant la numérotation hiérarchique adaptée, le nombre de nœuds communiquant avec un nœud est limité, hors échanges au sommet. Pour ces derniers, nous avons créé un niveau fictif qui permet de séparer les calculs des communications, c'est-à-dire les interpolations des échanges de blocs de directions.

Le problème de notre distribution statique est qu'elle peut être déséquilibrée, ce qui provoque une perte d'efficacité à deux moments de l'algorithme. Le premier se situe à proximité de la fin de l'étape de montée qui se termine obligatoirement par des communications pour que l'étape de descente puisse succéder. Si un nœud a moins de calculs à réaliser, il finira sa montée avant les autres mais sera bloqué, en attente de réception des données des autres nœuds. L'autre situation qui peut poser problème est quand un nœud a terminé ses calculs mais pas les autres. Pour l'attente à la fin de la remontée, nous avons proposé deux solutions. La première consiste à ajouter des priorités dans l'ordonnancement pour donner l'avantage aux tâches conduisant aux échanges du sommet. Cependant, cela repousse une partie du problème à la fin de la descente. C'est pourquoi, nous avons mis en place une deuxième solution qui est la la distribution des interactions proches dynamique. Quand un nœud se retrouve dans l'une des deux situations décrites précédemment, il va alors demander des interactions proches à traiter. Un nœud est désigné comme responsable pour le traitement des demandes. Cela peut être le nœud qui a le moins de calcul, même si le coût pour envoyer un intervalle est faible. Ainsi comme l'ont confirmé nos tests, cette distribution dynamique

sert de tampon pour masquer les déséquilibres de la distribution statique des interactions lointaines. Pour évaluer globalement notre parallélisation en mémoire distribuée, nous avons réalisé des tests préliminaires qui ont montré une scalabilité quasiment parfaite pour 8 nœuds.

Perspectives

La méthode multipôle est une méthode complexe qui ouvre à de vastes travaux de recherche dans tous les domaines. Même si nous nous cantonnons à l'optimisation de l'exécution sur machine hétérogène, le travail possible restant est vaste. Nous allons référencer ici les points qui nous semblent primordiaux pour aller plus loin, en continuité de notre démarche.

Notre étude des noyaux de calcul nous a permis d'établir des règles de construction des blocs, mais elle pourrait également apporter davantage. En connaissant la courbe précisément, nous pourrions prévoir le temps de chaque tâche. Cela conduirait à de nouvelles études à propos des choix du niveau de partage et de la profondeur. Pour nos tests, nous avons fait ces choix nous-mêmes, en comparant les temps d'exécution. Néanmoins, cette opération s'avère être fastidieuse et très consommatrice en temps, variable que nous cherchons à minimiser. Pour qu'un algorithme soit performant, il doit intégrer ce choix automatique. Pour donner une piste, nous allons énoncer une première façon de procéder. À partir des prévisions de temps pour chaque tâche, nous pouvons construire un système linéaire représentant le temps d'exécution, et pour lequel nous chercherons à minimiser le temps. Nous aurons ainsi, une évaluation du temps d'exécution de l'algorithme total. Nous pourrons répéter cette opération en prenant différentes valeurs pour les choix des niveaux. Une simple comparaison des temps trouvés, nous permettrait de connaître les meilleurs choix de paramètres.

Nous avons choisi de construire un octree ayant une profondeur constante, mais quand les objets à étudier ne sont pas réguliers, il peut être très avantageux de considérer l'algorithme dit adaptatif, ciblant la profondeur d'une branche de l'arbre à la géométrie de l'objet. La difficulté avec cette version réside dans la distribution des boîtes entre les nœuds. En effet, nous ne pouvons plus estimer le poids de chaque sous-arbre en comptant leur nombre de feuilles. Toutefois, en nous servant de l'estimation du temps des tâches et en procédant comme nous l'avons évoqué précédemment avec la piste pour le choix des niveaux, nous obtenons un temps estimé d'exécution, et donc une mesure pour réaliser notre distribution statique correctement. Il y aurait un autre avantage à posséder une estimation du temps minimal d'exécution. Cela nous donnerait un objectif à atteindre, et de ce fait, un critère d'évaluation de notre ordonnancement. Il serait possible d'intégrer dans notre algorithme d'autres améliorations de la méthode multipôle, telle l'utilisation de transformées de Fourier dans la translation.

Nous avons montré la validité de notre démarche en utilisant des machines de petite taille. Il serait important d'analyser le comportement de notre algorithme à plus grande échelle. Il faudrait d'abord vérifier que notre utilisation de l'ordonnanceur dynamique passe bien à l'échelle sur des nœuds comportant davantage de cœurs. Ensuite, il serait nécessaire de soumettre notre couplage de distributions statique et dynamique à davantage de nœuds. Il faudrait sûrement y apporter des modifications et notamment dans la distribution des interactions proches, qui pourrait être confiée à plusieurs nœuds en même temps. Une telle exécution nécessiterait d'employer des cas tests comportant davantage de points, permettant d'effectuer une validation en utilisant des cas tests réels, tels que des avions. Grâce à cela, nous pourrions également nous comparer à d'autres implémentations pour évaluer précisément l'impact de nos travaux.

Dans nos travaux, nous nous sommes intéressés à un seul type d'accélérateur, mais il en existe d'autres. Nous pouvons citer le Xeon Phi fabriqué par Intel, qui est intégré dans le supercalculateur Tianhe-2 qui occupe actuellement la première place du top500. Une étude sur son utilisation conjointement ou non à des GPU pourrait être réalisée. Avec l'évolution constante des machines et les besoins de résolution croissants, les recherches sur la parallélisation de la méthode multipôle continueront, tant que cette méthode restera utilisée.

Bibliographie

[1] M. Abramowitz and I.A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables. National Bureau of Standards Applied Mathematics Series 55. Tenth Printing.* Dover Publications, New-York, 1972.

[2] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers : principles, techniques, and tools*, volume 1009. Pearson/Addison Wesley, 2007.

[3] G. Almási, P. Hargrove, I.G. Tanase, and Y. Zheng. Upc collectives library 2.0. In *Proc. of the Fifth Conference on Partitioned Global Address Space Programming Models*, 2010.

[4] B. K. Alpert and V. Rokhlin. A fast algorithm for the evaluation of Legendre expansions. *SIAM J. Sci. Statist. Comput.*, 12(1) :158–179, 1991.

[5] AMD. User Guide AMD Core Math Library (ACML) 5.2.0, 2012.

[6] J.P. Anderson, S.A. Hoffman, J. Shifman, and R.J. Williams. D825-a multiple-computer system for command & control. In *Proceedings of the December 4-6, 1962, fall joint computer conference*, pages 86–96. ACM, 1962.

[7] C. Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators : a Runtime System's Perspective.* PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2011.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.A. Wacrenier. StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[9] R. Beatson and L. Greengard. A short course on fast multipole methods. In *Wavelets, multilevel methods and elliptic PDEs (Leicester, 1996)*, pages 1–37. Oxford Univ. Press, New York, 1997.

[10] M. Bebendorf. Approximation of boundary element matrices. *Numerische Mathematik*, 86(4) :565–589, 2000.

[11] A. Bendali and L. Vernhet. Résolution par éléments finis de frontière d'un problème de diffraction d'onde comportant une condition aux limites d'impédance généralisée. *C. R. Acad. Sci. Paris*, 321 :791–797, 1995.

[12] LS. Blackford, A. Petitet, R. Pozo, K. Remington, R.C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2) :135–151, 2002.

[13] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk : an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8) :207–216, August 1995.

[14] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the GPU : conjugate gradients and multigrid. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 917–924. ACM, 2003.

[15] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italie, February 2010.

[16] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[17] CAPS. Write once deploy manycores, 2012.

[18] Q. Carayol and F. Collino. Error estimates in the fast multipole method for scattering problems. I. Truncation of the Jacobi-Anger series. *M2AN Math. Model. Numer. Anal.*, 38(2) :371–394, 2004.

[19] Q. Carayol and F. Collino. Error estimates in the fast multipole method for scattering problems. II. Truncation of the Gegenbauer series. *M2AN Math. Model. Numer. Anal.*, 39(1) :183–221, 2005.

[20] Texas Advanced Computing Center. Gotoblas2, 2011.

[21] R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan Kaufmann, 2000.

[22] J.T. Chen and H.K. Hong. Boundary element method. *New World Publication*, 1992.

[23] W. C. Chew, J. M. Jin, E. Michielssen, and J. M. Song. *Fast and Efficient Algorithms in Computational Electromagnetics*. Artech House, 2001.

[24] R. Coifman, V. Rokhlin, and S. Wandzura. The fast multipole method for the wave equation : A pedestrian prescription. *Antennas and Propagation Magazine, IEEE*, 35(3) :7–12, 2002.

[25] D. Colton and R. Kress. *Inverse acoustic and electromagnetic scattering theory*, volume 93. Springer, 1992.

[26] David L. Colton and Rainer Kress. *Integral equation methods in scattering theory*. Pure and Applied Mathematics (New York). John Wiley & Sons Inc., New York, 1983. A Wiley-Interscience Publication.

[27] E. Darve. The fast multipole method I : Error analysis and asymptotic complexity. *SIAM Journal on Numerical Analysis*, 38(1) :98–128, 2000.

[28] E. Darve. The fast multipole method : Numerical implementation. *Journal of Computational Physics*, 160(1) :195–240, 2000.

[29] B.R. de Supinski. Progress on openmp specifications, 2012.

[30] R. Dolbeau, S. Bihan, and F. Bodin. HMPP : A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[31] J. Dongarra. Architecture-aware algorithms and software for peta and exascale computing. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, page 507. IEEE Computer Society, 2011.

[32] U. Drepper. What every programmer should know about memory. *Red Hat, Inc*, 2007.

[33] O. Ergül and L. Gürel. Hierarchical parallelisation strategy for multilevel fast multipole algorithm in computational electromagnetics. *Electronics Letters*, 44(1) :3–5, 2008.

[34] O. Ergül and L. Gürel. A hierarchical partitioning strategy for an efficient parallelization of the multilevel fast multipole algorithm. *IEEE Transactions on Antennas and Propagation*, 57(6) :1740–1750, 2009.

[35] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[36] HSA Foundation. Hsa foundation arm, amd, imagination, mediatek, qualcomm, samsung, ti, 2013.

[37] M. Frigo and S. n G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2) :216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[38] T. Gautier, X. Besseron, and L. Pigeon. KAAPI : A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 15–23. ACM, 2007.

[39] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Network Parallel Computing*. MIT press, 1994.

[40] GPGPU.org. About gpgpu.org, 2013.

[41] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2) :325–348, 1987.

[42] Portland Group. Pgi accelerator compilers, 2013.

[43] N.A. Gumerov and R. Duraiswami. Fast multipole methods on graphics processors. *Journal of Computational Physics*, 227(18) :8290–8313, 2008.

[44] M.J. Harris, G. Coombe, T. Scheuermann, and A. Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 109–118. Eurographics Association, 2002.

[45] J.L. Hennessy and D.A. Patterson. *Computer architecture : a quantitative approach*. Morgan Kaufmann, 2011.

[46] Intel. Intel® Math Kernel Library (Intel® MKL) 11.0, 2012.

[47] R. Jakob-Chien and K. Alpert. Fast spherical filter with uniform resolution. *Journal of Comput. Physics*, 136(2) :580–584, September 1987.

[48] D.S. Jones. Acoustic and electromagnetic waves. *Oxford/New York, Clarendon Press/Oxford University Press, 1986, 764 p.*, 1, 1986.

[49] L.V. Kale and S. Krishnan. *CHARM++ : a portable concurrent object oriented system based on C++*, volume 28. ACM, 1993.

[50] C.H. Koelbel, D.B. Loveman, and R.S. Schreiber. *The high performance Fortran handbook*. MIT press, 1993.

[51] J. Krüger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 908–916. ACM, 2003.

[52] Parallel Programming Laboratory. The Charm++ Parallel Programming System Manual, 2012.

[53] I. Lashuk, A. Chandramowlishwaran, H. Langston, T.A. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Communications of the ACM*, 55(5) :101–109, 2012.

[54] G.E. Moore. Cramming more components onto integrated circuits, electronics 38, 1965.

[55] G.E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11 – 13, 1975.

[56] JC Mouriño, A. Gómez, JM Taboada, L. Landesa, JM Bértolo, F. Obelleiro, and JL Rodríguez. High scalability multipole method. Solving half billion of unknowns. *Computer Science-Research and Development*, 23(3) :169–175, 2009.

[57] F. Mueller et al. A library implementation of posix threads under unix. In *Proceedings of the USENIX Conference*, pages 29–41, 1993.

[58] A. Munshi et al. The OpenCL specification Version 1.2. *Khronos OpenCL Working Group*, November 2012.

[59] R.W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[60] Nvidia. CUDA C Best Practices Guide, 2012.

[61] Nvidia. CUDA C Programming Guide, 2012.

[62] CUDA Nvidia. Compute unified device architecture programming guide, 2007.

[63] J.C. Nédélec. *Acoustic and Electromagnetic Equation. Integral Representation for Harmonic Problems*, volume 144. Springer-Verlag, 2001.

[64] S. Ohnuki and W. C. Chew. Numerical Accuracy of Multipole Expansion for 2-D MLFMA. *IEEE Trans. Antennas Propagat.*, 51(8) :1883–1890, AUGUST 2003.

[65] OpenACC. Openacc 2.0a spec, 2013.

[66] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. *SC Conference*, pages 1–11, 2010.

[67] J. Rahola. Diagonal forms of the translation operators in the fast multipole algorithm for scattering problems. *BIT*, 36(2) :333–358, 1996.

[68] V. Rokhlin. Diagonal forms of translation operators for helmholtz equation in three dimensions. Technical report, DTIC Document, 1992.

[69] J. Song and W.C Chew. The Fast Illinois Solver Code : Requirements and Scaling Properties. *IEEE Computational Science & Engineering*, 5(3) :19–23, July-September 1998.

[70] J. Song, C.C Lu, and W.C Chew. Multilevel fast-multipole algorithm for electromagnetic scattering by large complex objects. *IEEE Antennas and Propagation*, 45(10) :14–19, October 1997.

[71] JM Song and W.C. Chew. Multilevel fast-multipole algorithm for solving combined field integral equations of electromagnetic scattering. *Microwave and Optical Technology Letters*, 10(1) :14–19, 1995.

[72] J.A. Stratton. *Electromagnetic theory*, volume 33. Wiley-IEEE Press, 2007.

[73] X. Sun and N.P. Pitsianis. A matrix version of the fast multipole method. *Siam Review*, 43(2) :289–300, 2001.

[74] G. Sylvand. *La méthode multipôle rapide en électromagnétisme. Performances, parallélisation, applications*. PhD thesis, École des Ponts ParisTech, 2002.

[75] G. Sylvand. Performance of a parallel implementation of the FMM for electromagnetics applications. *International Journal for Numerical Methods in Fluids*, 43(8) :865–879, 2003.

[76] J.M. Taboada, M.G. Araujo, J.M. Bertolo, L. Landesa, F. Obelleiro, and J.L. Rodriguez. MLFMA-FFT parallel algorithm for the solution of large-scale problems in electromagnetics. *Progress In Electromagnetics Research*, 105 :15–30, 2010.

[77] J.M. Tendler, J.S. Dodson, JS Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1) :5–25, 2002.

[78] C.J. Thompson, S. Hahn, and M. Oskin. Using modern graphics architectures for general-purpose computing : a framework and analysis. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 306–317. IEEE Computer Society Press, 2002.

[79] J.E. Thomton. Parallel operation in the control data 6600. *International Workshop on Managing Requirements Knowledge*, 0 :33, 1964.

[80] TOP500.Org. Top500 supercomputer sites, 2013.

[81] H. Topcuoglu, S. Hariri, and M.Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3) :260–274, 2002.

[82] S. Velamparambil and W.C. Chew. Analysis and performance of a distributed memory multilevel fast multipole algorithm. *IEEE Transactions on Antennas and Propagation*, 53(8) :2719–2727, 2005.

[83] RL Wagner, J. Song, and WC Chew. Monte Carlo simulation of electromagnetic scattering from two-dimensional random rough surfaces. *IEEE Transactions on Antennas and Propagation*, 45(2) :235–245, 2002.

[84] C. Waltz, K. Sertel, M.A. Carr, B.C. Usner, and J.L. Volakis. Massively parallel fast multipole method solutions of large electromagnetic scattering problems. *IEEE Transactions on Antennas and Propagation*, 55(6) :1810–1816, 2007.

[85] G. N. Watson. *A treatise on the theory of Bessel functions*. Cambridge University Press, 1966.

[86] R. Clint Whaley. ATLAS Version 3.8 : Overview and Status, 2007.

[87] T. Willhalm and N. Popovici. Putting Intel Threading Building Blocks to work. In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pages 3–4, New York, NY, USA, 2008. ACM.

[88] K. Zhao, M.N. Vouvakis, and J.F. Lee. The adaptive cross approximation algorithm for accelerated method of moments computations of emc problems. *IEEE Transactions on Electromagnetic Compatibility*, 47(4) :763–773, 2005.