Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique : $\mathcal{H}$-Matrices. Parallélisme et applications industrielles

Benoît Lizé

To cite this version:

Benoît Lizé. Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique : $\mathcal{H}$-Matrices. Parallélisme et applications industrielles. Acoustique [physics.class-ph]. Université Paris-Nord - Paris XIII, 2014. Français. NNT : 2014PA132030 . tel-01244260

HAL Id: tel-01244260
https://theses.hal.science/tel-01244260v1

Submitted on 17 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

École Doctorale Galilée

Thèse de doctorat

Discipline : Mathématiques Appliquées

présentée par

Benoît Lizé

Résolution Directe Rapide pour les Éléments Finis de Frontière en Électromagnétisme et Acoustique : $\mathcal{H}$-Matrices. Parallélisme et Applications Industrielles.

---

dirigée par Olivier Lafitte

Soutenue le 17 juin 2014 devant le jury composé de :

M.	Eric Darve	Stanford University	Rapporteur
M.	Laurent Demanet	MIT	Rapporteur
M.	Luc Giraud	INRIA	Examinateur
Mme	Laurence Halpern	Université Paris 13	Examinateur
M.	Olivier Lafitte	Université Paris 13	Directeur
M.	Jean-Claude Nédélec	École Polytechnique	Président
M.	Guillaume Sylvand	Airbus Group Innovations	Examinateur

LAGA, UMR 7539 Institut Galilée
Université Paris 13
99 avenue J.B. Clément
93430 Villetaneuse

École doctorale Galilée, Institut Galilée
99 avenue J.B. Clément
93430 Villetaneuse

Remerciements

Ça ne marchera jamais.
– Thésard Anonyme

De tous temps les hommes...
– Thésard Anonyme

Je referme cette thèse en écrivant sa première page, celle qui sera la plus lue, bien que la page 212 lui volera peut-être la vedette.

Je souhaite en premier lieu remercier les rapporteurs d'outre-atlantique, Messieurs Éric Darve et Laurent Demanet, pour leur relecture, l'attention qu'ils ont portée à ces travaux, et leurs remarques et questions précises et constructives. Je remercie aussi chaleureusement les membres du jury, Luc Giraud, Laurence Halpern et Jean-Claude Nédélec de m'avoir fait l'honneur de leur présence. Merci Olivier, dont l'énergie bondissante et l'enthousiasme sans faille furent des constantes de toute cette thèse, parfois même jusque dans le voisinage d'une piscine tunisienne, alors que l'attrait des changements de variables apparaissait moins évident que celui du repos.

Je souhaite bien entendu remercier l'ensemble de l'équipe de mathématiques appliquées d'Airbus Group Innovations, à Suresnes et Toulouse, dans laquelle il fait bon être depuis 2009. Et en particulier Guillaume Sylvand, dont les ~~coups de fouet~~ encouragements me ~~persécutent~~ motivent depuis mon stage de fin d'études, et qui sera heureux de savoir que la rédaction a avancé. Un merci particulier au joyeux salon `iw_se_am`, machine à café virtuelle de l'équipe, lieu d'échange où se côtoient bons mots (ou pas), bonne humeur, bulletins météo, adoration de N.M., *running gags*, entraide et culture générale (ou pas). Et également au salon-qui-n'existe-pas, même si il-n'existe-plus (n'est-ce pas, Pfffffanny, Ariane et Isabelle ?). Merci à mes collègues de Suresnes, que je ne désignerai pas ici par les surnoms ridicules dont j'ai pu les affubler[1] : Chef Éric, Chef Fabien, Chef Pierre, Lionel, Michel, Nabil, Anabelle, Vassili, Régis et Sofiane. Et particulièrement à mes collègues ou ex-collègues de bureau : Jean-Loup, Fanny et Vincent, avec qui j'ai partagé bien plus qu'un bureau, que ce soit sous forme de Super Bock, malt, pensées philosophiques ou autres. Merci à ceux de Toulouse, surtout croisés lors de leurs venues à Suresnes, au travers du salon ou dans un restaurant toulousaing : Jérôme, Jayant, Nolwenn, Jean-Nicolas et Matthieu. Et aux Toulouso-Suresno-Palaisiens d'IMACS.

1. Une version non censurée de ces pages est disponible sur demande.

Bon courage aux thésards : Antoine, que deux moitiés de stage sur les $\mathcal{H}$-Matrices n'ont pas réussi à dissuader de faire une thèse, et également à Albert, qui veillera à faire de belles courbes et à choisir les bonnes couleurs (Vince veille !).

Merci aux nombreuses personnes rencontrées durant cette thèse, et particulièrement aux équipes HiePACS et RUNTIME d'INRIA Bordeaux. Un grand merci à Emmanuel Agullo pour sa relecture attentive, ses nombreux conseils, son enthousiasme et sa sympathie ; à Samuel Thibault et Nathalie Furmento pour leur réactivité, leur disponibilité et leur aide avec StarPU.

Merci à mes amis, que je ne cite pas ici nominativement, par peur d'un oubli qui pourrait réduire leur nombre. Une thèse sans eux aurait été comme une mauvaise métaphore, une blague qui ne serait faite qu'une fois, comme une phrase inache–
Un profond merci à mes parents, mon frère et mes sœurs. Merci aux 0 et aux 1, au papier à petits carreaux, à Georges Pompidou, à l'oxygène et à l'eau, sans qui je ne serais pas là.

Enfin, si je ne devais remercier qu'une personne, ce serait Andréa. Merci pour le passé, le présent, et le futur.

Résumé

La méthode des éléments finis de frontière (BEM) requiert la résolution de systèmes linéaires pleins, mal conditionnés de grande dimension avec un nombre important de seconds membres ; la difficulté de résolution est le principal obstacle pratique à son utilisation. Une matrice hiérarchique ($\mathcal{H}$-Matrice) est un format de stockage hiérarchique, creux et approché de matrices dont la manipulation permet la réalisation d'un solveur linéaire direct avec une complexité asymptotique en $\mathcal{O}(N \log_2^\alpha(N))$ en espace et en temps.
On s'intéresse aux $\mathcal{H}$-Matrices pour les BEM, avec l'évaluation et la mise en œuvre des $\mathcal{H}$-Matrices sur des cas académiques et industriels complexes d'une part ; et la parallélisation efficace des ces algorithmes d'autre part. Une étude paramétrique rigoureuse et détaillée sur des cas modèles et industriels est présentée. On précise le domaine d'application des $\mathcal{H}$-Matrices pour les BEM, donne des recommendations sur les paramètres des algorithmes, et décrit des optimisations réalisables. Nous montrons la pertinence des $\mathcal{H}$-Matrices en termes de précision et temps de calcul, en comparaison avec un solveur direct classique et avec un solveur itératif basé sur la méthode multipôle rapide.
La parallélisation des algorithmes, en mémoire partagée puis distribuée, repose sur l'expression des calculs sous forme de graphes de tâches composables, dont l'ordonnancement est effectué dynamiquement à l'aide d'un moteur d'exécution. Nous donnons une expression permettant de contrôler la granularité des opérations au travers d'une coupe paramétrable des $\mathcal{H}$-Matrices, et exposons diverses optimisations et extensions de ce formalisme. Une efficacité parallèle quasi-optimale est obtenue en mémoire partagée, ainsi que des résultats prometteurs en mémoire distribuée.

Mots-clefs BEM, Solveur Direct Rapide, H-Matrice, Parallélisme, Électromagnétisme, Acoustique

---

Fast Direct Solver for the Boundary Element Method in Electromagnetism and Acoustics: $\mathcal{H}$-Matrices. Parallelism and Industrial Applications.

Abstract

The Boundary Element Method (BEM) requires the solving of large, ill-conditionned, dense linear systems with a large number of righ-hand sides ; the solving difficulty is the main practical obstacle to its use. A hierarchical matrix ($\mathcal{H}-$Matrix) is a hierachical, approximate, data-sparse storage format for matrices that can be manipulated to produce a direct linear solver with an asymptotic space and time complexity of $\mathcal{O}(N \log_2^\alpha(N))$.
We consider the $\mathcal{H}-$Matrices for the BEM, with the assessment and application of $\mathcal{H}-$Matrices to complex academic and industrial test cases on the one hand ; and the efficient parallelization of these algorithms on the other hand. A rigorous and detailed parametric study on model and industrial test cases is shown. We define the working range of $\mathcal{H}$-Matrices for the BEM, provide recommendations for the algorithm parameters, and describe some practical optimizations. We show the relevance of the $\mathcal{H}$-Matrices in terms of precision and computation time, in comparison with a classical direct solver and with an iterative solver based on the Fast Multipole Method.
The parallelization of the algorithms, in shared and distributed memory, relies on the expression of the computations with composable task graphs, dynamically scheduled with the help of a runtime system. We give an expression controlling the operations granularity through a configurable cut of the $\mathcal{H}$-Matrices, and present various optimizations and extensions of this expression. An almost optimal parallel efficiency is achieved in shared memory, along with promising results in distributed memory.

Keywords BEM, Fast Direct Solver, H-Matrix, Parallelism, Electromagnetism, Acoustics

Table des matières

Liste des Acronymes **11**

Notations **13**

Introduction **15**

1 Contexte industriel **19**

1.1 Exemple de cavités en aéronautique 20

1.2 Remarques sur les cavités 22

1.3 Applications 23

1.4 Modélisation du canal de communication 24

1.4.1 Facteur de qualité 26

1.5 Position du problème 26

1.6 Brève présentation des méthodes de simulation classiques 27

1.6.1 Différences finies en domaine temporel 27

1.6.2 Éléments finis de frontière 29

1.6.3 Théorie géométrique de la diffraction 31

1.7 Méthodes énergétiques 33

1.7.1 Statistical Energy Analysis (SEA) 34

1.7.2 Radiosité 35

1.8 Conclusion 37

2 Introduction aux $\mathcal{H}$-Matrices **39**

2.1 Contexte et motivation 39

2.2 Diffraction d'une onde électromagnétique 40

2.2.1 Équations de Maxwell harmoniques 40

2.2.2 Théorème de représentation intégrale 42

2.2.3 Discrétisation 44

2.2.4 Écriture matricielle 45

2.2.5 Remarques sur la résolution numérique 48
2.3 $\mathcal{H}$-Matrices : Principes de base 50
2.3.1 Deux "ingrédients" 51
2.4 $\mathcal{R}k$-Matrices 53
2.4.1 Décomposition en valeurs singulières 54
2.4.2 Algèbre des $\mathcal{R}k$-Matrices 56
2.5 Algorithme mono-niveau 60
2.6 Algorithme hiérarchique 62

3 Algorithmes séquentiels 65
3.1 Découpage spatial 65
3.1.1 Arbres 66
3.1.2 Algorithme 67
3.1.3 Position du plan de séparation 67
3.1.4 Arbre de groupes 71
3.2 Compression 76
3.2.1 Adaptative Cross Approximation 77
3.2.2 Pivotage total 78
3.2.3 Pivotage partiel 80
3.2.4 Contre-exemple pour le pivotage partiel et ACA+ 83
3.3 Opérations sur les $\mathcal{H}$-Matrices 87
3.3.1 $\mathcal{H}$-BLAS et $\mathcal{H}$-LAPACK 88
3.3.2 Assemblage, Produit matrice-vecteur et Addition 90
3.3.3 Multiplication, Inversion, Factorisations 96
3.4 Complexité 106
3.4.1 Estimations 106
3.4.2 Structure de l'arbre et constantes 108
3.5 Implémentation et premiers résultats 109
3.5.1 Cylindres 110
3.5.2 Autres exemples 128
3.6 Analyse et optimisations 132
3.6.1 Analyse de la compression sur des cas réalistes 132
3.6.2 Consommation mémoire et répartition du temps de calcul 137
3.6.3 Recompression 139
3.7 Conclusion 142

4 Algorithmes parallèles en mémoire partagée **145**
4.1 Introduction . . . 146
4.1.1 Remarques préliminaires . . . 146
4.1.2 Cadre . . . 148
4.1.3 État de l'Art et contributions . . . 150
4.2 Efficacité parallèle . . . 153
4.3 Ordonnancement par liste et algorithmes *Bulk Synchronous Parallel* (BSP) 155
4.3.1 Assemblage . . . 155
4.3.2 Produit Matrice-Vecteur . . . 156
4.3.3 Produit matriciel . . . 157
4.3.4 Inversion . . . 160
4.3.5 Conclusions . . . 161
4.4 Graphe de tâches et moteur d'exécution . . . 162
4.4.1 Graphe de tâches . . . 162
4.4.2 Moteur d'exécution . . . 164
4.4.3 Présentation des moteurs d'exécution . . . 166
4.5 Algorithmes BSP pour les $\mathcal{H}$-Matrices à l'aide de graphes de tâches . . . 168
4.5.1 Multiplication . . . 168
4.5.2 Inversion . . . 170
4.5.3 Décomposition *LU* . . . 171
4.5.4 Performances . . . 171
4.6 Parallélisation à l'aide de graphes de tâches . . . 177
4.6.1 Exemples de composition . . . 177
4.6.2 Décomposition en tâches . . . 179
4.6.3 Découpage des tâches . . . 180
4.6.4 Données et tâches . . . 183
4.6.5 Opérations . . . 187
4.7 Implémentation et performances . . . 196
4.7.1 Implémentation . . . 196
4.7.2 Performances . . . 204
4.8 Conclusion . . . 210

5 Mémoire distribuée et architectures hybrides **213**
5.1 Mémoire distribuée . . . 214
5.1.1 Cadre . . . 214
5.1.2 Graphes de tâches et mémoire distribuée . . . 217
5.1.3 Implémentation . . . 221
5.1.4 Performances . . . 225
5.2 Perspectives : Architectures hybrides et *Out-Of-Core* . . . 229
5.2.1 Architectures hybrides . . . 230
5.2.2 Out-Of-Core . . . 236
5.2.3 Contexte . . . 236
5.2.4 Application aux $\mathcal{H}$-Matrices . . . 236
5.2.5 Conclusion . . . 238

6 Applications **239**

6.1 Exemples industriels . . . 239

6.1.1 Antenne VHF sur un avion . . . 240

6.1.2 Antenne patch . . . 241

6.1.3 Section équivalente radar de missile . . . 243

6.1.4 Méta-surface . . . 245

6.1.5 Cavité avec obstacle interne . . . 246

6.1.6 Cavité résonnante . . . 248

6.1.7 Acoustique : voiture . . . 249

6.1.8 Conclusions et perspectives . . . 251

6.2 Autres applications . . . 252

6.2.1 Guides d'ondes et décomposition de domaine . . . 252

6.2.2 Décomposition de domaine . . . 258

6.2.3 Application aux statistiques : exemple du Krigeage . . . 259

6.3 Conclusion . . . 263

Conclusion **265**

Bibliographie **267**

Liste des Acronymes

ACA	*Adaptative Cross Approximation*
BEM	*Boundary Element Method*
BSP	*Bulk Synchronous Parallel*
CFIE	*Combined Field Integral Equation*
CRBM	Chambre Réverbérantes à Brassage de Mode
DAG	*Directed Acyclic Graph*
EFIE	*Electric Field Integral Equation*
FDTD	Différences Finies en Domaine Temporel
FIFO	*First In, First Out*
FMM	*Fast Multipole Method*
HCA	*Hybrid Cross Approximation*
LIFO	*Last In, First Out*
LPT	*Longest Process Time*
MFIE	*Magnetic Field Integral Equation*
MPI	*Message Passing Interface*
ccNUMA	*Cache Coherent Non Uniform Memory Access*
RAM	*Random Access Memory*
SEA	*Statistical Energy Analysis*
SER	Section Efficace Radar
SHM	*Structural Health Monitoring*
SMP	*Symmetric Multi-Processing*
SPMD	*Same Program, Multiple Data*
SVD	*Singular Values Decomposition*
UTD	*Uniform Theory of Diffraction*

Notations

Ω	Objet fermé borné, compact de $\mathbb{R}^3$
Γ	Frontière de Ω. $\Gamma := \partial\Omega$
ε, μ	Permittivité électrique, perméabilité magnétique. Relatif au vide avec un indice 0.
κ	Nombre d'onde. $\kappa := \frac{2\pi}{\lambda}$, avec λ la longueur d'onde.
$G(r)$	Noyau de Green de l'équation de Helmholtz. $G(r) = \frac{e^{i\kappa r}}{4\pi\|r\|}$
I	Ensemble des degrés de liberté ligne. $\|I\| := M$
J	Ensemble des degrés de liberté colonne. $\|J\| := N$, $M \neq N$ à priori
$\mathcal{T}_I$	Arbre de groupes construit sur I
$\mathcal{T}_{I \times J}$	Arbre de blocs construit sur I et J
$\mathcal{L}(\mathcal{T})$	Ensemble des feuilles d'un arbre $\mathcal{T}$
$\mathcal{N}(\mathcal{T})$	Ensemble des nœuds d'un arbre $\mathcal{T}$
σ	Sous-ensemble des degrés de liberté « ligne » $\subset I$
τ	Sous-ensemble des degrés de liberté « colonne » $\subset J$
m	$\|\sigma\|$, nombre de degrés de liberté ligne
n	$\|\tau\|$, nombre de degrés de liberté colonne
M	Matrice d'interaction $\in \mathbb{C}^{M \times N}$
$M\|_{\sigma \times \tau}$	Sous-bloc de M correspondant aux degrés de liberté σ et τ
η	Constante d'admissibilité dans le critère (3.1.2)
$\tau(M)$	Taux de compression d'une $\mathcal{H}$-Matrice, définition 3.20
$L_0(M)$	Coupe dans une $\mathcal{H}$-Matrice M

Introduction

La propagation des ondes acoustiques et électromagnétiques est un sujet d'importance dans l'industrie aéronautique, du fait de la variété des domaines nécessitant une modélisation fine de celle-ci. Concernant les ondes électromagnétiques, nous pouvons citer la conception et l'installation des antennes (avec en particulier les interactions entre celles-ci), la compatibilité électromagnétique, la furtivité radar et la résistance aux agressions extérieures pour les applications militaires. Les applications acoustiques concernent entre autres la modélisation de la propagation du bruit généré par les moteurs d'avion, ou par frottement aérodynamique.

À ces applications traditionnelles s'ajoute un autre type dont le développement est plus récent : les transmissions sans fil. En effet, de plus en plus d'appareils sont dotés de capacités de communication sans fil, ce qui présente de nouvelles opportunités, mais soulève également des problèmes de compatibilité électromagnétique, de propagation, de modélisation de canaux de communication, et de fiabilité. Ce type de communication est avantageux, car il permet de simplifier le câblage des appareils qui, comme le démontre l'histoire récente de l'aviation, est un problème de conception significatif, réduisant ainsi le poids, et pouvant potentiellement augmenter la fiabilité des communications. Il permet aussi d'envisager de nouvelles approches comme le *Structural Health Monitoring* (SHM), dont la mise en œuvre repose sur un ensemble de capteurs distribués dans le système et ayant pour rôle de surveiller des indicateurs locaux de dégradation. De tels capteurs sont potentiellement difficiles d'accès, ce qui rend les communications sans fil nécessaires. Une autre application plus lointaine est celle du « *Fly by wireless* », visant à remplacer certains canaux de commande par des liaisons sans fil, dans le but de simplifier la conception, la maintenance et d'alléger la structure. Enfin, il existe une famille d'applications ayant trait au confort des passagers, et dont la criticité est plus faible. Nous pouvons citer ici la présence du *Wi-Fi* en cabine et la commande des appareils de divertissement tel la vidéo à la demande.

La méthode des éléments finis de frontière (BEM) a démontré son efficacité pour résoudre des problèmes difficiles avec précision. Elle est déployée industriellement avec succès depuis plus d'une décennie, en particulier dans le secteur aéronautique. Cependant, outre les difficultés de modélisation, des obstacles de taille s'opposent à la résolution numérique des équations issues de cette formulation. Essentiellement, ces obstacles peuvent se résumer dans la nécessité de résoudre un système d'équations linéaires complexes plein de grande dimension. La résolution de ce système – et dans une moindre mesure sa création – est le

point bloquant majeur dans l'application de cette méthode aux situations décrites ici. En effet, avec N le nombre d'inconnues, la création du système nécessite $\mathcal{O}(N^2)$ opérations (et autant de mémoire disponible), et sa résolution $\mathcal{O}(N^3)$, N étant généralement compris entre 10^6 et 10^8.

Diverses méthodes d'accélération applicables aux éléments finis de frontière existent, dont la méthode multipôle rapide (FMM). Celle-ci a rencontré un succès indéniable, permettant d'élargir considérablement le champ d'application de la méthode des éléments finis de frontière. Cette méthode permet de ramener la complexité de résolution des équations à $\mathcal{O}(n_{iter} n_{rhs} N \log(N))$, avec n_{iter} le nombre d'itérations avant convergence, et n_{rhs} le nombre de second membres des équations. Malheureusement, le nombre d'itérations n'est borné que par N, et peut en pratique devenir très grand.

De plus, les applications visées ici mènent à des valeurs élevées pour ces deux quantités n_{rhs} et n_{iter}. L'enjeu de cette thèse est de proposer un solveur ne souffrant pas de ces inconvénients. Ce solveur utilisera la famille d'algorithmes connue sous le nom de $\mathcal{H}$-Matrices. Ceux-ci permettent de réaliser un solveur direct rapide, avec une complexité asymptotique meilleure que $\mathcal{O}(N^2)$ en temps et en espace. L'application de ces algorithmes aux éléments finis de frontière pour la physique des ondes n'a été que faiblement étudiée. Nous proposons ici une étude pratique rigoureuse de ces algorithmes dans le contexte d'applications industrielles en électromagnétisme et acoustique. Par ailleurs, le besoin de performance lié à la complexité des applications requiert une parallélisation efficace de ces algorithmes. Ce sera l'objet de la seconde partie de cette thèse.

Organisation de la thèse

L'organisation de cette thèse suit naturellement la progression de cette introduction : après une présentation du problème industriel et de ses spécificités, nous introduisons les éléments finis de frontière et les $\mathcal{H}$-Matrices. La description détaillée des algorithmes dans leur version séquentielle suit, ainsi que l'évaluation de cette méthode pour des problèmes d'électromagnétisme. La présentation d'une parallélisation nouvelle et efficace des $\mathcal{H}$-Matrices est donnée, ainsi que ses extensions à des architectures plus complexes. Enfin, cette thèse s'achève par l'évaluation du solveur sur des configurations industrielles, et des ouvertures sur d'autres champs d'applications de la méthode et du solveur.

Le **chapitre 1** introduit le problème industriel motivant cette thèse. Il s'agit de l'étude de la propagation d'ondes électromagnétiques dans des cavités réfléchissantes de grande dimension. Nous exposons le problème, son importance pour les applications aéronautiques et ses particularités. Nous donnons ensuite une étude comparative et critique des méthodes de simulation courantes à la lumière de ces spécificités, en soulignant les points bloquants pour chacune de ces méthodes. La conclusion de cet état de l'art des méthodes de simulation nous invite à considérer le problème de la résolution numérique par la méthode des éléments finis de frontière, et son accélération.

Le **chapitre 2** a pour point de départ la méthode des éléments finis de frontière. La discrétisation des équations de Maxwell par cette méthode est rappelée, et s'achève par l'écriture du système matriciel. Une analyse comparative des méthodes de résolution de ce système introduit les $\mathcal{H}$-Matrices, dont les principes de base sont exposés. En particulier,

la compression des blocs d'interactions lointaines est illustrée, et l'algorithme mono-niveau l'appliquant est esquissé.

Le **chapitre 3** est consacré aux algorithmes séquentiels pour les $\mathcal{H}$-Matrices, et l'application aux éléments finis de frontière. La première partie de ce chapitre donne une description détaillée des algorithmes relatifs aux $\mathcal{H}$-Matrices, et en particulier de plusieurs méthodes de compression des blocs. L'analogue d'un sous-ensemble de BLAS et LAPACK pour les $\mathcal{H}$-Matrices est décrit, plus précisément les opérations nécessaires pour l'implémentation d'un solveur direct. La deuxième partie de ce chapitre est une étude pratique détaillée des algorithmes dans le contexte des éléments finis de frontière pour l'électromagnétisme. Les tests sont réalisés au sein d'un code industriel 3D, avec pour objectif de déterminer la précision et les performances du solveur, et de donner des valeurs recommandées des paramètres des algorithmes. Par ailleurs, une analyse du temps de calcul, de la consommation mémoire et des optimisations sont présentées.

Le **chapitre 4** a pour objet la parallélisation en mémoire partagée des algorithmes séquentiels du chapitre précédent. Un formalisme basé sur des graphes de tâches est décrit. Celui-ci s'appuie sur un moteur d'exécution pour la gestion du parallélisme dynamique. Les algorithmes de la littérature pour la parallélisation des $\mathcal{H}$-Matrices sont reformulés avec des graphes de tâches, et leur efficacité évaluée. Une formulation nouvelle à base de graphes de tâches pour les $\mathcal{H}$-Matrices est ensuite décrite, et ses performances comparées. Celle-ci permet une exploitation presque optimale des ressources de calcul en mémoire partagée, et est la contribution majeure de cette thèse.

Le **chapitre 5** explore des extensions du formalisme du chapitre 4 à d'autres architectures. Le cas de la mémoire distribuée est traité, et le formalisme adopté permet d'obtenir un bon passage à l'échelle sur ces architectures. Des perspectives pour l'élargissement des travaux de cette thèse aux architectures hybrides et à une implémentation *Out-Of-Core* clôt ce chapitre. Dans les deux cas, les extensions nécessaires, difficultés et bénéfices attendus sont précisés.

Enfin, le **chapitre 6** présente des applications industrielles du solveur $\mathcal{H}$-Matrice en électromagnétisme et acoustique. Les applications couvrent un large champ de configurations, et permettent de démontrer les avantages pratiques de la méthode. Une comparaison de la précision et du temps de calcul avec un solveur classique, FMM et des mesures est donnée lorsque cela est possible. Ce chapitre comprend également des ouvertures sur le traitement de fonctionnalités avancées pour les éléments finis de frontières, et les avantages du solveur $\mathcal{H}$-Matrice dans ce contexte. Ces fonctionnalités sont la gestion des guides d'onde et la décomposition de domaines. Une ouverture sur le champ d'application très prometteur est également donnée, avec le cas du krigeage en statistiques.

Contributions

Les principales contributions de cette thèse sont :

— D'une part, la mise en œuvre dans un contexte industriel des $\mathcal{H}$-Matrices. Le cadre commun est la famille de logiciels de simulation ASERIS pour l'électromagnétisme et ACTIPOLE pour l'acoustique, famille déployée industriellement depuis plus d'une décennie. Une étude paramétrique rigoureuse de la méthode est effectuée. Celle-ci

permet de dégager des recommandations pour les valeurs des paramètres laissés libres dans la description des algorithmes. Elle donne également des éléments d'explication sur le comportement asymptotique de la méthode, les points critiques, et des optimisations réalisables. Nous nous attachons également à ouvrir le domaine d'application le plus large possible à cette méthode par la gestion d'un nombre maximal de fonctionnalités de ces codes. Ainsi, le produit de ce travail est actuellement déployé industriellement au sein d'ASERIS et ACTIPOLE. Nous donnons des comparaisons avec l'état de l'Art sur des configurations réalistes connues pour leur difficulté.

— D'autre part, la parallélisation en mémoire partagée puis distribuée de cette méthode, en utilisant des techniques innovantes et récentes de parallélisation, avec une ouverture sur les architectures matérielles hybrides. Nous mettons en lumière les difficultés associées à la parallélisation de cette famille d'algorithme, et proposons une méthode permettant de les contourner. En particulier, nous faisons appel à des méthodes récentes de parallélisation reposant sur la formalisation du calcul sous forme de graphe de tâches, puis au traitement de ce graphe par un moteur d'exécution adapté (« runtime »). Il s'agit à notre connaissance de la première utilisation de ces méthodes pour des problèmes de cette complexité, et cette thèse a contribué à l'enrichissement des fonctionnalités d'un moteur d'exécution, StarPU. Une comparaison de cette approche avec la littérature est présentée ; l'implémentation et l'évaluation sont exposées. Les méthodes de parallélisation employées sont susceptible d'être pertinente pour d'autres problème, comme les solveurs directs creux ou la méthode multipôle rapide.

Chapitre 1

Contexte industriel

Sommaire

1.1 Exemple de cavités en aéronautique 20
1.2 Remarques sur les cavités 22
1.3 Applications 23
1.4 Modélisation du canal de communication 24
1.5 Position du problème 26
1.6 Brève présentation des méthodes de simulation classiques . . . 27
1.7 Méthodes énergétiques 33
1.8 Conclusion 37

La simulation numérique est une composante essentielle de la conception des systèmes complexes. Elle intervient à de multiples niveaux du cycle « en V » de développement et est intimement liée aux niveaux de précision, rapidité et robustesse souhaités. Ces étapes sont, dans l'ordre de descente du « V » : analyse des besoins et faisabilité, spécifications, conception architecturale, conception détaillée, réalisation ; et dans la remontée : tests unitaires, tests d'intégration, tests de validation et recette.

Dans les phases de haut niveau de ce cycle, les modèles employés sont robustes, rapides, facilemenent interprétables et généraux, alors que dans les phases de plus bas niveau, ils seront précis, discriminants, et spécialisés.

Ceci permet de guider la phase d'exploration, de faciliter la compréhension des phénomènes, et de tester virtuellement des configurations non réalisables par manque de temps, de moyens financiers ou d'instruments de mesure adaptés. De ce fait, la simulation permet de raccourcir les temps d'itération dans chacune des ces phases, et de limiter les retours en arrière lors des phases d'intégration et de validation.

Dans le domaine de la propagation des ondes, les codes de simulation permettent d'obtenir des résultats très précis sur une physique complexe, et ont été développés depuis de nombreuses années, en premier avec des méthodes de type Différences Finies en Domaine Temporel (FDTD), et plus récemment avec les méthodes de type éléments finis, volumiques ou de frontière.

Ces types de méthodes répondent à des besoins issus initialement en grande partie du domaine militaire, en particulier de la furtivité et de la résistance aux agressions extérieures. Cependant la liste des domaines d'application des méthodes de simulation de la propagation des ondes est bien plus large. En particulier, la démocratisation et la généralisation des moyens de communication sans fil est un des domaines dans lesquels la simulation est nécessaire. Dans le contexte aéronautique, cette propagation sans fil présente un ensemble de difficultés. Le milieu de propagation est relativement clos et réfléchissant d'un point de vue électromagnétique (essentiellement métallique), il est de grande dimension devant la longueur d'onde qui est la plupart du temps centimétrique pour les applications de communication sans fil, et de nombreux instruments sont présents, susceptibles de polluer le signal, ou de voir leur fonctionnement perturbé. Dans ce cas, le rapport entre une dimension caractéristique l du milieu et le nombre d'onde κ est tel que $\kappa l \gg 1$, ce qui correspond au cadre des applications dites haute fréquence.

Malheureusement, la majorité des méthodes de calcul disponibles à l'heure actuelle n'est pas adaptée à de telles configurations. L'enjeu de cette thèse est de proposer des méthodes et algorithmes de simulation numérique nouveaux à même d'élargir le champ d'application à un environnement réfléchissant de grande taille, soumis à des excitations multiples, et souvent mal caractérisées.

Plan

Nous présenterons ce problème dans le cadre de la modélisation des communications sans fil dans une cavité de grande dimension (devant la longueur d'onde). Après une présentation des caractéristiques des cavités rencontrées dans l'industrie aéronautique, nous donnerons un exemple d'application mettant en évidence les observables pertinentes pour la conception. Nous poursuivrons par une brève présentation des méthodes de simulation classiques. Nous nous attacherons à souligner les avantages et inconvénients de chaque méthode. Enfin, ce chapitre ouvrira vers l'approche retenue, à savoir l'augmentation de la performance de résolution des équations de Maxwell par la méthode des éléments de frontière.

L'approche proposée ayant des applications au-delà du problème industriel formulé dans ce chapitre, le reste de ce document n'y sera pas spécifique, bien que la motivation initiale se trouve ici.

1.1 Exemple de cavités en aéronautique

Les structures aéronautiques et le fuselage d'un avion en premier lieu, sont majoritairement constituées de pièces métalliques. Bien que les avions modernes soient de plus en plus des assemblages de matériaux composites, ils se comportent électriquement largement comme des matériaux conducteurs. Ceci peut être lié à la présence de matériaux métalliques au sein de la structure composite, parfois sous la forme d'un fin grillage. De plus, les résines carbones ont une conductivité électrique croissante avec la fréquence d'excitation, et sont largement conductrices dans les bandes de fréquences considérées ici. Ces structures (fuselage, aile, empennage, réservoir, soute, *etc.*) sont soumises à des excitations électromagnétiques diverses. Outre le réseau électrique de l'avion (dont la puissance est

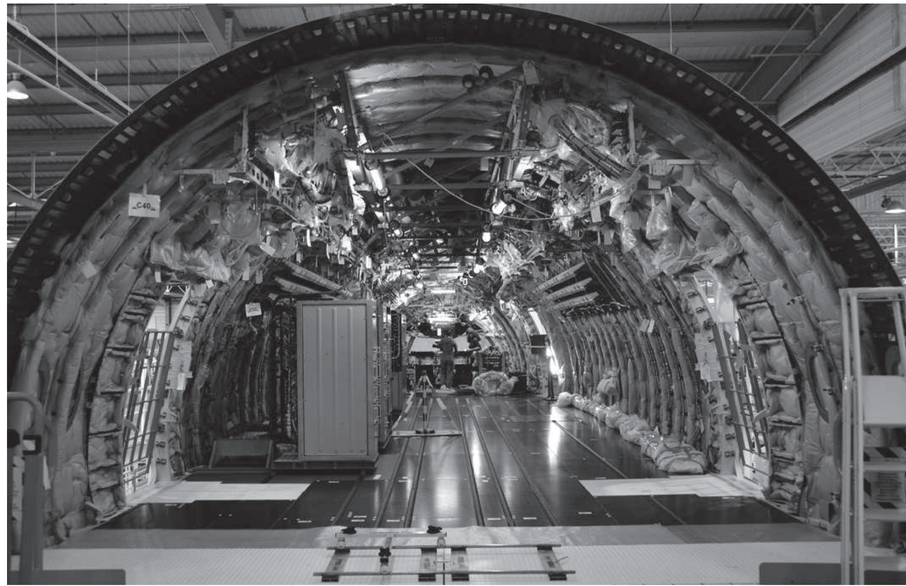


Figure 1.1 – Exemple de fuselage en cours de construction

en constante augmentation) à très basse fréquence, il faut noter la pollution large bande des divers équipements, des antennes de communication externes, et des divers dispositifs de communication sans fil.

Ces excitations couvrent une bande de fréquence très large, allant de quelque dizaines de Hz à la dizaine de GHz. Elles sont souvent mal connues, maîtrisées par des marges (modélisation du pire cas), et en constante évolution. Par exemple, les fréquences utilisées par les téléphones portables peuvent s'étaler de 700MHz (réseau 4G LTE) à plus de 2GHz (4G LTE à 2.6GHz, Wi-Fi à 5GHz, *etc.*), avec l'apparition de nouvelles bandes en quelques années, sur une échelle de temps bien inférieure à la durée de vie d'un appareil.

Dans cette introduction, nous prendrons un exemple très simplifié de cavité de type « cabine ». L'objectif de cette section n'est pas de présenter une spécification réaliste de cavité, mais de mettre en évidence les caractéristiques principales d'importance pour la modélisation. Cette description étant grossière, elle est également valable pour d'autres types de cavités présentant des caractéristiques plus ou moins éloignées, comme par exemple une soute, une aile ou une section d'empennage.

Nous considérons donc ici une cavité dont les caractéristiques grossières sont :

Dimensions $50 \times 6 \times 2$m

Matériaux Métal (enveloppe), diélectriques (habillage, sièges), passagers, *etc.*

Fréquences $1 - 60$GHz ($\lambda = 30$cm – 5mm respectivement)

Dimension maximale en longueurs d'onde $\sim 10000\lambda$

Il est important d'insister sur la grande variabilité de la configuration de ces cavités, et leur grande complexité géométrique, illustrée par la figure 1.1. Un fuselage d'avion comprend en effet un très grand nombre de sous-structures et de câbles. La connaissance fine de la géométrie et des caractéristiques de ces sous-structures n'est en général pas possible (du fait de la variabilité), non raisonnable du point de vue de la modélisation ou très tardive.

Il existe bien entendu un ensemble de variations possibles par rapport à ces caractéristiques, et il est possible de grossièrement classer les cavités en plusieurs types. On note par ailleurs que cette description peut également s'appliquer aux Chambre Réverbérantes à Brassage de Mode (CRBM), dont la modélisation est également un sujet d'intérêt.

Les CRBM sont des moyens de mesure utilisés en compatibilité électromagnétique. Ce sont des cavités métalliques conçues pour présenter le plus de réflexion possible, caractérisée par un « facteur de qualité ». Dans un modèle idéalisé, une telle cavité est close, ce qui n'autorise que la présence d'ondes stationnaires à l'intérieur de celle-ci, et donc toute solution des équations de Maxwell dans la cavité est une superposition de modes dépendants de la géométrie et des conditions aux limites (qui sont celles d'un conducteur parfait dans ce modèle idéalisé). La densité modale augmentant avec la fréquence, le champ devient, à partir d'une certaine fréquence, relativement isotrope et homogène, ce qui est l'objectif recherché. Ces cavités sont de plus munies d'un « brasseur de mode » qui est un appareil mobile dans la cavité dont le rôle est de modifier les caractéristiques modales de la structure pendant une mesure. Le but recherché est d'obtenir un champ le plus uniforme possible dans l'ensemble de la structure. Ceci est souhaitable afin de tester et qualifier la résistance d'équipements à des agressions électromagnétiques de manière plus exhaustive qu'avec d'autres moyens de mesure pour lesquels l'illumination est particulière et possiblement non représentative des modes d'agression réels de l'équipement.

1.2 Remarques sur les cavités

Une description aussi sommaire que celle-ci est néanmoins suffisante pour dégager les observations suivantes :

- La **géométrie** de la cavité ne peut être connue avec certitude au niveau de la longueur d'onde, du fait des tolérances de fabrication, des déformations intervenant au cours d'un vol ou de la durée de vie de l'avion ;
- Les **matériaux** composant la cavité sont mal connus, et varient au cours du temps en termes de composition, localisation, géométrie. De ce fait, les conditions limites précises ne peuvent être connues ;
- La **variabilité des paramètres** entraîne une instabilité des mesures effectuées dans la cavité. En particulier, la valeur instantanée du champ perd du sens. La phase des mesures est alors très instable, et de plus l'étendue géométrique des émetteurs et capteurs et la précision de leur localisation engendrent de grandes incertitudes sur la phase des mesures.
- La **configuration « désordonnée »** de la cavité ne tend pas à favoriser un comportement de type modal, qui est celui susceptible de ressortir d'une solution portant sur une cavité idéalisée et simplifiée, particulièrement dans le cas des méthodes opérant dans le domaine fréquentiel.

Ces contraintes rendent les mesures difficiles, et la compréhension de la phénoménologie ardue. Il est donc nécessaire de développer des méthodes numériques adaptées à cette classe de problèmes, et dont les observables sont pertinentes pour cette situation.

1.3 Applications

Comme évoqué précédemment, une application visée est celle des transmissions sans fil dans une cavité réfléchissante de grande dimension. Dans ce cadre, un point essentiel est celui de la modélisation du canal de communication. Il s'agit d'établir les caractéristiques de l'environnement pertinentes pour la sélection de technologies de communication sans fil. Ces caractéristiques peuvent avoir trait au positionnement des antennes, au type de correction d'erreur nécessaire, à la robustesse du canal aux variations de l'environnement, et au débit possible.

Dans les applications domestiques, aucune garantie de fiabilité des transmissions ou de disponibilité du canal de communication n'est offerte (on parle de « *best effort* »). L'exemple le plus courant est celui du « Wi-Fi » qui n'offre aucune garantie, et dont les performances sont susceptibles de varier fortement. Dans le cas de l'industrie aéronautique l'exigence de fiabilité et de certification du canal de communication renforce le besoin de modélisation.

Une des grandeurs d'importance pour la modélisation d'un canal de communication est celle de la perte médiane d'un chemin, ou « *median path loss* ». Il s'agit de caractériser la perte médiane d'énergie entre deux points, lesquels peuvent être en vue directe (*line of sight*, LOS) ou non (NLOS) en fonction de la présence ou non d'obstacles entre les deux points. Dans tous les cas, il est fortement probable que la transmission entre les deux points ait de multiples contributeurs, on parle alors de *multipath*.

L'existence de multiples chemins de propagation peut être un obstacle, en « brouillant » les signaux par déphasage, mais elle est également mise à profit dans les méthodes de transmission MIMO (*Multiple Input Multiple Output*) dont un exemple omniprésent est celui de la norme « Wi-Fi » IEEE 802.11n, au travers des très nombreux appareils connectés du quotidien.

Différents types de renseignements sont alors recherchés, que nous classons comme suit :

Variation spatiale Il est important de déterminer la distribution spatiale de l'énergie dans la cavité, afin de calculer le *path loss*, ainsi que le *fading*, à petite ou grande échelle, ce terme désignant les variations autour de la valeur médiane des pertes. À grande échelle, celui-ci correspond aux effets des phénomènes de masquage, et aux effets combinés de la multiplicité des chemins possibles entre un émetteur et un récepteur. La variation locale (*small scale fading*) ne peut sans doute pas être déterminée de manière précise et déterministe, car elle n'est pas suffisamment stable dans le temps. Il faut alors en rechercher une caractérisation statistique adaptée au problème de la modélisation de canal. En particulier, la recherche d'un intervalle de confiance raisonnable sera suffisante dans bien des cas pour le dimensionnement.

Réponse temporelle Une cavité peut être caractérisée en un certain sens par sa réponse impulsionnelle. De manière schématique, la réponse impulsionnelle présente 3 caractéristiques :

— Un ou plusieurs »pics« correspondant au trajet direct (si il existe) et aux premiers trajets les plus directs entre l'émetteur et le récepteur. Il est nécessaire de caractériser le profil de ces pics, et en particulier le délai entre ceux-ci, car

il est critique pour déterminer le débit maximal du canal de communication. Il est également important de connaître l'atténuation entre deux pics successifs.

— Une réponse « diffuse », pouvant être interprétée de manière schématique comme la somme de pics se recouvrant, empêchant une interprétation simple comme pour les premières réponses.

— Cette réponse diffuse se place ensuite dans une enveloppe « exponentielle », dont le temps caractéristique est une quantité d'intérêt. Le temps caractéristique de cette décroissance est appelé facteur de qualité. Celui-ci est une grandeur importante pour la caractérisation des CRBM par exemple. Il est accessible par la mesure, et est une grandeur permettant de discriminer l'applicabilité de méthodes de calcul à une cavité.

Selon la partie de la réponse dont on souhaite faire l'étude, les méthodes de modélisation seront différentes, en particulier une approche analytique simple ou statistique peut être nécessaire pour certaines de ces composantes. Une des grandeurs d'intérêt est le délai entre l'arrivée des différents rayons issus de la propagation selon plusieurs chemins. On s'intéresse soit à la moyenne quadratique, soit à l'écart maximal pour déterminer la bande passante maximale. En notant σ la racine de la moyenne quadratique des délais, la bande passante de cohérence est alors $B \simeq 1/5\sigma$, qui donne une estimation du temps pendant lequel le canal peut être considéré comme plat. Si la bande passante de cohérence est inférieure à la bande passante du canal de communication, il est alors nécessaire d'égaliser le signal pour créer un canal de communication plat.

Temps de cohérence Si l'émetteur ou le récepteur est en mouvement, le signal peut être élargi en fréquence par effet Doppler. Un paramètre d'intérêt pour le choix du type de modulation est celui du temps de cohérence T_c, fonction d'effets liés à l'environnement et à l'étalement fréquentiel dû à l'effet Doppler. Une expression simple de T_C pour la propagation en espace libre est $T_C = 1/\Delta f$, avec Δf le décalage maximal dû à l'effet Doppler. Cependant, d'autres expressions existent pour le temps de cohérence afin de prendre en compte d'autres effets liés à l'environnement. Dans le cas où le temps alloué à un symbole dans la transmission est plus faible que le temps de cohérence, on parle alors de canal lentement variable, et rapidement variable dans le cas contraire. Dans le second cas, il est nécessaire d'utiliser diverses méthodes pour pallier une variation rapide du canal de communication, susceptible de perturber la communication. Intuitivement, ceci signifie que les caractéristiques du canal de communication varient suffisamment vite pour qu'un bit se substitue à un autre dans la transmission du fait de la variation rapide du gain du canal durant sa transmission. Cette description est bien entendu simpliste, car elle ne tient pas compte de la modulation et du codage du signal, mais suffit pour comprendre le problème posé par un tel canal.

1.4 Modélisation du canal de communication

Dans cette section, nous présentons brièvement les outils de modélisation classiques utilisés pour les modèles de canaux de communication. Ces outils sont avant tout destinés à s'appliquer à des situations courantes, d'où la question de l'adéquation de ces méthodes au domaine aéronautique.

Un modèle courant est celui de la propagation en intérieur (bâtiment, *etc.*) avec pertes qui permet de prédire l'atténuation sur un chemin en fonction de coefficients fournis dans des tables. Ce modèle est celui de l'Union Internationale des Télécommunications (ITU), publié dans la recommandation ITU-R P.1238-7 (2012). L'équation donnant la perte en dB d'un chemin est :

$$L_{total} = 20\log_{10}(f) + N\log_{10}(d) + L_f(n) - 28\text{dB} \tag{1.4.1}$$

avec f la fréquence, N un coefficient de perte de puissance dépendant de la distance (20 pour l'espace libre), d la distance, $L_f(n)$ un coefficient de perte de pénétration, et n le nombre d'étages entre l'émetteur et le récepteur. Ces différents coefficients sont donnés en fonction du type de structure (béton, béton + métal, métal + verre, *etc.*), de construction (résidentiel, commercial, bureaux), et de la longueur d'onde (GSM, Wifi). Ce modèle est inadapté aux applications aéronautiques, ce que montrent les mesures relativement éloignées des résultats de ce modèle (ce qui est aisément compréhensible du fait de sa simplicité).

Dans le cas de la bande de fréquence autour de 60GHz, un exposé de cette méthode de modélisation, ainsi que des résultats expérimentaux se trouvent dans [91]. Dans cet article, le modèle employé est relativement similaire à (1.4.1), et est donné par :

$$L(d) = L(d_0) + 10n\log_{10}\left(\frac{d}{d_0}\right) + X_\Omega \quad [\text{dB}] \tag{1.4.2}$$

avec d_0 une distance de référence, n l'exposant de perte de chemin, et X_Ω la composante d'ombrage qui est supposée, dans ce cadre, être une variable gaussienne d'écart-type Ω et de moyenne nulle.

La perte est ensuite mesurée de la façon suivante :

$$L(d) = P_T - \bar{P}_R(d) + G_T + G_R \tag{1.4.3}$$

avec P_T la puissance transmise, et G_T et G_R les gains respectifs des antennes d'émission et de réception. Dans cette expression, $\bar{P}_R(d)$ est la moyenne sur quelques longueurs d'onde de la puissance reçue par l'antenne. Cette moyenne est nécessaire afin d'effacer les variations à petite échelle, puisqu'une mesure ponctuelle n'est pas pertinente du fait de ces variations.

La caractérisation des effets à petite échelle est purement statistique, et modélisée comme une somme de chemins, donnant alors des arrivées par "paquets", de type "*Multi-cluster Time-of-Arrival Model*" [92], dont le modèle de réponse impulsionnelle est :

$$h(\tau) = \sum_{l=0}^{L-1}\sum_{k=0}^{K_l-1} \beta_{k,l} e^{j\theta_{k,l}} \delta(\tau - T_l - \tau_{k,l}) \tag{1.4.4}$$

avec L le nombre de paquets, K_l le nombre de rayons dans le paquet l, $\beta_{k,l}$ et $\theta_{k,l}$ les amplitudes et phases du $k-$ième rayon du $l-$ième paquet, T_l le temps d'arrivée du premier rayon du $l-$ième paquet, et $\tau_{k,l}$ le délai d'arrivée du $k-$ième rayon de ce même paquet. Selon les modèles, la loi des amplitudes des rayons peut suivre une loi exponentielle, de Rayleigh, ou log-normale. Par ailleurs, il y a de nombreux paramètres à ajuster, et la dispersion dans les mesures est grande, ce qui montre les limites de cette modélisation.

	$10^2 < Q < 10^3$	$Q > 10^3$
$L \simeq 10^2\lambda$	-	Baies électroniques, satellites
$L \simeq 10^3\lambda$	Cabine A320 à 800MHz	CRBM
$L \simeq 10^4\lambda$	Cabine A380 à 60GHz	-

Table 1.1 – Classification schématique des cavités

1.4.1 Facteur de qualité

La facteur de qualité d'une structure donne l'enveloppe de la décroissance exponentielle de l'énergie dans une cavité. Il s'agit de l'analogue qualitatif du temps de réverbération, ou du facteur d'atténuation entre deux échos en acoustique. Il est défini par :

$$Q = \frac{\omega E_{totale}}{P_{diss}} \tag{1.4.5}$$

avec ω la pulsation, E_{totale} l'énergie contenue dans la cavité, et P_{diss} la puissance dissipée. Une telle définition nous amène donc naturellement à une équation différentielle linéaire du premier ordre, ce qui donne comme solution une décroissance exponentielle de la forme :

$$E(t) = E_0 \exp\left(-\frac{2\pi f t}{Q}\right) \tag{1.4.6}$$

Il est alors possible de faire une classification des types de cavités rencontrées en aéronautique en fonction de leur dimension en longueur d'onde et de leur facteur de qualité, comme celle du tableau 1.1.

1.5 Position du problème

À la lumière de la description qui précède, nous précisons ici le problème industriel dont ce document fait l'objet. Nous nous intéressons à la caractérisation de l'environnement électromagnétique dans une cavité de grande dimension devant la longueur d'onde, et partiellement réfléchissante.

Nous cherchons à établir une modélisation pertinente d'une cavité de grande dimension, de facteur de qualité élevé, et à en fournir une solution numérique efficace. Plus précisément, nous considérons une surface fermée, non nécessairement convexe, possédant des parois de conductivité élevée mais non nécessairement infinie, telle que la taille caractéristique du système est grande devant la longueur d'onde, soit $l\kappa \gg 1$, avec κ le nombre d'onde et l la taille caractéristique du système. L'observable retenue est la densité locale d'énergie dans le volume de la cavité, éventuellement moyennée d'une manière qui reste à préciser. Cette quantité (et sa moyenne) peut être obtenue par post-traitement d'un résultat portant sur les champs.

L'observable privilégiée est la densité locale d'énergie. En effet, les champs ne sont pas pertinents à cette échelle du fait des oscillations rapides de la phase en un point donné. Il en est de même de l'énergie, c'est pour cela que nous considérons dans la suite comme observable l'énergie moyennée en espace et/ou temps.

Le problème est donc celui de la résolution des équations de Maxwell dans les conditions présentées ici. Il existe un grand nombre de méthodes numériques ayant pour objectif la résolution de ces équations. La section suivante en présente un bref panorama.

1.6 Brève présentation des méthodes de simulation classiques

Dans cette section, nous ferons un succinct tour d'horizon des principales méthodes de simulation pour les problèmes de propagation des ondes, dans l'optique d'évaluer leur applicabilité au problème défini précédemment. Nous aborderons donc en particulier les méthodes pour lesquelles des codes de calcul existent et sont établis dans l'industrie, et en particulier dans l'industrie aéronautique. Un tel exemple est celui de la suite de logiciels Aseris, développée par EADS-IW, et dont les composantes sont utilisées au sein des entités du groupe EADS. Cette suite comporte des codes implémentant la méthode des différences finies (FDTD), celle des éléments finis de frontière (*Boundary Element Method* (BEM)), et celle de la théorie géométrique de la diffraction (*Uniform Theory of Diffraction* (UTD)).

1.6.1 Différences finies en domaine temporel

La méthode des différences finies est de façon générale une méthode de discrétisation des équations aux dérivées partielles reposant sur l'approximation discrète des opérateurs de l'équation, opérant sur une grille régulière. Cette présentation succincte est inspirée de celle trouvée dans [95], dans lequel figure un exposé beaucoup plus exhaustif, en particulier sur la stabilité et le caractère dissipatif du schéma.

1.6.1.1 Schéma de Yee

Elle repose fondamentalement sur la discrétisation du développement limité de chaque opérateur de dérivation de l'équation. Il s'agit donc d'une méthode plus directe que les méthodes de type éléments finis, et plus simple à mettre en œuvre. Dans le cas des équations de Maxwell dans le vide, le problème à résoudre dans le domaine temporel est, en fonction des champs $\boldsymbol{E}$ et $\boldsymbol{H}$:

$$\begin{cases} \boldsymbol{\nabla} \wedge \boldsymbol{E}(x,t) + \mu_0 \frac{\partial \boldsymbol{H}}{\partial t}(x,t) = 0 & x \in \mathbb{R}^3, t \in \mathbb{R}_+^* \\ \boldsymbol{\nabla} \wedge \boldsymbol{H}(x,t) - \varepsilon_0 \frac{\partial \boldsymbol{E}}{\partial t}(x,t) = 0 & x \in \mathbb{R}^3, t \in \mathbb{R}_+^* \\ \boldsymbol{E}(x,0) = \boldsymbol{E}_0 & x \in \mathbb{R}^3 \\ \boldsymbol{H}(x,0) = \boldsymbol{H}_0 & x \in \mathbb{R}^3 \end{cases} \tag{1.6.1}$$

avec les données initiales dans $L^2(\mathbb{R}^3)$, μ et ε les permittivité et perméabilité du milieu, exprimées en fonction de celles du vide (μ_0, ε_0).

Ce système d'équations devient, en introduisant $Z_0 = \sqrt{\frac{\mu_0}{\varepsilon_0}}$ l'impédance du vide, et avec $c = (\varepsilon_0 \mu_0)^{-1/2}$ la vitesse de la lumière dans le vide, et après le changement de variable

$\boldsymbol{H} \to Z_0\boldsymbol{H}$:

$$\begin{cases} \frac{1}{c}\frac{\partial \boldsymbol{H}}{\partial t}(x,t) + \boldsymbol{\nabla} \wedge \boldsymbol{E}(x,t) = 0 & x \in \mathbb{R}^3, t \in \mathbb{R}_+^* \\ \frac{1}{c}\frac{\partial \boldsymbol{E}}{\partial t}(x,t) - \boldsymbol{\nabla} \wedge \boldsymbol{H}(x,t) = 0 & x \in \mathbb{R}^3, t \in \mathbb{R}_+^* \\ \boldsymbol{E}(x,0) = \boldsymbol{E}_0 & x \in \mathbb{R}^3 \\ \boldsymbol{H}(x,0) = \boldsymbol{H}_0 & x \in \mathbb{R}^3 \end{cases} \tag{1.6.2}$$

Cette équation est discrétisée et résolue selon le schéma de Yee, qui repose sur deux grilles, décalées d'une demi-maille l'une par rapport à l'autre en espace et en temps. En notant les indices spatiaux en indice et les indices temporels en exposant, avec h le pas d'espace et n l'étape en temps (donc aux coordonnées $h \times (i,j,k)$ en espace et $n\Delta t$ en temps),

$$\boldsymbol{E}_h^n = \begin{pmatrix} E^n_{x,i+\frac{1}{2},j,k} \\ E^n_{y,i,j+\frac{1}{2},k} \\ E^n_{z,i,j,k+\frac{1}{2}} \end{pmatrix} \quad \text{et} \quad \boldsymbol{H}_h^{n+\frac{1}{2}} = \begin{pmatrix} H^{n+\frac{1}{2}}_{x,i,j+\frac{1}{2},k+\frac{1}{2}} \\ H^{n+\frac{1}{2}}_{y,i+\frac{1}{2},j,k+\frac{1}{2}} \\ H^{n+\frac{1}{2}}_{z,i+\frac{1}{2},j+\frac{1}{2},k} \end{pmatrix} \tag{1.6.3}$$

L'opérateur $\overrightarrow{rot}$ est alors approché classiquement sur ces grilles pour $\boldsymbol{E}$ et $\boldsymbol{H}$, et en posant respectivement $\boldsymbol{B}_h$ et $\boldsymbol{C}_h$ les opérateurs vectoriels approchant $-\overrightarrow{rot}$ sur $\boldsymbol{H}_h$ et $\overrightarrow{rot}$ sur $\boldsymbol{E}_h$, ce schéma s'écrit :

$$\begin{cases} \dfrac{1}{c}\dfrac{\boldsymbol{H}_h^{n+\frac{1}{2}} - \boldsymbol{H}_h^{n-\frac{1}{2}}}{\Delta t} + (\boldsymbol{C}_h \boldsymbol{E}_h^n)_h = 0 & n \geqslant 1 \\ \dfrac{1}{c}\dfrac{\boldsymbol{E}_h^{n} - \boldsymbol{E}_h^{n-1}}{\Delta t} + (\boldsymbol{B}_h \boldsymbol{H}_h^{n-\frac{1}{2}})_h = 0 & n \geqslant 1 \end{cases} \tag{1.6.4}$$

Ce schéma est stable à condition de respecter la condition CFL suivante :

$$c\Delta t \leqslant \frac{1}{\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}}} \tag{1.6.5}$$

Par ailleurs, ce schéma est d'ordre 2, dispersif et anisotrope, mais la dispersion est plus faible dans le cas limite de CFL maximum, avec le nombre CFL $\alpha = \frac{c\Delta t}{h}$, c'est-à-dire avec le pas de temps le plus large possible respectant la condition (1.6.5).

Application aux cavités haute fréquence

Une approche de type différence finie est attrayante par sa simplicité de mise en œuvre, la maturité des solveurs possédant des modèles riches à même de traiter une grande variété de situations, et l'information temporelle donnée par le solveur. La méthode de résolution présentée précédemment possède de nombreuses extensions permettant de prendre en compte des modèles équivalents de fils, matériaux composites, fentes, *etc.* De plus, les calculs sont peu consommateurs en mémoire et aisément parallélisables, ce qui permet de traiter des problèmes de grande taille.

En revanche, le nombre de mailles augmente proportionnellement au cube de la fréquence, et le pas de temps de la méthode diminue avec l'inverse de la fréquence, et donc le temps de calcul augmente avec la puissance 4 de la fréquence, ce qui pose le problème

des hautes fréquences. Par ailleurs, dans le cas d'un problème de cavité réfléchissante, le temps avant extinction est lié au facteur de qualité par (1.7.8), et celui-ci étant élevé, le temps de simulation est grand, ce qui pose le problème du temps de calcul, mais également de la précision du résultat, du fait du caractère dispersif du schéma.

Par exemple, pour une fréquence de 10GHz et un facteur de qualité de 10^3, la décroissance exponentielle prévue par (1.7.8) a un temps caractéristique $\tau = \frac{Q}{2\pi f} \simeq 10^{-8}s$. En prenant une discrétisation ayant 10 points par longueur d'onde, on obtient $h = 3 \times 10^{-3}m$, et donc pour respecter la condition CFL, il faut $\Delta t \leqslant \frac{h}{c\sqrt{3}} \simeq 10^{-11}$, soit au minimum 10^3 pas de temps, bien qu'en pratique le nombre de pas de temps nécessaire soit bien plus grand, puisque cette grandeur ne caractérise que la décroissance exponentielle finale de la solution. Pour une cavité cubique de dimension, $3 \times 3 \times 3$m, il faut alors 125×10^6 mailles.

1.6.2 Éléments finis de frontière

La méthode des éléments finis de frontière est une méthode de résolution des équations de Maxwell de type Galerkin, basée sur une formulation variationnelle de l'opérateur des ondes, et que nous présenterons ici dans le domaine fréquentiel pour un conducteur parfait, en s'inspirant de la présentation qui en est faite dans [94]. Par ailleurs, une présentation plus complète et rigoureuse se trouve dans le chapitre suivant de ce document, et dans [80, 95].

Physiquement, le calcul de l'interaction d'un champ incident avec un objet borné se fait par le calcul des courants surfaciques électriques et magnétiques. La connaissance de ceux-ci permet ensuite le calcul du champ diffracté (et du champ total) en tout point de l'espace, par intégration sur la surface de l'objet.

On considère un objet borné Ω de frontière $\partial\Omega$, illuminé par une onde $(\boldsymbol{H}_{inc}, \boldsymbol{E}_{inc})$ de pulsation ω, et on s'intéresse au calcul de $\boldsymbol{H}_{diff} = \boldsymbol{H} - \boldsymbol{H}_{inc}$, le champ diffracté (respectivement $\boldsymbol{E}_{diff}$).

Dans ce cas, le problème à résoudre est, avec Ω un objet borné parfaitement conducteur de frontière $\partial\Omega$, de normale extérieure $\boldsymbol{n}$, et en choisissant une dépendance en temps en $e^{-i\omega t}$:

$$\begin{cases} \boldsymbol{\nabla} \wedge \boldsymbol{E} - i\omega\mu\boldsymbol{H} = 0 & \text{dans } \mathbb{R}^3 - \Omega \\ \boldsymbol{\nabla} \wedge \boldsymbol{H} + i\omega\varepsilon\boldsymbol{E} = 0 & \text{dans } \mathbb{R}^3 - \Omega \\ \boldsymbol{E} \wedge \boldsymbol{n} = -\boldsymbol{E}_{inc} \wedge \boldsymbol{n} & \text{sur } \partial\Omega \\ \lim\limits_{r \to +\infty} r \left\| \sqrt{\varepsilon}\boldsymbol{E} - \sqrt{\mu}\boldsymbol{H} \wedge \frac{\boldsymbol{r}}{r} \right\| = 0 & \end{cases} \tag{1.6.6}$$

La troisième équation est celle décrivant la condition limite de conducteur parfait, et la dernière est la condition de radiation, permettant de ne conserver que la solution en onde sortante. En effet, sans cette condition, le problème admet deux solutions symétriques en temps, dont une seule est causale ; cette dernière propriété permet donc de ne sélectionner que celle-ci. On définit les courants surfaciques $\boldsymbol{j}$ et $\boldsymbol{m}$ comme les traces tangentielles des champs sur la frontière de l'objet :

$$\begin{cases} \boldsymbol{j} = \boldsymbol{n} \wedge \boldsymbol{H} \\ \boldsymbol{m} = \boldsymbol{n} \wedge \boldsymbol{E} \end{cases} \tag{1.6.7}$$

Dans le cas d'un conducteur parfait, $\boldsymbol{m}$ est nul, et par le théorème de représentation intégrale, on peut exprimer le champ en tout point de l'espace en fonction de $\boldsymbol{j}$, ce qui permet donc de changer d'inconnue dans le problème. On a donc le théorème de représentation intégrale :

$$\begin{cases} \boldsymbol{E}(x) = i\omega\mu \int_{\partial\Omega} G(|x-y|)\boldsymbol{j}(y)dy + \frac{i}{\omega\varepsilon}\boldsymbol{\nabla}_x \int_{\partial\Omega} G(|x-y|)\boldsymbol{\nabla}_{\partial\Omega}\cdot\boldsymbol{j}(y)dy & x \in \mathbb{R}^3 - \Omega \\ \boldsymbol{H}(x) = -\boldsymbol{\nabla}_x \wedge \int_{\partial\Omega} G(|x-y|)\boldsymbol{j}(y)dy & x \in \mathbb{R}^3 - \Omega \end{cases} \tag{1.6.8}$$

avec G le noyau de Green, solution élémentaire de l'équation de Helmholtz en 3D, défini par

$$G(R) = \frac{e^{ikR}}{4\pi R} \tag{1.6.9}$$

Le théorème de représentation intégrale permet de substituer aux inconnues volumiques classiques (champs $\boldsymbol{E}$ et $\boldsymbol{H}$) les inconnues surfaciques correspondants aux courants $\boldsymbol{j}$ et $\boldsymbol{m}$. Ceci permet alors d'écrire le problème sous la forme d'une équation de Fredholm, dont la résolution est équivalente après discrétisation par éléments finis à une résolution de système linéaire complexe, la solution de ce système étant les courants exprimés dans la base d'éléments finis retenue.

On trouve en effet la formulation variationnelle suivante, avec $\boldsymbol{j}_t$ une fonction test :

$$\int_{\partial\Omega\times\partial\Omega} G(|x-y|)\left(\boldsymbol{j}(y)\cdot\boldsymbol{j}_t(x) - \frac{1}{k^2}\boldsymbol{\nabla}_{\partial\Omega}\cdot\boldsymbol{j}(y)\boldsymbol{\nabla}_{\partial\Omega}\cdot\boldsymbol{j}_t(x)\right)dxdy = \frac{i}{kZ_0}\int_{\partial\Omega}\boldsymbol{E}_{inc}(y)\cdot\boldsymbol{j}_t(y)dy \tag{1.6.10}$$

dont la discrétisation sous forme matricielle est classique, et relativement transparente pour le lecteur familier des éléments finis.

La connaissance du champ en un point de l'espace est alors un calcul d'intégrale sur la surface de l'objet, assimilable à un produit matrice-vecteur dans le domaine discret. Le temps de calcul est donc dépendant de la taille de l'objet en fonction de la longueur d'onde, du fait de la nécessité d'avoir un maillage dont les triangles sont suffisamment petits pour représenter les oscillations des courants surfaciques (le critère usuel de maillage est $\lambda/10$), et du nombre de points auxquels le champ doit être calculé.

Cette méthode de résolution mène donc à un système matriciel à coefficients complexes, de matrice pleine, et ici symétrique. Le nombre d'inconnues dépend de la finesse de la discrétisation, qui est liée à la fréquence. En revanche, il est uniquement nécessaire de mailler la surface de l'objet, le nombre d'inconnues de la résolution est donc indépendant de la taille du domaine dans lequel les valeurs des champs doivent être calculées.

Application aux cavités haute fréquence

Pour un objet de surface S, le nombre d'inconnues est donné approximativement par $N \simeq 150\frac{S}{\lambda^2}$ [95, chapitre 1]. Selon la méthode de résolution retenue, on a une croissance du temps de résolution en $O(N^3)$ pour un solveur direct, N^2 pour un solveur itératif, ou

$O(N \log N)$ pour une résolution itérative dont le produit matrice-vecteur est accéléré par la *Fast Multipole Method* (FMM), ce qui donne un temps de calcul en $O(f^6)$, $O(f^4)$ et $O(f^2 \log f)$ respectivement. De plus, à l'exception d'un solveur FMM, l'espace mémoire requis croît en $\mathcal{O}(N^2)$ ($\mathcal{O}(f^4)$). La résolution FMM est en pratique indispensable pour traiter des problèmes de taille suffisante. En effet, en prenant le même exemple que dans la section précédente, soit un objet de 54m^2 de surface, et pour $\lambda = 3 \times 10^{-2}$m, on a $N \simeq 9 \times 10^5$, ce qui est proche de la limite des cas dont la résolution directe est possible.

Pour des cas de plus grande taille, il est alors nécessaire d'utiliser une résolution accélérée par la FMM, et si la surface est de l'ordre de $10^4\lambda$, le nombre d'inconnues devient prohibitif, même pour cette méthode. Par ailleurs, le caractère réfléchissant de la structure gêne la convergence des solveurs itératifs, qui parfois ne convergent pas en un nombre raisonnable d'itérations pour des cas réfléchissants, ce problème étant aggravé par la moindre précision du « produit » matrice-vecteur FMM.

Cette méthode, bien que précise et déployée industriellement, se heurte au problème de la résolution numérique des équations, puisque la meilleure méthode disponible (FMM) n'est pas adaptée au problème, et que les autres méthodes ont un coût prohibitif.

1.6.3 Théorie géométrique de la diffraction

La théorie géométrique de la diffraction [30,66] est une méthode haute fréquence basée sur un développement asymptotique de la solution de l'équation des ondes quand le nombre d'onde κ tend vers $+\infty$, que nous présenterons dans le cas d'une onde scalaire afin de simplifier l'exposé.

On cherche les solutions de l'équation des ondes dans un milieu homogène :

$$\begin{cases} \frac{\partial^2 U}{\partial t^2} - c^2 \Delta U = 0 & \text{dans } \mathbb{R}^3 \times \mathbb{R}_+^* \\ U_0(x,0) = u_0(x) & \text{dans } \mathbb{R}^3 \end{cases} \tag{1.6.11}$$

La présentation de cette section est inspirée de [81] et [20]. On remarque que les ondes planes sont des solutions de l'équation des ondes et on prend alors, comme ansatz de la solution, la forme suivante :

$$U(x,t,k) = \mathfrak{Re}\left(A(x,k)e^{i(\omega t + k\phi(x))}\right) \tag{1.6.12}$$

avec une convention temporelle en $+i\omega t$, A une amplitude complexe, et ϕ la phase. Formellement, il s'agit de décomposer un champ U solution de l'équation des ondes en série WKB (Wentzel-Kramers-Brillouin). On suppose que l'amplitude admet un développement asymptotique de la forme :

$$A(x,k) \sim \sum_{j=0}^{+\infty} \frac{1}{(ik)^j} A_j(x) \tag{1.6.13}$$

Remarque 1.1. *Cette somme s'entendu au sens d'une somme asymptotique et ne converge pas nécessairement.*

On substitue l'expression donnée par l'ansatz (1.6.12) dans (1.6.11). On obtient alors une suite d'équations, la première portant sur la phase, et les autres étant des équations

de transport inhomogènes identiques, dont la j-ième équation a un terme source calculé à partir de la solution de la $j-1$-ième équation.

L'équation sur la phase est appelée « équation eikonale » :

$$|\boldsymbol{\nabla}\psi|^2 = 1 \tag{1.6.14}$$

et le terme d'ordre 0 est celui de l'optique géométrique, solution de :

$$\boldsymbol{\nabla} \cdot \left(A_0^2 \boldsymbol{\nabla}\psi\right) = 0 \tag{1.6.15}$$

On a donc l'approximation suivante de la solution à l'ordre 0, dite de l'optique géométrique :

$$U(x,t,k) \sim_{k\to+\infty} \mathfrak{Re}\left(A_0(x)e^{i(-\omega t+k\phi(x))}\right) + o(1) \tag{1.6.16}$$

Les équations de l'optique géométrique peuvent être résolues par la méthode des caractéristiques. Celles-ci, que l'on appelle les rayons, sont les courbes intégrales du gradient de la phase. L'équation de transport traduit ici la conservation de l'énergie dans un tube de rayons.

La réflexion, transmission et diffraction des rayons sur une surface sont dépendantes des conditions et de la géométrie, et suivent la loi de Snell-Descartes pour la réflection et la transmission des rayons. Les amplitudes et les phases sont calculées en fonction des conditions limites, en tenant compte de la géométrie pour le calcul de l'intensité.

Cependant, la prise en compte du terme d'optique géométrique uniquement devient problématique dans le cas des caustiques et prédit de plus un champ nul dans les zones d'ombre. Intuitivement, une caustique peut correspondre à la tangence d'une infinité de rayons. Dans ce cas, l'amplitude présente une singularité, et la phase est multivaluée (la solution contient des contributions de toutes les valeurs possibles de la phase).

Le problème du champ nul dans les zones d'ombre vient du manque de prise en compte de la diffraction dans cette théorie. Cette difficulté a été levée dans [30] en prenant en compte les termes d'ordre supérieur dans la série (1.6.13), chaque terme correspondant à un type de phénomène physique, sous réserve de considérer une série généralisée avec des indices fractionnaires. Le terme d'ordre 0 est celui de l'optique géométrique, l'ordre 1/2 correspond à la diffraction-réflexion, le terme d'ordre 1 à la double diffraction ou à la diffraction par une pointe, et le terme $k^{1/6}e^{-ck^{1/3}}$ aux rayons rampants. La théorie rigoureuse fait appel à des outils généraux d'analyse et d'analyse microlocale.

Les corrections par rapport à l'optique géométrique sont calculées par la théorie uniforme de la diffraction, qui est une extension heuristique de la théorie géométrique de la diffraction de Keller. Dans les deux cas, les diffractions sont calculées sur des problèmes modèles, et appliquées au cas en s'appuyant sur les calculs analytiques (diffraction par un point, diffraction par une arête infinie, un demi-plan, *etc.*). Un ouvrage de référence sur ces méthodes et leurs fondements est [30]. La démonstration rigoureuse de certains résultats de cet ouvrage a été faite ; beaucoup reste à faire. Par exemple, la prise en compte de la géométrie pour la réflexion ne prend pas en compte toutes les caractéristiques de la surface dans l'UTD.

Calcul de la solution

Dans la pratique, le calcul d'une solution par la théorie uniforme de la diffraction se fait par lancer et recherche de rayons. Une définition des points sources et des observables est faite, et le code de calcul recherche ensuite les rayons reliant ces deux ensembles de points, dans un ensemble de familles de chemins prédéfinies par l'utilisateur. En particulier, le nombre de réflexions est borné, ainsi que le nombre de diffractions. Ainsi, le résultat final n'est jamais la somme de toutes les contributions, mais de toutes les contributions dans une famille prédéfinie, que l'on espère suffisamment large pour prendre en compte fidèlement la réalité. En particulier, il n'existe pas de garantie d'avoir les termes prépondérants retenus pour un ordre d'approximation donné à une fréquence fixée.

Ceci est raisonnable dans la plupart des cas, dans la mesure où les contributions décroissent très vite en intensité avec la complexification du chemin. En revanche, une telle méthode de recherche de chemins a une complexité algorithmique exponentielle dont la croissance dépend de la profondeur maximale des chemins recherchés. Une autre limitation inhérente à cette méthode concerne la précision du résultat dans le cas de réflexions multiples. En effet, des erreurs (théoriques et numériques) sont introduites à chaque réflexion. Elles peuvent croître de manière importante au cours de réflexions successives, et ce même si le code de calcul prend en compte les définitions analytiques des surfaces provenant d'un fichier CAO, l'exemple le plus défavorable étant celui d'un billard chaotique.

Le temps de calcul est donc indépendant de la fréquence, bien que celle-ci soit prise en compte dans le calcul du champ, ce qui est un grand avantage de cette méthode. Il dépend en revanche linéairement du nombre de récepteurs et du nombre de sources, et de façon exponentielle de la profondeur des chemins considérés.

Application aux cavités haute fréquence

Le caractère asymptotique en fréquence distingue cette méthode des précédentes. En revanche, cette méthode a le désavantage de ne donner qu'une vision partielle du champ, qui n'est pas nécessairement pertinente. En particulier, la nécessité de connaître le champ en un grand nombre de points de l'espace afin de calculer les moyennes adaptées est une difficulté. De plus, l'environnement réfléchissant nécessite de prendre en compte un grand nombre de réflexions, ce qui est très pénalisant pour une telle méthode.

Ce n'est donc pas l'aspect haute fréquence qui est bloquant pour cette méthode, mais la particularité des conditions aux limites, et de l'observable. Ces difficultés sont sérieuses, bien que ne posant que des problèmes de résolution numérique.

1.7 Méthodes énergétiques

Par opposition aux méthodes précédentes reposant sur la résolution des équations de Maxwell et le calcul des champs, un ensemble de méthodes basées sur le calcul direct de l'énergie existe. Ces approches reposent sur des lois phénoménologiques ou des analogies avec d'autres physiques.

1.7.1 Statistical Energy Analysis (SEA)

Il s'agit d'une méthode très globale, caractérisant les échanges d'énergie entre sous-systèmes, le découpage des sous-systèmes et des coefficients d'échange étant à la charge de l'ingénieur réalisant l'étude [28,36,75]. Cette méthode est utilisée depuis les années 70 [76] en vibro-acoustique principalement, et a été employée avec succès de façon industrielle. Elle présente cependant des inconvénients substantiels, que nous citons brièvement. Elle ne donne pas de description spatiale du champ à l'intérieur du sous-système mais une seule valeur globale par sous-système, les hypothèses sont très fortes et rarement vérifiées au sens fort [28], le découpage en sous-systèmes est arbitraire et source d'imprécisions, et les coefficients à fixer ne se rapportent pas de façon claire aux propriétés physiques de la cavité. Ces inconvénients limitent le caractère prédictif de la méthode, et la détermination des paramètres est souvent délicate.

On considère un système Σ en régime permanent, et une partition de ce système en n sous-parties $\Sigma_1, \ldots, \Sigma_n$. Pour chaque sous-système, un bilan d'énergie donne l'équation :

$$\forall i \in \{1, n\},\ P_i^{diss} + \sum_{j=1}^{n} P_{i\to j} = P_i^{inj} \tag{1.7.1}$$

avec $P_i^{inj,diss}$ la puissance injectée dans et dissipée par le système i, et $P_{i\to j}$ la puissance transférée depuis le sous-système i vers le sous-système j. La puissance injectée est une donnée du problème, et provient typiquement des émissions des équipements contenus dans chaque sous-système.

Les données du problème sont :

- La structuration en sous-systèmes ;
- Les puissances injectées par sous-système P_i^{inj} ;
- Le coefficient de dissipation par sous-système η_i ;
- Le nombre de modes par sous-système n_i ;
- Les coefficients de couplage η_{ij}.

et les inconnues du problème sont les énergies par sous-système E_i.

On suppose que la puissance dissipée peut s'écrire sous la forme :

$$P_i^{diss} = \omega \eta_i E_i \tag{1.7.2}$$

avec $\omega = 2\pi f$ la pulsation associée à la fréquence centrale de la bande de fréquence considérée, et E_i l'énergie électromagnétique contenue dans le système. De même, on suppose que la puissance transmise du système i vers le système j peut s'écrire sous la forme :

$$P_{i\to j} = \omega \eta_{ij} n_i \left(\frac{E_i}{n_i} - \frac{E_j}{n_j} \right) \tag{1.7.3}$$

avec η_{ij} un coefficient de couplage entre les deux systèmes, et n_i le nombre de modes dans le système i. On a bien entendu la relation de réciprocité $P_{i\to j} = -P_{j\to i}$, ce qui se traduit sur les coefficients de couplage par :

$$\eta_{ij} n_i = \eta_{ji} n_j$$

En remplaçant les équations (1.7.2) et (1.7.3) dans le système d'équations (1.7.1), on trouve un système linéaire de n équations à n inconnues :

$$\forall i \in \{1, n\},\ \omega \eta_i E_i + \omega \sum_{j=1}^{n} \eta_{ij} n_i \left(\frac{E_i}{n_i} - \frac{E_j}{n_j} \right) = \omega \eta_i E_i \tag{1.7.4}$$

La matrice de ce système linéaire est définie positive, à diagonale dominante, et le système admet une unique solution.

Application aux cavités haute fréquence

La *Statistical Energy Analysis* (SEA) est une méthode dont le champ d'application est celui des systèmes soumis à des excitations haute fréquence large bande, ce qui correspond aux applications visées ici. Cependant, elle présente de nombreux inconvénients. Ainsi, le très grand nombre de paramètres à fixer pose des problèmes du point de vue du caractère prédictif de la méthode. En premier lieu, la décomposition de la cavité en sous-systèmes faiblement couplés (le caractère faiblement couplé des systèmes est une hypothèse de la SEA) est problématique. Outre l'arbitraire de la détermination de ces systèmes, dans le cas d'une cavité relativement monolithique, il n'est sans doute pas possible de faire une décomposition pertinente. Ensuite, l'application de la SEA présuppose un champ diffus à l'intérieur de chaque sous-système, ce qui n'est pas toujours vérifié en pratique. En résumé, l'absence de prédiction spatiale et le faible caractère prédictif de la méthode la rendent inadaptée pour la modélisation recherchée.

1.7.2 Radiosité

Issue de la thermique et de la réalité virtuelle, c'est essentiellement une équation intégrale dont les inconnues vivent sur la frontière du domaine. Il existe plusieurs façons de traiter cette équation, par méthodes intégrales [79,84] ou Monte-Carlo [48,68,88], et celles-ci diffèrent dans leur champ d'application, en termes de géométrie, de type de conditions de réflexion et de dimension de problèmes qu'il est possible de traiter. Cette méthode a été appliquée à la vibro-acoustique, à l'acoustique des salles [8, 27, 27, 29], et à la propagation sans fil haute fréquence [88]. Nous allons décrire l'origine de cette méthode.

On suppose un milieu de propagation linéaire, isotrope, homogène en régime permanent. Ce système est soumis à une excitation de large bande centrée sur la pulsation ω. On suppose de plus que l'amortissement lors de la propagation est faible, on néglige le champ proche et les ondes évanescentes, et on néglige les interférences entre les ondes. Ce dernier point est essentiel, car il est nécessaire pour superposer linéairement les énergies dans la suite. La présentation de cette section est l'adaptation à l'électromagnétisme de l'application de l'équation de la radiosité à la vibro-acoustique [27].

On pose comme inconnues la densité d'énergie électromagnétique et son intensité :

$$\begin{cases} W = \frac{\mu}{2} \|\boldsymbol{H}\|^2 + \frac{\varepsilon}{2} \|\boldsymbol{E}\|^2 \\ \boldsymbol{I} = \boldsymbol{E} \wedge \boldsymbol{H} \end{cases} \tag{1.7.5}$$

On écrit ensuite la conservation de l'énergie :

$$\partial_t W + \nabla \cdot \boldsymbol{I} + P_{diss} = P_{inj} \tag{1.7.6}$$

avec P_{diss} et P_{inj} les puissances dissipée et injectée respectivement, et en régime stationnaire $\partial_t W = 0$.

Pour pouvoir fermer le système d'équations, il faut une relation supplémentaire entre W et $\boldsymbol{I}$, ainsi qu'une modélisation pour le terme de dissipation dans le volume. Il peut être pris nul dans le cas de la propagation dans le vide, ou alors sous une des deux formes suivantes, suggérées dans le cadre de la vibro-acoustique :

— $P_{diss} = \eta\omega W$

— $P_{diss} = mcW$

avec η et m des coefficients de perte. La première loi provient de la SEA, d'où l'utilisation de la lettre η. Dans la suite nous prendrons le second modèle.

On considère ensuite un cas particulier de champ, celui généré par une source ponctuelle placée en $\boldsymbol{x}$. Dans ce cas le champ ne dépend que de la distance r à la source par symétrie, ce qui donne donc alors pour l'équation de conservation de l'énergie :

$$\frac{1}{r^{n-1}} \frac{d}{dr}(r^{n-1} I) + mcW = 0 \tag{1.7.7}$$

On ferme alors le système par la relation suivante, valable pour les ondes sphériques [27] :

$$\boldsymbol{I} = cW\boldsymbol{u}_r \tag{1.7.8}$$

ce qui permet de réécrire l'équation (1.7.6) sous la forme :

$$\frac{1}{r^{n-1}} \frac{d}{dr}(r^{n-1} W) + mW = 0 \tag{1.7.9}$$

Le système est alors complet, et on cherche ses fonctions de Green G et $\boldsymbol{H}$ correspondant à la solution pour W et $\boldsymbol{I}$ avec un second membre égal à $\delta_{\boldsymbol{x}}$. Ces fonctions sont données par :

$$\begin{cases} G(\boldsymbol{x}, \boldsymbol{y}) = \dfrac{1}{c\gamma_0 \|\boldsymbol{x} - \boldsymbol{y}\|^{n-1}} e^{-m\|\boldsymbol{x}-\boldsymbol{y}\|} \\ \boldsymbol{H}(\boldsymbol{x}, \boldsymbol{y}) = \dfrac{1}{c\gamma_0 \|\boldsymbol{x} - \boldsymbol{y}\|^{n-1}} e^{-m\|\boldsymbol{x}-\boldsymbol{y}\|} \boldsymbol{u}_{xy} \end{cases} \tag{1.7.10}$$

avec γ_0 la surface de la sphère unité en dimension n.

On obtient ensuite l'équation intégrale :

$$\begin{cases} W(\boldsymbol{x}) = \displaystyle\int_\Omega \rho(\boldsymbol{y}) G(\boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{y} + \int_\Gamma \sigma(\boldsymbol{y}) f(\boldsymbol{u}_{xy}, \boldsymbol{n}_x) G(\boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{y} \\ \boldsymbol{I}(\boldsymbol{x}) = \displaystyle\int_\Omega \rho(\boldsymbol{y}) \boldsymbol{H}(\boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{y} + \int_\Gamma \sigma(\boldsymbol{y}) f(\boldsymbol{u}_{xy}, \boldsymbol{n}_x) \boldsymbol{H}(\boldsymbol{x}, \boldsymbol{y}) d\boldsymbol{y} \end{cases} \tag{1.7.11}$$

Cette équation peut être résolue par diverses méthodes, les plus courantes étant une résolution par éléments finis de frontière, ou par une méthode de lancer de rayons. La première mène à la résolution d'un système matriciel plein, la seconde à une méthode de Monte-Carlo.

Application aux cavités haute fréquence

Cette méthode a pour avantage de donner une description spatiale de la densité d'énergie. Elle est par ailleurs appliquée avec succès en thermique (sous le nom de transfert radiatif) [39] et dans le domaine de la synthèse d'images réalistes (sous le nom d'illumination globale) [48, 52].

En revanche, il n'existe pas de démonstration rigoureuse de son lien avec les équations de Maxwell dans les configurations cibles de ce chapitre. De plus, cette méthode ne fait pas intervenir la fréquence dans les équations, ni le type d'onde, et la transcription des conditions aux limites usuelles est problématique. Néanmoins, des investigations numériques sur des cavités modèles mettent en lumière un rapprochement entre les résultats de cette méthode et du post-traitement d'une résolution numérique des équations de Maxwell. Une justification théorique de ce lien n'est toutefois pas disponible dans la littérature. Une étude est en cours.

1.8 Conclusion

La modélisation et le calcul de l'environnement électromagnétique dans une cavité est un problème complexe, couvrant de nombreux domaines. Une approche exhaustive nécessite une compréhension fine des observables pertinentes, des phénomènes physiques dominants, et surtout une capacité de résolution numérique de problèmes de diffraction d'ondes ardus.

Parmi les méthodes présentées dans les sections précédentes, la méthode que l'on retient ici pour le traitement de problèmes de cavités est celle des éléments finis de frontière. Elle présente l'avantage d'être mature, d'une grande précision, de ne pas reposer due la connaissance de coefficients difficilement mesurables (par exemple η_{ij} pour la SEA). Le principal inconvénient de cette méthode dans le contexte des cavités est lié au coût de calcul de la solution. La dimension des cavités interdit l'utilisation d'un solveur direct ou itératif classique, et l'accélération d'un solveur itératif par FMM est problématique du point de vue de la convergence. L'objet de ces travaux est de proposer un solveur rapide, ou un préconditionneur adapté pour la résolution itérative par FMM.

L'objectif de cette thèse est donc de proposer une méthode de résolution efficace pour la méthode des éléments finis de frontière. Une telle méthode de résolution doit avoir les caractéristiques suivantes :

Performances Les problèmes considérés sont de grande dimension en nombre d'inconnues. Une méthode ayant une complexité asymptotique meilleure que $\mathcal{O}(N^2)$ avec N le nombre d'inconnues est nécessaire. En effet, une méthode classique de résolution itérative ou directe ne peut être adaptée du fait de la croissance du nombre de degrés de liberté, même en utilisant une implémentation massivement parallèle.

Multiples seconds membres Les excitations étant mal connues au sein de la cavité, de nombreuses simulations sont nécessaires afin de caractériser son comportement. Pour une configuration de la cavité fixée, chaque excitation se traduit par un second membre supplémentaire dans l'équation. Au-delà des configurations physiques, l'observable pertinente est bien souvent une valeur moyenne ou médiane d'une quantité

(par exemple le *Path Loss*). Le calcul d'une telle quantité est possible numériquement en multipliant les expériences, ce qui se traduit par un très grand nombre de seconds membres. Ceci est par ailleurs à rapprocher avec le domaine des « *Computer Experiments* » et de la gestion des incertitudes [89].

Variabilité de l'objet De même que les excitations sont incertaines, la configuration de la cavité l'est également. Il est donc nécessaire de réaliser plusieurs simulations sur des cavités différentes. Dans la plupart des cas la variation de la cavité se traduit par des différences locales, de géométrie et/ou de conditions aux limites. En utilisant des méthodes de décomposition de domaine, on réduit fortement le temps de calcul nécessaire à l'étude de cette variabilité. La méthode proposée doit donc être compatible avec la décomposition de domaines.

Un solveur direct rapide répondrait à ces trois exigences. Le reste de ce document sera consacré à la présentation, à l'étude et la parallélisation d'une classe de solveurs rapides : les $\mathcal{H}$-Matrices. Nous donnerons une présentation des algorithmes, tels qu'ils furent introduits dans les années 2000, et de leur application aux éléments finis de frontière pour l'électromagnétisme. Nous proposons une implémentation parallèle efficace nécessaire à l'étude d'objets de grande dimension devant la longueur d'onde. Cependant, le champ d'application de cette méthode étant bien plus large que les problèmes de cavités, l'essentiel de la présentation ne sera pas spécifique aux problèmes industriels introduits dans ce chapitre. Nous y reviendrons cependant dans le chapitre 6, consacré aux applications.

Chapitre 2

Introduction aux $\mathcal{H}$-Matrices

Sommaire

2.1 Contexte et motivation **39**
2.2 Diffraction d'une onde électromagnétique **40**
2.3 $\mathcal{H}$-Matrices : Principes de base **50**
2.4 $\mathcal{R}k$-Matrices **53**
2.5 Algorithme mono-niveau **60**
2.6 Algorithme hiérarchique **62**

2.1 Contexte et motivation

Comme ceci détaillé dans la section 1.6, les spécificités des environnements de type cavité considérés dans ce document mènent à des difficultés de calcul dans le formalisme des méthodes BEM. De manière très schématique, nous pouvons énoncer que ces difficultés s'articulent autour du caractère (a) haute fréquence et (b) réfléchissant du milieu. La première caractéristique rend difficile le calcul par un solveur direct du fait du grand nombre d'inconnues de la discrétisation. La seconde pénalise fortement la convergence d'un solveur itératif basé sur la FMM, qui serait adapté à cette taille de problème.

Une solution possible pour résoudre ce problème est le choix d'une méthode numérique pour laquelle ces caractéristiques ne sont pas problématiques, et dans cette partie nous proposons deux approches :

— L'emploi d'un solveur direct « rapide » – dans le sens de la FMM –, c'est-à-dire approché et présentant une complexité calculatoire asymptotique meilleure que $\mathcal{O}(N^2)$, où N est le nombre d'inconnues du problème ;

— La construction et l'implémentation d'un préconditionneur adapté, calculé rapidement (toujours au même sens) et permettant alors d'accélérer (en nombre d'itérations) la convergence d'un solveur itératif accéléré (produit multipôle) par FMM.

Ces deux approches sont deux applications possibles et reliées d'une famille de méthodes de résolution portant le nom de $\mathcal{H}$-Matrices [62, 63]. D'un point de vue pratique,

il s'agit d'un format de compression avec perte de matrices, associé à une famille d'opérations approchées. L'efficacité en termes de taux de compression et de précision de l'approximation repose sur certaines propriétés de la matrice initiale. Dans ce document, il s'agit d'une matrice d'interaction d'une discrétisation par éléments finis de frontière d'une équation intégrale.

Cette méthode repose sur un découpage hiérarchique en blocs de la matrice du problème, et sur la compression de certains de ces blocs. Une fois la compression de la matrice effectuée, il existe une algèbre approchée associée qui permet de réaliser des opérations sur cette matrice. Dans certains cas, cette algèbre est suffisamment complète pour permettre de réaliser une version approchée des algorithmes usuels d'algèbre linéaire tels que l'inversion, la factorisation LU ou LDL^T, et la résolution de systèmes linéaires qui y est associée. Dans d'autres cas, il n'est possible que de former le produit de cette matrice compressée avec un vecteur, ce qui est toutefois suffisant pour mettre en œuvre des méthodes itératives basées sur des espaces de Krylov.

Pour ce type de compression hiérarchique, les principes et les idées sous-jacentes sont proches de ceux à la base de la FMM. Les $\mathcal{H}$-Matrices peuvent être vues comme le pendant algébrique et matriciel de la méthode analytique qu'est la FMM, la mise en œuvre en étant néanmoins très différente.

Le cas qui nous intéresse particulièrement dans la suite de ce document est l'exploitation du riche ensemble d'opérations permis par les $\mathcal{H}$-Matrices afin de réaliser des opérations de type factorisation de matrice. Dans cette situation, il est alors possible d'obtenir un préconditionneur avec une faible précision, ou un solveur direct rapide avec une précision plus élevée, le temps de calcul étant bien entendu croissant avec l'augmentation de la précision demandée.

Dans ce chapitre, après des rappels sur l'approximation des équations de Maxwell par une formulation variationnelle de type éléments finis de frontière, nous aborderons les principes des approximations et algorithmes des $\mathcal{H}$-Matrices tels qu'ils furent introduits dans la littérature [22, 34, 55–57, 62, 63] au début des années 2000.

Nous ne présentons dans ce chapitre que les « grandes lignes » des idées sous-jacentes aux $\mathcal{H}$-Matrices. Pour une exposition plus complète couvrant les fondements mathématiques ainsi que la mise en œuvre numérique, le lecteur pourra se reporter à [80, 95].

2.2 Diffraction d'une onde électromagnétique

Dans cette section, nous donnons un aperçu de la démarche de modélisation d'un problème de diffraction électromagnétique, depuis les équations de Maxwell jusqu'à la construction et la résolution du système linéaire issu de la discrétisation suivant la méthode des éléments finis de frontière, qui est l'étape que les $\mathcal{H}$-Matrices permettent d'accélérer.

2.2.1 Équations de Maxwell harmoniques

Une onde électromagnétique est définie par un couple de champs vectoriels de $\mathbb{R}^3$, le champ électrique $\boldsymbol{E}$ et le champ magnétique $\boldsymbol{H}$. Dans cette section, nous considérons

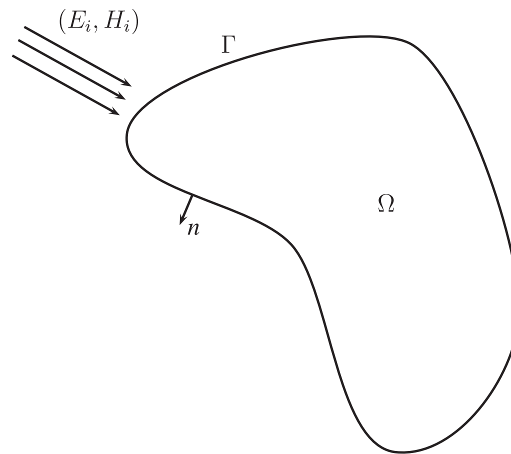


Figure 2.1 – Diffraction d'une onde par un objet

l'interaction d'une onde électromagnétique dite incidente $(\boldsymbol{E}_i, \boldsymbol{H}_i)$ avec un objet borné régulier parfaitement conducteur Ω de frontière Γ et de normale sortante $\boldsymbol{n}$, dans le vide de permittivité électrique ε_0 et de perméabilité magnétique μ_0. Cette situation est décrite par la figure 2.1.

Les équations de Maxwell s'écrivent dans ce milieu :

$$\begin{cases} -\varepsilon_0 \frac{\partial \boldsymbol{E}}{\partial t} + \boldsymbol{\nabla} \wedge \boldsymbol{H} = 0 & \text{dans } \mathbb{R}^3 \backslash \Omega \\ \mu_0 \frac{\partial \boldsymbol{H}}{\partial t} + \boldsymbol{\nabla} \wedge \boldsymbol{H} = 0 & \text{dans } \mathbb{R}^3 \backslash \Omega \end{cases} \tag{2.2.1}$$

Il faut bien entendu ajouter à ce système des conditions aux limites sur l'interface entre le milieu extérieur et l'objet, c'est-à-dire sur Γ. Dans le cas d'un conducteur parfait, ces relations sont :

$$\boldsymbol{E} \wedge \boldsymbol{n} = -\boldsymbol{E}_i \wedge \boldsymbol{n} \text{ sur } \Gamma \tag{2.2.2}$$

Par ailleurs, nous restreindrons notre étude à une onde incidente monochromatique plane de pulsation ω. Ce problème modèle intervient en particulier dans les calculs de furtivité radar dont nous verrons des exemples dans le chapitre consacré aux applications industrielles. Le problème de la signature radar est de caractériser l'écho renvoyé par un objet soumis à une onde incidente. Le radar étant situé loin de la cible par rapport à la longueur d'onde (plusieurs kilomètres pour une longueur d'onde métrique ou centimétrique), l'onde générée par celui-ci est localement plane, et elle est donc approximée de cette façon avec une grande fidélité. De plus, l'onde émise par le radar est monochromatique, et le vecteur d'intérêt est l'onde renvoyée, ou onde diffractée. On notera par ailleurs que la position du radar est fixe, et que bien souvent l'antenne de réception est également fixe. Il est alors nécessaire de soumettre l'objet à des illuminations suivant un grand nombre d'angles d'incidence pour caractériser complètement sa réponse, chaque illumination étant un second membre différent dans l'équation.

On peut alors écrire les équations dans le domaine fréquentiel. Nous poserons par abus

de notation $(\boldsymbol{E}, \boldsymbol{H})$ les champs vectoriels complexes $\mathbb{R}^3 \to \mathbb{C}^3$ tels que :

$$\begin{cases} \boldsymbol{E}(t,x) = \mathfrak{Re}\left(\boldsymbol{E}(x)e^{-i\omega t}\right) \\ \boldsymbol{H}(t,x) = \mathfrak{Re}\left(\boldsymbol{H}(x)e^{-i\omega t}\right) \end{cases} \tag{2.2.3}$$

soient des solutions du système de Maxwell (2.2.1). Ces champs sont alors solution du système de Maxwell harmonique, soit :

$$\begin{cases} \boldsymbol{\nabla} \wedge \boldsymbol{E} - i\omega\mu_0 \boldsymbol{H} = 0 & \text{dans } \mathbb{R}^3 \backslash \Omega \\ \boldsymbol{\nabla} \wedge \boldsymbol{H} + i\omega\varepsilon_0 \boldsymbol{E} = 0 & \text{dans } \mathbb{R}^3 \backslash \Omega \\ \boldsymbol{E} \wedge \boldsymbol{n} = -\boldsymbol{E}_i \wedge \boldsymbol{n} & \text{sur } \partial\Omega \\ \lim\limits_{r \to +\infty} r \left\| \sqrt{\varepsilon_0}\boldsymbol{E} - \sqrt{\mu_0}\boldsymbol{H} \wedge \dfrac{\boldsymbol{r}}{r} \right\| = 0 & \end{cases} \tag{2.2.4}$$

Il faut noter que la dépendance en temps dans (2.2.3) est en $-i\omega t$. Cette convention est importante car elle n'est pas universellement adoptée.

Les deux premières équations sont le système de Maxwell dans le domaine fréquentiel dans le milieu de propagation vérifié par la transformée de Fourier des champs, la troisième la condition aux limites correspondant à un objet parfaitement conducteur, et la dernière la condition de radiation de Silver-Müller.

Cette condition est nécessaire pour assurer l'unicité de la solution [80, 95]. En effet, on rappelle que dans le domaine temporel les équations de Maxwell admettent deux solutions, les solutions causales et les solutions anti-causales, ou solutions en potentiels avancés. Ces solutions « remontent » le temps et produisent des effets avant les causes, et résultent de la symétrie par renversement du temps des équations de Maxwell. Ces solutions sont bien entendu écartées trivialement dans le domaine temporel, mais dans le domaine fréquentiel la variable de temps n'apparaît plus, et cette condition permet de s'assurer du bon comportement des solutions au voisinage de l'infini, ce qui correspond à la solution causale [80, 95].

La quantité d'intérêt dans ce cadre est le couple de champs diffractés $(\boldsymbol{E}_{diff}, \boldsymbol{H}_{diff})$ définis par :

$$(\boldsymbol{E}_{diff}, \boldsymbol{H}_{diff}) = (\boldsymbol{E} - \boldsymbol{E}_i, \boldsymbol{H} - \boldsymbol{H}_i) \tag{2.2.5}$$

On rappelle par ailleurs l'impédance du milieu $Z := \sqrt{\mu_0\varepsilon_0}$, la vitesse de propagation des ondes dans le vide $c := 1/\sqrt{\mu_0\varepsilon_0}$ et la définition du nombre d'onde $\kappa := 2\pi/\lambda$.

2.2.2 Théorème de représentation intégrale

La question de l'espace dans lequel les solutions $(\boldsymbol{E}, \boldsymbol{H})$ sont recherchées se pose. Il faut noter que cet espace doit permettre de donner un sens aux opérateurs de divergence et de rotationnel ainsi qu'à la trace des champs, et que nous cherchons des solutions d'énergie finie.

On introduit les espaces de Sobolev suivants :

$$\begin{cases} H(div) := \left\{ v \in (L^2(\Omega))^3,\ \boldsymbol{\nabla} \cdot v \in L^2(\Omega) \right\} \\ H(rot) := \left\{ v \in (L^2(\Omega))^3,\ \boldsymbol{\nabla} \wedge v \in (L^2(\Omega))^3 \right\} \end{cases} \tag{2.2.6}$$

On note alors que l'espace

$$X = \left\{ \boldsymbol{E} \in H(rot),\ \boldsymbol{E} \in H(div),\ (\boldsymbol{E} \cdot \boldsymbol{n}) \in H^{1/2}(\Gamma) \right\}$$

est inclus dans $(H^1(\Omega))^3$.

On définit les courants surfaciques $\boldsymbol{j}$ et $\boldsymbol{m}$ fonctions sur Γ comme les traces tangentielles des champs sur la frontière de l'objet :

$$\begin{cases} \boldsymbol{j} = \boldsymbol{n} \wedge \boldsymbol{H} \\ \boldsymbol{m} = \boldsymbol{n} \wedge \boldsymbol{E} \end{cases} \tag{2.2.7}$$

Pour un conducteur parfait, les conditions aux limites impliquent la nullité de $\boldsymbol{m}$, ce qui n'est pas le cas en général. Le courant surfacique $\boldsymbol{j}$ est le saut de la composante tangentielle de $\boldsymbol{H}$ au passage de l'interface entre le milieu extérieur et le conducteur parfait. Il correspond à la limite du courant volumique dans un milieu dont l'épaisseur de peau tend vers 0 (soit un milieu dont la conductivité tend vers l'infini).

La connaissance de ce courant permet de retrouver la valeur du champ diffracté en tout point de l'espace hors de Γ, c'est le théorème de représentation intégrale :

$$\begin{cases} \boldsymbol{E}(x) = i\omega\mu \displaystyle\int_{\partial\Omega} G(|x-y|)\boldsymbol{j}(y)dy + \frac{i}{\omega\varepsilon}\boldsymbol{\nabla}_x \int_{\partial\Omega} G(|x-y|)\boldsymbol{\nabla}_{\partial\Omega} \cdot \boldsymbol{j}(y)dy & x \in \mathbb{R}^3 - \Omega \\ \boldsymbol{H}(x) = -\boldsymbol{\nabla}_x \wedge \displaystyle\int_{\partial\Omega} G(|x-y|)\boldsymbol{j}(y)dy & x \in \mathbb{R}^3 - \Omega \end{cases} \tag{2.2.8}$$

avec G le noyau de Green, solution élémentaire de l'équation de Helmholtz en 3D (associée à la condition de radiation)

$$\Delta u + \kappa^2 u = -\delta_0 \tag{2.2.9}$$

δ_0 étant la distribution de Dirac, et la solution de cette équation satisfaisant la condition de radiation est

$$G(|x|) = \frac{e^{i\kappa|x|}}{4\pi|x|} \tag{2.2.10}$$

Remarque 2.1 (Notations)**.** *Plusieurs notations différentes coexistent dans la littérature pour le noyau de Green, qui est soit noté $G(x, y)$, soit $G(x-y)$, soit $G(|x-y|)$. Dans le premier cas, c'est une fonction de $\mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{C}$, $\mathbb{R}^3 \to \mathbb{C}$ dans le second et $\mathbb{R}_+ \to \mathbb{C}$ dans le dernier. Ces notations désignent de façon non ambiguë la même fonction, ce qui justifie l'abus de notation confondant ces fonctions.*

Le théorème de représentation intégrale permet de substituer aux inconnues volumiques classiques (champs $\boldsymbol{E}$ et $\boldsymbol{H}$) les inconnues surfaciques correspondant aux courants $\boldsymbol{j}$ et $\boldsymbol{m}$. La justification physique intuitive de cette équivalence est claire ; les interactions entre le champ et l'objet étant localisées sur sa surface, celles-ci contiennent l'intégralité de l'information du problème.

Ce changement d'inconnue étant fait, le problème s'exprime alors sous la forme d'une équation de Fredholm. La mise sous forme variationnelle aboutit à la résolution d'un système linéaire, dont la solution est le couple de courants $(\boldsymbol{j}, \boldsymbol{m})$.

Soit une fonction test $\boldsymbol{j}^t$ dans le même espace que $\boldsymbol{j}$, alors le courant solution $\boldsymbol{j}$ est tel que :

$$\forall \boldsymbol{j}^t,\ \int_{\Gamma\times\Gamma} G(|x-y|)\left(\boldsymbol{j}(y)\cdot\boldsymbol{j}^t(x)-\frac{1}{\kappa^2}\boldsymbol{\nabla}_\Gamma\cdot\boldsymbol{j}(y)\boldsymbol{\nabla}_\Gamma\cdot\boldsymbol{j}^t(x)\right)dxdy = \frac{i}{\kappa Z_0}\int_\Gamma \boldsymbol{E}_i(y)\cdot\boldsymbol{j}^t(y)dy \quad (2.2.11)$$

Cette équation est obtenue directement à partir de la formule de représentation intégrale pour le champ électrique $\boldsymbol{E}$; elle est appelée *Electric Field Integral Equation* (EFIE). Suivant le même schéma, il est possible d'obtenir une formulation variationnelle basée sur la représentation du champ magnétique $\boldsymbol{H}$, appelée *Magnetic Field Integral Equation* (MFIE). L'EFIE est symétrique par permutation de $\boldsymbol{j}$ et $\boldsymbol{j}^t$, contrairement à la MFIE.

Il est courant de rencontrer une troisième formulation variationnelle, appelée *Combined Field Integral Equation* (CFIE), définie comme une combinaison convexe de chaque terme de l'EFIE (2.2.11) et de la MFIE [94] :

$$CFIE := \alpha EFIE + (1-\alpha)\frac{i}{\kappa}MFIE$$

Le choix de α est libre, cependant le choix $\alpha = 0,2$ semble empiriquement bon dans un certain nombre de configurations. Cette méthode a été introduite pour pallier le caractère « mal posé » de l'équation de l'EFIE lorsque la fréquence est un des modes propres du problème intérieur (dans Ω) associé. La formulation CFIE n'a pas de fréquence propre réelle [80].

2.2.3 Discrétisation

On utilise une méthode d'éléments finis, c'est-à-dire que l'objet sera approché par une surface polyèdrale, dans ce cas composée de triangles. On fixe ensuite un espace vectoriel de fonctions tests, dont on choisit une base. Dans ce cas, les éléments finis considérés sont ceux de Raviart-Thomas [86]. Les degrés de liberté du problème (coefficients de la combinaison linéaire des fonctions de base à trouver) sont localisés sur les arêtes des triangles, et représentent le flux du courant $\boldsymbol{j}$ ou $\boldsymbol{m}$ à travers cette arête (figure 2.2).

Une arête est partagée par un faible nombre de triangles (2 pour la majorité des arêtes). Le support des fonctions de base φ_i est ainsi restreint à un faible nombre de triangles. Pour chaque arête A_i de la surface discrétisée partagée entre les triangles $\mathcal{T}_{i_1}$ et $\mathcal{T}_{i_2}$, une orientation arbitraire est définie, ce qui donne une normale $\boldsymbol{n}_i$ à celle-ci, comme représenté sur la figure 2.2. Le degré de liberté λ_i associé est alors défini par (pour le courant $\boldsymbol{j}$) :

$$\lambda_i = \int_{A_i} \boldsymbol{j}(x)\cdot\boldsymbol{n}_i(x)dx \quad (2.2.12)$$

Sur le triangle $\mathcal{T}_{i_1}$ on note $S(\boldsymbol{s})$ le sommet opposé à A_i. On pose alors comme fonction de base :

$$\forall \boldsymbol{x}\in\mathcal{T}_{i_1},\ \varphi_i(\boldsymbol{x}) := \frac{\boldsymbol{x}-\boldsymbol{s}}{2|\mathcal{T}_{i_1}|}\in\mathbb{R}^3 \quad (2.2.13)$$

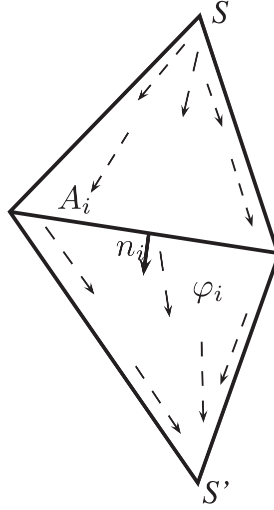


FIGURE 2.2 – Représentation d'une fonction de base φ_i

On note que cette fonction est vectorielle (à valeur dans la surface discrétisée), affine, de flux nul à travers les autres arêtes des deux triangles $\mathcal{T}_{i_1}$ et $\mathcal{T}_{i_2}$, et de flux 1 à travers A_i. De même, la définition de cette fonction sur le triangle $\mathcal{T}_{i_2}$ de sommet opposé $S'(\boldsymbol{s}')$ est :

$$\forall \boldsymbol{x} \in \mathcal{T}_{i_2},\ \varphi_i(\boldsymbol{x}) := -\frac{\boldsymbol{x} - \boldsymbol{s}'}{2|\mathcal{T}_{i_2}|} \in \mathbb{R}^3 \tag{2.2.14}$$

2.2.4 Écriture matricielle

À partir de la formulation variationnelle (2.2.11), et en choisissant comme fonction test chaque fonction de base φ_i définie précédemment, et en notant que $\boldsymbol{j} = \sum_i \lambda_i \varphi_i$, on obtient pour une discrétisation comportant N arêtes la version discrétisée du problème (2.2.11), c'est-à-dire la recherche du vecteur $(\lambda_i)_{i=1\dots N}$ tel que :

$$\sum_{i=1}^{N} \lambda_i \left(\int_{\Gamma\times\Gamma} G(|x-y|) \left(\varphi_i(y)\cdot\varphi_j(x) - \frac{1}{\kappa^2} \boldsymbol{\nabla}_\Gamma \cdot \varphi_i(x) \boldsymbol{\nabla}_\Gamma \cdot \varphi_j(y) \right) dxdy \right)$$
$$= \frac{i}{\kappa Z_0} \int_\Gamma \boldsymbol{E}_i(x)\cdot\varphi_j(x)dx \tag{2.2.15}$$

Cette égalité correspond à l'équation matricielle

$$A\lambda = b \tag{2.2.16}$$

avec

$$\begin{cases} A_{ij} = \displaystyle\int_{\Gamma\times\Gamma} G(|x-y|) \left(\varphi_i(y)\cdot\varphi_j(x) - \frac{1}{\kappa^2} \boldsymbol{\nabla}_\Gamma \cdot \varphi_i(x) \boldsymbol{\nabla}_\Gamma \cdot \varphi_j(y) \right) dxdy & (2.2.17) \\ b_j = \dfrac{i}{\kappa Z_0} \displaystyle\int_\Gamma \boldsymbol{E}_i(x)\cdot\varphi_j(x)dx & (2.2.18) \end{cases}$$

2.2.4.1 Commentaires

En premier lieu, on retrouve de manière classique pour les méthodes d'éléments finis, que la matrice A est une approximation de l'opérateur du problème dans l'espace des applications linéaires de l'espace vectoriel engendré par la base des fonctions tests φ_i. À ce titre, il ne dépend pas de l'illumination par une onde incidente. Ceci est important car, dans de nombreux cas d'application, on cherche le champ diffracté engendré par une série d'illuminations. Par exemple, dans le cas des calculs de signature radar, des milliers de directions d'onde incidente sont en général étudiées, ce qui mène à autant de seconds membres, leur nombre étant potentiellement multiplié par deux si l'utilisateur est intéressé par deux polarisations des ondes incidentes.

Ensuite, le support de chacune des intégrales doubles entrant dans le calcul du terme matriciel A_{ij} (2.2.17) est restreint à au plus 2 triangles pour chacune de ces intégrales, du fait de la définition de la base de fonctions φ_i retenue, ce qui limite l'effort de calcul de ces termes. Cependant, il est important de noter que ces intégrales sont à priori non nulles pour tout i et pour tout j, ce qui est inhabituel dans les méthodes d'éléments finis.

En effet, le théorème de représentation intégrale a permis de substituer aux inconnues spatiales (les champs) des inconnues surfaciques. Ces inconnues sont moins nombreuses, mais la réduction de leur nombre est accompagnée d'un passage d'une matrice creuse dans le cas des formulations variationnelles volumiques à une matrice pleine, demandant plus de calculs à ordre égal. Cette matrice est par ailleurs complexe, et dans le cas de l'EFIE symétrique, non hermitienne, non définie positive.

Enfin, le calcul des intégrales A_{ii} et des autres termes correspondant à des triangles proches est numériquement délicat. En effet, le noyau de Green $G(R)$ présente une singularité en $R = 0$. Celle-ci est intégrable, cependant le calcul numérique par une classique quadrature de Gauss n'est pas possible pour ces triangles. On procède alors à une intégration semi-analytique, utilisant une quadrature de Gauss pour une des intégrales, et un calcul analytique pour l'autre. Pour les triangles lointains, le problème ne se pose pas et les calculs se font à l'aide de points de Gauss sur les deux triangles.

2.2.4.2 Calcul du champ

La résolution de (2.2.16) permet d'obtenir uniquement les courants surfaciques, et il est alors nécessaire d'utiliser la formule de représentation intégrale (2.2.8) pour obtenir les champs. Ceci se fait en utilisant l'expression (2.2.8) en décomposant $\boldsymbol{j}$ sur les triangles.

Cela permet d'obtenir le champ en un point quelconque de l'espace $\mathbb{R}^3 \backslash \Omega$, sous sa forme discrétisée, c'est-à-dire en remplaçant les intégrales par des sommes sur les degrés de liberté, exactement comme dans le passage de (2.2.8) à (2.2.15).

On a, en notant abusivement le courant discrétisé $\boldsymbol{j}$ de la même manière que le courant surfacique :

$$\forall x \in \Gamma, \ \boldsymbol{j}(x) = \sum_{i=1}^{N} \lambda_i \varphi_i(x) \tag{2.2.19}$$

et donc pour tout $y \in \mathbb{R}^3 \backslash \Omega$

$$\begin{cases} \boldsymbol{E}(y) = i\omega\mu_0 \sum_{i=1}^{N} \lambda_i \left(\int_\Gamma G(|x-y|)\varphi_i(x)dx + \frac{1}{\kappa^2} \boldsymbol{\nabla}_y \int_\Gamma G(|x-y|) \boldsymbol{\nabla} \cdot \varphi_i(x)dx \right) \\ \boldsymbol{H}(y) = \sum_{i=1}^{N} \lambda_i \left(\boldsymbol{\nabla} \wedge \int_\Gamma G(|x-y|)\varphi_i(x)dx \right) \end{cases} \tag{2.2.20}$$

Ce calcul a un coût, qui est de $\mathcal{O}(N)$ par point pour lequel le champ est recherché, et a donc une croissance linéaire avec le nombre de points où le champ est demandé. Ces complications peuvent sembler inutiles et coûteuses, mais elles ont un vrai avantage. En effet, le point d'observation du champ peut être aussi loin que souhaité de l'objet, il n'est pas nécessaire de mailler le volume jusqu'à ce point, et de plus aucune information n'est perdue dans le calcul de la propagation du champ jusqu'à ce point.

C'est une des raisons pour laquelle la méthode BEM est connue pour sa grande précision. En effet, la représentation intégrale vérifie exactement l'équation aux dérivées partielles et la condition de radiation à l'infini. L'erreur de discrétisation provient, elle, de l'approximation de la géométrie et de la condition aux limites, qui n'est vérifiée qu'au sens faible.

Il faut noter que ce calcul peut être simplifié dans le cas où l'utilisateur est intéressé par le comportement asymptotique du champ à une grande distance de l'objet. L'onde diffractée peut se mettre sous la forme :

$$\boldsymbol{E}_{diff}(y) = \mathcal{F}\left(\frac{y}{|y|}\right) \frac{e^{i\kappa|y|}}{4\pi|y|} \left(1 + \mathcal{O}(\frac{1}{|y|})\right) \tag{2.2.21}$$

Elle est donc localement plane. Dans cette expression, seule la connaissance de $\mathcal{F}$ est recherchée, ce qui mène à des simplifications de calcul et correspond également à une grandeur d'intérêt naturelle pour le calcul de signature radar, ou pour le rayonnement d'une antenne.

2.2.4.3 Résumé

En résumé, les étapes nécessaires à une simulation par BEM sont :

— Discrétisation de la géométrie, en respectant un critère combinant la fidélité géométrique et la longueur d'onde, suivant la plus stricte des deux contraintes ;

— Calcul de la projection de la trace de l'onde incidente sur la base des φ_i, ce qui donne le second membre de l'équation matricielle ;

— Calcul de la matrice A (2.2.17) du système, qui ne dépend – rappelons-le – que de l'objet et non du second membre ;

— Résolution du système linéaire pour chaque second membre (2.2.18), donnant accès aux courants de surface ;

— Calcul des champs $(\boldsymbol{E}, \boldsymbol{H})$ aux points de l'espace désirés.

Il faudrait ajouter en pratique une étape de pré-traitement permettant de placer les degrés de liberté sur la géométrie. Cette étape est simple dans le cas d'un conducteur

parfait, mais devient bien plus compliquée (bien que systématique) pour un code de calcul traitant un grand nombre de conditions aux limites et d'éléments. On dénombre en effet plus d'une centaine de cas dans les situations courantes.

2.2.5 Remarques sur la résolution numérique

Une fois la mise en équation sous la forme d'un système matriciel faite, le problème devient celui de la résolution précise et rapide de ce système. Pour cela, il est utile de donner un ordre de grandeur de la taille de celui-ci afin de fixer les idées.

Étant donné que les inconnues sont réparties sur la surface de l'objet, le critère pertinent est donc la surface de celui-ci. Le premier critère à fixer est celui du pas de discrétisation. Afin d'assurer une bonne précision à la méthode, il est nécessaire de mailler l'objet avec des triangles tel que le nombre de points par longueur d'onde soit au moins égal à 7 (et dans l'idéal au moins égal à 10) de manière à représenter fidèlement les traces des ondes incidentes sur Γ et les courants de surface solutions. Il faut noter que ce critère ne doit pas se substituer au critère imposant le respect de la géométrie de l'objet par sa discrétisation. Dans les cas basse fréquence, ce critère sera donc remplacé par un autre basé par exemple sur l'erreur de corde de la discrétisation géométrique, cependant que nous ne considérerons pas ici pour simplifier.

L'objectif de tout mailleur étant de produire des triangles équilatéraux pour des raisons de précision numérique des calculs, nous supposerons que la frontière Γ de surface S est maillée avec de tels triangles de côté $h = \lambda/10$ avec λ la longueur d'onde. Notons N_{tri} le nombre de triangles, vérifiant :

$$S = N_{tri} h^2 \frac{\sqrt{3}}{4} \tag{2.2.22}$$

Un triangle ayant 3 arêtes partagées par 2 triangles chacune (pour un objet fermé), on a dans le cas d'un matériau parfaitement conducteur $N = \frac{3}{2} N_{tri}$, et donc finalement :

$$N = \frac{800}{3\sqrt{3}} \frac{S}{\lambda^2} \simeq 150 \frac{S}{\lambda^2}$$

Cette estimation étant fixée, il convient de poser la question du coût de résolution d'un système. Il existe essentiellement deux classes de solveurs, les solveurs directs et itératifs [51].

Les solveurs directs reposent sur une factorisation (LU ou LDL^T par exemple) de la matrice sous une forme permettant une résolution plus aisée du système linéaire. La complexité asymptotique optimale de ce type d'opération est une question ouverte, il est cependant connu qu'elle est inférieure à $\mathcal{O}(N^{\log_2(7)})$ (algorithme de Strassen) en pratique et supérieure à $\mathcal{O}(N^2 \log(N))$ [1] [87]. Néanmoins, pour les applications pratiques, la complexité asymptotique est $\mathcal{O}(N^3)$.

Ces solveurs présentent de nombreux atouts :

1. Ceci peut paraître contradictoire. L'aspect « pratique » de l'algorithme de Strassen le rend avantageux pour certaines tailles de matrices, mais dans le cas général, les solveurs utilisent quasi universellement l'algorithme naïf en $\mathcal{O}(N^3)$, d'où les deux sens de « pratique ».

— Une grande précision. En effet, lorsque la matrice est relativement bien conditionnée et que des techniques de pivotage sont employées, une précision de l'ordre de la précision machine (10^{-16}) peut être assurée [64].
— Un temps de calcul et un espace mémoire requis déterministes.
— Une faible dépendance du temps de calcul au nombre de seconds membres.

En effet, la résolution du système a une complexité en $\mathcal{O}(N^2)$ par second membre une fois la factorisation effectuée. Ces solveurs procèdent en deux étapes : dans un premier temps, la matrice est factorisée sous une forme particulière, le plus souvent LU pour une matrice non symétrique, avec L une matrice triangulaire inférieure dont la diagonale est constituée de 1, (« lower ») et U une matrice triangulaire supérieure (« upper »). Cette factorisation étant faite avec un coût $\mathcal{O}(N^3)$, la résolution du système se ramène à la résolution de deux systèmes triangulaires inférieurs et supérieurs, appelés descente et remontée suivant le sens de parcours de la matrice par l'algorithme. Cette deuxième étape a un coût de $\mathcal{O}(N^2)$ par second membre, ce qui est négligeable devant le temps de factorisation.

Les solveurs itératifs reposent, pour leur part, sur des opérations dont la complexité est $\mathcal{O}(N^2)$ par itération. Le nombre d'itérations n'est cependant pas connu à l'avance, et la seule borne (en l'absence d'erreurs numériques) est de N itérations, bien qu'en pratique la convergence soit atteinte bien avant. Une difficulté supplémentaire dans l'utilisation de ces solveurs est posée par le mauvais conditionnement des matrices BEM, ce qui gêne la convergence des solveurs itératifs. Il est alors nécessaire de préconditionner le système linéaire. La recherche d'un préconditionneur robuste, précis et peu coûteux en calcul est un sujet de recherche actif. En effet, le meilleur préconditionneur (assurant une convergence en une itération) est l'inverse de la matrice A, qui n'est pas connue ! La recherche d'un préconditionneur est donc celle d'une bonne approximation de A^{-1} pour un coût de calcul faible. De plus, la réduction du temps de calcul permise par les solveurs itératifs se fait au détriment de la précision du résultat et du caractère multi second membres, le temps de calcul croissant en première approximation linéairement avec leur nombre [2].

Par ailleurs, le temps de calcul des termes de la matrice devient non négligeable lorsque le coût de la résolution est plus faible que $\mathcal{O}(N^3)$. En effet, dans le cas d'un calcul avec 3 points de Gauss par triangle, le calcul de l'interaction entre deux degrés de liberté conduit à 36 évaluations du noyau, celui-ci nécessitant le calcul d'exponentielles complexes, dont le coût est grand sur les machines actuelles. Par exemple, sur un processeur « Intel Sandy Bridge » (génération la plus récente et courante sur les machines de calcul en 2013), la latence de calcul d'un cosinus est de 119 cycles sans possibilité de recouvrement, contre un seul pour une multiplication (et encore moins dans le cas où l'exploitation des instructions vectorielles est possible) [65, Annexe C.3.1]. Ceci signifie que le temps de calcul de la matrice (dont nous venons de donner une sous-estimation) ne peut être négligé dans le cas des solveurs itératifs.

Enfin, le coût de stockage des matrices est important. En effet, pour un problème avec $N = 10^6$, dans le cas de l'EFIE le coût de stockage de la matrice est de 8To (il serait de 16To pour la MFIE du fait de son caractère non symétrique), et grandit avec le carré du nombre de degrés de liberté.

2. En réalité deux facteurs modulent cette observation : les effets de cache (positif) et la vitesse de convergence variable d'un second membre à l'autre (positif ou négatif).

Toutes ces raisons ont motivé la recherche de solveurs ne présentant pas certains des inconvénients exposés ici. Une classe de solveurs dont le succès est indéniable est la FMM, permettant de réaliser une approximation du produit Ax pour tout vecteur x, cette opération étant au cœur des solveurs itératifs. Cette approximation a l'avantage de ne pas nécessiter d'assemblage de la matrice A, et d'offrir une complexité asymptotique en $\mathcal{O}(N\log_2^2(N))$ pour la méthode multi-niveau dans le cas de la résolution de l'équation des ondes par BEM. Découverte dans les années 80 pour la résolution du problème à N corps (attraction gravitationnelle de N corps célestes) [59], elle a depuis été étudiée en détail dans le cadre de l'équation des ondes, avec en particulier [41–43] pour un traitement détaillé de son implémentation numérique, et mise en œuvre industriellement avec des implantations parallèles dans le début des années 2000, et a permis de traiter des problèmes physiques d'au moins un ordre de grandeur supérieur aux méthodes BEM précédentes. Un exposé détaillé de cette méthode, de son optimisation, de sa parallélisation et de sa mise en œuvre industrielle pour l'électromagnétisme se trouve dans [94].

Cependant, la FMM ne permet pas à ce jour de proposer un solveur direct, et la nécessité de préconditionner le système est intacte. Ce problème est rendu plus aigu par la taille des problèmes rendus traitables par cette méthode, et ce sont à ces deux difficultés que les $\mathcal{H}$-Matrices proposent d'apporter une réponse.

2.3 $\mathcal{H}$-Matrices : Principes de base

Comme cela a été vu dans la section précédente, les méthodes intégrales permettent de ramener la résolution d'un problème de diffraction d'ondes électromagnétiques à la résolution d'un système linéaire. Il existe ensuite diverses méthodes pour résoudre le système, présentant des avantages et des inconvénients, mais du fait de la taille des problèmes considérés, seule une méthode plus rapide que $\mathcal{O}(N^2)$ est adaptée aux cas les plus grands.

Cependant, il est démontré qu'il est impossible de factoriser une matrice ou de faire un produit matrice-vecteur en moins de $\mathcal{O}(N^2\log^2(N))$ opérations [87]. Cette limitation semble entrer en contradiction avec l'existence de méthodes comme la FMM, mais il n'en est rien. Cette borne inférieure de complexité ne s'applique que dans le cas où la matrice est générale, et pour une résolution « exacte » (à la précision de la machine près), mais ne dit rien dans le cas contraire. Il apparaît donc que s'il est possible de faire mieux, c'est nécessairement en exploitant les spécificités de la matrice et/ou en utilisant des méthodes approchées.

Ces deux notions sont au cœur des $\mathcal{H}$-Matrices. En premier lieu, la structure de la matrice est importante, car, bien que certaines méthodes présentées ultérieurement fonctionnent en apparence en « boîte noire », leur efficacité repose en réalité sur des hypothèses vérifiées par les matrices BEM. Plus précisément, les algorithmes exploiteront la distribution spatiale des degrés de liberté pour en tirer une structure hiérarchique pour la matrice. Ensuite, dans toutes les variantes des $\mathcal{H}$-Matrices, la réduction de la quantité de calculs s'accompagne d'une perte d'information. Comme nous le verrons dans la suite, les blocs sont compressés suivant divers algorithmes, mais tous mènent au remplacement d'un bloc de matrice par un autre de rang plus faible, ce qui signifie qu'il est impossible de retrouver le bloc initial une fois la compression effectuée.

On retrouve alors ici les deux caractéristiques permettant de définir les $\mathcal{H}$-Matrices en une phrase : il s'agit d'un format de représentation de matrices hiérarchique compressé avec pertes, associé à une classe d'algorithmes opérant sur ce format, et permettant de réaliser les opérations usuelles sur les matrices à un coût réduit, avec en contrepartie une possible perte d'information [62].

2.3.1 Deux "ingrédients"

2.3.1.1 Découpage hiérarchique

Le premier « ingrédient » des $\mathcal{H}$-Matrices est le découpage hiérarchique de la matrice. Une $\mathcal{H}$-Matrice n'est en réalité pas une matrice au sens usuel du terme, mais il s'agit d'une structure arborescente pouvant être représentée conceptuellement par une matrice. Cette structure arborescente a la même raison d'être que celle apparaissant dans la FMM multi-niveau, ou MLFMM, qui est celle à laquelle nous ferons référence lorsque nous évoquons la FMM. Dans les deux cas, le but d'une structure arborescente est de traiter les interactions entre éléments de la façon la plus grossière possible, étant entendu que la proximité de deux groupes de degrés de liberté est le critère permettant de déterminer si une interaction doit être traitée plus finement ou non.

Dans le cas de la FMM, le découpage est le plus souvent réalisé à l'aide d'un arbre d'arité 8, appelé *octree*. En partant d'une boîte englobante de l'objet Ω (qui est borné), on procède récursivement à une division de cette boîte en 8 sous-boîtes, en coupant l'espace par 3 plans alignés suivant les axes, le découpage se poursuivant récursivement dans les boîtes non vides jusqu'à ce qu'une condition d'arrêt soit rencontrée. Dans le cas de la FMM « Plane Wave » [41], cette condition d'arrêt est liée à une taille minimale de boîte par rapport à la longueur d'onde λ.

Dans le cas des $\mathcal{H}$-Matrices, le découpage peut être effectué par un arbre binaire ou par un *octree*, le choix le plus courant étant un arbre binaire, mais ce n'est nullement une obligation. Dans ce cas, les boîtes ne sont coupées que par un seul plan à chaque niveau, dont la position peut être déterminée de diverses façons.

Par exemple, le plan peut être choisi orthogonal à la coordonnée selon laquelle l'extension de la boîte est la plus grande, et la position du plan séparateur peut être fixée au milieu de la boîte, ou à une coordonnée telle que le nombre de degrés de liberté est égal de part et d'autre.

Dans le premier cas, on s'assure de la plus grande réduction de volume des boîtes à chaque étape de division, et dans le deuxième de la décroissance la plus grande du nombre de degrés de liberté, et donc de la hauteur la plus faible de l'arbre. En effet, le critère d'arrêt le plus couramment utilisé est indépendant de la longueur d'onde et ne repose plus que sur des considérations de performance algorithmique. Il est courant de fixer pour ce critère un nombre maximal de degrés de liberté par feuille, qui sera choisi petit, comme nous le verrons dans la suite. De plus amples détails sur ces choix seront donnés à la section 3.1.3.

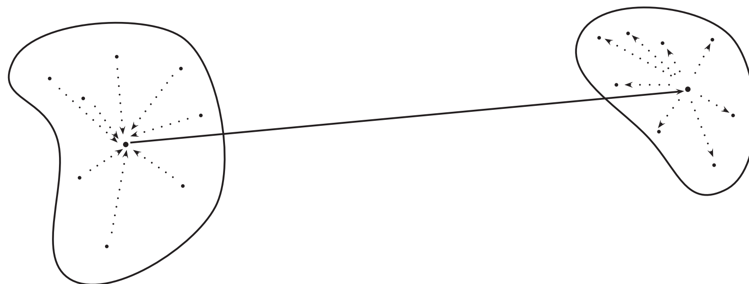


FIGURE 2.3 – Illustration de l'interaction optimisée entre deux groupes

2.3.1.2 Compression

Sans entrer dans les détails de la FMM, l'idée sous-jacente est de constituer des groupes de degrés de liberté, et de ne traiter les interactions entre ces groupes qu'à travers l'interaction entre les centres des groupes respectifs. Pour deux groupes de m et n degrés de liberté, l'interaction entre les éléments considérés individuellement mène au calcul de $m \times n$ interactions élémentaires. En supposant qu'il est possible de ramener toutes les interactions des deux groupes à un point arbitraire de celui-ci, puis de transmettre cette information à l'aide de k évaluations d'une « fonction de transfert » et enfin de redistribuer les interactions sur les m points du groupe « cible », on est alors ramené au calcul de $(m + n) \times k$ interactions, d'où la réduction en temps de calcul, comme illustré par la figure 2.3. Cette figure montre les trois étapes, avec à gauche le rassemblement de l'information sur un point, puis l'action de la fonction de transfert, et enfin la redistribution des interactions.

Pour une exposition plus complète du développement multipôlaire du noyau de Green, de sa décomposition en « rassemblement au centre », « transfert » et « distribution », le lecteur est invité à se reporter à [41] et [94].

Cette interprétation géométrique de la FMM qui repose avant tout sur un développement du noyau de Green G a une correspondance algébrique directe, qui est le point de vue adopté par les $\mathcal{H}$-Matrices. Prenons le point de vue analytique en premier. L'idée maîtresse dans la FMM est la séparation des variables. Notons g_r l'opération de rassemblement, t celle de transfert et g_d celle de distribution, x un point du premier groupe et y un point du second. On a alors :

$$G(x, y) \simeq G(|x - y|) = g_r(x, \nu)t(\nu)g_d(\nu, y) \tag{2.3.1}$$

avec ν la variable de la fonction de transfert.

La correspondance avec une forme matricielle est alors claire. En effet, l'interaction entre deux groupes de m et n degrés de liberté se traduit par une matrice d'interaction (sous-matrice de M définie par l'équation (2.2.17)) de taille $m \times n$. L'opération de rassemblement correspondrait dans ce cadre à une matrice de taille $m \times k$, la fonction de transfert à une matrice $k \times k$, et la redistribution à une matrice $k \times n$. Le développement du noyau peut donc se comprendre comme la recherche de matrices $A \in \mathbb{C}^{m \times k}, T \in \mathbb{C}^{k \times k}$ et $B \in \mathbb{C}^{n \times k}$ telles que, avec $M|_{\sigma \times \tau}$ le sous-bloc des indices ligne σ et colonne τ de M considéré, et

$|\sigma| := m$ $|\tau| := n$:

$$M|_{\sigma\times\tau} \simeq A.T.B^T$$

La matrice initiale comprend $m \times n$ termes, et les trois matrices A, T et B $(m+n) \times k + k^2$, ce qui permet d'assurer une réduction de la quantité de stockage requise si k est assez petit devant m et n. Par ailleurs, il est clair que le rang du produit $A.T.B^T$ est au plus de k, ce qui souligne la perte d'information de cette compression (sous l'hypothèse que $M|_{\sigma\times\tau}$ est de rang plein), qui appartient à la classe des méthodes avec pertes.

De même que pour le découpage hiérarchique de l'espace, les algorithmes $\mathcal{H}$-Matrices ne dépendent pas explicitement d'une méthode de compression, la dépendance ne se faisant qu'à travers la complexité calculatoire de la méthode de compression retenue, son taux de compression, et la précision de celle-ci.

Une présentation plus approfondie du découpage spatial sera faite dans la suite de ce document lors de la description de l'algorithme multi-niveau. Nous donnerons dans ce chapitre des détails sur les opérations sur les blocs compressés. Le format adopté est un peu différent de la forme $A.T.B^T$, puisque la matrice T n'apparaît pas dans celui-ci. Suivant la littérature [35], nous appellerons ce format $\mathcal{R}k$-Matrice.

2.4 $\mathcal{R}k$-Matrices

On suppose dans cette section l'existence d'un critère dont l'entrée est une ensemble d'indices « ligne » σ et un ensemble d'indices « colonne » τ, et renvoyant une décision binaire, basée sur les supports des fonctions de base associées aux éléments de σ et τ. On note de plus $m = |\sigma|$ et $n = |\tau|$.

Dans le cas où un ensemble $\sigma \times \tau$ est admissible (dans un sens que nous préciserons ultérieurement, pour l'instant ceci signifie que l'on s'autorise à compresser ce bloc), le sous-bloc matriciel associé $M|_{\sigma\times\tau}$ est représenté sous la forme de deux matrices, A et B (voir figure 2.4, telles que :

$$M|_{\sigma\times\tau} \rightarrow A.B^T \tag{2.4.1}$$

avec donc $M|_{\sigma\times\tau} \in \mathbb{C}^{m\times n}$, et nécessairement $A \in \mathbb{C}^{m\times k}$ et $B^T \in \mathbb{C}^{k\times n}$ pour assurer la compatibilité du produit [3].

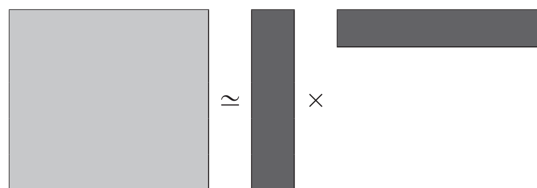


Figure 2.4 – Compression d'un bloc matriciel

La compression du bloc est le problème de la recherche des matrices A et B telles que l'écart en norme entre $A.B^T$ et $M|_{\sigma\times\tau}$ soit le plus faible possible, tout en assurant

3. On note que ces matrices n'ont à priori aucun rapport avec les matrices A et B de la section précédente.

$k < \min(m, n)$. [4] Cette représentation approchée d'une matrice est appelée dans la suite $\mathcal{R}k$-Matrice et son avantage réside dans sa compatibilité avec l'addition et la multiplication matricielle, une $\mathcal{R}k$-Matrice n'étant bien entendu pas inversible car de rang $k < \min(m, n)$.

La question de l'existence d'une bonne approximation sous cette forme se pose. La décomposition en valeurs singulières apporte une première réponse [51]. Nous verrons ultérieurement (section 3.2) d'autres formes d'approximation souvent plus efficaces en pratique.

2.4.1 Décomposition en valeurs singulières

La *Singular Values Decomposition* (SVD) est une factorisation valable pour toute matrice $M \in \mathbb{C}^{m\times n}$, dont la définition est donnée par :

Théorème 2.2 (Décomposition en valeurs singulières)**.** *Soit $M \in \mathbb{C}^{m\times n}$. Il existe une factorisation de M de la forme*

$$M = U\Sigma V^*$$

avec $U \in \mathbb{C}^{m\times m}$ une matrice unitaire, $\Sigma \in \mathbb{C}^{m\times n}$ une matrice dont les coefficients diagonaux sont réels positifs ou nuls et les autres sont nuls (matrice diagonale dans le cas carré) et dont les éléments diagonaux sont appelés valeurs singulières, *et $V^* \in \mathbb{C}^{m\times m}$ la matrice adjointe à V, unitaire.*

Remarque 2.3 (Non unicité)**.** *En ajoutant la contrainte que les valeurs singulières sont ordonnées par ordre décroissant, la matrice Σ est unique, mais pas les matrices U et V.*

Plusieurs liens existent entre la SVD et la décomposition en valeurs propres d'un endomorphisme diagonalisable ; par exemple, les valeurs propres de $M^T M$ sont les carrés des valeurs singulières de M. Cette factorisation est particulièrement intéressante du fait de l'approximation obtenue par la troncature de celle-ci, c'est-à-dire en ne conservant que les k plus grandes valeurs singulières et les vecteurs singuliers à gauche et à droite associés. Précisons le sens de « meilleure approximation » de la SVD [51].

Théorème 2.4. *Soit $A \in \mathbb{C}^{m\times n}$ avec $m \geqslant n$, et $\|\cdot\|$ une norme matricielle unitairement invariante (telle que pour toute matrice U unitaire, $\|UA\| = \|A\|$). La meilleure approximation A_k de rang au plus k de A,*

$$A_k := \arg\min \left\{ \|A - R\| \;\middle|\; R \in \mathbb{C}^{m\times n}, rang(R) \leqslant k \right\}$$

est

$$A_k = \sum_{i=1}^{k} U_i \Sigma_{ii} V_i^*$$

avec $A = U\Sigma V^*$ la décomposition en valeurs singulières de A, et U_i, V_i les vecteurs colonnes de U, V.

4. En réalité, pour assurer une réduction de la quantité de données à conserver il faut assurer $k \times (m + n) < mn$, et pour assurer une réduction des opérations à réaliser, un critère différent est à satisfaire, celui-ci étant dépendant des opérations.

De plus, cette approximation vérifie :

$$\min\left\{\|A-R\|_F^2 \;\mid\; rang(R) \leqslant k\right\} = \sum_{i=k+1}^{n} \sigma_i^2$$

$$\min\left\{\|A-R\|_2 \;\mid\; rang(R) \leqslant k\right\} = \sigma_{k+1}$$

avec $\sigma_i := \Sigma_{ii}$*,* $\|\cdot\|_F$ *la norme de Frobenius et* $\|\cdot\|_2$ *la norme spectrale.*

Ce théorème est important, car il donne une quantification de l'erreur liée à la troncature de la SVD utilisée pour obtenir une $\mathcal{R}k$-Matrice, et donne une borne inférieure pour la norme de Frobenius de l'erreur pour toute approximation de rang faible d'une matrice.

Il implique en particulier que la troncature d'une décomposition en valeurs singulières est la meilleure approximation de rang faible au sens de la norme L^2, et donne naturellement une décomposition d'une matrice sous la forme d'une $\mathcal{R}k$-Matrice en posant

$$A = \tilde{U}\tilde{\Sigma} \quad \text{et} \quad B = \tilde{V}$$

Remarque 2.5 (Considérations numériques)**.** *En pratique, on pourra préférer poser*

$$A = \tilde{U}\sqrt{\tilde{\Sigma}} \quad et \quad B = \sqrt{\tilde{\Sigma}}\tilde{V}$$

où $\sqrt{\tilde{\Sigma}}$ *est la matrice définie par :*

$$\left(\sqrt{\tilde{\Sigma}}\right)_{ii} := \sqrt{\tilde{\Sigma}_{ii}}$$

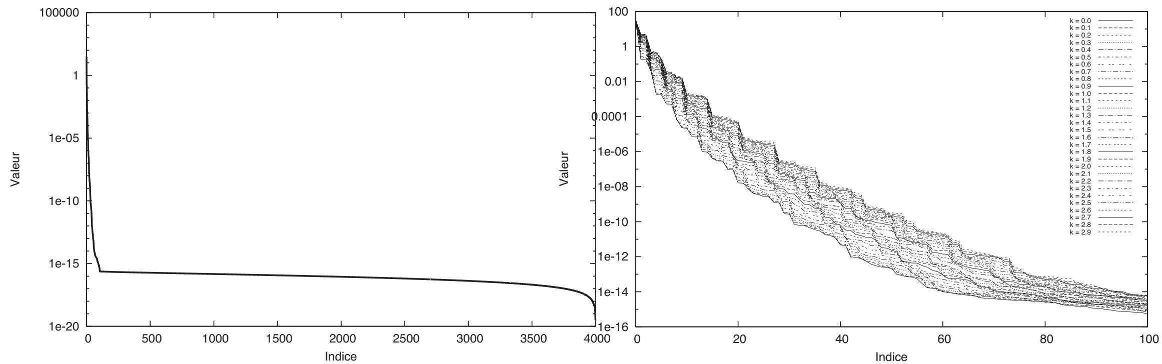
Ceci permet d'obtenir une moins grande dynamique entre les valeurs de A *et de* B*, et donc peut potentiellement éviter des problèmes numériques lors des calculs en précision finie [64].*

Il est donc possible de trouver une bonne approximation d'un bloc de matrice dans l'espace des $\mathcal{R}k$-Matrices, et donc de réduire la quantité de stockage à travers cette compression. Néanmoins, il reste à préciser les opérations possibles sur cette classe de matrices, et les coûts associés.

Remarque 2.6 (Rang et ordre d'approximation)**.** *Dans la suite de ce document, le terme « rang » appliqué à une* $\mathcal{R}k$*-Matrice* $R = AB^T$ *désigne l'ordre d'approximation, c'est-à-dire le nombre de colonnes* k *des matrices* A *et* B*, et non le rang effectif de* R*. Il est clair que* $rang(AB^T) \leqslant k$*, cependant la complexité algorithmique des opérations dépend de* k *et non du vrai rang, ce qui justifie cet abus de terminologie.*

Décroissance des valeurs singulières Le gain de calcul et de stockage des $\mathcal{R}k$-Matrices est lié au taux de compression qu'il est possible d'obtenir pour une précision fixée. Celui-ci est dépendant de la structure de la matrice et de la méthode de compression. Cependant, le théorème 2.4 assure que la meilleure approximation est fournie par la décomposition en valeurs singulière tronquée.

Examinons la décroissance des valeurs singulières d'une matrice « BEM ». On considère la matrice A définie par l'équation (2.2.17) page 45, avec $\{x_i\}$ un ensemble de 4000 points



(a) Toutes les valeurs singulières pour $\kappa = 0$ (b) 100 premières valeurs pour $\kappa = 0 \rightarrow 2.9$

FIGURE 2.5 – Décroissance des valeurs singulières d'un bloc 4000×4000 d'interactions lointaines. Les valeurs de k plus élevées donnent une décroissance plus lente des valeurs singulières, avec la même allure.

issus d'un maillage d'A319 (le même que celui de la figure 3.2), sur le nez de l'avion. L'ensemble $\{y_j\}$ est le même ensemble de points, décalés de 10m selon l'axe Ox. Ce maillage est adapté pour une valeur de k légèrement supérieure à 3 dans le cas d'une discrétisation BEM. La figure 2.5 représente la décroissance des valeurs singulières de ce bloc pour diverses valeurs de k. Dans tous les cas, cette décroissance est très rapide, et les valeurs convergent très vite vers un niveau comparable au bruit numérique [5].

La décroissance est plus lente avec l'augmentation de la valeur de κ, avec l'évolution suivante pour l'ordre d'approximation permettant d'obtenir une erreur inférieure à 10^{-8} en norme de Frobenius :

κ	**Ordre d'approximation**
0	16
1	22
1.5	25
2	28
2.9	28

2.4.2 Algèbre des $\mathcal{R}k$-Matrices

Les $\mathcal{R}k$-Matrices sont la « brique de base » des $\mathcal{H}$-Matrices, et il est donc naturel que leur algèbre soit relativement riche pour permettre une algèbre des $\mathcal{H}$-Matrices autorisant les opérations évoquées au début de ce chapitre. Les opérations nécessaires sont :

— Appliquer une $\mathcal{R}k$-Matrice à un vecteur ;

— Multiplier une $\mathcal{R}k$-Matrice avec une autre $\mathcal{R}k$-Matrice ou une matrice pleine (non compressée) ;

— Additionner une $\mathcal{R}k$-Matrice avec une autre $\mathcal{R}k$-Matrice ou une matrice pleine (non compressée) ;

5. Les calculs étant faits en double précision, ceci correspond à 16 chiffres décimaux au mieux.

et les deux dernières opérations doivent permettre de conserver la structure compressée de la matrice, et ne pas augmenter le rang de manière trop importante. En effet ces opérations vont être effectuées de nombreuses fois lors d'une factorisation, et l'efficacité de la méthode dépend de la conservation du taux de compression au cours du calcul.

La multiplication ne pose pas de problème du point de vue du rang de la matrice, car $rang(M_1M_2) \leqslant \min(rang(M_1), rang(M_2))$, et l'ordre d'approximation sera le cas d'égalité de cette expression. Cependant, pour l'addition, on a $rang(M_1 + M_2) \leqslant rang(M_1) + rang(M_2)$ avec possibilité d'égalité. Prendre pour l'ordre d'approximation le cas d'égalité de cette expression mène à une croissance du rang de la $\mathcal{R}k$-Matrice, ce qui aboutit finalement après plusieurs opérations à un ordre d'approximation plus grand que $\min(m, n)$! Cette opérations sera donc accompagnée d'une recompression, alors que les deux premières seront effectuées sans perte d'information.

2.4.2.1 Produit matrice-vecteur

Soit $R = AB^T$ une $\mathcal{R}k$-Matrice approximant une matrice de $\mathbb{C}^{m\times n}$ et $x \in \mathbb{C}^n$. Le produit $y \leftarrow R.x$ se fait alors naturellement en 2 étapes :

1. $z \leftarrow B^T x$
2. $y \leftarrow Az$

On note que ce produit n'entraîne pas de nouvelle approximation et nécessite $\mathcal{O}((m+n)k)$ opérations.

2.4.2.2 Multiplication

Le cas de la multiplication est le plus simple parmi les opérations matrice-matrice, car une multiplication entre une matrice de rang k et une matrice de rang k' donne une matrice de rang inférieur ou égal à $\min(k, k')$, et donc le taux de compression est conservé. Soit deux matrices $R_k = A.B^T \in \mathbb{C}^{m\times n}$ et $M \in \mathbb{C}^{n\times p}$, alors le produit $R'_k = A'B'^T := R_kM$ est une $\mathcal{R}k$-Matrice de rang k donnée par

$$A' = A \quad \text{et} \quad B' = M^T B$$

On note que le produit n'entraîne pas de nouvelle approximation. C'est une généralisation du produit par un vecteur, mais la forme matricielle sera préférée aux opérations colonne par colonne pour des raisons d'efficacité informatique (opérations BLAS3 contre BLAS2, qui sont plus efficaces sur les machines actuelles). Le nombre d'opérations effectuées par ce produit est $\mathcal{O}(knp)$ pour la multiplication à droite, et $\mathcal{O}(kmp)$ pour la multiplication à gauche (avec dans ce cas $M \in \mathbb{C}^{p\times m}$).

Dans le cas de la multiplication de deux $\mathcal{R}k$-Matrices, $R_1 = A_1B_1^T \in (\mathbb{C}^{m\times k_1} \times \mathbb{C}^{k_1\times p})$ et $R_2 = A_2B_2^T \in (\mathbb{C}^{p\times k_2} \times \mathbb{C}^{k_2\times n})$, il est possible de procéder de deux façons car

$$R_1.R_2 = (A_1)(B_1^T R_2) = (R_1A_2)(B_2^T)$$

qui nécessitent dans les deux cas la multiplication d'une $\mathcal{R}k$-Matrice par une matrice pleine à gauche ou à droite, l'autre terme de la $\mathcal{R}k$-Matrice ne nécessitant pas d'opérations arithmétiques puisque les deux termes sont stockés sous forme décomposée.

La première écriture nécessite $\mathcal{O}(k_1 k_2(n+p))$ opérations et donne une $\mathcal{R}k$-Matrice d'ordre d'approximation k_1, et la seconde $\mathcal{O}(k_1 k_2(m+p))$ opérations et donne une $\mathcal{R}k$-Matrice d'ordre k_2, illustrant au passage la différence entre rang effectif et ordre d'approximation. Il est possible de choisir entre les deux options, ce choix étant un compromis à trouver entre la recherche d'un rang plus faible et l'économie des opérations arithmétiques.

2.4.2.3 Addition

Le cas de l'addition est plus délicat, dans le sens où l'addition de deux matrices mène en règle générale à l'augmentation du rang, le rang de la somme d'une matrice de rang k_1 et d'une matrice de rang k_2 étant $k_3 \leqslant k_1 + k_2$. Nous présentons dans la suite le cas de l'addition de deux $\mathcal{R}k$-Matrices ; dans le cas où une seule des matrices est compressée, l'addition se fait par conversion préalable d'une des matrices en matrice pleine (en calculant $A.B^T$) ou en matrice compressée, selon l'opération la moins coûteuse en mémoire.

Soit $R_1 = A_1 B_1^T$ et $R_2 = A_2 B_2^T$ deux $\mathcal{R}k$-Matrices de rang k_1 et k_2. La somme R_3 est formée par la juxtaposition des colonnes des matrices A_1 et A_2 d'une part, et B_1 et B_2 d'autre part, et est donc d'ordre $k_3 := k_1 + k_2$:

$$A_3 = [A_1 \ A_2] \quad \text{et} \quad B_3 = [B_1 \ B_2]$$

Cette croissance de l'ordre est problématique. Dans la plupart des cas, une approximation de la somme peut être trouvée avec une précision suffisante avec un ordre proche de k_1 et k_2. On procède donc à une recompression de la matrice afin de déterminer un nouvel ordre $k_3' \leqslant k_3 = k_1 + k_2$, qui est une variante de la SVD adaptée aux matrices de rang faible. Cette compression se fait à l'aide de deux décompositions QR et d'une SVD [51]. Toute matrice M (non nécessairement carrée) peut se factoriser sous la forme $M = QR$, avec Q une matrice orthogonale et R une matrice triangulaire supérieure.

Remarque 2.7 (Décomposition QR réduite)**.** *Dans le cas où la matrice $M \in \mathbb{C}^{m\times n}$ dont on cherche la décomposition QR n'est pas carrée ($m \neq n$), cette décomposition a la forme suivante :*

$$A = QR = Q\begin{pmatrix} R_1 \\ 0 \end{pmatrix} = (Q_1 \ Q_2)\begin{pmatrix} R_1 \\ 0 \end{pmatrix} = Q_1 R_1$$

avec $Q_1 \in \mathbb{C}^{m\times n}$ et $R_1 \in \mathbb{C}^{n\times n}$. On qualifie cette décomposition $Q_1 R_1$ de « réduite » [51], et c'est celle que nous considérons dans ce document.

Dans le cas qui nous intéresse, avec une matrice de taille $m \times k$, $m \geqslant k$, la matrice Q est de taille $m \times k$ et R de taille $k \times k$.

On procède de la manière suivante pour une $\mathcal{R}k$-Matrice $R = AB^T$:

1. Calcul des décompositions QR réduites de A et B :

$$A = Q_A R_A \quad \text{et} \quad B = Q_B R_B$$

2. Produit des matrices R : $X := R_A R_B^T$
3. Décomposition en valeurs singulières de $X = U\Sigma V^*$ et troncature de celle-ci pour donner $\tilde{X} := \tilde{U}\tilde{\Sigma}\tilde{V}^*$ en ne gardant que les vecteurs correspondant aux k' plus grandes

valeurs singulières. La troncature est effectuée selon un critère de rang maximal k', ou en accord avec une tolérance $\varepsilon > 0$ fixée, suivant les résultats du théorème 2.4.

4. Multiplication des vecteurs singuliers à gauche et à droite pour obtenir $R' = A'.B'^T$

$$A' := Q_A U \sqrt{\tilde{\Sigma}} \quad \text{et} \quad B' := Q_B V^* \sqrt{\tilde{\Sigma}}$$

ce qui peut se résumer par les relations suivantes :

$$\begin{aligned} A.B^T &= (Q_A R_A).(Q_B.R_B)^T \\ &= Q_A(R_A R_B^T)Q_B^T \\ &= Q_A(U\Sigma V^*)Q_B^T \\ &\simeq Q_A(\tilde{U}\tilde{\Sigma}\tilde{V}^*)Q_B^T \\ &= (Q_A\tilde{U}\sqrt{\tilde{\Sigma}})(\sqrt{\tilde{\Sigma}}\tilde{V}^* Q_B^T) \end{aligned}$$

L'étape de perte d'information (et qui assure donc la compression) est la troisième, et elle est réalisée avec une SVD, ce qui est important du fait des bonnes propriétés d'approximations de la SVD tronquée vues à la section 2.4.1.

Le compte des opérations est le suivant

Factorisation QR de A	$\mathcal{O}(mk^2)$
Factorisation QR de B	$\mathcal{O}(nk^2)$
Multiplication $R_A R_B^T$	$\mathcal{O}(k^3)$
SVD tronquée de $R_A R_B^T$	$\mathcal{O}(k^3)$
Multiplication $Q_A U\sqrt{\tilde{\Sigma}}$	$\mathcal{O}(mkk')$
Multiplication $Q_B V\sqrt{\tilde{\Sigma}}$	$\mathcal{O}(nkk')$
Total (pour $k' = \mathcal{O}(k)$)	$\mathcal{O}((m+n)k^2 + k^3)$

Dans ce tableau, on considère que k' est du même ordre de grandeur que k, ce qui est vrai pour la classe de matrices qui nous intéresse en pratique. De plus, par rapport aux estimations de complexité données dans [35], nous considérons ici que k n'est pas négligeable devant m et n, d'où la présence du terme en $\mathcal{O}(k^3)$ (ce qui est confirmé par l'expérience, *cf.* section 3.6.2.2). Il faut également noter que tous les calculs n'ont pas les mêmes constantes associées, et la SVD est très coûteuse. Elle est néanmoins faite sur une matrice de petite taille ($k \times k$), ce qui rend le calcul raisonnable.

On obtient alors en pratique une faible variation du rang au cours d'un algorithme sur la matrice complète, au prix d'une perte de précision à chaque recompression, cette perte étant contrôlée par le seuil de troncature de la décomposition.

Remarque 2.8. *Dans le cas où $k > \min(m,n)$, il est moins coûteux de procéder de la manière suivante :*

— *Conversion de la $\mathcal{R}k$-Matrice $A.B^T$ en matrice pleine F ;*

— *Approximation de F par une $\mathcal{R}k$-Matrice.*

La première étape a un coût $\mathcal{O}(mnk)$, la seconde est en $\mathcal{O}(\min(m,n)^2 \max(m,n))$, ce qui donne un coût total de $\mathcal{O}(\min(m,n)\max(m,n)k)$, soit $\mathcal{O}(mnk)$.

Remarque 2.9 (R-SVD)**.** *La recompression adaptative d'une $\mathcal{R}k$-Matrice est à rapprocher de l'algorithme R-SVD de calcul de la décomposition en valeurs singulières [51, chapitre 5]. En effet, soit $A \in \mathbb{C}^{m\times n}$, avec $m > n$. Le calcul de la décomposition en valeurs singulières par R-SVD comporte les étapes suivantes :*

1. *Calcul de la décomposition QR de A, soit la recherche d'une matrice orthogonale $Q \in \mathbb{C}^{m\times m}$ telle que :*
$$Q^T A = \begin{pmatrix} R_1 \\ 0 \end{pmatrix}$$
avec $R_1 \in \mathbb{C}^{n\times n}$ triangulaire supérieure.
2. *Bidiagonalisation de R_1*
$$U_R^T R_1 V = B_1$$
avec U_R et V matrices orthogonales de taille $n \times n$, et $B_1 \in \mathbb{C}^{n\times n}$ une matrice bidiagonale supérieure.

La suite de l'algorithme (après la bidiagonalisation) est similaire au calcul « classique » de la décomposition en valeurs singulières.

2.5 Algorithme mono-niveau

Nous venons d'exposer les grandes lignes des opérations sur les blocs compressés, mais n'avons pas encore abordé en détail le caractère hiérarchique donnant son nom aux $\mathcal{H}$-Matrices. Il n'est cependant pas nécessaire d'exploiter une structure hiérarchique pour obtenir un solveur plus rapide que $\mathcal{O}(N^2)$. De même que le développement multipôlaire peut être mis à profit directement pour obtenir un produit matrice-vecteur rapide, il est possible de faire la même chose avec les $\mathcal{R}k$-Matrices, une fois qu'une méthode de compression est disponible. C'est cet algorithme non hiérarchique que nous présentons dans cette section, qui est un cas dégénéré de ceux présentés ultérieurement.

Pour le fonctionnement de cet algorithme, on procède à une renumérotation des degrés de liberté telle que la proximité de deux indices i et j signifie que les supports des fonctions de base associées à ces degrés de liberté sont proches spatialement. Il existe plusieurs manières de procéder à une telle renumérotation, que nous ne détaillerons pas car elles ne seront pas utilisées dans la suite. Par exemple, il est possible de diviser l'espace en une grille régulière comprenant la boîte englobante de l'objet considéré. Un bloc de matrice représente alors les interactions entre l'intersection de l'objet Γ avec deux subdivisions de la grille. Suivant leur distance relative, on décide alors d'appliquer ou non l'algorithme de compression à ce bloc matriciel. Un exemple d'un tel découpage est donné par la figure 2.6 dans le cas de la dimension 2.

Une fois cette renumérotation effectuée, la matrice est découpée en blocs de taille régulière ou tout du moins compatible pour les opérations suivantes. En fonction de la décision rendue par le critère d'admissibilité, le bloc est alors compressé ou conservé sous forme pleine.

Du fait de la relation entre la numérotation et la proximité spatiale, et du fait que des blocs suffisamment éloignés sont admissibles, les blocs loin de la diagonale de la matrice seront compressés, alors que les blocs comprenant la diagonale de la matrice ne le seront pas, puisque la distance vaut 0 dans ce cas.

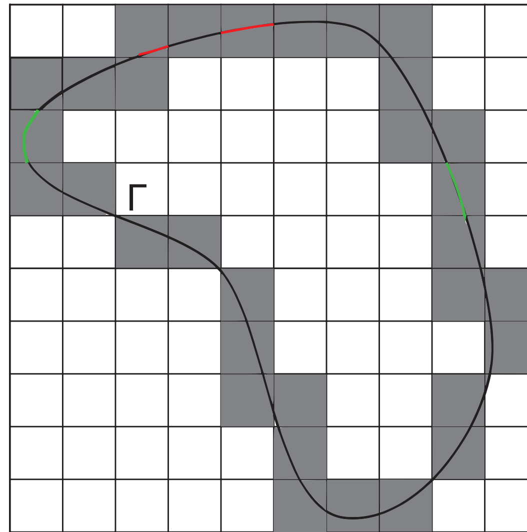


Figure 2.6 – Exemple de partition de l'espace. Les parties rouges représentent des interactions non compressées, les parties vertes des interactions compressées car suffisamment distantes. Les cellules grisées représentent les groupes non vides.

On a alors les bases nécessaires pour effectuer des opérations, de façon similaire aux algorithmes par blocs sur les matrices, puisque les tailles de bloc sont compatibles. Soit une matrice $M \in \mathbb{C}^{n \times n}$, renumérotée et divisée en n_b blocs ligne et colonne. Il faut remarquer que l'algorithme usuel de factorisation LU (sans pivotage) d'une matrice peut aussi être appliqué si les éléments de la matrice sont remplacés par des sous-blocs de celle-ci, et que seuls les éléments (respectivement blocs) diagonaux sont inversés, ce qui n'est pas un problème. En effet, si les blocs compressés ne sont pas inversibles car de rang non plein (et non nécessairement carrés), ce n'est pas le cas pour les blocs diagonaux qui sont systématiquement pleins. Comme ceux-ci sont inversibles, cela assure la compatibilité avec toutes les opérations nécessaires pour réaliser un solveur direct (factorisation LU ou LDL^t, descente et remontée pour la résolution des systèmes triangulaires résultant de la factorisation).

On a alors l'algorithme 2.1 pour la factorisation LU sur place pour une matrice compressée. Dans cet algorithme, les opérations impliquant une $\mathcal{R}k$-Matrice et une matrice

Algorithme 2.1 Factorisation LU par bloc pour une matrice compressée

for all $k = 1, \ldots, n_b - 1$ **do** $A_{kk} \leftarrow A_{kk}^{-1}$ ▷ Inversion pleine
  for all $i = k+1, \ldots, n_b$ **do** $A_{ik} \leftarrow A_{ik} A_{kk}^{-1}$ ▷ Plein ou compressé
    for all $j = k+1, \ldots, n_b$ **do** $A_{ij} \leftarrow A_{ij} - A_{ik} A_{kj}$ ▷ Plein ou compressé
    end for
  end for
end for

non compressée se font après conversion d'un des termes dans le format de l'autre, en fonction du format de la matrice dans laquelle le résultat sera rangé.

Cet algorithme, associé à une méthodes de compression adaptée comme par exemple l'approximation adaptative croisée (*Adaptative Cross Approximation*) [24] dont il sera

question dans le chapitre suivant, permet d'obtenir dans certains cas un solveur direct de complexité asymptotique $\mathcal{O}(N^{3/2})$, à comparer à $\mathcal{O}(N^3)$ pour un solveur direct et à $\mathcal{O}(n_{iter}N^2)$ pour un solveur itératif pour lequel la convergence est atteinte en n_{iter}. On remarque que cette complexité est la même que celle de la FMM mono-niveau, ce qui est raisonnable puisque les idées sous-jacentes sont fortement similaires. Bien que l'opération présentée ici soit une factorisation approchée au lieu d'une application d'un opérateur approché, la similarité se retrouvera dans le chapitre suivant. En effet, le passage d'un algorithme mono-niveau à un algorithme hiérarchique permettra de réduire la complexité de la factorisation, tout comme dans le cas de la FMM.

2.6 Algorithme hiérarchique

Comme dans le cas de la méthode multipôle, la réduction de temps de calcul apportée par une méthode mono-niveau est limitée par rapport à une approche multi-niveaux. En effet, les gains apportés par la compression seront d'autant plus grands que deux blocs sont éloignés l'un de l'autre et de grande taille, mais dans l'algorithme précédent celui-ci est de taille identique pour tous les blocs. Dans ce cas, il est naturel d'augmenter la taille (en indices) des groupes, mais dans ce cas une fraction croissante de la matrice devient non admissible, du fait de la trop grande proximité des blocs par rapport à leurs tailles (géométriques) relatives (*cf* le critère de l'équation 3.1.2).

Une solution à ce problème est d'utiliser une approche hiérarchique. Si un couple de blocs n'est pas admissible, les diviser en espérant que certains des sous-blocs le soient, et recommencer ce processus hiérarchiquement jusqu'à atteindre soit une admissibilité, soit un critère d'arrêt lié à une taille minimale de bloc.

Une fois de plus, l'idée est à rapprocher de celle de la FMM qui traite les interactions au niveau le plus grossier auquel celles-ci sont admissibles, et descend hiérarchiquement dans le cas contraire. La différence importante à noter ici est le caractère explicite de cette hiérarchie, et son lien avec la matrice.

On obtient alors une matrice découpée hiérarchiquement, et aux deux cas des blocs $\mathcal{R}k$-Matrice et pleins se rajoute un troisième cas à considérer dans les algorithmes, celui de la matrice subdivisée, appelée $\mathcal{H}$-Matrice.

Dans ce cas, le découpage des degrés de liberté n'est plus régulier, et doit donc respecter des conditions de compatibilité afin de permettre d'effectuer les mêmes opérations. Une autre conséquence de ce découpage hiérarchique est le changement des algorithmes, qui vont naturellement s'exprimer sous une forme récursive, leur structure épousant celle de la matrice.

Les étapes sont alors, en supposant ici pour simplifier que les degrés de liberté ligne et colonne sont les mêmes (et donc que la matrice est carrée) :

Découpage hiérarchique de l'espace Il peut se faire de plusieurs manières, les plus courantes étant basées sur l'utilisation de deux arbres $\mathcal{T}$ et $\mathcal{T}'$ associés aux degrés de liberté ligne et colonne, ces arbres pouvant être soit des arbres binaires résultant du découpage d'une boîte englobante selon un de ses axes à chaque niveau, ou des arbres d'arité 8, appelés *octrees*. Un critère d'arrêt est utilisé pour ce découpage,

celui-ci étant soit géométrique (comme dans certaines formulations FMM) soit guidé par le nombre de degrés de liberté portés par une feuille.

Découpage hiérarchique de la matrice Le découpage spatial donne une structure aux degrés de liberté ligne et colonne, et un bloc de la matrice est représenté par un couple de nœuds de $\mathcal{T} \times \mathcal{T}'$, donc le découpage de la matrice est un sous-ensemble du produit cartésien $\mathcal{T} \times \mathcal{T}'$. L'algorithme permettant d'établir ce découpage part de la racine de $\mathcal{T} \times \mathcal{T}'$. Si un élément $(\tau, \sigma) \in \mathcal{T} \times \mathcal{T}'$ est non admissible, on continue le procédé de découpage sur les fils de σ et τ. Dans le cas contraire, le bloc est admissible et sera compressé. Il peut aussi arriver que σ ou τ soit une feuille de $\mathcal{T}$, et le découpage ne peut plus se poursuivre. L'algorithme s'arrête alors en notant que le bloc courant n'est pas admissible. Il ne sera jamais compressé, mais est dans ce cas nécessairement de petite taille dans au moins une de ses dimensions, puisque les feuilles de $\mathcal{T}$ sont de petite taille.

Compression Il s'agit simplement de l'énumération des blocs de la $\mathcal{H}$-Matrice, et de la compression des blocs admissibles par un algorithme de type SVD, ou une autre variante moins coûteuse, ce qui sera précisé dans la suite.

Opérations sur la $\mathcal{H}$-Matrice Elles sont guidées par la structure de la matrice, à la fois dans leur écriture et dans leur coût en calcul. Dans le cas d'une matrice issue d'un découpage spatial par arbre binaire, la matrice a formellement la structure d'une matrice 2×2 par bloc, chaque bloc étant potentiellement lui-même subdivisé. Les algorithmes sont alors similaires à ceux utilisés sur les matrices 2x2 par bloc, avec un cas de récursion pour les opérations sur les blocs.

Ce sont ces diverses étapes que nous détaillerons dans le chapitre suivant, et dont nous donnerons une implémentation parallèle dans le chapitre 4.

Chapitre 3

Algorithmes séquentiels

Sommaire

3.1 Découpage spatial 65
3.2 Compression 76
3.3 Opérations sur les $\mathcal{H}$-Matrices 87
3.4 Complexité 106
3.5 Implémentation et premiers résultats 109
3.6 Analyse et optimisations 132
3.7 Conclusion 142

À LA FIN du chapitre précédent, nous avons dessiné les grandes lignes des algorithmes permettant de construire et de manipuler une $\mathcal{H}$-Matrice. Ce chapitre en donne une description plus approfondie, et suffisante pour implémenter ces algorithmes. Les estimations de complexité seront données, mais l'ensemble des détails des preuves ne sera pas exposé, ceci n'étant pas le point focal de ce travail.

Ce chapitre n'étant pas spécifique aux BEM pour l'équation des ondes, nous prendrons comme point de départ la connaissance de la localisation spatiale des degrés de liberté et la possibilité de calculer un terme arbitraire de la matrice d'interaction. Dans le cas de l'électromagnétisme, on prendra comme approximation de la position spatiale d'un degré de liberté le milieu de l'arête le portant, et le calcul d'un terme de la matrice sera fait de manière « boîte noire ».

3.1 Découpage spatial

La connaissance de la localisation spatiale des degrés de liberté se traduit par la donnée d'un nuage de points $\{x_i \in \mathbb{R}^3 \; | i = 1, \ldots, N \}$. L'objectif du découpage spatial est de regrouper les points proches dans le même groupe (ou *cluster* en anglais), de manière hiérarchique. Il existe diverses façons d'obtenir un découpage respectant ces contraintes, une méthode fréquente et simple est basée sur un arbre ; c'est celle que nous présentons ici, et qui sera employée dans la suite.

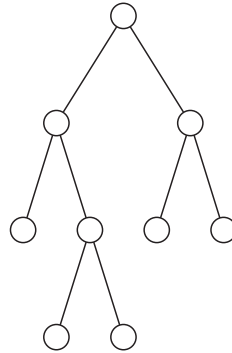


FIGURE 3.1 – Exemple d'arbre binaire entier

3.1.1 Arbres

Formellement, un arbre est un cas particulier de graphe acyclique orienté n'ayant qu'une seule source, appelée racine, et tel que tous les sommets, sauf la racine, n'aient qu'un seul père [67].

Précisons cette définition :

- Un **graphe** est un ensemble de sommets $(s_i)_{i=1,\dots,n}$ reliés par un ensemble d'arêtes $A \subset \{1, \dots, n\}^2$ mettant en relation deux sommets.
- Un graphe **orienté** associe à ces arêtes un sens, la relation entre deux sommets représentée par une arête devenant alors à sens unique. Dans la relation $s_1 \rightarrow s_2$, s_1 est un prédécesseur de s_2, et s_2 est un successeur de s_1. Dans la suite, on appellera tout prédécesseur **père**, et tout successeur **fils**.
- Un sommet n'ayant pas de père est appelé **source**.
- Lorsqu'un graphe ne comporte qu'une seule source, celle-ci est appelée **racine**.
- Un graphe **acyclique** est un graphe dans lequel il n'existe pas de chemin suivant les arêtes (dans leur sens de parcours autorisé) ayant les mêmes points de départ et d'arrivée.

Donnons un peu de vocabulaire et de notations :

- Tout sommet d'un arbre sera appelé **nœud** ;
- Tout nœud n'ayant pas de fils est appelé **feuille** ;
- On note $S(s)$ l'ensemble des fils du nœud s. $S(s) = \emptyset$ pour une feuille.
- On appelle **hauteur** d'un nœud la longueur du plus long chemin allant de ce nœud aux feuille de l'arbre dont il est la racine. La hauteur d'une feuille est 0.
- On appelle arbre complet, un arbre dont toutes les feuilles sont à la même distance de la racine.

Un arbre peut avoir un nombre de fils maximal par nœud ; ce nombre est appelé **arité**. Un arbre d'arité 2 est appelé arbre binaire, *quadtree* pour une arité 4 et *octree* pour une arité 8. Dans la suite de ce document, nous rencontrerons principalement des arbre binaires pour le découpage spatial et des *quadtrees* pour représenter une $\mathcal{H}$-Matrice construite sur deux arbres binaires, contrairement à la FMM qui opère usuellement sur des *octrees*.

Un exemple d'arbre binaire est donné par la figure 3.1. On remarque que toutes les branches de l'arbre n'ont pas la même longueur. Cet arbre est un cas particulier puisque tout nœud est soit une feuille, soit père de deux fils. Un tel arbre est dit **entier**.

3.1.2 Algorithme

On part d'un groupe $\sigma = I$, racine contenant tous les points. Ce groupe peut être représenté par sa boîte englobante, un pavé parallèle aux axes de coordonnées, caractérisé par ses deux points extrémaux x_{min} et $x_{max} \in \mathbb{R}^3$.

Tant que le nombre de points contenu dans ce pavé est supérieur à une limite N_{leaf} à fixer, il est divisé récursivement. On aboutit naturellement à une structure arborescente, dont l'arité est déterminée par le nombre de subdivisions et la structure par le choix de celles-ci. Concrètement, les choix les plus fréquents sont une division en 8, ce qui donne un *octree*, ou une bisection, ce qui donne un arbre binaire, suivant l'algorithme 3.1. Cet arbre est nommé « arbre de groupes » (*Cluster Tree*) dans la littérature, quel que soit le découpage adopté.

Algorithme 3.1 Création de l'arbre de groupes.

```
function CreateClusterTree(σ)
    if |σ| ⩽ N_leaf then
        S(σ) = ∅                                   ▷ Rien à faire
    else
        (σ1, σ2) ← Split(σ)
        CreateClusterTree(σ1)
        CreateClusterTree(σ2)
        S(σ) = {σ1, σ2}
    end if
end function
```

Le choix le plus courant est un arbre binaire, avec la division des nœuds par un plan orthogonal à l'axe de coordonnée selon lequel la boîte englobante a la plus grande projection, c'est-à-dire pour $x = (x^1, x^2, x^3) \in \mathbb{R}^3$:

$$i^* = \underset{i=1,2,3}{\arg\max}(x^i_{max} - x^i_{min})$$

3.1.3 Position du plan de séparation

Une fois l'axe i^* déterminé, il faut positionner le plan de séparation. Deux choix sont courants :

— Diviser la boîte de façon à ce que le nombre de points de part et d'autre de la division soit égal. Nous appellerons ce choix le découpage **médian**.

$$x^{i^*}_{sep} = mediane\left(\left(x^{i^*}_i\right)_{i=1,\dots,|\sigma|}\right)$$

— Diviser la boîte au milieu. Nous appellerons ce choix le découpage **géométrique**.

$$x_{sep}^{i*} = \frac{1}{2}\left(\max_{j=1,\ldots,|\sigma|} x_j^{i*} - \min_{j=1,\ldots,|\sigma|} x_j^{i*}\right)$$

On remarque que dans les deux cas, le placement du plan de séparation assure qu'aucune des boîtes filles n'est vide ; l'arbre résultant est donc nécessairement entier.

Le premier choix garantit que le nombre de degrés de liberté recouvert par un nœud de l'arbre soit divisé par deux à chaque niveau, et donne donc un arbre de hauteur minimale. Le second assure, lui, que le volume des boîtes englobantes des nœuds soit divisé par au moins deux à chaque niveau, et permet donc une décroissance rapide du volume.

Ces deux découpages sont très différents : dans le premier cas, il est possible d'avoir un arbre avec des feuilles dont la boîte englobante est très grande, et dans le second il est possible d'avoir un arbre très profond, ces deux situations étant assez naturellement défavorables. Le premier cas garantit également que toutes les boîtes d'un même niveau de l'arbre contiennent le même nombre de points, à un près.

Les figures 3.2 et 3.3 illustrent respectivement le découpage médian et géométrique. Dans les deux cas, l'objet considéré est un A319 dont le maillage comporte 112 991 degrés de liberté [1], avec $N_{leaf} = 100$. Les figures montrent les boîtes englobantes correspondant aux niveaux successifs de l'arbre, en commençant par la racine. Les boîtes englobantes épousent la géométrie, ce qui est lié au fait que leur taille effective est recalculée à chaque niveau. Ceci est important car le critère d'admissibilité utilisé pour la compression dépend largement de la taille des boîtes englobantes ; il est donc inutile et néfaste de considérer des boîtes plus grandes que nécessaire.

Concernant la division médiane, on observe la subsistance de deux boîtes de grande taille même dans les niveaux bas de l'arbre. Ceci est une conséquence malheureuse des caractéristiques de cette géométrie : en effet, cette boîte comprend un faible nombre de degrés de liberté localisés sur les extrémités des ailes, avec la majorité de ceux-ci sur l'empennage arrière de l'avion.

Ce problème ne se présente pas pour le découpage géométrique. Cependant, dans ce cas, la hauteur de l'arbre n'est pas la même dans toutes les parties de celui-ci. On constate en effet que les boîtes ne recouvrent pas tout l'avion à partir du niveau 11, car certaines parties de l'arbre sont moins profondes. De plus, pour des raisons de place, les niveaux les plus bas ne sont pas représentés pour ce découpage, la hauteur de l'arbre étant de 16 pour le découpage géométrique contre 12 pour le découpage médian.

La différence entre les deux méthodes est illustrée par la figure 3.4 dans le cas d'un cône attaché à une demi-sphère à 9852 degrés de liberté (l'arbre du cas précédent étant trop grand pour être représenté aisément), avec $N_{leaf} = 100$. On remarque que l'arbre médian est complet, alors que l'arbre géométrique est plus irrégulier. Cet arbre est plus « lourd » à gauche, ce qui n'est pas une caractéristique générale, mais une conséquence de la géométrie ayant plus de degrés de liberté sur la partie gauche d'abscisse plus faible.

1. Le radôme est manquant car cette géométrie est utilisée pour une simulation électromagmétique, et il est transparent aux ondes.

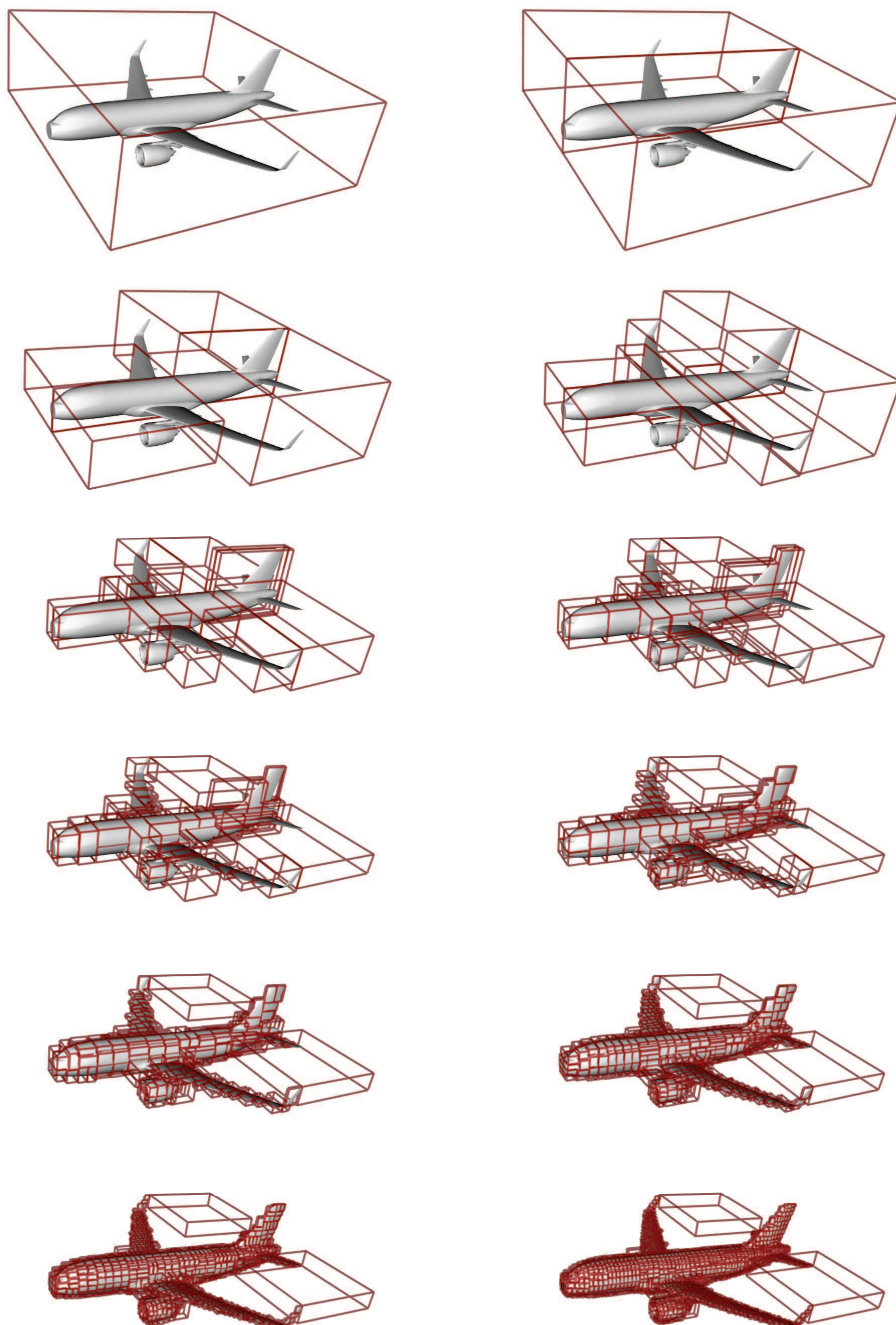
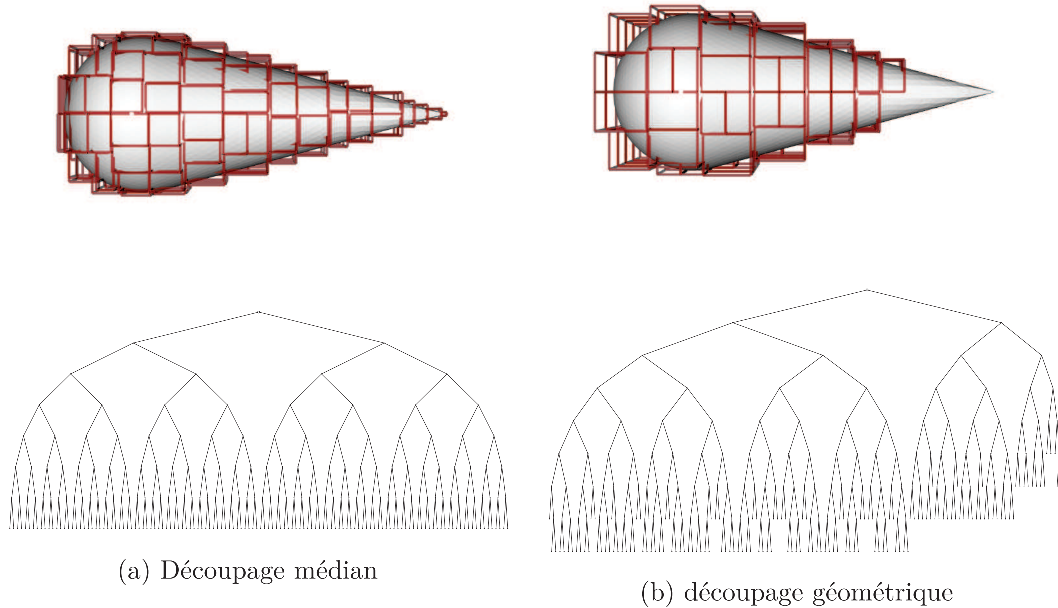


Figure 3.2 – Découpage médian. Le sens de lecture est de gauche à droite, de haut en bas.

Figure 3.3 – Découpage géométrique



(a) Découpage médian

(b) découpage géométrique

Figure 3.4 – Le niveau 7 et l'intégralité des arbres pour deux découpages.

Complexité Soit un nuage de N points. Dans le cas du découpage médian, l'arbre a une profondeur au plus égale à la profondeur d'un arbre binaire complet plus un, et cette hauteur vérifie donc :

$$h(N) \leqslant \lfloor \log_2(N) \rfloor + 2$$

et à chaque niveau de l'arbre, il faut N opérations pour déterminer la position du plan de division, d'où une complexité de construction de l'arbre en $\mathcal{O}(N \log_2(N))$.

Dans le cas de la division géométrique, il n'est pas possible de donner une borne meilleure que $\mathcal{O}(N^2)$ pour le pire cas (l'arbre est dans ce cas un « peigne »), mais le cas moyen est toujours $\mathcal{O}(N \log_2(N))$. Cette situation correspond à un arbre pour lequel chaque division laisse deux boîtes, une des deux ne contenant qu'un seul point. L'arbre est alors de hauteur N, et la quantité de travail à chaque niveau est toujours $\mathcal{O}(N)$, d'où cette estimation.

3.1.4 Arbre de groupes

Le découpage spatial effectué par l'arbre de groupes n'est que la première étape du découpage, l'objectif final de celui-ci étant de construire un arbre de blocs. Supposons dans cette section que la matrice M est carrée ($M = N$) et qu'elle s'appuie sur le même découpage pour les degrés de liberté lignes et colonnes qui seront confondus ($I = J$).

3.1.4.1 Sous-bloc et numérotation

Soient σ et τ deux nœuds de l'arbre de groupes $\mathcal{T}_I$, respectivement « ligne » et « colonne ». σ (respectivement τ) représente un sous-ensemble des degrés de liberté, et donc des lignes (respectivement colonnes) de la matrice. Le couple (σ, τ) représente ainsi le croisement de ces lignes et colonnes, soit un sous-ensemble de la matrice M (les croisements sombres sur la figure 3.5). Cette intersection peut se mettre sous la forme d'un bloc matriciel de taille $|\sigma| \times |\tau|$, qui est le point de vue adopté dans toute la suite de ce document.

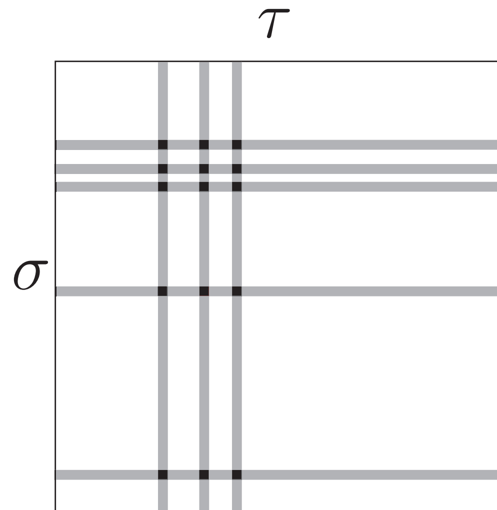


FIGURE 3.5 – Représententation de σ, τ et (σ, τ) dans la matrice M. Dans ce cas, $|\sigma| = 5$ et $|\tau| = 3$, donc $M|_{\sigma\times\tau} \in \mathbb{C}^{5\times3}$.

Remarque 3.1. *Il est possible d'utiliser la construction de l'arbre pour déterminer une bijection $\pi : [1, N] \to [1, N]$ renumérotant les degrés de liberté, de façon à ce que les éléments d'un bloc soient contigus dans la matrice renumérotée. C'est sur cette matrice renumérotée que toutes les opérations suivantes auront lieu, et nous confondrons dans la suite $\sigma \times \tau$ avec un bloc contigu de celle-ci.*

3.1.4.2 Construction de l'arbre de blocs

L'arbre de blocs est un arbre d'arité 4 (pour un découpage binaire) dont les nœuds portent un couple (σ, τ) de nœuds de $\mathcal{T}$. Chaque élément d'un arbre de blocs est identifié à un bloc de la matrice M, d'où son nom. Sa construction repose sur le critère d'admissibilité.

L'objectif de la construction de l'arbre de blocs est de déterminer les blocs les plus gros possibles de la matrice M pour lesquels on autorise la compression. Cette notion d'autorisation est floue. Elle repose en fait sur un critère d'admissibilité renvoyant une décision binaire $f : (\sigma \times \tau) \in \mathcal{T} \times \mathcal{T} \to \{0, 1\}$.

3.1.4.2.1 Critère d'admissibilité Le choix de ce critère est bien entendu intimement lié au choix de la méthode de compression et à la nature de la matrice à compresser.

Prenons le cas où la matrice M est issue d'une discrétisation par éléments finis de frontière d'une équation dont le noyau g est asymptotiquement lisse.

Définition 3.2 (Noyau asymptotiquement lisse). *Soit g un noyau utilisé dans une discrétisation par éléments finis de frontière. Il est dit asymptotiquement lisse si il existe des constantes c_1 et $c_2 \in \mathbb{R}$ et un degré de singularité $\sigma \in \mathbb{N}$ tels que pour toute composante $z \in \{x_j, y_j\}$ et $n \in \mathbb{N}$, l'inégalité*

$$\forall x, y \in \mathbb{R}^3,\ |\partial_z^n g(x,y)| \leqslant n! c_1 (c_2 \|x-y\|)^{-n-\sigma} \tag{3.1.1}$$

soit vérifiée.

Dans ce cas, il existe [22, 32, 63] une constante $\eta > 0$ telle que le critère suivant est adapté :

$$f(\sigma,\tau) = \begin{cases} 1 & \text{si} \quad \min(diam(\sigma), diam(\tau)) < \eta \cdot d(\sigma,\tau) \\ 0 & \text{sinon} \end{cases} \tag{3.1.2}$$

Remarque 3.3 (Attention). *La condition sur le noyau n'est pas vérifiée pour le noyau de l'équation de Helmholtz pour un nombre d'onde k quelconque ! En effet dans le cas du noyau de Green de l'équation de Helmholtz en dimension 3, on a [21] :*

$$\left|\frac{\partial^n G(x,y)}{\partial z}\right| \leqslant n! c_1 (1 + k\,\|x-y\|)^n (c_2 \|x-y\|)^{-n-\sigma}$$

Le noyau de l'équation des ondes ne vérifie pas la condition (3.1.1) *pour $k \neq 0$. Nous utiliserons néanmoins le critère d'admissibilité* (3.1.2) *dans la suite.*

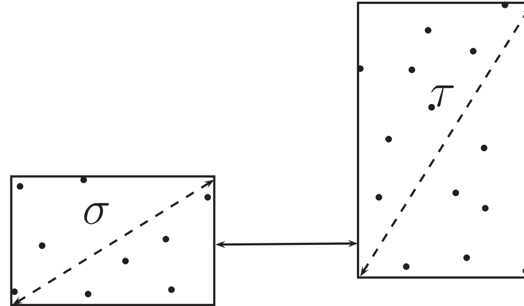


Figure 3.6 – Illustration du calcul du critère d'admissibilité.

3.1.4.2.2 Calcul du critère Le calcul du diamètre d'un groupe σ étant une opération coûteuse en dimension supérieure à 1, nous le remplacerons par le diamètre de sa boîte englobante, qui en est un majorant. De même, la distance sera remplacée par un minorant, la distance des faces les plus proches des boîtes englobantes.

Finalement, on aboutit aux formules suivantes, pour $x = (x^1, x^2, x^3) \in \mathbb{R}^3$:

$$diam(\sigma) \leqslant \sqrt{\sum_{i=1}^{3} (x^i_{max} - x^i_{min})^2} \tag{3.1.3}$$

et

$$d(\sigma,\tau) \geqslant \sqrt{\sum_{i=1}^{3} \left(\max(0, x^i_{min,\tau} - x^i_{max,\sigma})\right)^2 + \left(\max(0, x^i_{min,\sigma} - x^i_{max,\tau})\right)^2} \tag{3.1.4}$$

qui seront substituées dans (3.1.2).

Le calcul du critère d'admissibilité pour deux ensembles σ et τ se fait alors en $\mathcal{O}(1)$ opérations, car les boîtes englobantes sont déjà connues une fois l'arbre de groupes construit. Cette construction est illustrée par la figure 3.6.

3.1.4.2.3 Construction La construction de l'arbre de blocs est récursive. Soit un couple $(\sigma, \tau) \in \mathcal{T} \times \mathcal{T}$. Si ce couple est admissible, la construction s'arrête en marquant la feuille de l'arbre de blocs ainsi construite admissible. Sinon, si τ ou σ est une feuille de $\mathcal{T}$, la construction s'arrête en marquant la feuille non admissible. Si ni τ ni σ n'est une feuille de $\mathcal{T}$ (et que $\sigma \times \tau$ n'est pas admissible), la construction se poursuit récursivement sur le produit des fils de σ et τ, ce qui est donné en pseudo-code par l'algorithme 3.2.

Algorithme 3.2 Construction de l'arbre de blocs

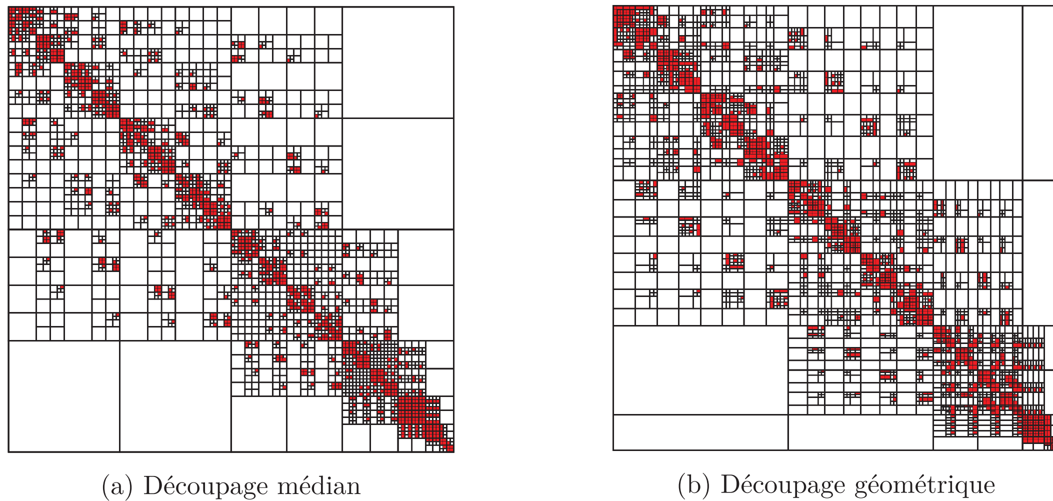
```
function CREATEBLOCKTREE(σ ∈ T, τ ∈ T)
    if f(σ,τ) = 1 ou S(σ) = ∅ ou S(τ) = ∅ then          ▷ cf l'équation 3.1.2
        S(σ × τ) = ∅
    else
        S(σ × τ) := {(σ', τ') | σ' ∈ S(σ), τ' ∈ S(τ)}
        for all (σ', τ') ∈ S(σ × τ) do
            CREATEBLOCKTREE(σ', τ')
        end for
    end if
end function
```

3.1.4.2.4 Remarques

- Il est naturel que l'arité de l'arbre de blocs soit le carré de celle de l'arbre de groupes, comme le montre l'étape de récursion ;
- L'arbre résultant n'est pas un arbre complet, même dans le cas de la division médiane ; il est néanmoins entier.
- Les feuilles non admissibles sont nécessairement de petite taille, puisque $\min(|\sigma|, |\tau|) \leqslant N_{leaf}$. Dans le cas de la division médiane, elle sera de petite taille selon les deux dimensions.
- Les feuilles admissibles peuvent être arbitrairement grandes.

L'arbre de blocs donne le squelette de la $\mathcal{H}$-Matrice, puisqu'une $\mathcal{H}$-Matrice est un arbre de blocs dont les feuilles ont été augmentées pour porter des données. Cette structure dépend :

- Du choix du découpage, médian ou géométrique ;

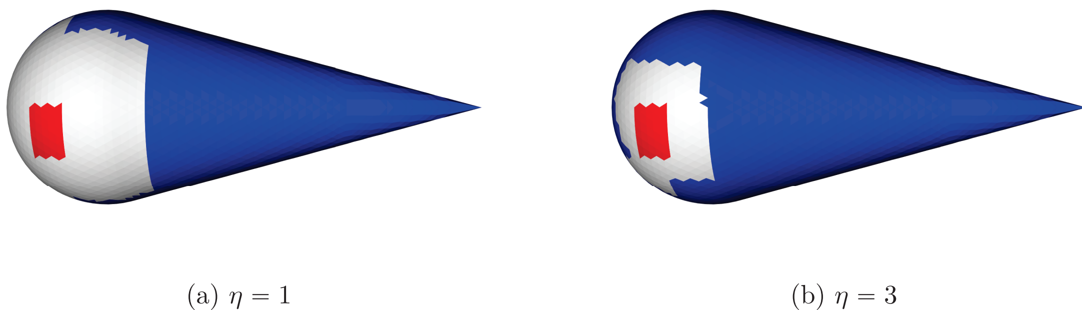


(a) Découpage médian (b) Découpage géométrique

FIGURE 3.7 – Comparaison de la structure pour deux découpages.

— De la valeur des paramètres η et N_{leaf}.

La figure 3.7 illustre ces différences pour deux choix de division spatiale, sur le même cas que celui de la figure 3.4 (9852 degrés de liberté), avec $N_{leaf} = 100$ et $\eta = 3$. Sur cette figure, les blocs rouges représentent des feuilles non admissibles et les blancs les feuilles admissibles de l'arbre de blocs. Puisque l'algorithme 3.2 s'arrête toujours sur un couple de feuilles au même niveau dans $\mathcal{T}$, toutes les feuilles de l'arbre de blocs sont presque carrées [2] pour le découpage médian, mais pas pour le découpage géométrique. De plus, le nombre de subdivisions horizontales et verticales est égal au nombre de feuilles de l'arbre de groupes, puisque le cas du bloc $\sigma \times \sigma$ descend toujours jusqu'aux feuilles (et constitue la diagonale rouge dans la figure 3.7).



(a) $\eta = 1$ (b) $\eta = 3$

FIGURE 3.8 – En bleu, ensemble des interactions admissibles avec le groupe rouge, deux valeurs de η.

Dans les deux cas, la majorité de la matrice est constituée de feuilles admissibles. Cette proportion est croissante avec la taille de la matrice, et avec la valeur de η. À titre d'exemple, la figure 3.8 montre sur la même géométrie l'impact d'une variation de η pour $N_{leaf} = 100$ et un découpage médian.

2. puisque σ et τ sont au même niveau de leurs arbres respectifs, dans le cas où ils sont confondus, $|\sigma| = |\tau| \pm 1$.

3.2 Compression

Comme évoqué dans le chapitre précédent, la compression des blocs admissibles est l'étape clé de la construction d'une $\mathcal{H}$-Matrice. Elle conditionne la précision, le stockage requis et le temps de calcul. Celui-ci est très fortement croissant avec le rang d'une $\mathcal{R}k$-Matrice (croissance en $\mathcal{O}(k^3 + (m+n)k^2)$) avec de surcroît une grande constante multiplicative, comme nous le verrons à la section 3.4.

L'arbre de blocs détermine le squelette de la $\mathcal{H}$-Matrice, qui n'est autre qu'un arbre de blocs augmenté sur ses feuilles admissibles par une $\mathcal{R}k$-Matrice et par une matrice pleine pour les feuilles non admissibles $\sigma \times \tau$, et la matrice attachée est simplement $M|_{\sigma\times\tau}$ (*cf.* définition 3.15).

Il existe diverses méthodes pour obtenir une représentation d'un bloc $M|_{\sigma\times\tau}$ sous forme de $\mathcal{R}k$-Matrice. La meilleure, au sens de la norme d'opérateur associée à la norme euclidienne, est la décomposition en valeurs singulières qui a été brièvement présentée dans le chapitre précédent, section 2.4.1. Elle n'est cependant pas adaptée pour la compression des blocs admissibles du fait de sa complexité en $\mathcal{O}(nm^2 + mn^2)$.

Il existe dans la littérature deux classes principales d'algorithmes :

— Les méthodes basées sur l'approximation adaptative croisée, *Adaptative Cross Approximation* (ACA). Elles utilisent des approximations successives de rang 1 du bloc. Il en existe plusieurs variantes. Le **pivotage total** est une méthode exacte pour les matrices de rang déficient et permet une complexité de calcul en $\mathcal{O}(kmn)$. Le **pivotage partiel** est une heuristique dérivée de la première et permet de ramener le temps de calcul à $\mathcal{O}(k^2(m+n)+k(m+n))$. Cette seconde heuristique a une faiblesse identifiée dans certaines situations que nous présenterons, et une variante de même complexité, appelée ACA+ a alors été proposée dans [56] pour y remédier.

— Une méthode basée sur un développement du noyau, appelée *Hybrid Cross Approximation* (HCA) [33]. Cette méthode n'est pas heuristique et des preuves de sa convergence existent. Elle nécessite néanmoins de pouvoir évaluer le noyau, alors que les méthodes précédentes sont indépendantes de la matrice.

La première classe de méthodes offre l'avantage de pouvoir opérer de manière totalement indépendante du contenu de la matrice et de son calcul. Ceci est important d'un point de vue pratique. En effet, dans la vie d'un code de calcul, les changements dans le calcul de la matrice sont nombreux : le passage à une autre physique (électromagnétisme vers acoustique, par exemple), la gestion d'une nouvelle condition aux limites, un changement de formulation mathématique, l'ajout de cas particuliers (symétrie, périodicité, décomposition de domaines), *etc.* Une méthode générique permet de réduire considérablement l'effort nécessaire à la gestion de toutes ces configurations. C'est un des inconvénients de la FMM, qui selon la formulation peut nécessiter des extensions non négligeables pouvant aller de la simple adaptation à un travail de recherche nouveau et conséquent.

Cependant, une méthode boîte noire dont la complexité est meilleure que $\mathcal{O}(mn)$ ne peut être générique ou uniformément bonne, puisqu'il lui est interdit de calculer tous les éléments du bloc, dont elle ne connaît rien par ailleurs. Elle permet cependant souvent en pratique d'obtenir des résultats de très bonne qualité, comme nous le montrerons à la section 3.5 de ce chapitre, ainsi qu'au chapitre 6.

Dans la suite, nous ne présenterons pas la méthode HCA, et renvoyons le lecteur à la bibliographie [33, 35] disponible sur le sujet.

3.2.1 Adaptative Cross Approximation

La décomposition en valeurs singulières permet de factoriser tout bloc $M|_{\sigma\times\tau} \in \mathbb{C}^{m\times n}$ par trois matrices U, Σ et V, et d'approcher celui-ci par la troncature de cette décomposition. L'inconvénient de cette méthode, outre son coût de calcul élevé, est la nécessité de calculer toute la matrice afin de construire son approximation.

Une autre famille de méthodes pour obtenir une approximation sous la forme d'une $\mathcal{R}k$-Matrice est appelée approximation croisée ou approximation squelette. Elle fut proposée pour répondre à ces deux problèmes, à partir de [53]. L'objectif est de pourvoir calculer une approximation d'une matrice en n'utilisant qu'un petit nombre de ses lignes et colonnes, appelées « squelettes ». Plus précisément, on cherche trois matrices, A, S et B telles que :

$$\tilde{M} = A.S.B^T \quad \text{et} \quad \|\tilde{M} - M|_{\sigma\times\tau}\| \leqslant \varepsilon$$

pour une précision $\varepsilon > 0$ fixée. On a dans cette écriture $A \in \mathbb{C}^{m\times k}$, $S \in \mathbb{C}^{k\times k}$ et $B \in \mathbb{C}^{n\times k}$, avec $k \leqslant \min(m, n)$. Les colonnes et les lignes utilisées sont qualifiées de pivots.

3.2.1.1 Existence

L'existence d'une approximation squelette n'est pas évidente, et semble même contre-intuitive, mais est assurée par le théorème suivant, dont la démonstration donnée dans [53] n'est malheureusement pas constructive :

Théorème 3.4 (Existence d'une approximation croisée)**.** *Soient $M|_{\sigma\times\tau}, R \in \mathbb{C}^{m\times n}$ deux matrices telles que $\|M|_{\sigma\times\tau} - R\| \leqslant \varepsilon$ et $rang(R) \leqslant k$. Alors il existe deux sous-ensembles $\sigma^* \subset \sigma$ et $\tau^* \subset \tau$ de lignes et colonnes pivots et une matrice $S \in \mathbb{C}^{k\times k}$ tels que :*

$$\|M|_{\sigma\times\tau} - M|_{\sigma\times\tau^*}.S.M|_{\sigma^*\times\tau}\|_2 \leqslant \varepsilon\left(1 + 2\sqrt{k}\left(\sqrt{|\sigma|} + \sqrt{|\tau|}\right)\right)$$

Ce théorème signifie que l'existence d'une approximation de rang faible d'une matrice avec une erreur ε fixée implique l'existence d'une approximation squelette. L'existence d'une telle approximation est à lier au théorème 2.4. Il ne donne cependant pas de méthode pour construire cette approximation, et dans la suite nous décrivons deux heuristiques donnant une représentation sous forme d'approximation croisée de $M|_{\sigma\times\tau}$. La première, qualifiée de pivotage total, a une complexité quadratique, et la seconde sera linéaire, pour un rang k de l'approximation fixé.

Les algorithmes d'approximation croisée avec pivotage ont été introduits en premier par Bebendorf dans [22, 24], dans la variante avec pivotage partiel. Nous présentons cependant en premier lieu la version « pivotage total » de l'algorithme, car elle possède des propriétés rassurantes supplémentaires par rapport à la version partiellement pivotée. Les preuves des différents lemmes de cette section se trouvent dans [35, chapitre 4].

Remarque 3.5 (Garantie)**.** *Il faut noter qu'aucune de ces deux méthodes n'assure de trouver une bonne approximation de rang faible de la matrice $M|_{\sigma\times\tau}$.*

Remarque 3.6 (À propos de S). *Dans la suite, nous présentons des heuristiques pour lesquelles la matrice S intervenant dans la décomposition $M|_{\sigma\times\tau} \simeq ASB^T$ est telle que $S = I_k$. Ces deux heuristiques peuvent néanmoins aisément s'écrire avec S une matrice diagonale, contenant les pivots.*

3.2.2 Pivotage total

Dans cette section, on suppose $M|_{\sigma\times\tau}$ connue, c'est-à-dire assemblée (en $\mathcal{O}(nm)$ opérations) et on posera $M := M|_{\sigma\times\tau}$ pour alléger les notations. L'élément de base de l'approximation est la construction d'une approximation squelette de rang 1. Cette approximation est réalisée de la façon suivante :

1. Recherche du pivot δ et détermination des ligne i^* et colonne j^* pivots :
$$\delta = M_{i*j*} \quad \text{avec} \quad M_{i*j*} = \max_{i=1,\ldots,m,j=1,\ldots,n} |M_{ij}|$$
2. On pose
$$a_i^1 := M_{i,j*} \quad \text{et} \quad b_j^1 := M_{i*j}/\delta$$
et la matrice $R_1 := a^1(b^1)^T \in C^{m\times n}$ est une matrice de rang 1 sous forme d'approximation squelette.

Ceci constitue la première étape de l'algorithme. Pour obtenir l'approximation de rang 2, on soustrait à $M|_{\sigma\times\tau}$ la matrice $a^1(b^1)^T$ et on reproduit les mêmes étapes. L'algorithme s'arrête si le pivot est nul (ce qui signifie que la matrice mise à jour est nulle) ou si un autre critère d'arrêt est satisfait, au bout de k itérations. La définition suivante précise la construction de l'approximation :

Définition 3.7 (Approximation ACA, pivotage total). *Soit $M|_{\sigma\times\tau} \in \mathbb{C}^{m\times n}$ une matrice. On appelle approximation ACA avec pivotage total le couple de matrices $(A, B) \in \mathbb{C}^{m\times k} \times \mathbb{C}^{n\times k}$ avec $k \leqslant \min(m, n)$ défini dans la suite.*

Soient deux suites d'indices de lignes et colonnes pivots $(i^*)_{\nu=1,\ldots,k}$ et $(i^*)_{\nu=1,\ldots,k}$ avec $i^*_\nu \in \{1, m\}$ et $j^*_\nu \in \{1, n\}$, et deux suites de vecteurs $(a^\nu)_\nu$ et $(b^\nu)_\nu$, avec $a^\nu \in \mathbb{C}^m$ et $b_\nu \in \mathbb{C}^n$.

Les suites d'indices des pivots sont définies par :

$$(i^*_\nu, j^*_\nu) := \operatorname*{arg\,max}_{(i,j)\in\{1,m\}\times\{1,n\}} \left| \left(M - \sum_{l=1}^{\nu-1} a^l(b^l)^T \right)_{ij} \right| \tag{3.2.1}$$

et la suite des pivots $(\delta_\nu)_\nu$ est :

$$\delta_\nu := \left(M|_{\sigma\times\tau} - \sum_{l=1}^{\nu-1} a^l(b^l)^T \right)_{i^*_\nu j^*_\nu}$$

Les suites (a^ν) et (b^ν) sont alors définies par :

$$a_i^\nu := \left(M|_{\sigma\times\tau} - \sum_{l=1}^{\nu-1} a^l(b^l)^T \right)_{ij^*_l} \quad \text{et} \quad a_j^\nu := \frac{1}{\delta_\nu} \left(M|_{\sigma\times\tau} - \sum_{l=1}^{\nu-1} a^l(b^l)^T \right)_{i^*_\nu j}$$

Les matrices $A := (a^\nu)_{\nu=1,\dots,k}$ et $B := (b^\nu)_{\nu=1,\dots,k}$ sont l'approximation ACA avec pivotage total.

Remarque 3.8 (Unicité)**.** *On montre aisément que l'approximation ACA avec pivotage total d'une matrice $M|_{\sigma\times\tau} \in \mathbb{C}^{m\times n}$ est unique.*

Algorithme 3.3 Approximation croisée avec pivotage total.

```
function AcaFull(M|σ×τ ∈ C^{m×n})
    k ← 1
    repeat
        (i*, j*) ← arg max_{i,j} |M_{i,j}|
        δ ← M_{i*j*}
        if δ = 0 then                    ▷ Le rang de la matrice est exactement k − 1
            return A := (a^ν)_{ν=1,...,k−1}   B := (b^ν)_{ν=1,...,k−1}
        else
            (a^ν)_i := M_{i,j*}
            (b^ν)_j := M_{i*,j}/δ
            M|σ×τ ← M|σ×τ − a^ν(b^ν)^T
            k ← k + 1
        end if
    until Convergence
    return A := (a^ν)_{ν=1,...,k}   B := (b^ν)_{ν=1,...,k}
end function
```

Le pseudo-code pour cette procédure est donné par l'algorithme 3.3. Le critère d'arrêt n'est pas précisé dans cette description. Il peut prendre plusieurs formes :

— Un rang maximal k_{max} défini à l'avance a été atteint ;

— L'estimation est considérée satisfaisante pour une précision ε fixée.

Le second cas est celui qui nous intéresse. On définit le résidu de l'approximation :

Définition 3.9 (Résidu de l'approximation)**.** *Soit $R_k := A.B^T$ une $\mathcal{R}k$-Matrice, avec $(A, B) \in \mathbb{C}^{m\times k} \times \mathbb{C}^{n\times k}$ l'approximation ACA avec pivotage total d'une matrice $M|_{\sigma\times\tau} \in \mathbb{C}^{m\times n}$. On appelle résidu de l'approximation le nombre*

$$r_k := \|R_k - M|_{\sigma\times\tau}\|_F \tag{3.2.2}$$

avec $\|\cdot\|_F$ la norme de Frobenius.

Le calcul de r_k nécessite $\mathcal{O}(mn)$ opérations une fois la matrice R_k calculée. Le produit $A.B^T$ n'étant pas évalué lors de la construction présentée dans la définition 3.7 (ni dans l'algorithme 3.3), le coût d'évaluation du résidu est $\mathcal{O}(mnk)$ par itération. On choisit alors le critère d'arrêt suivant :

$$r_k \leqslant \varepsilon$$

pour $\varepsilon > 0$ fixé. En remarquant que le résidu est évalué jusqu'à ce que le critère d'arrêt soit satisfait, le coût d'évaluation de celui-ci représente $\mathcal{O}(k^2mn)$ sur l'ensemble de l'algorithme 3.3 (avec k le rang final obtenu à la fin de cet algorithme).

Complexité Les opérations nécessaires pour k itérations de l'algorithme ACA avec pivotage total sont :

Construction de $M\|_{\sigma\times\tau}$	$\mathcal{O}(mn)$
Recherche du pivot	$\mathcal{O}(kmn)$
Construction de A et B	$\mathcal{O}(k(m+n))$
Mise à jour de la matrice	$\mathcal{O}(kmn)$
Calcul du résidu	$\mathcal{O}(k^2mn)$
Total	$\mathcal{O}(mn + k^2mn + kmn)$

Cet algorithme est bien de complexité quadratique. Dans la littérature, le coût de cet algorithme est souvent donné en $\mathcal{O}(kmn)$, ce qui n'est pas le cas ici. Cette différence s'explique par le calcul du résidu (3.2.3). En réalité, il est possible d'approcher ce calcul comme nous le ferons dans la section suivante, et d'utiliser une forme de complexité inférieure (équation (3.2.5) pour le calcul de $\|R_k\|$, et (3.2.4) pour le critère d'arrêt). Nous donnons néanmoins l'algorithme avec ce calcul du résidu, car il a l'avantage de permettre d'assurer que la convergence de l'algorithme soit une vraie convergence, et non un artefact de l'approximation de son calcul. De plus, dans la majorité des situations pratiques, ce calcul sera moins coûteux que le calcul du bloc $M|_{\sigma\times\tau}$, étant donné la constante associée (*cf.* section 2.2.4.1).

Propriétés de l'approximation (pivotage total) On rappelle que les énoncés de ces lemmes et leurs preuves se trouvent dans [35, chapitre 4].

Lemme 3.10 (Reproduction exacte des matrices de rang k)**.** *Soit R une matrice de rang k. Alors la $\mathcal{R}k$-Matrice $R_k = \sum_{\nu=1}^{k} a^\nu (b^\nu)^T$ est égale à R.*

Lemme 3.11 (Reproduction exacte des lignes et colonnes pivots)**.** *Soit $M|_{\sigma\times\tau}$ une matrice de rang supérieur ou égal à k, et R_k son approximation de rang k donnée par l'algorithme 3.3. Alors pour tous les pivots (i^*, j^*) choisis dans cet algorithme, on a :*

$$R_k e_{j*} = M|_{\sigma\times\{j*\}} \quad et \quad e_{i*}^T R_k = M|_{\{i*\}\times\tau}$$

avec e_i le $i-$ème vecteur de base de la base canonique.

Lemme 3.12. *Soient $\mathcal{P}_l$ et $\mathcal{P}_c$ les ensembles d'indices pivots ligne et colonne, et soit R_k l'approximation obtenue après k itérations de l'algorithme 3.3. Alors R_k peut s'écrire sous la forme :*

$$R_k = \sum_{\nu=1}^{k} a^\nu (b^\nu)^T = M|_{\sigma\times\mathcal{P}_c}.\left(M|_{\mathcal{P}_l\times\mathcal{P}_c}\right)^{-1}.M|_{\mathcal{P}_l\times\sigma}$$

3.2.3 Pivotage partiel

Les étapes coûteuses (quadratiques) de l'algorithme précédent sont :

- La construction du bloc ;
- La détermination du pivot ;
- La mise à jour de la matrice ;

— Le calcul du résidu.

Pourtant, dans le déroulement de l'algorithme, seules les lignes et colonnes choisies comme pivots sont réellement utilisées (en-dehors du calcul du résidu) ! Ainsi, si les lignes et colonnes pivots pouvaient être connues à l'avance, le calcul de la matrice deviendrait inutile, et il ne resterait que le problème du critère d'arrêt que nous négligeons pour l'instant.

Supposons donc la suite des pivots $(i^\nu, j^\nu)_{\nu=1,\dots,k}$ connue à l'avance. On ne procède alors qu'au calcul des k lignes et colonnes correspondantes, ce qui nécessite $\mathcal{O}(k(m+n))$ opérations. De même, la mise à jour de la matrice est restreinte à ces lignes et colonnes, en $\mathcal{O}(k^2(m+n))$.

Cependant, la construction des suites de pivots (3.2.1) est itérative. De par la structure de celle-ci, sa connaissance a priori n'est donc pas possible avec cette construction. Néanmoins, en restreignant la recherche du pivot non plus à toute la matrice mais à une partie de celle-ci déjà calculée, nous pouvons conserver toutes ces propriétés et obtenir ainsi un algorithme en $\mathcal{O}(k^2(m+n))$, ce qui est le but recherché. C'est cette recherche de pivot restreinte qui donne le nom « pivotage partiel » à cette méthode. Elle est, de plus, à rapprocher dans l'idée de la décomposition LU avec pivotage. Pour plus de détails sur cette relation, le lecteur est invité à consulter [22, section 3].

L'initialisation de l'algorithme se fait par une ligne, par exemple la première ligne de $M|_{\sigma\times\tau}$ qui n'a pas été pré-calculée. Cette ligne est désignée arbitrairement comme ligne pivot ($i^* = 1$), et on recherche le pivot sur celle-ci, de la même façon que précédemment.

Une fois la colonne pivot déterminée, on effectue une itération de l'algorithme 3.3. On fixe ensuite le pivot colonne j^* à la colonne trouvée précédemment, et on cherche la ligne pivot sur cette colonne, et ainsi de suite.

L'algorithme fait donc un « tour » dans la matrice en alternant son exploration sur les lignes et les colonnes pour la recherche du prochain pivot. L'avantage de cette approche est clair : dans le cas de la recherche d'un pivot ligne, la colonne parcourue vient d'être évaluée pour l'itération précédente. Ainsi, sous réserve que le pivot ne soit pas nul, sa recherche ne nécessite jamais de calculer une nouvelle ligne ou colonne de la matrice, et l'algorithme ne nécessite l'évaluation que de $k(m+n)$ éléments de la matrice pour k itérations. L'argument d'entrée de l'algorithme 3.4 n'est plus $M|_{\sigma\times\tau}$, mais une fonction f permettant d'évaluer un élément quelconque de $M|_{\sigma\times\tau}$.

Pour les itérations suivant la première, la colonne ou la ligne calculée n'est pas directement une ligne ou une colonne de $M|_{\sigma\times\tau}$, mais une ligne ou colonne de $M|_{\sigma\times\tau} - \sum_{\nu=1}^{k} a^\nu (b^\nu)^T$. Le calcul est donc remplacé par la combinaison calcul-mise à jour, de complexité $\mathcal{O}(k'(m+n))$ pour l'étape k', soit $\mathcal{O}(k^2(m+n))$ au total.

Dans l'algorithme 3.4, on note $\mathcal{P}_l$ et $\mathcal{P}_c$ les ensembles de lignes et colonnes ayant déjà été utilisées comme pivot.

Remarque 3.13 (Optimisation)**.** *L'algorithme peut s'écrire de la même façon en débutant par une colonne plutôt que par une ligne. Dans ce cas, la rencontre d'un pivot nul mène au calcul d'une colonne supplémentaire et non d'une ligne. De plus, le critère de sortie sans convergence est également modifié. Ceci est avantageux si le nombre de colonnes est inférieur au nombre de lignes. Il est donc possible d'obtenir une optimisation mineure du*

Algorithme 3.4 Approximation croisée avec pivotage partiel.

```
function AcaPartial(f)
    k ← 1, P_l ← ∅, P_c ← ∅
    i* ← 1                                   ▷ La première ligne est choisie comme pivot
    repeat
        P_l ← P_l ∪ {i*}
        b_j^k ← M_{i*j} − Σ_{ν=1}^{k−1} a_{i*}^ν b_j^ν          ▷ Calcul et mise à jour
        j* ← argmax_{j∈τ−P_c} |b^ν|,  δ ← b_{j*}^k             ▷ Recherche de la colonne pivot
        if δ = 0 then
            if σ − P_l = ∅ then      ▷ Il n'est plus possible de trouver une ligne pivot.
                return                                 ▷ Pas de convergence
            end if
            i* = min{i ∈ σ | i ∉ P_l}               ▷ Ou tout autre choix dans σ − P_l
        else                                          ▷ Un pivot non nul est trouvé
            P_c ← P_c ∪ j*
            b ← b/δ
            a_i^k ← M_{ij*} − Σ_{ν=1}^{k−1} a_i^ν b_{j*}^ν      ▷ Calcul et mise à jour
            i* ← argmax_{i∈σ−P_l} |a^ν|                 ▷ Recherche de la ligne pivot
            k ← k + 1
        end if
    until Convergence
    return A := (a^ν)_{ν=1,...,k}   B := (b^ν)_{ν=1,...,k}
end function
```

temps de calcul en choisissant de procéder sur les lignes ou les colonnes en fonction de $|\sigma|$ *et* $|\tau|$.

Critère d'arrêt La discussion précédente a reporté le problème du calcul du critère d'arrêt, qui est pourtant coûteux dans l'algorithme 3.3. Le calculer supprimerait tout avantage au pivotage partiel, puisqu'il nécessite la connaissance de l'intégralité de $M|_{\sigma\times\tau}$.

Le critère que nous souhaitons approcher est

$$\|M|_{\sigma\times\tau} - R_k\|_F \leqslant \varepsilon \, \|M|_{\sigma\times\tau}\|_F \tag{3.2.3}$$

Remplaçons $M|_{\sigma\times\tau}$ par son approximation de rang k et R_k par R_{k-1} dans l'expression précédente :

$$\|R_k - R_{k-1}\|_F \leqslant \varepsilon \, \|R_k\|_F$$

Il est clair que $R_k - R_{k-1} = a^k (b^k)^T$, et cette expression devient donc :

$$\|a^k\|_2 \|b^k\|_2 \leqslant \varepsilon \, \|R_k\|_F \tag{3.2.4}$$

qui ne dépend plus de $M|_{\sigma\times\tau}$, mais uniquement de quantités calculées par l'algorithme

3.4. Le calcul de $\|R_k\|_F$ peut se faire itérativement, avec $\langle a, b\rangle = ab^*$ le produit scalaire :

$$\|R_k\|_F^2 = \|R_{k-1}\|_F^2 + \sum_{l=1}^{k-1} \left(\left\langle a^k, a^l \right\rangle \left\langle b^k, b^l \right\rangle + \left\langle a^l, a^k \right\rangle \left\langle b^l, b^k \right\rangle\right) + \left\|a^k\right\|_2^2 + \left\|b^k\right\|_2^2 \quad (3.2.5)$$

pour un coût total de $\mathcal{O}(k^2(m+n))$.

Complexité Le coût total des opérations nécessaires (en l'absence d'un trop grand nombre de lignes nulles) est :

Calcul des lignes et colonnes	$\mathcal{O}(k(m+n))$
Recherche des pivots	$\mathcal{O}(k(m+n))$
Construction de A et B	$\mathcal{O}(k^2(m+n))$
Calcul du critère d'arrêt	$\mathcal{O}(k^2(m+n))$
Total	$\mathcal{O}(k^2(m+n) + k(m+n))$

Remarque 3.14 (Performance)**.** *En réalité, le coût de calcul des lignes de la matrice d'interaction est de loin le facteur dominant dans le coût du calcul. En effet, les autres calculs ne font appel qu'à des multiplications et additions (plus une division du pivot par itération) qui sont bien plus rapides que les calculs d'intégrales détaillés à la section 2.2.4. L'algorithme se comporte donc* en pratique *comme si sa complexité était* $\mathcal{O}(k(m+n))$.

3.2.4 Contre-exemple pour le pivotage partiel et ACA+

La complexité linéaire de l'algorithme partiellement pivoté est extrêmement importante pour réaliser les promesses des $\mathcal{H}$-Matrices (complexité asymptotique meilleure que $\mathcal{O}(N^2)$). Sans algorithme capable d'assembler une $\mathcal{R}k$-Matrice sans calculer le bloc correspondant, il est impossible de réaliser des algorithmes ayant une complexité meilleure que $\mathcal{O}(N^2)$. Ceci rend cette heuristique d'un très grand intérêt, comme en témoigne sa large utilisation dans la littérature [23, 24, 49], y compris pour l'électromagnétisme [90, 100] et l'acoustique [77]. Ce n'est cependant qu'une heuristique, et à ce titre elle peut en particulier exhiber les problèmes suivants :

— Ne pas satisfaire le critère d'arrêt alors que la précision désirée est atteinte ;

— Arrêter l'algorithme par le critère d'arrêt alors que la précision requise n'est pas atteinte. Un exemple de cette situation est présenté dans la section suivante.

3.2.4.1 Contre-exemple

Des contre-exemples divers sont présents dans la littérature, le plus marquant étant le cas du potentiel de double couche en dimension 3 pour le noyau $\frac{1}{r}$, qui est asymptotiquement lisse. Un tel contre-exemple est présenté dans [35], et nous n'en répéterons pas l'exposition ici.

De façon générale, si $M|_{\sigma\times\tau}$ a la forme de la figure 3.9, alors il est aisé de montrer que l'algorithme ACA avec pivotage partiel n'explorera que le bloc dans lequel l'algorithme a été initialisé.

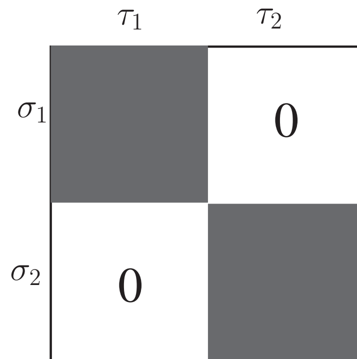


Figure 3.9 – Contre-exemple pour ACA avec pivotage partiel.

Soit $\{\sigma_1, \sigma_2\}$ et $\{\tau_1, \tau_2\}$ deux partitions de σ et τ, respectivement. Supposons le premier pivot ligne $i^* \in \sigma_1$ au début de l'algorithme. Alors le premier pivot colonne sera dans τ_1 puisque les éléments de M_{i*j} sont nuls pour tout $j \in \tau_2$. La soustraction du squelette préservant les lignes et les colonnes nulles de la matrice dans cette configuration, les pivots lignes et colonnes suivants seront dans le sous-ensemble des indices $\sigma_1 \times \tau_1$. Ainsi, l'approximation et le critère d'arrêt seront identiques au cas où le bloc $\sigma_2 \times \tau_2$ est remplacé par un bloc nul. L'erreur est donc alors du deuxième type décrit, c'est-à-dire un critère d'arrêt satisfait sans que la précision demandée ne soit atteinte.

Cette situation peut sembler académique, et, présentée ainsi, l'est certainement. Cependant il est aisé de se convaincre que le même problème se produit dans le cas où les indices de σ_1 et de τ_1 ne sont pas contigus dans σ et τ. En effet, dans la démonstration précédente, les indices des éléments de ces partitions ne sont pas nécessairement contigus. Cette démonstration est donc valable pour une situation plus générale que celle de la figure 3.9, qui en est un cas particulier. De manière analogue, le même phénomène se produit si le premier pivot ligne est dans σ_2.

Il suffit donc qu'au sein d'un bloc, deux groupes de degrés de liberté existent et n'interagissent pas entre eux. Cette situation se produit en pratique dans le cadre du formalisme adopté par la suite logicielle ASERIS pour certaines combinaisons de conditions aux limites et de géométries (*cf.* section 6.1.2). Ceci a motivé l'utilisation d'une heuristique robuste à ce cas précis, appelée ACA+ dans la littérature.

Avant de la présenter, il convient de noter que dès lors que σ et τ sont partitionnés différemment, ce problème disparaît. Par exemple, supposons que $\sigma = \sigma_1 \cup \sigma_2 \cup \sigma_3$ (idem pour τ), avec les interactions entre les groupes $\sigma_{1,2}$ et $\tau_{1,2}$ comme dans la figure 3.9. Si les groupes σ_3 et τ_3 interagissent avec tous les autres, alors le contre-exemple présenté ici n'en est plus un ! Cette situation a été observée, nous y reviendrons au chapitre 6.

3.2.4.2 ACA+

Supposons que le bloc $M|_{\sigma \times \tau}$ ait la forme de la figure 3.9, à une permutation près des lignes et des colonnes. La difficulté posée à l'algorithme 3.4 dans ce cas est liée au fait qu'à aucun moment l'heuristique n'a la « connaissance » de l'existence du bloc $\sigma_2 \times \tau_2$. Il faudrait donc un « oracle » indiquant à l'algorithme la présence de ce bloc, et lui permettant de

s'autoriser à chercher les pivots dans celui-ci. C'est précisément ce que l'heuristique ACA+ propose [35, 56].

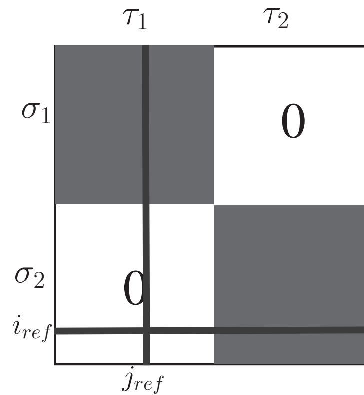


Figure 3.10 – Ligne et colonne de référence pour ACA+

L'algorithme débute par le choix arbitraire d'une colonne de référence, d'indice j_{ref}. On choisit alors une ligne pivot à partir des valeurs le long de cette colonne. L'objectif étant d'obtenir une situation comparable à la figure 3.10, un choix naturel pour i_{ref} est donc :

$$i_{ref} = \arg\min_{i \in \sigma} |M_{ij_{ref}}|$$

La ligne et la colonne de référence guident le choix des pivots dans l'algorithme, ce qui permet de prendre en compte les valeurs de la matrice dans les deux blocs $\sigma_1 \times \tau_1$ et $\sigma_2 \times \tau_2$. On cherche l'élément de plus grande norme selon la ligne et la colonne de référence. Supposons que cet élément se trouve sur la ligne de référence, c'est-à-dire nécessairement dans le bloc $\sigma_2 \times \tau_2$, et notons cet élément $M_{i_{ref}j^*}$. On cherche alors les indices du pivot (i^*, j^*) le long de la colonne j^* de la même manière que dans l'algorithme 3.4, et ce pivot sera dans $\sigma_2 \times \tau_2$. À l'inverse, si la plus grande valeur de la ligne et de la colonne de référence se trouve le long de la colonne de référence, le pivot sera choisi dans $\sigma_1 \times \tau_1$. Cette modification de l'heuristique assure ainsi de chercher les éléments dans les deux parties non nulles de $M|_{\sigma\times\tau}$, grâce à la ligne et colonne de référence.

Dans le cas où un des indices pivot choisi est égal à celui de la ligne ou colonne de référence ($i^* = i_{ref}$ et/ou $j^* = j_{ref}$), il faut changer de ligne/colonne de référence. En effet, une ligne utilisée dans la matrice se retrouve nulle dans le résidu, et ne peut donc plus servir d'observateur. Dans ce cas, si il est nécessaire de choisir un seul nouveau vecteur, on choisit le nouvel indice de référence en utilisant l'élément de plus faible norme. Dans le cas où les deux indices sont à renouveler, on procède comme lors de l'initialisation de l'algorithme.

Par souci de concision et de clarté, nous appellerons update l'opération de mise à jour d'une ligne ou colonne de matrice apparaissant dans l'algorithme 3.4 dans la description d'ACA+.

3.2.4.2.1 Remarques Nous pouvons faire les remarques suivantes sur cette variante de l'algorithme :

Algorithme 3.5 ACA+

function AcaPlus(f)
  $k \leftarrow 1,\ \mathcal{P}_l \leftarrow \varnothing,\ \mathcal{P}_c \leftarrow \varnothing$
  $j_{ref} \leftarrow j_0$ ▷ j_{ref} est arbitraire
  $c_i^{ref} \leftarrow M_{ij_{ref}}$
  $i_{ref} = \arg\min_{i \in \sigma} |c_{ref}|$
  $l_j^{ref} \leftarrow M_{i_{ref}j}$
  repeat
    Update($c^{ref}, \{a_1, \dots, a_{k-1}\}, \{b_1, \dots, b_{k-1}\}$)
    Update($l^{ref}, \{a_1, \dots, a_{k-1}\}, \{b_1, \dots, b_{k-1}\}$)
    $i^* \leftarrow \arg\max_i |c^{ref}|$
    $j^* \leftarrow \arg\max_j |l^{ref}|$
    if $c_{i^*}^{ref} > l_{j^*}^{ref}$ **then** ▷ i^* est fixé, on cherche j^*
      $b_j^k \leftarrow M_{i^*j}$
      Update($b_j^k, \{a_1, \dots, a_{k-1}\}, \{b_1, \dots, b_{k-1}\}$)
      $j^* = \arg\max_{j \in \tau} |b_j^k|$
      $\delta = b_{j^*}^k$
      $a_i^k \leftarrow M_{ij^*}$
      Update($a_i^k, \{a_1, \dots, a_{k-1}\}, \{b_1, \dots, b_{k-1}\}$)
      $a^k \leftarrow a^k / \delta$
    else ▷ j^* est fixé, on cherche i^*
      Idem avec i^* au lieu de j^*
    end if
    $\mathcal{P}_l \leftarrow \mathcal{P}_l \cup \{i^*\}$
    $\mathcal{P}_c \leftarrow \mathcal{P}_c \cup \{j^*\}$
    if $i^* = i_{ref}$ et/ou $j^* = j_{ref}$ **then**
      if $i^* = i_{ref}$ et $j^* = j_{ref}$ **then**
        Choisir $j_{ref} \in \tau - \mathcal{P}_c$ arbitrairement
        $c_i^{ref} \leftarrow M_{ij_{ref}}$
        $i_{ref} = \arg\min_{i \in \sigma} |c_{ref}|$
        $l_j^{ref} \leftarrow M_{i_{ref}j}$
      end if
    else
      if $i^* = i_{ref}$ **then** ▷ Il faut choisir un nouveau j_{ref}
        $j_{ref} = \arg\min_{j \in \tau - \mathcal{P}_c} |b_j^{ref}|$
        $c_i^{ref} \leftarrow M_{ij_{ref}}$
      else ▷ Il faut choisir un nouveau i_{ref}
        $i_{ref} = \arg\min_{i \in \sigma - \mathcal{P}_l} |a_i^{ref}|$
        $l_j^{ref} \leftarrow M_{i_{ref}j}$
      end if
    end if
  until Convergence
end function

— La complexité est toujours la même, à savoir $\mathcal{O}(k^2(m+n))$ en théorie et $\mathcal{O}(k(m+n))$ en pratique (*cf.* remarque 3.14). Il est nécessaire de calculer une colonne et une ligne de référence en plus, mais dans tous les cas le surcoût pour l'intégralité de l'algorithme est exactement le calcul (et la mise à jour) d'une colonne et d'une ligne supplémentaire. En effet, une ligne/colonne n'est remplacée que lorsqu'elle est utilisée comme pivot, et donc réutilisée dans le calcul.

— Cette heuristique permet de traiter avec succès le cas problématique de la figure 3.9. Il est cependant aisé de trouver des contre-exemples pour lesquels cette heuristique ne donne pas de résultats satisfaisants. Néanmoins, de tels cas n'ont à ce jour pas été rencontrés dans les applications qui nous concernent.

— Dans [56], l'algorithme ACA+ permet dans la majorité des cas d'obtenir un rang numérique plus faible pour la $\mathcal{R}k$-Matrice résultat qu'ACA avec pivotage partiel, sans qu'une explication à ce fait ne soit avancée.

3.3 Opérations sur les $\mathcal{H}$-Matrices

Nous venons de détailler la construction du squelette de la $\mathcal{H}$-Matrice dans la section 3.1.4.2, puis le remplissage d'un élément de cette ossature dans la section 3.2. Il est donc enfin possible de donner une définition d'une $\mathcal{H}$-Matrice :

Définition 3.15 ($\mathcal{H}$-Matrice)**.** *Soient I et J deux ensembles d'indices, $\mathcal{T}_I$ et $\mathcal{T}_J$ des arbres de groupes construits sur ces indices, et $\mathcal{T}_{I\times J}$ un arbre de blocs construit sur les arbres de groupes $\mathcal{T}_I$ et $\mathcal{T}_J$ suivant l'algorithme 3.2. Une $\mathcal{H}$-Matrice construite sur un arbre $\mathcal{T}_{I\times J}$ est un arbre de blocs dont les feuilles ont été augmentées par :*

— *Une $\mathcal{R}k$-Matrice pour les feuilles admissibles ;*

— *Une matrice pleine pour les feuilles non admissibles.*

Nous donnons également une définition supplémentaire, celle de l'ensemble des $\mathcal{H}$-Matrices de rang k par bloc, qui sera utile dans la suite :

Définition 3.16 (Ensemble des $\mathcal{H}$-Matrices de rang k par bloc)**.** *Avec les mêmes notations que dans la définition précédente, on définit l'ensemble des $\mathcal{H}$-Matrices de rang (numérique) k par :*

$$\mathcal{H}(\mathcal{T}_{I\times J}, k) := \left\{ M \in \mathbb{C}^{M\times N} \mid rang(M|_{\sigma\times\tau}) \leqslant k \quad \forall(\sigma\times\tau) \in \mathcal{T}_{I\times J} \text{ admissible} \right\}$$

avec $|I| = M$, $|J| = N$.

Afin de faciliter la discussion sur les propriétés d'approximation de ces algorithmes, nous définissons aussi :

Définition 3.17 (Troncature de rang k pour une $\mathcal{R}k$-Matrice)**.** *Soit $M \in \mathbb{C}^{m\times n}$ et $k \in N$. On définit l'opération de troncature $\mathcal{T}_k$ d'une matrice dans l'ensemble $\mathcal{R}(m,n,k)$ des $\mathcal{R}k$-Matrices de rang au plus k par :*

$$\begin{array}{rccc} \mathcal{T}_k : & \mathbb{C}^{m\times n} & \longrightarrow & \mathcal{R}(m,n,k) \\ & M & \longmapsto & \tilde{M} \end{array}$$

avec $\tilde{M}$ la meilleure approximation de M dans $\mathcal{R}(m,n,k)$ au sens de la norme de Frobenius.

Définition 3.18 (Troncature de rang k pour une $\mathcal{H}$-Matrice)**.** *Soit $M \in \mathbb{C}^{M\times N}$. On définit la troncature de rang k de M dans $\mathcal{H}(\mathcal{T}_{I\times J}, k)$*

$$\begin{array}{rccl} \mathcal{T}_k : & \mathbb{C}^{M\times N} & \longrightarrow & \mathcal{H}(\mathcal{T}_{I\times J}, k) \\ & M & \longmapsto & \tilde{M} \end{array}$$

par une troncature bloc par bloc de ses feuilles admissibles, c'est-à-dire :

$$\forall(\sigma,\tau) \in Feuilles(\mathcal{T}_{I\times J}),\ \tilde{M}|_{\sigma\times\tau} := \begin{cases} \mathcal{T}_k(M|_{\sigma\times\tau}) & si\ (\sigma\times\tau)\ est\ admissible \\ M|_{\sigma\times\tau} & sinon \end{cases}$$

Tous les éléments nécessaires pour écrire les algorithmes opérant sur les $\mathcal{H}$-Matrices sont donc en place. Comme évoqué précédemment, du fait de la **structure arborescente** de la $\mathcal{H}$-Matrice, les algorithmes prendront une forme naturellement **récursive**.

Ces algorithmes se divisent en deux grandes classes :

— Ceux dont les opérandes sont nécessairement des feuilles de la $\mathcal{H}$-Matrice. Ils ne contiennent en eux aucun caractère hiérarchique, et sont de ce fait plus simples à implémenter et, nous le verrons plus tard, à paralléliser. Cette classe d'algorithmes contient l'assemblage, le produit d'une $\mathcal{H}$-Matrice par un vecteur et l'addition de deux $\mathcal{H}$-Matrices.

— Les algorithmes véritablement hiérarchiques, pouvant opérer à différents niveaux de l'arbre, et même à différents niveaux d'un même sous-arbre, ce qui complique leur implémentation. L'algorithme modèle de cette classe est la multiplication matricielle, et contient également la résolution de systèmes triangulaires matriciels.

Il est possible de rajouter une troisième classe d'algorithmes, utilisant les deux premières comme briques de base. Cette classe contient l'inversion et les diverses factorisations, *LU* en particulier. Il est intéressant de rapprocher cette classification de deux bibliothèques d'algèbre linéaire dense, BLAS et LAPACK. En particulier, les opérations décrites dans ce chapitre peuvent être présentées comme un sous-ensemble de BLAS et LAPACK basé sur les $\mathcal{H}$-Matrices.

3.3.1 $\mathcal{H}$-Blas et $\mathcal{H}$-Lapack

3.3.1.1 Blas et Lapack

Blas (pour *Basic Linear Algebra Subprograms*) [47] est une interface de programmation pour l'algèbre linéaire dense, devenue un standard de fait depuis l'apparition de la première version en 1979. Il ne s'agit pas d'un composant informatique, mais d'un ensemble de conventions d'appel de fonctions pour effectuer des opérations vecteur-vecteur (niveau 1), matrice-vecteur (niveau 2) et matrice-matrice (niveau 3). Il en existe de nombreuses implémentations, optimisées pour une grande variété de machines. Au-dessus de BLAS, LAPACK (Linear Algebra PACK) [25] a été proposé à partir de 1992 pour implémenter

des opérations de plus haut niveau, telles que l'inversion ou la décomposition en valeurs singulières, sur le même modèle que BLAS, sur laquelle cette bibliothèque s'appuie.

Le succès de ces deux interfaces repose en partie sur la séparation des problèmes. Ainsi, un utilisateur d'algèbre linéaire est déchargé des questions de performance, précision et stabilité numérique, et est assuré que l'utilisation d'une version adaptée de ces deux bibliothèques permettra d'obtenir des performances satisfaisantes quelle que soit la machine sous-jacente, qu'elle soit vectorielle, scalaire, super-scalaire, à mémoire partagée [16], ou composée de cartes graphiques et/ou d'autres accélérateurs [15]. Il profite également des améliorations en termes de stabilité numérique ou de flexibilité des algorithmes [46].

3.3.1.2 Retour aux $\mathcal{H}$-Matrices

Les $\mathcal{H}$-Matrices ayant pour but d'effectuer les mêmes opérations que celles proposées dans ces standards, nous proposons d'adopter ce modèle pour construire la bibliothèque $\mathcal{H}$-Matrice. Ainsi la première classe correspond aux niveaux 1 et 2 de BLAS avec les opérations H-AXPY (addition) et H-GEMV (produit matrice-vecteur). La deuxième classe s'appuie sur ces opérations pour proposer H-GEMM (produit matrice-matrice) et H-TRSM (résolution de système triangulaire). Enfin, la dernière classe correspond à des opérations de LAPACK et non plus BLAS, avec H-GETRF pour la factorisation LU et H-GETRI pour l'inversion.

Le parallèle ne s'arrête pas là. De la même façon que les détails du contenu de la matrice sont transparents à tous les niveaux de BLAS [3], la méthode de compression de la matrice est transparente pour toutes les opérations présentées dans la suite de ce chapitre. Les détails des opérations de niveau inférieur sont à leur tour largement transparents aux niveaux supérieurs, bien que cette séparation soit moins étanche que la précédente. Enfin, les opérations d'inversion et de décomposition LU auront une écriture très concise du fait de cette isolation.

Ceci a un impact important pour l'utilisateur. En effet, une des forces du couple BLAS/LAPACK est de permettre au code « utilisateur » d'ignorer les détails de l'implémentation et d'écrire un code performant quelle que soit la machine sans effort. Nous souhaitons préserver cet avantage avec les $\mathcal{H}$-Matrices. Ainsi, le code « utilisateur » ne sera pas différent pour une implémentation séquentielle, parallèle en mémoire partagée ou distribuée, ou sur architecture hybride. De plus, les adaptations dans le code seront mineures et périphériques, ce qui est la grande force de cette méthode.

L'exposition des algorithmes suivra donc ce modèle. Ainsi, nous décrirons en premier les opérations non hiérarchiques (H-BLAS1 et 2), puis les opérations H-BLAS3, et enfin les opérations de plus haut niveau (H-LAPACK). Nous préciserons les noms des fonctions équivalentes de BLAS et LAPACK dans la suite, en y ajoutant le préfixe « H ».

Remarque 3.19 (Pivotage)**.** *Un grand soin est apporté dans* LAPACK *aux diverses techniques de pivotage. En règle générale, toute matrice inversible n'admet pas de décomposition LU, puisqu'un des pivots peut être nul. De plus, pour des raisons de stabilité numérique sur lesquelles nous reviendrons à la section 3.3.3.3, il est souvent nécessaire d'effectuer des permutations sur les lignes et/ou les colonnes de la matrice à factoriser afin d'éviter une accumulation d'erreur catastrophique.*

3. En-dehors des différents types de matrice.

Les algorithmes présentés dans les sections suivantes n'utilisent pas de pivotage, à part localement dans le traitement des blocs diagonaux pleins dans les cas de base des récursions. Ceci n'est en principe pas problématique pour les matrices considérées dans ce document, mais est une différence notable avec d'autres solveurs directs.

3.3.2 Assemblage, Produit matrice-vecteur et Addition

De manière générale, les algorithmes de la première classe évoquée ci-dessus sont de la forme suivante lorsqu'ils opèrent sur une $\mathcal{H}$-Matrice H :

— Si H n'est pas une feuille, faire récursivement les opérations sur les fils de H (cas de récursion) ;

— Sinon, faire l'opération de base (cas de base).

3.3.2.1 Assemblage

Dans le cas de l'assemblage, l'opération de base est l'assemblage d'un bloc. Si le bloc n'est pas admissible, alors l'assemblage est le calcul du bloc matriciel plein (il est nécessairement de petite taille). Si le bloc est admissible, alors le bloc est compressé avec l'algorithme choisi (et il est de taille arbitrairement grande).

Algorithme 3.6 Assemblage d'une $\mathcal{H}$-Matrice

```
function CreateHMatrix(σ × τ)
    if IsLeaf(σ × τ) then
        if IsAdmissible(σ × τ) then
            CompressBlock(σ × τ)                      ▻ ACA ou SVD
        else
            AssembleFullBlock(σ × τ)                  ▻ Bloc non compressé
        end if
    else
        for all (σ′, τ′) ∈ S(σ × τ) do    ▻ Récursion sur les fils dans l'arbre de blocs
            CreateHMatrix(σ′ × τ′)
        end for
    end if
end function
```

Une version en pseudo-code de cet algorithme est donnée par l'algorithme 3.6, l'appel initial à la fonction CreateHMatrix étant fait avec σ et τ les racines des arbres de groupes des degrés de liberté ligne et colonne, soit I et J. Cet algorithme est très similaire à celui de la création d'un arbre de blocs (algorithme 3.2), ce qui est normal, une $\mathcal{H}$-Matrice n'étant qu'un arbre de blocs dont les feuilles sont augmentées soit par une matrice pleine, soit par une $\mathcal{R}k$-Matrice.

Définition 3.20 (Taux de Compression)**.** *Soit $R \in \mathbb{C}^{m\times n}$ une $\mathcal{R}k$-Matrice de rang k. On définit le taux de compression de cette matrice par :*

$$\frac{k(m+n)}{mn}$$

Pour une $\mathcal{H}$-Matrice $H \in \mathbb{C}^{M \times N}$, ce taux est défini par :

$$\tau(H) := \left(\sum_{\sigma \times \tau \ admissible} k(|\sigma| + |\tau|) + \sum_{\sigma \times \tau \ non \ adm} |\sigma||\tau| \right) \times \frac{1}{NM}$$

Ce taux est simplement le rapport de l'espace mémoire requis pour une $\mathcal{H}$-Matrice sur celui nécessaire pour une matrice pleine de même taille. On note que dans le cas d'une $\mathcal{R}k$-Matrice, ce taux peut être supérieur à 1. Celui-ci ne peut être connu à l'avance, puisque nous utilisons les variantes adaptatives des algorithmes, et non celles avec un rang fixe.

3.3.2.2 Quelques optimisations

Nous avons entrevu dans le chapitre précédent que le temps de calcul des algorithmes opérant sur les $\mathcal{H}$-Matrices est très fortement dépendant du rang des blocs compressés. Il est donc logique de chercher à minimiser celui-ci dès l'étape de création de la matrice.

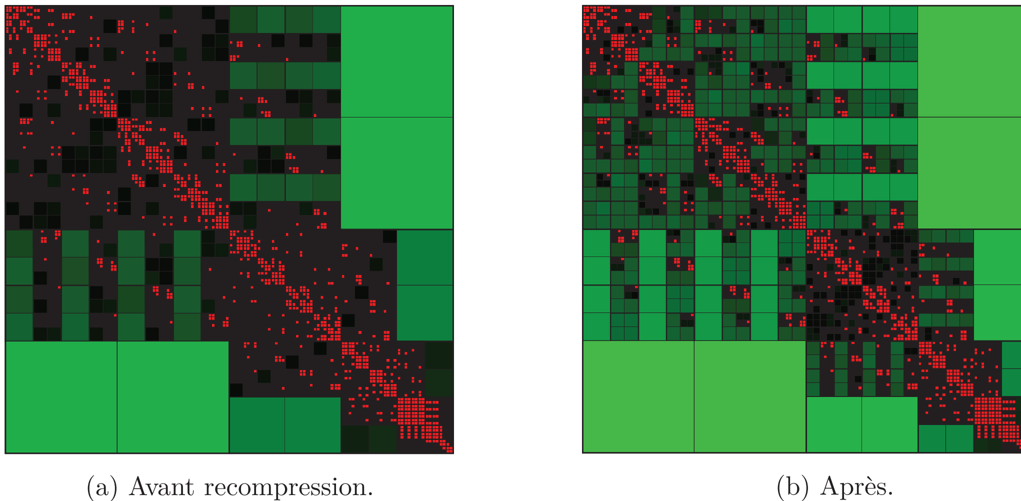
Supposons le rang k d'une $\mathcal{R}k$-Matrice non optimal. Cette matrice entrera comme opérande dans de nombreuses opérations par la suite, ce qui se lit aisément sur l'algorithme 2.1 dans le cas mono-niveau. Ce rang finira par être réduit, après une opération de recompression à la suite d'une addition par exemple. Cependant, si il est possible de réduire le rang du bloc avant de commencer à faire des calculs, le nombre total d'opérations s'en trouve réduit. C'est pour cela que nous présentons ici trois optimisations simples et heuristiques permettant une telle réduction. Les deux premières trouvent leur source dans [56], la troisième a été appliquée indépendamment.

Nous prendrons dans les trois cas l'exemple de la géométrie utilisée dans 3.4. Elle possède un faible nombre de degrés de liberté (9852), ce qui rend ce calcul quelque peu atypique, des exemples plus significatifs seront donnés dans le chapitre dédié. Dans tous les exemples de cette section, la compression utilisée sera ACA avec pivotage partiel, $\varepsilon = 10^{-4}$, une division médiane et $N_{leaf} = 100$.

3.3.2.2.1 Recompression Dans le cas où la méthode de compression utilisée n'est pas la SVD, on procède à une recompression de la matrice immédiatement après son assemblage. Ceci fait dans la majorité des cas baisser le rang, ce qui est naturel du fait de la propriété de meilleure approximation de la SVD. Cette recompression se fait telle qu'elle est décrite dans la section 2.4.2.3 pour la recompression du résultat de l'addition de deux $\mathcal{R}k$-Matrices, avec un coût $\mathcal{O}((m+n)k^2 + k^3)$.

La figure 3.11 illustre la différence entre les deux approches. Sans recompression, la matrice a un taux de compression de 25.21%, et avec le taux passe à 18.40% [4]. L'explication intuitive de la réduction de ce taux est confirmée par le taux de compression de la matrice LU (multi-niveaux), qui est de 19.26% sans recompression et 18.51% avec. Dans les deux cas, les opérations d'addition ont fait chuter le taux dans l'algorithme LU, qui reste presque constant pour la matrice recompressée.

4. La version électronique de ce document contient les figures sous leur forme vectorielle, où tous les blocs sont visibles.



(a) Avant recompression. (b) Après.

FIGURE 3.11 – Comparaison de la compression de la $\mathcal{H}$-Matrice sans et avec recompression. Les blocs compressés sont en vert, un taux de compression moins bon que 20% est représenté en noir, les nuances de vert les plus claires représentant les meilleurs taux.

3.3.2.2.2 Fusion de sous-arbres Le critère d'admissibilité donnant la structure de la matrice est utile pour connaître les approximations des blocs, mais une fois celles-ci déterminées, le découpage obtenu peut ne pas être optimal. Se pose alors la question de la fusion de certaines parties de l'arbre. Cette fusion doit préserver la diagonale de la matrice qui doit toujours rester pleine, et se fait récursivement, en partant du bas de l'arbre :

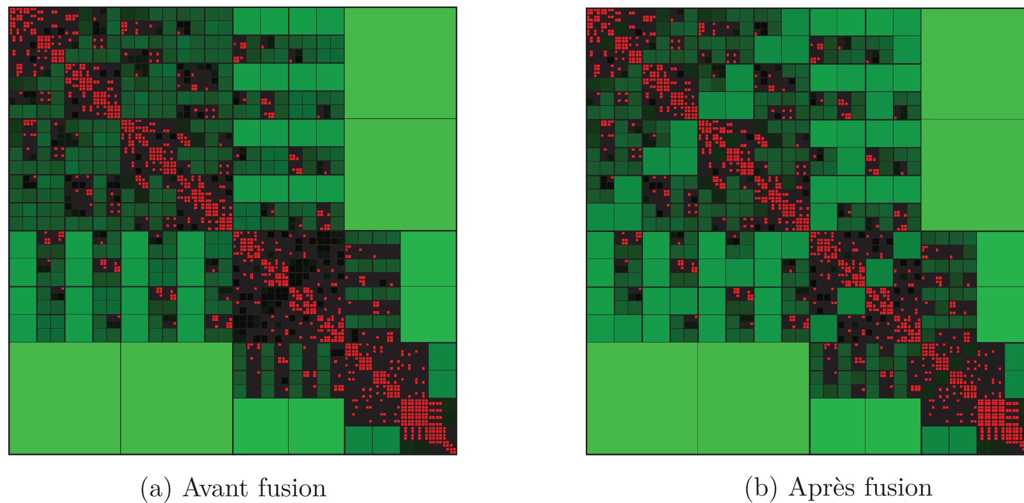
— Si un nœud de la $\mathcal{H}$-Matrice a 4 fils qui sont tous des feuilles compressées ($\mathcal{R}k$-Matrices), on les additionne ensembles dans une seule $\mathcal{R}k$-Matrice. Si la $\mathcal{R}k$-Matrice résultante a un meilleur taux de compression que l'arbre, elle remplace son sous-arbre par une nouvelle feuille compressée.

— On répète cette opération sur le père de la feuille actuelle si une fusion a eu lieu, sinon l'algorithme s'arrête.

La figure 3.12 illustre le résultat sur la même matrice, qui avait été recompressée auparavant. La matrice de gauche est donc la même que celle de la figure 3.11. Le taux de compression passe à 17.7225% avant factorisation.

3.3.2.2.3 Simple précision Les machines actuelles possèdent deux modes de calcul pour les nombres non entiers, appelés simple et double précision (correspondant aux types `float` et `double` dans le langage C). Ces modes de représentation entraînent de nombreuses subtilités [50,64], mais pour l'essentiel il est ici suffisant de savoir que les machines traitent les nombres sous une forme proche de la notation scientifique en base 2, en séparant *mantisse* et *exposant* ($x = 1.mantisse \times 2^{exposant}$ pour $x \neq 0$). Cette notation est celle de la norme IEEE-754. Dans le cas des nombres simple précision, la mantisse est de 23 bits, ce qui une fois traduit en base 10 donne environ 6 chiffres significatifs, contre 16 (52 bits) pour la double précision.

Les $\mathcal{H}$-Matrices opérant une troncature de l'information, il est naturel de penser que l'erreur de troncature est supérieure à l'erreur de représentation des nombres en simple précision, ce qui invite à utiliser ce mode de représentation. Cependant, les termes entrant



(a) Avant fusion (b) Après fusion

FIGURE 3.12 – Matrice avant et après fusion hiérarchique de certaines de ses branches. Les blocs non fusionnés sont identiques entre les deux $\mathcal{H}$-Matrices. La matrice de gauche est identique à la figure 3.11b.

dans la composition de la matrice d'interaction sont sensibles aux erreurs numériques, et sont usuellement toujours calculés en double précision. On procède alors en deux temps :

— Assemblage de la matrice en double précision (y compris l'algorithme ACA) ;

— Conversion en simple précision et suite du calcul.

L'avantage de cette approche est double :

— La consommation mémoire est divisée par deux ;

— Le temps de calcul des opérations sur la $\mathcal{H}$-Matrice est également divisé par deux.

En pratique, le gain en temps de calcul sera plus faible, en particulier sur les cas de faible taille pour lesquels le temps d'assemblage domine. Il faut néanmoins remarquer que du fait de la prédominance du temps de calcul des intégrales dans l'assemblage, la différence de temps entre un passage en simple précision avant ou après l'algorithme ACA est négligeable ; des gains de précision sont en revanche constatés.

Les résultats de ces trois optimisations sont résumés dans le tableau suivant, dans lequel les optimisations sont cumulatives :

	Taux de compression	Temps d'assemblage	Temps de factorisation
Référence	25.31%	27.5s	85.4s
Recompression	18.40%	28.2s	73.9s
+ Fusion	17.72%	28.7s	63.9s
+ Simple précision	17.72%	27.9s	38.6s

Les temps de calcul dans ce tableau sont obtenus en exploitant un cœur d'une machine équipée d'un processeur Xeon « Nehalem » à 2.67GHz. À titre de référence, le solveur direct pour le même problème a nécessité 50.9s d'assemblage et 388.1s de factorisation sur la même machine, alors que la factorisation exploitait la symétrie de la matrice. La

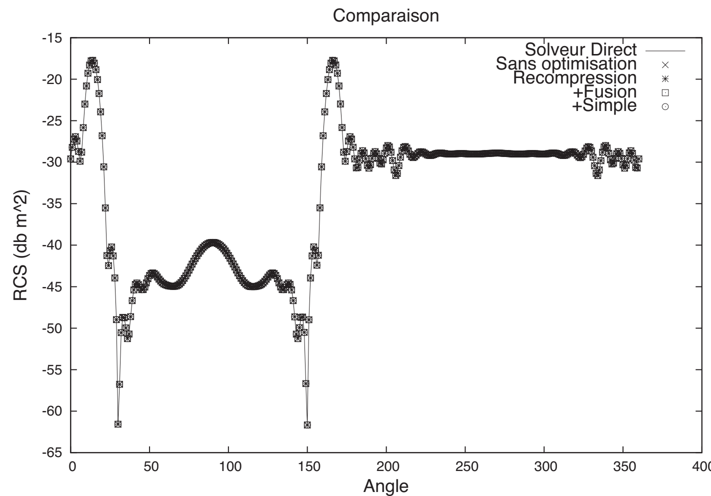


FIGURE 3.13 – Comparaison des résultats. Le solveur direct est en trait plein.

factorisation calculée était dans ce cas une factorisation LDL^T avec un coût deux fois plus faible.

Précisons le calcul effectué. Il s'agit du calcul du champ lointain (2.2.21), et plus précisément de la représentation de $10\log_{10}(|\boldsymbol{E}_{diff}|/|\boldsymbol{E}_{inc}|)$ pour une illumination de l'objet par une onde plane dont le vecteur d'onde tourne autour de l'objet. Dans ce cas, l'objet est soumis à 360 illuminations distinctes, séparées de 1 degré chacune, ce qui mène à 360 seconds membres. Il est important de noter que la dynamique de la réponse est grande, plus de 40 dB ici et jusqu'à 80dB pour des cas de plus grande taille. En ce qui concerne la précision, les résultats sont indiscernables, comme le montre la figure 3.13. Ces figures sont usuellement tracées en coordonnées polaires, mais elle est ici donnée sous cette forme pour permettre de mieux distinguer les écarts.

3.3.2.3 Addition

L'addition de deux $\mathcal{H}$-Matrices correspond à l'opération AXPY de BLAS, et n'opère également que sur les feuilles de la $\mathcal{H}$-Matrice. Elle constitue par ailleurs une étape de perte d'information, comme cela est visible dans le cas de l'addition de deux $\mathcal{R}k$-Matrices section 2.4.2.3.

Soient deux $\mathcal{H}$-Matrices H_1 et H_2 construites sur le même arbre de blocs, c'est-à-dire sur les mêmes arbres de groupes $\mathcal{T}$ et $\mathcal{T}'$. L'opération $H_1 \leftarrow \alpha H_2 + H_1$ est donnée par l'algorithme 3.7, initialisé avec les racines des arbres $\mathcal{T}$ et $\mathcal{T}'$.

On remarque qu'il n'y a que deux cas de base ici, malgré les trois formes que peuvent prendre une $\mathcal{H}$-Matrice. Ceci sera important dans la suite : une $\mathcal{H}$-Matrice peut être soit une $\mathcal{H}$-Matrice (c'est-à-dire un nœud interne de l'arbre de blocs), soit une feuille compressée ($\mathcal{R}k$-Matrice) ou non (matrice pleine). Ici, du fait de la structure partagée par les deux matrices à additionner, $H_1|_{\sigma\times\tau}$ et $H_2|_{\sigma\times\tau}$ sont toujours de la même forme, ce qui simplifie l'algorithme.

Algorithme 3.7 Addition de deux $\mathcal{H}$-Matrices

```
function H-Axpy(H1, H2, α, σ × τ)
    if IsLeaf(σ × τ) then
        H1|σ×τ ← αH2|σ×τ + H1|σ×τ                ▷ Pleine ou compressée
    else
        for all (σ', τ') ∈ S(σ × τ) do
            Axpy(H1, H2, α, σ' × τ')
        end for
    end if
end function
```

Lorsqu'il s'agit de deux matrices pleines, l'opération est faite avec AXPY de BLAS et, dans le cas des deux $\mathcal{R}k$-Matrices, on applique la méthode présentée à la section 2.4.2.3. On rappelle que l'addition de deux $\mathcal{R}k$-Matrices entraîne une recompression adaptative de la matrice résultat, ce qui a pour conséquence de faire varier le rang de celle-ci, sans que cette évolution ne soit prévisible. Ceci est important pour l'implémentation informatique, car les structures de données retenues doivent permettre cette flexibilité. Nous y reviendrons dans le chapitre consacré à l'implémentation parallèle de la méthode.

Définition 3.21 (Addition formatée)**.** *On définit l'addition formatée de deux $\mathcal{H}$-Matrices H_1 et $H_2 \in \mathcal{H}(\mathcal{T}_{I\times J}, k)$ par :*

$$\oplus : \begin{pmatrix} \mathcal{H}(\mathcal{T}_{I\times J}, k) \times \mathcal{H}(\mathcal{T}_{I\times J}, k) & \to & \mathcal{H}(\mathcal{T}_{I\times J}, k) \\ H_1 \oplus H_2 & \mapsto & \mathcal{T}_k(H_1 + H_2) \end{pmatrix}$$

Dans ce cas, l'addition telle que décrite ici est bien l'addition formatée, et donc la meilleure approximation dans l'ensemble des $\mathcal{H}$-Matrices de rang k.

3.3.2.4 Produit Matrice–Vecteur

La dernière opération appartenant à la première classe est le produit matrice-vecteur, analogue à GEMV dans BLAS2, c'est-à-dire $y \leftarrow \alpha Hx + \beta y$, avec H une $\mathcal{H}$-Matrice, $x \in \mathbb{C}^n$ et $y \in \mathbb{C}^m$, et $\alpha, \beta \in \mathbb{C}$.

Étant donné qu'une seule $\mathcal{H}$-Matrice apparaît dans son écriture, il n'y a que deux cas de base, avec une feuille pleine ou compressée. Dans le cas d'une feuille compressée, le produit se fait comme à la section 2.4.2.1 ; il est donné par l'algorithme 3.8.

Cet algorithme appelle quelques observations :

— Une fois la structure de la matrice déterminée (son arbre de blocs) et la matrice assemblée, cet algorithme est très simple à implémenter. Or, cet algorithme est le seul nécessaire pour un solveur itératif. C'est en effet cette unique opération qui est fournie par la méthode FMM. Il est donc aisé de réaliser un solveur itératif avec les $\mathcal{H}$-Matrices.

— Rappelons que les degrés de liberté de la matrice ont été renumérotés à la construction des arbres de groupes. Le produit avec un vecteur non réordonné est (également

Algorithme 3.8 Produit Matrice-vecteur

```
function H-GEMV(H, x, y, σ × τ, α, β)
    if ISROOT(σ) then
        y ← βy
    end if
    if ISLEAF(σ × τ) then
        y|σ ← αM|σ×τ · x|τ + y|σ                ▷ Plein ou compressé
    else
        for all (σ', τ') ∈ S(σ × τ) do
            H-GEMV(H, x, y, σ' × τ' , α, β)
        end for
    end if
end function
```

pour des raisons d'efficacité informatique) fait sur des vecteurs réordonnés. L'algorithme 3.8 est donc en pratique entouré de deux opérations, dans lesquelles les degrés de liberté sont permutés par π (définie dans la remarque 3.1) et π^{-1}.

3.3.3 Multiplication, Inversion, Factorisations

Les opérations suivantes sont plus complexes, parce qu'elles n'opèrent pas nécessairement au niveau le plus bas de l'arbre. Le squelette des algorithmes suivants sera :

— Tant qu'aucune des opérandes n'est une feuille, l'algorithme se poursuit récursivement sur les fils ;

— Sinon, faire une des opérations de base.

Contrairement au cas de l'addition, la multiplication ne nécessite pas que les opérandes partagent la même structure. Il est donc possible qu'une des matrices soit une feuille, et les autres non. Dans ce cas, l'opération se fait à un niveau autre que celui des feuilles.

Les divers cas de récursion des algorithmes que nous présentons ici sont très similaires à l'écriture de ces mêmes algorithmes sur des matrices 2×2 par bloc (dans le cas d'un arbre de groupes binaire). Alors, si A n'est pas une feuille, elle est de la forme :

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \tag{3.3.1}$$

et de même pour B et C. Dans cette écriture, les matrices A_{ij} peuvent être soit des nœuds internes, soit des feuilles compressées ou non. De plus, on note que ces sous-matrices ne sont pas nécessairement de la même taille, puisque si A est une matrice portant les indices $\sigma \times \tau$, et que $S(\sigma) = \{\sigma_1, \sigma_2\}$ et $S(\tau) = \{\tau_1, \tau_2\}$, alors la matrice A_{12} porte les indices $\sigma_1 \times \tau_2$. Il n'est pas nécessaire que $|\sigma_1| = |\tau_2|$ même si $|\sigma| = |\tau|$, et ceci ne sera en général pas vrai.

3.3.3.1 Multiplication

De la même façon que l'addition et le produit matrice-vecteur ont été présentés à l'aide des opérations correspondantes dans BLAS, le cas de la multiplication est celui de

l'opération GEMM, c'est-à-dire $C \leftarrow \alpha AB + \beta C$, pour A, B et C des matrices de tailles compatibles.

La notion de compatibilité pour les $\mathcal{H}$-Matrices est dérivée de celle sur les matrices usuelles. Soient I, J et K des ensembles d'indices et $\mathcal{T}_{\{I,J,K\}}$ les arbres de groupes correspondants. Alors si A est construite sur l'arbre de blocs $\mathcal{T}_{I\times K}$, B sur $\mathcal{T}_{K\times J}$ et C sur $\mathcal{T}_{I\times J}$, les opérations sont compatibles. Ces arbres sont distincts, et dans le cas général, dans l'opération $C \leftarrow \alpha AB + \beta C$, chacune des opérandes matricielles peut être soit une $\mathcal{H}$-Matrice (elle n'est pas une feuille de son arbre de blocs), soit une feuille compressée, soit une feuille pleine. Il y a donc un total de $3^3 = 27$ combinaisons, et seul le cas où aucune des matrices n'est une feuille est un cas de récursion, ce qui mène à un algorithme comportant un cas de récursion et 26 cas de base.

Algorithme 3.9 Produit Matrice-Matrice

```
function H-GEMM(C, α, A, B, β, σ ∈ T_I, τ ∈ T_J, ρ ∈ T_K)
    if ISROOT(C) then
        C ← βC
    end if
    if ¬ISLEAF(σ × τ) et ¬ISLEAF(σ × ρ) et ¬ISLEAF(ρ × τ) then
        for all σ' ∈ S(σ) do
            for all τ' ∈ S(τ) do
                for all ρ' ∈ S(ρ) do
                    H-GEMM(C, α, A, B, 1, σ', τ', ρ')
                end for
            end for
        end for
    else
        C|σ×τ ← αA|σ×ρ · B|ρ×τ + C|σ×τ                ▷ Cas de base
    end if
end function
```

On remarque que la mise à l'échelle par β se fait avant le début de l'algorithme, ce qui justifie le fait que β soit fixé à 1 dans les appels récursifs. Le cas de récursion de cet algorithme correspond au produit matriciel avec des matrices 2×2 par bloc, mais les cas de base sont plus complexes. La difficulté liée à ces cas de base est celle de la structure « naturelle » du produit entre deux nœuds de l'arbre, qui n'est pas nécessairement la même que celle de la matrice dans laquelle le produit doit être stocké. Ceci peut se produire même si les trois matrices A, B et C ont la même structure.

Prenons un exemple, issu de [35]. Fixons $\alpha = 1$, $\beta = 0$ et supposons que A, B et C aient la même structure, de la forme (3.3.1). De plus, supposons que les matrices A_{11}, A_{12} et A_{21} (respectivement B, C) soient de nouveau subdivisées, comme sur la figure 3.14.

Le terme du produit stocké dans C_{22} s'écrit alors :

$$C_{22} \leftarrow A_{21}B_{12} + A_{22}B_{22}$$

Le second terme est au format $\mathcal{R}k$-Matrice et ne pose donc pas de problème. Le premier est en revanche naturellement une $\mathcal{H}$-Matrice et doit être rangé dans une $\mathcal{R}k$-Matrice. Il est donc nécessaire de le convertir, ce que nous présentons maintenant dans la généralité,

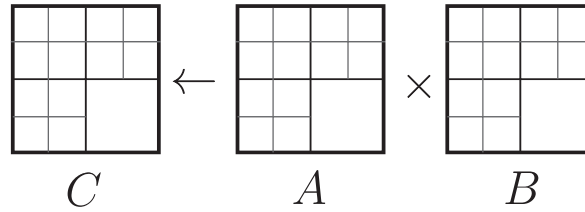
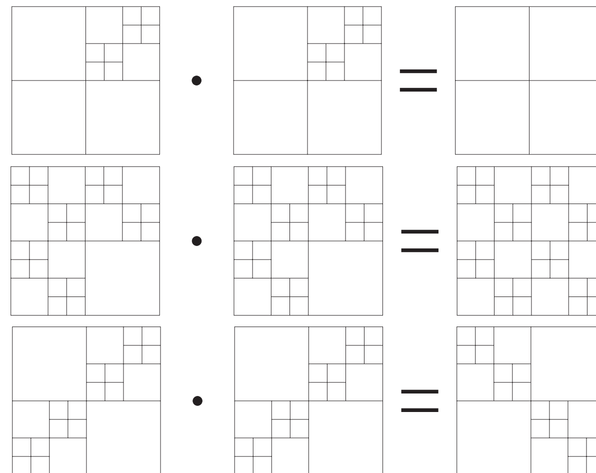


FIGURE 3.14 – Exemple de configuration

c'est-à-dire le cas où la conversion doit se faire sur plusieurs niveaux successifs de la matrice.

Remarque 3.22 (Modification de la structure)**.** *L'exemple précédent n'est pas le seul cas de modification de la structure. Le produit de deux structures d'arbres de blocs peut mener à une structure plus grossière, plus fine, ou complètement différente, et ce même dans le cas où les arbres sont identiques. Trois exemples, reproduits à partir de [35] sont donnés ici :*



3.3.3.1.1 Conversion hiérarchique Pour cela, il sera nécessaire d'additionner à une $\mathcal{R}k$-Matrice R des $\mathcal{R}k$-Matrices dont les indices ligne et colonne sont des sous-ensembles de ceux de R. Cette addition est possible en étendant les sous-matrices par des 0 sur les lignes et les colonnes qui ne font pas partie de leur support. Supposons le support de R égal à $\sigma \times \tau$, et celui de $\tilde{R}$ égal à $\sigma_1 \times \tau_1$, avec $\sigma_1 \subset \sigma$ et $\tau_1 \subset \tau$. L'extension de la matrice R_1 en $\tilde{R}_1$ est :

$$\forall (i,j) \in \sigma \times \tau, (\tilde{R})_{ij} := \begin{cases} R_{ij} & \text{si } (i,j) \in \sigma_1 \times \tau_1 \\ 0 & \text{sinon} \end{cases}$$

Nous pouvons détailler la conversion hiérarchique d'une $\mathcal{H}$-Matrice H en une $\mathcal{R}k$-Matrice R, en partant du bas de l'arbre de H, et en notant le nœud courant de l'arbre de blocs $\sigma' \times \tau'$:

- Si le nœud courant est une feuille :
  - Si elle est admissible, ne rien faire ;

    — Sinon, la convertir dans le format $\mathcal{R}k$-Matrice.
- Sinon, pour tous les fils (qui sont des feuilles admissibles) :
    - Créer une nouvelle $\mathcal{R}k$-Matrice

$$R' := \sum_{(\sigma'',\tau'')\in S(\sigma'\times\tau')} R''_{\sigma''\times\tau''}$$

      avec $R''_{\sigma''\times\tau''}$ les fils étendus comme précédemment.
    - Recompresser cette $\mathcal{R}k$-Matrice.

Cette approximation hiérarchique est rapide, mais non optimale, comme le théorème suivant le montre :

Théorème 3.23 (Erreur de conversion hiérarchique)**.** *Soit $\mathcal{T}_{I\times J}$ un arbre de blocs de profondeur p, et $M \in \mathcal{H}(\mathcal{T}_{I\times J}, k)$ et soit $M_{\mathcal{H}}$ la $\mathcal{R}k$-Matrice obtenue par conversion hiérarchique. Alors on a :*

$$\|M - M_{\mathcal{H}}\|_F \leqslant (2^{p+1} + 1)\,\|M - \mathcal{T}_k(M)\|_F$$

L'erreur commise par cette approximation par rapport à la meilleure troncature possible peut donc être d'un facteur $\mathcal{O}(2^p) = \mathcal{O}(N)$ plus grande. Cependant, cette erreur est en pratique bien plus faible [35], suffisamment pour que cet algorithme soit employé.

3.3.3.1.2 Multiplication et conversion hiérarchique En pratique, la multiplication et la conversion hiérarchique sont faites simultanément. Cette situation se produit lorsque le résultat doit être stocké dans une feuille admissible, et est traitée comme dans l'algorithme 3.10, que nous donnons pour simplifier avec $\beta = 1$.

Algorithme 3.10 Produit et conversion hiérarchique

```
function GemmRk(C, α, A, B, σ ∈ T_I, τ ∈ T_J, ρ ∈ T_K)
    if ¬IsLeaf(A) et ¬IsLeaf(B) then
        for all σ' ∈ S(σ) do
            for all τ' ∈ S(τ) do
                R'_{σ'×τ'} ← 0                                   ▷ Rk-Matrice temporaire
                for all ρ' ∈ S(ρ) do
                    GemmRk(R'_{σ'×τ'}, α, A, B, σ', τ', ρ')
                end for
            end for
        end for
        C ← C + Σ_{(σ',τ')∈S(σ×τ)} R'_{σ'×τ'}   ▷ Addition des sous-matrices et recompression
    else                                         ▷ Une des matrices est une feuille
        C ← αA · B + C                                           ▷ Cas de base
    end if
end function
```

3.3.3.1.3 Cas de base Donnons une description systématique des cas de base. Nous les classons ici en fonction de la structure de C, qui dirige le produit, comme ceci est visible dans le cas où C est une feuille compressée. Nous avons donc :

- C est une $\mathcal{H}$-Matrice :
  - A ou B est une matrice pleine : Si l'autre est une $\mathcal{H}$-Matrice, la multiplication se fait comme dans le cas de la multiplication d'une $\mathcal{H}$-Matrice par un vecteur, colonne par colonne (ou ligne par ligne). Le résultat a la même structure que l'autre matrice, et est ensuite additionné à C.
  - A ou B est une $\mathcal{R}k$-Matrice : idem.
- C est une matrice pleine :
  - L'opération se fait avec sa structure résultat naturelle.
  - Conversion en matrice pleine, puis addition à C.
- C est une matrice compressée : ce cas a été traité précédemment.

Remarque 3.24 (Transposition)**.** *Dans* BLAS3*, les matrices A et B peuvent être transposées dans le produit. Ceci est aisément implémentable pour le produit de $\mathcal{H}$-Matrices, en remarquant que :*

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}^T = \begin{pmatrix} A_{11}^T & A_{21}^T \\ A_{12}^T & A_{22}^T \end{pmatrix}$$

3.3.3.2 Inversion

Une fois le produit et l'addition des $\mathcal{H}$-Matrices énoncés, un algorithme adapté pour l'inversion se déduit facilement. En effet, les sections précédentes nous donnent tous les outils pour pouvoir considérer les $\mathcal{H}$-Matrices à un niveau plus élevé comme des matrices par bloc 2×2 dans le cas de la partition binaire de l'espace.

Prenons l'exemple d'une matrice $M \in \mathbb{C}^{N \times N}$ inversible dont l'écriture est de la forme, avec les matrices A, B, C et D de tailles non nécessairement égales :

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Alors, en notant $S = D - CA^{-1}B$ le complément de Schur, l'inverse de M a pour expression :

$$M^{-1} = \begin{pmatrix} A^{-1} + A^{-1}BS^{-1}CA^{-1} & -A^{-1}BS^{-1} \\ -S^{-1}CA^{-1} & S^{-1} \end{pmatrix}$$

Cette écriture trouve une correspondance directe dans le cas où M est une $\mathcal{H}$-Matrice, les calculs de l'inverse du bloc A et du complément de Schur étant faits par récursion, ce qui se lit dans l'algorithme 3.11. Celui-ci effectue une inversion sur place, mais utilise une matrice X comme espace de stockage temporaire. Cette matrice est construite sur le même arbre de blocs que M et est initialisée à 0.

Dans les commentaires de cet algorithme, il est indiqué que le cas de base correspond nécessairement à une matrice non compressée. Ceci est important, car il n'est pas possible d'inverser une $\mathcal{R}k$-Matrice, car son rang est déficient.

Les seuls appels récursifs à la fonction d'inversion sont faits sur la diagonale de la $\mathcal{H}$-Matrice, c'est-à-dire pour des sous-arbres de la forme $\sigma \times \sigma$, avec $\sigma \in \mathcal{T}_I$. Le cas de

Algorithme 3.11 Inversion d'une $\mathcal{H}$-Matrice. A est écrasée par A^{-1} au retour de la fonction.

```
function Inverse(M, X)
    if IsLeaf(A) then
        M ← M^{-1}    ▷ M est nécessairement une matrice pleine, inversion avec lapack
    else
        M := ( M11  M12 ; M21  M22 ) et X := ( X11  X12 ; X21  X22 )
        Inverse(M11, X11)                          ▷ M11 ← M11^{-1}
        H-Gemm(X12, −1, M11, M12, 0)               ▷ X12 ← −M11 M12
        H-Gemm(X21, 1, M21, M11, 0)                ▷ X21 ← M21 M11
        H-Gemm(M22, 1, M21, X12, 1)                ▷ M22 ← M22 + M21 X12 := S
        Inverse(M22, X22)                          ▷ M22 ← M22^{-1}
        H-Gemm(M12, 1, X12, M22, 0)                ▷ M12 ← X12 M22
        H-Gemm(M11, −1, M12, X21, 1)               ▷ M11 ← M11 − M12 X21
        H-Gemm(M21, −1, M22, X21, 0)               ▷ M21 ← −M22 X21
    end if
end function
```

base de cette récursion est donc bien une feuille non admissible, car la distance entre un groupe de degrés de liberté et lui-même est nulle. Il n'y a ainsi qu'un seul cas de base, et les seules matrices à inverser sont bien inversibles.

3.3.3.3 Factorisation LU

La méthode usuelle de résolution d'un système linéaire $Ax = b$ n'emploie pas l'inversion, mais procède en deux étapes, avec, tout d'abord, une factorisation de la matrice sous une forme rendant aisée la résolution du système linéaire [51, 97]. Ceci est fait pour plusieurs raisons. En premier lieu, cette méthode est plus rapide, avec un nombre d'opérations asymptotiquement deux fois plus faible [25] pour la factorisation LU d'une matrice dense. Ensuite, elle est plus stable numériquement [64], et enfin ne nécessite pas de stockage temporaire dans notre cas.

Il existe diverses formes adaptées à la résolution de systèmes linéaires. Celle que nous détaillons ici est la factorisation LU, également appelée élimination gaussienne du fait de son rapport avec la méthode dite « du pivot de Gauss ». Toute matrice carrée de rang plein $A \in \mathbb{C}^{n\times n}$ peut s'écrire sous la forme $A = P^{-1}LU$, avec P une matrice de permutation, L une matrice triangulaire inférieure, et U une matrice triangulaire supérieure, une des ces deux matrices ayant une diagonale composée de 1. La résolution du système linéaire $Ax = b$ est remplacée par celle de $P^{-1}LUx = b$, ce qui se fait par deux résolutions successives de systèmes triangulaires.

Il est connu [51, 64, 97] que cette méthode est instable numériquement dans le cas $P = I_n$, ce qui est un problème, traité par le pivotage, c'est-à-dire l'utilisation d'une matrice de permutation P adaptée. La version sans pivotage satisfait le théorème suivant [97] :

Théorème 3.25. *Supposons la décomposition $A = LU \in \mathbb{C}^{n\times n}$ d'une matrice non sin-*

gulière A calculée par élimination gaussienne sans pivotage. Alors, sous des hypothèses raisonnables sur la machine calculant la décomposition (vérifiée par les machines implémentant la norme IEEE-754), la décomposition calculée en virgule flottante $\tilde{L}\tilde{U}$ vérifie :

$$\tilde{L}\tilde{U} = A + \delta A \quad \textit{avec} \quad \frac{\|\delta A\|}{\|L\|\|U\|} = \mathcal{O}(\varepsilon_{machine})$$

On note que le dénominateur est $\|L\|\|U\|$, et non $\|A\|$. En règle générale, $\|L\|\|U\| \neq \mathcal{O}(\|A\|)$, ce qui est la raison de l'instabilité.

Cependant, dans le cas du pivotage partiel :

Théorème 3.26 (Pivotage partiel)**.** *Dans le cas d'une décomposition $A = P^{-1}LU$ avec P une matrice de permutation telle que calculée lors de l'algorithme d'élimination gaussienne avec pivotage partiel, les matrices en virgule flottante $\tilde{P}$, $\tilde{L}$ et $\tilde{U}$ sont telles que :*

$$\tilde{L}\tilde{U} = \tilde{P}A + \delta A \quad \textit{avec} \quad \frac{\|\delta A\|}{A} = \mathcal{O}(\rho\varepsilon_{machine})$$

et ρ un facteur de croissance borné par 2^{n-1} (avec possibilité d'égalité).

Cet algorithme est stable pour n fixé puisque le facteur de croissance de l'erreur ρ est borné, cependant celui-ci est en réalité tellement grand que ce résultat n'est pas très utile en pratique. Néanmoins, cet algorithme est universellement employé, avec très peu de problèmes. Cette digression est importante, car elle a un lien avec la discussion sur la croissance de l'erreur pour l'algorithme ACA avec pivotage partiel, qui est étroitement lié à l'élimination gaussienne avec pivotage partiel [22].

Dans la suite, nous discuterons de l'élimination gaussienne sans pivotage, qui est stable pour les algorithmes par blocs dans le cas où la matrice est symétrique définie positive ou à diagonale dominante par ligne ou colonne [64, chapitre 13] [44, 45].

Algorithme 3.12 Décomposition LU, H-GETRF (sans pivotage). M est écrasée par les facteurs L et U.

```
function LUDECOMPOSITION(M)
    if ISLEAF(M) then
        M ← LU                         ▷ M est pleine, utilisation de LAPACK (GETRF)
    else
        M := ( M11  M12 ; M21  M22 ), L := ( L11  0 ; L21  L22 ), U := ( U11  U12 ; 0  U22 )
        LUDECOMPOSITION(M11)           ▷ M11 ← L11 U11 (H-GETRF)
        Solve L11 U12 = M12            ▷ Système triangulaire inférieur (H-TRSM)
        Solve L21 U11 = M21            ▷ Système triangulaire supérieur (H-TRSM)
        M22 ← M22 − L21 U12            ▷ Complément de Schur (H-GEMM)
        LUDECOMPOSITION(M22)           ▷ M22 ← L22 U22 (H-GETRF)
    end if
end function
```

L'algorithme 3.12 de factorisation LU que nous donnons ici est une adaptation directe de [44] aux opérations sur les $\mathcal{H}$-Matrices. Il nécessite de pouvoir faire la résolution de

systèmes triangulaires matriciels inférieurs ($LX = B$) et supérieurs ($XU = B$), ce qui est également nécessaire pour la descente et la remontée.

Algorithme 3.13 Résolution de $LX = B$ (H-Trsm), avec X écrasant B

```
function SolveLowerTriangular(L, B)
    if ¬ IsLeaf(L) ∧ ¬ IsLeaf(X) then
        L := ( L11  0 ; L21  L22 ), B := ( B11  B12 ; B21  B22 )
        SolveLowerTriangular(L11, B11)                          ▷ H-TRSM
        SolveLowerTriangular(L11, B12)                          ▷ H-TRSM
        B21 ← B21 − L21 B11                                     ▷ H-GEMM
        SolveLowerTriangular(L22, B21)                          ▷ H-TRSM
        B22 ← B22 − L21 B12                                     ▷ H-GEMM
        SolveLowerTriangular(L22, B22)                          ▷ H-TRSM
    else
        if IsLeaf(B) then
            Résolution colonne par colonne  ▷ 2 cas selon si B est pleine ou Rk-Matrice
        else
            Conversion de B en matrice pleine
            Résolution colonne par colonne              ▷ L est une matrice pleine
        end if
    end if
end function
```

L'algorithme 3.13 donne l'implémentation de la résolution d'un système triangulaire inférieur, le cas supérieur étant similaire. Dans les deux algorithmes précédents, le nombre de cas de base est réduit, du fait de la récursion ne se poursuivant que sur les blocs diagonaux, tout comme pour l'inversion.

Remarque 3.27 (Inversion)**.** *Il est possible d'implémenter une version plus efficace de l'inversion d'une $\mathcal{H}$-Matrice que l'algorithme 3.11 en utilisant en premier une décomposition LU, comme ceci est fait dans* lapack *avec le couple de fonctions* getrf/getri*. Ceci est détaillé dans [35].*

3.3.3.4 Factorisation LDL^T

Dans le cas où la matrice est symétrique (mais non définie positive), il est possible de décomposer une matrice $A \in \mathbb{C}^{n \times n}$ sous la forme $M = LDL^T$ avec $L \in \mathbb{C}^{n \times n}$ une matrice triangulaire inférieure dont la diagonale est constituée de 1, et D une matrice diagonale [51].

Cette décomposition peut se comprendre comme une variante de la décomposition de Cholesky dans le cas où la matrice est de plus définie positive. En effet, si $A = LDL^T$, alors $A = \tilde{L}\tilde{L}^T$, avec $\tilde{L} := L\sqrt{D}$, puisque dans ce cas les éléments de la diagonale de A sont des nombres positifs. Elle peut aussi se comprendre comme une généralisation de la décomposition LU.

Comme précédemment, la première étape est d'exprimer récursivement la décomposition LDL^T d'une matrice bloc 2×2, ce qui permet ensuite de donner un algorithme récursif

pour la décomposition LDL^T d'une $\mathcal{H}$-Matrice. Soit $M \in \mathbb{C}^{n\times n}$ une matrice symétrique, telle que :

$$M = \begin{pmatrix} M_{11} & M_{21}^T \\ M_{21} & M_{22} \end{pmatrix}$$

L'équation matricielle :

$$M = \begin{pmatrix} M_{11} & M_{21}^T \\ M_{21} & M_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} D_{11} & 0 \\ 0 & D_{22} \end{pmatrix} \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}^T$$

donne alors :

$$\begin{cases} M_{11} = L_{11}D_{11}L_{11}^T \\ M_{21} = L_{21}D_{11}L_{11}^T = L_{21}(L_{11}D_{11})^T \\ M_{22} = L_{21}D_{11}L_{21}^T + L_{22}D_{22}L_{22}^T \end{cases}$$

ce qui fournit la base de l'algorithme récursif de calcul. On remarque que la première équation est directement un appel récursif à la décomposition LDL^T, et la seconde en est un, après un produit matriciel. La deuxième équation est une résolution de système triangulaire. On note que dans cette équation les termes L_{11} et L_{22} sont des matrices triangulaires inférieures, et D_{11} et D_{22} des matrices diagonales.

Pour l'algorithme analogue opérant sur les $\mathcal{H}$-Matrices, le cas de base de la récursion de la décomposition est une fois de plus nécessairement une matrice pleine, et est traité classiquement. On introduit par ailleurs trois nouvelles opérations :

MDMTPRODUCT(H, M, D) Effectue l'opération $H \leftarrow H - MDM^T$, avec H et M des $\mathcal{H}$-Matrices, et D une matrice diagonale ;

MULTIPLYWITHDIAG(M, D) Effectue l'opération $M \leftarrow MD$ avec M une $\mathcal{H}$-Matrice et D une matrice diagonale ;

TRANSPOSE(M) Effectue l'opération $M \leftarrow M^T$ avec M une $\mathcal{H}$-Matrice. On note que cette opération modifie potentiellement la structure de M dans le cas où l'arbre de groupes de M n'est pas de la forme $\mathcal{T}_{I\times I}$. Par ailleurs, cette opération a un coût faible car la transposition d'une $\mathcal{R}k$-Matrice $R = AB^T$ est $R^T = BA^T$, qui ne nécessite informatiquement que l'échange de deux pointeurs, et ce à un coût constant. La transposition des feuilles pleines se fait de manière classique.

Les deux dernières opérations sont peu coûteuses et simples, la première nécessite plus de détails :

$$\begin{aligned} H - MDM^T =& H - \begin{pmatrix} M_{11} & M_{21}^T \\ M_{21} & M_{22} \end{pmatrix} \begin{pmatrix} D_{11} & 0 \\ 0 & D_{22} \end{pmatrix} \begin{pmatrix} M_{11} & M_{21}^T \\ M_{21} & M_{22} \end{pmatrix}^T \\ =& H - \begin{pmatrix} M_{11}D_{11}M_{11}^T + M_{12}D_{22}M_{12}^T & \cdot \\ M_{21}D_{11}M_{21}^T + M_{22}D_{22}M_{12}^T & M_{21}D_{11}M_{21}^T + M_{22}D_{22}M_{22}^T \end{pmatrix} \end{aligned}$$

H étant une matrice symétrique, le résultat de cette opération l'est aussi, et la modification de H_{12} n'est donc pas écrite. L'algorithme 3.14 décrit cette opération, dans lequel les divers cas de base ne sont pas détaillés. On note que pour la structure de H, le seul cas de base possible est une matrice pleine, puisque les appels récursifs sont faits sur la diagonale de H. Les cas de base sont donc :

Algorithme 3.14 $H \leftarrow H - MDM^T$

function MdmtProduct(H, M, D)
  if $\neg$ IsLeaf(H) $\wedge$ $\neg$ IsLeaf(M) **then**
    $H := \begin{pmatrix} H_{11} & H_{21}^T \\ H_{21} & H_{22} \end{pmatrix}$, $M := \begin{pmatrix} M_{11} & M_{21}^T \\ M_{21} & M_{22} \end{pmatrix}$, $M := \begin{pmatrix} D_{11} & 0 \\ 0 & D_{22} \end{pmatrix}$
    MdmtProduct(H_{11}, M_{11}, D_{11})
    MdmtProduct(H_{11}, M_{12}, D_{22})
    $X \leftarrow M_{21}$ ▷ Copie
    MultiplyWithDiag(X, D_{11})
    Gemm('N', 'T', H_{21}, -1, X, M_{11}, 1) ▷ $H_{21} \leftarrow H_{21} - X^T M_{11}$
    $Y \leftarrow M_{22}$ ▷ Copie
    MultiplyWithDiag(Y, D_{22})
    Gemm('N', 'T', H_{21}, -1, Y, M_{12}, 1) ▷ $H_{21} \leftarrow H_{21} - Y^T M_{12}$
    MdmtProduct(H_{22}, M_{21}, D_{11})
    MdmtProduct(H_{11}, M_{22}, D_{22})
  else
    $H \leftarrow H - MDM^T$ ▷ Cas de base
  end if
end function

— H est une matrice pleine : alors M et D sont aussi des feuilles.
— M est une feuille, pleine ou compressée.

Cette opération utilise des copies temporaires de données. Il serait possible de les éviter, en implémentant une opération de la forme $M_1 D M_2^T$, ce qui n'a pas été fait ici pour simplifier. Par ailleurs, on remarque l'utilisation d'une extension de la fonction Gemm de l'algorithme 3.9 pour gérer les produits dont les opérandes sont transposées, avec les mêmes notations que blas (`'N'` pour « non transposé », `'T'` pour « transposé »). La transposition d'une $\mathcal{H}$-Matrice étant une opération simple, l'extension du produit aux transpositions peut se faire soit par transposition des opérandes, soit en adaptant l'implémentation du produit. C'est cette seconde solution qui a été retenue.

Algorithme 3.15 Décomposition LDL^T de M (H-Sytrf, sans pivotage), $\mathcal{H}$-Matrice.

function LdltDecomposition(M)
  if IsLeaf(M) **then**
    $M \leftarrow LDL^T$ ▷ M est pleine
  else
    $M := \begin{pmatrix} M_{11} & M_{21}^T \\ M_{21} & M_{22} \end{pmatrix}$, $L := \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}$, $D := \begin{pmatrix} D_{11} & 0 \\ 0 & D_{22} \end{pmatrix}$
    LdltDecomposition(M_{11})
    $X \leftarrow M_{11}$, MultiplyWithDiag(X, D_{11}), Transpose(X) ▷ $X \leftarrow (M_{11} D_{11})^T$
    SolveUpper(X, H_{21}) ▷ $UX = B, U = X$
    MdmtProduct(M_{22}, M_{21}, M_{11})
    LdltDecomposition(M_{22})
  end if
end function

La résolution après une factorisation LDL^T doit également être adaptée, avec une

étape « diagonale » entre la descente et la remontée, et l'adaptation de la résolution pour résoudre $L^T x = b$.

Les avantages de la décomposition LDL^T sont :

Nombre d'opérations Dans le cas des matrices denses, le nombre d'opérations requises pour une décomposition LDL^T est environ la moitié de celui nécessaire pour une décomposition LU.

Quantité de mémoire Seule la partie inférieure de la matrice est stockée (plus la diagonale), ce qui divise la consommation mémoire par un facteur 2. Cependant, dans l'implémentation présentée ici, les copies temporaires réduisent cette économie.

3.4 Complexité

Cette section donne les estimations de complexité des opérations sur les $\mathcal{H}$-Matrices, telles qu'elles apparaissent dans [55, 57]. En particulier, nous supposons dans toute cette section uniquement que le rang k de l'approximation des $\mathcal{R}k$-Matrices est fixe et identique pour tous les blocs admissibles.

3.4.1 Estimations

Une partie des estimations est basée sur la notion de constante de rareté (*sparsity constant* en anglais), qui est une caractéristique de l'arbre de blocs, dont nous donnons la définition :

Définition 3.28 (*Sparsity Constant*)**.** *Soit $\mathcal{T}_{I\times J}$ un arbre de blocs construit à partir des arbres de groupes $\mathcal{T}_I$ sur I et $\mathcal{T}_J$ sur J. Alors la constante de rareté C_{sp} est :*

$$C_{sp} := \max\left\{\max_{\sigma\in\mathcal{T}_I} |\{\tau \in \mathcal{T}_J \mid \sigma\times\tau \in \mathcal{T}_{I\times J}\}|\,, \max_{\tau\in\mathcal{T}_J} |\{\sigma \in \mathcal{T}_I \mid \sigma\times\tau \in \mathcal{T}_{I\times J}\}|\right\} \tag{3.4.1}$$

Cette constante peut se lire simplement sur une représentation de l'arbre de blocs, par exemple celui de la figure 3.7 page 75. Toute subdivision verticale, frontière d'un bloc, est un élément de $\mathcal{T}_I$, et tout bloc (même subdivisé) est un élément de $\mathcal{T}_{I\times J}$. La constante de rareté est le nombre maximal de blocs rencontrés en suivant une division horizontale ou verticale de cette représentation.

L'espace nécessaire (en nombre d'éléments) pour une $\mathcal{H}$-Matrice avec un rang fixe k est donné par :

Lemme 3.29 (Espace mémoire)**.** *Soit $\mathcal{T}_{I\times I}$ un arbre de blocs construit à partir de l'arbre de groupes $\mathcal{T}_I$, de profondeur p et constante de rareté C_{sp}. Alors, pour une matrice $M \in \mathcal{H}(\mathcal{T}_{I\times I}, k)$, l'espace mémoire est borné par :*

$$N_{st}(\mathcal{T}_{I\times I}, k) \leqslant 2C_{sp}(p+1)\max\{k, N_{leaf}\}|I|$$

et le produit matrice-vecteur satisfait l'inégalité suivante :

Lemme 3.30 (Produit Matrice-vecteur)**.** *Sous les hypothèses précédentes, la complexité temporelle du produit matrice vecteur est telle que :*

$$N_{\mathcal{H}\cdot v}(\mathcal{T}_{I\times I}, k) \leqslant 2N_{st}(\mathcal{T}_{I\times I}, k)$$

Il convient de remarquer que cette estimation est de même nature que celle du produit matrice-vecteur pour les matrices non compressées. En effet, dans ce cas, il faut $\mathcal{O}(N^2)$ opérations pour une matrice de taille $N \times N$, et une quantité de mémoire N^2. Ceci est naturel car toutes les opérations entrant dans la composition du produit matrice-vecteur sont les mêmes que pour les matrices pleines, bloc par bloc.

L'opération de conversion hiérarchique intervient dans le produit de $\mathcal{H}$-Matrices, et est coûteuse, comme nous l'avons mentionné section 3.3.3.1.1. Son coût est estimé par :

Lemme 3.31 (Approximation hiérarchique)**.** *Avec les mêmes hypothèses que précédemment, et en notant p la hauteur de $\mathcal{T}_{I\times I}$, C_{fils} le nombre maximal de fils d'un nœud de $\mathcal{T}_I$ (2 dans la plupart des cas), et en supposant pour simplifier $n_{min} \leqslant k$, on a :*

$$N_{\mathcal{H}app}(\mathcal{T}_{I\times I}) \leqslant K_1 C_{sp} \max\{4, C_{fils}^2\}(p+1)k^2|I| + K_2 C_{fils}^3 k^3 |\mathcal{T}_{I\times I}|$$

avec K_1 et K_2 des constantes. Et par ailleurs,

$$|\mathcal{T}_{I\times I}| \leqslant 2C_{sp}|I|$$

Remarque 3.32 (Constantes)**.** *Dans [35], des valeurs explicites sont données pour K_1 et K_2 (12 et 23). Nous ne les donnons pas ici, car ces valeurs ne sont valables que pour des calculs sur les nombres réels, et sous réserve que le calcul de la SVD ait une constante de son terme dominant égale à 23. K_2 est toutefois liée à la constante intervenant dans la complexité de la SVD.*

De même, dans le cas de l'addition, avec dans [35] $K_3 = 24$ et $K_4 = 184$:

Lemme 3.33 (Addition)**.** *Avec les mêmes hypothèses que précédemment, la complexité de l'addition est majorée par (en notant $\mathcal{L}(\mathcal{T}_{I\times I})$ l'ensemble des feuilles de cet arbre) :*

$$N_+(\mathcal{T}_{I\times I}, k) \leqslant K_3 k N_{st}(\mathcal{T}_{I\times I}, k) + K_4 k^3 |\mathcal{L}(\mathcal{T}_{I\times I})|$$

On note que pour l'addition et la conversion hiérarchique, les constantes associées aux termes des estimations sont grandes. Ces constantes sont liées pour K_2 à la décomposition en valeurs singulières, et pour K_4 à la recompression des blocs, laquelle fait intervenir plusieurs opérations coûteuses (*cf.* section 2.4.2.3).

L'estimation de la complexité temporelle de la multiplication est plus délicate. En effet, comme cela a été souligné à la section 3.3.3.1, la structure naturelle du produit de deux $\mathcal{H}$-Matrices peut être très différente de sa structure de destination, ce qui nécessite de coûteuses conversions. On introduit donc une constante d'idempotence :

Définition 3.34 (Constante d'idempotence)**.** *Soit $\mathcal{T}_{I\times I}$ un arbre de blocs basé sur $\mathcal{T}_I$. On définit la constante d'idempotence $C_{id}(\sigma \times \tau)$ par bloc et la constante d'idempotence $C_{id}(\mathcal{T}_{I\times I})$ par :*

$$C_{id}(\sigma\times\tau) := \left|\{\sigma' \times \tau' \mid \sigma' \in S(\sigma), \tau' \in S(\tau) \text{ et } \exists \rho' \in \mathcal{T}_I \text{ tel que } \sigma' \times \rho' \in \mathcal{T}_{I\times I}, \rho' \times \tau' \in \mathcal{T}_{I\times I}\}\right|$$

et

$$C_{id}(\mathcal{T}_{I\times I} := \max_{\sigma\times\tau\in\mathcal{L}(\mathcal{T}_{I\times I})} C_{id}(\sigma\times\tau)$$

Cette constante décrit le nombre de termes entrant dans la composition d'un terme du produit matriciel, ce qui explique son rôle dans l'estimation suivante :

Théorème 3.35 (Multiplication exacte)**.** *Soit $\mathcal{T}_{I\times I}$ comme précédemment, de hauteur p, avec les constantes de rareté et d'idempotence C_{sp} et C_{id} associées. On suppose de plus $N_{leaf} \leqslant k$. La multiplication exacte est une application $\mathcal{H}(\mathcal{T}_{I\times I}, k) \times \mathcal{H}(\mathcal{T}_{I\times I}, k) \to \mathcal{H}(\mathcal{T}_{I\times I}, \tilde{k})$, avec le rang $\tilde{k}$ tel que :*

$$\tilde{k} \leqslant C_{id}(\mathcal{T}_{I\times I})C_{sp}(\mathcal{T}_{I\times I})(p+1)k$$

Cette multiplication exacte correspond à une mutiplication sans recompression des blocs, ce qui explique l'augmentation du rang et le rôle de la constante d'idempotence.

Finalement, l'estimation suivante est valable pour la multiplication :

Théorème 3.36 (Multiplication)**.** *La multiplication formatée (avec recompression) est une application $\mathcal{H}(\mathcal{T}_{I\times I}, k)\times\mathcal{H}(\mathcal{T}_{I\times I}, k) \to \mathcal{H}(\mathcal{T}_{I\times I}, k)$ dont la complexité temporelle $N_\times(\mathcal{T}_{I\times I}, k)$ satisfait l'estimation suivante :*

$$N_\times(\mathcal{T}_{I\times I}, k) \leqslant K_5 C_{sp}^3 C_{id}^3 k^3 (p+1)^3 \max\{|I|, |\mathcal{L}(\mathcal{T}_{I\times I})|\}$$

Il est, de plus, aisé de montrer que la complexité de l'inversion et de la décomposition *LU* sont bornées par celles de la multiplication formatée.

3.4.2 Structure de l'arbre et constantes

La discussion précédente de la complexité des opérations formatées sur les $\mathcal{H}$-Matrices dépend fortement de la valeur des constantes de rareté C_{sp} et d'idempotence C_{id} de l'arbre de blocs, ainsi que de la hauteur de cet arbre. Ceci est un artifice lié au choix d'une approximation de rang fixe, mais se retrouve en pratique dans le cas des approximations adaptatives.

La structure de l'arbre est donc importante, et plusieurs choix sont possibles, comme ceux présentés à la section 3.1. L'objectif étant d'obtenir un arbre de groupes de faible hauteur (donc large plutôt que profond), et tel que l'arbre de blocs associé ait une constante de rareté et d'idempotence faible.

Dans [35], le cas d'un arbre géométriquement équilibré pour l'arbre de groupes est décrit. Il s'agit en dimension 2 d'un *quadtree*, et en dimension 3 d'un *octree*. Cet arbre est construit en partant d'un cube englobant l'objet, puis en divisant ce cube au milieu de chacune de ses dimensions, récursivement. Cette construction est identique à celle fréquemment utilisée pour la FMM.

En associant à cet arbre de groupes un arbre de blocs construit avec le critère :

$$\min\{diam(\sigma), diam(\tau)\} \leqslant \eta \cdot d(\sigma,\tau)$$

qui est celui que nous utilisons, il est possible de prouver :

Théorème 3.37 (Constantes pour un arbre géométriquement équilibré)**.** *Notons Ω_i le support de la fonction de base associée au degré de liberté $i \in I$. On suppose qu'il existe deux constantes C_{sep} et N_{leaf} telles que :*

$$\max_{i \in \mathcal{I}} \left| \{ j \in I \mid d(\Omega_i, \Omega_j) \leqslant C_{sep}^{-1} diam(\Omega_i) \} \right| \leqslant N_{leaf}$$

On a alors, en notant $h := \min_{i \in I}(diam(\Omega_i))$ et H le côté du cube englobant :

— *La hauteur de l'arbre de blocs est bornée par :*

$$h(\mathcal{T}_{I \times I}) \leqslant 1 + \log_2 \left((1 + 2C_{sep}) \sqrt{2} \frac{H}{h_{min}} \right)$$

— *La constante de rareté est bornée par :*

$$C_{sp} \leqslant \left(2 + 8\sqrt{2} C_{sep} + 4\sqrt{2}\eta^{-1}(1 + 2C_{sep}) \right)^2$$

— *La constante d'idempotence est bornée par :*

$$C_{id} \leqslant \left(2 + 4C_{sep} + 2\eta(1 + 2\sqrt{2}C_{sep}) \right)^4$$

L'hypothèse de ce théorème affirme qu'il existe un nombre limité de fonctions de base ayant leurs supports « proches », ce qui est vrai dans le cas des éléments finis de frontière sur une triangulation.

Les estimations de ce théorème montrent qu'il est possible de construire un arbre de groupes tel que l'arbre de blocs associé a des constantes de rareté et d'idempotence *indépendantes de* $|I|$, et que la hauteur d'un tel arbre croît avec le logarithme de $|I|$. Dans ce cas, les termes « p » peuvent êtres majorés par un terme grandissant avec le logarithme de $N = |I|$ dans les estimations de la section précédente. Ceci permet donc de retrouver la complexité annoncée en $\mathcal{O}(N \log_2^{\alpha}(N))$ des opérations sur les $\mathcal{H}$-Matrices.

On note cependant que l'arbre géométriquement équilibré n'est pas employé dans notre implémentation, et que ces estimations ne sont donc pas valables pour les arbres utilisés ici. Cependant, la croissance de la hauteur de l'arbre médian est elle aussi logarithmique.

3.5 Implémentation et premiers résultats

Nous présentons ici quelques résultats issus de l'implémentation séquentielle des algorithmes précédents. Cette implémentation ne s'appuie pas sur du code existant, et est un des résultats concrets de cette thèse. Elle ne présente néanmoins pas de nouveautés par rapport à l'État de l'Art, il existe de nombreuses bibliothèques implémentant les mêmes opérations [1, 3, 4], avec lesquelles elle ne partage pas de code.

La bibliothèque est implémentée dans le langage C++, et repose sur BLAS/LAPACK pour les opérations d'algèbre linéaire dense de bas niveau. Elle a les caractéristiques suivantes :

— Indépendance de l'équation par l'utilisation de méthodes de compression « boîte noire » ;

— Calculs en simple et double précision, nombres réels et complexes (« SDCZ » dans les conventions de BLAS) ;
— Opérations disponibles : addition, produit matrice-vecteur, matrice-matrice, inversion et décompositions LU et LDL^T (ainsi que la résolution de systèmes linéaires) ;
— Algorithmes de compression : SVD, ACA avec pivotage total et partiel, ACA+.

Cette bibliothèque peut être utilisée au sein de la suite logicielle Aseris, ou indépendamment.

Cette section présente des résultats issus de l'intégration de cette bibliothèque dans la suite Aseris. Ce sont donc des calculs d'électromagnétisme 3D dans le domaine fréquentiel, par la méthode des éléments finis de frontière. Dans un premier temps, une série de tests est réalisée sur une famille de cylindres section 3.5.1. Un sous-ensemble de ces tests est reproduit sur une autre géométrie, le cône-sphère, fréquemment employé pour estimer la précision des codes de diffraction électromagnétique, section 3.5.2.1. Une comparaison à une solution analytique sur une sphère est effectuée section 3.5.2.1.

3.5.1 Cylindres

L'objectif de cette section est de caractériser la précision et performance du solveur $\mathcal{H}$-Matrice, et de déterminer les valeurs pertinentes pour les paramètres. On rappelle ces paramètres et choix :

— Arbre de blocs : valeur du paramètre η dans le critère d'admissibilité (3.1.2) (section 3.1.4.2.1) ;
— Assemblage :
  — Choix de l'algorithme : SVD, ACA avec pivotage total ou partiel, ACA+ (section 3.2)
  — Choix de la valeur de ε dans le critère d'arrêt (section 3.2.3) ;
— Factorisations : Inversion (section 3.3.3.2), décomposition LU (section 3.3.3.3) ou LDL^T (section 3.3.3.4).

Du point de vue de la précision et de la performance du solveur, les points suivants sont à étudier :

— Qualité de l'approximation ;
— Taux de compression (définition 3.20) ;
— Croissance du temps de calcul ;
— Sensibilité au raffinement du maillage et à la forme de l'objet.

Cette étude pratique est d'autant plus importante que les théorème garantissant la convergence exponentielle de l'approximation ne s'appliquent pas à l'équation des ondes.

3.5.1.1 Présentation des cas tests

Dans la suite de cette section, sauf précision contraire nous considérons des cylindres d'axe (Oz), ouverts aux deux extrémités, et dont le maillage est généré de manière procédurale ; il est constitué de triangles équilatéraux. Les valeurs données ici pour les divers paramètres sont les valeurs par défaut. Sauf mention contraire, elles sont utilisées dans la suite.

Figure 3.15 – Cylindre de référence

Matériau	Parfaitement conducteur
Diamètre	40cm
Longueur	1m
Fréquence	10GHz
Maillage	Taille des arêtes $\lambda/10$
Degrés de liberté	482 269
Illumination	— Onde plane $\frac{\boldsymbol{\kappa}}{\lVert\boldsymbol{\kappa}\rVert} = (0, 0, 1)$, — Amplitude 1 — Polarisation horizontale

Table 3.1 – Configuration physique par défaut.

Arithmétique	Simple précision
Arbre de groupes	Médian
Tolérance	$\varepsilon = 10^{-4}$
Critère d'admissibilité	$\eta = 3$
Compression	ACA+ et recompression
Décomposition	LDL^T

Table 3.2 – Configuration de calcul par défaut.

3.5.1.1.1 Configuration physique Les caractéristiques par défaut des calculs sont résumées par la table 3.1, et le cylindre correspondant est représenté par la figure 3.15.

3.5.1.1.2 Configuration de calcul La configuration de calcul par défaut est résumée par la table 3.2. On note que l'utilisation de la décomposition LDL^T est possible, car on choisit la formulation EFIE pour ces calculs (et la matrice a donc l'expression de l'équation (2.2.17)). Par ailleurs, on rappelle que même dans le cas des calculs en simple précision, l'assemblage (calcul des termes A_{ij} de la matrice d'interaction et algorithme ACA+) est fait en double précision.

Processeur	2× Intel Xeon « Sandy Bridge » EP E5-2680 à 2.7GHz (avec AVX)
Mémoire	2 × 32Go, DDR3 ECC
Logiciels	Linux 2.6.32, Intel Composer XE 2013, Intel MKL 11.0, BullxMPI
Réseau	Infiniband QDR, « Full Fat Tree »

TABLE 3.3 – Principales caractéristiques de la machine Curie-16.

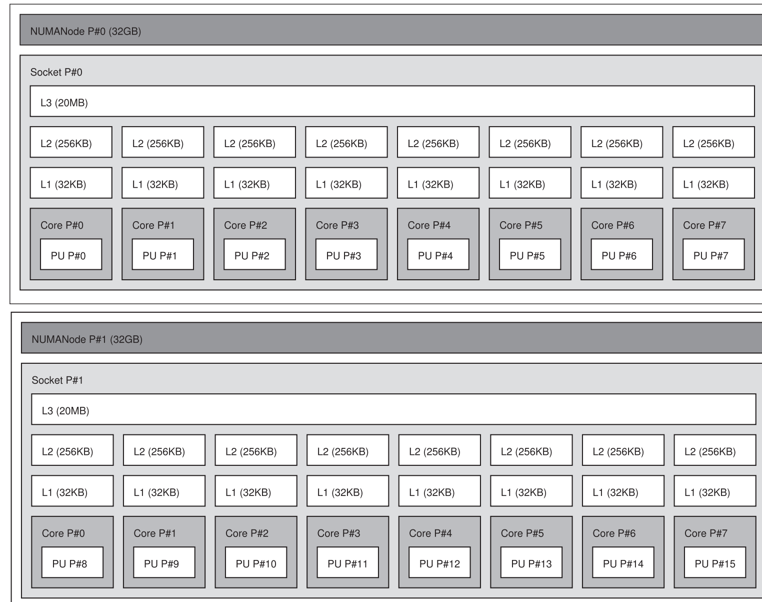


FIGURE 3.16 – Topologie simplifiée d'un nœud « fin » de Curie, Curie-16.

3.5.1.1.3 Configuration informatique Les calculs sont effectués sur un nœud « fin » du calculateur Curie du TGCC (Très Grand Centre de Calcul) du CEA [2]. Cette machine sera nommée dans la suite de ce document Curie-16. Les caractéristiques de cette machine sont résumées par la table 3.3, et sa topologie simplifiée donnée par la figure 3.16.

On note que la technologie *Turbo Boost* d'Intel supportée par ce processeur est désactivée, ce qui permet de ne pas fausser les tests de passage à l'échelle. Cette technologie autorise le processeur à augmenter sa fréquence d'horloge automatiquement lorsque tous les cœurs ne sont pas utilisés, et/ou lorsque les conditions de température et consommation électrique le permettent, ce qui ajoute une variabilité indésirable dans les mesures. Par ailleurs, dans ce chapitre uniquement, un seul processeur de cette machine sera utilisé dans la majorité des tests.

3.5.1.2 Assemblage

On s'intéresse dans cette section au comportement de l'assemblage dans les situations suivantes :

- Dépendance en temps et précision par rapport au nombre de degrés de liberté ;
- Dépendance au mode de compression : SVD, ACA avec pivotage total, partiel et ACA+ ;

— Dépendance à la discrétisation de l'objet.

Pour les calculs dans lesquels le nombre de degrés de liberté n'est pas constant, nous utilisons des objets homothétiques au cylindre de référence, avec la même fréquence et finesse de maillage.

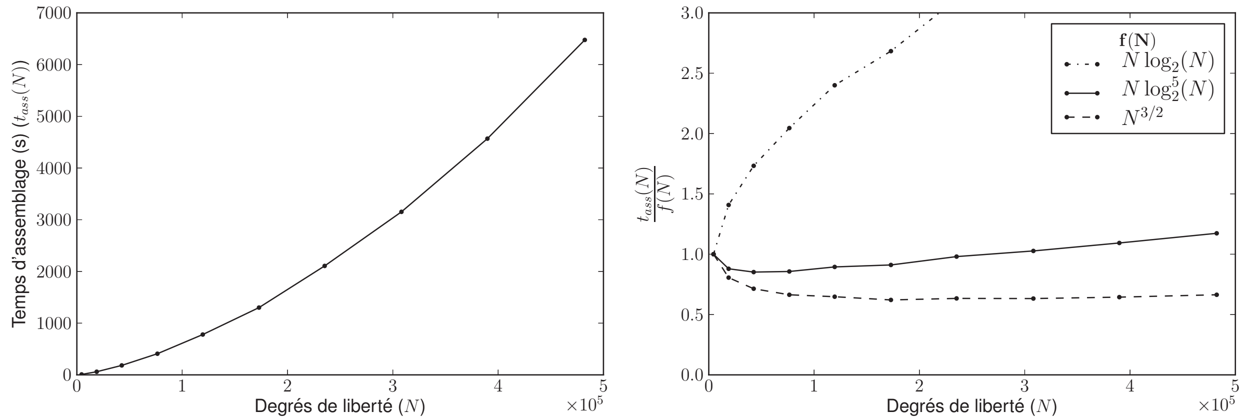


Figure 3.17 – Temps d'assemblage en fonction du nombre de degrés de liberté. Le graphique de droite compare la croissance du temps de calcul à diverses vitesses.

3.5.1.2.1 Temps d'assemblage en fonction du nombre d'inconnues Le temps d'assemblage en fonction du nombre de degrés de liberté est donné par la figure 3.17. Ce temps croît plus vite que $\mathcal{O}(N \log_2(N))$, comme le montre le graphique de droite de cette même figure.

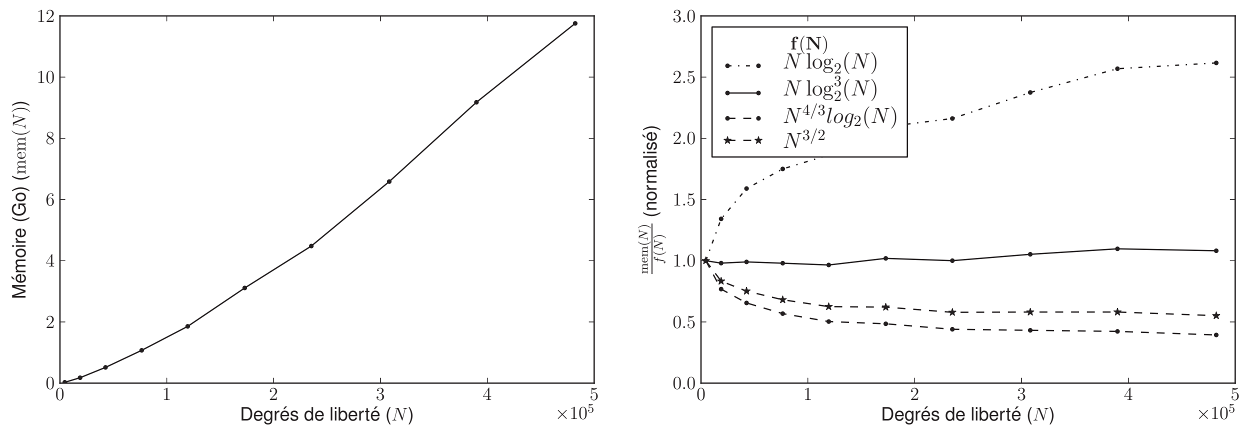


Figure 3.18 – Quantité de mémoire requise en fonction du nombre de degrés de liberté.

Cette croissance plus rapide peut être liée à plusieurs facteurs :

1. La croissance du rang k des blocs admissibles avec la taille du problème considéré ;
2. Une compression non optimale effectuée par l'algorithme ACA+.

En effet, l'algorithme ACA+ n'assure pas de trouver la compression optimale d'un bloc de la $\mathcal{H}$-Matrice. La recompression de ce bloc par SVD doit en principe partiellement éliminer

ce problème. Afin de distinguer les deux situations, la figure 3.18 représente la consommation mémoire de la matrice compressée, dont la croissance est $\mathcal{O}(kN\log_2(N))$ dans le cas où le rang k est fixe. En utilisant le taux de compression $\tau(M)$ de la définition 3.20, l'occupation mémoire d'une $\mathcal{H}$-Matrice de taille $N \times N$ est : $\frac{1}{2}8\tau(M)N^2$, puisqu'un nombre complexe en simple précision occupe 8 octets. La figure 3.18 montre l'évolution de la consommation mémoire, ainsi que sa croissance comparée à diverses asymptotiques. Le comportement observé corrobore les deux hypothèses précédentes.

La quantité de stockage est en $\mathcal{O}(kN\log_2(N))$, contre $\mathcal{O}(k^2N\log_2(N))$ pour k fixé. Les tests n'étant pas faits avec k fixé mais de manière adaptative, la croissance plus rapide du temps de calcul par rapport à la quantité de mémoire requise est donc attendue. On note également que la croissance de la quantité de mémoire nécessaire est clairement plus lente que $\mathcal{O}(N^{\frac{3}{2}})$, et plus lente que la croissance empirique $\mathcal{O}(N^{4/3}\log_2(N))$ trouvée dans [100]. La croissance la plus en rapport avec les expériences dans ce cas est proche de $\mathcal{O}(\log_2^2(N)N\log_2(N))$, avec donc un facteur $\log_2^2(N)$ au lieu de k fixe, comme ceci est visible sur la figure 3.18. Ce remplacement d'un rang constant par un rang effectif de croissance logarithmique est renforcé par la croissance du temps d'assemblage de la figure 3.17.

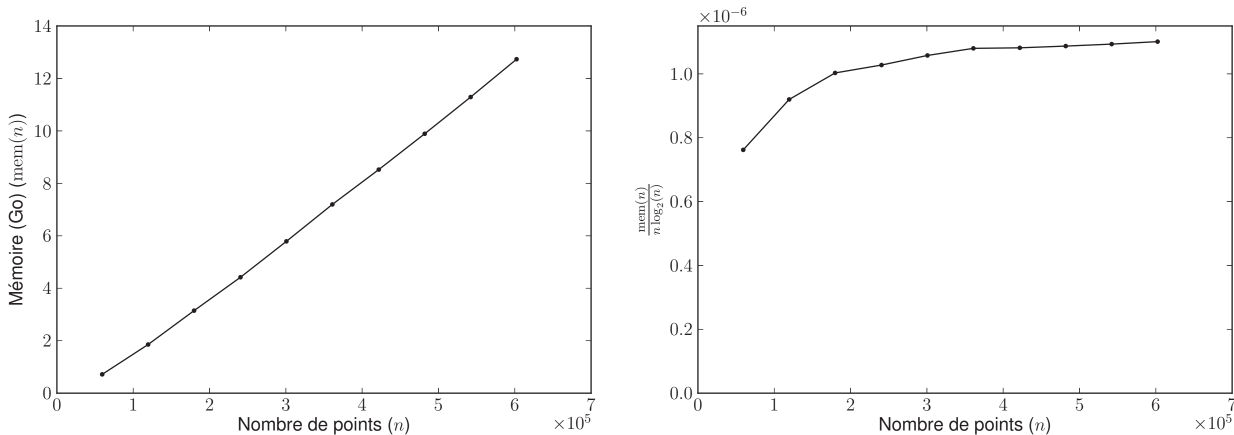


FIGURE 3.19 – Quantité de mémoire requise en fonction du nombre de degrés de liberté, cylindres "allongés".

Cylindres « Allongés » Il faut par ailleurs noter que cette croissance est fortement dépendante de la forme de l'objet. Afin d'illustrer ceci, nous considérons un ensemble de cylindres « allongés ». Ces cylindres ont un diamètre de 20cm, et une longueur de $25cm \times i, i = 1, \ldots, 10$. Les autres caractéristiques sont identiques à celles de la description de la section 3.5.1.1.1.

La croissance de l'espace mémoire requis pour la matrice d'interaction est donnée par la figure 3.19. Cette croissance est conforme à l'asymptotique $\mathcal{O}(N\log_2(N))$, comme le montre le graphique de droite sur cette figure.

3.5.1.2.2 Erreur de compression La compression par l'algorithme ACA+ ne permet pas de donner de garanties sur la précision du résultat. Cependant, il est possible

de calculer l'écart entre la matrice compressée et la matrice utilisée par un solveur direct usuel. Les feuilles non admissibles de la $\mathcal{H}$-Matrice étant identiques aux éléments correspondants d'une matrice non compressée, l'erreur sur ces éléments est nulle. Pour les feuilles compressées, on calcule l'erreur en norme de Frobenius bloc par bloc. En notant $\tilde{M}$ la $\mathcal{H}$-Matrice correspondant à la matrice d'interaction M, l'expression de l'erreur en norme de Frobenius est, avec $AB^T = \tilde{M}|_{\sigma\times\tau}$ dans la somme :

$$Err(M, \tilde{M}) := \sqrt{\frac{\sum_{\sigma\times\tau \text{ admissible}} \|M|_{\sigma\times\tau} - AB^T\|_F^2}{\|M\|_F^2}}$$

On rappelle que la norme de Frobenius $\|\cdot\|_F$ est définie pour une matrice $M \in \mathbb{C}^{m\times n}$ par :

$$\|M\|_F := \sqrt{\sum_{i,j} |M_{ij}|^2}$$

ce qui permet de décomposer le calcul de l'erreur comme précédemment. Ce calcul peut donc être effectué avec une faible quantité de mémoire, cependant il reste coûteux pour une grande taille de matrice du fait de la nécessité de calculer tous les termes de la matrice M, en plus de $\tilde{M}$.

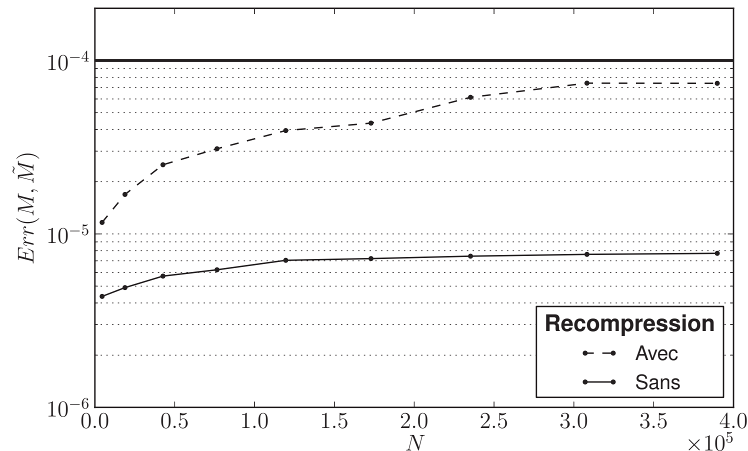


Figure 3.20 – Erreur de compression pour ACA+ en fonction du nombre de degrés de liberté, avec et sans recompression.

La figure 3.20 illustre l'erreur de compression pour la compression par l'algorithme ACA+ avec $\varepsilon = 10^{-4}$ et une recompression de la matrice à la même précision. La précision est bien dans tous les cas meilleure que 10^{-4}, et ne présente pas de croissance forte avec le nombre de degrés de liberté. Il faut souligner que l'erreur de compression réelle n'est pas nécessairement reliée à ε, mais le fait que l'erreur effective soit inférieure à ε permet d'utiliser cette valeur comme indicateur de la précision souhaitée pour l'approximation. On note par ailleurs que l'erreur en norme de Frobenius sans recompression est bien inférieure à ε dans tous les cas, et que la recompression fait remonter celle-ci, tout en restant inférieure au seuil fixé. Cette remontée de l'erreur est liée à la tolérance de recompression, fixée ici à 10^{-4}.

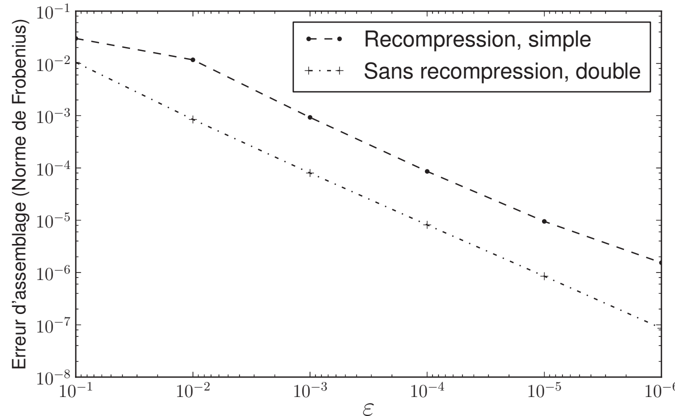


FIGURE 3.21 – Erreur de compression pour ACA+ en fonction de ε, avec et sans recompression, en simple et double précision.

Par ailleurs, la figure 3.21 représente l'évolution de l'erreur de compression avec la variation de ε pour le plus grand cylindre (482.269 degrés de liberté). Cette figure montre une décroissance attendue de l'erreur avec la variation de ε, et donc la possibilité d'atteindre une grande précision avec la diminution de la tolérance lors de la compression. Nous utiliserons cette observation dans la suite pour simplifier et accélerer l'estimation de l'erreur après résolution des systèmes linéaires. Dans le cas où la recompression des blocs est utilisée en simple précision, l'erreur en norme de Frobenius suit de très près la tolérance ε fixée, à l'exception d'un léger décollement pour $\varepsilon = 10^{-6}$, qui peut être attribué à la limite de précision des nombres flottants en simple précision. Néanmoins, pour une compression sans recompression en double précision, la précision obtenue est bien meilleure que ε pour toutes les valeurs testées.

3.5.1.2.3 Méthodes de compression Dans ce paragraphe, nous comparons la précision et le temps de calcul nécessaires pour l'assemblage d'une $\mathcal{H}$-Matrice à l'aide de la SVD et des algorithmes ACA avec pivotages total et partiel, et ACA+. On rappelle les complexités asymptotiques pour un rang d'approximation k fixé, et pour un bloc $M|_{\sigma\times\tau}$ de taille $m \times n$:

Méthode	**Complexité**
SVD	$\mathcal{O}(k(m^2n + n^2m))$
ACA, pivotage total	$\mathcal{O}(kmn)$
ACA, pivotage partiel	$\mathcal{O}((k + k^2)(m + n))$
ACA+	$\mathcal{O}((k + k^2)(m + n))$

Il est clair que le temps d'assemblage est plus important pour les méthodes de complexité non linéaire par rapport à la taille du bloc, ce qui est confirmé par la figure 3.22. Les taux de compression après assemblage ne sont pas représentés, car ils sont essentiellement identiques entre les diverses méthodes de compression. On remarque que l'heuristique ACA+ n'est pas plus lente que la méthode ACA avec pivotage partiel, ce qui est attendu

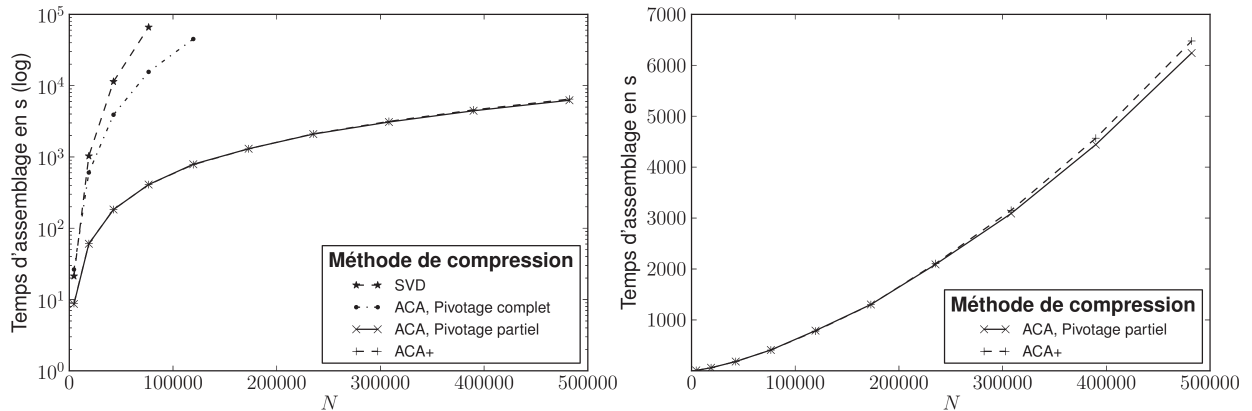


Figure 3.22 – Temps d'assemblage pour les diverses méthodes. Le temps est en échelle logarithmique sur la figure de gauche, linéaire à droite.

du point de vue de la complexité asymptotique, mais non évident. En effet, une ligne et une colonne de plus sont calculées pour chaque bloc dans cette méthode par rapport à la méthode ACA avec pivotage partiel, pour un rang égal. Sur cette figure, les données pour la SVD et ACA avec pivotage total sont tronquées pour deux raisons :

1. Temps de calcul ;
2. Quantité de mémoire disponible.

La première raison est claire, la seconde résulte de la nécessité de construire le bloc $M|_{\sigma\times\tau}$ en mémoire avant de faire la compression pour ces deux méthodes. La taille maximale des blocs n'étant pas bornée, elle augmente avec la taille du problème, ce qui pose des problèmes, même sur une machine comportant 64Go de mémoire. Sur une telle machine, la taille de bloc maximale est d'environ 4.4×10^4, puisque l'assemblage est toujours réalisé en double précision, en négligeant la consommation mémoire des autres tableaux nécessaires pour la compression.

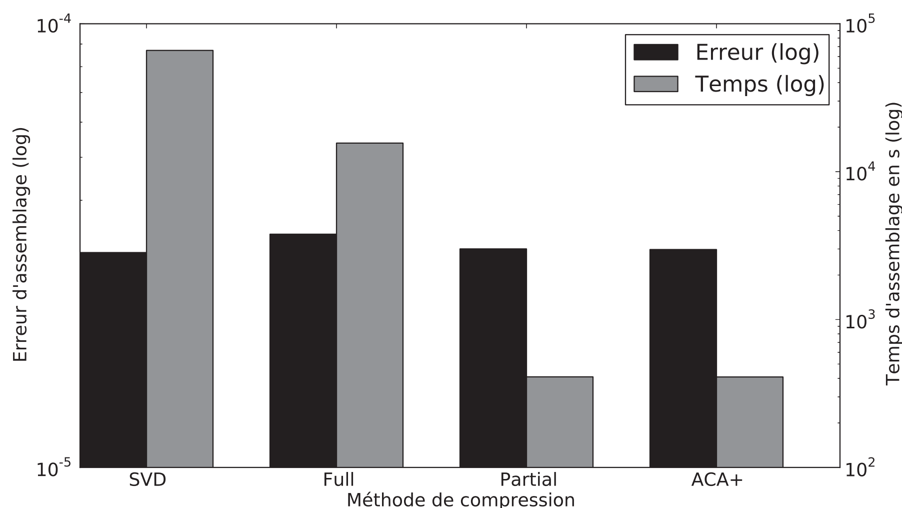


Figure 3.23 – Variations de l'erreur et du temps d'assemblage par méthode, pour un cylindre à 76486 inconnues.

La variation de l'erreur pour les trois méthodes est donnée par la figure 3.23. On constate sur ce cas que les erreurs données (pour une tolérance $\varepsilon = 10^{-4}$) par les diverses méthodes de compression sont très proches. En revanche, une différence radicale de temps de calcul est constatée, ce qui incite à privilégier la compression ACA+. Elle offre en effet le meilleur temps de calcul, associé à un taux de compression et une erreur très proche des autres méthodes plus coûteuses, y compris la compression SVD.

3.5.1.2.4 Raffinement de maillage Un cadre d'application important d'un point de vue pratique est la réutilisation du même maillage pour différents calculs à des fréquences variées. Dans de nombreuses situations (par exemple pour le calcul avec un solveur fréquentiel de réponses à un signal temporel), il est nécessaire d'effectuer des calculs sur le même objet pour un ensemble de fréquences. L'amplitude des rampes de fréquence peut être grande, par exemple entre 20Hz et 20.000Hz pour un calcul acoustique (cet intervalle correspondant à l'ensemble des fréquences audibles pour un humain). Il est souvent difficile de produire un maillage adapté à chaque fréquence, et ce d'autant plus que le critère de maillage devient partiellement subjectif pour les basses fréquences, lorsque la règle des dix points par longueur d'onde n'est plus suffisante. On s'intéresse alors à la capacité du solveur $\mathcal{H}$-Matrice à produire des résultats satisfaisants sur un maillage raffiné.

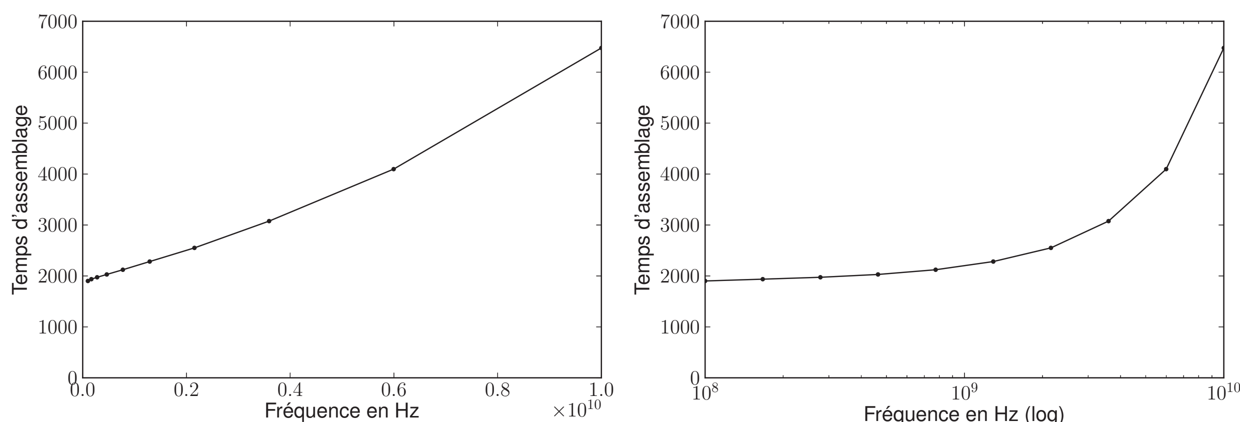


FIGURE 3.24 – Temps d'assemblage pour un maillage raffiné. La figure de droite est en échelle logarithmique sur l'axe des fréquences. Le maillage le plus raffiné est en $\lambda/1000$, le moins raffiné en $\lambda/10$.

On utilise pour ceci le maillage du cylindre de référence, pour des fréquences comprises entre 100MHz ($\lambda/1000$) et 10GHz ($\lambda/10$). Pour les fréquences, un échantillonnage logarithmique est utilisé. Les résultats en termes de taux de compression et temps d'assemblage sont donnés par les figures 3.24 et 3.25. On constate une diminution du temps de calcul avec le raffinement du maillage, à nombre d'inconnues constant, et une évolution quasi-linéaire de l'espace mémoire requis avec la fréquence, lorsque le cas de calcul est sur-maillé, ce qui n'est pas le cas du temps d'assemblage dont la croissance est plus rapide.

3.5.1.2.5 Influence de η La figure 3.26 représente l'erreur de compression pour η allant de 1 à 4, avec $\varepsilon = 10^{-4}$, et sans recompression. On constate une légère augmentation de l'erreur (de 6.4×10^{-6} à 8.16×10^{-6}) avec l'augmentation de η, ce qui est attendu,

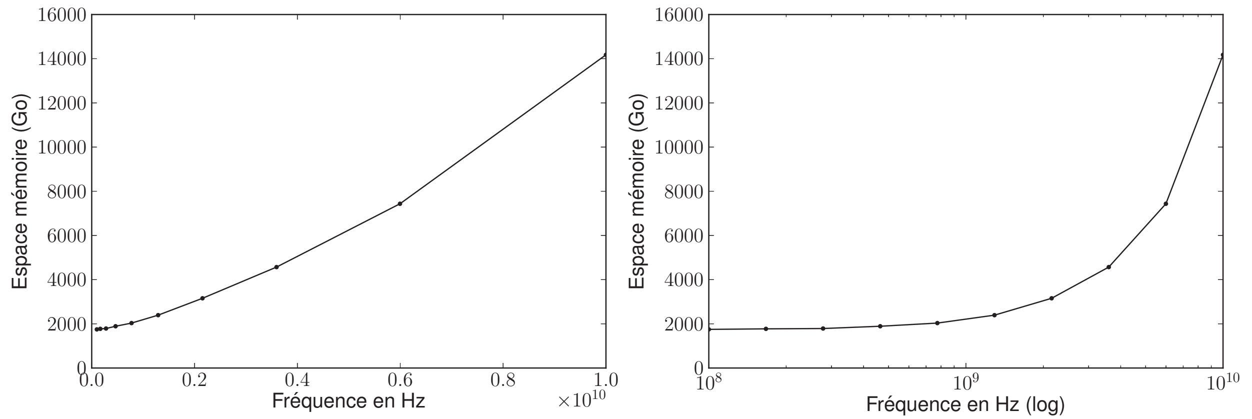


Figure 3.25 – Espace mémoire pour un maillage raffiné. La figure de droite est en échelle logarithmique sur l'axe des fréquences.

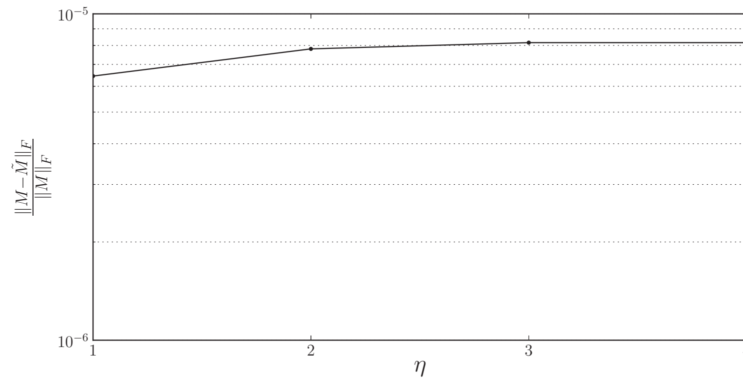


Figure 3.26 – Erreur de compression en fonction de η.

puisqu'une grande valeur de η mène à l'admissibilité de groupes de degrés de liberté proches et de plus grande dimension, ce qui tend à augmenter l'erreur de compression. Néanmoins, celle-ci reste contrôlée dans tous les cas. Par ailleurs, la figure 3.27 montre l'évolution du taux de compression (après recompression) pour les mêmes valeurs de η. L'augmentation de η mène à une diminution du taux de compression, avec néanmoins un palier atteint à partir de $\eta = 2$.

3.5.1.2.6 Conclusions Les tests précédents permettent de dégager plusieurs observations :

1. Il est possible de construire efficacement une représentation $\mathcal{H}$-Matrice d'une matrice d'interaction, même pour des objets de grande dimension ;
2. La croissance de la consommation mémoire est moins bonne que les estimations théoriques pour les noyaux asymptotiquement lisses. Celle-ci est néanmoins très dépendante de la géométrie de l'objet. En particulier, dans le cas d'un objet présentant une direction privilégiée (cylindres allongés), les résultats obtenus sont proches des résultats théoriques pour un noyau asymptotiquement lisse.

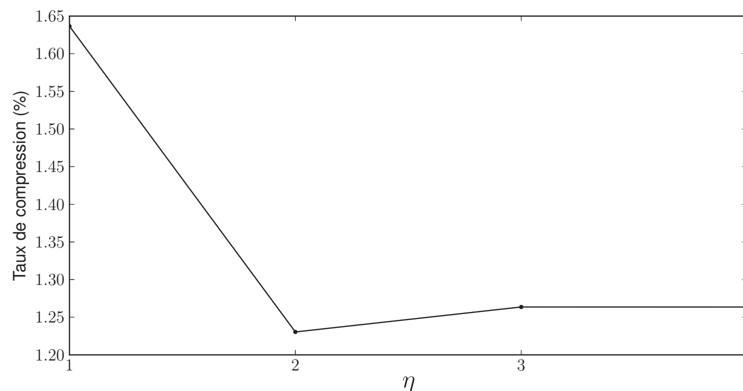


FIGURE 3.27 – Taux de compression en fonction de η.

3. La précision de la construction est maîtrisée, et la valeur ε du critère d'arrêt donne dans les cas présentés une sur-estimation de l'erreur en norme de Frobenius ;
4. La méthode ACA donne des résultats pleinement satisfaisants, particulièrement dans sa variante ACA+, dont le temps d'exécution est compétitif avec la méthode ACA avec pivotage partiel.
5. La diminution de la fréquence d'étude à maillage fixé permet de réduire le temps de calcul et la consommation mémoire, à précision maîtrisée.
6. L'augmentation de la valeur de η permet d'améliorer le taux de compression, au prix d'une très légère remontée de l'erreur. Le taux de compression atteignant un palier à partir de $\eta = 2$, il est raisonnable de fixer $\eta = 2$ ou 3 dans les cas présentés ici.

Le second point représente potentiellement un obstacle de taille à l'application des $\mathcal{H}$-Matrices pour le noyau d'Helmholtz. En particulier, ceci semble signifier que les $\mathcal{H}$-Matrices ne sont pas compétitives par rapport à la FMM pour les problèmes de grande dimension, du fait de la différence de comportement asymptotique. Il faut néanmoins nuancer cette conclusion, du fait de la dépendance du taux de compression à la géométrie de l'objet, et de la difficulté de discerner les différentes complexités par les résultats expérimentaux. En effet, une complexité asymptotique en $\mathcal{O}(N \log_2^3(N))$ n'est en pratique pas très différente de $\mathcal{O}(N \log_2(N))$ pour N de taille « raisonnable », et la méthode la plus avantageuse dépend essentiellement de la constante multiplicative associée, comme noté dans [35]. Le chapitre 6 présentera des comparaisons pratiques entre des calculs par la méthode FMM et $\mathcal{H}$-Matrice pour des problèmes d'électromagnétisme et d'acoustique.

3.5.1.3 Factorisation et résolution

L'objectif de ce travail étant de réaliser un solveur direct, il est important de s'intéresser au comportement de la factorisation des $\mathcal{H}$-Matrices. En particulier, le problème de l'évolution de l'erreur est d'importance, puisque la factorisation d'une $\mathcal{H}$-Matrice se fait sans pivotage, et que les matrices BEM sont en règle générale mal conditionnées.

On s'intéresse dans cette section au comportement de la factorisation d'une $\mathcal{H}$-Matrice en fonction des paramètres du problème et des algorithmes opérant sur les $\mathcal{H}$-Matrices.

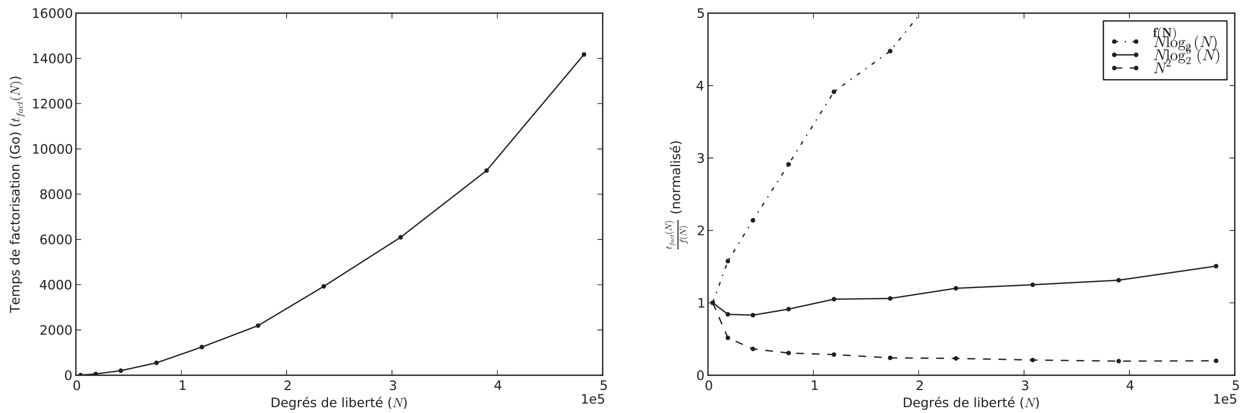


Figure 3.28 – Temps de décomposition en fonction du nombre d'inconnues.

3.5.1.3.1 Temps de décomposition en fonction du nombre d'inconnues La figure 3.28 illustre la croissance du temps de factorisation avec la taille de la matrice. On observe le même phénomène que pour l'assemblage : la croissance du temps de calcul est plus rapide que les estimations à k fixe, mais plus lente que $\mathcal{O}(N^2)$. La croissance plus rapide que le temps d'assemblage s'explique par les démonstrations de complexité à k fixe donnant des estimations supérieures à la factorisation par rapport à l'assemblage, et par la détérioration de la compression pendant la factorisation.

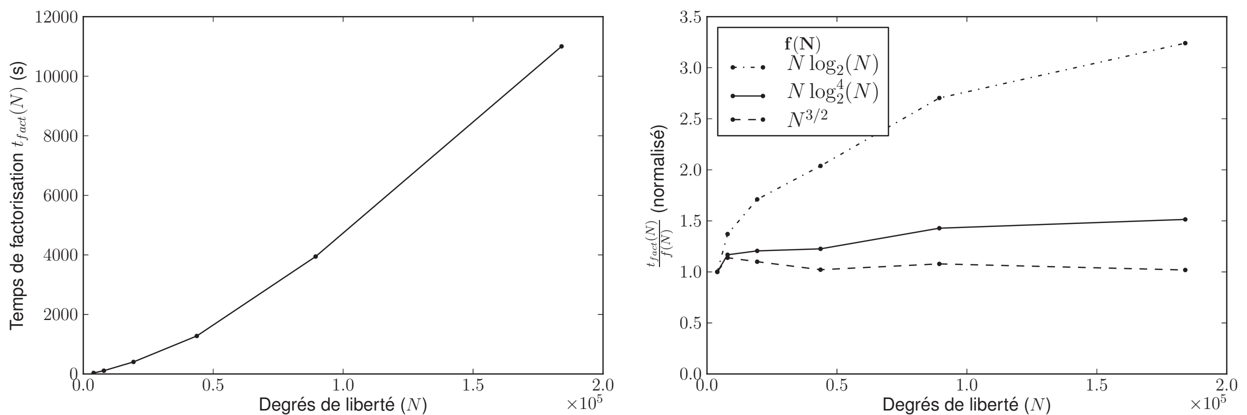


Figure 3.29 – Temps de décomposition en fonction du nombre d'inconnues pour un noyau asymptotiquement lisse. Les données sont issues de [70, section 6.7.1].

La croissance plus rapide que les estimations lors d'une factorisation avec un noyau asymptotiquement lisse à tolérance ε fixée n'est à notre connaissance pas notée dans la littérature, mais est présente dans les données. Par exemple, la figure 3.29 représente la croissance du temps de factorisation LU d'une $\mathcal{H}$-Matrice pour un noyau asymptotiquement lisse. Les données sont issues de [70, section 6.7.1]. Elles montrent le temps de factorisation séquentielle d'une $\mathcal{H}$-Matrice issue de la discrétisation par collocation d'un problème électrostatique, dont le noyau de Green est $\frac{1}{4\pi r}$, qui est asymptotiquement lisse. Les temps de calcul ne sont bien entendu pas comparables, puisque ni l'arithmétique (réelle contre complexe), ni l'implémentation, ni l'équation, ni la géométrie ne sont identiques.

Cette figure illustre néanmoins une croissance plus rapide lorsque le rang n'est pas fixé, ce qui ne se retrouve pas dans [70] pour un rang fixe.

Ceci n'est pas nécessairement incohérent avec la théorie, puisque les estimations de complexité (par exemple, celles de la section 3.4.1) supposent un rang fixe. Il est par ailleurs intéressant de noter la difficulté de discriminer les différentes asymptotiques sur la figure 3.29. Il n'est cependant pas possible de donner des explications plus précises basées uniquement sur les données de [70].

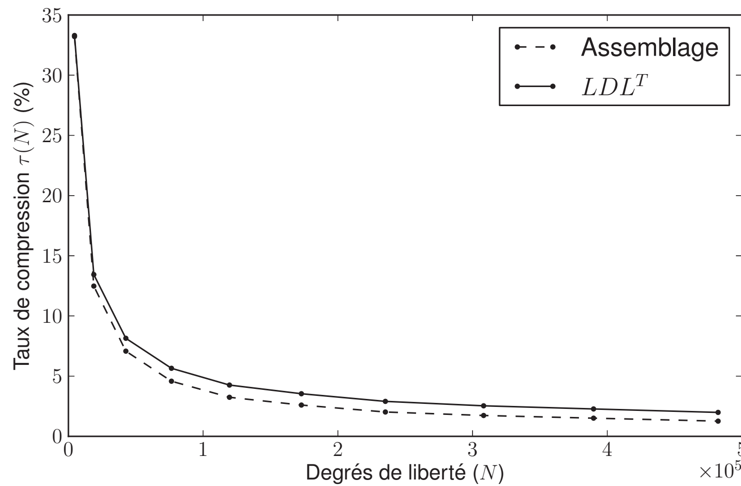


Figure 3.30 – Comparaison du taux de compression avant et après décomposition.

En revanche, pour le code et les exemples présentés ici, il est possible de formuler des hypothèses. Une des explications de cette croissance plus rapide réside dans la perte de compression de la $\mathcal{H}$-Matrice au cours de la factorisation. La figure 3.30 compare le taux de compression avant et après décomposition, en fonction de la taille du problème. On constate pour ces cas une perte de compression lors de la factorisation de la $\mathcal{H}$-Matrice. Nous pouvons conjecturer plusieurs causes possibles à ce phénomène :

— L'interaction entre des blocs de rang faible et de rang élevé a des effets sur le rang des matrices, à la tolérance fixée. Cet effet serait analogue au phénomène de remplissage des matrices creuses dans les solveurs directs.
— L'accumulation des erreurs numériques et des erreurs de recompression fait augmenter le rang des blocs compressés, à une tolérance fixée.

Il n'est malheureusement pas possible de confirmer ou infirmer ces hypothèses avec les données disponibles ici.

Comparaison à un solveur direct La figure 3.31 compare le temps de factorisation entre le solveur $\mathcal{H}$-Matrice et un solveur direct. Le nombre d'opérations pour une factorisation LDL^T d'une matrice de taille $N \times N$ complexe est $\frac{4}{3}N^3$ [25]. Le nombre d'opérations par cycle en simple précision pour un processeur « Sandy Bridge » est de 16 (instructions AVX). On suppose une efficacité de 80% (comparable à celle atteinte par MKL pour une décomposition LDL^T), et l'utilisation pleine des instructions AVX, et on néglige le surcoût d'un solveur *Out-Of-Core*, ce qui rend l'estimation optimiste. Le solveur $\mathcal{H}$-Matrice est bien entendu plus rapide, l'avantage étant fortement croissant avec

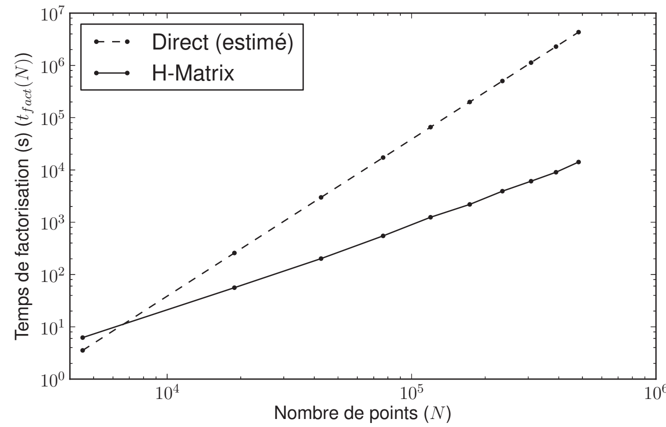


FIGURE 3.31 – Comparaison du temps de factorisation entre un solveur $\mathcal{H}$-Matrice et un solveur direct (factorisation LDL^T).

le nombre de degrés de liberté. Il faut également noter que cette figure ne prend pas en compte le temps d'assemblage, dont la croissance est également plus lente avec le solveur $\mathcal{H}$-Matrice. Le croisement entre les deux solveur se fait pour un faible nombre de degrés de liberté, de l'ordre de 10^3. On peut donc estimer qu'il est toujours intéressant d'utiliser le solveur $\mathcal{H}$-Matrice par rapport à un solveur direct.

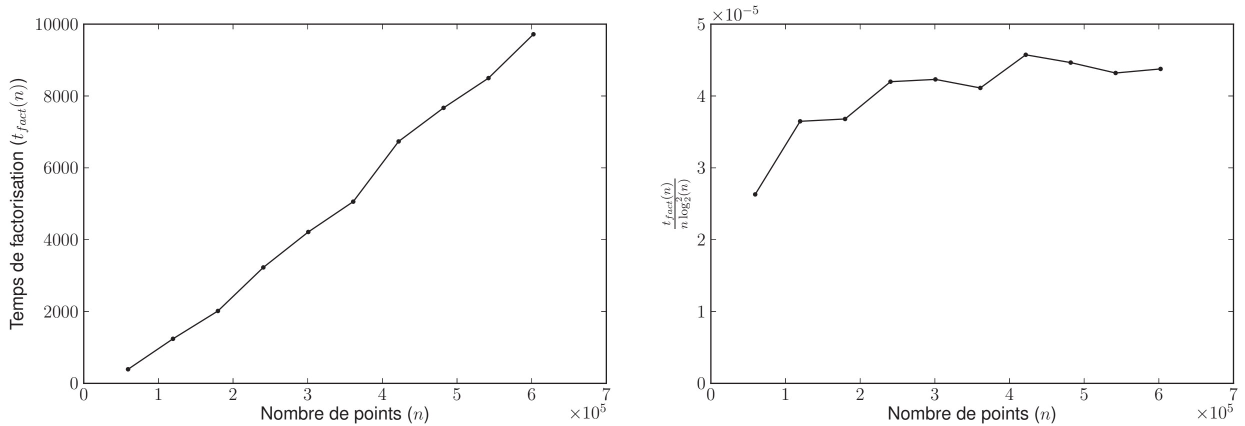


FIGURE 3.32 – Temps de décomposition en fonction du nombre d'inconnues pour les cylindres « allongés ».

Cylindres « allongés » La dépendance du temps de calcul à la forme de l'objet est également constatée pour la factorisation. La figure 3.32 montre la dépendance du temps de factorisation au nombre de degrés de liberté pour les cylindres allongés de la section précédente, et on retrouve alors un comportement asymptotique comparable à celui de la figure 3.29 pour un noyau asymptotiquement lisse.

3.5.1.3.2 Erreur de résolution On note $(LDL^T)^{-1}b$ la solution du système linéaire $LDL^Tx = b$, bien que celle-ci ne soit pas calculée par inversion de la matrice LDL^T, mais par résolution de deux systèmes linéaires triangulaires, et d'un système diagonal. Par ailleurs, on note $\tilde{M}$ l'approximation $\mathcal{H}$-Matrice de la matrice d'interaction M, $\tilde{L}\tilde{D}\tilde{L}^T$ sa décomposition LDL^T, et $\tilde{x}$ la solution obtenue par décomposition LDL^T de $\tilde{M}$.

Pour des raisons de coût de stockage et de temps de calcul, l'erreur de résolution est définie dans cette section par :

$$Err(M, \tilde{M}, b) := \frac{\left\|b - \tilde{M}(\tilde{L}\tilde{D}\tilde{L}^T)^{-1}b\right\|_2}{\|b\|_2} = \frac{\left\|b - \tilde{M}\tilde{x}\right\|_2}{\|b\|_2}$$

On note que cette erreur est de même nature que celle utilisée dans le critère d'arrêt d'un solveur itératif, et que son estimation est identique dans le cas des méthodes FMM, puisque le produit Ax est trop coûteux pour être formé. Par ailleurs, la section 3.5.1.2.2 a montré que l'erreur $\|M - \tilde{M}\|$ est contrôlée, ce qui donne une base à $Err(M, \tilde{M}, b)$.

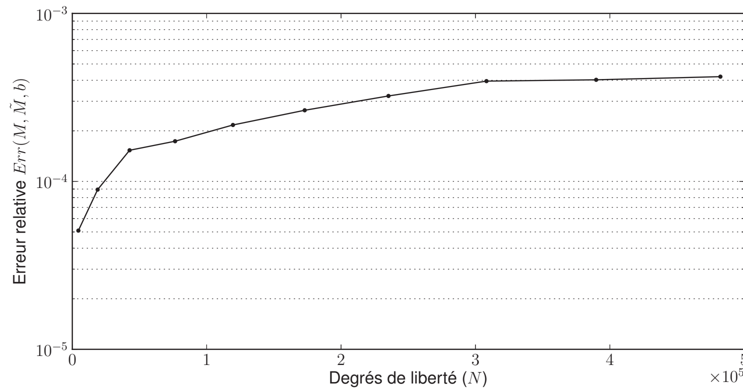


FIGURE 3.33 – Erreur de résolution en fonction du nombre de degrés de liberté.

L'erreur après résolution des systèmes linéaires est représentée par la figure 3.33. On note une croissance relativement lente de l'erreur avec le nombre de degrés de liberté. Une telle croissance de l'erreur est attendue, et constatée dans la littérature, par exemple dans [34]. Elle est par ailleurs à mettre en relation avec la croissance usuelle de l'erreur dans une factorisation dense, même avec pivotage.

3.5.1.3.3 Comparaison entre LU et LDL^T Dans le cas où la matrice est symétrique (lors d'un calcul EFIE par exemple), il est possible d'employer une décomposition LDL^T, alors que la décomposition LU est également utilisable lorsque la matrice ne l'est pas. En algèbre linéaire dense, la décomposition LDL^T nécessite deux fois moins d'opérations que la décomposition LU, avec un nombre d'opérations de $\frac{4}{3}N^3$ pour une matrice de taille N en nombres complexes, contre $\frac{8}{3}N^3$ pour la décomposition LU [25, Annexe C]. La comparaison du temps de calcul des deux méthodes de décomposition est donnée par la figure 3.34. On constate que le rapport de temps de calcul est presque de deux en pratique, tout comme pour l'algèbre linéaire dense. Le taux de compression est quasiment identique pour les deux méthodes à quelques centièmes de pourcent près, de même que l'erreur de résolution. La décomposition LDL^T est donc à préférer lorsque cela est possible,

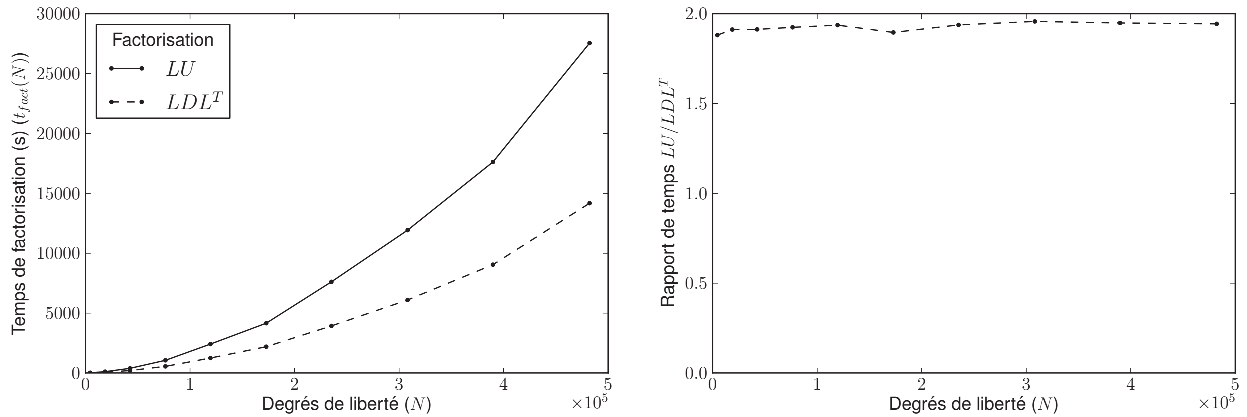


Figure 3.34 – Comparaison du temps de factorisation pour LU et LDL^T. La figure de droite représente le rapport de temps entre les deux méthodes.

puisqu'elle nécessite une quantité de mémoire divisée par deux, avec un temps de calcul diminué d'autant, à précision égale.

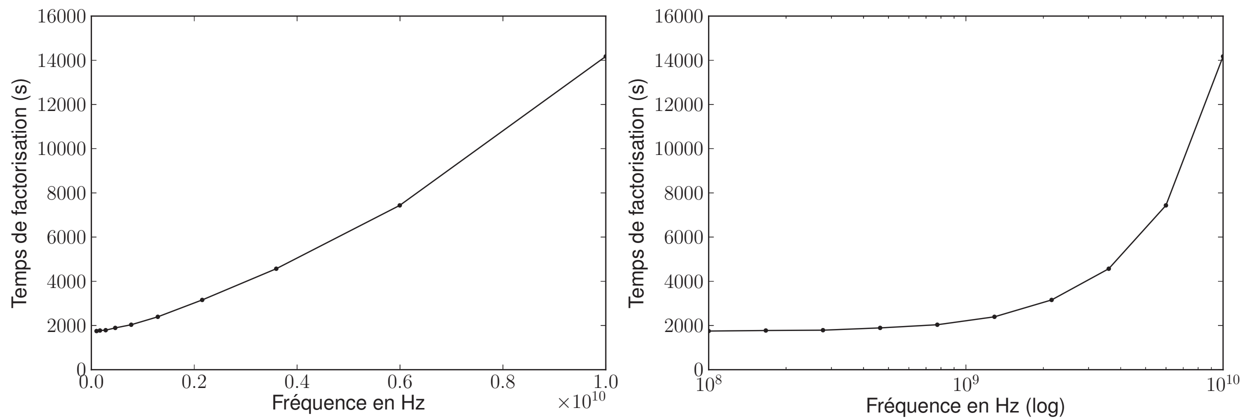


Figure 3.35 – Temps de factorisation pour un maillage raffiné. La figure de droite est en échelle logarithmique sur l'axe des fréquences.

3.5.1.3.4 Raffinement de maillage La figure 3.35 illustre l'effet de la diminution de la fréquence pour un maillage fixé. L'évolution est similaire à celle constatée pour l'assemblage, figure 3.24, et suggère que l'utilisation d'un seul maillage pour des calculs à des fréquences variées est possible avec un solveur $\mathcal{H}$-Matrice. La figure 3.36 montre de plus que l'erreur reste maîtrisée sur toute la plage de fréquences. En partant de 10GHz, cette erreur diminue, puis augmente pour atteindre un niveau sensiblement équivalent au maillage à 10GHz pour une fréquence de 100MHz ($\lambda/1000$). Il faut rappeler que le conditionnement des matrices BEM augmente avec la finesse du maillage, ce qui peut expliquer l'augmentation de l'erreur à partir d'un certain degré de raffinement.

Cette instabilité peut se comprendre intuitivement de la manière suivante. On rappelle

l'expression d'un élément de la matrice A (équation (2.2.17) page 45) :

$$\begin{aligned} A_{ij} &= \int_{\Gamma\times\Gamma} G(|x-y|)\left(\varphi_i(y)\cdot\varphi_j(x) - \frac{1}{\kappa^2}\nabla_\Gamma\cdot\varphi_i(x)\nabla_\Gamma\cdot\varphi_j(y)\right)dxdy \\ &= \int_{\Gamma\times\Gamma} G(|x-y|)\varphi_i(y)\cdot\varphi_j(x)dxdy + \frac{1}{(i\kappa)^2}\int_{\Gamma\times\Gamma} G(|x-y|)\nabla_\Gamma\cdot\varphi_i(x)\nabla_\Gamma\cdot\varphi_j(y)dxdy \end{aligned}$$

Par ailleurs, l'application de la divergence aux fonctions tests dans le second terme a pour effet de faire apparaître un facteur $\frac{1}{h}$, avec h la taille de l'arête du triangle. L'équation précédente peut alors s'écrire sous la forme de la somme de deux termes de même ordre de grandeur, I_1 et I_2, de la façon suivante :

$$A_{ij} = I_1 - \frac{1}{\kappa^2}\frac{1}{h^2}I_2$$

et on a $\kappa = \frac{2\pi}{\lambda}$ et $h = \frac{\lambda}{p}$, avec p le raffinement fréquentiel du maillage, variant ici de 10 à 1000. On a donc :

$$A_{ij} = I_1 - \frac{1}{(\kappa h)^2}I_2 = I_1 - \frac{p^2}{4\pi^2}I_2$$

Le second terme de la somme grandit alors avec p^2, et l'information contenue dans le premier terme « disparaît » avec le raffinement du maillage, ce qui pose des problèmes de précision. Ce problème est connu, et il est possible de le contourner avec une formulation adaptée, que nous n'employons pas ici. Il est également possible que l'impact des calculs en simple précision ou de la troncature liée à la compression renforce ce problème.

Remarque 3.38 (Acoustique)**.** *L'analyse précédente des ordres de grandeur en fonction du raffinement du maillage est spécifique aux équations intégrales pour l'électromagnétisme. Une telle étude serait à refaire pour une autre physique, par exemple pour l'acoustique.*

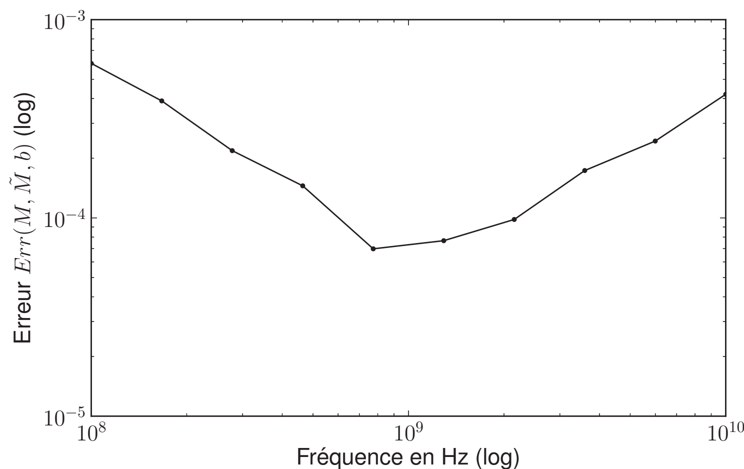


Figure 3.36 – Erreur en fonction de la fréquence pour un maillage raffiné.

Enfin, la variation du taux de compression entre l'assemblage et la $\mathcal{H}$-Matrice factorisée est représentée sur la figure 3.37. En fonction de la fréquence, le taux de compression de

la matrice factorisée peut être meilleur que celui de la matrice après assemblage, ce qui est contraire à ce qui est observé pour un maillage adapté (figure 3.30).

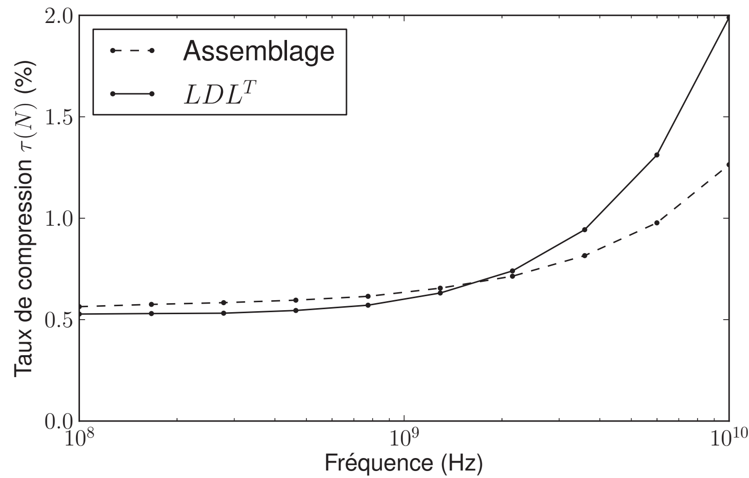


Figure 3.37 – Taux de compression en fonction de la fréquence pour un maillage raffiné.

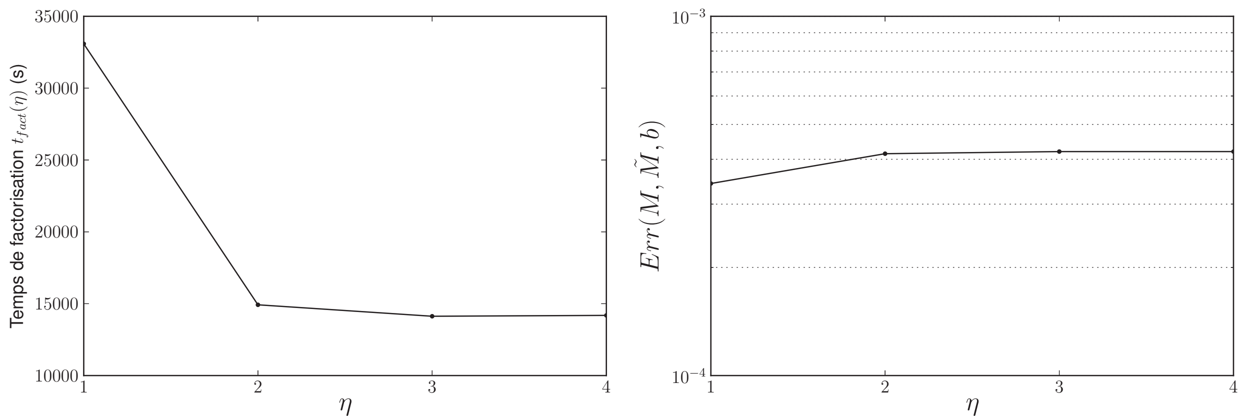


Figure 3.38 – Influence de η sur le temps de décomposition et l'erreur.

3.5.1.3.5 Influence de η Pour la compression, il a été mis en évidence dans les cas considérés une valeur de η comprise entre 2 et 3 est recommandable. La figure 3.38 montre l'impact d'une variation de η sur le temps de factorisation, et sur la précision des résultats. Un palier de temps de calcul et de compression apparaît à partir de $\eta = 2$, avec le temps de calcul le plus faible pour $\eta = 3$. L'erreur reste similaire pour $2 \leqslant \eta \leqslant 4$, cependant la légère remontée du temps de calcul pour $\eta = 4$ permet de recommander une valeur plus faible de η, 2 ou 3.

3.5.1.3.6 Conclusions De même que pour la compression, il est possible de dégager des observations des tests précédents :

1. Il est possible de résoudre avec une précision contrôlée un système linéaire issu d'une formulation BEM pour les ondes, même à haute fréquence ($\kappa l \gg 1$ avec l une longueur caractéristique de l'objet) ;

2. La croissance du temps de décomposition est moins bonne que les estimations théoriques pour les noyaux asymptotiquement lisses. Néanmoins,

   — Une croissance plus rapide est également observée dans la littérature pour un noyau asymptotiquement lisse ;

   — Cette croissance est dépendante de la forme de l'objet ;

   — Les observations sont compatibles avec celles de l'assemblage.

3. La diminution de la fréquence d'étude à maillage fixé permet de réduire le temps de calcul global. La précision reste bonne jusqu'à un maillage en $\lambda/1000$, avec une augmentation de l'erreur pour les cas les plus raffinés. Cette augmentation de l'erreur est un élément connu des formulations BEM présentées dans ce document. Il est possible de passer au-delà de cette limitation avec une formulation adaptée, dont l'intégration à la bibliothèque $\mathcal{H}$-Matrice fera l'objet de travaux futurs.

4. La précision est maîtrisée, avec une légère augmentation de l'erreur avec la taille du problème. De plus, le paramètre ε est globalement une bonne mesure de l'erreur finale sur les cas représentés ici.

5. L'augmentation de la valeur de η permet de diminuer le temps de décomposition, au prix d'une légère augmentation de l'erreur. Le meilleur compromis est ici $\eta = 2$ ou 3.

3.5.2 Autres exemples

3.5.2.1 Sphère

Il n'existe que peu de configurations dans lesquelles le problème de la diffraction d'une onde électromagnétique par un obstacle borné a une solution analytique connue. La diffraction par une sphère est un de ces cas, et la solution est alors donnée par les séries de Mie [31]. On note h_n les fonctions de Hankel sphériques du premier type, $j_n := \mathfrak{Re}(h_n)$ et P_n^l les fonctions de Legendre (dont l'expression est donnée dans [10]). Pour une sphère parfaitement conductrice de rayon r illuminée par une onde plane arrivant de la direction (en coordonnées sphériques) $\theta = \phi = 0$ et polarisée verticalement, la Section Efficace Radar (SER) dans la direction (θ, ϕ) est :

$$\sigma(\theta, \phi) = \frac{4\pi}{\kappa^2} \left(|S_1(\theta)|^2 \cos^2 \phi + |S_2(\theta)|^2 \sin^2 \phi \right) \tag{3.5.1}$$

avec

$$\begin{cases} l\psi_n(x) = xj_n(x) \\ \zeta_n(x) = xh_n(x) \\ a_n = \dfrac{\psi_n(\kappa r)}{\zeta(\kappa r)} \\ b_n = \dfrac{\psi_n'(\kappa r)}{\zeta_n'(\kappa r)} \\ S_1(\theta) = -i \displaystyle\sum_{n=1}^{+\infty} (-1)^n \frac{2n+1}{n(n+1)} \left(b_n \frac{\partial P_n^1(\cos\theta)}{\partial\theta} - a_n \frac{P_n^1(\cos\theta)}{\sin\theta} \right) \\ S_2(\theta) = i \displaystyle\sum_{n=1}^{+\infty} (-1)^n \frac{2n+1}{n(n+1)} \left(-a_n \frac{\partial P_n^1(\cos\theta)}{\partial\theta} + b_n \frac{P_n^1(\cos\theta)}{\sin\theta} \right) \end{cases}$$

et il existe des solutions similaires pour d'autres types de conditions limites, que nous ne détaillerons pas ici.

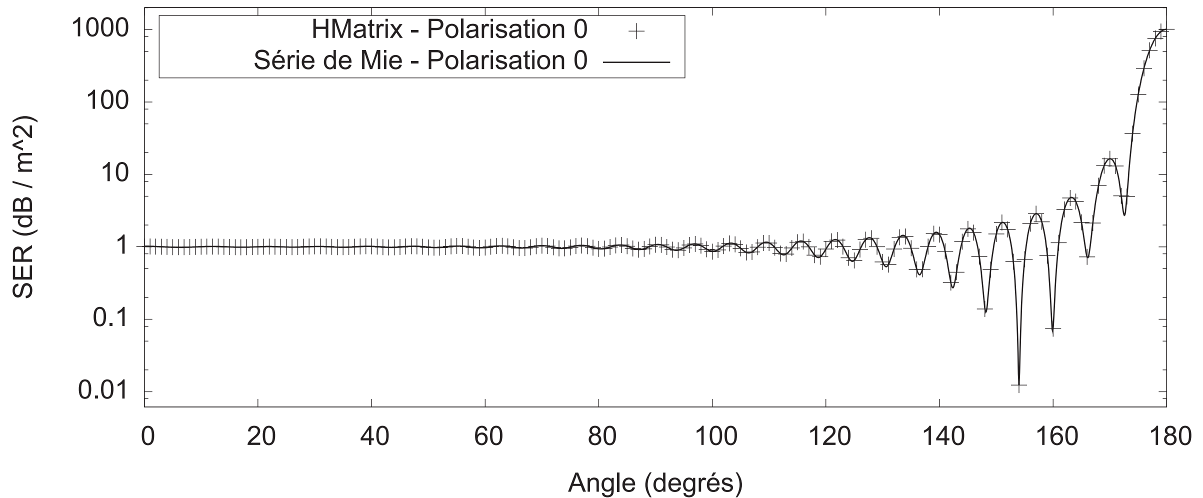


Figure 3.39 – Comparaison entre la solution analytique (3.5.1) et le solveur $\mathcal{H}$-Matrice.

On considère ici une sphère de diamètre 1m, dont le maillage est généré de façon paramétrique. Ce maillage est très homogène, la variation de taille entre les arêtes les plus petites et les plus grandes étant environ de 20%. La longueur d'onde est de 10cm, et la taille moyenne d'une arête est $\lambda/10 = 1$cm, pour un nombre de degrés de liberté égal à 451 632. On utilise les paramètres de calcul de la table 3.2. La figure 3.39 représente la comparaison entre le résultat calculé par le solveur $\mathcal{H}$-Matrice et la solution analytique. On constate un accord excellent pour tous les angles entre les deux solutions.

3.5.2.2 Cône-Sphère

Les calculs de la section précédente ont été reproduits avec une autre classe d'objets, servant souvent de point de comparaison pour la diffraction d'ondes électromagnétiques : le « cône-sphère ». Cette classe est d'intérêt pour la validation de logiciels de calculs, car elle présente des caractéristiques proches à nombre d'objets « furtifs ».

Matériau	Parfaitement conducteur
Rayon de la sphère	de 1 à 10 λ
Angle au sommet	15°
Fréquence	15GHz
Maillage	Taille des arêtes $\lambda/12$, raffiné sur 3 mailles à la pointe
Degrés de liberté	15243 – 960 717
Illumination	— Onde plane $\frac{\kappa}{\|\kappa\|} = (0,0,1)$, — Amplitude 1 — Polarisation horizontale

TABLE 3.4 – Configuration physique pour les cônes-sphères.

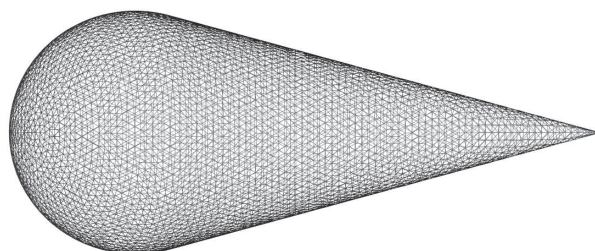


FIGURE 3.40 – Maillage de cône-sphère à 15243 inconnues.

De plus, la simulation de cet objet est un problème difficile. Il possède une singularité (pointe), et le champ diffracté par cet objet est issu de phénomènes physiques fins, en particulier la recombinaison de rayons rampants, qui sont difficilement reproduits avec précision par nombre de méthodes numériques. Les résultats de surface équivalente radar ont une très forte dynamique sur cet objet.

3.5.2.2.1 Configuration physique Un cône-sphère est un objet présentant une symétrie de révolution. Il est composé d'un cône, prolongé par une demi-sphère. Les paramètres choisis pour générer les objets de cette section sont résumés par la table 3.4, et une représentation de l'objet est donnée par la figure 3.40. Le maillage est légèrement plus fin que pour les cylindres ($\lambda/12$ au lieu de $\lambda/10$). Ceci est lié à la difficulté de l'objet, et pour une étude réaliste le maillage serait probablement encore plus fin. Nous avons cependant décidé de ne pas l'affiner pour ne pas artificiellement favoriser le solveur $\mathcal{H}$-Matrice.

3.5.2.2.2 Résultats On utilise les mêmes configurations de calcul (table 3.2) et informatique (table 3.3) que pour les cylindres. Les résultats sont globalement fortement similaires à ceux obtenus dans la section précédente, en termes de passage à l'échelle et précision. Pour cette raison, nous ne reproduisons ici que le graphique de consommation de mémoire, correspondant dans le cas des cylindres au graphique 3.18.

De même que pour les cylindres, utiliser un cône-sphère dont l'angle au sommet est plus faible (et donc plus allongé) a les mêmes conséquences. Une illustration de la dynamique du

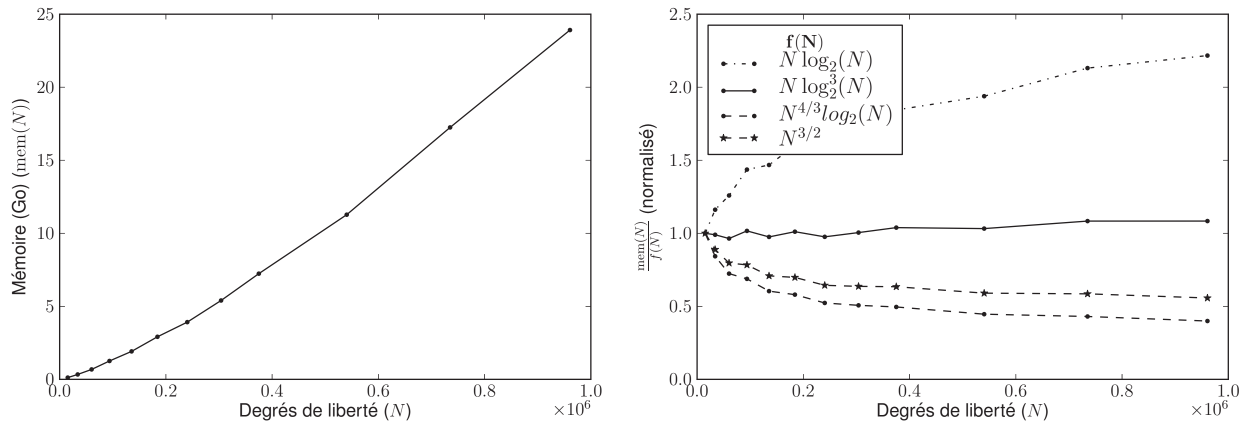


Figure 3.41 – Quantité de mémoire en fonction du nombre de degrés de liberté, cône-sphère.

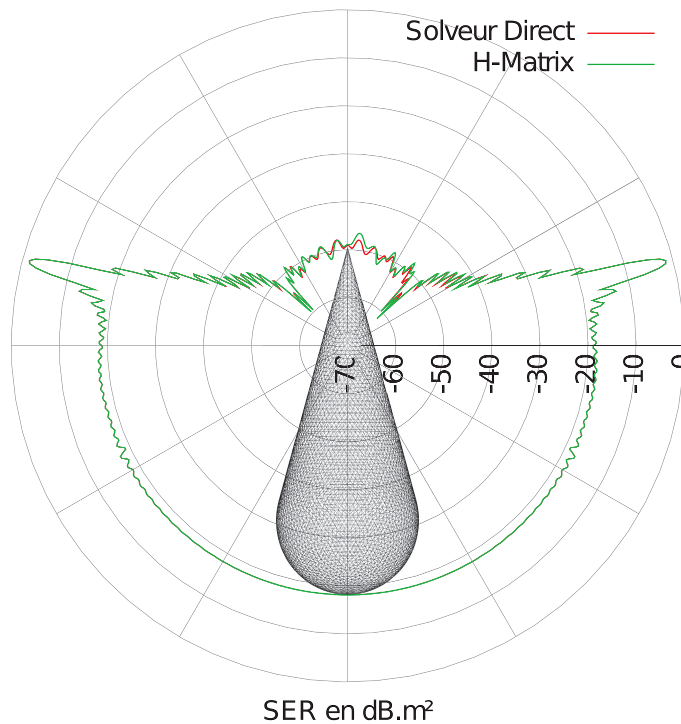


Figure 3.42 – Comparaison des résultats entre le solveur direct et le solveur $\mathcal{H}$-Matrice.

résultat et de la précision obtenue est donnée par la figure 3.42. Celle-ci compare le résultat obtenu après le même calcul effectué avec un solveur direct et le solveur $\mathcal{H}$-Matrice. Il s'agit d'un calcul de SER monostatique, c'est-à-dire que l'antenne d'illumination et l'antenne de réception se situent au même endroit. Pour ce faire, on calcule le champ lointain (*cf.* l'équation (2.2.21)). Il s'agit de la grandeur d'intérêt en pratique, et c'est celle-ci qui sera mesurée, plus précisément :

$$S(\theta, \phi) = 10 \frac{\log_{10}(|\boldsymbol{E}_{diff}|)}{|\boldsymbol{E}_{inc}|}$$

avec une onde plane incidente. On constate un excellent accord entre les deux solveurs, à l'exception des très faibles niveaux (entre -60 et -70dB), ce qui est attendu d'un solveur approché. L'objet géométrique est superposé sur cette figure, permettant de l'interpréter plus facilement. La partie basse de la figure est très proche de la SER monostatique d'une sphère, c'est-à-dire une valeur constante en θ et en ϕ, dépendante de sa taille. Les deux pics symétriques correspondent à une onde incidente orthogonale à la partie conique, et les faibles niveaux correspondent à une onde incidente sur la pointe.

Solveur	**Assemblage (CPU.s)**	**Factorisation (CPU.s)**
Direct	18983.424	1304949.632 (700314.752)
$\mathcal{H}$-Matrice	1044.81	1762.85
Ratio	18.1	740.2 (397.2)

TABLE 3.5 – Comparaison du temps de calcul entre le solveur direct et le solveur $\mathcal{H}$-Matrice pour le cône-sphère à 184125 inconnues.

Temps de calcul Le résultat de la figure 3.42 correspond à un calcul avec 184 125 degrés de liberté. Le calcul a été effectué sur 4 nœuds identiques à la machine Curie-16 pour le solveur direct (64 processeurs), et sur un cœur d'un nœud de cette machine pour le solveur $\mathcal{H}$-Matrice. Du fait de la configuration particulière de la machine, l'espace de stockage local sur les nœuds n'est pas suffisant pour conserver la matrice dans le cas du solveur direct. Un système de fichiers global est utilisé, ce qui pénalise grandement les performances. Les nombres entre parenthèses dans la table 3.5 ont été corrigés pour retirer l'impact de l'accès aux fichiers non locaux pour le solveur direct. Le solveur utilisé est le solveur direct d'ASERIS, SPIDO2, avec une factorisation LDL^T. C'est un solveur direct, *Out-Of-Core*, avec une parallélisation hybride OpenMP/MPI.

3.6 Analyse et optimisations

3.6.1 Analyse de la compression sur des cas réalistes

Les tests de la section précédente renvoient une image contrastée sur le taux de compression attendu d'un bloc de $\mathcal{H}$-Matrice pour l'équation des ondes. Pour un noyau asymptotiquement lisse, il est attendu que le rang d'approximation ne dépende pas de la taille du bloc, sous réserve que la valeur du paramètre d'admissibilité η soit bien choisie.

Dans le cas du noyau de Helmholtz, qui est oscillant, une faible dépendance à la taille des blocs semble être observée. Néanmoins, les tests sur des objets canoniques mettent en évidence la dépendance de la compression à la forme de l'objet. Il est donc naturel de s'interroger sur le comportement de la compression pour des objets réalistes. Nous prenons dans cette section deux calculs, en électromagnétisme et acoustique, issus d'applications réelles. Le comportement de la compression des blocs est examiné, afin de dégager des observations issues de situations d'intérêt pratique.

3.6.1.1 Description des cas

Électromagnétisme On considère un A319neo, le même avion que celui représenté sur la figure 3.2. Cet avion est inscrit dans une boîte englobante de $34 \times 34 \times 11$m. Il est ici maillé en configuration « lisse » de croisière, c'est-à-dire sans train d'atterrissage, et sans le radôme, qui est transparent aux ondes électromagnétiques. Le maillage est généré à partir d'une représentation CAO de l'avion, et la taille moyenne des arêtes est de 75mm, ce qui donne un maillage adapté pour 333MHz, qui est la fréquence d'étude. Les autres paramètres de simulation sont identiques à ceux résumés par la table 3.2, et on fait un calcul EFIE. Le nombre d'inconnues est 475 012.

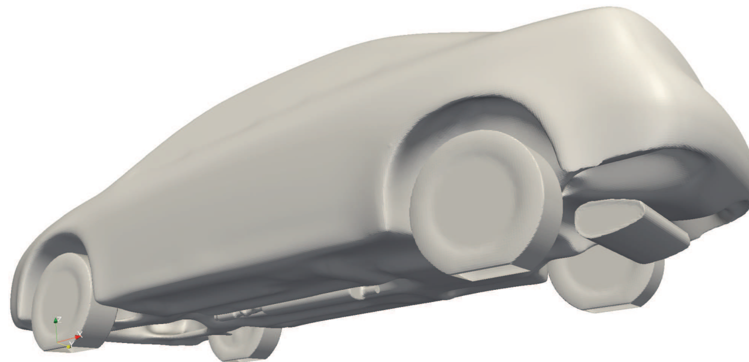


Figure 3.43 – Voiture. Les pneus sont tronqués car celle-ci est posée sur un plan de sol infini non représenté.

Acoustique L'objet considéré est une voiture (figure 3.43), placée sur un plan rigide infini, représentant le sol. Mathématiquement, ce plan correspond à un plan de symétrie pour les interactions, ce qui ajoute des termes dans l'expression des éléments de la matrice d'interaction. Le logiciel de simulation utilisé est Actipole, le pendant acoustique d'Elfipole. Une formulation analogie à l'EFIE est utilisée, nous ne la détaillerons pas ici. Le nombre d'inconnues est 310 208, et la fréquence d'étude est fixée à 800Hz. Le maillage a une finesse variable, comprise entre $\lambda/10$ et $\lambda/30$ suivant les zones, et a été généré à partir d'une représentation CAO. La voiture est composée d'un matériau rigide, et les paramètres de simulation sont ceux de la table 3.2.

3.6.1.2 Compression

Le premier point à examiner est celui de la croissance du rang d'un bloc avec sa taille. Si cette taille est constante (ou du moins majorée indépendamment de la taille), alors les estimations de complexité de la section 3.4.1 sont valides. Dans le cas contraire le comportement asymptotique attendu sera moins bon. Il est alors important de caractériser la vitesse de croissance du rang. Une croissance trop rapide rendrait une implémentation multi-niveaux inutile, avec dans le cas extrême un gain nul par rapport à une implémentation mono-niveau plus simple.

3.6.1.2.1 Taille des blocs La figure 3.44 décrit la variation du rang avec la taille des blocs, représentée par $\max(|\sigma|, |\tau|)$ pour un bloc $M|_{\sigma\times\tau}$. On rappelle que, puisque la

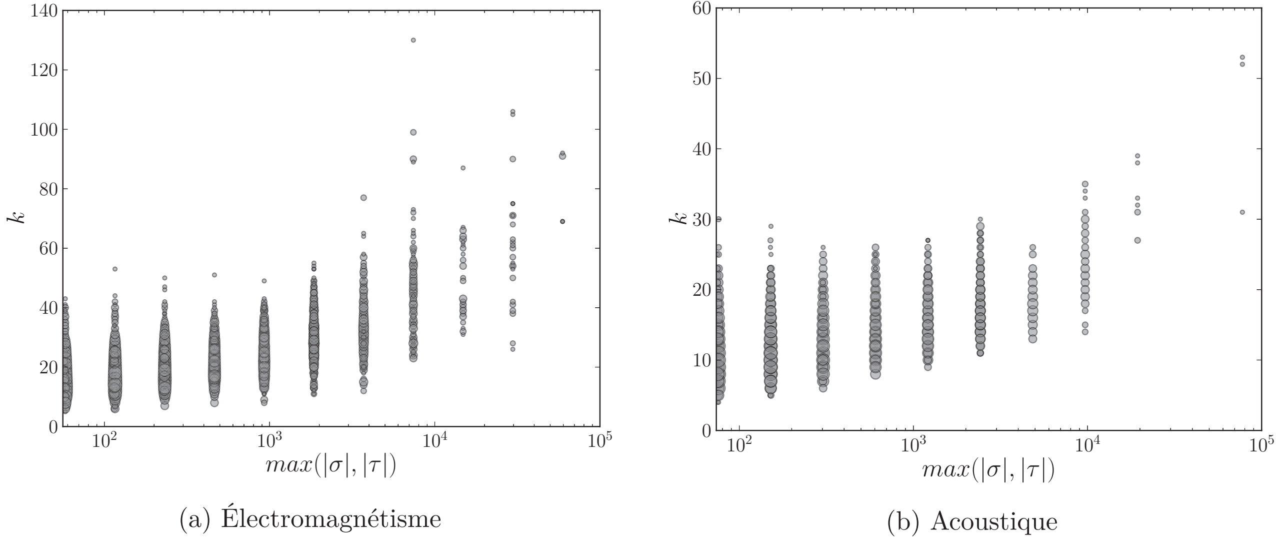


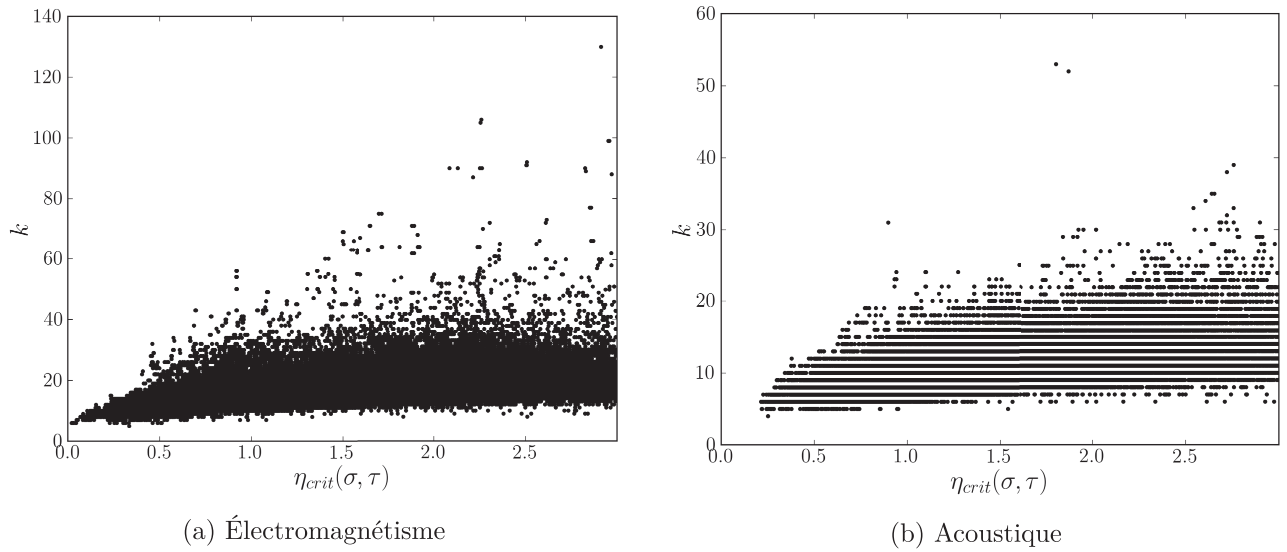
FIGURE 3.44 – Ordre d'approximation en fonction de la taille de bloc. La taille des cercles est logarithmique au nombre d'occurrence d'un couple taille-ordre d'approximation. On note que l'axe des abscisses est en échelle logarithmique.

$\mathcal{H}$-Matrice est construite sur un arbre de bloc $\mathcal{T}_{I\times I}$ (même arbre de groupes pour les lignes et les colonnes), et que l'arbre de groupes est construit par un découpage médian, tous les blocs de la $\mathcal{H}$-Matrice sont presque carrés. Sur cette figure, $\max(|\sigma|, |\tau|)$ est représenté en échelle logarithmique, et le rang en échelle linéaire. Les cercles ont un rayon proportionnel au logarithme de la taille du bloc. On constate donc dans les deux cas une croissance très modeste du rang des blocs. Cette croissance concerne de plus un très petit nombre de blocs, pour la majorité des blocs aucune croissance n'est observée. Enfin, la dispersion sur les résultats est grande. Ceci est attendu, puisque la compression est spécifique à chaque bloc, sa composition influe sur la convergence de l'approximation, et ce d'autant plus que les deux maillages ne présentent pas les régularités des objets canoniques étudiés précédemment.

3.6.1.2.2 Constante d'admissibilité Pour tenter de mieux comprendre la dispersion sur les valeurs, il est utile de visualiser l'évolution du rang d'approximation avec la variation de la constante d'admissibilité critique $\eta_{crit}(\sigma, \tau)$. Celle-ci est définie par :

$$\eta_{crit}(\sigma, \tau) := \frac{\min(diam(\sigma), diam(\tau))}{d(\sigma, \tau)} \tag{3.6.1}$$

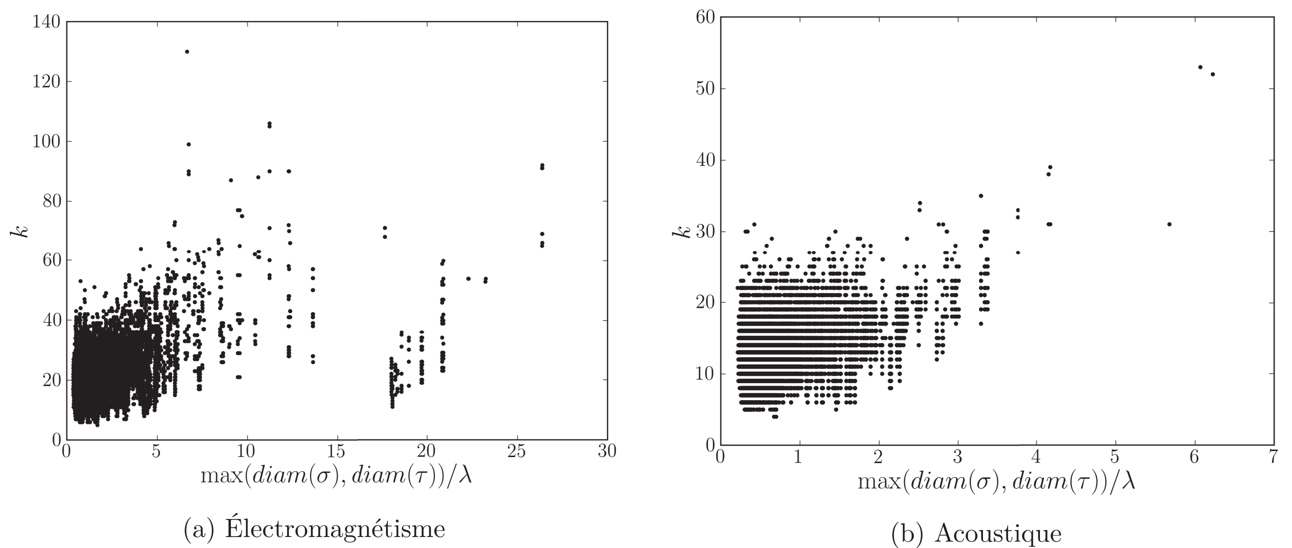
Tout bloc $M|_{\sigma\times\tau}$ admissible pour une valeur donnée de η l'est pour toutes les valeurs de η' telles que $\eta' \leqslant \eta_{crit}(\sigma, \tau)$. Ce nombre représente donc la frontière de l'admissibilité pour un bloc compressé donné. Une faible valeur de $\eta_{crit}(\sigma, \tau)$ correspond à deux groupes dont l'éloignement est grand au regard de leurs tailles respectives, et une valeur infinie à deux groupes adjacents. Dans le cas des noyaux asymptotiquement lisses, il existe des théorèmes garantissant la convergence exponentielle de l'approximation des intégrales de simple et double couche par interpolation pour certaines valeurs de η [22, 24], la vitesse de convergence étant fixée par le choix pratique de η.



(a) Électromagnétisme (b) Acoustique

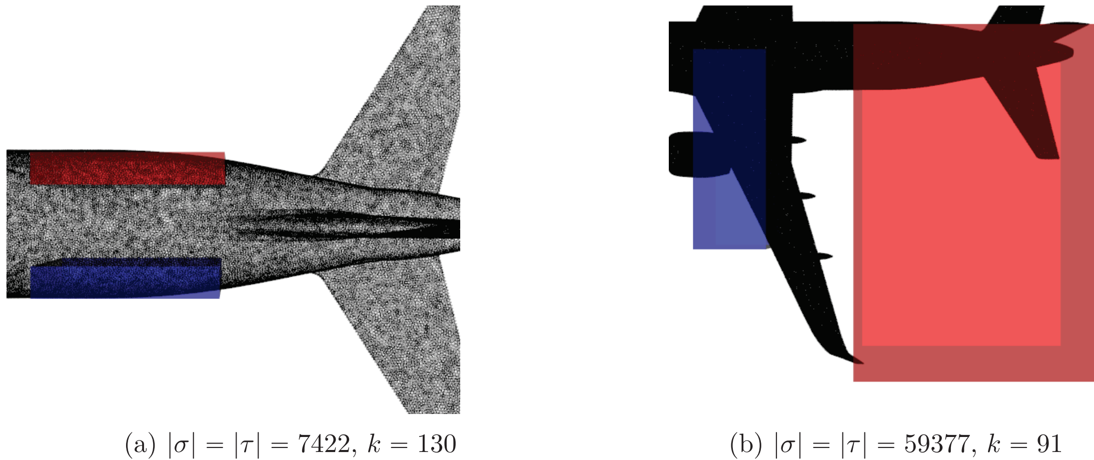
FIGURE 3.45 – Ordre d'appoximation en fonction de η_{crit}.

On constate généralement une augmentation de la dispersion avec l'augmentation de la valeur de $\eta_{crit}(\sigma, \tau)$, avec des valeurs en-dehors de la tendance. Il y a également une légère tendance croissante, ce qui est attendu. En effet, une valeur élevée de η_{crit} correspond à des blocs faiblement séparés par rapport à leurs tailles respective. Pour l'exemple en électromagnétisme, on observe également un plus grand nombre de données en-dehors de la tendance, avec un rang élevé. Cette observation ne peut cependant pas être généralisée à d'autres cas.



(a) Électromagnétisme (b) Acoustique

FIGURE 3.46 – Ordre d'approximation en fonction de la taille des groupes en longueur d'onde.



(a) $|\sigma| = |\tau| = 7422$, $k = 130$

(b) $|\sigma| = |\tau| = 59377$, $k = 91$

FIGURE 3.47 – Blocs de la $\mathcal{H}$-Matrice ayant un ordre d'approximation élevé. Le groupe σ est représenté en bleu, τ en rouge.

3.6.1.2.3 Taille des groupes Un autre paramètre d'influence sur la compression est la taille des groupes par rapport à la longueur d'onde. Dans le cas d'un noyau non asymptotiquement lisse, la taille d'un groupe en fonction du nombre de longueurs d'onde peut avoir un impact sur la compression. Ainsi, pour de la FMM directionnelle [77, 78], le critère de séparation tient compte de la longueur d'onde, et deux groupes proches mais de grande extension par rapport à la longueur d'onde ne seront pas admissibles à un niveau donné (ceci est également fonction de la géométrie du groupe, par le critère d'ouverture angulaire). La figure 3.46 représente le rang de l'approximation en fonction de la taille en longueur d'onde des groupes composant les blocs de la $\mathcal{H}$-Matrice. Une fois de plus, il n'y a pas de croissance constatée avec l'augmentation de la taille des groupes.

Il est même intéressant de noter que dans le cas de l'A319, les blocs ayant un rang élevé ne sont pas nécessairement des blocs de grande taille au regard de la longueur d'onde. La figure 3.47 montre deux blocs ayant un rang d'approximation élevé : le premier est un bloc de taille 7422×7422 avec $k = 130$, et le second est de taille 59377×59377 avec $k = 91$. Dans le premier cas, la position relative des deux blocs inviterait à considérer en plus du critère de séparation des groupes, un critère lié à l'ouverture angulaire, inspiré de celui de la FMM directionnelle. Le second exemple invite, lui, à considérer un découpage plus intelligent que le découpage médian dans l'arbre de groupes, avec par exemple un découpage hybride géométrique / médian. Ces pistes d'amélioration n'ont néanmoins pas été explorées.

3.6.1.2.4 Conclusions Sur les cas pratiques présentés, l'ordre d'approximation n'augmente pas de manière significative avec la taille des blocs. Une grande variabilité est néanmoins observée. Celle-ci ne peut être expliquée simplement par le nombre de degrés de liberté dans un bloc, ni par la valeur de $\eta_{crit}(\sigma, \tau)$, ni par la taille des blocs en longueur d'onde. Une partie de cette variabilité est certainement liée à la compression adaptative bloc par bloc. Néanmoins, il est probablement possible de réduire le nombre de blocs ayant un ordre d'approximation élevé en affinant la définition du critère d'admissibilité, ou en divisant ces blocs au moment de la compression. Ces modifications apporteraient

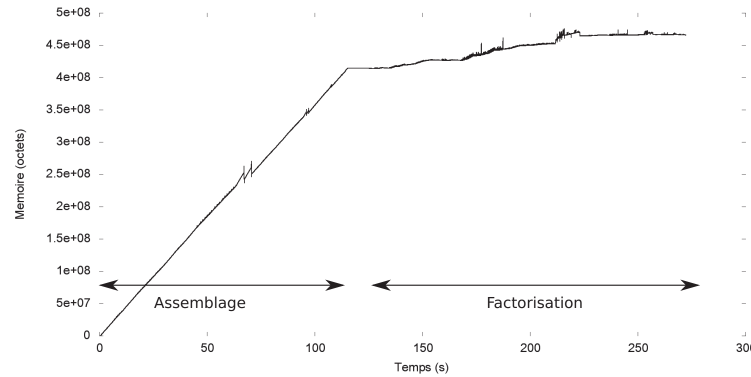


Figure 3.48 – Évolution de la consommation mémoire pendant le calcul

sans doute des améliorations notables au temps d'exécution des algorithmes ainsi qu'à la consommation mémoire, et feront l'objet de travaux futurs.

3.6.2 Consommation mémoire et répartition du temps de calcul

3.6.2.1 Consommation mémoire

La consommation mémoire d'un solveur $\mathcal{H}$-Matrice n'est pas aisément prévisible, sauf dans le cas où l'ordre d'approximation des $\mathcal{R}k$-Matrices est fixé, ce qui n'est pas le cas ici. Elle dépend pour l'assemblage de l'algorithme de compression, de la précision cible, de la méthode de construction de l'arbre de groupes, du critère d'admissibilité, de la géométrie, de la fréquence, *etc.*

De plus, une fois la matrice assemblée, la consommation mémoire n'est pas fixe, elle peut varier lors des additions par recompression adaptative. Il est donc utile de caractériser le comportement mémoire d'un solveur $\mathcal{H}$-Matrice, et en particulier le pic de consommation, qui est l'élément limitant la taille des calculs possibles. La consommation mémoire maximale est par ailleurs fortement contraignante pour les méthodes nécessitant d'assembler l'intégralité du bloc, comme l'algorithme ACA avec pivotage total, ou la SVD, puisque la consommation mémoire est au moins $\mathcal{O}(N_{max}^2)$, avec N_{max} la taille maximale d'une feuille [5]. Nous considérons donc dans cette section la consommation dans le temps d'un calcul utilisant l'algorithme ACA avec pivotage partiel.

On considère une géométrie de « cône-sphère » semblable à la figure 3.8 à 29.535 inconnues, et dont le maillage a une taille moyenne d'arête de $\lambda/10$ pour la fréquence d'étude (15GHz ici). La figure 3.48 représente l'évolution de la consommation mémoire au cours du calcul. Dans ce cas, le taux de compression à l'issue de l'assemblage est meilleur qu'après factorisation (5,93% contre 6,67%), ce qui explique l'augmentation de la consommation mémoire durant la factorisation. Ce n'est cependant pas une caractéristique générale ; la consommation mémoire diminue durant la factorisation pour certains cas, et reste en général relativement proche de sa valeur après assemblage, avec de faibles variations. La première partie de la courbe est proche d'une droite. Ceci est attendu, car comme la remarque 3.14 le précise, le terme dominant dans la complexité de l'assemblage

5. Et $\mathcal{O}(pN_{max}^2)$ pour une implémentation parallèle avec p processeurs.

est le calcul des lignes et des colonnes de la matrice utilisées dans l'algorithme ACA. Ce calcul est linéaire en temps et en consommation mémoire, ce qui explique le résultat. Par ailleurs, cette ligne comporte des accidents de plus ou moins grande ampleur, la plupart n'étant pas visibles sur la figure 3.48. Ils sont dus au stockage temporaire des lignes et colonnes lors de l'assemblage, et également à la recompression adaptative par SVD après assemblage d'un bloc admissible, faisant diminuer la consommation mémoire.

Des accidents de même nature se retrouvent lors de la factorisation. Ils sont ici liés à la recompression adaptative après addition de deux $\mathcal{R}k$-Matrices et aux conversions hiérarchiques. Enfin, il faut noter que le nombre d'allocations mémoires effectuées par un solveur $\mathcal{H}$-Matrice est élevé : ce calcul comporte 6.070.138 allocations mémoire (et autant de libérations) pour une $\mathcal{H}$-Matrice à 17.965 feuilles.

3.6.2.2 Répartition du temps de calcul

Les estimations de la section 3.4 donnent des complexités asymptotiques pour les opérations sur les $\mathcal{H}$-Matrices, mais dans le cadre restrictif d'un ordre d'approximation fixe. Elles ne sont par ailleurs qu'asymptotiques, et ne prennent pas en compte l'efficacité informatique relative des diverses opérations. L'efficacité d'une décomposition en valeurs singulières est en pratique bien inférieure à la puissance théorique de crête d'une machine, en particulier pour les matrices de petite taille, qui nous importent ici. L'efficacité dépend par ailleurs également de l'algorithme employé, qui sont au nombre de deux dans LAPACK (il en existe d'autres).

3.6.2.2.1 Conditions de calcul Nous prenons comme exemple dans cette section le même calcul que celui de la section précédente, et donnons des rapports pour l'implémentation séquentielle des algorithmes. Le temps de calcul est de 115,4s pour l'assemblage de la $\mathcal{H}$-Matrice et 147,8s pour la factorisation.

Les paramètres de ce calcul sont :

Matériel Intel Xeon « Sandy Bridge », 2,3GHz

Logiciel Compilateur `g++` 4.7, Intel MKL 10.3 (BLAS/LAPACK), Linux 64 bits

SVD XGESDD dans LAPACK, version « *divide and conquer* » de la factorisation SVD.

Cette machine partage la même architecture que la machine Curie-16, et les résultats de cette section devraient être directement transposables à cette machine.

Appels élémentaires Avec M une matrice pleine, T une matrice triangulaire, R une $\mathcal{R}k$-Matrice, et H une $\mathcal{H}$-Matrice, les appels élémentaires les plus significatifs sont, par ordre décroissant :

Nom	Fonction	Temps cumulé (s)
compute_row	Crée $M_{\{i\}\times\tau}$	44,6
svd	Factorise $M = U\Sigma V^*$	44,0
qr	Factorise $M = QR$	42,9
compute_col	Crée $M_{\sigma\times\{j\}}$	36,6
prod_q	Calcule $M \leftarrow M * Q$	27,5
assemble_full	Calcule $M\|_{\sigma\times\tau}$	25,1
mul_h_rk	Calcule HR	4.6
mul_rk_h	Calcule RH	4,5
mul_rk_rk	Calcule RR	2,8
mul_tr	Calcule MT	2,5

Les rapports entre ces opérations varient avec le nombre de degrés de liberté. La figure 3.49 donne cette répartition pour les mêmes opérations que celles du tableau précédent. Les objets calculés correspondent au même objet que précédemment, avec la même finesse de maillage et fréquence d'étude. Le maillage est généré à partir d'un profil analytique, et est mis à l'échelle pour obtenir des maillages de tailles diverses.

Les tendances suivantes se dégagent :

- Le temps d'assemblage des blocs pleins (`assemble_full`) devient négligeable quand la taille de l'objet grandit ;
- L'opération la plus coûteuse devient la décomposition QR, particulièrement si le temps de multiplication par Q est compté ;
- La décomposition SVD est également coûteuse ;
- L'assemblage des lignes et des colonnes des blocs matriciels est proportionnellement moins importante avec l'augmentation du nombre de degrés de liberté ;
- Les autres opérations sont peu significatives.

Outre l'assemblage, les opérations les plus coûteuses sont les décompositions SVD et QR. Le produit par la matrice Q de la décomposition QR est coûteux car plus complexe qu'un simple produit. Cette matrice n'est pas construite, seul le produit est calculé à la volée.

3.6.3 Recompression

La recompression d'une $\mathcal{R}k$-Matrice décrite à la section 2.4.2.3 représente une partie importante du temps de calcul de la factorisation d'une $\mathcal{H}$-Matrice, comme l'atteste la figure 3.49. L'optimisation de cette opération est donc importante, et cette section a pour objet de discuter son implémentation.

Dans la suite de cette section, les comptages d'opérations donnés sont ceux trouvés dans [25, Annexe C], et donc valables pour l'implémentation de référence « netlib » de BLAS et LAPACK. Pour le calcul de la décomposition en valeurs singulières, nous prendrons comme référence la SVD de Golub-Reinsch [51, section 5.4.5]. De plus, ces comptages sont donnés pour le cas réel, le cas complexe dépendant de l'implémentation du produit matriciel. En effet, certaines implémentations de BLAS comme MKL, OpenBLAS ou GotoBLAS fournissent l'opération non standard `gemm3m` permettant de réduire le nombre d'opérations nécessaires à un produit matriciel complexe.

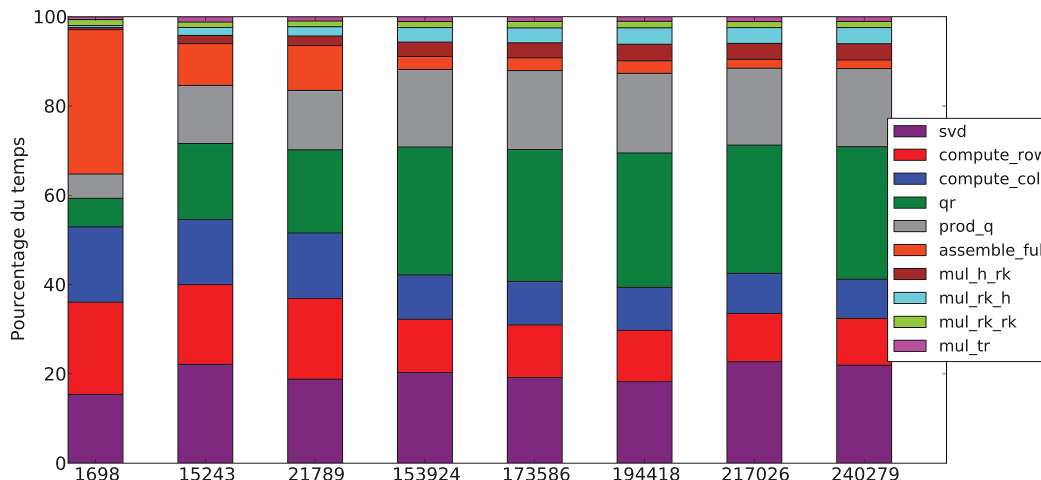


FIGURE 3.49 – Rapport de temps d'exécution des opérations élémentaires en fonction du nombre de degrés de liberté.

3.6.3.1 Décomposition QR

Dans cette section, nous considérons un bloc matriciel $M|_{\sigma\times\tau}$ représenté par une $\mathcal{R}k$-Matrice, avec $|\sigma| = m$ et $|\tau| = n$, soit $M|_{\sigma\times\tau} \simeq A.B^T$, $A \in \mathbb{C}^{m\times k}$ et $B \in \mathbb{C}^{n\times k}$. Nous donnons plus de précisions sur son calcul qui représente une part importante du temps de calcul global.

3.6.3.1.1 Calcul de la décomposition QR L'implémentation de la factorisation QR dans LAPACK ne calcule pas explicitement Q. Cette matrice est représentée sous la forme d'un produit de transformations élémentaires, différentes selon la méthode de calcul (Householder, Givens, *etc.*) [51]. Dans le cas de l'implémentation de référence de LAPACK, le nombre d'opérations nécessaires pour le calcul de la décomposition est pour la matrice $A \in \mathbb{C}^{m\times k}$:

Additions	$mk^2 - \frac{1}{3}k^3 + mk + \frac{1}{2}k^2 + \frac{23}{6}k$
Multiplications	$mk^2 - \frac{1}{3}k^3 + \frac{1}{2}k^2 + \frac{5}{6}k$
Total	$2mk^2 - \frac{2}{3}k^3 + mk + k^2 + \frac{14}{3}k$

Le comptage est identique pour la décomposition de B, en remplaçant m par n dans les expressions précédentes.

3.6.3.1.2 Calcul ou application de Q La décomposition QR ne forme pas la matrice Q, et il est donc possible de (a) calculer Q puis de multiplier Q par $\tilde{U}$ et $\tilde{V}$ (`orgqr` et `gemm`) ; ou (b) multiplier les matrices $\tilde{U}$ et $\tilde{V}$ par Q (`ormqr`).

On suppose ici que le nombre de valeurs singulières conservées est $k' \leqslant k$, donc $\tilde{U} \in \mathbb{C}^{k\times k'}$. Les comptes d'opérations pour ces deux choix sont :

	SORGQR + GEMM	**SORMQR**
Additions	$mk^2 - \frac{1}{3}k^3 + k^2 - \frac{5}{3}k + mkk'$	$2mkk' - kk'^2 + 2kk'$
Multiplications	$mk^2 - \frac{1}{3}k^3 + k^2 - mk + \frac{1}{3}k + mkk'$	$2mkk' - kk'^2 + kk'$
Total	$2mk^2 - \frac{2}{3}k^3 + 2k^2 - mk - \frac{4}{3}k + 2mkk'$	$4mkk' - 2kk' + 3kk'$

Le dernier terme dans la première colonne de ce tableau correspond au produit de la matrice Q résultante avec la matrice $\tilde{U}$. Afin d'illustrer le rapport entre les deux méthodes, la figure 3.50 représente le ratio entre le nombre d'opérations flottantes nécessaires pour les deux implémentations possibles, en fonction du rapport k/k', pour $m = 4 \times 10^4$ et k entre 1 et 100. On constate qu'à l'exception de l'absence de gain en compression ($k = k'$), la seconde méthode est plus efficace en nombre d'opérations flottantes.

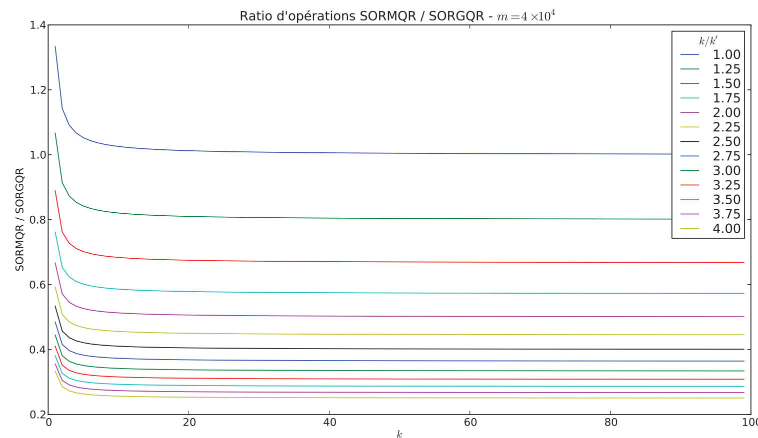
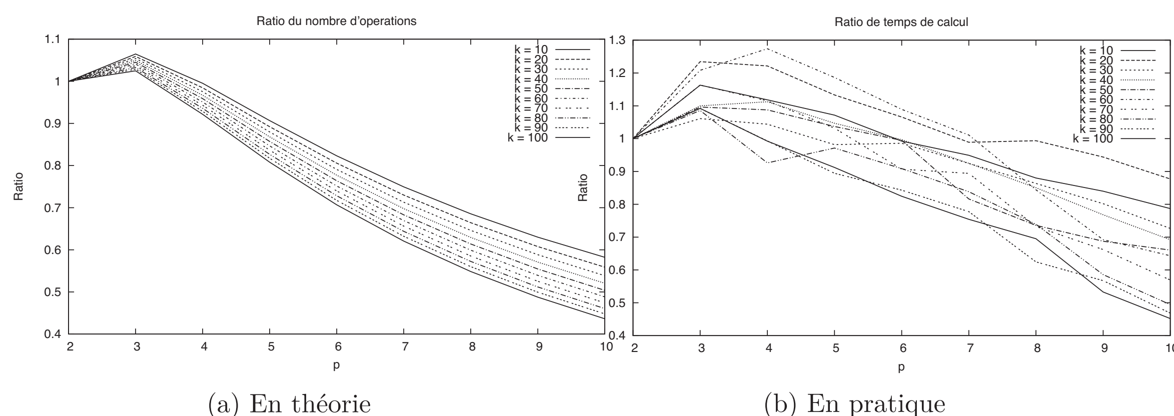


Figure 3.50 – Rapport du nombre d'opérations flottantes entre les deux méthodes de multiplication par Q.

3.6.3.2 Recompresser à chaque fois ?

Le nombre d'opérations nécessaires pour la recompression d'une $\mathcal{R}k$-Matrice n'est pas linéaire en k, mais quadratique ou cubique selon les termes. Dès lors, dans le cas d'une suite d'additions, il est plus avantageux en nombre d'opérations flottantes de recompresser après chaque addition. Cependant, ceci n'est pas nécessairement le cas en pratique. En effet, pour une $\mathcal{R}k$-Matrice comportant un faible nombre de colonnes (k petit), les unités de calcul ne sont pas exploitées de manière optimale. Effectivement, les factorisations sont alors proches d'opérations de type BLAS2, qui ne sont pas favorables aux machines modernes, contrairement aux opérations de type BLAS3. Il est ainsi possible que dans le cas de l'addition de $p > 2$ $\mathcal{R}k$-Matrices, il soit plus avantageux de ne recompresser qu'une fois toutes les additions effectuées. Ceci nécessite plus d'opérations, mais ces dernières sont plus efficaces sur les processeurs modernes.

Afin de tester cela, nous considérons un bloc $R := A.B^T$ carré de taille 4×10^4 représentant des interactions lointaines. Celui-ci est obtenu de la même manière que pour les tests



(a) En théorie (b) En pratique

Figure 3.51 – Comparaison de la recompression globale ou à chaque étape, pour $m = n = 10^4$ et $k = 10 - 100$. Dans les deux cas, le ratio entre une seule recompression et une recompression globale est représenté. Une valeur inférieure à 1 avantage une recompression à chaque somme.

de la section 2.4.1, c'est-à-dire en considérant l'interaction du nez d'un A319 avec cette même géométrie translatée. Soit $T_{k \to k'}(R)$ l'opération de recompression d'une $\mathcal{R}k$-Matrice R. On compare le temps de calcul des opérations :

$$T_{k \to k'}\left(\sum_{i=1}^{p} R\right) \quad \text{et} \quad T_{k \to k'}(R + T_{k \to k'}(R + R) \dots)$$

L'addition répétée de la même $\mathcal{R}k$-Matrice permet de contrôler l'ordre d'approximation après recompression. En effet, dans le cas de l'addition d'une $\mathcal{R}k$-Matrice avec elle-même, le rang n'augmente pas, et l'ordre d'approximation après recompression est donc constant quel que soit le nombre de termes de la somme (ce qui est vérifié en pratique). Dans le premier cas, il est nécessaire de faire une recompression d'une $\mathcal{R}k$-Matrice d'ordre pk avec k l'ordre de R, et dans le second, $p - 1$ recompression de $\mathcal{R}k$-Matrices d'ordre $2k$. La figure 3.51 montre le ratio de nombre d'opérations entre les deux méthodes, pour $m = n = 10^4$ en fonction de p et de l'ordre initial k, en nombre d'opérations et en pratique. Les calculs pratiques ont été réalisés avec un processeur Intel « Sandy Bridge », avec Intel MKL 10.3, version séquentielle. On constate que la recompression à chaque étape est en théorie plus avantageuse qu'elle ne l'est en pratique, ce qui est lié aux effets de cache et de vectorisation du code (exploitation ou non de blas3). Une étude plus approfondie serait alors nécessaire pour obtenir une heuristique permettant d'optimiser les performances.

3.7 Conclusion

Ce chapitre présente la théorie et la pratique de la construction et de la manipulation des $\mathcal{H}$-Matrices. Nous avons illustré les idées directrices de cette méthode, ainsi que l'écriture et l'implémentation des algorithmes. Nous avons en particulier pris soin de présenter les algorithmes dans un niveau de détail suffisant à leur implémentation, en tâchant de préciser les points délicats de celle-ci.

Tout au long de ce chapitre, il est fait mention de choix ou d'heuristiques. Ces alternatives ne trouvent bien souvent pas de justification théorique dans la littérature, et ce document ne prétend pas ouvrir une compréhension nouvelle de ces dernières. En revanche, des illustrations concrètes sont données, et permettent d'entrevoir le grand potentiel applicatif de cette méthode. Il est important de noter que bien que le noyau de l'équation de Helmholtz qui est à la base des méthodes BEM pour les ondes ne soit pas asymptotiquement lisse, d'excellents résultats sont atteints en pratique, en termes de précision et de temps de calcul. Ceci a été mis en évidence à l'aide d'une étude systématique de la méthode dans le cadre des éléments finis de frontière pour l'électromagnétisme. Des éléments d'explication empiriques du comportement des algorithmes ont été proposés, et des pistes d'optimisation des algorithmes énoncées.

Il en résulte une implémentation optimisée d'un solveur $\mathcal{H}$-Matrice, intégrée à un code de calcul industriel, et dont l'efficacité sur des configurations réalistes a été montrée. Il est cependant clair que l'intérêt de cette méthode reste limité tant que celle-ci n'a pas d'implémentation parallèle, en particulier en comparaison aux méthodes FMM parallèles et *Out-Of-Core* actuellement déployées industriellement [94]. La parallélisation des algorithmes est donc une nécessité pour son application à grande échelle sur des problèmes industriels. La suite de ce document sera largement consacrée à la problématique de la parallélisation des algorithmes opérant sur les $\mathcal{H}$-Matrices.

Chapitre 4

Algorithmes parallèles en mémoire partagée

Sommaire

4.1 Introduction 146
4.2 Efficacité parallèle 153
4.3 Ordonnancement par liste et algorithmes BSP 155
4.4 Graphe de tâches et moteur d'exécution 162
4.5 Algorithmes BSP pour les $\mathcal{H}$-Matrices à l'aide de graphes de tâches 168
4.6 Parallélisation à l'aide de graphes de tâches 177
4.7 Implémentation et performances 196
4.8 Conclusion 210

L'implémentation d'un solveur basé sur l'arithmétique des $\mathcal{H}$-Matrices permet un gain considérable en temps de calcul par rapport aux solveurs directs classiques, comme le chapitre précédent l'illustre. Cet avantage est néanmoins à relativiser, dans la mesure où une implémentation massivement parallèle d'un solveur classique peut devenir compétitive, et une implémentation parallèle d'un solveur basé sur la FMM est bien plus performante que les résultats séquentiels du solveur $\mathcal{H}$-Matrice. Ces deux types de solveurs sont disponibles dans la suite logicielle ASERIS, et un solveur basé sur les $\mathcal{H}$-Matrices ne peut trouver sa place que s'il est parallèle. De plus, la taille des problèmes traités en électromagnétisme et acoustique est telle que le temps de calcul devient fortement pénalisant pour l'utilisateur final (fréquemment plus de 10^6 inconnues).

À ces deux raisons s'en ajoute une troisième, liée à l'évolution de l'informatique. Au milieu des années 60, Gordon Moore, co-fondateur d'Intel a énoncé une « loi » basée sur l'observation que le degré d'intégration des circuits électroniques doublait à peu près tous les deux ans. Cette croissance exponentielle s'est longtemps traduite par l'augmentation des performances des processeurs. Malheureusement, cette époque semble désormais révolue, avec en particulier la fin de la « course au GHz » ; le « *free lunch* » de l'augmentation exponentielle de la performance d'un programme séquentiel [93] est terminé, laissant place

à une augmentation du nombre de processeurs. Il est donc de plus en plus important d'exploiter le nombre grandissant d'unités de calcul disponibles sur les machines actuelles. Courant 2013, une station de travail usuelle peut être dotée de 16 processeurs, et un nœud de centre de calcul de 32, ce qui souligne l'importance de l'exploitation de ces ressources au travers d'un code parallèle. Ce code doit, de plus, être à même d'assurer un bon passage à l'échelle, c'est-à-dire une bonne décroissance du temps de calcul avec l'augmentation du nombre de processeurs.

4.1 Introduction

Ce chapitre présente la parallélisation des algorithmes du chapitre précédent dans le cadre d'une machine à mémoire partagée. Ce type de machine se caractérise par la présence de multiples unités de calcul partageant une vue unifiée de la mémoire. Chaque unité peut donc accéder à toutes les données simultanément, et moyennant le respect de certaines règles de synchronisation, peut opérer sur l'ensemble des données indépendamment des autres unités. Ceci est à opposer aux machines à mémoire distribuée, pour lesquelles il est nécessaire de communiquer explicitement les données, dont seule une partie est accessible par une unité de calcul donnée. Ceci complexifie l'implémentation, et sera l'objet du chapitre 5.

Mémoire partagée Cette classe de machine est très large : elle comprend en 2013 la quasi-totalité des ordinateurs portables, des stations de travail, des calculateurs et même une grande majorité des téléphones portables de type « *smartphone* ». Dans certains cas (et en particulier pour les calculateurs haut de gamme), ces machines font partie de la classe des architectures *Cache Coherent Non Uniform Memory Access* (ccNUMA), pour lesquelles l'accès à la mémoire a des caractéristiques de latence et bande passante différentes suivant la zone mémoire et l'unité de calcul considérées. Dans le cas où cette hétérogénéité est la plus marquée, il pourra être nécessaire de considérer artificiellement que ces machines sont à mémoire distribuée ; dans les autres nous négligerons cette spécificité.

Parallélisation La parallélisation d'un algorithme en mémoire partagée est avant tout guidée par les dépendances entre les étapes de calcul, et par la gestion des accès concurrents aux données. Il est nécessaire de minimiser ces dépendances, parfois au prix de l'augmentation du nombre total de calculs, et de planifier les opérations de façon à réduire les surcoûts liés à la gestion des accès concurrents aux données. Il est en particulier important de minimiser les étapes purement séquentielles du code, qui limitent sévèrement le passage à l'échelle, cette observation étant souvent connue sous le nom de « loi d'Amdahl » [17] (lemme 4.2).

4.1.1 Remarques préliminaires

Examinons brièvement les algorithmes à la lumière de ces observations :

— L'assemblage d'une $\mathcal{H}$-Matrice (compression des blocs) ne présente pas de dépendances de données. En effet, les blocs étant disjoints, les seuls accès concurrents sont ceux aux données relatives aux degrés de liberté. Ces accès ne sont que des lectures, ils n'engendrent donc pas de contraintes d'ordonnancement. Ce type de problème est qualifié de problème *embarassingly parallel* dans la littérature, et ne pose pas de difficultés dans sa parallélisation.

— Les algorithmes de H-BLAS1 et H-BLAS2 (décrits section 3.3.2) sont également simples à paralléliser. Ces algorithmes sont l'addition (H-AXPY) et le produit matrice-vecteur (H-GEMV).

— Les algorithmes de la seconde classe sont eux plus complexes, du fait de l'apparition de dépendances entre les données. C'est le cas du produit et de la résolution d'un système triangulaire.

— Les algorithmes de plus haut niveau reposent sur les précédents, et s'expriment comme une séquence d'opérations de plus bas niveau. Ces opérations sont complexes (car potentiellement du second type), et une contrainte supplémentaire s'ajoute. En effet, l'enchaînement de deux produits matriciels nécessite naïvement d'attendre la complétion du premier pour effectuer le second, ce qui introduit une séquentialisation dans l'algorithme, limitant ainsi le passage à l'échelle.

Addition L'addition est une tâche *embarassingly parallel*. Elle mène néanmoins au changement de taille des données, ce qui est lié à la recompression adaptative des $\mathcal{R}k$-Matrices.

Produit Matrice-Vecteur Soit A une $\mathcal{H}$-Matrice, x et y deux vecteurs de dimensions adaptées. L'opération $y \leftarrow Ax$ nécessite de réordonner x et y dans les deux sens avant et après le produit (*cf.* les remarques section 3.3.2.4), puis le produit se fait sur les feuilles de A, avec des accès en lecture à des parties de x et des accès en écriture à des parties de y. Les accès concurrents en écriture aux parties de y semblent problématiques, mais il est en réalité aisé de contourner cette difficulté, car l'opération est du point de vue de y une *réduction*.

Supposons que chaque processeur $i \in \{1, \dots, p\}$ effectue une partie des opérations sur les feuilles, et enregistre indépendamment les résultats dans y_i. Ceci correspond à l'opération $y_i \leftarrow \tilde{A}_i x$, avec $\tilde{A}_i$ des $\mathcal{H}$-Matrices dans lesquelles certaines des feuilles sont mises à 0, et telles que $A = \sum_i \tilde{A}_i$. On a alors $y = \sum_i y_i$, ce qui peut se faire de diverses manières (une des plus courantes étant d'utiliser un *arbre de réduction*), et permet de supprimer les dépendances sur les opérations $y_i \leftarrow \tilde{A}_i x$ qui sont désormais indépendantes. Le produit matrice-vecteur est donc une opération « simple » du point de vue de la parallélisation. Bien que les permutations sur x et y soient séquentielles, elles ne sont en pratique pas assez coûteuses pour entraver les performances parallèles.

Multiplication La multiplication $C \leftarrow A \times B$ est délicate du point de vue de la parallélisation, du fait de l'existence de nombreuses dépendances de données dans l'algorithme. En effet, un même sous-arbre de la matrice C peut être accédé en écriture par diverses opérations, et ce à différents niveaux de celui-ci (*cf.* figure 4.3). À ce problème s'ajoute

celui de la recompression adaptative des $\mathcal{R}k$-Matrices, ce qui nécessite des structures informatiques capables de gérer cette flexibilité. C'est la parallélisation de cette opération qui justifiera le recours à une parallélisation à base de tâches dans le reste de ce chapitre.

4.1.2 Cadre

Dans tout ce chapitre, nous présentons l'implémentation d'algorithmes parallèles pour les machines à mémoire partagée. Il convient donc de préciser ces définitions. De manière très générale, un ordinateur effectue des opérations dans le but d'agir sur des données, et un modèle presque universellement adopté depuis la fin des années 40 est le modèle de Von Neumann, du nom de son inventeur. Dans ce modèle, une machine est composée d'une unité de calcul effectuant des opérations dont le séquencement est décrit dans la mémoire du système, dans laquelle les données sont également conservées. Cette mémoire est de type *Random Access Memory* (RAM), c'est-à-dire qu'elle permet un accès en écriture ou en lecture à une partie de celle-ci sans que le temps d'accès ne dépende de la position de la donnée dans la mémoire. Les unités de calcul sont reliées à la mémoire par un « bus » assurant la communication entre le processeur et la mémoire.

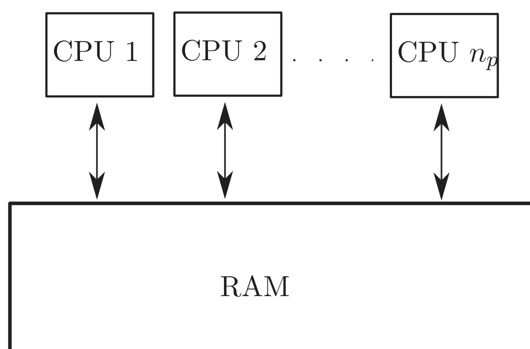


Figure 4.1 – Plusieurs processeurs partageant la même mémoire

Machines parallèles Une machine parallèle est constituée de plusieurs processeurs capables d'effectuer des calculs de manière indépendante. Ces processeurs peuvent être reliés à la même mémoire par le même bus ou un bus différent comme sur la figure 4.1. Il s'agit du modèle appelé PRAM [37]. Dans ce cas, tous les processeurs ont la même vue de la mémoire, et tout changement à une partie de la mémoire est immédiatement visible par tous les autres. Si les processeurs sont identiques et connectés au même bus mémoire, cette configuration est qualifiée de *Symmetric Multi-Processing* (SMP). Il est cependant nécessaire de synchroniser les opérations de façon à ce que les processeurs ne modifient pas en même temps la même zone mémoire, pour éviter les conflits.

Architectures non uniformes Cependant, la réalité est plus complexe. La figure 4.2 représente la topologie réelle d'une machine de type « station de travail ». On constate la présence de « mémoires caches » qui sont des sous-ensembles de la mémoire centrale dont la vue est privée à un ou plusieurs cœurs de calcul, et la mémoire centrale n'est plus

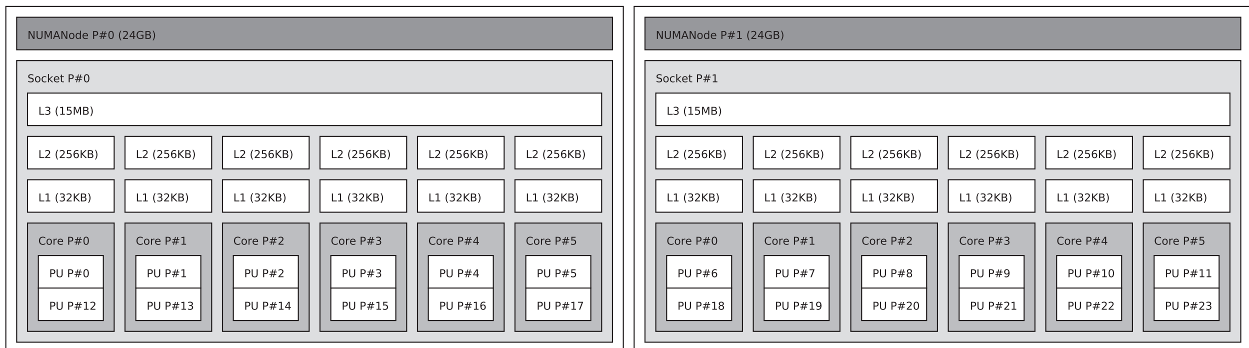


FIGURE 4.2 – Topologie d'une machine réelle. Les cœurs de calcul sont en bas (« Core #1 ») et la hiérarchie mémoire haut-dessus.

attachée à l'ensemble des processeurs, mais à une partie de ceux-ci, ces choix étant motivés par des questions de performance et consommation électrique. Dans ce cas, toute la mémoire centrale reste accessible par tous les processeurs, mais avec un coût supplémentaire, car il est nécessaire de franchir une interconnexion supplémentaire pour accéder à la mémoire distante. Néanmoins, la cohérence des différentes mémoires caches est assurée par un mécanisme matériel, dont il existe plusieurs variantes (MESI, MOSI, MOESI, *etc.*), codifiées et documentées en fonction de l'architecture. Ce type d'architecture est appelé *Cache Coherent Non Uniform Memory Access* (ccNUMA).

Exclusion mutuelle Il est donc nécessaire de s'assurer de la cohérence des accès en mémoire. À cette fin, une primitive de synchronisation universellement implémentable est appelée *mutex*, pour *Mutual Exclusion*. Intuitivement, il s'agit d'un jeton unique donnant accès à une ressource, et dont la possession peut se passer de manière atomique entre plusieurs processeurs. Ainsi, seul le détenteur de ce jeton peut accéder une zone mémoire en même temps, réglant ainsi le problème de la synchronisation. Il faut néanmoins préciser deux problèmes dans cette approche :

— Le temps d'acquisition d'un jeton est très long au regard des performances de calcul des processeurs actuels, limitant ainsi sévèrement les performances en cas d'utilisation intensive. Par ailleurs, le coût d'acquisition est croissant avec le nombre de processeurs.

— Seul le jeton est protégé contre la possession concurrente, pas la mémoire qu'il représente, et il appartient au programmeur de suivre les conventions d'usage, et de s'assurer de la correction de son programme, ce qui est souvent délicat.

Processus et Threads Sur la majorité des machines à mémoire partagée, le partage de la mémoire entre différents programmes s'exécutant sur la machine est optionnel, le système d'exploitation pouvant isoler les divers programmes. Le modèle de programmation est alors dit à base de *processus* et non de *threads*, pour lesquels la mémoire est partagée. La suite de ce chapitre traitera donc du cas de la programmation parallèle dans le contexte des *threads*.

Caches et localité des données La vue simpliste de la figure 4.1 représentant un processeur connecté à sa mémoire n'est pas actuelle. En effet, comme visible sur la figure 4.2, de nombreux niveaux de *cache* sont présents entre le processeur et la mémoire centrale. Ces caches sont de petits sous-ensembles de la mémoire dont le temps d'accès est plus court et le débit plus élevé que les niveaux supérieurs. Fin 2012, un accès à la mémoire centrale peut « coûter » 400 cycles de calcul, contre seulement 4 pour un accès à la mémoire cache L1. Il est donc important d'optimiser l'utilisation du cache en réutilisant autant que possible des données dont la taille est assez faible pour le cache. Par ailleurs, les processeurs modernes (depuis le Pentium Pro d'Intel en 1996) sont dotés d'unités de prédiction de branchement, d'unités de calcul vectoriel, de traitement en étapes des instructions, *etc.* Tous ces éléments nécessitent de prendre un soin particulier à la forme des données pour maximiser l'utilisation des ressources, ceci étant encore plus vrai pour les architectures de calcul reposant sur des cartes graphiques.

4.1.3 État de l'Art et contributions

4.1.3.1 État de l'Art

La littérature relative à la parallélisation des algorithmes reliés aux $\mathcal{H}$-Matrices est rare [23, 58, 69, 70], ces références partageant par ailleurs le même auteur. Dans cette littérature, les références [23,69,70] concernent la parallélisation en mémoire partagée des algorithmes opérant sur les $\mathcal{H}$-Matrices, et [58] traite le cas du préconditionnement d'un système linéaire creux par une décomposition LU d'une $\mathcal{H}$-Matrice en mémoire distribuée.

Les techniques employées dans ces articles sont similaires et reposent sur les mêmes principes et la même implémentation [3,4]. Diverses méthodes de parallélisation sont discutées et comparées dans ces références, celles-ci s'articulant majoritairement (et en pratique) autour de méthodes d'ordonnancement par liste. Une liste de tâches à effectuer est constituée, et ces dernières sont distribuées statiquement ou dynamiquement à un ensemble de processeurs. Le calcul comporte ainsi deux étapes : la construction de la liste de tâches et son exécution. Les deux approches les plus simples consistent alors à associer à chaque processeur un ensemble de tâches a priori, ou à exécuter chaque tâche sur le premier processeur disponible. Soit $L = \{t_1, \dots, t_{n_t}\}$ la liste de tâches et p le nombre de processeurs. Une fois la liste L construite, chaque processeur exécute la boucle suivante :

Algorithme 4.1 Boucle d'exécution de tâches par un processeur.

```
while L ≠ ∅ do
    L ← L\{t}
    EXECUTE(t)
end while
```

La gestion de la liste L est rendue plus délicate du fait des accès concurrents à celle-ci et de sa modification qui doit être atomique (indivisible du point de vue de tous les processeurs), mais des implémentations offrant cette garantie sont disponibles.

Cette approche très générale a déjà été appliquée à de nombreuses situations (y compris pour la FMM [94]), et est spécifiquement adaptée aux problèmes pour lesquels les

sous-tâches n'ont pas de dépendances complexes entre elles, et dans les cas où le temps d'exécution d'une tâche n'est pas connu à l'avance.

Néanmoins, la gestion de dépendances complexes dans ce type de formulation algorithmique mène à des difficultés. Dans ce cas, l'étape de sélection de la tâche à effectuer dans l'algorithme 4.1 ne se fait plus dans L, mais dans $L_{ready} \subset L$, qui est le sous-ensemble des tâches dont l'exécution est possible. La génération de cette liste est plus délicate et consommatrice de temps processeur, ce qui complexifie l'implémentation.

C'est le cadre adopté par la littérature. Dans le cas de l'assemblage et du produit matrice-vecteur, divers découpages statiques et dynamiques sont proposés dans le but d'optimiser l'ordonnancement pour l'assemblage et de réduire les conflits. L'énumération de ceux-ci est faite au début de l'exécution, et leur gestion est manuelle. En ce qui concerne la multiplication, les tâches sont regroupées de manière à minimiser les conflits, et des primitives de synchronisation appelées *mutex* sont employées pour gérer la liste des tâches prêtes.

Pour les opérations de plus haut niveau, une parallélisation de type *Bulk Synchronous Parallel* (BSP) [98], également appelé *Fork-Join* est proposée. Ceci signifie que l'exécution procède par étapes, chaque étape parallèle étant liée à la suivante par une synchronisation globale. Par exemple, l'algorithme 3.11 peut être instancié selon ce modèle en effectuant chacune des opérations le constituant (H-Gemm, Inverse) en parallèle, et en imposant une synchronisation entre chacune d'elles.

4.1.3.2 Contributions

Dans ce chapitre, nous présentons une parallélisation de tous les algorithmes présentés dans le chapitre 3 à l'aide d'un modèle unifié de programmation à base de tâches. Ce modèle, objet de récents développements, est basé sur l'énumération d'un graphe de tâches, les sommets représentant des tâches, et les arêtes les dépendances entre celles-ci. L'ordonnancement de ces tâches est purement dynamique, avec des indications de priorités optionnelles ajoutées, et avec un apprentissage « en ligne » des paramètres guidant les choix de l'ordonnanceur, suivant un modèle de performance écrit au préalable.

Nous donnons une description du graphe de tâches associé aux algorithmes présentés précédemment, ce qui nécessite formellement d'aplanir leur récursivité naturelle. Cette description, associée au formalisme de graphe de tâches a les caractéristiques suivantes :

- Indépendance du nombre de processeurs (et de l'architecture matérielle) ;
- Compatibilité avec une architecture hétérogène, constituée de processeurs « rapides » et « lents » ;
- Grande adaptabilité ;
- Absence totale de synchronisation globale, même pour les opérations de haut niveau.

Le dernier point est important, car toute synchronisation gêne fortement le passage à l'échelle quand le nombre de processeurs augmente, en rendant le temps d'exécution total de l'algorithme dépendant du maximum du temps de travail de chacun des processeurs. De plus, les graphes de tâches associés aux algorithmes numériques présentent la caractéristique qualitative commune de se « resserrer » autour du début et de la fin de l'exécution : le degré de parallélisme n'est pas constant, et entraîne une attente de certains processeurs.

Une approche de type BSP ne peut donc espérer atteindre une utilisation optimale des ressources. Cette limitation est repoussée dans le cadre de l'approche en tâches.

Les méthodes de parallélisation reposant sur le formalisme des graphes de tâches est un domaine de recherche actif, avec de nombreux travaux sur les ordonnanceurs, avec en particulier StarPU [19], QUARK [99] et DaGuE [26]. Ces méthodes ont récemment été utilisées pour l'algèbre linéaire dense [11–13, 15], et très récemment à la FMM pour le potentiel électrostatique [14]. Nous appliquons ici ces méthodes à une classe d'algorithmes plus complexes, irréguliers, et dont la parallélisation n'est pas répandue. Nous démontrons son intérêt par des gains en efficacité parallèle, sans que cela n'ait d'influence sur l'utilisation du solveur, dont le caractère parallèle reste totalement transparent à l'utilisateur, de même qu'une bibliothèque BLAS/LAPACK parallèle est transparente.

De plus, les méthodes employées pour formaliser les algorithmes récursifs opérant sur des arbres de tailles variables sous forme de graphes de tâches sont susceptible de présenter un intérêt pour d'autres classes d'algorithmes. Nous pouvons citer ici l'algèbre linéaire creuse, ainsi que la FMM pour l'équation de Helmholtz, moins régulière que la FMM pour le potentiel électrostatique.

Plan

Trois méthodes de parallélisation seront présentées dans ce chapitre :

1. La parallélisation utilisant des algorithmes de type BSP, tels que décrits dans la littérature.
2. Une reformulation de ces algorithmes dans le cadre de l'utilisation d'un *runtime* basé sur le concept de graphe de tâches.
3. Une parallélisation des algorithmes « nativement » basée sur un graphe de tâche au-dessus d'un *runtime*.

Les deux dernières méthodes sont présentées dans le cadre d'une implémentation originale, la première ne faisant pas l'objet de tests de performance du fait de l'absence d'implémentation disponible. Néanmoins, la seconde méthode exhibe les mêmes caractéristiques que la première, et l'utilisation d'un *runtime* peut dans ce cas être vue comme un remplacement de la bibliothèque de gestion des *threads* et des *mutexes* utilisés dans la littérature.

Après un ensemble de définitions concernant la notion d'efficacité parallèle (section 4.2), la description de la première méthode (ordonnancement par listes et algorithmes BSP, section 4.3) est l'occasion de préciser les particularités et difficultés inhérentes aux algorithmes opérant sur les $\mathcal{H}$-Matrices, et de souligner les points importants traités dans les sections suivantes.

Suit une introduction au concept de graphe de tâches et aux moteurs d'exécution, section 4.4. Leur utilisation est illustrée par la seconde méthode (algorithmes BSP et graphes de tâches, section 4.5), donnant une illustration pratique de ce formalisme. Cette implémentation est également une présentation des particularités des $\mathcal{H}$-Matrices vis-à-vis de la gestion des dépendances, et met en évidence les difficultés qui y sont reliées. Il s'agit par ailleurs d'une approche initialement retenue pour l'expression parallèle des algorithmes du chapitre précédent.

Enfin, la section 4.6 présente une parallélisation plus étroitement reliée au concept de graphe de tâches. Celle-ci est en effet pleinement asynchrone, et s'affranchit des limitations du modèle BSP. La levée des contraintes permet alors d'avoir un *pipelining* complet des algorithmes, permettant en particulier un meilleur passage à l'échelle.

Les deux implémentations réalisées sont comparées sur des cas réalistes sections 4.5.4 et 4.7.2.

4.2 Efficacité parallèle

Ce chapitre étant consacré à la parallélisation des algorithmes opérant sur les $\mathcal{H}$-Matrices, la question de l'efficacité de la parallélisation est centrale. Elle mesure la décroissance du temps de calcul avec le nombre de processeurs participant au calcul. Elle est définie par [37] :

Définition 4.1 (Efficacité parallèle). *Soit p le nombre de processeurs identiques d'une machine, et $t(p)$ le temps d'exécution d'un algorithme utilisant p processeurs. L'efficacité parallèle $E(p)$ est :*

$$E(p) := \frac{t(1)}{pt(p)}$$

La valeur de l'efficacité est en générale inférieure à 1, en particulier dans les cas où le calcul sur divers processeurs nécessite d'effectuer plus d'opérations que le calcul sur un seul processeur. Cette valeur décroît de plus très rapidement avec le nombre de processeurs dans le cas où l'intégralité d'un calcul ne peut être parallélisé. Ceci est connu sous le nom de loi d'Amdahl :

Lemme 4.2 (Loi d'Amdahl, [17]). *Soit un algorithme exécuté par p processeurs identiques. On suppose qu'une fraction $s \leqslant 1$ du travail effectué par l'algorithme est séquentielle. Alors l'accélération du temps de calcul sur p processeurs par rapport à une exécution séquentielle est bornée par :*

$$Acc(p) := \frac{t(1)}{t(p)} = \frac{1}{s + \frac{1-s}{p}}$$

et sa limite pour $p \to +\infty$ est :

$$\lim_{p \to +\infty} Acc(p) = \frac{1}{s}$$

L'efficacité parallèle $E(p)$ est bornée par :

$$E(p) \leqslant \frac{1}{1 + s(p-1)}$$

La loi d'Amdahl signifie que tout algorithme possédant une partie séquentielle verra son passage à l'échelle sévèrement limité, et ce même dans le cas où cette partie ne représente qu'une petite portion du temps d'exécution, du fait de la borne sur l'efficacité parallèle.

Dans certaines situations, le calcul de l'efficacité parallèle n'est cependant pas suffisant pour permettre une évaluation fine du comportement parallèle d'un algorithme. En effet, une perte de passage à l'échelle peut être liée à divers facteurs :

1. Des conflits d'accès à des ressources partagées sur la machine de calcul, comme la mémoire cache ou centrale ;
2. Le surcoût ajouté par le système de gestion du parallélisme ;
3. Le manque de parallélisme de l'algorithme employé.

Le temps d'exécution $t(p)$ est réparti pour chaque processeur entre le temps de calcul $t_c(p)$, le temps nécessaire à la gestion du parallélisme $t_s(p)$ et le temps d'attente $t_i(p)$. Ces trois temps sont sommés sur tous les processeurs du calcul, contrairement à la définition de $t(p)$. On a de plus $t_c(1) = t(1)$ dans le cas où la gestion du parallélisme n'a pas de coût fixe.

On a la relation :

$$p.t(p) = t_c(p) + t_s(p) + t_i(p)$$

Et on définit les efficacités $e_l(p)$, $e_s(p)$ et $e_p(p)$ par :

$$\begin{aligned} E(p) := \frac{t(1)}{p.t(p)} &= \frac{t_c(1)}{t_c(p) + t_s(p) + t_i(p)} \\ &= \underbrace{\frac{t_c(1)}{t_c(p)}}_{e_l(p)} \cdot \underbrace{\frac{t_c(p)}{t_c(p) + t_s(p)}}_{e_s(p)} \cdot \underbrace{\frac{t_c(p) + t_s(p)}{t_c(p) + t_s(p) + t_i(p)}}_{e_p(p)} \end{aligned} \tag{4.2.1}$$

Examinons ces trois termes dans le cadre de la décomposition d'un algorithme en tâches avec l'utilisation d'un moteur d'exécution :

Le premier terme correspond à la perte d'efficacité lié à l'exécution de plusieurs tâches simultanément sur le système. Ces pertes d'efficacité peuvent être liées à la perte de localité pour les machines ccNUMA, à l'encombrement de la mémoire cache des processeurs (les niveaux supérieurs étant partagés, *cf.* figure 4.2), ou à l'encombrement du bus d'accès mémoire. L'optimisation de cette efficacité dépend du programmeur dans le cas d'un placement statique des différentes parties du calcul sur les processeurs, ou du moteur d'exécution dans la cas d'un placement dynamique. Il est en effet possible de prendre en compte la localité des données lors de l'ordonnancement, en exécutant préférentiellement les tâches dont les données sont locales au nœud ccNUMA du processeur courant. Dans le cas contraire, il est possible de copier les données afin de les rapprocher des unités d'exécution. Cependant, une efficacité inférieure à 1 est à attendre des machines ccNUMA.

Le second terme correspond au coût d'ordonnancement des tâches. La résolution des contraintes d'ordonnancement nécessite en effet des calculs plus ou moins lourds selon l'algorithme utilisé. Par ailleurs, des verrous de type « *mutex* » sont nécessaires pour leur gestion, et les coûts d'acquisition et de gestion de ceux-ci augmentent avec le nombre de processeurs. L'optimisation de ce terme est la responsabilité du moteur d'exécution (ou du programmeur dans le cas d'un placement statique) et du programmeur. Le moteur d'exécution peut fournir des méthodes optimisées de coût moindre pour augmenter cette efficacité. Le programmeur peut chercher à réduire le nombre de dépendances et les risques de compétition pour les mêmes données.

Le troisième terme est une métrique de la qualité de l'algorithme parallèle. Il est contrôlé par le programmeur, puisque les dépendances sont spécifiées de façon exhaustive

par ce dernier, sans modification possible par le moteur d'exécution. Cependant, le choix d'un algorithme d'ordonnancement peut influencer cette valeur, comme ceci sera montré ultérieurement.

4.3 Ordonnancement par liste et algorithmes BSP

Dans cette section, nous décrivons une méthode de parallélisation, qui est employée dans la littérature [23,69,70] dans le cadre des $\mathcal{H}$-Matrices. Nous soulignons également au fil de cette présentation les particularités des algorithmes importantes pour la parallélisation. L'objectif de cette section est donc double : d'une part, présenter les enjeux et spécificités de la parallélisation des algorithmes opérant sur les $\mathcal{H}$-Matrices ; de l'autre, décrire le mode de parallélisation utilisé dans la littérature.

4.3.1 Assemblage

Comme cela a été mentionné dans l'introduction de ce chapitre, section 4.1.1, l'assemblage d'une $\mathcal{H}$-Matrice est une opération dont la parallélisation est simple, du fait du grand nombre de feuilles de l'arbre de blocs. Donnons un exemple illustratif. Nous considérons un problème électromagnétique issu de la discrétisation d'un A319, à 290.030 inconnues. Avec $n_{leaf} = 100$, $\eta = 3$ et un découpage médian, l'arbre de groupes de ce problème comporte 8191 nœuds [1], et l'arbre de blocs 185.093 dont 138.820 feuilles, ce nombre étant croissant avec le nombre de degrés de liberté, et lorsque n_{leaf} ou η décroît.

Il est donc possible de découper l'assemblage en autant d'étapes qu'il y a de feuilles dans l'arbre de blocs, et ces tâches n'ont pas de dépendance entre elles, ce qui les rend adaptées à l'ordonnancement par liste. Néanmoins, il faut noter que l'ordonnancement par liste n'est pas nécessairement optimal. En effet, considérons la fin du traitement des tâches, lorsque tous les processeurs n'ont plus qu'une tâche à effectuer (ce qui correspond au cas $L = \varnothing$ dans l'algorithme 4.1). Toutes ces tâches ne se finissent pas en même temps, ce qui entraîne une perte d'efficacité puisque des processeurs sont inactifs.

Le lemme suivant cité dans [69] et issu de [54] donne des bornes sur l'efficacité de l'ordonnancement par liste d'une série de tâches sans dépendances entre elles :

Lemme 4.3 (Efficacité parallèle de l'ordonnancement par liste)**.** *Soit un ensemble L de tâches sans dépendances entre elles et soit $t_{min}(L,p)$ le temps minimal pour exécuter ces tâches sur une machine comportant p processeurs identiques. Alors le temps d'exécution de ces tâches par ordonnancement par liste $t_{LS}(L,p)$ est tel que :*

$$t_{LS}(L,p) \leqslant \left(2 - \frac{1}{p}\right) t_{min}(L,p) \tag{4.3.1}$$

La borne supérieure de (4.3.1) est atteinte lorsque les tâches les plus coûteuses sont exécutées en dernier. Dans le cas où la liste L est exécutée dans l'ordre décroissant de temps d'exécution des tâches, le facteur multiplicatif de (4.3.1) peut être ramené à $\frac{4}{3} - \frac{1}{3p}$. On

1. L'arbre de groupes étant binaire et parfait dans ce cas, il comporte $2^{h+1} - 1$ nœuds pour une hauteur h.

parle alors d'ordonnancement *Longest Process Time* (LPT) [54]. Ceci n'est cependant pas possible dans le cas qui nous intéresse, puisque le temps d'assemblage d'un bloc, fortement dépendant de k (avec croissance en $\mathcal{O}(k^2)$ pour les algorithmes ACA avec pivotage partiel et ACA+), n'est pas connu à l'avance.

Une autre contrainte est relative au coût induit par l'ordonnanceur. Il s'agit du coût de construction de la liste L, du coût d'accès concurrent à cette liste, et du coût de lancement d'une tâche. Ces coûts étant proportionnels à la taille de L pour le premier et également au nombre de processeurs p pour les autres, il peut être avantageux de ne pas assembler les feuilles de la $\mathcal{H}$-Matrice une par une afin de les minimiser. Il faut donc trouver un équilibre entre le degré de parallélisme autorisé (qui est égal à $|L|$ ici) et la granularité des tâches pour maximiser le parallélisme en évitant des surcoûts (*overhead*) trop élevés.

Malgré l'inégalité (4.3.1) et le surcoût lié à l'ordonnancement, il est aisé d'obtenir une efficacité parallèle très élevée, supérieure à 99% dans [69] pour une machine comportant 16 processeurs.

Remarque 4.4 (Addition)**.** *Le cas de l'addition est similaire à celui de l'assemblage. Néanmoins, lors de l'addition de deux $\mathcal{H}$-Matrices, le rang de toutes les feuilles admissibles est connu à l'avance, si les matrices sont assemblées complètement avant l'exécution. Il est donc possible de grouper les tâches de manière à diminuer le surcoût lié à l'ordonnancement à l'avance, et il est également possible d'appliquer un ordonnancement de type LPT.*

Remarque 4.5 (Stockage)**.** *Il est usuel d'optimiser l'occupation mémoire, l'utilisation du cache et de faciliter la prédiction de branchement dans les algorithmes numériques en représentant les structures de données de manière compacte en mémoire. Ceci est possible pour les divers arbres utilisés, mais ne l'est pas pour les données.*

Il n'est pas possible de préallouer les structures de données avant l'assemblage, puisque le nombre de colonnes k des feuilles admissibles n'est pas connu à l'avance. De même, il n'est pas possible de « compacter » la représentation de la matrice une fois assemblée, du fait des recompressions adaptatives lors de l'addition, et des allocations mémoires et copies ne seront pas évitées dans les algorithmes.

4.3.2 Produit Matrice-Vecteur

Considérons le cas du produit matrice-vecteur sous la forme de GEMV dans BLAS, c'est-à-dire :

$$y \leftarrow \alpha M x + \beta y$$

avec M une $\mathcal{H}$-Matrice.

Cette opération ne procède à des calculs que sur les feuilles de l'arbre de blocs, comme décrit section 3.3.2.4. Elle bénéficie donc a priori des mêmes qualités que l'assemblage discuté à la section précédente. Cependant, de nombreuses opérations sur les feuilles de M concernent les mêmes éléments de y, et il convient de protéger ces accès. Il est possible de reprendre le même algorithme que pour l'assemblage, en utilisant des *mutexes* pour protéger les accès aux parties de y. Ceci est toutefois délicat, car un accès à une partie de y peut être concurrent à un accès simultané d'un sous ou sur-ensemble de cette partie par une autre tâche. Ceci est fonction du niveau dans l'arbre de groupes ligne concerné par la feuille de l'arbre de blocs en cours de traitement.

Il faut alors soit employer un *mutex* par feuille de l'arbre de groupes associé à y, ce qui nécessite de l'ordre de $|I|/n_{leaf}$ *mutexes* (8191 dans l'exemple de la section précédente) et entraîne un surcoût important à l'exécution, soit faire un verrouillage plus grossier, ce qui réduit le parallélisme.

Alternativement, il est possible d'associer à chaque processeur un vecteur y_i, $i \in \{1, \ldots, p\}$ initialisé à 0, et d'effectuer sur chaque processeur une partie de l'opération $y_i \leftarrow Ax$, avec un ordonnancement par liste. Il est ensuite nécessaire de reconstituer $y \in \mathbb{C}^M$ par l'opération :

$$y \leftarrow \sum_{i=1}^{p} y_i + \beta y$$

L'opération de sommation peut être optimisée en ne faisant la somme que sur les parties de y_i non nulles, ramenant le coût de cette opération de $\mathcal{O}(pM)$ à $\mathcal{O}(d_{sh}M)$, avec d_{sh} le degré de partage de l'algorithme employé, avec d_{sh} défini par :

Définition 4.6. *Degré de partage Soit $\mathcal{L}_p \subset \mathcal{L}_{I \times J}$ le sous-ensemble des feuilles de $\mathcal{T}_{I \times J}$ affecté au processeur p. Pour un élément $i \in I$, on définit le degré de partage $d_{sh}(i)$ par :*

$$d_{sh}(i) := \max\{d^r_{sh}(i), d^c_{sh}(i)\}$$

avec les degrés de partage lignes et colonnes définis par, respectivement

$$d^{r;c}_{sh}(i) := |\{p \mid \exists(\sigma, \tau) \in \mathcal{L}_p,\ i \in \sigma, \tau\}|$$

Il est montré dans [69] que le degré de partage d_{sh} est proche de p pour des valeurs raisonnables de p, ce qui diminue l'efficacité parallèle de l'algorithme. Cet article utilise alors un regroupement des feuilles de l'arbre par courbe remplissant l'espace, avec une courbe de Hilbert dans ce cas pour minimiser ce degré ; l'ordonnancement n'est alors plus dynamique, mais statique. Il est en effet possible de connaître le coût des opérations à l'avance, puisque la $\mathcal{H}$-Matrice est dans cette approche assemblée avant le produit matrice-vecteur, et les rangs numériques des blocs admissibles sont connus. Il ne reste plus qu'à utiliser le comptage des opérations de la section 2.4.2.1 page 57 pour connaître le coût des opérations, et faire un équilibrage de charge statique. Dans [69], l'efficacité parallèle de cet algorithme atteint alors jusqu'à 88% sur 16 processeurs.

Remarque 4.7 (Arbre de réduction)**.** *Il est probable qu'un ordonnancement par liste couplé à une réduction utilisant un arbre permettrait d'obtenir une bonne efficacité parallèle sur une machine à mémoire partagée, en parallélisant l'étape de réduction. Ceci n'est pas mentionné dans [69] et n'a pas été testé, mais est similaire à ce qui est couramment implémenté pour le produit matrice-vecteur dense et creux.*

4.3.3 Produit matriciel

De même que la section précédente traitait le cas de l'opération H-Gemv, cette section s'intéresse à l'implémentation parallèle de l'opération H-Gemm :

$$C \leftarrow \alpha A \times B + \beta C \tag{4.3.2}$$

Cette opération, dont l'implémentation est exposée section 3.3.3.1 est plus délicate dans sa version parallèle du fait des dépendances entre ses différentes sous-tâches. En effet, considérons l'algorithme 3.9 page 97, et plus précisément ses cas de base, pour lesquels au moins A, B ou C est une feuille. Un cas de base mène à la modification d'une partie de C, $C|_{\sigma\times\tau}$. Cependant, il n'est pas garanti que cette partie de C soit une feuille, ni que tous les termes de produit contribuant à la mise à jour $C|_{\sigma\times\tau}$ le soient sur ce cas de base. Autrement dit, il est possible que des écritures se fassent à des niveaux différents du même sous-arbre de C. Cette situation est illustrée par la figure 4.3, dans le cas où les structures des trois $\mathcal{H}$-Matrices A, B et C sont identiques. Sur cette figure, la partie grisée de C

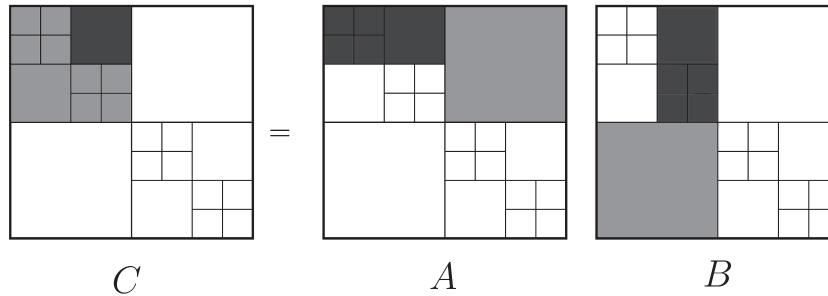


Figure 4.3 – Exemple d'accès concurrents à un sous-arbre de C. La partie sombre de C est mise à jour par les produits sombres, et son sur-ensemble clair est mis à jour par le produit des parties claires de A et B.

est mise à jour par les produits gris sombres ainsi que les produits gris clairs, lesquels mettent à jour le sur-ensemble clair (un nœud parent) du nœud sombre de C. Ces deux opérations n'ont pas de relation d'ordre dans leur exécution du fait de la commutativité et de l'associativité de l'addition, mais ne peuvent s'exécuter en même temps. On parle alors *d'exclusion mutuelle* pour les deux tâches, ce qui est précisément le mécanisme fourni par les conventions d'utilisation des *mutexes*.

Par ailleurs, il est également possible que plusieurs tâches de mise à jour d'un nœud $C|_{\sigma\times\tau}$ soient requises. En effet, considérons :

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \leftarrow \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \times \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} + \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Les termes mettant à jour C_{11} sont :

$$C_{11} \leftarrow A_{11} \times B_{11} + C_{11} \quad \text{et} \quad C_{11} \leftarrow A_{12} \times B_{21} + C_{11}$$

Il y a également une relation d'exclusion mutuelle entre ces tâches, mais il est aisément possible d'éviter le coût des *mutexes* dans ce cas. La solution retenue dans [69] est de regrouper les opérations suivant leur sous-arbre de destination $C|_{\sigma\times\tau}$, et d'exécuter ces tâches sur un même processeur. De cette manière, les accès concurrents au même nœud n'ont pas besoin d'être protégés, mais le problème représenté par la figure 4.3 existe toujours, et est traité par l'utilisation d'un *mutex*.

L'ordonnancement dynamique par liste est adapté pour cette opération, du fait de la difficulté de prévoir le coût d'une opération. La multiplication parallèle se décompose alors en deux étapes : la construction de la liste de tâches et son exécution. Pour tout nœud $\sigma \times \tau$

de C, on note $\mathcal{P}_{\sigma\times\tau}$ l'ensemble des cas de base $\{(A_{\sigma\times\rho}, B_{\rho\times\tau})\}_\rho$ de la multiplication dont la destination est $C|_{\sigma\times\tau}$. On note également L_{GEMM} la liste de tâches. La construction de la liste est réalisée par l'algorithme 4.2, avec les mêmes notations que celles de l'algorithme 3.9. L'algorithme 4.3 décrit l'exécution de cette liste L_{GEMM}. Dans cet algorithme,

Algorithme 4.2 Création de la liste de tâches pour le produit.

```
function CreateGemmList(C, A, B, σ ∈ T_I, τ ∈ T_J, ρ ∈ T_K)
    if ¬IsLeaf(σ × τ) et ¬IsLeaf(σ × ρ) et ¬IsLeaf(ρ × τ) then
        for all σ' ∈ S(σ) do
            for all τ' ∈ S(τ) do
                for all ρ' ∈ S(ρ) do
                    CreateGemmList(C, A, B, σ', τ', ρ')
                end for
            end for
        end for
    else
        P_{σ×τ} ← P_{σ×τ} ∪ {(A|_{σ×ρ}, B|_{ρ×τ})}
        L_GEMM ← L_GEMM ∪ {C|_{σ×τ}}
    end if
end function
```

Algorithme 4.3 Exécution de la liste de tâches.

```
function GemmTask(C|_{σ×τ}, P_{σ×τ}, α ∈ ℂ)
    for all (A_{σ×ρ}, B_{ρ×τ} ∈ P_{σ×τ} do
        C|_{σ×τ} ← α A_{σ×ρ} B_{ρ×τ} + C|_{σ×τ}
    end for
end function
function ParallelGemm(A, B, C, σ ∈ T_I, τ ∈ T_J, ρ ∈ T_K, α, β)
    L_GEMM ← ∅    P ← ∅
    CreateGemmList(C, A, B, I, J, K)
    C ← βC                                          ▷ Parallèle
    for all C|_{σ×τ} ∈ L_GEMM do
        GemmTask(C_{σ×τ}, P_{σ×τ}, α)               ▷ Parallèle
    end for
end function
```

les deux étapes marquées « Parallèle » sont exécutées à l'aide d'un ordonnancement par liste, chaque processeur effectuant une boucle similaire à celle de l'algorithme 4.1, les accès concurrents aux sous-arbres de C étant protégés par des *mutexes*.

Les facteurs limitant théoriquement le passage à l'échelle de cet algorithme sont liés à la construction de la liste, aux limites de l'ordonnancement par liste, et à l'utilisation de *mutexes* pour empêcher les accès concurrents. La construction de la liste L_{GEMM} et des ensembles $\mathcal{P}_{\sigma\times\tau}$ est en pratique très peu coûteuse ; elle a une complexité temporelle de $\mathcal{O}(C_{sp}(\mathcal{T}_{I\times I})|\mathcal{T}_{I\times I}|))$ dans le cas où $I = J = K$. Ceci, couplé à l'efficacité pratique de l'ordonnancement par liste et à la relative rareté des mises à jour concurrentes, mène à une bonne efficacité parallèle, supérieure à 80% dans [69] sur 16 processeurs.

4.3.4 Inversion

Dans cette section, nous présentons l'inversion sur place d'une $\mathcal{H}$-Matrice, utilisant l'algorithme 3.11. Cet algorithme comprend deux appels récursifs à la fonction d'inversion, et six appels à GEMM. Il fait appel à une fonction dont la parallélisation a été vue dans la section précédente. L'écriture parallèle de cet algorithme peut alors se comprendre comme la composition de plusieurs opérations elles-mêmes parallèles, mais ayant des dépendances à satisfaire. Dans cette optique, la méthode la plus simple consiste à suivre un modèle BSP.

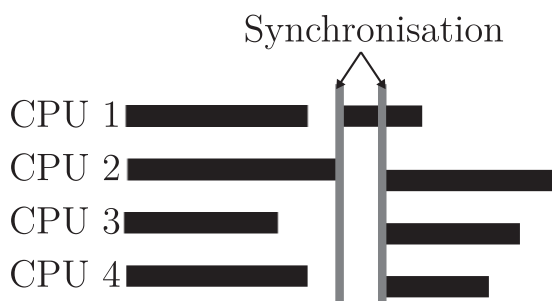


Figure 4.4 – Illustration d'un algorithme parallèle « Fork-Join ». L'axe horizontal représente le temps, les lignes noires l'exécution de tâches.

Dans ce modèle, des sous-ensembles d'un calcul sont effectués en parallèle, et reliés par une synchronisation globale (également appelée barrière), laquelle peut s'accompagner ou non d'échanges de données. Cette méthode de parallélisation est courante, de par sa simplicité d'implémentation, et supportée par un grand nombre d'outils (*Message Passing Interface* (MPI) avec `MPI_Barrier`, OpenMP avec les régions parallèles, Intel Threading Building Blocks, Apple Grand Central Dispatch, *etc.*). Elle n'est cependant pas optimale. Nous renvoyons le lecteur à [98] pour plus de précisions, en particulier sur les bornes de temps de calcul. Intuitivement, cette méthode présente les mêmes inconvénients que l'ordonnancement par liste, puisque le temps d'exécution de chaque section parallèle est limité par le temps d'exécution du processeur le plus lent, ou ayant le plus de travail à effectuer. La sévérité de cette perte de temps augmente avec le nombre de synchronisations, le coût d'une barrière n'étant de plus pas nul. Il faut aussi noter que dans certains cas, certaines étapes sont purement séquentielles, et entravent alors grandement le passage à l'échelle lorsque le nombre de processeurs grandit.

Ceci est illustré par la figure 4.4. Sur celle-ci, on considère l'exécution d'un algorithme comportant deux barrières, encadrant une partie séquentielle. Dans cet exemple, l'axe horizontal représente le temps, et un processeur occupé est représenté en noir.

L'algorithme 4.4 donne une parallélisation possible de l'inversion d'une $\mathcal{H}$-Matrice, proche de celle décrite dans [69]. Dans cet algorithme, les appels à GEMM sont remplacés par des appels à la fonction définie dans la section précédente et par l'algorithme 4.3. Les opérations GEMM apparaissant sur la même ligne sont indépendantes, et peuvent donc être effectuées en même temps, en ne faisant qu'une synchronisation globale au terme de ces deux produits. Ceci diminue le nombre de synchronisations globales, et permet donc une meilleure exploitation des ressources de calcul, cependant [69] ne mentionne pas ceci, et il est donc difficile de savoir si cette optimisation mineure a été utilisée dans cette référence.

Algorithme 4.4 Inversion d'une $\mathcal{H}$-Matrice, BSP. X est une $\mathcal{H}$-Matrice temporaire.

```
function ParallelInverse(M, X)
    if IsLeaf(M) then                                              ▷ Séquentiel
        M ← M^{-1}
    else
        M := ( M11  M12 ; M21  M22 ) et X := ( X11  X12 ; X21  X22 )
        ParallelInverse(M11, X11)
        ParallelGemm(X12, −1, M11, M12, 0) ‖ ParallelGemm(X21, 1, M21, M11, 0)
        ParallelGemm(M22, 1, M21, X12, 1)
        ParallelInverse(M22, X22)
        ParallelGemm(M12, 1, X12, M22, 0)
        ParallelGemm(M11, −1, M12, X21, 1) ‖ ParallelGemm(M21, −1, M22, X21, 0)
    end if
end function
```

Du fait des nombreuses synchronisations, et des étapes séquentielles (inversion des blocs pleins diagonaux), le passage à l'échelle de cet algorithme est logiquement moins bon que celui des algorithmes précédents, avec une efficacité parallèle d'environ 64% sur 16 processeurs dans [69], même pour des problèmes de grande taille. On note cependant que les opérations purement séquentielles sont peu nombreuses, car elles ne concernent que les blocs diagonaux pleins $M_{\sigma\times\sigma}$, avec $\sigma \in \mathcal{L}(\mathcal{T}_I)$ (une feuille de l'arbre de groupes).

4.3.5 Conclusions

La littérature citée dans cette section démontre la possibilité de paralléliser les algorithmes de tous les niveaux opérant sur les $\mathcal{H}$-Matrices. Malgré les difficultés inhérentes aux $\mathcal{H}$-Matrices (irrégularité des données, équilibrage de charge, récursivité, opérations à différents niveaux d'un même sous-arbre), des techniques simples permettent d'atteindre de bons résultats sur une machine à mémoire partagée, particulièrement pour les opérations de type BLAS 1 ou 2.

Pour le produit matriciel, les résultats sont globalement bons, mais la présence de *mutexes* complexifie l'implémentation, tant dans la création et le maintien de ceux-ci que dans l'algorithme d'ordonnancement. Celui-ci doit alors être modifié pour prendre en compte les relations d'exclusions mutuelles, au risque de créer des attentes actives sur un ou plusieurs processeurs.

Il faut noter que les résultats exposés ici seraient probablement moins bons dans le cadre de l'équation des ondes, du fait de la plus grande irrégularité des données (taux de compression plus variables), et nous nous efforcerons de donner un aperçu de ceci lors de la comparaison avec l'implémentation proposée dans ce document, section 4.5.4.

Enfin, l'opération de plus haut niveau qu'est l'inversion présente des caractéristiques parallèles médiocres dans cette implémentation, et un comportement similaire serait observé pour une factorisation LU. Ceci est lié au manque de composabilité des opérations dans cette implémentation. En effet, il n'est pas possible aves les mécanismes présentés

d'effectuer deux produits matriciels interdépendants sans recourir à une synchronisation globale. Nous montrerons dans la suite de ce document qu'il est possible de s'affranchir de ces synchronisations avec un modèle de programmation en graphe de tâches, et que ceci permet de gagner en efficacité parallèle.

4.4 Graphe de tâches et moteur d'exécution

4.4.1 Graphe de tâches

Comme évoqué dans l'introduction de ce chapitre, un aspect important de la parallélisation d'un algorithme de calcul est la gestion des dépendances entre les données. De façon générale, un algorithme est une suite d'opérations effectuées sur des données, et le couple des opérations et des données définit des contraintes sur l'ordre d'exécution des opérations. Le problème de la parallélisation est alors de trouver un ordre d'exécution le plus parallèle possible respectant ces contraintes, ou de réécrire l'algorithme dans le but de minimiser les contraintes.

Ainsi, un problème ne comportant pas de contraintes est trivial à paralléliser, par exemple à l'aide d'ordonnancement par liste. En revanche, un problème fortement contraint (par exemple une suite d'opérations non commutatives et indivisibles sur une même donnée) ne se prête pas à une exécution parallèle.

Il est donc naturel de considérer les dépendances entre les données dans les différentes étapes d'un algorithme, et de les exprimer dans un cadre permettant une exécution satisfaisant ces contraintes. Un exemple de formalisme adapté à une telle expression est celui du « flot de données » (*Data Flow*). Nous retenons ici le modèle d'exécution séquentiellement consistant [72], c'est-à-dire que toute exécution valide du programme doit produire le même résultat que l'exécution séquentielle de celui-ci dans l'ordre d'écriture des instructions.

Considérons un algorithme, constitué d'un ensemble fini ordonné de tâches $T = \{t_i \mid i = 1, \ldots, N\}$ opérant sur un ensemble de données $D = \{d_j \mid j = 1, \ldots, M\}$. Chaque tâche t_i nécessite la connaissance d'un sous-ensemble $In(t_i) \subset D$ et modifie ou produit un sous-ensemble $Out(t_i) \subset D$. Afin d'assurer la consistance séquentielle, les contraintes suivantes sont à respecter :

— Tout accès en lecture à une donnée d_j par une tâche t_i (*ie* $d_j \in In(t_i)$) nécessite que toutes les modifications de cette donnée par les tâches $\{t_{i'} | i' < i \wedge d_j \in Out(t_{i'})\}$ antérieures soient effectuées ;

— Tout accès en écriture à une donnée d_j par une tâche t_i (*ie* $d_j \in Out(t_i)$) nécessite que toutes les lectures et écritures de cette donnée par les tâches $\{t_{i'} | i' < i \wedge d_j \in In(t_{i'}) \cup Out(t_{i'})\}$ antérieures soient effectuées.

Ces deux contraintes permettent de définir un graphe, dont les sommets sont les tâches t_i et l'arête $t_{i'} \to t_i$ signifie « $t_{i'}$ précède t_i » :

Définition 4.8 (Graphe de tâches)**.** *Soit un problème composé d'un ensemble ordonné de tâches $T = \{t_i \mid i = 1, \ldots, N\}$, opérant sur un ensemble (non ordonné) de données $D = \{d_i \mid i = 1, \ldots, M\}$. Pour chaque tâche $t_i \in T$, on définit un ensemble de données*

d'entrée $In(t_i) \subset D$ et de sortie $Out(t_i) \subset D$, et on pose $In := \{In(t_i) | i = 1, \ldots, N\}$ et $Out := \{Out(t_i) | i = 1, \ldots, N\}$.
On appelle graphe de tâches associé au problème (T, D, In, Out) le graphe dirigé d'ensemble de sommets S et d'arêtes A tel que :

$$S := T$$

$$(i', i) \in A \iff \begin{array}{l} \exists d_j \in D,\ (d_j \in Out(t_{i'})) \wedge (d_j \in In(d_i)) \wedge (i' < i) \\ ou \\ \exists d_j \in D,\ (d_j \in In(t_{i'}) \cup Out(t_{i'})) \wedge (d_j \in Out(d_i)) \wedge (i' < i) \end{array} \tag{4.4.1}$$

Remarque 4.9.

— *Une donnée d_j modifiée par une tâche t_i est telle que $d_j \in In(t_i) \cap Out(t_i)$. Une donnée appartenant à l'ensemble des sorties et non à l'ensemble des entrées d'une tâche est soit créée par cette tâche, soit écrasée, puisque son contenu est indifférent à la tâche. Un exemple simple serait celui de l'assemblage d'une partie d'une matrice, représentée par la donnée.*

— *Un tel graphe de tâches possède bien souvent un grand nombre d'arêtes. Il est cependant possible de le simplifier, en utilisant la transitivité des contraintes induites.*

— *Un graphe de tâches correspondant à un programme séquentiel ne contient pas de cycle. Supposons l'existence d'un cycle $t_i \to t_{i_1} \to \cdots \to t_{i_p} \to t_i$ dans un tel graphe. Alors t_{i_p} précède t_i, et par induction t_i précède t_i, ce qui est impossible.*

La deuxième remarque est importante. Il est en effet aisé de montrer que la relation $i' \to i$ est transitive (et irréflexive). Il est donc possible de réduire l'ensemble des arêtes du graphe de tâches en conservant les mêmes contraintes, et c'est ce graphe que nous considérons dans la suite :

Définition 4.10 (Graphe de tâches réduit). *Soit (S, A) un graphe acyclique dirigé tel que défini par 4.8. On définit un graphe de tâches réduit (S', A') par $A' := A$ et*

$$A' := A \backslash \left\{(i', i) \in A \mid \exists \{i_1, \ldots, i_p\}, (i', i_1) \in A \wedge (i_1, i_2) \in A \wedge \cdots \wedge (i_p, i) \in A\right\}$$

Les arêtes retirées dans le graphe de tâche réduit représentent des contraintes redondantes, puisque déjà assurées par la transitivité de la relation $\to$. Une interprétation de ce graphe réduit revient à conserver une relation de dépendance sur la tâche qui a le plus récemment accédé à la donnée, selon les règles de la définition 4.8.

On peut alors définir la notion de parcours valide, c'est-à-dire une énumération des tâches respectant les contraintes d'un graphe de tâches réduit ou non :

Définition 4.11 (Parcours valide). *On appelle parcours valide d'un graphe de tâches $G = (S, A)$ toute énumération (ordonnée) des sommets de G telle que, pour tout $j \in S$, tous les sommets $i \in S$ tels que $(i, j) \in A$ ont été énumérés avant j. Ce parcours n'est en général pas unique.*

Il est aisé de montrer que les parcours valides sont identiques entre un graphe de tâches et son graphe réduit. Nous ne considérons dans la suite que des graphes de tâches réduits.

Example 4.4.1 (Multiples parcours valides). La figure 4.5 donne un exemple de graphe de tâches réduit. Il existe plusieurs parcours valides de ce graphe de tâches, comme $(1, 2, 3, 5, 4, 6)$ ou $(1, 3, 2, 4, 5, 6)$.

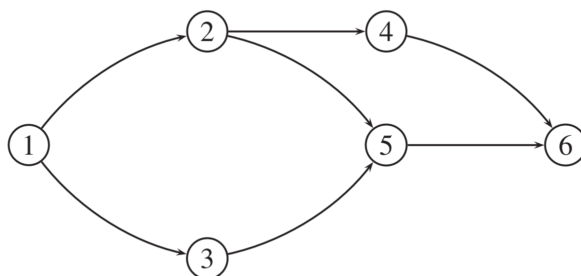


Figure 4.5 – Exemple de graphe de tâches

Exécution parallèle d'un graphe de tâches L'exécution parallèle des tâches du graphe de la figure 4.5 est claire : il est possible d'exécuter les tâches 2 et 3 en parallèle, mais il est également possible d'exécuter la tâche 4 en parallèle de la tâche 3, suivant les mêmes règles que celles définissant un parcours valide séquentiel. Une tâche t_i est considérée éligible pour exécution si toutes ses dépendances $\{t_{i'} \mid (i', i) \in A\}$ ont été exécutées. Dans ce cas, l'exécution parallèle du graphe de tâches respecte la contrainte de consistance séquentielle, et le résultat est le même quel que soit le nombre de processeurs. Une propriété supplémentaire intéressante est même obtenue : les résultats sont dits exactement reproductibles. La représentation des nombres en virgule flottante a brièvement été mentionnée section 3.3.2.2.3 page 92. Elle a pour conséquence de rendre les opérations arithmétiques usuelles non commutatives [50], et de mener en général à des résultats de calculs non reproductibles exactement pour deux exécutions parallèles. Or le graphe de tâches détermine de façon précise les dépendances entre les tâches, assurant que les tâches interdépendantes sont toujours exécutées dans le même ordre entre elles. Sous réserve que les deux machines adhèrent au même standard de représentation des nombres et offrent exactement les mêmes précisions sur les opérations élémentaires, le résultat sera alors rigoureusement identique. Ainsi, plusieurs exécutions sur une même machine, avec un nombre variable de cœurs de calculs exploités donnera le même résultat, alors même que l'ordonnancement des tâches est différent. Cette garantie n'est cependant plus valable dans le cas de l'utilisation d'extensions de ce formalisme, comme par exemple celle de la remarque 4.12, l'addition de la norme IEEE-754 n'étant pas associative.

4.4.2 Moteur d'exécution

Le formalisme de graphe de tâches exposé dans la section précédente a de nombreux avantages. Il est expressif, simple, et la formalisation des contraintes permet une grande liberté d'exécution parallèle. En effet, certaines méthodes de parallélisation sont centrées autour de l'expression des régions parallèles, et de l'identification des tâches pouvant être effectuées de façon concurrente. Le formalisme présenté ici est inversé, puisqu'il exprime les contraintes, et évite dans certains cas d'imposer des barrières artificielles liées à la programmation manuelle d'une exécution parallèle.

Cependant, le graphe de tâches ne permet pas de déterminer une exécution parallèle. Cette tâche est déléguée à un système opérant « en ligne » lors de l'exécution du programme. Il est chargé d'explorer le graphe de dépendances dans un ordre valide en parallèle, de manière à maximiser l'utilisation des ressources.

Il existe plusieurs projets utilisant le formalisme précédent pour exécuter des graphes de tâches, parmi lesquels QUARK [99] et StarPU [18, 19]. Le premier est lié au projet PLASMA [16], et a pour cible une architecture à mémoire partagée. StarPU vise les architectures hétérogènes, composées de processeurs et d'accélérateurs spécialisés. Ceux-ci ne partagent pas la même mémoire que les processeurs, et ne peuvent pas nécessairement exécuter les mêmes tâches que les processeurs, et le font avec un niveau de performance variable. StarPU supporte également la mémoire distribuée au travers de MPI, nous y reviendrons au chapitre suivant.

Pour ces deux ordonnanceurs, la spécification des tâches et de leurs dépendances se fait par une suite d'appels à une fonction équivalente à la fonction `insert_task`, suivant le modèle :

```
insert_task(gemm, C, IN | OUT, A, IN, B, IN, ...);
```

Cet appel ajoute une tâche à effectuer par la fonction `gemm`. Cette fonction accède à la variable `C` en lecture et écriture, et aux variables `A` et `B` en lecture. Cet appel ne fait qu'ajouter un sommet dans le graphe de tâches, et les dépendances entre les tâches sont déduites des dépendances sur les données `A`, `B` et `C`, et n'appelle pas la fonction `gemm`.

Exemple 4.4.2. Supposons trois matrices A, B, et C, décomposées en un tableau de $n \times n$ tuiles, `A[i][j]` représentant la tuile (i, j). Alors une écriture de la multiplication $C \leftarrow AB + C$ peut se donner dans ce formalisme par :

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    for (int k = 0; k < n; k++) {
      insert_task(gemm, C[i][j], IN | OUT, A[i][k], IN, B[k][j], IN);
    }
  }
}
wait_all_tasks();
```

L'insertion des tâches est non bloquante, ce qui rend nécessaire l'appel à la fonction `wait_all_tasks()` qui permet de s'assurer que toutes les opérations sont terminées avant de poursuivre le programme principal.

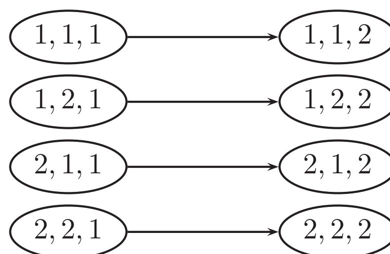


Figure 4.6 – Graphe de tâches induit pour un produit matriciel.

Dans le cas où $n = 2$ dans l'algorithme précédent, le graphe de tâches est donné par la figure 4.6, sur laquelle la tache (i, j, k) représente l'opération (séquentielle) $C_{ij} \leftarrow$

$C_{ij} + A_{ik}B_{kj}$. L'ordre d'exploration du graphe n'est pas spécifié, et 4 parties connexes donc totalement indépendantes du point de vue de l'exécution apparaissent.

Remarque 4.12 (Accumulation). *Dans l'exemple précédent, les tâches dans la boucle la plus interne sont commutatives. L'expression des dépendances formulée dans l'exemple précédent ne permet pas d'exposer cette caractéristique. Il est possible dans certains moteurs d'exécution (y compris QUARK et StarPU) de donner cette indication, sans laquelle les tâches de la boucle la plus interne ont des dépendances trop fortes (ordre strict au lieu d'exclusion mutuelle).*

Remarque 4.13 (Graphe équivalent). *Pour cela, il faudrait introduire la notion de graphe équivalent, lié au fait qu'en mémoire partagée, nous ne nous intéressons qu'aux dépendances et non aux données qui les portent.*

4.4.3 Présentation des moteurs d'exécution

4.4.3.1 QUARK

QUARK (QUeuing And Runtime for Kernels) [99] est un moteur d'exécution pour les architectures à mémoire partagée, développé dans le cadre du projet PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures) [15]. Ce projet a pour but de fournir une interface et des opérations similaires à BLAS et LAPACK, formulées à l'aide de graphes de tâches, et exécutées par un moteur pour permettre un ordonnancement dynamique. Le formalisme de graphe de tâches permet d'éviter les synchronisations globales, et l'ordonnancement dynamique de maximiser l'utilisation des ressources, y compris dans le cas de machines ccNUMA.

Afin de conserver une utilisation mémoire raisonnable et de borner le temps d'exploration du graphe de dépendances, QUARK traite ce graphe partiellement, en utilisant une fenêtre glissante sur celui-ci, de taille paramétrable. L'algorithme d'ordonnancement utilisé est de type « vol de tâches ». Chaque processeur possède une file d'attente de tâches prêtes à être exécutées. Lorsqu'une tâche est finie, le processeur exécute la première tâche de la file d'attente, en mode *First In, First Out* (FIFO). Si aucune tâche n'est disponible sur sa file, il « vole » une tâche à un autre processeur dont la file d'attente n'est pas vide, en mode *Last In, First Out* (LIFO).

Du fait de l'orientation « mémoire partagée » de QUARK, les données ne sont pas déclarées au moteur d'exécution, qui n'utilise que des pointeurs pour identifier une donnée, facilitant ainsi son utilisation pour des structures arborescentes comme les $\mathcal{H}$-Matrices.

4.4.3.2 StarPU

StarPU [18] (lire « $*PU$ », avec $* \in \{G, P, S, \ldots\}$) est un moteur d'exécution dont l'objectif est de permettre une exploitation efficace des machines hétérogènes. En effet, de plus en plus de machines contiennent diverses unités de calcul, dont la versatilité et les performances varient grandement. La méthode de programmation traditionnelle de ces machines se concentre sur une unité de calcul, au détriment des autres, avec par exemple l'utilisation de CUDA ou OpenCL pour les cartes graphiques. L'objectif de StarPU est

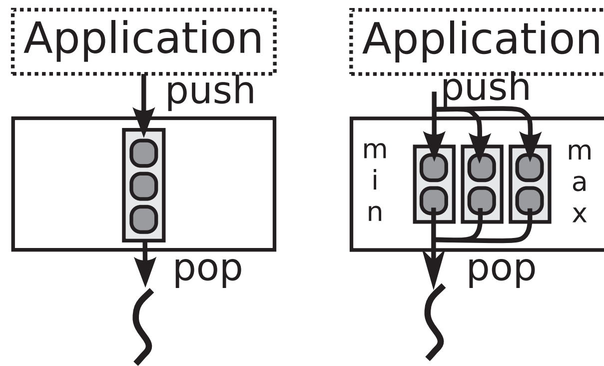


Figure 4.7 – Ordonnancement sans (`eager`) et avec priorités (`prio`).

de rendre transparente l'utilisation de toutes les ressources d'une machine, en affectant les tâches à une unité de calcul ou l'autre, de manière transparente pour le programmeur. Celui-ci a la charge de fournir pour chaque type de tâche une ou plusieurs implémentations, ciblant un ou plusieurs types d'unités de calcul.

Dans le cas des cartes graphiques, la mémoire n'est pas partagée avec la mémoire centrale du système, ce qui oblige StarPU à gérer les copies. De même, la notion de « type » de tâche nécessite de déclarer ces types au préalable. Il est donc nécessaire pour StarPU de déclarer un type de tâche, appelé `codelet` pour chaque opération élémentaire de la section précédente. Il est également nécessaire de permettre à StarPU de copier une $\mathcal{H}$-Matrice pour exploiter les cartes graphiques et la mémoire distribuée. Nous y reviendrons dans le chapitre suivant.

StarPU a une architecture modulaire, permettant de cibler aisément de nouvelles unités de calcul (courant 2013, les CPUs, CUDA, OpenCL et les accélérateurs SPUs du processeur Cell), et d'utiliser de nouveaux algorithmes d'ordonnancement. Il en existe plusieurs, dont :

— Un algorithme glouton simple, `eager`. Cet ordonnanceur utilise une unique file d'attente pour les tâches prêtes, que tous les processeurs utilisent pour exécuter les tâches en mode FIFO.

— Une variante de l'algorithme glouton prenant en compte les priorités des tâches, `prio`. Cet ordonnanceur utilise une file d'attente par valeur de priorité (dont le nombre est fini et fixe) commune à tous les processeurs, et chaque processeur exécute la première tâche disponible sur la file d'attente de priorité la plus élevée, en mode FIFO. Nous y reviendrons à la section 4.7.1.3.

— Un algorithme de type « vol de tâches », `ws`

— Divers algorithmes plus avancés, prenant en compte des *modèles de performance*. Ces algorithmes prennent tout leur sens dans le cas d'une architecture hétérogène, nous y reviendrons au chapitre suivant.

4.4.3.3 Points communs et différences

Les deux moteurs d'exécution présentent une interface de type « `insert_task` » similaire, et sont compatibles dans leur principe, ce qui permet à une implémentation de les

cibler sans modifications majeures. StarPU possède également une interface de programmation basée sur des annotations, similaires dans le principe aux annotations `#pragma omp` d'OpenMP. Ce support repose sur l'utilisation d'un greffon du compilateur GCC, et n'est pas utilisé ici pour des raisons de portabilité entre les moteurs d'exécution et les compilateurs.

Le domaine d'application de QUARK étant plus restreint, sa mise en œuvre est plus aisée du fait de l'absence de déclaration des données au moteur d'exécution et de l'absence du concept de `codelet`. Cependant, QUARK implémente également de ce fait des extensions au formalisme de graphe de tâches non supportées par StarPU, certaines n'étant pas compatibles avec une architecture hétérogène et/ou à mémoire distribuée. L'implémentation du solveur $\mathcal{H}$-Matrice de ce document s'appuie donc indifféremment sur les deux moteurs d'exécution, en utilisant les fonctionnalités qui leur sont communes.

4.5 Algorithmes BSP pour les $\mathcal{H}$-Matrices à l'aide de graphes de tâches

Dans cette section, nous présentons une écriture parallèle de l'algèbre des $\mathcal{H}$-Matrices suivant le formalisme présenté dans la section précédente. Le point de départ de cette écriture est celle de la littérature. Elle est exprimée à l'aide d'un graphe de tâches, au-dessus d'un moteur d'exécution dynamique. La section précédente fournit les outils permettant de donner une expression des algorithmes de la section 4.3 sous la forme de graphes de tâches utilisant la primitive `insert_task()`. Nous ne donnons ici que la présentation de la multiplication matricielle et de l'inversion, les autres opérations ne posant pas de difficultés particulières.

4.5.1 Multiplication

Examinons les deux étapes parallèles de l'algorithme 4.3 :

1. La mise à l'échelle de la matrice se fait pour une feuille avec comme seule dépendance en lecture/écriture celle-ci. Cette opération ne pose pas de problèmes, et est suivie d'une synchronisation globale (attente de toutes les tâches).
2. Une étape du produit (opération GEMMTASK) a les dépendances suivantes :
   — $C|_{\sigma\times\tau}$ en lecture/écriture ;
   — Les éléments de $\mathcal{P}_{\sigma\times\tau}$ en lecture.

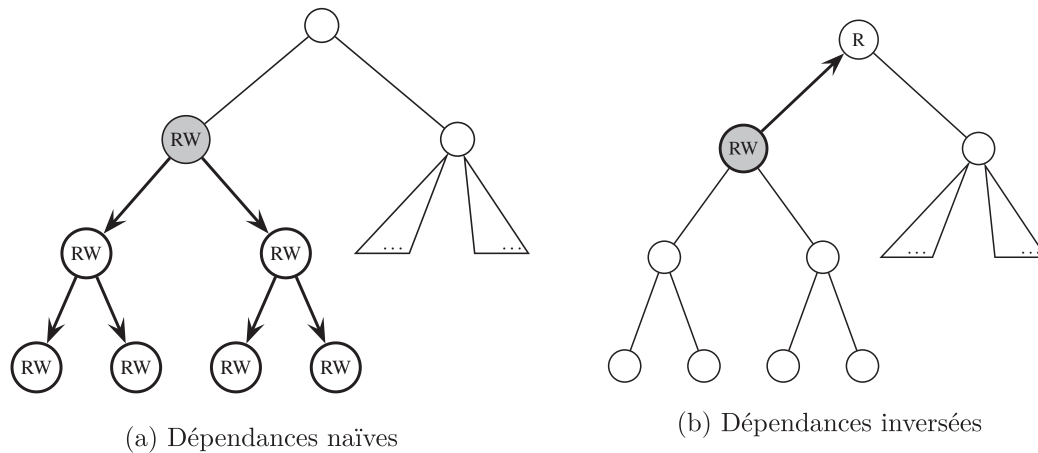
La conversion vers un formalisme de graphe de tâches se fait ici simplement en remplaçant les appels à ces deux opérations par des appels à la fonction `insert_task` avec les paramètres décrits ici. Cette approche n'est cependant pas suffisante, puisqu'elle ne traite pas le cas de l'accès concurrent à deux parties de C en écriture (cas de la figure 4.3). Ceci est fait par l'ajout de dépendances fictives dont la présence ajoute des arcs dans le graphe de tâches, et interdit ainsi l'exécution simultanée de deux tâches dans la configuration de la figure 4.3.

On ajoute alors à l'ensemble des données en écriture de chaque tâche GEMMTASK l'ensemble des nœuds fils de l'arbre de blocs $C_{\sigma\times\tau}$. Ceci est suffisant pour assurer l'exclusion, car :

— Toute tâche écrivant dans un fils de $C_{\sigma\times\tau}$ insérée avant dans le moteur d'exécution doit s'exécuter avant la tâche courante ; toute tâche écrivant dans $C_{\sigma\times\tau}$ insérée après doit s'exécuter après celle-ci.

— Toute tâche écrivant dans un parent de $C_{\sigma\times\tau}$ déclare également une écriture dans $C_{\sigma\times\tau}$, et les contraintes de l'item précédent s'appliquent.

Ces contraintes supplémentaires n'empêchent par ailleurs pas l'exécution parallèle de plusieurs tâches écrivant dans des sous-arbres disjoints de la $\mathcal{H}$-Matrice C. Elles correspondent de plus aux opérations d'écriture effectives. En effet, une écriture dans un nœud de l'arbre a des conséquences sur tous ses fils, puisque les données ne sont réellement stockées que dans les feuilles. Ces dépendances ajoutées sont malheureusement trop nombreuses, ce qui pose des problèmes de performance. En effet, le nombre de ces dépendances a une croissance en $\mathcal{O}(4^{h-p})$, avec p la distance de $\sigma \times \tau$ à la racine de $\mathcal{T}_{I\times J}$, et h la hauteur de cet arbre. Afin d'obtenir un meilleur comportement de l'algorithme, nous effectuons deux optimisations : l'inversion des dépendances artificielles, et le relâchement des contraintes d'exclusion mutuelle.



(a) Dépendances naïves

(b) Dépendances inversées

FIGURE 4.8 – Dépendances. Le nœud grisé est celui qui est écrit, les flèches indiquent le sens de propagation des dépendances. R indique une dépendance en lecture, W en écriture.

Inversion des dépendances Il est possible de résoudre ce problème en inversant les dépendances artificielles, ce qui est illustré par la figure 4.8. Au lieu de rajouter des dépendances en écriture « vers le bas », elles sont ajoutées « vers le haut ». Les dépendances sur la $\mathcal{H}$-Matrice C dans GEMMTASK sont les suivantes :

— Accès en lecture/écriture à $C|_{\sigma\times\tau}$;

— Accès en lecture à tous les parents de $C|_{\sigma\times\tau}$.

Le nombre de dépendances supplémentaires est ici borné par la hauteur h de l'arbre de blocs $\mathcal{T}_{I\times J}$, dont la croissance est logarithmique par rapport au nombre de degrés de liberté de C. Il est aisé de montrer que ces dépendances sont suffisantes :

— Toute tâche écrivant dans un fils de $C|_{\sigma\times\tau}$ ne peut être exécutée en concurrence d'une écrivant dans $C|_{\sigma\times\tau}$, puisqu'un accès en lecture et un accès en écriture à $C|_{\sigma\times\tau}$ ne peuvent être concurrents.

— Idem pour une tâche écrivant dans un parent de $C|_{\sigma\times\tau}$.

Relâchement de contraintes Les contraintes artificielles ajoutées pour éviter des mises-à-jour concurrentes d'un même arbre ne sont pas optimales, car elles induisent un ordre strict entre deux tâches écrivant dans le même sous-arbre, au lieu d'une relation d'exclusion mutuelle, comme dans la remarque 4.12. Cet ordre dépend de l'ordre d'insertion des tâches. Ainsi, si les tâches les plus profondes de l'arbre sont insérées en premier, alors l'exécution se fera de bas en haut, ce qui peut être préjudiciable puisque les tâches plus hautes dans l'arbre tendent à être plus coûteuses. Ainsi, un parcours en largeur de L_{GEMM} plutôt qu'en profondeur permet de s'assurer que les tâches les plus hautes dans l'arbre sont exécutées en premier. Dans le cas où l'ordonnanceur le supporte, il est possible de préciser que tous les accès en écriture peuvent être réordonnés, ce qui permet d'exposer plus de parallélisme dans l'algorithme.

4.5.2 Inversion

L'implémentation de l'inversion est conforme à celle de l'algorithme 4.4, chaque ligne de celui-ci étant séparée par une synchronisation globale. Explicitons la nécessité de cette synchronisation. Rappelons deux lignes de cet algorithme :

$$M_{12} \leftarrow X_{12}M_{22} + M_{12}$$
$$M_{11} \leftarrow -M_{12}X_{21} + M_{21}$$

Ces deux lignes ne sont pas indépendantes, puisque la première ligne modifie M_{12}, qui est une opérande de la seconde, et les dépendances ne sont pas assez fortes pour permettre un séquencement correct des opérations.

Examinons des solutions possibles à ce problème. Soient deux nœuds M_{12}^p et M_{12}^f de M_{12} tels que le premier soit un parent du second, et deux tâches T_1 et T_2 intervenant dans le premier et le second produit respectivement. On considère alors les cas où T_1 modifie M_{12}^f, T_2 lit M_{12}^p, et inversement.

T_1 modifie M_{12}^f, T_2 lit M_{12}^p En l'absence de synchronisation globale, il n'y a pas de garantie que T_2 s'exécute après T_1, ce qui est pourtant nécessaire. En effet, T_1 et T_2 ont une dépendance en lecture sur M_{12}^p (pour T_1, ceci est vrai car M_{12}^p est un parent de M_{12}^f). Cependant, une dépendance en lecture n'entraîne pas d'exclusion mutuelle, ni de relation d'ordre sur l'exécution des tâches. L'exécution de ces deux opérations ne produit pas un résultat identique à un algorithme séquentiel, et n'est donc pas admissible.

Une solution simple pour assurer le bon séquencement des opérations est de modifier les dépendances artificielles pour convertir l'accès en lecture aux parents en un accès en écriture. On vérifie aisément que ceci interdit la situation précédente. Malheureusement, ceci interdit aussi tout parallélisme, puisque toutes les tâches d'un produit accèdent à la racine en écriture ! Certains ordonnanceurs (QUARK par exemple avec l'attribut `GATHERV`) permettent de marquer des accès en écriture compatibles entre eux. L'intention est de permettre au programmeur de modifier des sous-ensembles distincts d'une même donnée de manière concurrente, ce qui est le cas pour les accès dans les parents. Un accès dans ce mode à une donnée ne bloque pas les autres accès avec ce mode, mais les accès en lecture et écriture suivants. Le remplacement des accès en écriture dans les parents de M_{12}^f par des

accès en écriture compatibles entre eux évite donc de rendre les opérations séquentielles, et permet de retrouver une exécution admissible pour ce cas.

T_1 modifie M_{12}^p, T_2 lit M_{12}^f Dans ce cas, l'exécution parallèle n'est pas admissible. En effet, T_2 n'a une dépendance en lecture que sur M_{12}^f, ce qui n'impose pas que T_1 soit terminée avant son exécution, puisque T_1 n'a de dépendance que sur M_{12}^p et ses parents. Il est possible de rajouter des dépendances en lecture sur les ancêtres de M_{12}^f dans T_2, ce qui évite donc cette situation. Cependant, conjugué avec l'ajout des dépendances en écriture sur les ancêtres du paragraphe précédent, ceci revient à imposer une synchronisation globale ! En effet, toutes les tâches du second produit ont une dépendance en lecture sur la racine de M_{12}, et toutes les tâches du premier une dépendance en écriture dessus.

Remarques Ceci est une conséquence de la nature arborescente de l'algorithme : toute modification d'un nœud d'une $\mathcal{H}$-Matrice est en réalité une modification de l'intégralité de l'arbre dont ce nœud est la racine. Le formalisme adopté dans ce chapitre n'a pas connaissance de cette structure. Une méthode pour obtenir un tel comportement est de faire porter les dépendances par les feuilles de l'arbre. Ainsi, une écriture dans M_{12}^f se traduit par une écriture dans toutes ses feuilles, et une lecture dans M_{12}^p par une lecture dans toutes ses feuilles, qui constituent un sur-ensemble de celles de M_{12}^f, introduisant ainsi la dépendance requise. Cette approche n'est cependant pas réalisable, du fait de la taille de l'ensemble des feuilles. Ainsi, dans l'exemple de la section 4.3.1, un calcul comportant 290.030 inconnues mène à un arbre de blocs à 138.820 feuilles.

L'utilisation d'extensions proposées par certains ordonnanceurs n'est pas suffisante pour éviter ce problème, et les algorithmes présentés dans cette section ne sont donc pas composables entre eux ; une synchronisation explicite est donc requise pour permettre une exécution parallèle admissible sans explosion du nombre de dépendances.

4.5.3 Décomposition LU

De même que la parallélisation de l'inversion repose exclusivement sur la parallélisation de la multiplication, il est possible de procéder façon similaire pour la factorisation LU. C'est l'approche qui a été retenue dans [70], et c'est celle que nous présentons ici.

Les algorithmes 4.5 et 4.6 réalisent la décomposition LU et la résolution des systèmes triangulaires inférieurs, le cas des systèmes triangulaires supérieurs étant identique. Dans ces deux algorithmes, toutes les lignes sont séparées par des synchronisations (non représentées pour des raisons de concision).

4.5.4 Performances

Dans cette section, nous explorons la performance parallèle des algorithmes BSP au-dessus d'un moteur d'exécution. Les résultats de cette section sont à rapprocher de ceux de [70], et nous préciserons les comparaisons possibles dans le reste de cette section.

Algorithme 4.5 Décomposition LU, algorithme BSP. Les notations sont celles de l'algorithme 3.12.

```
function ParallelLuDecomposition(M)
    if IsLeaf(M) then
        LuDecomposition(M)
    else
        ParallelLuDecomposition(M11)
        ParallelSolveLower(L11, M12) ‖ ParallelSolveUpper(L21, M21)
        ParallelGemm(M22, −1, L21, U12, 1)
        ParallelLuDecomposition(M22)
    end if
end function
```

Algorithme 4.6 Résolution de $LX = B$, algorithme BSP. Les notations sont celles de l'algorithme 3.13.

```
function ParalleleSolveLower(L, B)
    if ¬ IsLeaf(L) ∧ ¬ IsLeaf(B) then
        ParallelSolveLower(L11, B11) ‖ ParallelSolveLower(L11, B12)
        ParallelGemm(()B21, −1, L21, B11, 1)
        ParallelSolveLower(L22, B21) ParallelGemm(B22, −1, L21, B12, 1)
        ParallelSolveLower(L22, B22)
    else
        SolveLower(L, B)
    end if
end function
```

4.5.4.1 Configuration

Configuration physique Nous présenterons des tests effectués sur des cas réalistes, pour l'assemblage d'une $\mathcal{H}$-Matrice et l'inversion. Ces deux étapes sont de loin les plus consommatrices de temps dans un calcul, comme mis en évidence dans le chapitre précédent. Les temps de calcul sont dans cette section relatifs à des calculs d'électromagnétisme à l'aide du logiciel Aseris BE, pour des calculs EFIE sur des objets tri-dimensionnels dans le domaine fréquentiel. La géométrie utilisée est celle d'un « Cône-Sphère », semblable à celui de la figure 3.8 page 75. Cet objet parfaitement conducteur est caractérisé par un angle au sommet de 15 degrés, et une taille de la partie sphérique variable en nombre de longueur d'onde. La fréquence est fixée à 15GHz et la taille des arêtes est $\lambda/12$.

Configuration de calcul Pour tous les calculs, on s'intéresse au temps d'assemblage de la matrice d'interaction A (équation (2.2.17)) dans le format $\mathcal{H}$-Matrice, et au calcul de son inverse approchée par l'algorithme 4.4. Dans tous les cas, le calcul est effectué en double précision. La compression est réalisée par l'algorithme ACA avec pivotage partiel, avec une tolérance $\varepsilon = 10^{-4}$. Les blocs sont recompressés après assemblage avec la même tolérance. Les autres paramètres sont $\eta = 3$ et $n_{leaf} = 100$, et le découpage est médian.

Configuration informatique Les calculs sont effectués sur la machine Curie-16 présentée section 3.5.1.1.3 page 111. La figure 3.16 donne la topologie simplifiée de cette machine, et le tableau 3.3 ses principales caractéristiques. Cette machine est de type ccNUMA, et en l'absence de gestion fine de le localité par le code de calcul et l'ordonnanceur, la politique de placement de la mémoire est contrôlée avec l'utilitaire `numactl`. Celui-ci permet de spécifier une politique de placement des allocations sur les différents bancs de mémoire. La politique par défaut sous Linux est de type « *first touch* » [71], ce qui signifie que la mémoire est allouée sur le nœud NUMA de sa première utilisation. Ce choix n'est pas toujours adapté car il peut engendrer un engorgement d'un nœud, nous utilisons donc la politique *interleave*, qui étale la mémoire sur tous les nœuds NUMA d'une machine.

Implémentations Deux implémentations sont considérées dans cette section :

- L'implémentation séquentielle du chapitre précédent. Elle est utilisée pour les temps de référence sur un processeur. Les deux implémentations effectuant exactement les mêmes calculs, l'utilisation de l'implémentation séquentielle permet d'exclure tous les surcoûts cachés de l'implémentation parallèle lorsqu'elle est utlisée sur un seul cœur.
- L'implémentation parallèle décrite dans cette section. Le moteur d'exécution utilisé dans ce cas est QUARK (*cf.* section 4.4.3.1). Elle repose pour toutes les opérations de base sur la version séquentielle, avec laquelle elle partage la majorité de son code source.

4.5.4.2 Assemblage

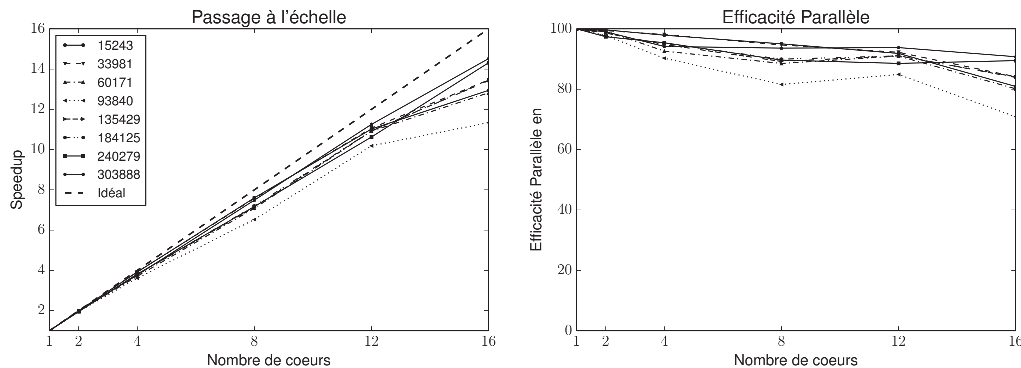


FIGURE 4.9 – Passage à l'échelle de l'assemblage sur Curie-16.

L'assemblage d'une $\mathcal{H}$-Matrice est une tâche de type « *embarassingly parallel* » (massivement parallèle), c'est-à-dire que son découpage en un grand nombre de tâches indépendantes est trivial. Ceci explique le très bon passage à l'échelle observé dans ce cas, puisqu'aucune synchronisation n'est réellement nécessaire.

La faible baisse d'efficacité parallèle (figure 4.9 et table 4.1) peut s'expliquer par plusieurs facteurs :

1. Le surcoût du moteur d'exécution ;

	$p=1$	$p=2$		$p=4$		$p=8$		$p=12$		$p=16$	
N	t	t	E	t	E	t	E	t	E	t	E
33981	232.5	116.9	99.4	59.3	98.1	30.7	94.7	21.0	92.2	17.3	84.1
60171	505.5	254.6	99.3	136.4	92.6	71.4	88.5	46.3	91.0	39.4	80.1
93840	861.7	440.8	97.7	238.7	90.3	132.1	81.6	84.6	84.9	76.0	70.9
135429	1461.9	741.4	98.6	385.5	94.8	204.9	89.2	133.8	91.1	108.9	83.9
184125	2155.0	1104.5	97.6	566.0	95.2	299.5	89.9	197.3	91.0	160.2	84.1
240279	3262.7	1675.2	97.4	855.2	95.4	455.0	89.6	307.1	88.5	227.9	89.5
303888	4517.7	2281.4	99.0	1199.6	94.2	603.3	93.6	401.4	93.8	311.1	90.8

Table 4.1 – Passage à l'échelle de l'assemblage sur Curie-16. Les notations sont celles de la définition 4.1, les temps en seconde et les efficacités en pourcentage.

2. La saturation du bus mémoire ;
3. Les effets ccNUMA ;
4. L'inefficacité de l'ordonnancement par liste.

Le premier point est une possibilité, néanmoins relativement peu probable. En effet, les tâches sont relativement peu nombreuses (égales au nombre de feuilles de la $\mathcal{H}$-Matrice), et n'ont pas de dépendances entre elles, ce qui ne nécessite pas d'actions complexes de la part de l'ordonnanceur.

Le second et le troisième interviennent dans une certaine mesure, sans qu'il soit possible de le quantifier précisément. À titre de comparaison, les tests d'Intel avec la bibliothèque d'algèbre linéaire optimisée MKL donnent une efficacité parallèle maximale de 87% pour l'opération gemm avec une configuration très proche (2 processeurs Intel Xeon « Sandy Bridge » E5-2690, MKL 11.0) [2]. L'assemblage et la compression d'une $\mathcal{H}$-Matrice comprend deux parties : (a) le calcul des lignes et colonnes de la matrice d'interaction, et (b) la recherche des pivots et la mise à jour de la $\mathcal{R}k$-Matrice. La première partie est intensive en calculs et peu exigeante du point de vue de la mémoire du fait du coût élevé des opérations. La seconde est peu dense en calculs, et essentiellement limitée par la bande passante mémoire (blas1 et 2). Il est donc probable que l'efficacité parallèle maximale de cette opération soit supérieure à celle de gemm.

Enfin, l'inefficacité de l'ordonnancement par liste est probablement le facteur dominant. En effet, le caractère imprévisible du rang d'un bloc compressé rend un ordonnancement statique impossible, et le temps de calcul global en est pénalisé. Ceci est à mettre en rapport avec le lemme 4.3.

4.5.4.3 Inversion

Dans le cas de l'inversion, les algorithmes de type BSP ne donnent pas des résultats aussi satisfaisants, comme le montrent la figure 4.10 et la table 4.2. Outre les facteurs relatifs à l'architecture matérielle de la machine, la raison principale de ce mauvais passage à l'échelle est liée à la nature de la parallélisation, et en particulier aux nombreuses synchronisations.

2. `http://software.intel.com/en-us/intel-mkl`, section « Benchmarks »

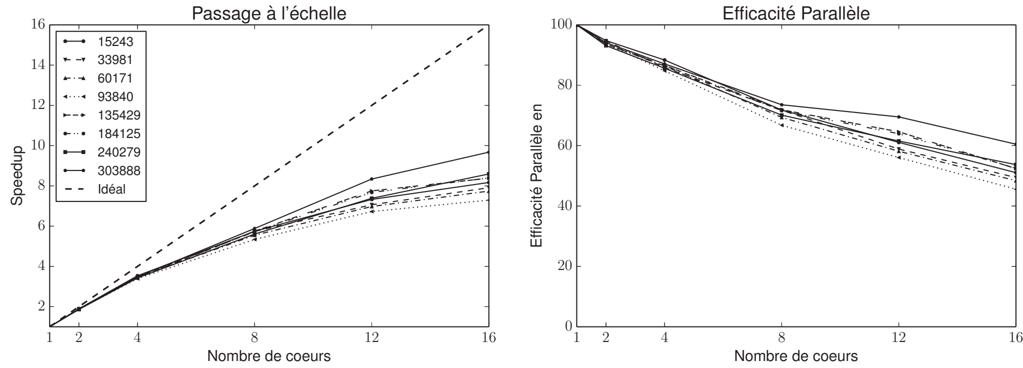


Figure 4.10 – Passage à l'échelle sur Curie de l'inversion.

	$p = 1$	$p = 2$		$p = 4$		$p = 8$		$p = 12$		$p = 16$	
N	t	t	E	t	E	t	E	t	E	t	E
15243	246.1	129.8	94.8	69.6	88.3	42.8	71.8	33.6	61.1	30.1	51.1
33981	787.5	415.9	94.7	229.3	85.9	137.3	71.7	111.5	58.9	99.6	49.4
60171	2121.9	1133.3	93.6	614.0	86.4	382.5	69.4	304.2	58.1	274.1	48.4
93840	3607.0	1923.8	93.7	1063.5	84.8	675.5	66.7	536.2	56.1	494.8	45.6
135429	8010.9	4278.0	93.6	2313.2	86.6	1392.9	71.9	1033.6	64.6	955.6	52.4
184125	12258.4	6542.4	93.7	3541.8	86.5	2140.4	71.6	1597.7	63.9	1460.2	52.5
240279	21714.0	11659.8	93.1	6339.2	85.6	3872.3	70.1	2941.1	61.5	2527.1	53.7
303888	32639.6	17342.5	94.1	9371.0	87.1	5548.4	73.5	3913.2	69.5	3373.4	60.5

Table 4.2 – Passage à l'échelle sur Curie de l'inversion. Les notations sont celles de la définition 4.1, les temps en seconde et les efficacités en pourcentage.

En effet, l'algorithme 4.4 comporte de nombreuses synchronisations globales : 6 dans chaque appel à la fonction ParallelInverse, sans compter les synchronisations des appels récursifs. Le coût de ces synchronisations croit avec le nombre de processeurs, de même que le coût des primitives de synchronisation utilisées par le moteur d'exécution. De plus, il existe des étapes purement séquentielles dans cet algorithme (l'inversion des blocs diagonaux et certains cas de base des produits), dont l'importance relative augmente avec le nombre de processeurs (« loi d'Amdahl »). Enfin, la disparité dans le temps d'exécution de chaque tâche élémentaire et l'impossibilité de prévoir ce temps expose les faiblesses de l'ordonnancement par liste, qui se retrouvent dans ces algorithmes.

Remarque 4.14 (Mémoire distribuée). *Il faut noter que ces inconvénients seraient exacerbés dans le cas d'un calcul en mémoire distribuée. Dans ce cas, le coût des primitives de synchronisation augmente fortement, de même que le coût des synchronisations globales.*

Par ailleurs, il est noté dans [70] que l'efficacité parallèle de ces algorithmes décroît avec l'augmentation de η. Ceci est à rapprocher de la figure 3.45 page 135 qui montre une augmentation de la dispersion dans le rang des blocs compressés avec l'augmentation de η_{crit}. Une plus grande dispersion entraîne un plus grand déséquilibre entre le temps d'exécution des opérations, ce qui se traduit par une perte d'efficacité parallèle.

4.5.4.4 Décomposition LU

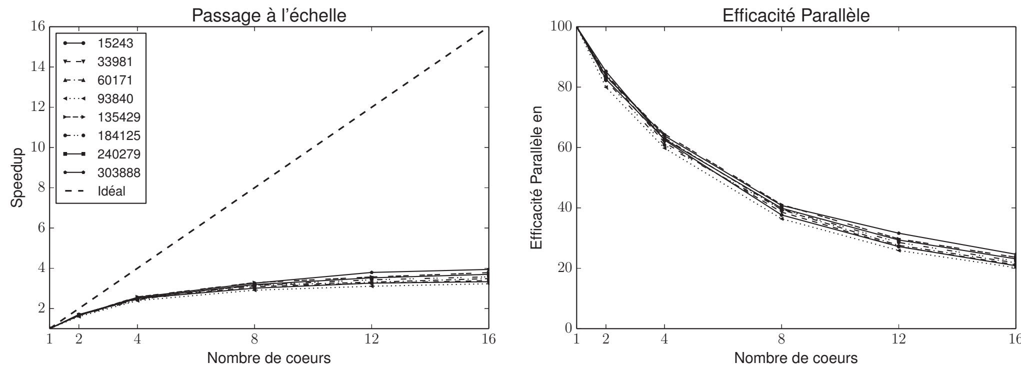


FIGURE 4.11 – Passage à l'échelle sur Curie de la décomposition LU.

	$p = 1$	$p = 2$		$p = 4$		$p = 8$		$p = 12$		$p = 16$	
N	t	t	E	t	E	t	E	t	E	t	E
15243	98.0	57.5	85.2	39.2	62.5	32.5	37.6	30.0	27.2	29.4	20.8
33981	337.3	200.8	84.0	136.8	61.6	109.2	38.6	103.5	27.2	99.5	21.2
60171	904.8	536.6	84.3	358.1	63.2	287.1	39.4	273.3	27.6	259.5	21.8
93840	1635.0	1022.2	80.0	683.6	59.8	561.7	36.4	526.2	25.9	506.1	20.2
135429	3513.0	2104.3	83.5	1362.3	64.5	1068.6	41.1	984.6	29.7	928.5	23.6
184125	5472.3	3327.2	82.2	2249.6	60.8	1728.4	39.6	1597.2	28.6	1530.9	22.3
240279	9562.2	5786.4	82.6	3808.7	62.8	2996.4	39.9	2712.9	29.4	2585.9	23.1
303888	14606.7	8721.4	83.7	5716.1	63.9	4475.8	40.8	3850.8	31.6	3705.0	24.6

FIGURE 4.12 – Passage à l'échelle sur Curie de la décomposition LU. Les notations sont celles de la définition 4.1, les temps en seconde et les efficacités en pourcentage.

Il est noté dans [70] que le passage à l'échelle de la décomposition LU est mauvais avec les algorithmes BSP. Cette observation est confirmée par les tests, dont les résultats sont donnés par les figures 4.11 et 4.12. Plusieurs observations peuvent expliquer ceci :

— Le nombre de synchronisations nécessaires est comparable à l'inversion, alors que les calculs sont moins nombreux. En effet, le temps de calcul pour une factorisation LU séquentielle d'une $\mathcal{H}$-Matrice de taille 135.429 est de 3 373s, à comparer aux 13 197s nécessaires pour l'inversion de la même $\mathcal{H}$-Matrice. Le coût relatif des synchronisations est alors plus grand.

— Les opérations séquentielles (factorisation des blocs diagonaux et cas de base de l'algorithme 4.6 ont un poids relatif plus important dans le calcul. En effet, les cas de base de la résolution des systèmes triangulaires ne concernent pas uniquement des petits blocs de la $\mathcal{H}$-Matrice à factoriser, et toute étape purement séquentielle pénalise fortement le passage à l'échelle.

— Le produit matrice-matrice PARALLELGEMM dont le passage à l'échelle est bon représente une part plus faible des opérations.

La conjonction de ces observations montre la plus grande difficulté de parallélisation de la factorisation LU par rapport à l'inversion. Malheureusement, cette opération est

celle qui est utilisée en pratique, pour des raisons de temps de calcul et de précision. Il est donc important d'utiliser une méthode de parallélisation plus efficace, qui est l'objet de la section suivante.

Remarque 4.15 (Décomposition LDL^T)**.** *Pour les mêmes raisons, le passage à l'échelle de la décomposition LDL^T serait également mauvais, avec probablement un comportement moins bon que la décomposition LU. Il n'a cependant pas été réalisé d'implémentation BSP de cette décomposition permettant d'appuyer ces suppositions.*

4.6 Parallélisation à l'aide de graphes de tâches

L'utilisation de graphes de tâches pour les algorithmes BSP simplifie l'écriture parallèle en déchargeant le programmeur de la gestion d'une liste de tâches et des *mutexes*, mais n'est pas pleinement satisfaisante. Nous présentons donc dans cette section une implémentation des algorithmes sous forme de graphes de tâches, et dont la parallélisation ne fait pas intervenir de synchronisations explicites globales. À cette fin, après avoir introduit deux exemples de composition des opérations, nous donnons une description des éléments constitutifs du graphe de tâches utilisé. En particulier, les problèmes de la granularité et du contrôle du nombre de dépendances seront traités. Nous décrivons ensuite les opérations implémentées avec ce formalisme, puis donnerons des résultats numériques.

4.6.1 Exemples de composition

L'expression d'un algorithme sous forme d'un graphe de tâches capturant les dépendances entre des opérations élémentaires invite naturellement à considérer la composition des opérations. En effet, tant que les dépendances entre les opérations sont suivies de manière précise, il n'est pas nécessaire d'ajouter des appels de synchronisation entre les opérations. Afin d'illustrer l'impact de la composition sur un graphe de tâche, retournons à l'exemple 4.4.2 de la multiplication $C \leftarrow AB + C$, avec A, B et C des matrices 2×2 par bloc. Supposons que le produit soit précédé par l'assemblage des matrices A et B par le code précédent :

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    insert_task(assembly, A[i][j], IN | OUT);
    insert_task(assembly, B[i][j], IN | OUT);
  }
}
wait_all_tasks();
```

L'assemblage ajoute 8 tâches n'ayant aucune dépendances entre elles. Néanmoins, la présence de la synchronisation `wait_all_tasks()` à la fin de l'assemblage force une attente générale avant de procéder au produit. Si cette attente est supprimée, le graphe de tâches de la figure 4.6 devient celui de la figure 4.13.

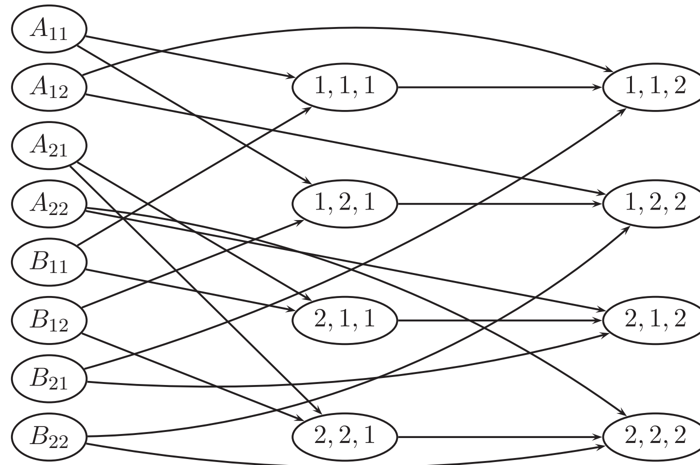


FIGURE 4.13 – Graphe de tâches, assemblage et produit matriciel. Les tâches d'assemblage sont à gauche, étiquetées par le bloc matriciel assemblé ; les tâches du produit sont représentées comme sur la figure 4.6.

Ce graphe présente de plus grandes possibilités de réorganisations des tâches. Par exemple, il n'est pas indispensable d'assembler les blocs A_{22} ou B_{22} tout de suite, puisqu'ils ne sont nécessaires que pour des tâches dont l'exécution est plus tardive que les premières tâches du produit. Cette flexibilité supplémentaire se traduit par des possibilités d'optimisation plus grandes, et un parallélisme potentiellement plus important.

Algorithme 4.7 Décompositon LU, énumération des tâches.

INSERTTASK(ASSEMBLE, M_{11}) INSERTTASK(ASSEMBLE, M_{12})
INSERTTASK(ASSEMBLE, M_{21}) INSERTTASK(ASSEMBLE, M_{22})
INSERTTASK(LULEAF, M_{11})
INSERTTASK(SOLVELOWER, M_{11}, M_{12})
INSERTTASK(SOLVEUPPER, M_{11}, M_{21})
INSERTTASK(GEMM, M_{22}, M_{21}, M_{12})
INSERTTASK(LULEAF, M_{22})

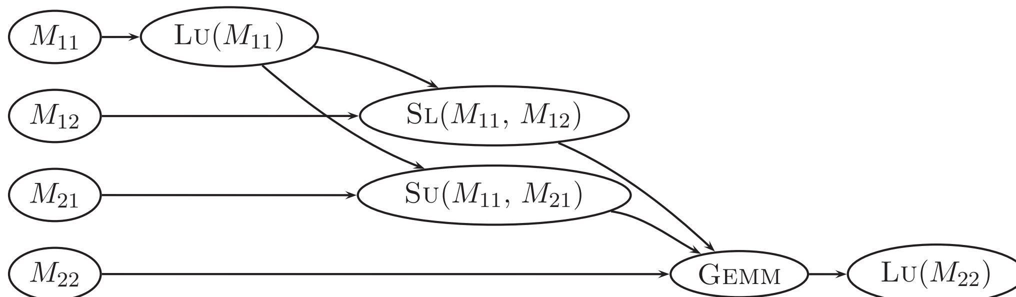


FIGURE 4.14 – Graphe de tâches pour l'algorithme 4.7. Les tâches à gauche représentent l'assemblage, SL et SU signifient SOLVELOWER et SOLVEUPPER, respectivement.

Un autre exemple est celui de la décomposition LU d'une matrice 2×2 par blocs. Dans ce cas, la liste des tâches à effectuer est énuméré par l'algorithme 4.7, et le graphe de tâches correspondant représenté sur la figure 4.14. Dans ce dernier, l'assemblage de M_{22} peut par exemple être placé à n'importe quel endroit avant l'exécution de Gemm.

En règle générale, la structure d'une $\mathcal{H}$-Matrice est plus complexe que celle d'une matrice 2×2 par bloc. L'approche retenue dans la suite consiste à considérer les cas de base des algorithmes récursifs comme des tâches élémentaires. La description de cette approche est l'objet de la section suivante.

4.6.2 Décomposition en tâches

La forme générale des algorithmes du chapitre 3 comprend un cas de récursion, et plusieurs cas de base, lesquels portent les calculs (à l'exception de la conversion hiérarchique). Considérer ces cas de bases comme des tâches élémentaires est donc naturel, d'autant plus que certains cas de base sont difficilement décomposables (par exemple, l'addition). Ceci permet également de réutiliser les algorithmes séquentiels pour ces cas de base sans modification, ce qui apporte des gains de lisibilité et de concision dans l'expression.

Néanmoins, comme cela a été évoqué dans la section 4.5, le nombre de dépendances à suivre devient alors très grand. De plus, le niveau de granularité des tâches n'est plus adapté à une exploitation efficace des ressources. Il est donc nécessaire d'adapter la granularité et de réduire le nombre de dépendances. Ceci sera fait au travers d'une coupe de l'arbre de blocs de chaque $\mathcal{H}$-Matrice intervenant dans les algorithmes.

Nous présentons ici les diverses tâches élémentaires exécutées par le moteur d'exécution pour instancier les algorithmes $\mathcal{H}$-Matrice en parallèle. Initialement, ces tâches opèrent principalement sur des feuilles de leurs $\mathcal{H}$-Matrices respectives. Cet aspect sera néanmoins généralisé dans la suite.

Du point de vue du moteur d'exécution, une tâche est définie par :

- Une fonction à exécuter ;
- Un nombre d'arguments ;
- Un nombre de dépendances, avec un mode d'accès (lecture et/ou écriture) associé.

Il convient de distinguer les deux derniers éléments. Ainsi, certains arguments peuvent ne pas participer à l'établissement des contraintes d'ordonnancement. De même, certaines dépendances peuvent ne pas être des arguments de la fonction, mais des éléments ajoutés pour garantir un séquencement correct des opérations.

Assembly Assemblage d'un bloc $M|_{\sigma\times\tau}$, admissible ou non. Dans le cas d'un bloc admissible, cette opération peut être accompagnée de la recompression de ce bloc après assemblage (section 3.3.2.2.1). Cette recompression peut être isolée ou non de l'assemblage initial.

> **Lecture** Aucune
>
> **Écriture** $M|_{\sigma\times\tau}$
>
> **Structure** $M|_{\sigma\times\tau}$ est une feuille

SYMMETRICASSEMBLY Assemblage de deux blocs $M|_{\sigma\times\tau}$ et $M_{\tau\times\sigma}$ pour une matrice symétrique. Cette opération n'est utilisée ici que dans le cas d'algorithmes non symétriques appliqués sur une matrice symétrique (par exemple LU pour une matrice issue d'une formulation EFIE). Il s'agit de l'assemblage du bloc inférieur, puis de la copie avec transposition dans le bloc supérieur.

Lecture Aucune

Écriture $M|_{\sigma\times\tau}$, $M|_{\tau\times\sigma}$

Structure $M|_{\sigma\times\tau}$ et $M|_{\tau\times\sigma}$ sont des feuilles de M, par symétrie du critère d'admissibilité.

AXPY Effectue l'opération $A \leftarrow \alpha B + A$.

Lecture A, B

Écriture A

Structure A et B sont des feuilles d'arbres de même structure.

GEMM Effectue l'opération $C \leftarrow \alpha AB + \beta C$.

Lecture A, B, C

Écriture C

Structure A et/ou B et/ou C est une feuille

SOLVEUPPER Résolution de $XU = B$, où B est écrasée par la solution X. U est une $\mathcal{H}$-Matrice triangulaire supérieure et B une $\mathcal{H}$-Matrice.

Lecture U, B

Écriture B

Structure U et/ou B est une feuille.

SOLVELOWER Résolution de $LX = B$, où B est écrasé par la solution X. L est une $\mathcal{H}$-Matrice triangulaire inférieure et B une $\mathcal{H}$-Matrice.

Lecture L, B

Écriture B

Structure L et/ou B est une feuille

LULEAF Décomposition $M|_{\sigma\times\tau} = LU$ sur place.

Lecture $M|_{\sigma\times\tau}$

Écriture $M|_{\sigma\times\tau}$

Structure $M|_{\sigma\times\tau}$ est une feuille

INVERSELEAF Inversion $M|_{\sigma\times\tau} \leftarrow M|_{\sigma\times\tau}{}^{-1}$ sur place. Utilise une $\mathcal{H}$-Matrice temporaire $X|_{\sigma\times\tau}$.

Lecture $M|_{\sigma\times\tau}$, $X|_{\sigma\times\tau}$

Écriture $M|_{\sigma\times\tau}$, $X|_{\sigma\times\tau}$

Structure $M|_{\sigma\times\tau}$ et $X|_{\sigma\times\tau}$ sont des feuilles.

4.6.3 Découpage des tâches

4.6.3.1 Nombre de dépendances et granularité

Une fois les tâches élémentaires définies, la traduction sous forme de graphe de tâches des algorithmes du chapitre 3 est relativement directe, en remplaçant les appels aux cas de bases par des insertions de tâches. Néanmoins, pour garantir un bon séquencement

des opérations, les dépendances doivent être traduites au niveau des feuilles de chaque $\mathcal{H}$-Matrice. L'ensemble des dépendances des tâches élémentaires est donc $\mathcal{L}(H)$, l'ensemble des feuilles de H.

Algorithme 4.8 Produit Matrice-Matrice, version « fine ». On suppose ici $\beta = 1$ pour simplifier l'écriture.

```
function FineDagGemm(C, α, A, B, σ ∈ T_I, τ ∈ T_J, ρ ∈ T_K)
    if ¬IsLeaf(σ × τ) et ¬IsLeaf(σ × ρ) et ¬IsLeaf(ρ × τ) then
        for all (σ', τ', ρ') ∈ S(σ) × S(τ) × S(ρ) do
            FineDagGemm(C, α, A, B, σ', τ', ρ')
        end for
    else
        W ← L(C|σ×τ)
        R ← L(A|σ×ρ) ∪ L(B|ρ×τ)
        InsertTask(Gemm, C|σ×τ, W, IN | OUT, α, A|σ×ρ, B|ρ×τ, R, IN)
    end if
end function
```

Un exemple de transcription pour la multiplication est donné par l'algorithme 4.8. Il permet une composition correcte des opérations, au coût (a) d'une explosion du nombre de dépendances, et (b) d'un mauvais contrôle de la granularité.

Nombre de dépendances L'explosion du nombre de dépendances est lié à la structure d'une $\mathcal{H}$-Matrice. Supposons une $\mathcal{H}$-Matrice A construite sur un arbre de groupes ligne et colonne identiques, $\mathcal{T}_I$, de hauteur h, et notons

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

Supposons de plus que le bloc extra-diagonal A_{12} de plus haut niveau de l'arbre de blocs soit admissible, c'est-à-dire un bloc de profondeur 1 dans l'arbre de groupes. Alors le produit $A_{11}A_{12}$ est un cas de base, puisque A_{12} est une feuille de l'arbre de groupes, et le nombre de dépendances en lecture est $|R| = 1 + |\mathcal{L}(A_{11})|$. En supposant que l'arbre de groupes $\mathcal{T}_I$ est entier, alors $|\mathcal{L}(A_{11})| \geqslant 2^{h-1} = \mathcal{O}(|I|)$. En effet, pour toute feuille σ de l'arbre de groupes $\mathcal{T}_I$, le bloc $A|_{\sigma\times\sigma}$ est une feuille non admissible de l'arbre de blocs, et donc une feuille de A_{11} (la diagonale de la $\mathcal{H}$-Matrice). L'ensemble des dépendances en lecture pour ce produit satisfait donc :

$$|R| \geqslant 1 + 2^{h-1} = \mathcal{O}(|I|),$$

ce qui est trop élevé pour un moteur d'exécution.

Granularité Inversement, considérons le produit $A_{11}A_{11}$. À chaque niveau, la récursion se poursuit sur la diagonale, puisqu'un bloc $A|_{\sigma\times\sigma}$ n'est jamais admissible. La récursion s'interrompt donc sur une feuille non admissible de la diagonale de A_{11}. Cette feuille est de taille bornée par $N_{leaf} \times N_{leaf}$. Dans le cas où $N_{leaf} = 32$, le nombre d'opérations nécessaires pour effectuer ce produit est de l'ordre de $2^{15} = 32768$, ce qui nécessite

quelques μs de calcul pour un processeur moderne. Cette granularité est trop faible pour une exploitation optimale des ressources, en particulier au regard du surcoût de l'appel `insert_task()`, qui est du même ordre de grandeur sur une machine courante [18, chapitre 7]. Il faut par ailleurs noter que la tâche de multiplication matricielle d'une feuille non admissible représente un cas favorable de ce point de vue, du fait de sa complexité en $\mathcal{O}(N_{leaf}^3)$. Une addition ou un produit matrice-vecteur serait plus pénalisant, du fait de la complexité en $\mathcal{O}(N_{leaf}^2)$ de ces opérations. Certaines opérations sont à l'inverse très coûteuses, et ce déséquilibre nuit également à l'efficacité des algorithmes parallèles.

4.6.3.2 Coupe de l'arbre

Le suivi précis des dépendances permet de composer les opérations, avec comme inconvénients une mauvaise maîtrise du nombre de dépendances et de la granularité des opérations. Ces deux problèmes ont pour origine la différence de profondeur entre les feuilles les plus grosses et les plus petites d'une $\mathcal{H}$-Matrice. Une solution pour remédier à ce problème est d'imposer une taille maximale aux feuilles, ce qui n'est pas acceptable car ceci entraîne une détérioration de la complexité temporelle et spatiale des algorithmes. La solution contraire consistant à agrandir la taille minimale des feuilles par l'augmentation de N_{leaf} a les mêmes effets.

L'arbre d'une $\mathcal{H}$-Matrice est en général de profondeur très inégale suivant ses branches (les feuilles correspondant à des groupes de degrés de liberté éloignés étant plus grandes), et la quantité de données (et donc de calcul) n'est pas distribuée de manière homogène pour un niveau donné. La solution au problème de granularité des opérations passe par une coupe dans l'arbre, en arrêtant de façon anticipée la récursion lorsque la granularité des opérations sur les fils est jugée trop faible. Cependant, du fait de l'irrégularité de l'arbre, il n'est pas souhaitable d'opérer une coupe à niveau constant (c'est-à-dire à une distance fixe de la racine).

En effet, supposons une coupe à une distance p de la racine de l'arbre. Ceci suppose que l'arbre ne possède pas de feuilles à une profondeur inférieure à p, ce qui est une limitation arbitraire potentiellement pénalisante. De plus, cela mène à des déséquilibres de charge importants entre les différentes branches de l'arbre. Il est donc préférable d'opérer une coupe de profondeur variable dans l'arbre, permettant ainsi une plus grande souplesse, et l'utilisation de diverses stratégies visant à obtenir des tâches plus équilibrées en temps de calcul.

En contrôlant le nombre d'éléments de la coupe, le nombre de dépendances de chaque tâche est borné. Le choix de la forme de cette coupe a lui un impact sur la granularité des opérations, et sur le degré de parallélisme exhibé par cette formulation.

Nous notons dans la suite de ce document une telle coupe L_0, définie ainsi :

Définition 4.16 (Coupe L_0)**.** *Soit un arbre $\mathcal{T}$. Une coupe L_0 de cet arbre est un ensemble de nœuds ou de feuilles de $\mathcal{T}$ tel que toute feuille de $\mathcal{T}$ a un unique ancêtre dans L_0. On note de plus $L_0(A)$ la coupe d'une $\mathcal{H}$-Matrice A dans le cas où la $\mathcal{H}$-Matrice désignée est ambiguë.*

Cette définition implique que tout chemin partant de la racine rencontre un unique nœud d'une coupe de celui-ci, ce qui est visible sur l'exemple de la figure 4.15. Autrement

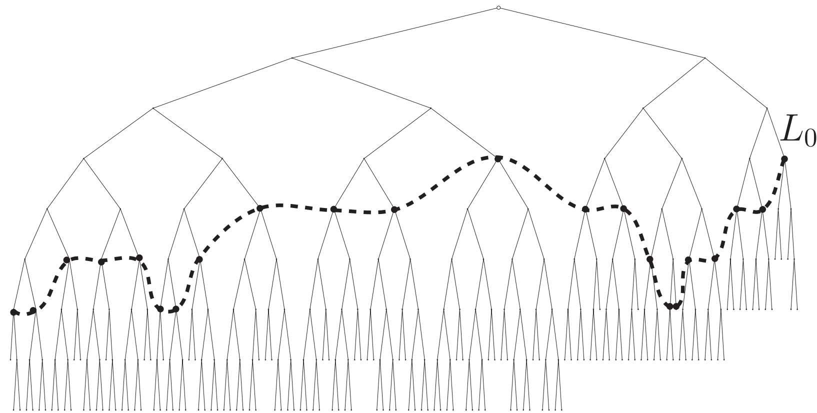


Figure 4.15 – Exemple de coupe L_0. L'arbre représenté est binaire pour des raisons de lisibilité, les coupes considérées ici sont effectuées sur des *quadtrees*.

dit, pour un arbre $\mathcal{T}$ et une coupe $L_0(\mathcal{T})$ de cet arbre, la restriction de cet arbre aux ancêtres des éléments de la coupe est également un arbre, de même arité que l'arbre initial.

4.6.4 Données et tâches

L'objectif de la coupe L_0 dans l'arbre de blocs est d'assurer une granularité contrôlée des tâches, mais également de borner le nombre de dépendances maintenues par le moteur d'exécution. Ce deuxième objectif est atteint en imposant que les seules données élémentaires dont le moteur d'exécution a connaissance sont des éléments de L_0, qui ne sont pas nécessairement des feuilles de leur $\mathcal{H}$-Matrice.

Ceci nécessite l'extension des tâches élémentaires de la section 4.6.2, dont les opérandes ne sont plus obligatoirement des feuilles. Ce changement ne requiert pas d'ajustement dans ces tâches, puisqu'elles consistent en l'application des algorithmes séquentiels sur leurs opérandes. En revanche, les dépendances sur ces opérations ne sont plus les mêmes. Dans l'algorithme 4.8, les dépendances en lecture et écriture sont des ensembles de feuilles. Dans ce qui suit, les dépendances sont des ensembles de nœuds de L_0. Pour chaque $\mathcal{H}$-Matrice H, une coupe $L_0(H)$ est construite, et ses éléments sont déclarés comme données au moteur d'exécution.

Dans la suite, nous prendrons l'exemple de la multiplication matricielle présentée par l'algorithme 4.8 afin d'exposer les principes de l'approche retenue. Afin d'adapter l'algorithme aux nouvelles contraintes (les dépendances appartiennent toutes à $L_0(A)$, $L_0(B)$ et $L_0(C)$), il est nécessaire de modifier (a) le critère d'arrêt de la récursion, et (b) l'énumération des dépendances.

4.6.4.1 Arrêt de la récursion

La condition d'arrêt de la récursion dans l'algorithme 4.8 est :

$$\textsc{IsLeaf}(\sigma \times \tau) \vee \textsc{IsLeaf}(\sigma \times \rho) \vee \textsc{IsLeaf}(\rho \times \tau) \tag{4.6.1}$$

C'est-à-dire que la récursion prend fin quand une des opérandes est une feuille. Cette condition est ici modifiée de la manière suivante :

$$C|_{\sigma\times\tau} \in L_0(C) \vee (\text{IsLeaf}(\sigma\times\tau) \vee \text{IsLeaf}(\sigma\times\rho) \vee \text{IsLeaf}(\rho\times\tau)) \tag{4.6.2}$$

Cette condition peut être vérifiée dans deux situations :

1. $C|_{\sigma\times\tau}$ est un élément de $L_0(C)$;
2. L'autre clause de la condition est vérifiée. Dans ce cas $C|_{\sigma\times\tau}$ est au-dessus de $L_0(C)$, et n'est pas une feuille.

Dans le premier cas, $A|_{\sigma\times\rho}$ et $B|_{\rho\times\tau}$ peuvent être au-dessus ou en-dessous de leurs coupes respectives. Dans le second cas, au moins un de ces deux nœuds est sur ou en-dessous de sa coupe, puisque toute feuille d'une $\mathcal{H}$-Matrice ne peut être que sur ou sous la coupe. Ces diverses situations nécessitent un traitement différent du point de vue de l'énumération des dépendances.

Remarque 4.17 (Critère d'arrêt)**.** *Il est possible de choisir d'autres critères d'arrêt. Par exemple, la récursion peut être arrêtée lorsque l'une des opérandes est un élément de sa coupe, mais cette solution n'a pas été retenue ici. En revanche, il n'est pas possible d'imposer que l'arrêt de la récursion se fasse lorsque les trois opérandes sont des éléments de leurs coupes respectives, de même qu'il est impossible dans les algorithmes séquentiels d'imposer que les trois opérandes soient des feuilles. Ceci a un impact sur la complexité de l'énumération des dépendances, qui est l'objet de la section suivante.*

4.6.4.2 Énumération des dépendances

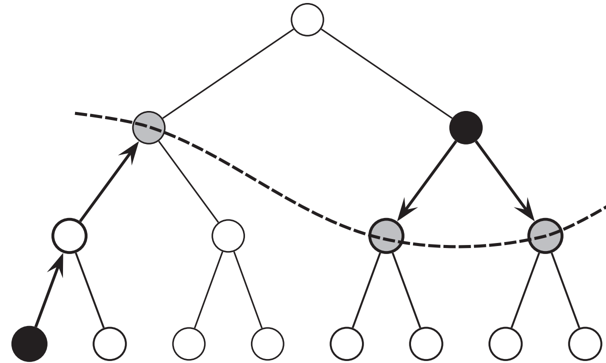


FIGURE 4.16 – Exemple de dépendances sous et au-dessus de L_0.

On considère une dépendance $M|_{\sigma\times\tau}$ apparaissant dans un cas de base. On cherche à exprimer l'ensemble des dépendances $D \subset L_0(M)$ à associer à ce nœud de M pour le moteur d'exécution. Cette dépendance peut se situer (a) sur, (b) au-dessus ou (c) en-dessous de $L_0(M)$. Dans le premier cas, l'énumération de la dépendance est triviale, et $D = \{M|_{\sigma\times\tau}\}$.

Dans le second cas, on procède de manière identique à l'énumération des dépendances dans l'algorithme 4.8. Dans celui-ci, on définit $D = \mathcal{L}(M|_{\sigma\times\tau})$. La construction équivalente dans ce cas est de prendre l'ensemble des éléments de $L_0(M)$ qui sont des descendants de

$M|_{\sigma\times\tau}$. En notant $\mathcal{N}(M|_{\sigma\times\tau})$ l'ensemble des nœuds de l'arbre dont $M|_{\sigma\times\tau}$ est la racine, on a $W = \mathcal{N}(M|_{\sigma\times\tau}) \cap L_0(M)$. On a $|D| \leqslant |L_0(M)|$, et en pratique bien moins dans la majorité des situations.

Enfin, $M|_{\sigma\times\tau}$ peut être au-dessous de $L_0(M)$. Dans ce cas, D est composé de l'unique ancêtre de $M|_{\sigma\times\tau}$ appartenant à $L_0(M)$.

Situation de $M\|_{\sigma\times\tau}$	**Construction de D**	**Nombre**
Sur $L_0(M)$	$D = \{M\|_{\sigma\times\tau}\}$	1
Au-dessus de $L_0(M)$	$D = \mathcal{N}(M\|_{\sigma\times\tau}) \cap L_0(A)$	$\leqslant \|L_0(M)\|$
Au-dessous de $L_0(M)$	$D = \{M\|_{\sigma'\times\tau'} \mid M\|_{\sigma\times\tau} \in \mathcal{N}(M\|_{\sigma'\times\tau'})\}$	1

Table 4.3 – Énumération des dépendances pour un nœud $M|_{\sigma\times\tau}$.

La table 4.3 résume les différentes situations, et la figure 4.16 les illustre. Sur cette figure, les éléments de L_0 (et donc les seules données connues par le moteur d'exécution) sont en gris. Dans la situation de gauche, $M|_{\sigma\times\tau}$ se situe sous L_0, et la dépendance déclarée au moteur d'exécution est alor trouvée en remontant vers la racine. Dans la situation de droite, $M|_{\sigma\times\tau}$ est au-dessus de L_0, et les dépendances déclarées sont trouvées en descendant vers les feuilles jusqu'à L_0.

Cette procédure de construction de D pour un nœud $M|_{\sigma\times\tau}$ quelconque d'une $\mathcal{H}$-Matrice A est notée dans la suite par la fonction EnumerateDeps($M|_{\sigma\times\tau}$). Afin d'alléger l'écriture, nous écrirons également $D(M|_{\sigma\times\tau}) :=$ EnumerateDeps($M|_{\sigma\times\tau}$) dans la suite.

Exclusion Il est aisé de montrer que cette gestion des dépendances assure l'exclusion mutuelle recherchée. On rappelle que lorsqu'un nœud $M|_{\sigma\times\tau}$ d'une $\mathcal{H}$-Matrice est accédé, les accès concurrents interdits (écriture en même temps que lecture ou écriture, lecture en même temps qu'écriture) doivent être forcés pour tous les éléments de $\mathcal{N}(M|_{\sigma\times\tau})$. Deux types d'accès concurrents incompatibles doivent être bloqués :

1. L'accès à un nœud $M|_{\sigma'\times\tau'}$ ancêtre de $M|_{\sigma\times\tau}$;
2. L'accès à un nœud $M|_{\sigma'\times\tau'}$ descendant de $M|_{\sigma\times\tau}$.

$M|_{\sigma'\times\tau'}$ est un ancêtre de $M|_{\sigma\times\tau}$ Ce cas se décompose en deux possibilités :

$M|_{\sigma'\times\tau'}$ est sous $L_0(M)$ Alors $M|_{\sigma\times\tau}$ est également sous $L_0(A)$, et EnumerateDeps renvoie dans les deux cas le même nœud de A pour $M|_{\sigma\times\tau}$ et $M|_{\sigma\times\tau}2$, en « remontant » vers le même élément de $L_0(M)$;

$M|_{\sigma'\times\tau'}$ est au-dessus de $L_0(M)$ Alors $D(M|_{\sigma'\times\tau'})$ est un sur-ensemble de $D(M|_{\sigma\times\tau})$.

$M|_{\sigma'\times\tau'}$ est un descendant de $M|_{\sigma\times\tau}$ On a également deux possibilités :

$M|_{\sigma'\times\tau'}$ est sous $L_0(M)$ Alors $D(M|_{\sigma'\times\tau'})$ est un sous-ensemble de $D(M|_{\sigma\times\tau})$ contenant un unique élément ;

$M|_{\sigma'\times\tau'}$ est au-dessus de $L_0(M)$ Alors $M|_{\sigma\times\tau}$ est également au-dessus de $L_0(M)$, et $D(M|_{\sigma'\times\tau'})$ est un sous-ensemble de $D(M|_{\sigma\times\tau})$.

Dans tous les cas, l'intersection entre $D(M|_{\sigma\times\tau})$ et $D(M|_{\sigma'\times\tau'})$ est non vide, ce qui assure que les relations d'exclusion sont bien respectées. En effet, pour que le moteur d'exécution déclare une tâche comme exécutable, il faut que toutes ses dépendances soient

libres. Une intersection non vide entre les dépendances de deux tâches est donc suffisantw pour garantir le bon séquencement des opérations.

Remarque 4.18 (Dépendances trop grossières)**.** *Il faut noter que dans le cas où la « vraie » dépendance est située sous la L_0 de sa $\mathcal{H}$-Matrice, alors le fait de remonter jusqu'à L_0 implique un verrouillage trop grossier d'une sous-partie de la $\mathcal{H}$-Matrice. En effet, puisque les dépendances sont gérées par le moteur d'exécution au niveau de L_0 uniquement, l'acquisition d'une dépendance sur un élément $M|_{\sigma\times\tau} \in L_0(M)$ est équivalente à l'acquisition d'une dépendance sur tout élément $M|_{\sigma'\times\tau'} \in \mathcal{N}(M|_{\sigma\times\tau})$.*

4.6.4.3 Optimisation des dépendances

Algorithme 4.9 Produit Matrice-Matrice. On suppose ici $\beta = 1$ pour simplifier l'écriture.

```
function CoarseDagGemm(C, α, A, B, σ ∈ T_I, τ ∈ T_J, ρ ∈ T_K)
    if C|σ×τ ∉ L0(C) et ¬IsLeaf(σ × τ) et ¬IsLeaf(σ × ρ) et ¬IsLeaf(ρ × τ) then
        for all (σ', τ', ρ') ∈ S(σ) × S(τ) × S(ρ) do
            CoarseDagGemm(C, α, A, B, σ', τ', ρ')
        end for
    else
        W ← EnumerateDeps(C|σ×τ)
        R ← EnumerateDeps(A|σ×ρ) ∪ EnumerateDeps(B|ρ×τ)
        InsertTask(Gemm, C|σ×τ, W, IN | OUT, α, A|σ×ρ, B|ρ×τ, R, IN)
    end if
end function
```

Avec les conventions de suivi des dépendances introduites dans la section précédente, il est possible d'écrire une version du produit matriciel permettant la composition des opérations. Cette version est donnée par l'algorithme 4.9. Dans cette dernière, le nombre de dépendances en écriture $|W|$ est borné par $|L_0(C)|$, et le nombre de dépendances en lecture $|R|$ est borné par $|L_0(A)| + |L_0(B)|$.

Cependant, toutes les dépendances ne sont pas aussi pénalisantes :

— Un accès à une donnée en lecture ne bloque pas les accès concurrents en lecture ;

— Un accès en écriture bloque les accès concurrents en lecture et écriture.

Il est donc judicieux de minimiser les dépendances en écriture, qui réduisent le parallélisme de façon plus importante que les dépendances en lecture.

Cette observation fait écho à la remarque 4.17. En effet, arrêter la récursion lorsqu'une des opérandes du produit appartient à sa coupe mène à un plus grand nombre de cas de base de cette récursion pour lesquels $C|_{\sigma\times\tau} \notin L_0(C)$. Ces cas nécessitent donc d'avoir plusieurs dépendances en écriture, ce que nous cherchons à éviter.

Une autre motivation pour décomposer des opérations de façon à n'avoir qu'une seule dépendance en écriture est relative à la granularité. En effet, comme le montre la section 3.6.2.2, la majorité du temps de calcul lors d'une décomposition est consacrée à la recompression adaptative des $\mathcal{R}k$-Matrices. Celle-ci intervient lors de la conversion hiérarchique et lors de l'addition, et la taille de cette opération est en rapport avec la taille de la donnée C. En supposant que le choix de la coupe $L_0(C)$ est tel que tous les éléments de

$L_0(C)$ portent des données de même ordre de grandeur en taille, alors imposer aux écritures de ne se faire que dans un seul nœud de $L_0(C)$ tend à égaliser le coût des diverses opérations de base, ce qui est un des objectifs poursuivis. Dans la suite, le but sera donc de minimiser le nombre de dépendances en écriture.

Composition et consistance séquentielle Cette gestion des dépendances conduit donc à aplatir l'algorithme. Ainsi, du point de vue du moteur d'exécution, aucun caractère hiérarchique n'existe dans les opérations, et ceci permet d'adapter un algorithme naturellement récursif au formalisme de graphes de tâches qui ne permet pas de représenter la récursivité. Toutes les opérations sont suivies au travers de la coupe L_0 dans l'arbre de blocs, ce qui assure la composabilité des opérations. Ainsi, si l'assemblage et la factorisation sont constitués de séquences d'appels (non bloquants) à `insert_task()`, la séquence `assemble(M); luDecomposition(M)` avec M une $\mathcal{H}$-Matrice donne le même résultat que l'exécution séquentielle de ces opérations.

4.6.5 Opérations

Nous décrivons ici la déclinaison des principes exposés dans la section précédente pour les opérations supportées par l'implémentation présentée dans le chapitre 3. Nous utilisons la même terminologie que celle de la section 3.3.1, et exposons en premier les opérations de $\mathcal{H}$-Blas niveau 1 et 2, pour poursuivre par le niveau 3 et les opérations de $\mathcal{H}$-Lapack.

4.6.5.1 $\mathcal{H}$-Blas niveau 1 et 2

Les opérations de niveau 1 et 2 (assemblage, addition, produit matrice-vecteur) ont soit une seule opérande de type $\mathcal{H}$-Matrice (pour l'assemblage et le produit matrice-vecteur), soit deux opérandes ayant la même structure (addition). Pour le cas de l'addition, on impose de plus que les deux matrices aient la même coupe, ce qui permet de simplifier l'écriture de l'algorithme.

Algorithme 4.10 Addition $X \leftarrow \alpha X + Y$.

function DagAxpy(X, Y, α, $\sigma \times \tau$)
  if $X|_{\sigma\times\tau} \in L_0(H_1)$ **then**
    InsertTask(Axpy, X, Y, α, $\sigma \times \tau$, $X|_{\sigma\times\tau}$, `IN | OUT`, $Y|_{\sigma\times\tau}$, `IN`)
  else
    for all $(\sigma', \tau') \in S(\sigma \times \tau)$ **do**
      DagAxpy(X, Y, α, $\sigma' \times \tau'$)
    end for
  end if
end function

Dans tous les cas, la récursion se poursuit jusqu'à ce qu'un élément de la coupe soit rencontré. Le cas de base correspond aux algorithmes séquentiels, et le nombre de dépendances de chaque opération est de 1 pour l'assemblage et le produit matrice-vecteur, et de deux pour l'addition, avec une dépendance en lecture et une en écriture.

L'algorithme 4.10 donne le modèle de ces opérations, pour l'addition. On note que du fait du caractère non bloquant de l'appel `insert_task()`, le temps d'exécution de la fonction DAGAXPY est très court, proportionnel au temps de parcours de l'arbre, avec en plus le temps d'insertion des tâches. Ce temps dépend du moteur d'exécution, mais un ordre de grandeur de quelques μs par appel est raisonnable sur les machines actuelles. Il existe quelques spécificités par rapport à ce modèle :

Assemblage symétrique Le cas de l'assemblage est spécifique. Dans certains cas, on souhaite travailler sur une matrice symétrique (issue d'une formulation EFIE par exemple) comme si elle ne l'était pas (par exemple une décomposition LU). L'assemblage est alors modifié pour assembler la partie triangulaire inférieure de la matrice, puis copier celle-ci dans la partie triangulaire supérieure en transposant. Afin de conserver la simplicité des algorithmes de cette section, on impose alors la règle suivante dans la construction de $L_0(M)$, avec M la $\mathcal{H}$-Matrice à assembler :

$$\forall M|_{\sigma\times\tau} \in L_0(M),\ M|_{\tau\times\sigma} \in L_0(M)$$

Algorithme 4.11 Produit Matrice-vecteur

function DAGGEMV(H, x, y, $\sigma \times \tau$, α, β)
  if $\beta \neq 1$ **then**
    for all $y|_\sigma \in L_0(y)$ **do**
      INSERTTASK(SCALE, β, $y|_\sigma$, `IN | OUT`)
    end for
    $\beta \leftarrow 1$
  end if
  if $y|_\sigma \in L_0(y)$ ou ISLEAF($H|_{\sigma\times\tau}$) **then**
    $R \leftarrow$ ENUMERATEDEPS(x_τ) $\cup$ ENUMERATEDEPS($H_{\sigma\times\tau}$)
    $W \leftarrow$ ENUMERATEDEPS(y_σ)
    INSERTTASK(GEMV, $H|_{\sigma\times\tau}$, $x|_\tau$, $y|_\sigma$, α, β, R, `IN`, W, `IN | OUT`)
  else
    for all $(\sigma', \tau') \in S(\sigma \times \tau)$ **do**
      DAGGEMV(H, x, y, $\sigma' \times \tau'$, α, β)
    end for
  end if
end function

Produit matrice-vecteur Le produit matrice-vecteur (H-GEMV, section 3.3.2.4 page 95) est différent, puisqu'il manipule à la fois une $\mathcal{H}$-Matrice en lecture, et des vecteurs en lecture et écriture. En effet, dans l'opération

$$y \leftarrow \alpha H x + \beta y$$

avec x et y des vecteurs et H une $\mathcal{H}$-Matrice, H et x sont lus et y est écrit. Il est possible d'utiliser une réduction comme dans la section précédente, et de ne suivre que les accès en lecture sur H.

Il est également possible de définir un $\mathcal{H}$-Vecteur, associé à un arbre de groupes. La définition d'un $\mathcal{H}$-Vecteur est similaire à la définition 3.15 d'une $\mathcal{H}$-Matrice :

Définition 4.19 ($\mathcal{H}$-Vecteur)**.** *Soit $M \in \mathbb{C}^{N \times p}$ une matrice, I un ensemble d'indices tels que $|I| = N$, et $\mathcal{T}_I$ un arbre de groupes construit sur I. On appelle $\mathcal{H}$-Vecteur l'arbre de groupes $\mathcal{T}_I$ dont les feuilles $\sigma \subset I$ sont augmentées par $M|_\sigma \in C^{|\sigma| \times p}$.*

Un $\mathcal{H}$-Vecteur est donc un vecteur à plusieurs colonnes (donc une matrice, mais manipulée comme un vecteur), et n'est découpé que selon les lignes, suivant le même arbre que la $\mathcal{H}$-Matrice par laquelle il est multiplié.

Remarque 4.20 (Vecteurs multi-colonnes)**.** *On considère un vecteur à plusieurs colonnes pour des raisons d'efficacité informatique. Il est plus efficace d'effectuer une seule opération* H-GEMV *avec un vecteur à plusieurs colonnes qu'une succession d'opérations avec de vrais vecteurs.*

On définit ensuite une coupe L_0 dans ce $\mathcal{H}$-Vecteur (qui est un arbre binaire), typiquement à un niveau fixe, et la récursion s'interrompt au niveau de la coupe dans le vecteur.

C'est cette approche qui a été retenue ici, et l'algorithme 4.11 la décrit. Dans cet algorithme, le nombre de dépendances en lecture et en écriture n'est pas fixe. Néanmoins, le produit matrice-vecteur est une opération comparativement rapide, il est donc courant de fixer la coupe dans les vecteurs assez haute. Dans ce cas, si la récursion s'interrompt avant d'arriver aux feuilles de H, alors il n'y a qu'une dépendance en écriture. Enfin, il est possible d'utiliser les extensions des moteurs d'exécution pour spécifier que l'accès au vecteur y est une réduction, ce qui diminue les dépendances.

4.6.5.2 Produit matriciel

On rappelle qu'il est possible de satisfaire la condition d'arrêt (4.6.2) de la récursion de l'algorithme 4.9 de deux manières :

1. La condition $C|_{\sigma \times \tau} \in L_0(C)$ est rencontrée avant le cas de base de l'algorithme séquentiel (c'est-à-dire, la seconde partie de cette condition). Le cas de base de la récursion de l'algorithme par tâches n'est alors pas un cas de base de l'algorithme séquentiel, et il est possible d'insérer une tâche séquentielle sans autre difficulté.
2. Le cas de base de l'algorithme séquentiel est rencontré avant l'élément de $L_0(C)$. Il n'est pas possible de poursuivre la récursion, et il faut traiter cette situation spécialement. En effet, insérer une tâche séquentielle comme cela est fait dans l'algorithme 4.9 nécessite plusieurs dépendances en écriture sur C, ce que nous souhaitons éviter.

Premier cas On a $C|_{\sigma \times \tau} \in L_0(C)$, et non nécessairement $A|_{\sigma \times \rho} \in L_0(A)$, ni $B|_{\rho \times \tau} \in L_0(B)$. Les dépendances en lecture portant sur A et B sont énumérées (comme illustré par la figure 4.16) dans un ensemble R, tel que $2 \leqslant |R| \leqslant |L_0(A)| + |L_0(B)|$, et une tâche élémentaire est insérée par

`insert_task`(GEMM, $C|_{\sigma\times\tau}$, `IN | OUT`, $\{R_1, \ldots\}$, `IN`, $A|_{\sigma\times\rho}$, $B|_{\rho\times\tau}$)

En pratique, la borne sur la taille de R est bien trop large, puisqu'elle correspond au cas où $C|_{\sigma\times\tau}$ est la racine de C, ce qui ne se produit que dans le cas où les trois arbres A, B et C ne comportent qu'un seul nœud (et donc $|L_0| = 1$ pour ces trois $\mathcal{H}$-Matrices).

Second cas Ici, $C|_{\sigma\times\tau}$ est au-dessus de $L_0(C)$; $A|_{\sigma\times\rho}$ et $B|_{\rho\times\tau}$ peuvent être au-dessus, sur ou sous $L_0(A)$ et $L_0(B)$, respectivement. La récursion ne peut se poursuivre car un cas de base de l'algorithme séquentiel est atteint. La solution la plus simple consiste à insérer une tâche élémentaire de produit comme précédemment, avec un ensemble de dépendances en lecture W au lieu d'une seule dépendance comme dans l'algorithme 4.9. Ce choix est pénalisant (*cf.* la section précédente), et on découpe alors l'opération en deux parties :

— Calcul du produit dans une variable temporaire : $X \leftarrow A|_{\sigma\times\rho}B|_{\rho\times\tau}$.
— Addition de la variable temporaire à $C|_{\sigma\times\tau}$: $C|_{\sigma\times\tau} \leftarrow C|_{\sigma\times\tau} + X$.

Décomposition du cas de base Ce choix requiert la création d'une variable temporaire, qui est nécessairement une feuille, compressée ou non. En effet, la récursion est stoppée si une des opérandes est une feuille, et $C|_{\sigma\times\tau}$ n'est pas une feuille, car c'est un nœud au-dessus de $L_0(C)$, donc interne. $A|_{\sigma\times\rho}$ ou $B|_{\rho\times\tau}$ est donc une feuille, et le résultat est une feuille. L'algorithme séquentiel nécessite également la création d'une variable temporaire dans ce cas, et il n'y a donc pas ici de pénalité en terme de mémoire ou temps de calcul.

Cette séparation permet également d'augmenter le degré de parallélisme de l'algorithme. En effet, la première opération n'a que des dépendances en lecture, puisque la feuille temporaire est créée par cette opération. L'addition de la feuille temporaire est elle-même décomposée sur tous les éléments de L_0 sous $C|_{\sigma\times\tau}$, et chacune de ces tâches a exactement une dépendance en écriture sur un élément de $L_0(C)$, et une dépendance en lecture sur la feuille temporaire. Les dépendances en lecture ne sont alors nécessaires que pour la première partie, ce qui les libère pour des écritures ultérieures, et les tâches d'addition n'ont qu'une seule dépendance, ce qui limite le nombre de contraintes dans le graphe de tâches.

Nouvelles tâches élémentaires On introduit deux nouvelles opérations élémentaires :

PRODSCALE Effectue l'opération $X \leftarrow AB$

> **Lecture** A, B
>
> **Écriture** X
>
> **Structure** X est une feuille, A et/ou B en est une.

ADDLEAF Effectue l'opération $M|_{\sigma\times\tau} \leftarrow M|_{\sigma\times\tau} + X|_{\sigma\times\tau}$ où $X|_{\sigma\times\tau}$ est une restriction de X a l'ensemble d'indices $\sigma \times \tau$. L'ensemble d'indices de X est un sur-ensemble strict de $\sigma \times \tau$.

> **Lecture** X
>
> **Écriture** $M|_{\sigma\times\tau}$
>
> **Structure** X est une feuille

À l'aide de ces deux opérations et des précédentes, il est possible de donner une description de la multiplication sans synchronisation globale par l'algorithme 4.12.

Algorithme 4.12 Multiplication matricielle, sans synchronisation globale.

function DagGemm(C, α, A, B, β, $\sigma \in \mathcal{T}_I$, $\tau \in \mathcal{T}_J$, $\rho \in \mathcal{T}_K$)
  if $\beta \neq 1$ **then**
    for all $C|_{\sigma' \times \tau'} \in L_0(C)$ **do**
      InsertTask(Scale, $C|_{\sigma' \times \tau'}$, `IN | OUT`, α)
    end for
  end if
  if $C|_{\sigma \times \tau} \notin L_0(C)$ et $\neg$IsLeaf($\sigma \times \tau$) et $\neg$IsLeaf($\sigma \times \rho$) et $\neg$IsLeaf($\rho \times \tau$) **then**
    for all $(\sigma', \tau', \rho') \in S(\sigma) \times S(\tau) \times S(\rho)$ **do**
      DagGemm(C, A, B, σ', τ', ρ')
    end for
  else
    $R \leftarrow$ EnumerateDeps($A|_{\sigma \times \rho}$) $\cup$ EnumerateDeps($B|_{\rho \times \tau}$)
    if $C|_{\sigma \times \tau} \in L_0(C)$ **then**
      InsertTask(`Gemm`, $C|_{\sigma \times \tau}$, `IN | OUT`, α, $A|_{\sigma}$, $B|_{\rho \times \tau}$, 1, $\{R_1, \ldots\}$, `IN`)
    else
      $W \leftarrow$ EnumerateDeps($C|_{\sigma \times \tau}$)
      $X \leftarrow 0$ ▷ (1)
      InsertTask(ProdScale, X, `OUT`, $A|_{\sigma \times \rho}$, $B|_{\rho \times \tau}$, $\{R_1, \ldots\}$, `IN`)
      for all $C|_{\sigma' \times \tau'} \in W$ **do**
        InsertTask(AddLeaf, $C|_{\sigma' \times \tau'}$, `IN | OUT`, X, `IN`)
      end for
      InsertTask(Delete, X, `IN | OUT`)
    end if
  end if
end function

Gestion de la mémoire La gestion de la mémoire liée à la variable temporaire X n'est pas présentée dans cet algorithme. En effet, les appels à `insert_task()` étant non bloquants, l'allocation de cette donnée ligne (1) lors de l'insertion des tâches mène à une surconsommation mémoire, puisque toutes les matrices temporaires sont allouées au début de l'algorithme, ce qui n'est pas le cas dans l'algorithme séquentiel. De même, la libération de cette matrice ne peut pas être faite dans l'algorithme 4.12 puisque les tâches opérant dessus ne sont pas encore exécutées.

L'allocation de la matrice temporaire peut se faire dans la tâche ProdScale. De même, il est possible d'ajouter une tâche de suppression

InsertTask(Delete, X, `IN | OUT`)

après que les tâches AddLeaf ont été ajoutées, la gestion des contraintes assurant le bon séquencement de cette suppression.

4.6.5.3 Systèmes triangulaires matriciels

Une fois la présentation des opérations de niveaux 1 et 2 effectuées, ainsi que celle du produit matricielle, l'opération d'intérêt est la décomposition LU. Celle-ci utilise la résolution de systèmes triangulaires matriciels, qui est l'objet de cette section.

La résolution d'un système triangulaire est semblable au produit matriciel du point de vue des dépendances et de la récursion. Dans le cas de la résolution du système $LX = B$ avec L, X et B des $\mathcal{H}$-Matrices (et X écrasant B), l'arrêt de la récursion dans le cas séquentiel correspond à la situation où L ou B est une feuille. De la même façon que pour la multiplication, à cette condition s'ajoute celle portant sur l'appartenance des opérandes à leurs coupes L_0 respectives, c'est-à-dire que la condition d'arrêt dans l'algorithme 3.13 page 103 est remplacée par :

$$B \in L_0 \tag{4.6.3}$$

Cette condition semble au premier abord trop faible, puisqu'elle ne porte en particulier pas sur L. Le cas posant problème est :

***L* est une feuille, *B* au-dessus de L_0** Ce cas n'est pas possible, car L est de la forme $L|_{\sigma\times\sigma}$. En effet, seule la décomposition d'une matrice carrée est considérée (construite sur un arbre de blocs de la forme $\mathcal{T}_{I\times I}$), et les résolutions triangulaires inférieures sont toujours appelées sur un fils diagonal de la matrice à décomposer, donc sur une matrice ayant les mêmes ensembles d'indices ligne et colonne. Le cas où L est une feuille conduit nécessairement à une feuille pleine, avec σ une feuille de l'arbre de groupes $\mathcal{T}_I$, et X est de la forme $X|_{\sigma\times\tau}$, X étant construite sur l'arbre de blocs $\mathcal{T}_{I\times J}$. Puisque σ est une feuille de $\mathcal{T}_I$, alors $\sigma \times \tau$ est une feuille de $\mathcal{T}_{I\times J}$. Une feuille étant toujours sous ou dans la coupe L_0, ce cas est ainsi exclu.

La condition (4.6.3) est donc suffisante pour garantir le bon arrêt de la récursion, et le cas de base de celle-ci correspond à l'exécution de l'algorithme séquentiel, ce qui permet de donner l'expression de la résolution triangulaire par l'algorithme 4.13, lequel nécessite l'opération élémentaire supplémentaire suivante :

SOLVELOWERLEAF Résout le système $LX = B$, X écrasant B
- **Lecture** L, B
- **Écriture** B
- **Structure** B est une feuille

Algorithme 4.13 Résolution de $LX = B$, sans synchronisation globale.

```
function DAGSOLVELOWER(L, B)
    if B ∈ L0 then
        R ← ENUMERATEDEPS(L)
        INSERTTASK(SOLVELOWERLEAF, L, {R1, ...}, IN, X, IN|OUT)
    else
        DAGSOLVELOWER(L11, B11)
        DAGSOLVELOWER(L11, B12)
        DAGGEMM(B21, −1, L21, B11, 1)
        DAGSOLVELOWER(L22, B21)
        DAGGEMM(B22, −1, L21, B12, 1)
        DAGSOLVELOWER(L22, B22)
    end if
end function
```

Les tâches de résolutions triangulaires des cas de base ont toujours une seule dépendance en écriture, laquelle n'est pas artificiellement trop grossière (puisque B ne peut pas

être sous L_0), et un nombre de dépendances en lecture borné par $|L_0|$, mais en pratique bien plus réduit. On observe par ailleurs que les appels à la fonction DagGemm ne sont accompagnés d'aucune synchronisation explicite. La résolution des systèmes triangulaires supérieurs étant similaire, elle ne sera pas détaillée.

Remarque 4.21 (Dépendances en lecture)**.** *Dans l'algorithme 4.13, le cas d'arrêt de l'algorithme peut se faire au-dessous ou sous la coupe de la matrice* L*. Dans le second cas, la dépendance en lecture est trop grossière. Cette situation est en pratique rare, car la matrice* L *est un bloc diagonal de la forme* $\sigma \times \sigma$*, et tend donc à contenir une quantité de données plus importante qu'un bloc extra-diagonal. De ce fait, il est probable que l'heuristique de détermination de la coupe* L_0 *mène à une coupe plus basse près de la diagonale. Dans ce cas, il est également probable que la dépendance soit en réalité au-dessus de la coupe pour le cas de base de la récursion.*

4.6.5.4 Inversion et décomposition *LU*

Algorithme 4.14 Inversion d'une $\mathcal{H}$-Matrice, sans synchronisation globale.

```
function DagInverse(A, X)
    if A ∈ L_o then
        InsertTask(InverseLeaf, A, IN|OUT, X, OUT)
    else
        DagInverse(A11, X11)
        DagGemm(X12, −1, M11, M12, 0)
        DagGemm(X21, 1, M21, M11, 0)
        DagGemm(M22, 1, M21, X12, 1)
        DagInverse(M22, X22)
        DagGemm(M12, 1, X12, M22, 0)
        DagGemm(M11, −1, M12, X21, 1)
        DagGemm(M21, −1, M22, X21, 0)
    end if
end function
```

Dans la classification des opérations de la section 3.3, il reste à traiter les opérations de $\mathcal{H}$-Lapack, la décomposition *LU* et l'inversion sur place d'une $\mathcal{H}$-Matrice. Ces opérations ont des implémentations très similaires aux algorithmes séquentiels correspondants, le cas de base de la récursion étant remplacé par l'appartenance de l'opérande à L_0, et se traduit par l'exécution de l'algorithme séquentiel, comme exposé par les algorithmes 4.14 et 4.15, similaires aux algorithmes 3.11 et 3.12, respectivement. Notons que pour l'inversion, on suppose que les coupes dans les matrices M et X sont les mêmes, ce qui est cohérent, ces matrices partageant le même arbre de blocs.

La simplicité de ces implémentations est confortable, car la séparation entre divers niveaux d'opérations est conservée, sans propagation entre les niveaux. Ainsi, la gestion des dépendances par l'algorithme est locale, et les détails d'implémentations de DagGemm sont indifférents pour l'implémentation de DagInverse ou DagLu.

Algorithme 4.15 Décomposition LU, sans synchronisation globale.

```
function DAGLU(M)
    if M ∈ L0 then
        INSERTTASK(LULEAF, A, IN|OUT)
    else
        DAGLU(M11)
        DAGSOLVELOWER(L11, M12)
        DAGSOLVEUPPER(U11, M21)
        DAGGEMM(M22, −1, L21, U12)
        DAGLU(M22)
    end if
end function
```

4.6.5.5 Résolution : Descente et remontée

Une fois la matrice factorisée (LU ici), la résolution du système linéaire nécessite une étape de descente et une de remontée, c'est-à-dire la résolution de systèmes triangulaires inférieurs et supérieurs, mais dont le second membre est un vecteur, et non une $\mathcal{H}$-Matrice.

Parallélisation sur les seconds membres Dans le cas des éléments finis de frontière, ces seconds membres sont bien souvent nombreux. Ceci est d'autant plus vrai que le champ d'application particulièrement visé par un solveur direct rapide concerne les problèmes ayant un grand nombre de seconds membres, puisque la FMM est pénalisée dans ce cas. La résolution de chaque second membre étant indépendante des autres, le problème est aisément parallélisable. Il suffit en effet de créer une tâche élémentaire de résolution par second membre, dont les dépendances en lecture portent sur l'intégralité de la $\mathcal{H}$-Matrice (pour un nombre de dépendances égal à $|L_0|$), et ayant une unique dépendance en écriture sur le second membre considéré.

Cette approche a plusieurs inconvénients :

- La résolution d'un système triangulaire introduit une synchronisation globale à la fin de la factorisation ;
- Les cas n'ayant qu'un seul second membre sont traités de manière séquentielle ;
- L'efficacité de la résolution est plus faible.

La perte d'efficacité informatique lors de la résolution à un seul second membre est liée à l'utilisation d'opérations BLAS2 dans ce cas, au lieu d'opérations BLAS3 dont l'efficacité est supérieure.

Parallélisation fine Ceci invite à utiliser une approche similaire à celle employée pour le produit et les autres opérations de niveau supérieur, et en particulier pour l'opération H-GEMV, section 4.6.5.1.

On définit une coupe $L_0(V)$ dans un $\mathcal{H}$-Vecteur V de la même manière que pour une $\mathcal{H}$-Matrice, et la parallélisation utilise les éléments de $L_0(V)$ de la même façon. La figure 4.17 compare les deux méthodes de découpage des seconds membres. La première suit la méthode naïve de découpage évoquée au début de cette section, c'est-à-dire que

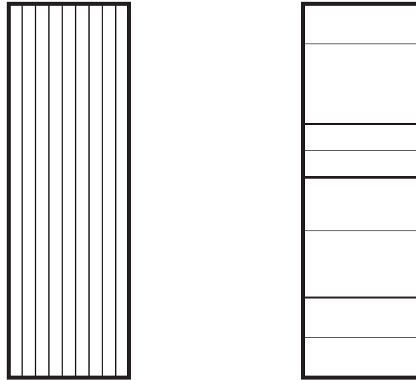


FIGURE 4.17 – Découpage des seconds membres.

Algorithme 4.16 Résolution de $Lx = b$.

function DAGSOLVELOWER(M, b, σ)
  if $b|_\sigma \in L_0(b)$ **then**
    $R \leftarrow$ ENUMERATEDEPS(M)
    INSERTTASK(SOLVELOWER, M, σ, R, `IN`, $b|_\sigma$, `IN | OUT`)
  else
    $S(\sigma) = \{\sigma_1, \sigma_2\}$
    DAGSOLVELOWER(M, b, σ_1)
    DAGGEMV(M, b, b, $\sigma_2 \times \sigma_1$, -1, 1)
    DAGSOLVELOWER(M, b, σ_2)
  end if
end function

chaque second membre est traité séparément, et chaque résolution est séquentielle. La seconde représente le découpage selon un arbre de groupes, ce qui explique le caractère irrégulier des divisions, puisque les nœuds de la coupe $L_0(V)$ ne sont pas nécessairement de tailles égales.

L'algorithme 4.16 de résolution des systèmes triangulaires, identique dans le principe à l'algorithme 4.13, a les propriétés suivantes :

- Absence de synchronisation globale ;
- Parallélisation de la résolution, même dans le cas où il n'y a qu'un seul second membre ;
- Utilisation de BLAS3 pour toutes les opérations (vecteurs à plusieurs colonnes).

Remarque 4.22 (Granularité)**.** *La résolution de système étant moins coûteuse que la factorisation, la coupe $L_0(V)$ du $\mathcal{H}$-Vecteur des seconds membres comporte en général moins de nœuds que la coupe de la $\mathcal{H}$-Matrice d'interaction, afin de conserver une granularité adaptée. De ce fait, le parallélisme obtenu est moins important, et il peut alors être judicieux dans le cas d'un grand nombre de seconds membres d'adopter une approche « mixte ».*

Parallélisation mixte L'avantage de la parallélisation second membre par second membre est sa simplicité, et son grand degré de parallélisme lorsque le nombre de second membres

devient grand. D'un autre côté, la deuxième approche permet d'atteindre une efficacité informatique plus élevée par l'utilisation de BLAS3, avec un degré de parallélisme plus limité.

Dans le cas où le nombre de seconds membres est élevé, il est utile de combiner les deux approches. Ainsi, supposons une $\mathcal{H}$-Matrice M de taille $N \times N$, et un vecteur (à plusieurs colonnes) de seconds membres de $b \in \mathbb{C}^{N \times n_{rhs}}$. Au lieu de créer un $\mathcal{H}$-Vecteur représentant ce vecteur, il est possible de créer q groupes de n_{rhs}/q seconds membres, c'est-à-dire un ensemble $\{b_1^h, \dots, b_q^h\}$ de $\mathcal{H}$-Vecteurs représentant chacun une partie des seconds membres.

Ce découpage « dans les deux directions » des seconds membres permet de conserver l'efficacité informatique, tant que n_{rhs}/q est assez grand (typiquement, de l'ordre de 100), tout en laissant une plus grande liberté au moteur d'exécution, puisque chaque $\mathcal{H}$-Vecteur engendre un graphe de tâches indépendant des autres. L'utilisation de ce mode de parallélisation prend la forme suivante :

for all $i = 1, \dots, q$ **do**
  DagSolveLower(M, b_i^h)
end for
WaitForAll()

4.7 Implémentation et performances

Cette section détaille l'implémentation des algorithmes de la section précédente, au-dessus de deux moteurs d'exécution, QUARK et StarPU. Elle concerne l'optimisation de l'implémentation, avec en particulier des heuristiques de choix de la coupe L_0, du nombre de nœuds de celle-ci, et des différents algorithmes d'ordonnancement disponibles, en particulier dans le cas de StarPU.

4.7.1 Implémentation

L'implémentation suit les algorithmes de la section précédente. Un calcul par éléments finis de frontière comprend les étapes suivantes : assemblage de la matrice, factorisation, assemblage des seconds membres et résolution des systèmes linéaires. Dans le cadre des algorithmes de la section précédente, ces étapes se déroulent sans synchronisation globale, et sont les suivantes :

1. Lecture des coordonnées des degrés de liberté, construction de l'arbre de groupes et de l'arbre de blocs du problème.
2. Création de la $\mathcal{H}$-Matrice à partir de l'arbre de groupes. Détermination de la coupe $L_0(M)$ de cette matrice.
3. Création du $\mathcal{H}$-Vecteur des seconds membres et détermination de sa coupe $L_0(B)$.
4. Assemblage de M, B, factorisation de M, résolution de $LU = B$.
5. Attente de la fin des tâches et libération des données.

Les trois première étapes sont séquentielles, mais de coût négligeable par rapport à la suivante. La quatrième étape comprend l'intégralité du calcul dans un seul graphe de

tâches, mais ne consiste qu'en l'insertion des tâches dans le moteur d'exécution, l'essentiel du temps pour le *thread* principal étant un état de « sommeil » dans la fonction `wait_all_tasks()` définie dans l'exemple 4.4.2.

Dans la suite de cette section, nous utilisons les outils de StarPU permettant d'obtenir des informations sur l'exécution d'un algorithme avec ce moteur. En particulier, StarPU permet de générer un diagramme de Gantt de l'exécution, de visualiser le graphe de tâches, et d'obtenir des informations sur le temps supplémentaire rajouté par l'utilisation de l'ordonnanceur.

4.7.1.1 Choix de L_0

Un paramètre essentiel de l'algorithme parallèle est le choix de la coupe L_0 dans la $\mathcal{H}$-Matrice à factoriser. Les deux facteurs importants sont (a) le nombre de nœuds de cette coupe, et (b) le tracé de celle-ci. Le nombre de nœuds conditionne fortement le nombre de tâches qui seront soumises à l'ordonnanceur. Le placement de la ligne l'influence également, mais a surtout un impact sur la granularité des tâches.

Tout comme pour l'ordonnancement par liste, le cas idéal pour un moteur d'exécution est de disposer de tâches dont la durée d'exécution est égale, autant que possible. Ceci permet à la fois d'obtenir de bons résultats même avec un ordonnancement naïf comme la politique `eager` de StarPU (et de minimiser le nombre de vols de tâches pour QUARK), mais également de diminuer l'impact du moteur d'exécution, dont le coût est indépendant de la durée d'exécution de la tâche.

L'objectif est donc d'avoir un nombre suffisant de tâches pouvant s'exécuter en parallèle, ayant une durée la plus proche possible, et avec un nombre minimal de tâches. Le premier point permet de maximiser l'occupation des processeurs, le second d'optimiser l'ordonnancement, et le dernier de diminuer le coût induit par le moteur d'exécution.

Malheureusement, la détermination du coût des tâches ne peut être qu'heuristique dans le cas des $\mathcal{H}$-Matrices, et dans le but de permettre l'utilisation de plusieurs heuristiques différentes, nous présentons la méthode générale permettant de placer la ligne L_0, en supposant le nombre maximal de nœuds de L_0 fixe, et égal à n_{max}.

On suppose l'existence d'une fonction $f : \mathcal{T}_{I \times J} \to \mathbb{R}_+$ associant à un nœud de l'arbre de groupes un score représentant une estimation de la quantité de travail associé à cette feuille. On note que cette fonction opère sur un arbre de blocs et non sur une $\mathcal{H}$-Matrice. En effet, l'estimation du coût se fait avant l'assemblage de la $\mathcal{H}$-Matrice. Les algorithmes présentés dans la section précédente supposent la préexistence de la coupe L_0, et l'assemblage et la factorisation faisant partie d'un seul graphe de tâches, la détermination de L_0 doit se faire sans la connaissance de la $\mathcal{H}$-Matrice assemblée.

La détermination de la coupe est ensuite en deux étapes, explicitées par l'algorithme 4.17 :

1. Détermination du score limite en fonction de la valeur de n_{max} ;
2. Détermination de la coupe L_0 en fonction du score limite.

Le coût de calcul de la coupe est dépendant du coût d'évaluation de CountNodes, qui est à son tour dépendant du coût de calcul de f. Ce calcul n'est cependant en pratique jamais significatif dans le temps d'exécution total.

Algorithme 4.17

function COUNTNODES($(\sigma, \tau) \in \mathcal{T}_{I \times J}$, f, $t \in \mathbb{R}_+$)
  if $f(\sigma, \tau) \leqslant t$ ou $S(\sigma \times \tau) = \varnothing$ **then**
    return 1
  else
    $n \leftarrow 0$
    for all $(\sigma', \tau') \in S(\sigma \times \tau)$ **do**
      $n \leftarrow n+$ COUNTNODES((σ', τ'), f, t)
    end for
    return n
  end if
end function
function FINDL0($(\sigma, \tau) \in \mathcal{T}_{I \times J}$, f, n_{max})
  $h \leftarrow f(\sigma, \tau)$
  $l \leftarrow 0$
  Recherche par dichotomie sur la valeur de t (frontière) en fonction de COUNTNODES((σ, τ), f, t)
  Marquage des nœuds de $\mathcal{T}_{I \times J}$ en fonction de la valeur limite t
end function

Dans le cas d'une matrice pleine, le coût de calcul est guidé par la taille d'un bloc, et une bonne approximation de celui-ci est alors $f(\sigma, \tau) := |\sigma||\tau|$. Cette fonction est très rapide à calculer, et tend à produire une coupe « droite » dans l'arbre quand cela est possible (c'est-à-dire quand il est possible de descendre au niveau de la coupe sans rencontrer de feuille auparavant) dans le cas d'arbres de groupes médians.

Ce choix n'est cependant pas nécessairement optimal pour les $\mathcal{H}$-Matrices. En effet, la section 3.4 montre que la complexité des opérations sur les $\mathcal{H}$-Matrices est très fortement dépendante du nombre de colonnes k conservées dans les $\mathcal{R}k$-Matrices. Pour l'addition, ce coût a une croissance en $\mathcal{O}(k^3)$ (section 2.4.2.3).

Par ailleurs, la figure 3.45 montre une tendance à la croissance du nombre k de colonnes dans les $\mathcal{R}k$-Matrices avec l'augmentation de la valeur de $\eta_{crit}(\sigma, \tau)$ (3.6.1). Il est donc possible de rajouter ce critère pour donner $f(\sigma, \tau) := \eta_{crit}(\sigma, \tau)|\sigma||\tau|$, dont le coût d'évaluation est toujours $\mathcal{O}(1)$.

Une autre stratégie suggère d'abaisser le niveau de la coupe sur la diagonale de la $\mathcal{H}$-Matrice. Celle-ci est d'une part moins bien compressée en général, et d'autre part un point de sérialisation dans les algorithmes. On peut alors multiplier le score par un facteur β dans le cas où $\sigma = \tau$ dans f à cet effet.

Exemples On considère deux fonctions f, et la factorisation LU (avec assemblage) d'une $\mathcal{H}$-Matrice. La première est $f(\sigma, \tau) := |\sigma||\tau|$, et la seconde est « alourdie » d'un facteur 2 pour $\sigma = \tau$. Le cas de calcul est le même que celui de la figure 3.8, et dont la $\mathcal{H}$-Matrice est représentée sur la figure 3.11. Il comporte 9852 inconnues. La figure 4.18 montre les limites des éléments de L_0 pour ces deux choix, et la figure 4.19 les graphes de tâches résultants.

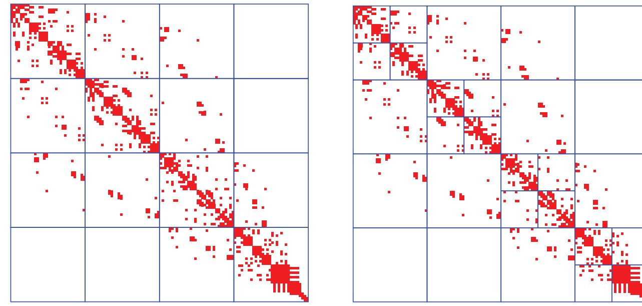


Figure 4.18 – Structures des nœuds de L_0 pour et DAG pour f naïve et « alourdie » (9852 inconnues). Seules les feuilles non compressées ont été représentées sur cette figure en rouge. Les éléments de L_0 sont en bleu.

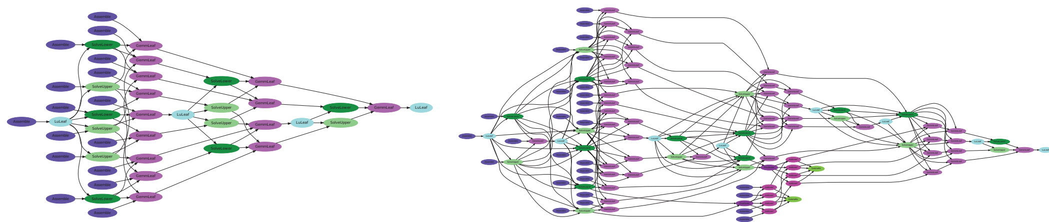


Figure 4.19 – Matrice et DAG pour f naïve et « alourdie » (9852 inconnues).

$\mathbf{\vert L_0\vert}$	796	1528	2584	3604	6892	7948	8062
Nombre de tâches	9289	19612	40274	58982	137687	160405	163357
Temps (s)	1343,1	1033,3	786,0	635,5	582,3	583,2	574,3

Table 4.4 – Influence de $|L_0|$ sur le calcul (290.030 inconnues, 12 cœurs).

Dans les deux cas, le nombre maximal de nœuds dans L_0 a été fixé à 32, cependant dans la première configuration, $|L_0| = 16$ et dans la seconde, $|L_0| = 28$. Ceci est une conséquence de la détermination des éléments de la coupe, puisque dans les deux cas de nombreux nœuds de la $\mathcal{H}$-Matrice ont le même score (ce qui est exacerbé par le découpage médian). Le graphe de tâches est donc plus fourni pour le deuxième exemple.

La recherche d'une bonne fonction de coût pour la localisation de L_0 est un sujet d'importance pour le passage à l'échelle des algorithmes. Toutefois, les résultats du reste de ce chapitre seront donnés avec la fonction du coût naïve, sauf indication contraire.

4.7.1.2 Nombre de tâches

Le nombre total de nœuds dans le graphe de tâches est fortement croissant en fonction du nombre d'éléments dans L_0. Il est également dépendant de la « forme » de L_0. Dans le cas de l'algèbre linéaire dense, en utilisant une représentation par tuiles de la matrice de taille $n_{tile} \times n_{tile}$, le nombre de tâches grandit avec n_{tile}^3 pour le produit matriciel. En effet, les opérations sur les tuiles sont similaires aux opérations sur les éléments d'une matrice de taille $n_{tile} \times n_{tile}$.

Si la coupe L_0 est déterminée avec la fonction f naïve présentée ci-dessus, et que tous

les éléments de cette coupe sont au même niveau de l'arbre (c'est-à-dire qu'aucune feuille n'est rencontrée avant le niveau de la coupe), alors la situation est similaire, et on peut s'attendre à une croissance en $\mathcal{O}(|L_0|^{3/2})$. Cependant, cette situation ne se produit pas en pratique, et la coupe n'est que très rarement de profondeur uniforme. Ceci a le potentiel de rajouter des tâches élémentaires ou d'en retirer, et l'évolution du nombre de tâches avec $|L_0|$ est plus difficile à anticiper.

Pour un calcul d'électromagnétisme à 290.030 inconnues comprenant assemblage et factorisation LU (le même que celui de la section 2.4), la variation du nombre de tâches avec $|L_0|$ est représentée sur le tableau 4.4. Il s'agit d'un calcul EFIE, donc le nombre de tâches d'assemblage est environ $|L_0|/2$ puisque la matrice est symétrique. Le temps de calcul est donné à titre indicatif pour une machine à 12 processeurs. On constate clairement une tendance à la diminution du temps de calcul avec l'augmentation du nombre de tâches, qui se traduit par un élargissement du graphe de tâches.

4.7.1.3 Ordonnancement

Dans le cas de StarPU, il est possible de choisir la politique d'ordonnancement pour les tâches. Dans ce chapitre, nous ne considérons que les algorithmes d'ordonnancement les plus simples, ne nécessitant pas l'ajout d'indications supplémentaires complexes au moteur d'exécution.

StarPU définit une interface générique permettant d'implémenter divers ordonnanceurs [18]. Cette interface comprend deux opérations, `push` et `pop`. L'opération `push` est effectuée quand une tâche est prête pour l'exécution, c'est-à-dire lorsque toutes ses dépendances sont satisfaites dans le graphe de tâches. Chaque processeur exécute l'opération `pop` pour demander une tâche à l'ordonnanceur. L'algorithme d'ordonnancement le plus simple (`eager`) utilise une file FIFO, laquelle fournit nativement ces opérations. Dans le cas de l'ordonnanceur `prio`, une file d'attente par priorité est définie, et une tâche est dirigée vers la file d'attente correspondant à sa priorité lors de l'opération `push`. L'opération `pop` renvoie alors la tâche en tête de la file d'attente de priorité la plus élevée. Ces différences sont illustrées par la figure 4.7.

Du fait de cette construction et de la taille des graphes de tâches considérés dans ce chapitre, il est important de fournir des indications à l'ordonnanceur. En effet, les tâches d'assemblage étant insérées en premier, l'ordonnanceur `eager` tend à les exécuter en premier, ce qui n'est pas optimal. En effet, les tâches d'assemblage peuvent être exécutées à tout moment avant toute autre tâche utilisant la donnée assemblée. L'ordonnanceur possède ainsi une grande liberté pour le placement de cette tâche, et il est préférable de les utiliser pour masquer la sous-utilisation des processeurs créée par les dépendances entre les autres tâches. Il est donc souhaitable de n'exécuter une tâche d'assemblage que lorsqu'aucune autre tâche n'est disponible, c'est-à-dire de leur donner une priorité faible.

La figure 4.20 illustre la différence faite par l'ajout de priorités. Le nombre de tâches prêtes (en noir) est important : si un processeur a fini une tâche et qu'aucune tâche n'est prête, alors il exécute une boucle d'attente. Le graphique du haut exhibe ce comportement à partir d'environ 200s après le début. On observe clairement deux phases dans ce calcul : dans la première, la majorité des tâches sont de l'assemblage de la $\mathcal{H}$-Matrice, et tous les processeurs sont occupés (ceci n'est pas visible sur ce graphique, mais une autre

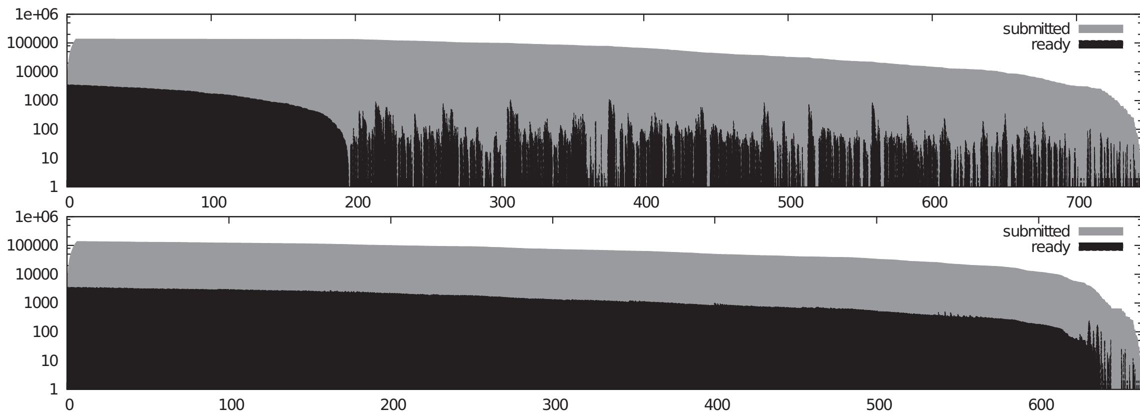


Figure 4.20 – Activité sans (haut) et avec (bas) une priorité faible pour les tâches d'assemblage. Le nombre de tâches soumises est en gris, et le nombre de tâches prêtes en noir.

sortie le confirme) ; la seconde contient des attentes périodiques, quand aucune tâche n'est disponible, ce qui nuit à l'efficacité.

Le graphique du bas ne présente pas de tels problèmes avant la fin de l'exécution, ce qui mène à une meilleure utilisation des unités de calcul, et un temps d'exécution plus faible (663s contre 745s sur 12 processeurs). Dans le premier cas, les unités d'exécution ont été inactives entre 15,45% et 17,64% du temps d'exécution, contre 2,34-3,00% dans le second. Pour ce calcul, le temps d'assemblage représente moins d'un tiers du temps d'exécution total, ce qui est néanmoins suffisant pour réduire significativement l'attente au cours de l'exécution.

La figure 4.21 montre deux traces d'exécution d'un même calcul avec les deux ordonnanceurs `eager` et `prio`. Dans les deux cas, il s'agit de l'assemblage et de la décomposition LU pour un avion parfaitement conducteur à 66596 inconnues, avec $|L_0| = 2158$. La machine est celle de la figure 4.2. Sur ce graphique, chaque ligne représente un processeur, l'axe des abscisses est celui du temps et chaque rectangle est une tâche. Les tâches d'assemblage sont en noir, les autres tâches de calcul en gris, et le temps d'attente est en rouge. Il apparaît que le temps d'attente est fortement réduit par l'utilisation de priorités, ce qui se traduit par un temps de calcul plus faible. Lorsque les priorités ne sont pas utilisées, les tâches d'assemblage sont toutes effectuées au début du calcul, ce qui mène à un manque de parallélisme plus loin dans l'algorithme. L'utilisation des priorités permet d'effectuer les tâches d'assemblage le plus tard possible, ce qui contribue à combler le manque de parallélisme dans l'algorithme.

4.7.1.4 Occupation mémoire

Dans le chapitre précédent, des éléments sur l'occupation mémoire du solveur séquentiel ont été donnés. Les algorithmes parallèles étant très proches des algorithmes séquentiels, il est naturel d'envisager une occupation mémoire similaire entre les deux solveurs. Cependant, deux éléments nouveaux influent sur la consommation mémoire :

— Le surcoût mémoire lié au moteur d'exécution ;

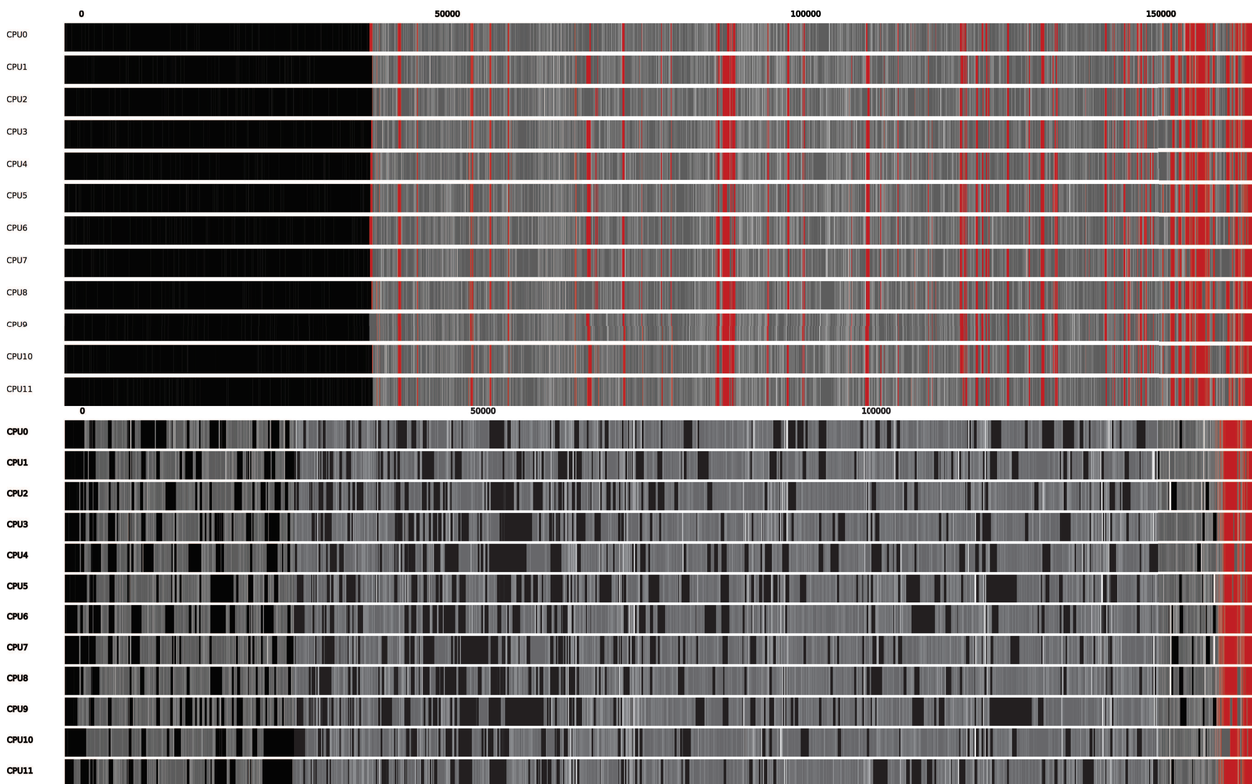


FIGURE 4.21 – Trace d'activité sans (haut) et avec (bas) gestion des priorités.

— La simultanéité possible de l'occurrence d'opérations produisant un pic de consommation mémoire.

Le premier élément augmente la consommation mémoire globale du solveur, le second la consommation maximale. Le surcoût mémoire engendré par le moteur d'exécution est difficile à quantifier. Il peut néanmoins être en première approximation supposé proportionnel au nombre de tâches, et dépend de nombreux paramètres. En pratique, ce surcoût n'est pas négligeable en valeur absolue, mais est faible relativement à la consommation mémoire totale du solveur.

La figure 4.22 compare la consommation mémoire en fonction de l'ordonnanceur avec StarPU, pour le même calcul que celui de la figure 3.48, sur une machine à 12 processeurs. Deux différences sont notables :

1. La différence d'allure des deux courbes ;
2. La surconsommation mémoire lors de l'utilisation des priorités.

Profil général Le profil global de la courbe correspondant à l'ordonnanceur `eager` est très proche de celui de la courbe 3.48, contracté dans le temps. Il est toujours possible d'identifier une phase constituée principalement de tâches d'assemblage, et une phase de factorisation. Dans le cas de l'utilisation des priorités, il n'est pas possible d'isoler ces deux phases, ce qui est cohérent avec les observations précédentes. Par ailleurs, le profil accidenté de la courbe de l'ordonnanceur `eager` se superpose avec le profil d'activité des processeurs, comme visible sur le premier graphique de la figure 4.20.

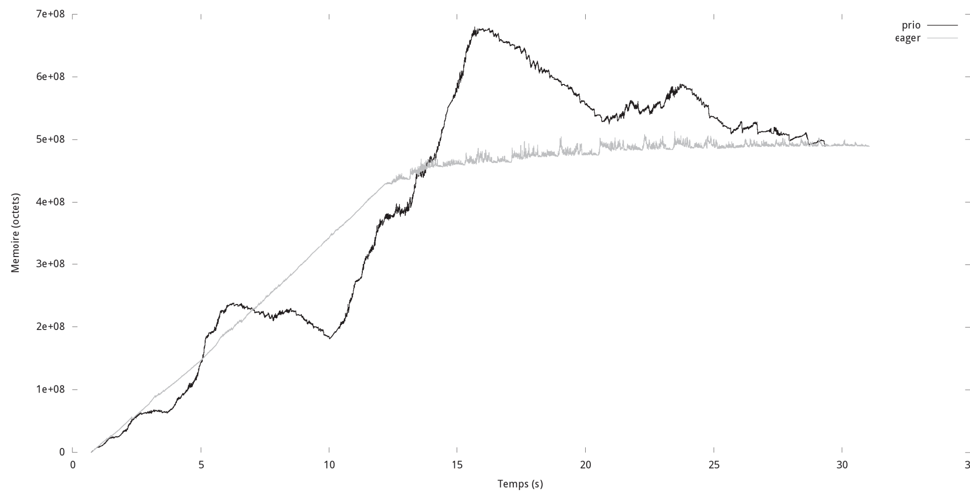


FIGURE 4.22 – Évolution de la consommation mémoire au cours d'un calcul parallèle. Comparaison des ordonnanceurs `eager` et `prio`.

Surconsommation avec priorités La consommation mémoire est significativement plus élevée que dans le cas séquentiel pour l'ordonnanceur `prio`, alors qu'elle n'est que modestement supérieure pour `eager`.

Une consommation mémoire maximale plus élevée pour le solveur parallèle est attendue. La factorisation, bien qu'effectuée sur place, nécessite la création de variables temporaires, qui produisent des pics locaux sur la courbe de consommation mémoire. Dans le cas d'un solveur parallèle, plusieurs allocations peuvent survenir en même temps pour des tâches non dépendantes, ce qui augmente la consommation mémoire instantanée.

La différence entre les deux algorithmes d'ordonnancement n'entre pas dans ce cadre. L'ordonnanceur `prio` permet d'atteindre un parallélisme plus grand, mais la différence n'est pas suffisante pour expliquer le résultat, ce qui se vérifie en restreignant le calcul à un seul processeur. Par ailleurs, les deux ordonnanceurs exécutent le même graphe de tâches, et réalisent exactement les mêmes allocations mémoire, dont seul l'ordre change.

Prenons l'exemple de l'algorithme 4.12. Dans le cas de base où $C|_{\sigma\times\tau} \notin L_0(C)$, une variable temporaire X est créée, modifiée par PRODSCALE, et plusieurs tâches ADDLEAF sont ajoutées. Ces tâches nécessitent l'assemblage préalable de $C_{\sigma'\times\tau'}$. Lorsque les priorités sont utilisées, l'assemblage de ces blocs est de priorité faible, et peut ne pas avoir été effectué. La durée de vie de la variable temporaire X est donc allongée, puisque les tâches ADDLEAF sont bloquées par les tâches ASSEMBLE pour tous les blocs $C_{\sigma'\times\tau'}$. Ceci explique la plus grande consommation mémoire observée, et également la convergence des deux courbes vers la fin de l'exécution. Lorsque les tâches d'assemblage ont été exécutées, l'allongement de la durée de vie des variables temporaires ne se produit plus, et la consommation mémoire devient identique, aux effets de parallélisme près.

Dans le cas où cette surconsommation est problématique, il est possible d'ajouter une dépendance artificielle de l'opération PRODSCALE sur les blocs $C|_{\sigma'\times\tau'}$. Ceci contraint davantage le graphe de tâches, et mène potentiellement à une perte d'efficacité parallèle.

4.7.2 Performances

Nous présentons ici une analyse de la performance parallèle de la méthode de parallélisation proposée. Les premiers tests sont la reproduction des calculs de la section 4.5.4, afin de donner une comparaison aisée avec les algorithmes BSP. Une exploration plus détaillée du passage à l'échelle est ensuite fournie.

4.7.2.1 Cône-Sphère sur Curie

On ne présente pas ici de résultats pour l'assemblage ; ils seraient redondants avec ceux de la section 4.5.4.2, les algorithmes étant très proches de ceux de cette section. En revanche, du fait de la gestion fine des dépendances, l'assemblage et le calcul suivant (inversion ou factorisation) sont faits dans le même graphe de tâches. La comparaison pour le temps de calcul séquentiel est donc faite avec la somme du temps d'assemblage et de l'opération considérée.

De plus, les matrices étant ici symétriques (formulation EFIE), une optimisation dans l'assemblage est faite (à la fois pour la version séquentielle et parallèle). Seule la partie triangulaire de la matrice est assemblée. Elle est ensuite recopiée vers la partie triangulaire supérieure. Ceci permet de diviser le temps d'assemblage par presque deux, mais est en réalité un désavantage pour le code parallèle.

En effet, les tâches d'assemblage n'ont pas de dépendance en lecture et sont de priorité faible. Elles servent donc à combler les « trous » de parallélisme, et contribuent à augmenter l'efficacité parallèle. Diviser par deux le nombre de tâches réduit alors cet avantage.

Enfin, nous donnerons une analyse plus précise du passage à l'échelle pour la factorisation *LU* uniquement pour des raisons de concision. L'inversion n'est pas utilisée en pratique ; elle est en effet moins précise, nécessite plus de mémoire et plus de temps de calcul. De plus, la section 4.5.4.4 montre que la décomposition *LU* est plus délicate du point de vue de la parallélisation pour les algorithmes de type BSP.

Conditions de calcul Pour tous les calculs de cette section, le moteur d'exécution choisi est StarPU 1.1. Une implémentation de l'inversion existe également avec QUARK, mais pas de la factorisation (pour des raisons de temps). Par ailleurs, le support de divers ordonnanceurs par StarPU et l'accès à des traces d'exécution précises expliquent le choix de ce moteur d'exécution. L'algorithme d'ordonnancement `prio` est utilisé, et la taille maximale de L_0 est fixée à 10000.

Inversion Les résultats pour l'inversion sont donnés par la figure 4.23 et le tableau 4.24. On observe un très bon passage à l'échelle, aussi bien faible que fort. Le passage à l'échelle faible (*weak scaling*) caractérise le comportement dans le cas où la quantité de travail par processeur reste constante. Ici, ceci correspond à considérer les efficacités parallèles pour une taille de problème croissante avec le nombre de processeurs. Le passage à l'échelle fort (*strong scaling*) caractérise le comportement à taille de problème fixe. On note en effet sur le tableau 4.24 un excellent passage à l'échelle même pour les plus petits problèmes.

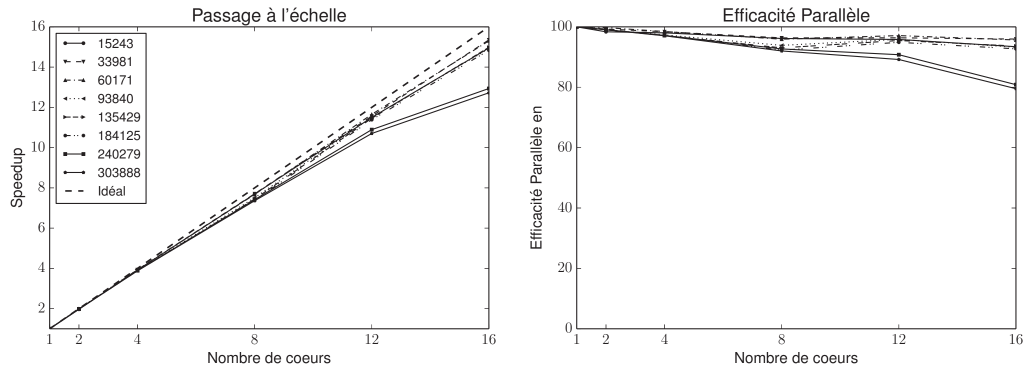


Figure 4.23 – Passage à l'échelle sur Curie de l'inversion.

	$p = 1$	$p = 2$		$p = 4$		$p = 8$		$p = 12$		$p = 16$	
N	t	t	E	t	E	t	E	t	E	t	E
15243	328.9	167.2	98.4	83.9	98.0	42.8	96.1	28.6	95.8	22.0	93.4
33981	1020.1	513.8	99.3	259.2	98.4	132.3	96.4	88.2	96.4	66.4	96.0
60171	2627.4	1317.1	99.7	667.2	98.5	342.4	95.9	225.5	97.1	171.7	95.7
93840	4468.7	2255.3	99.1	1147.7	97.3	595.0	93.9	388.5	95.9	299.5	93.2
135429	9472.8	4783.8	99.0	2438.0	97.1	1272.7	93.0	826.8	95.5	632.3	93.6
184125	14413.4	7272.8	99.1	3711.1	97.1	1952.2	92.3	1266.0	94.9	971.6	92.7
240279	24976.7	12606.6	99.1	6426.9	97.2	3368.4	92.7	2292.9	90.8	1930.7	80.9
303888	37157.3	18759.2	99.0	9572.8	97.0	5047.7	92.0	3471.7	89.2	2920.3	79.5

Figure 4.24 – Passage à l'échelle sur Curie de l'inversion. Les notations sont celles de la définition 4.1, les temps en seconde et les efficacités en pourcentage.

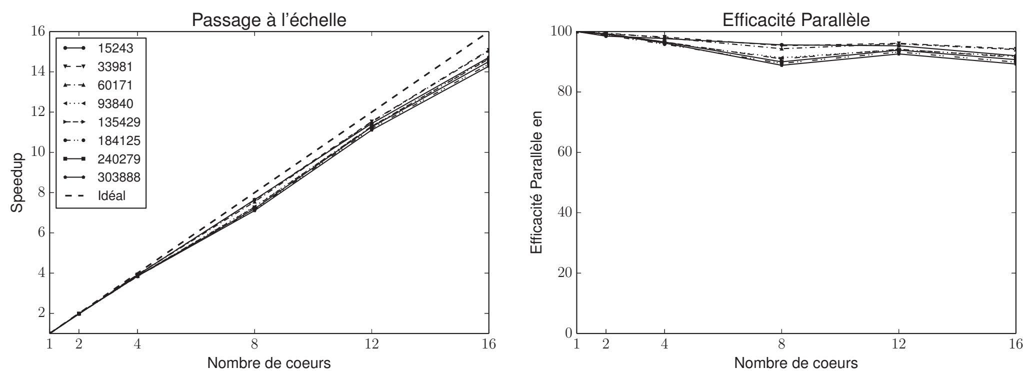


Figure 4.25 – Passage à l'échelle sur Curie de la décomposition LU.

Décomposition LU Le passage à l'échelle de la décomposition LU est également très bon, comme l'illustrent la figure 4.25 et le tableau 4.26. Contrairement aux résultats de la section 4.5.4.4, la décomposition n'exhibe pas de comportement moins favorable que l'inversion, montrant ainsi les avantages du formalisme présenté ici.

	$p=1$	$p=2$		$p=4$		$p=8$		$p=12$		$p=16$	
N	t	t	E	t	E	t	E	t	E	t	E
15243	141.6	71.9	98.5	36.2	97.7	18.5	95.6	12.4	95.3	9.6	92.0
33981	458.9	231.2	99.2	116.8	98.2	60.2	95.3	39.8	96.1	30.4	94.3
60171	1165.1	585.3	99.5	297.4	97.9	154.4	94.3	101.2	95.9	77.5	94.0
93840	2086.5	1053.0	99.1	542.4	96.2	285.6	91.3	184.9	94.0	142.1	91.8
135429	4268.4	2151.2	99.2	1105.2	96.6	586.6	91.0	377.8	94.1	291.6	91.5
184125	6591.8	3336.0	98.8	1719.9	95.8	921.0	89.5	589.2	93.2	458.3	89.9
240279	11255.7	5671.2	99.2	2914.7	96.5	1563.8	90.0	999.3	93.9	775.6	90.7
303888	16928.8	8536.5	99.2	4399.3	96.2	2382.1	88.8	1523.6	92.6	1185.5	89.2

FIGURE 4.26 – Passage à l'échelle sur Curie de la décomposition LU. Les notations sont celles de la définition 4.1, les temps en seconde et les efficacités en pourcentage.

Analyse L'efficacité parallèle représentée sur les figures précédentes est celle de la définition 4.1. Elle ne permet cependant pas de distinguer les diverses causes du manque de passage à l'échelle. En particulier, il est intéressant de distinguer les effets liés à l'architecture matérielle de la machine utilisée, ceux liés à l'ordonnanceur et ceux dus à un manque de parallélisme dans le graphe de tâches.

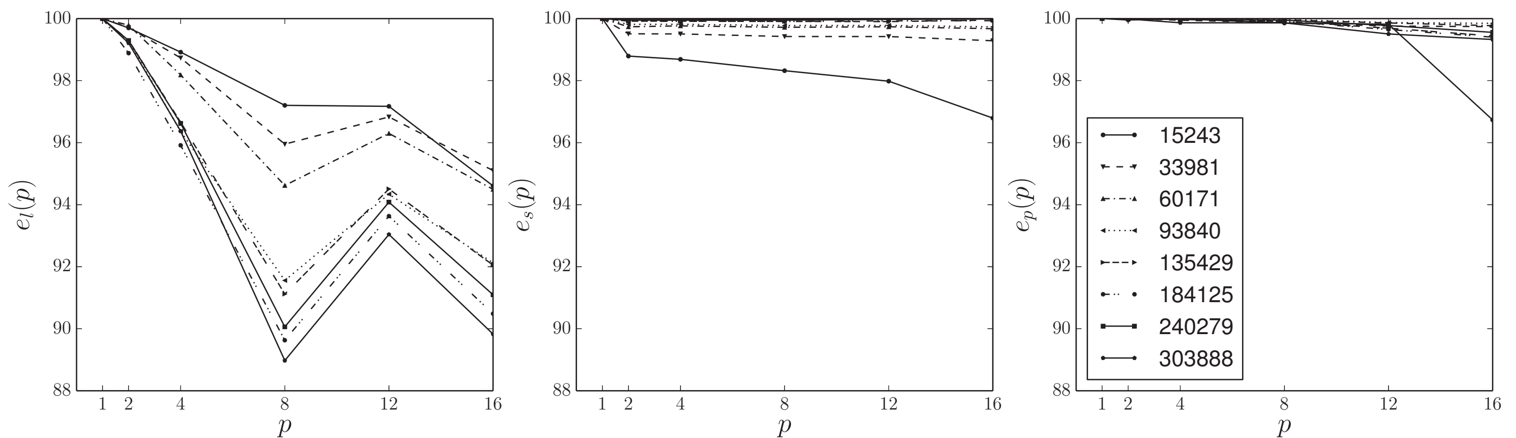


FIGURE 4.27 – Efficacités parallèles pour la décomposition LU, en pourcentage.

Ceci est possible avec les trois efficacités de l'équation 4.2.1, dont les valeurs sont représentées sur la figure 4.27. On rappelle la relation :

$$E(p) = e_l(p)e_s(p)e_p(p)$$

Le premier terme est dépendant de l'architecture matérielle de la machine de calcul, et de la capacité de l'ordonnanceur à conserver une bonne localité des données. Le second est le surcoût du moteur d'exécution, lié à l'optimisation de celui-ci et à la complexité du graphe de tâches (nombre de dépendances, par exemple). Le dernier terme représente le degré de parallélisme dans le graphe de tâches.

La figure 4.27 appelle les observations suivantes :

— L'efficacité de localité décroît entre 1 et 8 processeurs, remonte au-delà puis diminue de nouveau. Notons que la machine Curie-16 est constituée de deux processeurs à 8 cœurs. De plus, StarPU place par défaut les différents *threads* de calcul sur le même processeur. Ceci permet d'expliquer la forme générale de $e_l(p)$, et en particulier son minimum local pour $p = 8$, correspondant à une compétition pour des ressources partagées sur le même processeur. On note par ailleurs qu'aucune autre efficacité n'est en-dessous de 95%, et c'est donc cette efficacité qui explique l'essentiel du manque de passage à l'échelle de la décomposition LU. Enfin, l'efficactié $e_l(p)$ est décroissante avec l'augmentation de la taille du problème.

— En-dehors du plus petit cas de calcul (15243 inconnues), le surcoût du moteur d'exécution est très faible. Une valeur plus grande pour le plus petit calcul est attendue, puisque la taille de la coupe de l'arbre est du même ordre de grandeur quel que soit le cas de calcul, conduisant ainsi à une complexité des graphes de tâches comparable dans les différentes configurations. Le calcul étant plus court dans ce cas, il est naturel que l'efficacité soit moindre.

— De la même manière que pour le surcoût de l'ordonnanceur, en-dehors du cas de plus petite taille, le parallélisme du graphe de tâches est très bon. Cette valeur met en évidence le gain important apporté par la parallélisation décrite dans cette section.

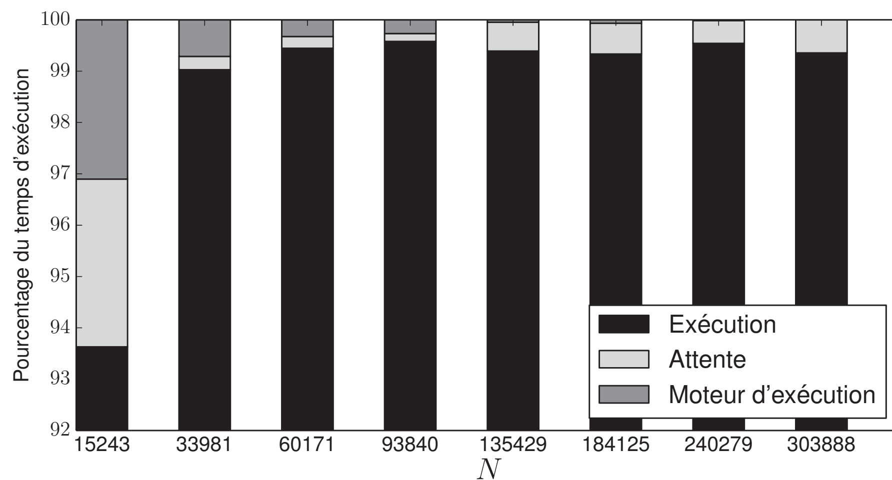


Figure 4.28 – Répartition du temps entre le calcul, l'ordonnanceur et l'attente pour la décomposition LU. On note que seuls les 8 derniers pourcents du temps d'exécution sont représentés.

La figure 4.28 décrit la répartition du temps de calcul entre les diverses composantes (calcul, surcoût du moteur d'exécution et attente) pour différentes tailles de problèmes. Cette figure illustre les conclusions précédentes :

— Le temps d'ordonnancement est négligeable ;

— Le degré de parallélisme du graphe de tâches est suffisant.

4.7.2.2 Passage à l'échelle en nombre de processeurs

Les résultats de la section précédente montrent un bon passage à l'échelle des algorithmes sur une machine à mémoire partagée de taille moyenne courant 2013. Dans cette

Processeur	4× Intel Xeon « Westmere EX » E7-4870 à 2.4GHz, 10 cœurs
Mémoire	4 × 256Go, DDR3 ECC
Logiciels	Linux 2.6.32, Intel Composer XE 2013, Intel MKL 11.0

TABLE 4.5 – Principales caractéristiques de la machine Plafrim-40.

section, nous explorons le passage à l'échelle sur une machine plus fortement parallèle, en reproduisant les tests précédents. Les configurations physiques et de calcul sont les mêmes que précédemment.

FIGURE 4.29 – Topologie simplifiée de la machine Plafrim-40.

Configuration informatique La machine utilisée fait partie de la plateforme de calcul Plafrim [5] et sera notée dans la suite Plafrim-40. Ses caractéristiques sont données par la table 4.5 et sa topologie simplifiée par la figure 4.29. On note que les processeurs ne sont pas de la même génération que ceux de la machine Curie-16 ; ceux de Plafrim-40 ne supportent en particulier pas les instructions AVX. Ceci signifie que le nombre d'opérations en virgule flottante par seconde (flops) maximal est deux fois plus faible à fréquence égale par rapport à Curie-16. Les performances ne sont donc pas directement comparables entre ces deux machines.

	$p = 1$	$p = 4$		$p = 8$		$p = 16$		$p = 32$		$p = 40$	
N	t	t	E	t	E	t	E	t	E	t	E
184125	9997.5	2599.7	96.1	1356.3	92.1	688.4	90.8	394.2	79.2	363.6	68.7
240279	16965.6	4404.7	96.3	2296.6	92.3	1167.6	90.8	640.2	82.8	583.8	72.7
303888	25795.9	6777.6	95.2	3527.5	91.4	1785.7	90.3	967.2	83.3	848.7	76.0

TABLE 4.6 – Passage à l'échelle sur Plafrim-40 de la décomposition LU. Les notations sont celles de la définition 4.1, les temps en seconde et les efficacités en pourcentage.

Décomposition LU Le passage à l'échelle de la décomposition est illustré par la figure 4.30 et la table 4.6. L'efficacité parallèle est comparable à celle obtenue dans la

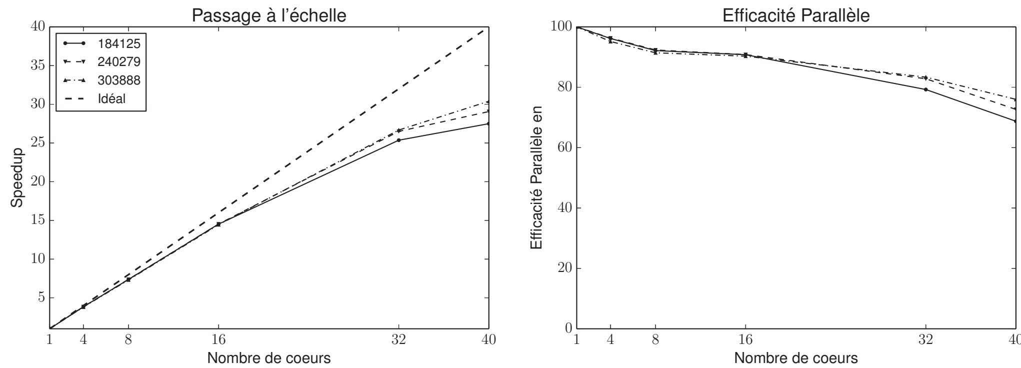


FIGURE 4.30 – Passage à l'échelle sur Plafrim-40 de la décomposition LU.

section précédente pour un nombre de processeurs compris entre 1 et 16, et diminue ensuite, avec une décroissance notable entre 32 et 40 processeurs. Par ailleurs, l'efficacité parallèle est supérieure pour les problèmes de plus grande dimension, sans qu'il ne soit possible d'attribuer cette différence au parallélisme du graphe de tâches ou à la localité des données.

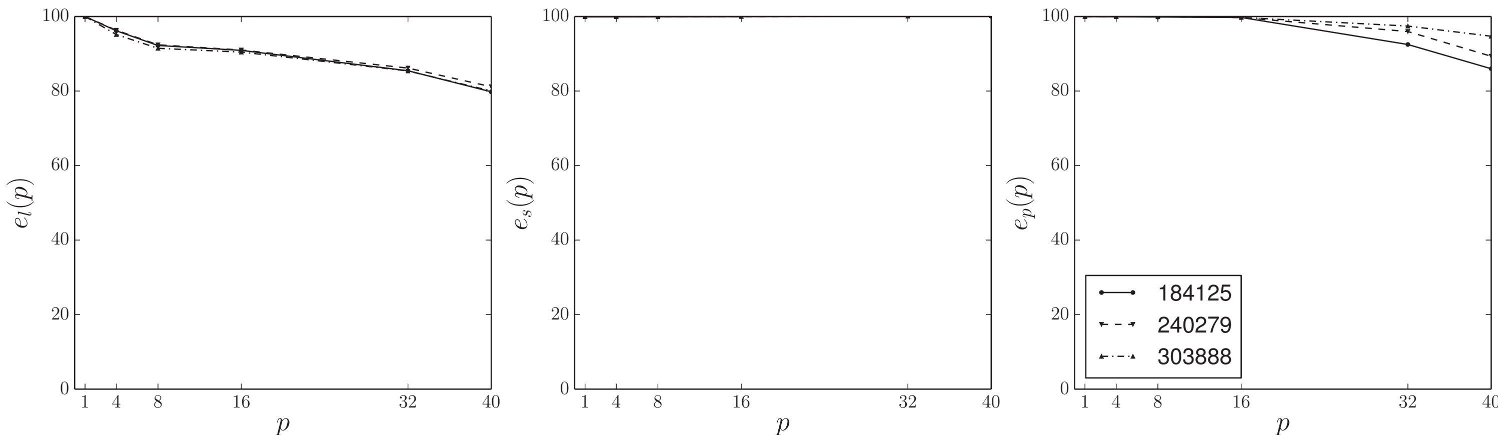


FIGURE 4.31 – Efficacités parallèles pour la décomposition LU, en pourcentage.

Analyse La décomposition des efficacités parallèles est représentée sur la figure 4.31. L'efficacité du moteur d'exécution $e_s(p)$ n'a pas de valeur pertinente ici. En effet, dans un certain nombre de cas, sa valeur est supérieure à 1, ce qui est lié à la précision finie des chronomètres utilisés pour calculer $t_s(p)$. Néanmoins, cette efficacité est très proche de 1 dans toutes les autres situation, ce qui laisse supposer que sa valeur réelle est effectivement proche de 1. Les valeurs des efficacités $e_l(p)$ et $e_p(p)$ permet de dégager les observations suivantes :

— La majorité de la perte d'efficacité parallèle pour $n > 32$ est liée à la localité et au partage des ressources, comme la bande passante mémoire. Cette efficacité limite l'efficacité parallèle globale à environ 80% pour un calcul à 40 processeurs. De plus, cette efficacité décroît de manière notable entre 32 et 40 processeurs, ce qui peut

être lié à des conflits d'accès aux ressources partagées (bus de communication entre processeurs, bande passante mémoire, mémoire cache de niveau 3, *etc.*).

— Le parallélisme du graphe de tâches est globalement suffisant, bien qu'un manque de parallélisme soit constaté pour les plus petits cas. La solution consiste alors à augmenter $|L_0(M)|$ pour accroître le degré de parallélisme exprimé par le graphe de tâches. La répartition entre attente et calcul est illustrée par la figure 4.32.

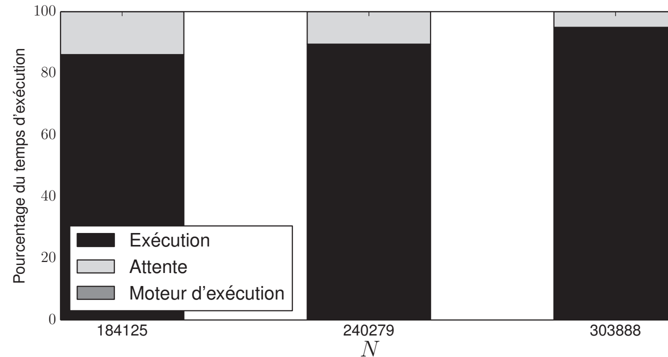


Figure 4.32 – Répartition du temps entre le calcul, l'ordonnanceur et l'attente pour la décomposition LU.

4.8 Conclusion

Nous avons montré dans ce chapitre la parallélisation en mémoire partagée d'un solveur $\mathcal{H}$-Matrice. Celle-ci fait appel à des méthodes récentes, dont nous avons démontré la pertinence pour traiter un algorithme irrégulier et hiérarchique. Cette approche a permis d'obtenir des résultats meilleurs que la littérature, et de dégager une méthode permettant de rapidement et efficacement implémenter de nouveaux algorithmes en parallèle.

En effet, les algorithmes parallèles présentés dans ce chapitre reprennent la structure et les opérations de base de leur version séquentielle, ce qui permet de paralléliser aisément un algorithme séquentiel. Ceci est en cours, avec l'ajout d'un solveur symétrique, particulièrement utile pour les formulations EFIE. Cette approche a pour avantage de conserver le découplage entre les diverses étapes d'un calcul. Le suivi des dépendances est local à un cas de base, et n'a pas de conséquences sur les autres opérations, tout en permettant à celles-ci de se composer naturellement. Cela est important pour un solveur industriel, car ceci permet de conserver la même facilité d'écriture et de ne pas modifier les parties du code « clientes » de l'algèbre linéaire, et le rend souple d'utilisation.

L'utilisation d'une coupe L_0 dans l'arbre de blocs permet de séparer les heuristiques visant à optimiser la granularité des tâches de l'expression des algorithmes, et autorise ainsi une expérimentation simple et aisé de plusieurs heuristiques. Le choix d'une heuristique adaptée fait partie des perspectives de ce travail. Quelques pistes ont été présentées dans la section 4.7.1.1, mais une étude systématique serait nécessaire pour déterminer une heuristique significativement plus efficace que la fonction f « naïve » employée dans l'essentiel de ce chapitre.

Par ailleurs, l'utilisation d'une granularité différente pour diverses tâches serait à explorer. Ainsi, certaines tâches coûteuses (assemblage, recompression adaptative) pourraient être divisées lorsque cela est possible (dans le cas de l'assemblage, lorsque l'élément de L_0 base de la récursion n'est pas une feuille de l'arbre de blocs), et des opérations moins lourdes comme la mise à l'échelle pourraient au contraire être fusionnées (ce qui entraînerait plusieurs dépendances en écriture).

Chapitre 5

Algorithmes parallèles en mémoire distribuée et perspectives

Sommaire

5.1 **Mémoire distribuée** 214
5.2 **Perspectives : Architectures hybrides et *Out-Of-Core*** 229

L'ÉVOLUTION des machines de calcul se poursuit depuis plusieurs années selon deux axes : parallélisme et spécialisation. La première place du « Top 500 » [6] début 2013 représente bien ces deux caractéristiques : cette machine est composée de 299.008 cœurs, et 18.688 cartes graphiques, en autant de nœuds indépendants. Ainsi l'exploitation des architectures actuelles et à venir nécessite d'appréhender ces deux réalités, les architectures à mémoire partagée hybrides.

Le parallélisme massif rend la gestion de la cohérence de cache nécessaire aux machines à mémoire partagée coûteuse. De plus, les impératifs de fiabilité et de maîtrise des coûts mènent à la conception de machines composées de nombreux nœuds reliés par un réseau de communication haute performance. Le modèle de programmation le plus représenté sur ce type de machine est lié à la norme MPI, spécifiant un ensemble d'interfaces de programmation. Cette norme est basée sur le concept d'échange de messages entre divers processus exécutant le même programme. Les échanges peuvent être « point à point » ou collectifs, synchrones ou asynchrones, leur définition et gestion étant laissée à la charge du programmeur. Cette architecture ajoute donc au problème de l'accès concurrent aux données celui de la communication efficace de celles-ci. En particulier, la communication entre deux nœuds nécessite une action coordonnée de l'émetteur et du récepteur.

Le cas de l'exploitation des architectures hétérogènes comportant des accélérateurs spécialisés est autre : il s'agit de prendre en compte l'hétérogénéité des performances et capacités des diverses unités de calcul, ce qui complique l'ordonnancement. De plus, l'espace mémoire de ces accélérateurs est le plus souvent distinct de la mémoire principale, et de taille restreinte. Les difficultés associées à ces accélérateurs ont trait à l'exploitation efficace de leurs unités de calcul ; ceci peut nécessiter une reformulation des algorithmes.

Les voies de communication entre la mémoire principale et la mémoire limitée d'un accélérateur étant à forte latence et faible débit (comparativement à leur puissance de calcul), le placement judicieux des données est indispensable pour obtenir des performances satisfaisantes.

Plan et contributions

Ce chapitre présente l'implémentation des algorithmes des $\mathcal{H}$-Matrices sur une architecture à mémoire distribuée, ainsi que des perspectives ayant trait à la gestion d'une mémoire à plusieurs niveaux et aux architectures hétérogènes. Nous montrerons que la formulation usant de graphes de tâches du chapitre précédent permet d'exploiter ces architectures, et préciserons le modèle d'exécution retenu.

Dans le cas de la mémoire partagée, nous présentons l'extension du formalisme de graphe de tâches permettant au moteur d'exécution (StarPU) d'exécuter un graphe de tâches sur une grappe de machines. Nous précisons les modifications à apporter à la bibliothèque $\mathcal{H}$-Matrice pour exploiter ces architectures, et donnons des résultats sur la performance parallèle des algorithmes en mémoire distribuée.

Le reste du chapitre est consacré à deux extensions en cours d'investigation : l'exploitation des architectures hybrides et le déchargement des données sur le disque. Dans le premier cas, nous décrivons les gains à attendre de l'exploitation des accélérateurs de type « carte graphique ». En particulier, l'accélération de l'assemblage des $\mathcal{H}$-Matrices est illustré par une expérience numérique. Le déchargement des données sur le disque est également décrit dans le principe. On donne de plus des remarques sur l'utilisation d'indications pour l'ordonnanceur permettant d'optimiser le déchargement des données. Ces deux extensions représentent des axes de travail actuel, et des prolongement naturels et directs de cette thèse.

5.1 Mémoire distribuée

5.1.1 Cadre

Nous considérons dans cette section un ensemble de machines à mémoire partagée, reliées par un réseau de communication, tel que représenté sur la figure 5.1. Ces machines sont capables d'exécuter le même programme, et de communiquer des informations entre elles par envoi de messages (*message passing*). On parle de modèle de programmation *Same Program, Multiple Data* (SPMD) [40], c'est-à-dire l'exécution par plusieurs unités du même programme, opérant sur des données différentes. Il existe plusieurs environnements de ce type, le plus universellement employé étant MPI [60, 61], et le reste de ce chapitre s'inscrit dans ce cadre.

5.1.1.1 Brève présentation de MPI

Le standard MPI définit un ensemble d'interfaces de programmation dont le comportement est spécifié, mais non l'implémentation. Ces interfaces permettent à diverses machines

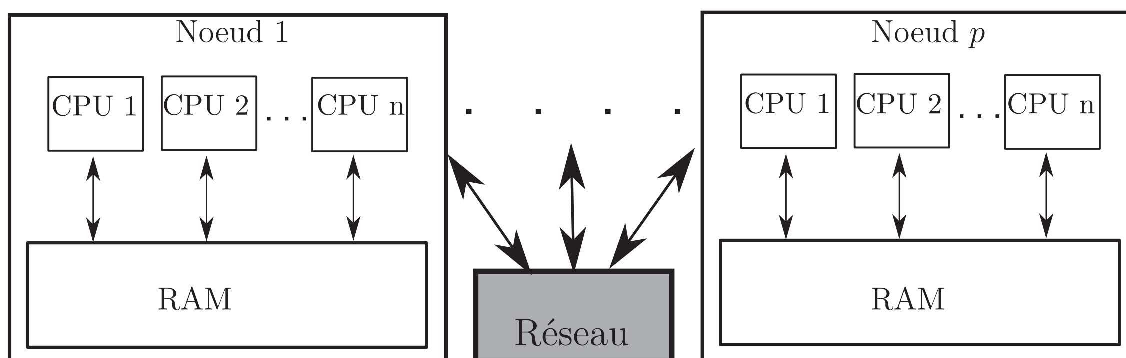


Figure 5.1 – Représentation schématique d'un ensemble de machines reliées par un réseau de communication. Chaque machine est elle-même une machine parallèle.

participant à un même calcul (et exécutant le même programme) de communiquer et de se synchroniser. Chaque machine (ou processus MPI) est nommée dans la suite **nœud**, et est identifiée au sein de l'ensemble des machines par un indice unique, appelé **rang**. Des sous-ensembles de nœuds peuvent être désignés sous le nom de **communicateur**, l'ensemble global (le « monde ») ayant le nom `MPI_COMM_WORLD`.

Remarque 5.1. *Cette présentation de MPI n'est pas exhaustive. De nombreux points de la norme ne sont pas discutés ici, de même que les détails d'implémentation sont passés sous silence. En particulier, il n'est pas fait mention des opérations «* one-sided *» ni des types dérivés. L'objectif de cette section est de donner un aperçu de haut niveau du modèle de programmation, bien que les détails d'implémentation soient importants pour optimiser les performances des communications.*

Dans la majorité des programmes parallèles, la première étape est d'obtenir la taille du communicateur global (c'est-à-dire le nombre de nœuds participant au calcul) et le rang du nœud courant dans ce communicateur, par les appels `MPI_Comm_rank()` et `MPI_Comm_size()`.

Les machines peuvent ensuite communiquer entre elles par échange de messages, c'est-à-dire de tableaux de données avec le modèle suivant :

— L'émetteur envoie le message à un destinataire identifié par son rang dans le communicateur, dont la taille et le type sont spécifiés. Un identifiant (***tag***) est joint à ce message, dont la valeur et le sens sont laissés au choix de l'application. L'appel résultant a la forme :
`MPI_Send(données, taille, type, destinataire, tag, communicateur)`
Cet appel est bloquant jusqu'à ce que les données soient envoyées.

— Le récepteur demande la réception du message, avec les mêmes paramètres que l'émetteur. Il doit donc y avoir accord entre l'émetteur et le récepteur sur le couple provenance/destination du message, son type, sa taille et son *tag*. L'appel est de la forme :
`MPI_Recv(données, taille, type, émetteur, tag, communicateur)`
Cet appel est bloquant. En effet, si les données ne sont pas disponibles ou que cet appel ne correspond pas à un envoi, le programme attend.

Ces communications, dites « point à point » ont l'inconvénient d'être synchrones, et dans de nombreux cas, il n'est pas souhaitable de bloquer l'exécution du programme

ainsi. En effet, il est bien souvent possible de déterminer à l'avance les données dont le programme local aura besoin. Ceci permet, sous réserve que les transferts de données sans utilisation de temps processeur soient supportés [1], d'éviter de bloquer les calculs. À cette fin, le standard MPI propose des variantes asynchrones `MPI_Isend/Irecv()` des opérations précédentes. Celles-ci ne bloquent pas, et l'opération n'est donc pas achevée lorsque la fonction se termine. Il est alors nécessaire de tester (`MPI_Test()`) ou d'attendre (`MPI_Wait()`) la terminaison de la requête avant de pouvoir libérer ou réutiliser les données du côté de l'émetteur, et d'utiliser les données du côté du récepteur.

Il existe par ailleurs des opérations « collectives », qui permettent des échanges de données entre plusieurs nœuds simultanément, dont il ne sera pas question dans ce chapitre. Enfin, il est possible de bloquer l'exécution du programme jusqu'à ce que tous les nœuds d'un même communicateur aient atteint une **barrière** avec l'appel `MPI_Barrier()` qui sert de primitive de synchronisation. Ceci permet d'implémenter les synchronisation globales du modèle BSP de la figure 4.4 page 160.

Remarques Ce modèle de programmation est relativement rigide. Pour qu'une communication réussisse, il est en effet nécessaire que l'émetteur et le récepteur connaissent l'ensemble des paramètres de la communication, c'est-à-dire le couple émetteur/récepteur, la taille et le type des données, et le *tag*.

Il est possible de contourner cette difficulté, en construisant une sémantique plus riche et flexible au-dessus des fonctions disponibles dans le standard, cependant ceci reste à la charge de l'application. Ceci complexifie l'expression de certains modèles de parallélisme. En particulier, un parallélisme dynamique dont les motifs de communication ne sont pas figés est plus difficile à exprimer avec MPI qu'un modèle statique, où la distribution des données, le nombre et la taille des communications sont connus à l'avance.

Un modèle statique est par ailleurs plus adapté à l'exploitation des opérations collectives, dont l'implémentation peut permettre des gains significatifs. Les deux variables importantes pour la performances des communications sont la latence et la bande passante ; une exploitation judicieuse des opérations collectives peut permettre de diminuer le nombre de communications (et donc l'impact de la latence), et d'éliminer une partie de la redondance dans le transfert de données (et donc de tirer le meilleur parti de la bande passante disponible).

5.1.1.2 Parallélisme à plusieurs niveaux

Les machines considérées dans cette section sont composées de plusieurs nœuds, dont chacun est à mémoire partagée, possédant en général plusieurs unités de calcul [2]. Une approche naïve (et souvent employée pour des raisons historiques) considère chaque unité de calcul comme un nœud isolé. Dans ce cas, chaque processeur communique avec tous les autres à travers d'échanges de messages, que les partenaires de communication appartiennent physiquement au même nœud ou non [3]. Ceci n'est la plupart du temps pas

1. Ce qui est le cas pour de nombreuses implémentations logicielles et matérielles
2. Par exemple, la topologie de la figure 4.2 page 149.
3. Dans le cas où la communication est physiquement locale, des optimisations dans l'implémentation MPI sont possibles et souhaitables [38].

optimal, car l'impact des communications sur le temps d'exécution est croissant avec le nombre de processus dans un calcul MPI. Il est donc souhaitable de se tourner vers une parallélisation à deux niveaux :

— Algorithmes en mémoire partagée à l'intérieur d'un nœud ;
— Communication par messages entre les nœuds physiquement distincts.

En plus de réduire le nombre de communications (et de copies mémoire), ceci permet également d'optimiser l'utilisation des caches du processeur et de la mémoire. En effet, le nombre de changements de contexte est réduit, et les données n'ont pas besoin d'être répliquées pour être utilisées conjointement à l'intérieur d'un nœud. Enfin, la plus grande souplesse d'ordonnancement des opérations possible sur une machine à mémoire partagée n'est pas perdue.

Ceci nécessite en revanche un effort supplémentaire de la part du programmeur, qui doit faire cohabiter deux modèles de parallélisation distincts dans la même application, avec les difficultés d'équilibrage de charge et de synchronisation afférentes. C'est pour ces raisons que nous examinons dans la suite de ce document l'adaptation d'une parallélisation par graphes de tâches à ce contexte, et son impact sur la facilité de programmation. Celle-ci permet d'exploiter un parallélisme dynamique à plusieurs niveaux, sans la complexité décrite ici.

5.1.2 Graphes de tâches et mémoire distribuée

Dans cette section, une description de l'extension du modèle de graphe de tâches à une architecture à mémoire distribuée est donnée. Celle-ci correspond au modèle retenu par StarPU, qui est le support de l'implémentation présentée dans ce chapitre.

Le formalisme de graphe de tâches du chapitre précédent permet d'exprimer de façon explicite les dépendances entre les tâches au travers des données et de l'ordre d'énumération des tâches. Ce formalisme contient donc naturellement une description des données dont la connaissance est nécessaire pour exécuter une tâche, sous réserve que les dépendances soient exhaustives.

Ceci constitue un changement par rapport au cadre du chapitre précédent. En effet, en mémoire partagée, si une dépendance vers un ensemble de données dont l'accès concurrent ne pose jamais problème (par exemple des données en lecture seule) n'est pas spécifiée dans les dépendances des tâches, alors l'exécution est tout de même correcte, puisque les contraintes imposées sont assez fortes. Autrement dit, une donnée n'étant jamais accédée en écriture n'ajoute pas d'arête dans le graphe de tâches. Ce n'est plus le cas en mémoire distribuée, puisque la connaissance de toutes les données par tous les nœuds n'est plus assurée. Nous supposons donc dans la suite que toutes les dépendances de données sont explicitées dans le graphe de tâche. Par ailleurs, notons que le reste de ce chapitre est relatif aux choix réalisés dans StarPU, et décrits dans [18, chapitre 5].

5.1.2.1 Partition du graphe de tâches

Soit un graphe de tâches (S, A). On considère une partition de l'ensemble de ses sommets $S = S_1 \cup \cdots \cup S_p$ à p éléments, c'est-à-dire p sous-ensembles disjoints de tâches. On

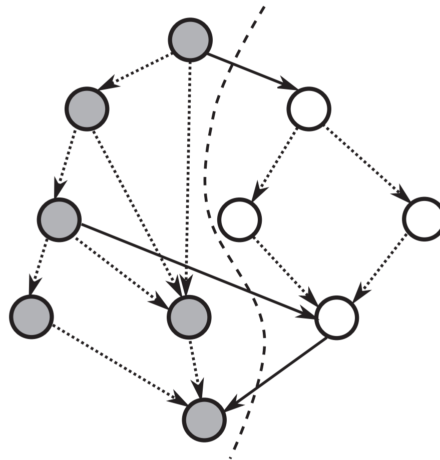


Figure 5.2 – Exemple de partition d'un graphe de tâches, $p = 2$. Les arcs en trait plein représentent les dépendances inter-nœuds, les autres les dépendances locales. La couleur des sommets indique leur appartenance à chaque partie.

rappelle que les arêtes du graphe représentent des dépendances induites par les données, et peuvent être annotées par le sous-ensemble des données relatives à cette dépendances. La suite de ce chapitre travaillera sur un tel graphe dont les arêtes sont annotées.

Définition 5.2 (Dépendances)**.** *Soit un graphe de tâches* (S, A) *et* $S = S_1 \cup \cdots \cup S_p$ *une partition de ses sommets.*
On appelle ***dépendance locale*** *ou intra-nœud une arête* $(i, j) \in A$ *telle que* $i, j \in S_k$ *pour un* k*. Dans le cas contraire (les deux extrémités de l'arête ne sont pas dans le même sous-graphe), cette arête est qualifié de* ***dépendance distante*** *ou inter-nœuds.*

Un exemple de partition d'un graphe de tâches est donné par la figure 5.2. Ce modèle permet de décrire l'exécution parallèle d'un graphe de tâche sur p nœuds. En effet, chaque tâche s'exécute sur un seul des processeurs, et toutes les tâches s'exécutent, ce qui correspond bien à une partition de S. Les dépendances représentant des contraintes sur les données, elles correspondent à des échanges de données. Dans le cas des dépendances locales, un échange explicite n'est pas nécessaire puisque toutes les données sont accessibles par tous les processeurs d'un même nœud. Cependant, en mémoire distribuée ces dépendances sont suffisantes pour gérer les communication nécessaires.

5.1.2.2 Gestion des dépendances locales et distantes

On rappelle les règles de création d'arêtes dans le graphe de tâches réduit pour une tâche t ayant comme dépendances en lecture $In(t)$ et en écriture $Out(t)$:

- Tout donnée $d \in In(t)$ ajoute une arête (t', t), avec t' la tâche la plus récente (si elle existe) telle que $d \in Out(t')$;
- Toute donnée $d \in Out(t)$ ajoute une arête (t', t), avec t' la tâche la plus récente (si elle existe) telle que $d \in In(t') \cup Out(t')$

Toute arête est annotée avec la ou les données responsables de sa création dans le graphe de tâches.

Afin de permettre de faciliter l'explication des algorithmes suivants, nous ajoutons l'hypothèse suivante :

$$\forall d \in In \cup Out, \underset{i \in \{1,n\}}{\arg\min}\{t_i | d \in In(t_i) \cup Out(t_i)\} = \underset{i \in \{1,n\}}{\arg\min}\{t_i | d \in Out(t_i)\}$$

Ceci signifie que la première tâche utilisant une donnée doit l'écrire. Cette contrainte est à relier avec le commentaire de la section précédente : les dépendances doivent être exhaustives. La tâche de création d'une donnée doit donc apparaître explicitement dans le graphe de tâche.

Précisons la nécessité de cette hypothèse. Soit une donnée d n'étant utilisée qu'en lecture. Celle-ci ne crée pas d'arête dans le graphe de tâches, et n'engendre alors ni dépendance locale, ni dépendance distante. Le graphe de tâches est alors identique à un graphe dont les tâches n'opèrent pas sur cette donnée, et les communications nécessaires ne sont pas effectuées.

Dépendances locales Une dépendance locale est annotée par un ensemble de données ayant été modifiées (ou lues) pour la dernière fois dans la même partie du graphe que la tâche courante. Ces données sont donc localement disponibles et à jour, et il n'y a pas de différence de traitement d'une telle dépendance par rapport au cas de la mémoire partagée. Supposons qu'une partie du graphe de tâche n'ait aucune dépendance distante. Alors, l'exécution parallèle de cette partie du graphe de tâches est en tout point identique au cas de la mémoire partagée, sans communications entre les nœuds.

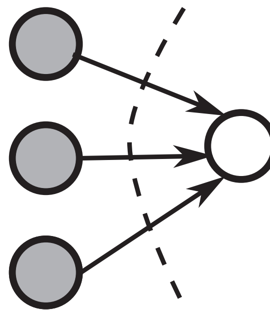


Figure 5.3 – Multiples dépendances distantes. Par exemple, toutes les tâches grises accèdent à la même donnée en lecture, qui est écrite par la tâche blanche.

Dépendances distantes Une dépendance distante $t_i \rightarrow t_j$ est annotée par un ensemble de données dont la dernière modification (ou lecture) a été effectuée par une tâche d'une autre partie du graphe de tâches. Il est donc nécessaire d'effectuer une communication pour récupérer cette donnée.

Néanmoins, toute arête correspondant à une dépendance distante n'entraîne pas de communication. Prenons l'exemple de la configuration de la figure 5.3. Supposons que les dépendances portent toutes sur la même donnée, qui est lue par toutes les tâches grises, et modifiée par la tâche blanche. Dans ce cas, il est clair que chaque arête ne se traduit pas par une communication. Il est néanmoins nécessaire d'effectuer une communication, sauf dans le cas où il existe également une dépendance locale sur la même donnée. Il

existe plusieurs méthodes pour déterminer les communications à effectuer ; celle utilisée par StarPU est l'objet de la section suivante.

5.1.2.3 Gestion des communications et partition du graphe de tâches

On considère un calcul s'exécutant sur p nœuds. À chaque donnée d est associée à sa création un nœud de référence $h(d) \in \{1, p\}$. Ce nœud est celui sur lequel s'exécute la tâche de création de cette donnée. On impose la propriété suivante : le nœud $h(d)$ détient toujours une copie à jour de la donnée d. Cette propriété est assurée par la règle suivante : toute tâche t modifiant une donnée d ($d \in Out(t)$) doit l'envoyer au nœud $h(d)$ une fois son exécution terminée.

Les règles sont alors, du côté du récepteur (tâche ayant des dépendances distantes) :

- Avant l'exécution de la tâche, récupérer les données nécessaires depuis leur nœud de référence, de manière asynchrone ;
- Après l'exécution, envoyer les données modifiées par la tâche à leur nœud de référence. Ceci est inutile pour les données n'ayant pas été modifiées.

Et du côté de l'émetteur :

- Pour toute tâche ne s'exécutant pas sur le nœud courant et ayant des dépendances sur des données dont le nœud courant est le nœud de référence, faire un envoi asynchrone.
- Recevoir les données ayant été modifiées par la tâche distante.

Ce protocole permet de résoudre le problème des dépendances distantes, sous réserve que le graphe de tâche soit connu par tous les nœuds et que les affectations des tâches aux nœuds soient statiques. Afin de réduire le nombre et le volume des communications, il est naturel d'affecter les tâches aux nœuds sur lesquels les dépendances en écriture sont rattachées. Lorsqu'il n'y a qu'un seule dépendance en écriture, la tâche est automatiquement affectée au nœud de référence de cette donnée [4]. Dans les autres cas, l'affectation d'une tâche à un nœud est à la charge de l'utilisateur.

Cache de communications Il est possible de réduire le nombre de communications en exploitant la connaissance du graphe de tâches par tous les nœuds, sans nécessiter de gestion centralisée des divers caches de données (ce qui ferait de StarPU un système de « *Distributed Shared Memory* » [74]). Ceci peut se faire de manière distrbuée de la façon suivante :

- Lors de l'envoi d'une donnée d de son nœud de référence $q_1 = h(d)$ vers un autre nœud q_2,
  - Le nœud q_1 note que le nœud q_2 possède une copie à jour de la donnée d ;
  - Le nœud q_2 note qu'il possède une copie à jour de la donnée d.
- Lors d'une demande de transfert ultérieure de la donnée d vers le même nœud q_2,
  - Le nœud émetteur $q_1 = h(d)$ trouve dans une table que q_2 possède une copie à jour, et ne procède pas à un envoi ;

4. Ceci peut être modifié par l'utilisateur.

— Le nœud q_2 trouve également dans une table qu'il possède une copie à jour de d et ne demande pas de réception.

Ce mécanisme est satisfaisant dans le cas où d n'a pas été modifiée entre les deux envois. Il reste à expliquer le mécanisme d'invalidation du cache. Cette invalidation doit se produire correctement du côté de l'émetteur et du récepteur.

Émetteur L'émetteur d'une donnée d est nécessairement le nœud de référence de cette donnée, q_1. Ce nœud connaît toutes les modifications faites sur cette donnée. En effet,

— Les modifications locales de cette donnée entraînent l'invalidation du cache. Il est noté que toutes les copies distantes sont désormais invalides.

— Les modifications distantes de cette donnée entraînent le rapatriement de cette donnée sur le nœud q_1 après exécution. Ces modifications sont donc suivies par q_1, qui marque les copies distantes comme invalides à ce moment.

Récepteur Lors de l'insertion d'une tâche non locale modifiant d, une invalidation différée est soumise localement pour marquer la copie locale comme invalide, si elle existe. Cette invalidation différée repose sur le mécanisme de résolution des dépendances de StarPU, et est garantie de s'exécuter avant l'exécution de la tâche demandant effectivement d. Le caractère différé de cette invalidation est nécessaire puisque l'insertion des tâches est non bloquante.

5.1.3 Implémentation

Dans cette section, nous décrivons les modifications nécessaires dans la librairie $\mathcal{H}$-Matrice pour l'utilisation de StarPU en mémoire partagée, ainsi que l'implémentation réalisée. Certaines des modifications requises sont spécifiques à une version de StarPU, et des contraintes sont susceptibles d'être levées dans le futur. Les principales extensions requises par le modèle de la section précédente sont :

— Association de *tags* MPI et de nœuds de référence $h(d)$ aux éléments $d \in L_0$;

— Partition du graphe de tâches ;

— Communication et gestion par StarPU de sous-ensembles de la matrice.

Les deux premiers points sont spécifiques à l'implémentation actuelle de StarPU, et susceptibles d'évoluer. On note que le partitionnement à priori du graphe par l'utilisateur rigidifie le modèle de programmation par rapport au cas de la mémoire partagée. Les possibilités d'ordonnancement et d'affectation des tâches sont réduites, ce qui n'est pas souhaitable.

Le dernier point est nécessaire, car StarPU est décorrélé de la structure des $\mathcal{H}$-Matrices. Les dépendances distantes nécessitent cependant la communication de sous-ensembles d'une $\mathcal{H}$-Matrice (les éléments de L_0) par les opérations MPI de la section 5.1.1.1. Ces opérations permettent de communiquer des données contenues dans un tableau linéaire, et dont la taille est connue par l'émetteur et le récepteur de la communication. Ces deux contraintes ne sont pas respectées ici. En effet,

— Un élément de L_0 est une structure arborescente, dont le stockage en mémoire n'est pas linéaire ;
— La taille d'une feuille compressée d'une $\mathcal{R}k$-Matrice est susceptible d'évoluer au cours d'un calcul par les opérations de recompression adaptative. Par ailleurs, la taille d'une feuille compressée n'est pas connue avant son assemblage, qui est tardif dans l'implémentation décrite dans le chapitre précédent (priorité faible des tâches d'assemblage, section 4.7.1.3).

5.1.3.1 Partition des données

Dans le modèle retenu par StarPU, il est nécessaire d'associer à chaque donnée d un nœud de référence $h(d)$. Cette affectation doit être faite a priori, et est statique au cours d'une exécution (ou tout du moins, au sein d'un même graphe de tâches).

Dans le cas présent, une donnée est un élément de $L_0(M)$ pour une $\mathcal{H}$-Matrice M donnée, et son affectation à un nœud se fait avant l'assemblage de celle-ci. La solution la plus simple, et celle qui est retenue dans l'implémentation actuelle, est celle de type « tourniquet » (*round-robin* en anglais). Pour chaque élément $\{d_1, \ldots, d_{|L_0|}\}$, on définit l'association :

$$h(d_i) = (i\ [p]) + 1$$

avec $[\cdot]$ l'opérateur modulo et p le nombre de nœuds.. Cette affectation est bien entendu très simpliste, et il serait utile de l'améliorer. Pour cela, un rapprochement avec la sélection de la coupe L_0 est utile.

Rapport entre le choix de L_0 et de h Le choix de L_0, qui est l'objet de la section 4.7.1.1 a en particulier comme objectif d'équilibrer la granularité des tâches effectuées sur une $\mathcal{H}$-Matrice. Pour cela, le choix de la position de L_0 s'appuie sur une estimation heuristique de la quantité de travail associée à chaque nœud d'une $\mathcal{H}$-Matrice, en lui attribuant un score.

Par ailleurs, les objectifs du partitionnement des données sont de minimiser les communications (en conjonction avec le partitionnement du graphe de tâches), et d'égaliser la quantité de travail et de mémoire nécessaire sur les différents nœuds. Ce second objectif est proche de celui poursuivi dans la construction de L_0, et il est probable qu'une bonne heuristique attribuant un score significatifs aux éléments de L_0 puisse également être utilisée pour équilibrer la charge entre les différents nœuds d'un calcul. Ce point n'a toutefois pas été exploré, et fera l'objet de travaux ultérieurs.

5.1.3.2 Partition du graphe de tâches

Dans le chapitre précédent, un soin particulier a été apporté à la minimisation du nombre de dépendances en écriture des diverses tâches, avec une seule dépendance en écriture pour la majorité. Dans ce cas, le comportement par défaut de StarPU est d'affecter la tâche au nœud possédant son unique dépendance en écriture.

Ce comportement est en première approximation raisonnable. En effet, dans le cas où une tâche est exécutée sur un nœud ne possédant pas sa dépendance en écriture d, le

nombre de communications minimal est 2. Il est nécessaire de recevoir d avant l'exécution de la tâche, et de renvoyée cette donnée modifiée après. En revanche, dans le cas où la tâche est exécutée sur le nœud $h(d)$, le nombre minimal de communications est 0.

Ceci est d'autant plus important avec l'utilisation du cache de communication. Son utilisation permet en effet de grandement réduire le nombre de communication pour les données non modifiées, réduisant à 0 le nombre de communications nécessaires pour un grand nombre de tâches.

5.1.3.3 Communication des données

Les données dont le moteur d'exécution a la connaissance sont des éléments de $L_0(M)$ pour une $\mathcal{H}$-Matrice M. Ce sont donc des sous-arbres, dont la taille des données n'est pas connue au moment de l'insertion des tâches, qui se fait avant que toute opération ne soit effectuée. Il y a donc deux difficultés :

1. Les données ne sont pas contiguës en mémoire ;
2. La taille des données n'est pas connue au moment de la soumission de la tâche, et n'est pas connue par le récepteur au moment de la réception de la donnée.

Dans l'implémentation initiale de StarPU-MPI, il n'était pas possible de gérer ces deux problèmes. Des extensions ont été ajoutées afin de permettre de supporter ce type de configuration, à savoir :

1. Ajout de fonctions d'empaquetage et de dépaquetage des données définies par l'utilisateur ;
2. Gestion de messages de taille inconnue par envoi d'un message de contrôle.

Données non contiguës Le modèle retenu par StarPU pour la gestion de la mémoire distribuée repose sur la connaissance de l'intégralité du graphe de tâche par tous les nœuds participant à un calcul. Pour cela, les mêmes données doivent être déclarées sur tous les nœuds, et la même séquence d'appels à `insert_task()` doit être faite.

Ceci invite donc à conserver le squelette des données sur tous les nœuds. Dans le cas des $\mathcal{H}$-Matrices, les arbres de groupes et arbres de blocs sont construits sur tous les nœuds. La structure des données (structure de l'arbre et type de feuille) est donc connue par l'émetteur et le récepteur. En particulier, l'ordre d'énumération des feuilles est le même sur tous les nœuds.

Le problème de la transmission de données non contiguës est traité par StarPU au travers de deux fonctions définies par l'utilisateur, et appelées par le moteur d'exécution avant envoi et après réception. Ces deux fonctions symétriques, appelées `pack()` et `unpack()`, doivent respectivement copier une donnée dans un tableau, et reconstituer une donnée à partir d'un tableau. On note que dans tous les cas, le contenu des tableaux reste opaque du point de vue de StarPU.

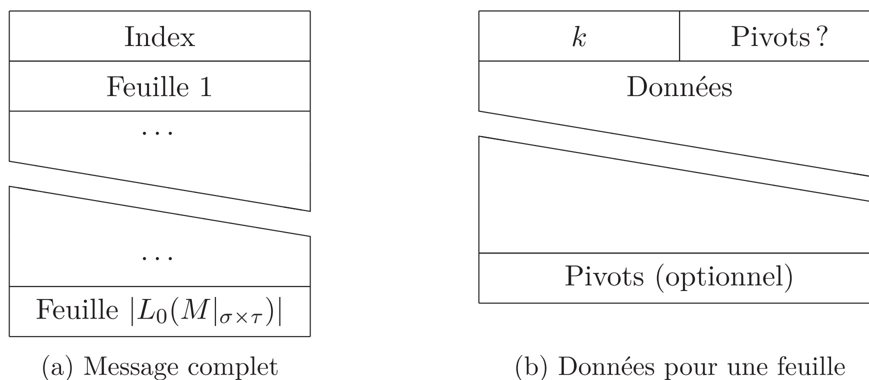
Pack La construction du tableau de donnée est faite en deux étapes pour une donnée $d \in L_0(M)$, représentant un bloc $M|_{\sigma\times\tau}$:

1. Énumération des feuilles $\mathcal{L}(M|_{\sigma\times\tau})$ et construction d'un index au début du tableau ;

2. Pour chaque feuille $M|_{\sigma'\times\tau'} \in \mathcal{L}(M|_{\sigma\times\tau})$, copie de ses données au bon endroit dans le tableau.

La première étape a un temps d'exécution proportionnel à $|\mathcal{L}(M|_{\sigma\times\tau})|$ et est très courte, la seconde est essentiellement limitée par la performance de la mémoire locale (de l'ordre de quelques Go par seconde et par cœur sur une architecture récente).

Pour les feuilles compressées $\mathcal{R}k$-Matrices, le rang n'est pas connu par le récepteur, et doit donc figurer dans l'en-tête. Par ailleurs, dans le cas d'une feuille non compressée placée sur la diagonale, il faut aussi transmettre le tableau des pivots, dans le cas où cette feuille a été factorisée par LU (puisque l'opération de base de factorisation dans LAPACK utilise le pivotage partiel).



(a) Message complet (b) Données pour une feuille

FIGURE 5.4 – Composition des messages

La structure des messages est représentée sur la figure 5.4. Le message commence par un index de taille $8|\mathcal{L}(M|_{\sigma\times\tau})|$ contenant un tableau de décalage des données des feuilles. En notant $\mathcal{L}(M|_{\sigma\times\tau}) = \{M_1, \ldots, M_s\}$ les feuilles et $|M_i|$ l'espace mémoire requis pour le codage de la feuille M_i, alors l'élément i du tableau d'index vaut $8s + \sum_{l=1}^{i-1} |M_i|$.

Le champ k dans la structure des données d'une feuille vaut -1 dans le cas d'une feuille pleine, et l'indicateur de présence des pivots permet du côté du récepteur de connaître la taille totale de la feuille, qui est différente en fonction de la présence ou non des pivots. Finalement, la taille totale du paquet de données est :

$$16|\mathcal{L}(M|_{\sigma\times\tau})| + \sum_{l=1}^{|\mathcal{L}(M|_{\sigma\times\tau})|} |M_i|$$

en supposant que la taille des données supplémentaires pour chaque feuille soit égale à 8 (ce qui est le cas sur la majorité des architectures). Ceci donne un surcoût très faible, égal à un nombre complexe double précision par feuille. Ce faible surcoût est rendu possible par la connaissance de la structure de la $\mathcal{H}$-Matrice (arbre de groupes et arbre de blocs) du côté du récepteur, permetttant d'économiser de la place dans l'en-tête des données.

Remarque 5.3 (Liste de feuilles)**.** *La liste des feuilles d'un nœud $M|_{\sigma\times\tau} \in L_0(M)$ ne change jamais au cours du calcul. Il est donc possible d'optimiser la première étape (création de la liste des feuilles) en gardant en mémoire cette liste dans tous les cas, accélérant ainsi le calcul de la taille et la construction de l'index.*

Messages de taille inconnue Dans le modèle de communication de MPI présenté à la section 5.1.1.1, le récepteur doit connaître la taille d'un message avant sa réception. Ceci n'est pas possible ici, et la solution la plus générique consiste à échanger deux messages par communication :

1. Un message de contrôle, contenant un identifiant unique à la donnée et la taille de celle-ci. La taille de ce message est fixe.
2. Un message contenant les données.

Du côté du récepteur, une première réception asynchrone est soumise pour le message de contrôle. Lorsque cette réception est terminée, une seconde réception asynchrone avec la taille contenue dans le message de contrôle est soumise.

Ceci est potentiellement pénalisant dans le cas où le message contenant les données est de petite taille, puisque la latence est au moins doublée par un tel mécanisme. De plus, la soumission de la réception du second message doit attendre que le premier soit réceptionné, ce qui réduit la fenêtre de recouvrement possible pour les calculs et les communications. Il faut néanmoins noter que des dispositions dans la norme MPI et des optimisations dans les implémentations permettent de réduire l'impact de ceci. En particulier, il n'est pas nécessaire pour l'émetteur d'attendre la réception du message de contrôle pour envoyer le message de données.

Il est de plus possible d'anticiper le plus possible l'envoi du message de contrôle. Dans le paragraphe précédent, la construction du message de données est séparée en deux étapes : construction de l'index et copie des données. La construction de l'index est très rapide, et à l'issue de cette étape la taille du message de donnée est connue. En revanche, la copie des données peut prendre un temps non négligeable, comparable dans certaines situations au temps de transfert des données sur le réseau. La séquence d'envoi des données devient :

1. Calcul de la taille du message ;
2. Envoi du message de contrôle ;
3. Copie des données dans le tampon d'envoi ;
4. Envoi du message de données.

5.1.4 Performances

Dans cette section, nous donnons des résultats de performance obtenus avec le formalisme détaillé dans la section précédente. Il faut noter que ces résultats ne sont pas définitifs. En effet,

- Le choix de la fonction $h(d)$ est naïf. Ceci influence le nombre et la taille des communications à effectuer, pénalisant ainsi le passage à l'échelle.
- L'implémentation des communications dans StarPU 1.1 n'est pas optimisée, particulièrement dans les situations avec un grand nombre de données, et pour les données de taille variable. En particulier, l'implémentation actuelle nécessite qu'un fil d'exécution soit dédié aux communication réseau. Ce fil occupe complètement un cœur puisqu'il effectue une attente active.
- Enfin, un algorithme d'ordonnancement prenant en compte la localité des données permettrait d'améliorer les performances, en anticipant les échanges de données.

Ces limitations sont temporaires, et font l'objet de travaux actuels, pour le moteur d'exécution et pour la bibliothèque présentée dans ce document.

Les calculs présentés dans cette section sont effectués sur plusieurs nœuds identiques à la machine Curie-16, pour la factorisation LU du plus grand cas de calcul de la section 4.7.2.1 (cône-sphère, 303.888 inconnues). Pour tous les calculs, $|L_0(M)| = 13570$.

5.1.4.1 Nombre de nœuds

Nous considérons trois configurations, utilisant 4, 15 et 16 processeurs par nœud. La première configuration est choisie afin de tester le passage à l'échelle avec le nombre de nœuds en minimisant les effets négatifs liés au manque de parallélisme dans le graphe de tâches. Le second laisse un cœur par nœud disponible pour la gestion des communications, et le dernier explore les conséquences de l'utilisation complète de la machine de calcul. Dans ce dernier cas, il faut noter qu'il existe nécessairement une compétition pour les ressources de calcul entre le fil d'exécution dédié aux communications et les fils de calcul.

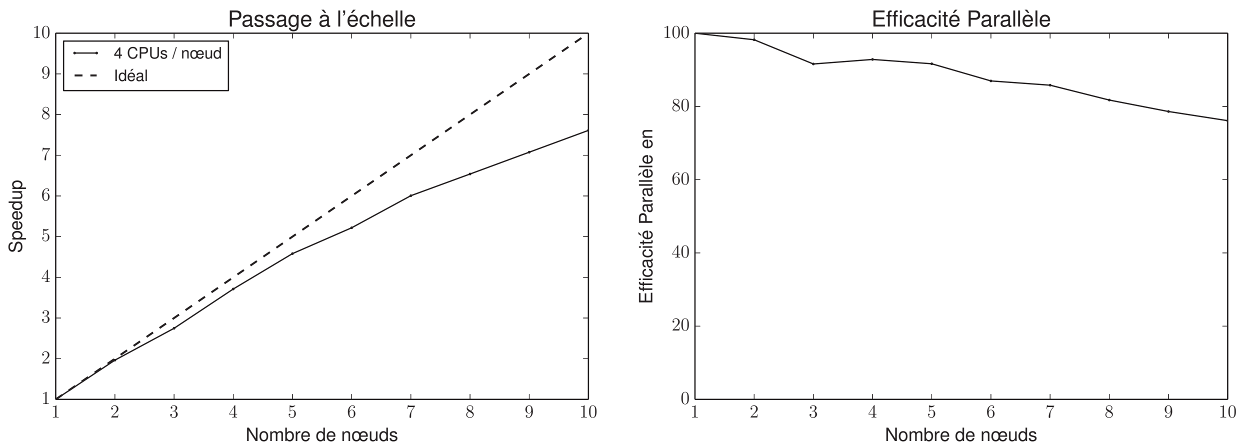


FIGURE 5.5 – Passage à l'échelle avec 4 cœurs utilisés par nœud.

La figure 5.5 représente l'accélération et l'efficacité parallèle avec 4 cœurs utilisés par nœud, soit entre 4 et 40 processeurs au total. On constate un bon passage à l'échelle, avec une efficacité parallèle de 76% sur 10 nœuds. Il est intéressant de noter que cette efficacité est comparable à celle obtenue dans la table 4.6 pour 40 processeurs en mémoire partagée. Les architectures matérielles étant différentes, une comparaison précise n'est pas possible. Cependant, cela signifie intuitivement que le coût des communications est compensé par la meilleure localité des données dans le cas de la mémoire partagée, et par un encombrement plus faible des ressources partagées. C'est donc une indication qu'un moteur d'exécution prenant en compte les contraintes de localité a des chances d'améliorer les résultats de passage à l'échelle en mémoire partagée. Par ailleurs, on rappelle que l'efficacité liée au graphe de tâche $e_p(p)$ n'était pas parfaite pour 40 processeurs sur la figure 4.30. Le surcoût lié à la gestion des communications est donc relativement restreint pour cette configuration matérielle et logicielle.

La figure 5.6 compare le passage à l'échelle avec 15 et 16 fils de calcul par nœud. Le nombre total de processeurs est ici compris entre 15 et 90 dans le premier cas, et 16 et

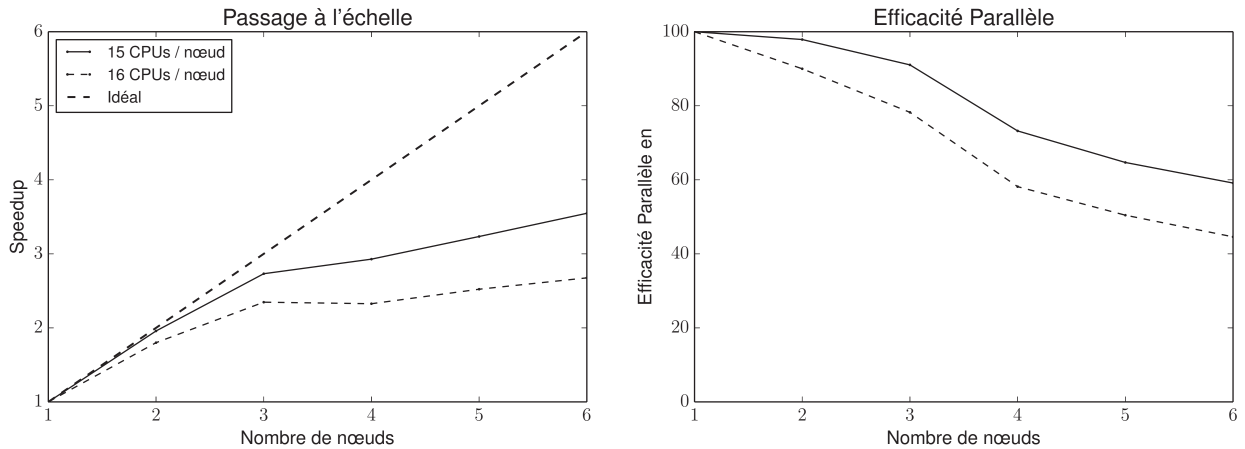


Figure 5.6 – Passage à l'échelle avec 15 et 16 cœurs utilisés par nœud.

96 dans le second. Dans le cas où 16 fils de calcul sont utilisés par nœuds, un des cœurs a la charge du calcul et des communications en même temps. Néanmoins, pour un calcul sur un seul nœud (et donc sans communications MPI), le fil de communication n'utilise pas de ressources, et son impact négatif est nul. Il est donc naturel d'observer une moins bonne efficacité parallèle dans ce cas, ce qui est confirmé par l'expérience.

Le passage à l'échelle est ici moins bons, spécifiquement pour un nombre de nœuds supérieur à 4 (et donc plus de 60 processeurs). Les causes de cette perte d'efficacité sont la combinaison de plusieurs facteurs :

(a) Le coût des communications ;
(b) Le manque de parallélisme dans le graphe de tâche ;
(c) Une partition non optimale du graphe de tâche ;
(d) Le manque de liberté d'ordonnancement lié à la partition statique du graphe de tâches.

L'impact de ces divers facteurs n'a cependant pas pu être mesuré, et leur coût peut être réduit par, respectivement :

(a) L'optimisation de la gestion des communications dans StarPU, et la réduction de leur nombre (choix de $h(d)$) ;
(b) Un meilleur choix de L_0 ;
(c) Un meilleur choix de $h(d)$;
(d) L'évolution du modèle de distribution des tâches vers une partition dynamique du graphe de tâches.

5.1.4.2 15 ou 16 fils par nœud ?

Les résultats de passage à l'échelle renseignent sur l'efficacité de l'utilisation des ressources de calcul. Néanmoins, à nombre de nœuds égal, le facteur de décision est le temps total de calcul, et non l'efficacité d'exploitation des ressources. En effet, certaines processeurs tels ceux utilisés pour Curie-16 intègre la fonctionnalité *Hyper-threading*. Celle-ci permet de simuler l'existence de deux cœurs de calcul virtuels pour chaque cœur physique.

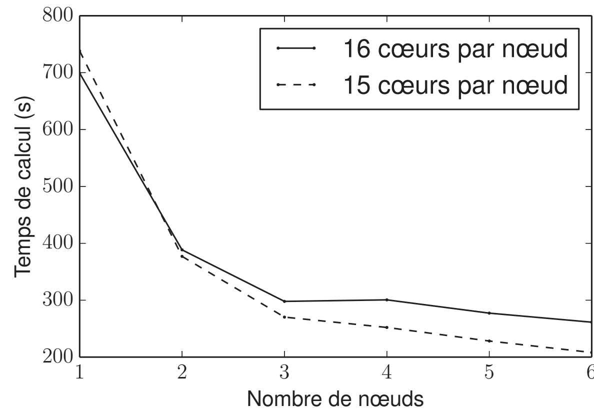


Figure 5.7 – Comparaison du temps de calcul pour 15 et 16 fils de calcul par nœud.

Lorsqu'un cœur virtuel ne peut pas effectuer d'opérations (étant bloqué par un accès mémoire par exemple), les instructions affectées à l'autre cœur virtuel peuvent s'exécuter, sous certaines conditions. Dans certaines circonstances, ceci permet d'augmenter les performances. Les opérations du fil de communication étant de nature à causer de nombreuses interruptions, il est possible que l'utilisation de cette technique permette d'augmenter les performances. La figure 5.7 montre qu'il n'est pas plus avantageux d'utiliser 16 fils de calcul par nœud dans ce cas. Cette observation est dépendante de la configuration matérielle ainsi que du moteur d'exécution, et les conclusions pourraient changer à l'avenir.

5.1.4.3 Quantité de communications

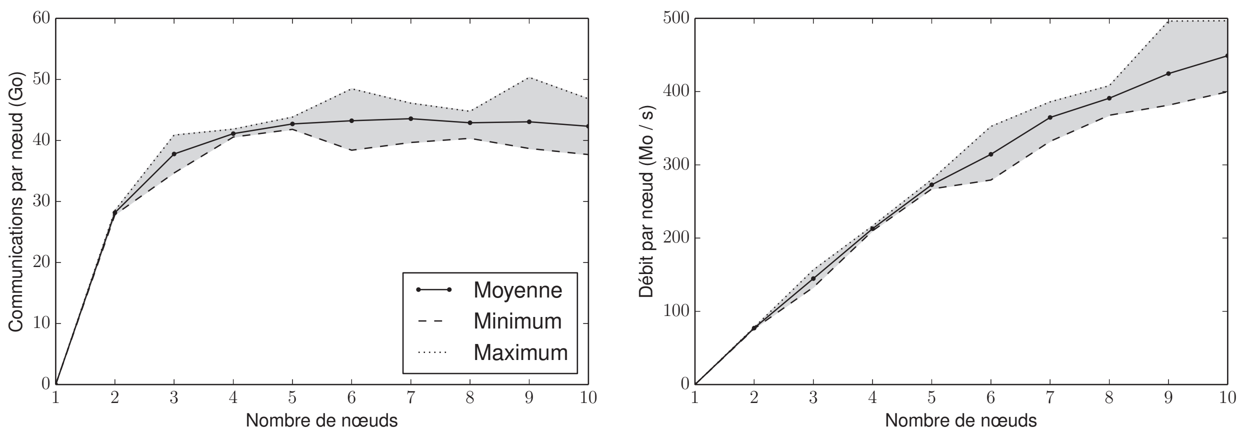


Figure 5.8 – Quantité et débit des communications en fonction du nombre de nœuds. Le débit est calculé par rapport au temps total de calcul.

La figure 5.8 représente l'évolution du volume et du débit des communications en fonction du nombre de nœuds. Le volume de données échangées au cours du calcul croît avec le nombre de nœud, ce qui est attendu. En effet, la proportion des données non locales à un nœud grandit avec le nombre de nœuds participant à un calcul. Ceci implique

nécessairement une augmentation du débit de communication, puisque le temps de calcul global décroît avec le nombre de nœuds.

On note que les débits de communication sont importants. Ceci signifie que l'implémentation proposée ici est dépendante de la présence d'un réseau de communication performant, comme InfiniBand. Il ne serait pas possible d'obtenir des résultats satisfaisants avec une interconnexion de type ethernet 1Gb, offrant une bande passante de l'ordre de 100 Mo/s.

On constate par ailleurs une variabilité des quantités de communication par nœud. Cette variabilité est pénalisante, car révélatrice d'un déséquilibre entre les différents nœuds de calcul. De plus, le facteur limitant le passage à l'échelle est le débit maximal de communication. Il est donc important de réduire le déséquilibre : un découpage plus intelligent du graphe de tâches (au travers de $h(d)$) serait ici indiqué.

5.1.4.4 Conclusions

Nous avons présenté une implémentation en mémoire distribuée des algorithmes opérant sur les $\mathcal{H}$-Matrices. L'expression des algorithmes sous forme de graphe de tâches et l'utilisation d'un moteur d'exécution permet d'étendre une implémentation de la mémoire partagée à la mémoire distribuée simplement. En effet, les extensions détaillées dans la section 5.1.3 ne sont ni majeures ni intrusives. De plus, l'expression n'est en aucun cas adhérente à un ensemble d'appels MPI, ce qui permet de bénéficier aisément des optimisations introduites dans le moteur d'exécution sans modification du code. Les performances sont satisfaisantes pour un faible nombre de nœuds, malgré des choix non optimaux tant dans l'implémentation présentée ici que dans le moteur d'exécution.

En particulier, il est possible d'améliorer le comportement (réduction du volume de communications) par un choix judicieux de $h(d)$, qui fera l'objet de travaux futurs. Il serait également possible d'introduire un parallélisme à plusieurs niveaux. On rappelle que chaque élément de $L_0(M)$ est un sous-arbre de la $\mathcal{H}$-Matrice M. Dans de nombreux cas, il est possible de continuer à diviser les opérations sous $L_0(M)$, ce qui n'est pas fait pour des raisons de granularité et de suivi des dépendances. Il est néanmoins possible de tracer une coupe $L_1(M)$ sous $L_0(M)$, permettant de diviser les tâches élémentaires. Chaque tâche élémentaire peut ainsi être affectée à plusieurs processeurs. Un tel parallélisme à plusieurs niveaux permet de réduire le nombre de transferts de données entre les nœuds, et d'alléger la charge du moteur d'exécution. Cette piste importante pour le passage à l'échelle sur des calculateurs comportant de nombreux nœuds fera l'objet de travaux ultérieurs.

Enfin, StarPU permet à l'utilisateur (ici le code de calcul) de spécifier un ordonnanceur pour les tâches. Il serait bénéfique d'utiliser un algorithme d'ordonnancement permettant d'anticiper plus en amont les communications, au travers d'indications ajoutées au code de calcul.

5.2 Perspectives : Architectures hybrides et *Out-Of-Core*

Comme l'introduction de ce chapitre le souligne, les architectures de calcul s'orientent ces dernières années vers des solutions hétérogènes, avec un ou plusieurs accélérateurs par

nœud. Ces accélérateurs peuvent prendre diverses formes : cartes graphiques (GPU) reconverties pour le calcul, ensemble de processeurs munis d'unités vectorielles (Xeon Phi), ou co-processeurs spécialisés (SPU sur le processeur Cell BE). Ces architectures permettent d'augmenter significativement la performance des calculateurs, au prix d'ajustements significatifs dans les codes et les algorithmes.

Par ailleurs, le reste de ce document suppose jusqu'à présent que la quantité de mémoire disponible est infinie sur les machines de calcul. Ceci n'est évidemment pas réaliste, et un solveur $\mathcal{H}$-Matrice est bien souvent plus sévèrement limité par la quantité de mémoire disponible que par le temps de calcul. De plus, l'évolution prévisible des calculateurs mène à une dimimution de la quantité de mémoire disponible par unité de calcul. Ce ratio, mesuré en octets par flops (*Floating-Point Operation per Second*) diminuera dans le futur du fait de contraintes de dissipation thermique pour les composants d'un système. Il est donc essentiel d'adapter les algorithmes dans le cas où le système ne possède pas assez de mémoire.

Dans les deux cas, nous donnons ici un plan pour les travaux futurs portant sur le solveur $\mathcal{H}$-Matrice afin de gérer ces deux configurations, illustré par des résultats préliminaires.

5.2.1 Architectures hybrides

5.2.1.1 Contexte

Dans la grande majorité des cas, les accélérateurs partagent les caractéristiques suivantes :

- Espace mémoire séparé de l'espace du processeur central ;
- Faible quantité de mémoire locale rapide, grande latence des communications avec le processeur et faible débit de celles-ci ;
- Importantes capacités de calcul vectoriel, au détriment de l'efficacité sur un code non vectorisé.

Du point de vue du code de calcul, il est nécessaire de planifier en avance les transferts entre la mémoire centrale et la mémoire de l'accélérateur. Ceci permet de masquer la latence et le faible débit de la liaison en effectuant une copie asynchrone. Il est également souhaitable de conserver des copies des données dans la mémoire de l'accélérateur tant que cela est possible, afin de réduire le nombre de transferts. Ces contraintes alourdissent l'écriture d'un code hybride utilisant à la fois des accélérateurs et des processeurs. Néanmoins, le moteur d'exécution peut gérer ces éléments, à condition de pouvoir prendre les décisions de transfert suffisamment de temps en avance.

Les capacités de calcul des accélérateurs sont délicates à gérer du point de vue du code de calcul. En effet, outre les grandes variations entre accélérateurs (simple/double précision, seuils de performance optimale), le langage de programmation privilégié est un langage de bas niveau, non uniforme entre les différents accélérateurs (CUDA, OpenCL, *etc.*). Néanmoins, des bibliothèques permettant d'abstraire ces détails existent. Nous pouvons citer ici cuBLAS [83] pour une implémentation de BLAS pour les accélérateurs nvidia, et MAGMA [15] pour une implémentation de LAPACK.

Nous considérons ici l'exploitation des accélérateurs à l'aide d'un moteur d'exécution, ici StarPU. Celui-ci gère le placement et le transfert des données entre la mémoire centrale et la mémoire de l'accélérateur. Il prend également en charge l'initialisation des accélérateur et l'ordonnancement des tâches sur ceux-ci. Du point de vue de l'utilisateur, l'exploitation d'un accélérateur se fait par la définition d'une implémentation adaptée à un type d'accélérateur d'une tâche. Par exemple, la fonction d'assemblage a deux implémentations : une fonctionnant sur un processeur, l'autre sur un accélérateur compatible CUDA. Dans tous les cas, au début de l'exécution de la tâche, toutes les données sont résidentes et à jour dans la mémoire de l'unité de calcul.

Pour chaque tâche s'exécutant sur un accélérateur, un cœur de calcul est mis à sa disposition, ce qui permet de partager les calculs entre le processeur et l'accélérateur. Il est ainsi possible de ne déporter qu'une partie du calcul sur l'accélérateur, ou de conserver les méta-données en mémoire centrale lorsque les données ont été transférées sur l'accélérateur. Ici, on conservera la structure de l'arbre de blocs en mémoire centrale.

5.2.1.2 Application aux $\mathcal{H}$-Matrices

La section 3.6.2.2 a donné une répartition du temps de calcul pour l'assemblage et la décomposition LU. Le temps de calcul est essentiellement réparti entre deux opérations :

- Assemblage ;
- Recompression.

Pour l'assemblage, l'essentiel du temps de calcul est occupé par le calcul de lignes et de colonnes des blocs matriciels, le reste de l'algorithme ACA occupant une partie négligeable du calcul. Pour la recompression, le temps de calcul est partagé entre le calcul de décompositions en valeurs singulière et de décompositions QR.

Il est donc naturel de cibler ces deux opérations pour une implémentation sur accélérateur graphique. Dans le cas le plus simple, le calcul d'une ligne ou d'une colonne d'un bloc matriciel est une opération fortement vectorisable, ce qui est un atoût. Les décompositions QR sont faites sur des matrices dites « *tall and skinny* » (grandes et minces), ce qui limite le degré de vectorisation possible. Enfin, les calculs des décompositions SVD sont faits sur des matrices de petite dimension ($k \times k$), ce qui est également défavorable pour les accélérateurs.

5.2.1.3 Assemblage

On rappelle l'expression d'un élément A_{ij} de la matrice d'interaction :

$$A_{ij} = \int_{\Gamma \times \Gamma} G(|x-y|) \left(\varphi_i(y) \cdot \varphi_j(x) - \frac{1}{\kappa^2} \boldsymbol{\nabla}_\Gamma \cdot \varphi_i(x) \boldsymbol{\nabla}_\Gamma \cdot \varphi_j(y) \right) dxdy \qquad (5.2.1)$$

Selon la position relative des triangles supports des fonctions de base φ_i et φ_j, le calcul de l'interaction est effectué de manière semi-analytique ou numérique [42, chapitre 2]. L'intégration numérique est utilisée lorsque les supports des fonctions sont éloignés, et correspond à la majorité des éléments de la matrice d'interaction. On utilise alors une

méthode de quadrature de Gauss, s'écrivant pour une fonction intégrable sur le segment $[-1, 1]$:

$$\int_{-1}^{1} f(x)dx \simeq \sum_{i=1}^{n} w_i f(x_i)$$

avec $(x_i)_{i=1,\dots,n}$ les points de Gauss et $(w_i)_{i=1,\dots,n}$ les poids associés. On note que les points et les poids sont indépendants de la fonction intégrée, mais dépendants du domaine d'intégration.

Considérons l'interaction du degré de liberté i avec le degré de liberté j. On note $\mathcal{T}_{i_1}, \mathcal{T}_{i_2}$ et $\mathcal{T}_{j_1}, \mathcal{T}_{j_2}$ les triangles supports de φ_i et φ_j, respectivement [5]. L'équation (5.2.1) se réécrit :

$$A_{ij} = \sum_{l=i_1,i_2} \sum_{m=j_i,j_2} \int_{\mathcal{T}_l \times \mathcal{T}_m} G(|x-y|) \left(\varphi_i(y) \cdot \varphi_j(x) - \frac{1}{\kappa^2} \boldsymbol{\nabla}_\Gamma \cdot \varphi_i(x) \boldsymbol{\nabla}_\Gamma \cdot \varphi_j(y) \right) dxdy$$

On note (w_l^p, x_l^p) les poids et points de Gauss associés au triangle $\mathcal{T}_l$, et (w_m^q, y_m^q) ceux associés au triangle $\mathcal{T}_m$. Une approximation de A_{ij} par quadrature de Gauss s'écrit alors :

$$A_{ij} \simeq \sum_{l=i_1,i_2} \sum_{m=j_i,j_2} \sum_p \sum_q w_l^p w_m^q G(|x_l^p - y_m^q|) \left(\varphi_i(y_m^q) \cdot \varphi_j(x_l^p) - \frac{1}{\kappa^2} \boldsymbol{\nabla}_\Gamma \cdot \varphi_i(x_l^p) \boldsymbol{\nabla}_\Gamma \cdot \varphi_j(y_m^q) \right) \tag{5.2.2}$$

Dans cette expression, les points de Gauss et leurs poids associés dépendent uniquement de la géométrie du maillage, et sont donc précalculés, en $\mathcal{O}(N)$ opérations au début du calcul. Les expressions des valeurs prises par les fonctions tests en ces points sont également précalculées, de même que leurs divergences. On note ces facteurs précalculés :

$$\begin{cases} f_{ilp}^1 := w_l^p \varphi_i(x_l^p) \\ f_{ilp}^2 := w_l^p \boldsymbol{\nabla}_\Gamma \cdot \varphi_i(x_l^p) \\ g_{jmq}^1 := w_m^q \varphi_j(y_m^q) \\ g_{jmq}^2 := w_m^q \boldsymbol{\nabla}_\Gamma \cdot \varphi_j(x_m^q) \end{cases}$$

ce qui permet de réécrire l'équation (5.2.2) sous la forme suivante :

$$A_{ij} \simeq \sum_{l=i_1,i_2} \sum_{m=j_i,j_2} \sum_p \sum_q G(|x_l^p - y_m^q|) \left(f_{ilp}^1 g_{jmq}^1 - \frac{1}{\kappa^2} f_{ilp}^2 g_{jmq}^2 \right)$$

Avec 3 points de Gauss par triangle, le calcul de cette expression nécessite 36 évaluations du noyau, $4 \times 36 = 144$ multiplications et 36 soustractions. De plus, aucun transfert de données n'est nécessaire, en supposant que les données précalculées sont déjà présentes dans la mémoire de l'accélérateur.

Les étapes de calcul sont :

1. Calcul des points de Gauss x_l^p et y_m^q. On note que ces deux familles de points sont confondues pour une matrice ayant les mêmes indices ligne et colonne (arbre de blocs de la forme $\mathcal{T}_{I \times I}$).

5. On se restreint ici pour alléger les notations au cas où une arête n'est partagée que par deux triangles.

2. Calcul des quantités $f^{1,2}_{ilp}$ et $g^{1,2}_{jmq}$. On note que ces deux quantités sont confondues dans les mêmes conditions.
3. Transfert des précalculs sur l'accélérateur.
4. Calcul des lignes et colonnes sur l'accélérateur.

Les phases de pré-calcul ont une complexité $\mathcal{O}(N)$ en temps et en espace, nécessitant $15N$ unités de stockage (nombre à virgule flottante) pour un arbre de groupes de la forme $\mathcal{T}_{I\times I}$.

5.2.1.3.1 Calcul du noyau de Green sur GPU On rappelle (*cf.* section 2.2.5) que le calcul du noyau $G(r)$ est très coûteux sur un processeur courant. Il nécessite en effet une division, un calcul de racine carrée (pour r) et l'évaluation d'une exponentielle complexe. Cette dernière nécessite plus de 100 cycles sur un processeur courant en 2013 [65, Annexe C.3.1]. En revanche, cette opération est beaucoup moins coûteuse sur un accélérateur de type « carte graphique ». En effet, les calculs nécessaires à la représentation de scènes 3D font un large usage des fonctions trigonométriques, qui sont donc fortement optimisées sur ce type de matériel. Il en est de même de la fonction « racine carrée inverse », qui est utilisée pour normaliser les vecteurs. À titre d'exemple, une carte graphique K20 de la génération « Kepler » [6] comporte 480 unités matérielles à même d'exécuter ces instructions [82]. Chaque unité est capable d'effectuer le calcul d'un cosinus, sinus ou racine carrée inverse par cycle [7], ce qui donne un débit maximal de 3.576×10^{11} opérations trigonométriques par seconde en simple précision, contre 7.26×10^{8} pour un nœud complet de Curie-16.

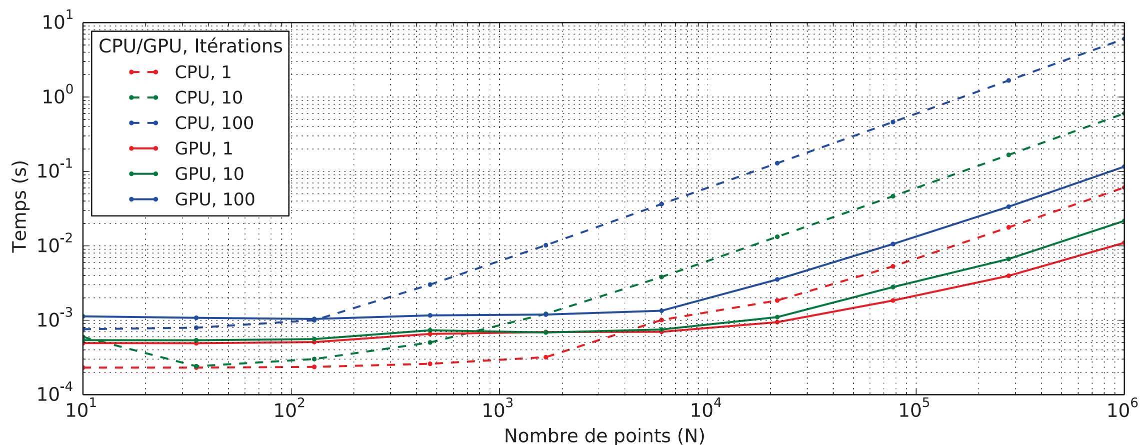


FIGURE 5.9 – Comparaison du temps de calcul de $G(R)$ entre CPU et GPU.

La figure 5.9 donne une comparaison pratique du temps de calcul entre une carte graphique Quadro 2000 de faible puissance [8] avec un processeur Intel Xeon « Sandy Bridge » à 2.3GHz. Cette figure représente le calcul de $\sum_{j=1}^{iter} G(x_i)$ pour $i = 1, \ldots, N$ sur CPU et GPU. Les calculs sont faits en simple précision, et on s'assure que l'erreur relative entre le CPU et le GPU est inférieure à 10^{-6}. Les temps de transferts des données d'entrée (x_i) et

6. Cet accélérateur équipe en particulier la machine Titan mentionnée au début de ce chapitre
7. Il est même possible, à la fois sur une carte graphique et un processeur de calculer le sinus et cosinus du même angle en même temps.
8. Puissance maximale théorique de 480 GFlops en simple précision contre 3519 pour une carte K20.

de sortie depuis et vers le GPU sont comptés dans le temps total. De plus, chaque itération correspond au lancement d'un noyau pour le GPU, avec la latence associée (de l'ordre de 10μs par lancement). Le compilateur utilisé pour le CPU est `icc` 2013, avec le support des instructions AVX et les optimisations activées. On constate un avantage de taille pour le GPU, particulièrement pour un nombre d'itérations élevé.

5.2.1.3.2 Accélération de l'assemblage sur GPU La compression d'un bloc admissible par l'algorithme ACA+ (algorithme 3.5 page 86) utilise les opérations suivantes :

— Calcul de lignes et colonnes de $M|_{\sigma\times\tau}$;

— Recherche de l'élément de norme maximale ;

— Mise à jour de lignes et colonnes.

Il est possible de ne déporter que la première étape sur GPU, celle-ci dominant largement le temps de calcul. Ceci nécessite néanmoins le lancement de $2k + 2$ noyaux sur le GPU, chaque lancement ayant une latence minimale de l'ordre de 10μs. De plus, le reste de l'algorithme nécessite le transfert des données entre le CPU et le GPU, ce qui pénalise fortement cette approche. Une meilleure solution est de déporter l'ensemble de l'algorithme sur GPU. La recherche de l'élément de norme maximale correspond à la fonction ISAMAX de BLAS, et la mise à jour des lignes et colonnes à la fonction AXPY. Il est donc possible d'utiliser les implémentations de cuBLAS de ces fonctions pour déporter complètement l'algorithme ACA+ sur GPU. Néanmoins, ceci nécessite l'appel d'un noyau depuis un autre noyau pour éviter un transfert de l'exécution au CPU, ce qui n'est supporté pour les cartes Nvidia qu'à partir de la génération « Kepler » [82].

5.2.1.4 Recompression

Soit $M|_{\sigma\times\tau} \subset M$ une feuille admissible de M. $M|_{\sigma\times\tau}$ est une $\mathcal{R}k$-Matrice de taille $m \times n$ et de rang k. On rappelle les opérations nécessaires à sa recompression avec le rang k' :

1. Deux factorisations QR de matrices $m \times k$ et $n \times k$;
2. Un produit de deux matrices triangulaires de taille $k \times k$;
3. Une décomposition SVD d'une matrice de taille $k \times k$;
4. La recherche du rang k' ;
5. Deux multiplications de « matrices » Q d'une décomposition QR par des matrices $k \times k'$.

L'ensemble des ces opérations, à l'exception de la quatrième sont des opérations disponibles dans LAPACK. L'accélérations de ces opérations est ainsi directement possible en utilisant une implémentation de LAPACK sur accélérateur comme MAGMA. Néanmoins, la forme des matrices n'est pas adaptées à une exploitation efficace des accélérateurs. Par exemple, la décomposition SVD d'une matrice de petite taille ($k \times k$) est handicapée par le manque de vectorisation et le coût de lancement d'un noyau et de synchronisation des flux.

Une solution utilisable avec CUDA est le recours aux « *streams* », permettant d'exécuter plusieurs noyaux en même temps sur la carte graphique. Rappelons qu'une tâche de recompression opère sur un élément de $L_0(M)$, qui est soit une feuille soit un nœud de M.

Dans le premier cas, cette feuille est souvent de grande dimension (en fonction du choix de $L_0(M)$), et l'exploitation de l'accélérateur ne pose pas de problème de performance. Dans le second, une recompression est en réalité la recompression de toutes les $\mathcal{R}k$-Matrices de $\mathcal{L}(M|_{\sigma\times\tau})$. Il existe ici un parallélisme naturel, puisque chaque recompression est indépendante des autres, et l'utilisation des flux permet d'optimiser le rendement de la carte graphique.

5.2.1.4.1 Modèles de performance L'exploitation des accélérateurs est facilitée par StarPU, en particulier par la prise en charge des transferts entre la mémoire centrale et celle de l'accélérateur. Cependant, du fait de la faible bande passante du bus de communication entre la mémoire centrale et l'accélérateur, l'anticipation des transferts est nécessaire. Ceci est possible avec l'aide d'un algorithme d'ordonnancement adapté, tel que `heft` [18, 96], pour « *Heterogeneous Earliest Finish Time* ». Cet heuristique prend en compte le temps de calcul nécessaire par tâche, ainsi que le temps de transfert des données. Son utilisation nécessite que la connaissance de ces éléments soit accessible à StarPU. Le rôle des modèles de performance est de fournir la première information. Un modèle de performance est formellement une fonction associée à un type de tâche. Celle-ci a les mêmes arguments que la tâche, et renvoie un entier positif représentant le coût de cette instance de la tâche.

Ici, le coût associé peut être le nombre d'opérations en virgule flottante nécessaire pour effectuer une tâche. Ce coût est accessible à partir de la connaissance de la structure de la $\mathcal{H}$-Matrice et du rang de toutes les $\mathcal{R}k$-Matrices. Celui-ci n'étant connu que tardivement (pour chaque dépendance, lorsque celle-ci est « prête » pour l'exécution de la tâche), le calcul du coût doit être réalisé lorsque les dépendances sont disponibles.

La seconde information (coût de transfert des données) est déterminée par l'application (et nécessite également la connaissance des rangs des $\mathcal{R}k$-Matrices). Le coût réel de transfert est ensuite estimé par StarPU en fonction de la performance réelle du bus de communication.

5.2.1.5 Conclusions

Nous avons présenté les principes de l'extension des algorithmes $\mathcal{H}$-Matrices aux architectures hybrides. Nous avons identifié les opérations dont l'accélération est bénéfique, et avons fourni des pistes permettant d'obtenir un gain de performance. Dans le cas de l'assemblage, l'implémentation de l'algorithme ACA+ complet sur GPU permettra une accélération significative. Elle est liée à l'accélération matérielle des calculs trigonométriques sur les GPUs actuels.

Les gains attendus seront particulièrement visibles pour l'acoustique, puisque le support des fonctions tests est dans ce cas plus grand en nombre d'éléments. En effet, dans la formulation retenue par ACTIPOLE, les degrés de libertés relatifs au saut de pression sont des fonctions P1 localisées sur les sommets du maillage. Le suport d'une telle fonction recouvre tous les triangles partageant ce sommet. Dans un cas favorable (triangles équilatéraux), ce nombre est de 6, contre 2 pour fonctions de base vues dans cette section. Le nombre de termes de l'expression équivalente å l'équation (5.2.2) est alors bien plus grand.

Pour la recompression, l'exploitation du parallélisme à l'intérieur d'une tâche semble nécessaire pour atteindre des performances satisfaisantes. Pour ces deux opérations, l'optimisation des transferts de données est souhaitable. Ceci nécessite l'utilisation de modèles de performance, dont nous avons donné une expression possible. L'implémentation sur accélérateurs de ces deux opérations fera l'objet de travaux futurs.

5.2.2 Out-Of-Core

5.2.3 Contexte

La quantité de mémoire disponible sur une machine de calcul est limitée par plusieurs facteurs, dont la consommation électrique. En conséquence, la quantité de mémoire disponible par unité de calcul est bornée, et est bien souvent de l'ordre de 4Go par cœur, ce qui est le cas de la machine Curie-16 utilisée dans les chapitres précédents. La solution pour pallier cette limitation est de modifier l'implémentation pour la rendre *Out-Of-Core*, c'est-à-dire capable d'opérer lorsque toutes les données ne peuvent être contenues en mémoire centrale, mais sur disque uniquement.

Le code de calcul effectue le chargement et déchargement des données en fonction de la quantité de mémoire disponible et des données nécessaires pour effectuer les calculs. La difficulté associée à ces opérations réside dans la latence et le débit de l'espace de stockage. Le tableau suivant donne une comparaison des performances relatives des différents systèmes, pour une machine typique en 2013 :

	RAM	**SSD**	**Disque Dur**
Capacité	16-128Go	64-512Go	1-4To
Latence	< 100ns	< 0.1ms	> 3ms
Débit	> 10Go/s	< 1Go/s	< 200Mo/s

Pour les grappes de calcul (*cluster*), un quatrième type de stockage est souvent disponible sous la forme de disque réseau. Ce sont des disques durs (ou SSD) répartis sur des machines distantes, utilisant un système de fichier distribué. Pour ce type de stockage partagé par toute la grappe, la bande passante est similaire à celle d'un disque local (en l'absence de congestion et pour des transfert de grande taille), mais la latence bien plus élevée. La capacité est en revanche bien plus élevée, de plusieurs Po pour les plus grands systèmes.

Une implémentation efficace doit donc anticiper les transferts entre la mémoire centrale et les disques, de manières à diminuer l'impact de la latence et du faible débit du système de stockage.

5.2.4 Application aux $\mathcal{H}$-Matrices

Grâce à l'expression des algorithmes sous forme de graphe de tâches contenant les dépendances de chaque tâches, il est possible de déterminer l'ensemble des données devant être présentes en mémoire pour l'exécution d'une tâche donnée. De plus, en-dehors des dépendances grossières de la remarque 4.18, l'ensemble des dépendances ainsi exprimées est minimal.

Du point de vue du moteur d'exécution, la mémoire centrale est une zone mémoire à gérer, tout comme la mémoire disponible sur les accélérateurs. Les transferts entre la mémoire centrale et l'espace de stockage nécessitent d'empaqueter les données. Ceci a été présenté à la section 5.1.3.3 pour l'implémentation en mémoire distribuée. Un mécanisme similaire est employé ici.

Les éléments nécessaires pour permettre la gestion d'un espace de stockage externe par le moteur d'exécution sont :

— Lecture et écriture des éléments de L_0 ;

— Modèles de performance donnant une prédiction de temps d'exécution des tâches.

De même que pour l'implémentation sur architecture hybride, le travail de gestion de la mémoire est déchargé au moteur d'exécution, lequel est aidé par les modèles de performance et son ordonnanceur. Néanmoins, il est possible d'indiquer au moteur d'exécution les données non nécessaires dans un futur proche.

Par exemple, dans le cadre d'un calcul complet, on distingue l'étape de factorisation de l'étape de résolution. Ces deux étapes peuvent être séparées par d'autres calculs, dans ce cas la matrice factorisée peut être éliminée de la mémoire une fois son calcul effectué, et placée sur disque. Cette élimination peut être anticipée à l'aide du graphe de tâches, ce qui est illustré par l'algorithme 5.1.

Algorithme 5.1 Éviction de la mémoire vers le disque.

```
DagAssembly(M)
DagLU(M)
for all M|σ×τ in L0(M) do
    InsertTask(WriteToDisk, M|σ×τ, IN | OUT)
end for
WaitForAll()
```

L'insertion des tâches WRITETODISK autorise le moteur d'exécution à écrire les données sur le disque dès qu'elles ne sont plus utilisées par la factorisation. Par ailleurs, ces tâches sont de priorité maximale afin d'avancer le plus possible la libération des ressources.

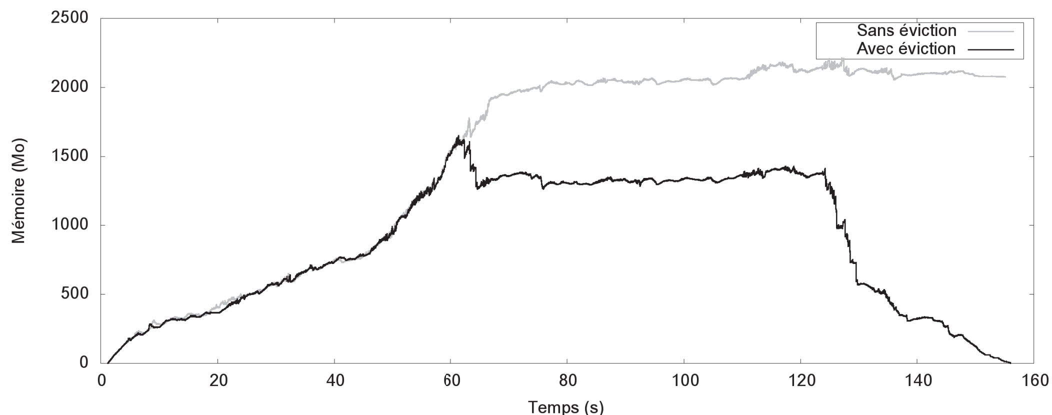


FIGURE 5.10 – Comparaison de la consommation mémoire avec et sans éviction.

Ceci permet de réduire la consommation mémoire maximale de l'algorithme, qui est le point déterminant dans un calcul. La figure 5.10 compare la consommation mémoire d'une factorisation avec et sans éviction des données vers le disque. Il s'agit d'un calcul EFIE sur un A319 à 66.596 inconnues, effectué en parallèle sur 12 cœurs. L'algorithme d'ordonnancement utilisé est `prio`, afin de permettre la prise en compte de la priorité des tâches d'éviction. On rappelle que l'utilisation de cet ordonnanceur augmente la consommation mémoire de la factorisation, comme évoqué section 4.7.1.4. Néanmoins, cette figure montre la diminution de la consommation maximale avec l'ajout des tâches d'éviction.

Afin de réaliser une implémentation efficace de la factorisation *Out-Of-Core*, il est néanmoins nécessaire de modifier plus profondément l'ordonnanceur. En effet, la diminution modeste de la consommation mémoire maximale observée ici est partiellement liée au manque de prise en compte de la consommation mémoire par le moteur d'exécution. Un calcul de grande dimension nécessite donc l'éviction de données allant servir dans la suite de la factorisation. L'ajout d'une éviction manuelle permet cependant de réduire le nombre de ces évictions automatiques, ce qui a un impact positif sur le comportement de l'exécution.

Lors de l'exécution d'un calcul, le nombre de tâches prêtes à être exécutées est supérieur à 1 durant la majorité du calcul. Ceci est illustré par le second graphique de la figure 4.20. Dès lors, en supposant l'existence d'un modèle de consommation mémoire, il est possible de prendre en compte la pression mémoire dans le choix des tâches à effectuer. Un tel modèle ne sera qu'heuristique pour la majorité des tâches, du fait du caractère adaptatif des algorithmes (assemblage et recompression).

5.2.5 Conclusion

Nous avons présenté dans cette section des perspectives d'élargissement de l'implémentation des algorithmes présentés dans ce document aux architectures hybrides, et à une hiérarchie mémoire plus profonde. Dans les deux cas, nous avons présenté des éléments permettant d'entrevoir les gains escomptés et les modifications nécessaires dans les algorithmes.

L'expression des algorithmes sous forme de graphe de tâches a pour avantage de séparer l'expression des contraintes d'exécution de la réalisation d'une exécution conforme à celles-ci. Le choix d'un parcours parallèle particulier dans le graphe de tâches est délégué au moteur d'exécution, au travers d'un algorithme d'ordonnancement. Cet algorithme peut s'appuyer sur des indications supplémentaires et optionnelles pour optimiser les décisions.

Ces indications prennent des formes diverses : priorités dans le chapitre précédent, modèles de performance et de consommation mémoire ici. Du fait du caractère adaptatif des algorithmes, elles prennent la forme d'heuristiques, dont la détermination et la calibration est un sujet non traité dans ce document.

Chapitre 6

Applications

Sommaire

6.1 Exemples industriels **239**
6.2 Autres applications **252**
6.3 Conclusion **263**

Les chapitres précédents ont présenté les principes et la mise en œuvre des algorithmes, indépendamment d'applications réelles, par ailleurs évoquées dans le chapitre 1. Comme précisé plus haut dans ce document, le solveur parallèle $\mathcal{H}$-Matrice est intégré à la suite logicielle Aseris pour l'électromagnétisme, et dans Actipole pour l'acoustique. Ces codes industriels couvrent une large gamme de problèmes, pour lesquels le solveur $\mathcal{H}$-Matrice représente un apport. Ce chapitre présente quelques applications pratiques de ce solveur, ainsi que la comparaison avec les meilleures méthodes disponibles dans la suite logicielle d'accueil, en électromagnétisme et acoustique.

Ces applications appartiennent à deux classes :

- D'une part, les problèmes dont la structure permet le traitement direct par le solveur $\mathcal{H}$-Matrice. Nous illustrons ici la polyvalence et l'efficacité du solveur, et mettons en valeur les différences avec d'autres types de solveurs.
- D'autre part, les problèmes dont la structure requiert des ajouts dans le solveur. Ceci met en valeur la richesse des opérations possibles avec les $\mathcal{H}$-Matrices. En particulier, la résolution de systèmes par blocs permet par exemple le traitement des guides d'ondes et la décomposition de domaine. Ces deux applications sont ici plus directes et efficaces qu'avec un solveur itératif (FMM par exemple). Enfin, nous avons fait un exemple d'application des $\mathcal{H}$-Matrices au domaine des statistiques.

6.1 Exemples industriels

Les exemples de cette section sont issus d'études réalisées à EADS Innovation Works, aussi bien en acoustique qu'en électromagnétisme. Ces applications recouvrent :

— L'étude des diagrammes de rayonnement d'antennes, en isolation ou installées sur une structure de grande dimension ;
— Le calcul de signature radar pour la furtivité ;
— Des problèmes de cavités réfléchissantes de grande dimension ;
— L'émission et la propagation de bruit acoustique.

Nous les présentons ci-dessous.

6.1.1 Antenne VHF sur un avion

FIGURE 6.1 – Courant de surface généré par l'antenne sur un A319.

Un avion utilise de nombreuses antennes pour communiquer et se situer dans l'espace. Les caractéristiques importantes lors de la conception de ces antennes sont la puissance rayonnée (particulièrement relativement à la puissance injectée), et la directivité des émissions. Ces deux quantités sont caractérisées par un diagramme de rayonnement, c'est-à-dire le champ lointain émis par l'antenne dans toutes les directions (θ, φ). L'installation d'une antenne sur un avion modifie le diagramme de rayonnement, nécessitant des essais et ajustements afin d'obtenir les performances souhaitées.

Nous présentons ici le calcul du rayonnement d'une antenne en bande VHF (127MHz pour l'aviation civile) sur un A319neo. La surface de l'avion est modélisée par un conducteur parfait, et le maillage est d'une finesse moyenne de $\lambda/20$. Il comporte 288.108 inconnues. Les conditions de calcul sont les mêmes que celles de la table 3.2 avec une décomposition LU au lieu de LDL^T, et les calculs effectués sur Curie-16. On note que les résultats ont été multipliés par une fonction identique pour tous les calculs, pour des raisons de confidentialité.

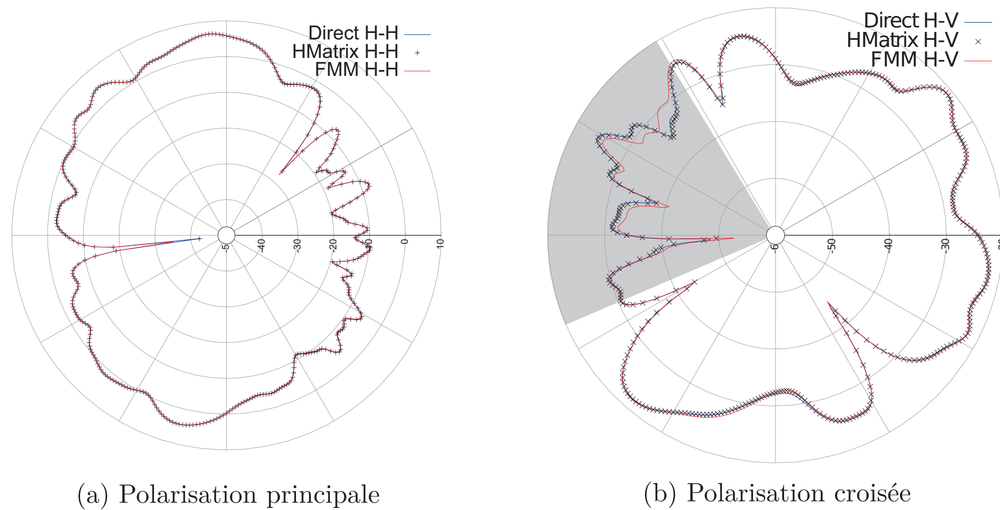
On compare trois solveurs intégrés à ASERIS : un solveur direct, un solveur itératif FMM (tous deux parallèles et *Out-Of-Core*), et le solveur $\mathcal{H}$-Matrice. Les paramètres de calcul importants sont résumés par le tableau suivant :

Solveur	Nombre de processeurs	Paramètres
Direct	5×16	Factorisation LDL^T
FMM	16	$\varepsilon_r = 10^{-4}$
$\mathcal{H}$-Matrice	16	LU, $\varepsilon = 10^{-4}$, $\eta = 3$

Le solveur FMM (décrit dans [94]) utilise MPI pour les communications. Il permet d'accélérer les produits matrice-vecteur d'un solveur itératif GMRES, avec un préconditionneur SPAI. Soit A la matrice d'interaction, b le second membre et x la solution approchée. On note $\tilde{A}x$ le produit vectoriel approché réalisé par la FMM. Le critère d'arrêt est :

$$\left\|\tilde{A}x - b\right\|_2 \leqslant \varepsilon_r \left\|b\right\|_2$$

On rappelle que le champ lointain est calculé à partir de la solution x par intégration (section 2.2.4.2), et que le résidu est en norme euclidienne, et non en norme L_∞.



(a) Polarisation principale

(b) Polarisation croisée

Figure 6.2 – Champ lointain rayonné par l'antenne. Le secteur angulaire pour lequel le solveur FMM diverge significativement du solveur direct est grisé pour la polarisation croisée.

La figure 6.2 compare les diagrammes de rayonnement entre les trois solveurs, la référence étant ici le solveur direct. Pour la polarisation principale, les trois solveurs donnent des résultats presque identiques, à l'exception du point le plus bas qui n'est pas au même niveau pour le solveur FMM. La polarisation croisée est en revanche de meilleure qualité pour le solveur $\mathcal{H}$-Matrice. Les temps de calculs sont :

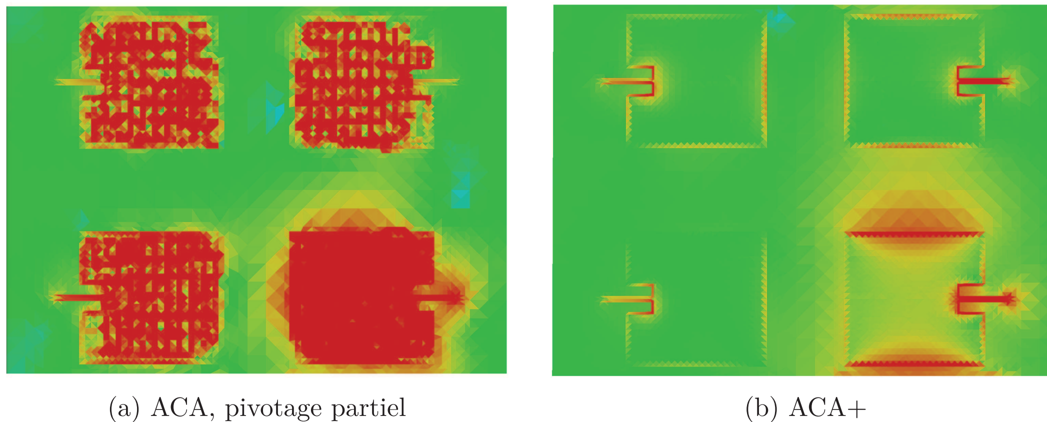
Solveur	**Temps (CPU.s)**	**Ratio (/direct)**
Direct	5850120	-
FMM	7728	757
$\mathcal{H}$-Matrice	8608	679

Le temps de calcul est comparable entre le solveur FMM et le solveur $\mathcal{H}$-Matrice. Il faut néanmoins signaler qu'une factorisation LU est utilisée ici, alors que la formulation utilisée (EFIE) mène à une matrice symétrique. Il serait donc possible d'utiliser une factorisation LDL^T, réduisant ainsi le temps de calcul et la consommation mémoire.

6.1.2 Antenne patch

Outre les effets d'installation, la conception des antennes est un sujet complexe, nécessitant de nombreux essais, réels ou numériques. Dans certaines situations, on souhaite

réduire l'extension verticale d'une antenne. Par exemple, la traînée aérodynamique générée par une antenne n'est pas négligeable pour un avion, et l'utilisation d'antennes planes est ici clairement un avantage. On considère ainsi une antenne patch composée de 4 surfaces métalliques, sur un substrat diélectrique d'indice $\frac{\varepsilon}{\varepsilon_0} = 2.65$. Chaque surface métallique est alimentée par un fil relié à un générateur de tension, et la fréquence d'étude est comprise entre 3.8 et 4.6GHz. Le maillage est ici contraint par le respect de la géométrie plus que par le critère relatif à la longueur d'onde, et le nombre d'inconnues est de 44.292. Le nombre d'éléments par longueur d'onde est compris entre 10 et 111, avec une moyenne de 32. L'antenne est posée sur un plan métallique infini, et on observe son diagramme de rayonnement dans le demi-espace au-dessus du plan de masse.



(a) ACA, pivotage partiel (b) ACA+

FIGURE 6.3 – Comparaison de $\log(\|\boldsymbol{j}\|)$ sur l'antenne patch, le générateur en bas à droite est alimenté dans les deux cas. La représentation est saturée et utilise la même échelle sur les deux figures pour mettre en évidence les différences, et l'échelle est volontairement manquante.

Du fait des choix de modélisation utilisés par ASERIS, la présence d'une métallisation sur un substrat diélectrique entraîne le placement de degrés de liberté « sur » et « sous » la métallisation. Ces degrés de liberté n'interagissent pas entre eux. Il est alors possible d'avoir des blocs de composition similaire à celui de la figure 3.9 page 84, ceci étant un contre-exemple connu pour l'algorithme ACA avec pivotage partiel.

La structure des blocs est en effet problématique, comme le montre la figure 6.3. Sur le graphique de gauche, le courant surfacique $\boldsymbol{j}$ n'est pas reproduit correctement, donc le résultat du calcul est faux. En revanche, l'utilisation de la compression ACA+ permet d'obtenir un résultat identique au solveur direct. Nous avons donc mis en évidence une situation pratique dans laquelle l'algorithme ACA avec pivotage partiel annonce une convergence sans l'avoir atteinte.

L'utilisation de l'algorithme ACA+ permet d'obtenir les courants attendus, et le diagramme de rayonnement est identique entre le solveur direct et le solveur $\mathcal{H}$-Matrice. Par ailleurs, ces deux solveurs sont également en accord avec des mesures de diagramme de rayonnement sur cette antenne. Ces résultats ne sont pas présentés ici pour des raisons de confidentialité.

Les calculs sont effectués sur une machine comportant 2×6 cœurs Intel « Sandy Bridge » à 2.3GHz, avec les mêmes paramètres que précédemment pour les trois solveurs.

Chaque fil est alimenté séparément, pour un total de quatre seconds membres. Les temps de calculs sont :

Solveur	Temps	Ratio
Direct	1150s[1]	-
FMM	1923.1s	-
$\mathcal{H}$-Matrice	131.2s	8.7 (26.0)[2]

Le solveur FMM a convergé en 363 itérations. Le temps par itération est d'environ 4.1s, ce qui s'explique par le raffinement du maillage.

Pour le solveur $\mathcal{H}$-Matrice, le faible facteur d'accélération par rapport au solveur direct s'explique par la prépondérance du temps d'assemblage dans le calcul par $\mathcal{H}$-Matrice. La présence de matériaux diélectriques augmente le nombre de termes à calculer pour chaque élément de la matrice d'interaction ; la présence du plan de masse le double. La figure 6.4 montre la répartition du temps de calcul par processeur. On constate que le temps d'assemblage représente plus de 70% du temps de calcul, ce qui explique le ratio de temps total par rapport au solveur direct. En ne considérant que le temps de factorisation, le solveur $\mathcal{H}$-Matrice est 26 fois plus rapide que le solveur direct. En revanche, des améliorations importantes des performances de l'assemblage sont possibles, la structure du code existant n'étant pas parfaitement adaptée au solveur $\mathcal{H}$-Matrice. Le portage sur GPU de l'assemblage, évoqué dans la section 5.2.1.3 montre ici tout son intérêt.

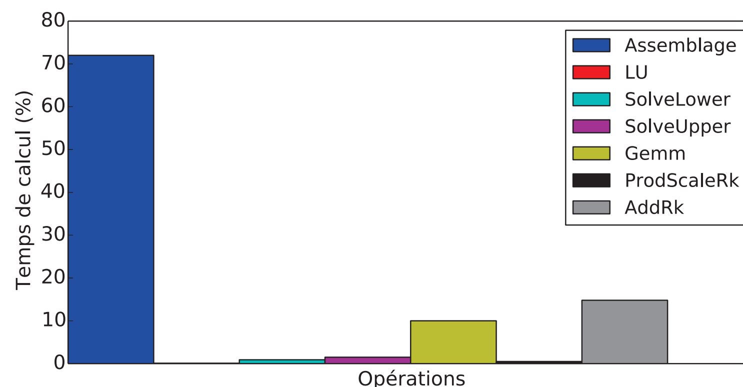


FIGURE 6.4 – Répartition du temps de calcul de l'antenne patch pour le solveur $\mathcal{H}$-Matrice.

6.1.3 Section équivalente radar de missile

Le calcul de signature radar d'un objet est un sujet d'importance pour EADS. L'objectif est double : déterminer celle-ci, et la minimiser. En pratique, un objet est illuminé par une série d'ondes planes de direction d'incidence (θ, φ) variée, et on observe le champ lointain diffracté dans la direction de provenance pour chaque onde illuminant l'objet (calcul de SER monostatique).

1. Assemblage : 126, résolution : 1024s
2. En ne considérant que la factorisation.

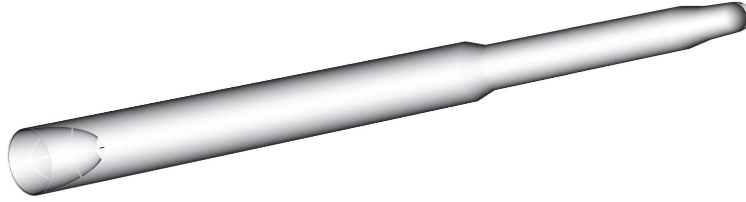


FIGURE 6.5 – Pseudo-missile.

L'objet considéré ici est un pseudo-missile balistique (figure 6.5), composé de deux étages propulsifs. Les divers empennages et éléments de contrôle d'attitude ne sont pas modélisés. Cette omission a pour but d'obtenir un objet ayant une symétrie de révolution pour disposer d'une référence pour les calculs. La longueur de l'objet est de 30m, et le diamètre maximal 2.5m. Il comporte une tuyère à sa base, et est intégralement modélisé par un conducteur parfait.

La fréquence d'étude est 1GHz, et les ondes planes sont échantillonées en θ ($\varphi = 0$) avec un pas de 0.1 degré, pour un total de 3600 seconds membres. L'objet présente une symétrie axiale, et le maillage est relativement homogène avec une taille d'arête de $\lambda/10$. Il comporte 961.452 degrés de liberté.

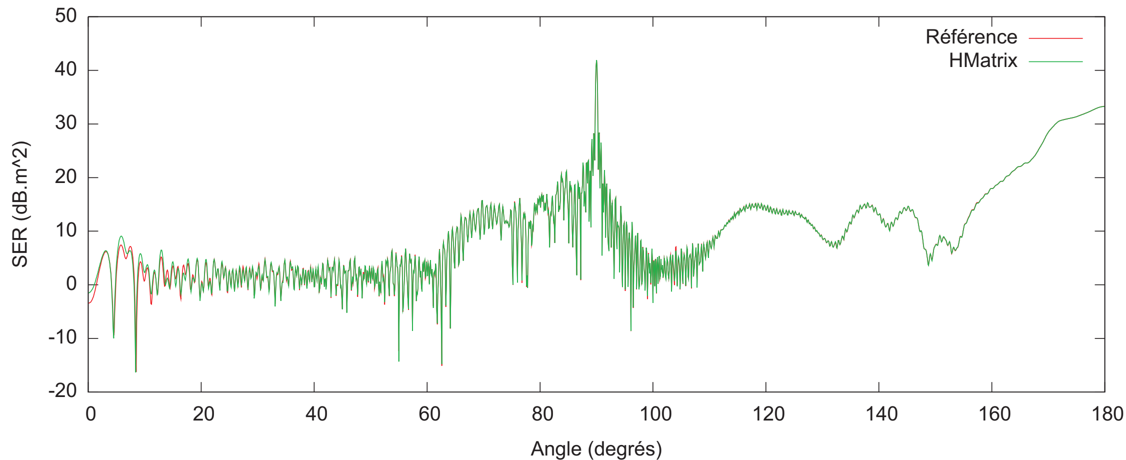


FIGURE 6.6 – SER du pseudo-missile, polarisation horizontale.

Le calcul de référence est fourni par un solveur direct axisymétrique. Celui-ci repose également sur la méthode des éléments finis de frontière, et offre une très grande précision du fait de la prise en compte analytique de la symétrie de révolution. La figure 6.6 compare les résultats des deux solveurs. On observe un très bon accord entre les deux courbes, malgré la grande dynamique et le caractère très accidenté des résultats. Les calculs sont effectués sur la machine Plafrim-40, avec les temps suivants pour le solveur $\mathcal{H}$-Matrice :

Assemblage/Factorisation	1h11m54s
Résolution	10m57s (0.18s par second membre)
Taux de compression	1.01%

En utilisant 2 nœuds de Curie-16 (soit 32 processeurs, plus puissants que ceux de Plafrim-40 [1]) et en réduisant le nombre de seconds membres de 3600 à 6, le solveur FMM converge avec $\varepsilon_r = 10^{-3}$ en 754 itérations, pour un temps de calcul de 1h43m56s. On utilise un solveur GMRES avec *restart* infini afin de ne pas pénaliser la convergence. Le temps nécessaire à un produit FMM est d'environ 5.4s dans cette configuration. L'extrapolation de ces temps au cas complet est délicate, mais la croissance du temps de calcul est approximativement linéaire en le nombre de seconds membres. En ne tenant pas compte des temps nécessaires à l'orthogonalisation de la base de Krylov et à la résolution du problème de minimisation (pour un coût de $\mathcal{O}(n_{rhs}kN)$ à l'itération k), le temps d'un produit FMM serait d'environ 54 minutes par itération pour 3600 seconds membres, soit plus de 678h pour 754 itérations. Ceci signifie que la méthode FMM n'est pas compétitive sur le cas complet, d'autant plus que la forte croissance de la base de Krylov pose des problèmes de stockage et de temps de calcul au sein du solveur itératif.

6.1.4 Méta-surface

Un domaine actif d'étude actuel est celui des métamatériaux. De manière générale, un métamatériau est un matériau composite artificiel ayant des propriétés électromagnétiques particulières. Par exemple, on peut souhaiter obtenir un matériau fortement absorbant dans une bande de fréquence étroite. La création, la caractérisation et l'interaction de tels matériaux est un sujet ardu, ce qui renforce le besoin de simulation et de modélisation.

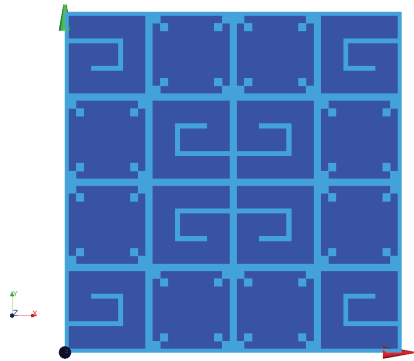


FIGURE 6.7 – Cellule élémentaire du cas 3a du Workshop EM ISAE 2012.

La surface présentée ici est issue du Workshop EM ISAE 2012, cas 3a [7]. Ce colloque bisannuel réunit des membres de la communauté de la simulation électromagnétique, et compare la précision (et performance) des codes de calculs sur des cas sélectionnés pour leur difficulté.

L'objet est une surface comportant une grille de 10×10 un motif élémentaire, représenté sur la figure 6.7. Elle est composée de cuivre sur un substrat diélectrique ($\frac{\varepsilon}{\varepsilon_0} = 4.1$) d'épaisseur 0.2mm posée sur un plan métallique parfaitement conducteur. Cette géométrie est composée de détails bien plus petits que la longueur d'onde. Le maillage est ici contraint par le respect de la géométrie, et non par la fréquence d'étude (entre 6 et 18GHz). On s'intéresse au calcul du champ lointain rayonné lorsque la surface est illuminée par une

1. Le nombre d'opérations maximal théorique est 2.25 fois supérieur pour un cœur de Curie-16 par rapport à un cœur de Plafrim-40.

onde plane en incidence normale, dans cette même direction. Le nombre d'inconnues est 1.300.693, et le temps de calcul est de 1h17mn pour l'assemblage et la décomposition *LU* sur Plafrim-40.

Le solveur FMM n'a pas convergé en 5000 itérations, avec un résidu de 1.1×10^{-2} et 2.7×10^{-2} pour les deux seconds membres après ce nombre d'itérations. Avec 32 cœurs de Curie-16, le temps par itération est d'environ 7s pour le produit FMM, et jusqu'à 15s par itération pour l'orthonormalisation des vecteurs de la base de Krylov. On note que ce temps est croissant avec les itérations, puisqu'aucun *restart* n'est utilisé dans le solveur GMRES. Le temps de calcul total pour les 5000 itérations du solveur FMM est de 20h35mn, illustrant l'avantage de disposer d'un solveur direct, même pour un faible nombre de seconds membres.

Il n'est pas possible de comparer la précision des résultats par rapport au solveur direct, du fait du trop grand nombre de degrés de liberté. En revanche, un participant du Workshop a fourni des résultats obtenus par éléments finis de frontières. À l'aide d'une méthode de décomposition de domaines, une résolution directe de ce problème est rendue possible. Les résultats obtenus par ASERIS avec le solveur $\mathcal{H}$-Matrice sont très proches de ceux obtenus par ce participant avec un solveur direct, et l'écart plus faible que la dispersion des résultats des divers participants.

6.1.5 Cavité avec obstacle interne

Cet exemple est également issu du Workshop EM ISAE 2012, cas 5. L'objet est une cavité métallique rectangulaire de 30cm de hauteur et 77cm de longueur. Celle-ci présente un décrochage de 3mm sur l'intérieur de ses faces internes, visible sur la figure 6.8. L'objectif est de caractériser le champ lointain rayonné par l'objet lorsqu'il est illuminé par une série d'ondes planes d'incidence variée, à une fréquence de 4.5 et 13.5GHz.

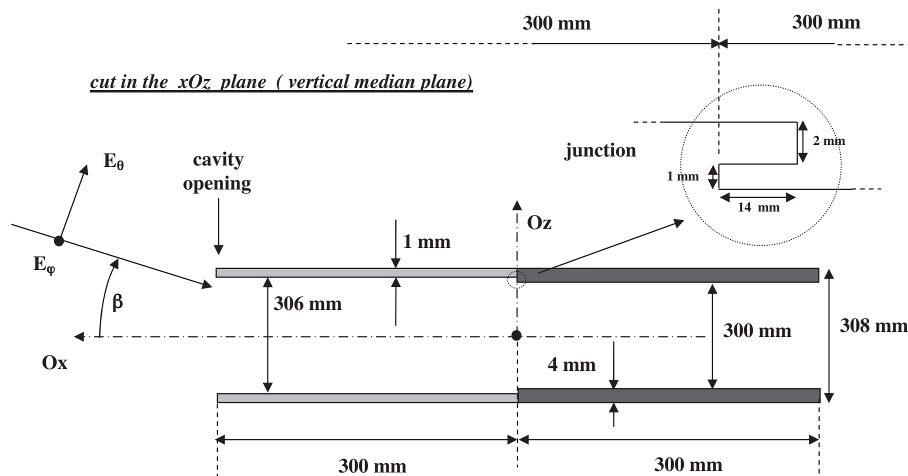


FIGURE 6.8 – Plan de coupe de l'objet

Le maillage adapté pour le problème à 13.5GHz comporte 2.298.669 degrés de liberté, et 482 seconds membres. Les temps de calcul sur Plafrim-40 sont :

Assemblage/Factorisation	6h20m56s
Résolution	13m19s (1.6s par second membre)
Taux de compression	0.64%

Les résultats ne sont pas reproduits dans cette thèse, ils montrent néanmoins un bon accord du solveur $\mathcal{H}$-Matrice avec d'autres solveurs industriels lors de la synthèse des résultats du Workshop.

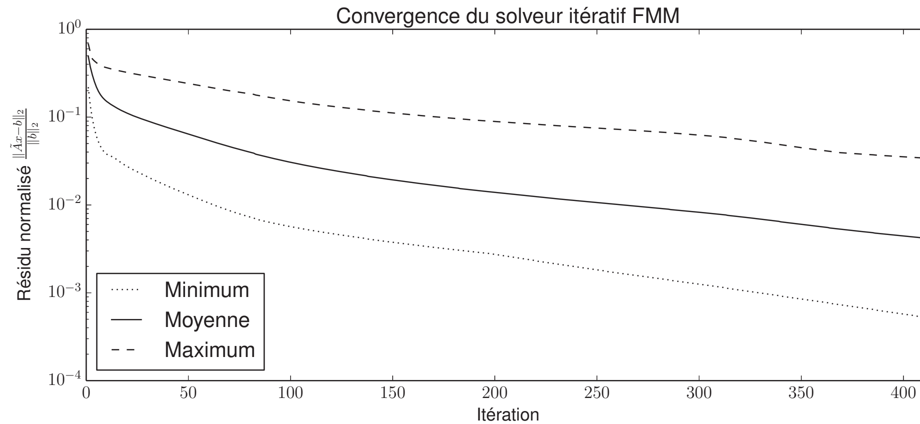


Figure 6.9 – Courbe de convergence du solveur itératif pour le cas ISAE 2012 n°5.

Pour le solveur FMM, on utilise 32 nœuds de Curie-16, en n'exploitant qu'un seul processeur par nœud. Ce choix est effectué afin de ne pas pénaliser les performances des disques locaux aux nœuds de calcul, et de disposer d'une quantité suffisante d'espace disque local. Le solveur BlockGMRES de la suite logicielle Elfipole effectue une décomposition en valeurs singulières tronquée des seconds membres afin de diminuer le nombre de seconds membres à traiter dans le cas où ceux-ci soient presque linéairement dépendants [73]. On obtient ici un problème ne comportant plus que 80 seconds membres, par rapport aux 482 initiaux, soit un gain d'un facteur 6. Dans ces conditions, le temps du produit matrice-vecteur est d'environ 56s par itération, et le temps total d'une itération est dominé par l'orthogonalisation de la base de Krylov. On observe une convergence lente, illustrée par la figure 6.9. Le nombre d'itérations représenté sur cette figure correspond à un calcul de 24h, au bout duquel les résidus normalisés minimum et maximum étaient $\varepsilon_{min} = 5.15 \times 10^{-4}$, $\varepsilon_{max} = 3.39 \times 10^{-2}$. À l'issue de ce calcul, seuls 8 des 80 seconds membres avaient atteint un résidu normalisé inférieur à 10^{-3}.

Cet exemple illustre deux avantages du solveur $\mathcal{H}$-Matrice sur un solveur itératif FMM : d'une part, la capactié à prendre en compte efficacement un grand nombre de seconds membres ; d'autre part, l'affranchissement des problèmes de convergence inhérents aux solveurs itératifs. Notons que la convergence lente du solveur FMM est ici sans doute à rapprocher qualitativement de l'aspect résonnant de la propagation des ondes dans l'objet.

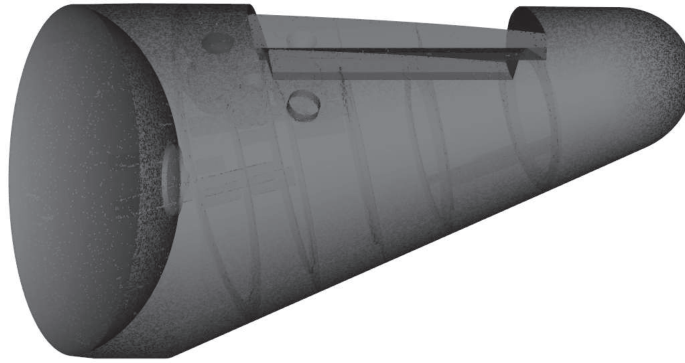


FIGURE 6.10 – Partie supérieure d'un étage d'accélération à poudre d'Ariane 5.

6.1.6 Cavité résonnante

L'étude des $\mathcal{H}$-Matrices est particulièrement motivée par la problématique industrielle décrite au chapitre 1, c'est-à-dire la modélisation des cavités réfléchissantes de grande dimension. Nous donnons ici un exemple d'une telle cavité, issue d'une application industrielle.

La cavité étudiée est la partie supérieure d'un étage d'accélération à poudre (*booster*) de la fusée Ariane 5, dont la géométrie simplifiée est représentée par la figure 6.10. L'objet a un diamètre de 3m et une hauteur de 5.25m, soit une longueur de 40λ à la fréquence d'étude de 2.4GHz. Il est parfaitement conducteur, et le maillage est relativement homogène, avec une taille d'arête moyenne de $\lambda/7$. Le nombre d'inconnues est 1.362.122, et on étudie une unique source ponctuelle placée à l'intérieur de la cavité. La machine de calcul est Plafrim-40 ; on choisit $\varepsilon = 10^{-4}$, $\eta = 2$, un calcul en simple précision, et on effectue une décomposition LDL^T. Les résultats de ce calcul sont :

Assemblage/Factorisation	3h30m
Résolution	41s
Taux de compression	1.84%

On effectue le même calcul avec le solveur itératif FMM sur 32 processeurs de Curie-16, répartis sur 4 nœuds pour les mêmes raisons que pour l'exemple précédent. Le nombre maximal d'itérations est fixé à 5000, ce qui correspond pour cette configuration à un temps de calcul total de 9h30m. Le calcul n'a pas convergé en 5000 itérations avec le solveur FMM, avec une convergence très lente, représentée sur la figure 6.11. Le résidu normalisé final est 1.18×10^{-2}. Un calcul similaire effectué avec une configuration informatique différente montre l'absence de convergence en 20.000 itérations, avec un résidu cible de 10^{-3}. Il n'est par ailleurs pas possible de donner une comparaison avec le solveur direct, du fait du nombre d'inconnues du problème.

Cet exemple est une bonne illustration des difficultés posées par les cavités réfléchissantes de grande dimension, évoquées dans le premier chapitre de ce document. En effet, le calcul du champ par la méthode des éléments finis de frontière n'est pas possible par une méthode itérative, et le nombre d'inconnues empêche l'utilisation d'un solveur direct classique. Par ailleurs, la géométrie complexe de la cavité, et l'absence de sous-structuration simple de celle-ci rend peu applicable des méthodes telles que la SEA ou l'UTD.

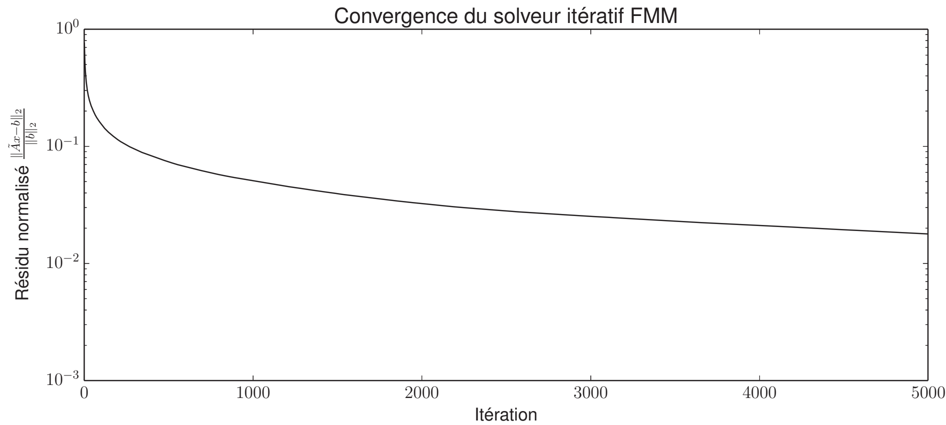


Figure 6.11 – Courbe de convergence du solveur itératif pour Ariane 5.

Cet exemple met donc en évidence un avantage important des $\mathcal{H}$-Matrices sur un solveur itératif, et sa capacité à traiter des problèmes de rayonnement dans une cavité résonnante de grande dimension.

6.1.7 Acoustique : voiture

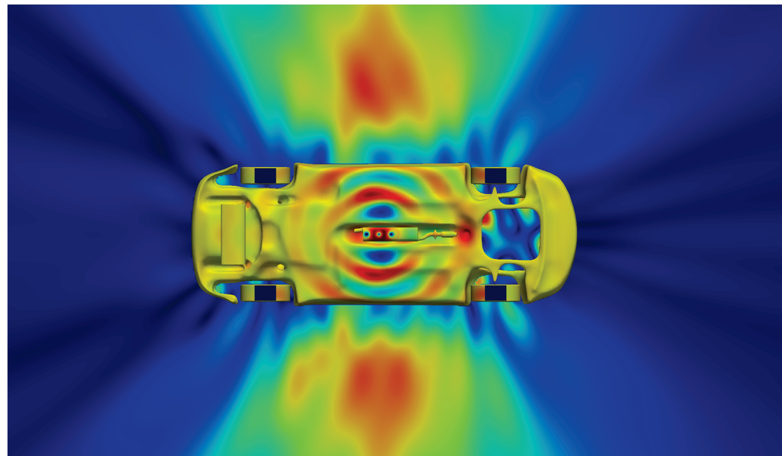


Figure 6.12 – Champ de pression rayonné par le silencieux autour de la voiture.

La nature algébrique de l'algorithme ACA permet d'étendre à faible coût le solveur $\mathcal{H}$-Matrice à d'autres systèmes d'équations physiques, comme l'acoustique. Pour les développements mathématiques associés à l'acoustique, on pourra se reporter à [80, 95]. La méthode des éléments finis de frontière est largement utilisée en acoustique, à l'intérieur et l'extérieur d'EADS, au travers du code Actipole. Dans le domaine aéronautique, il est utilisé en particulier pour le calcul des éléments absorbants disposés à l'intérieur de la nacelle des réacteurs. Une application proche sera présentée à la section 6.2.1.1.4.

Système linéaire pour l'acoustique Nous donnons ici l'expression du système linéaire pour l'acoustique, pour une frontière rigide. On se place dans la situation analogue à celle de la figure 2.1 page 41. On considère un objet fermé borné de frontière $\Gamma \subset \mathbb{R}^3$, soumis à un champ de pression incident p^{inc}.

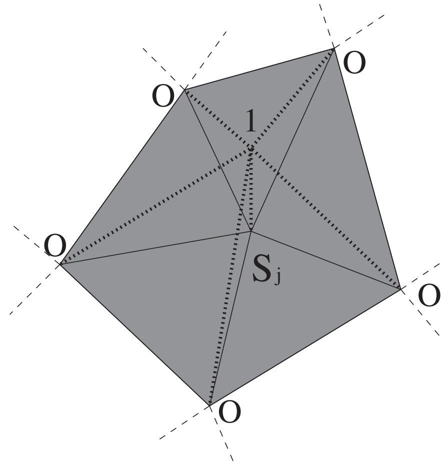


FIGURE 6.13 – Fonction test P^1 pour l'acoustique.

On note $\mu = p|_\Gamma$ la pression inconnue sur la surface, qui est l'inconnue du problème. Les fonctions tests ne sont ici plus des éléments finis de Raviart-Thomas, mais des éléments finis P^1, portés par les sommets du maillage, que la figure 6.13 représente. Une fonction de base (continue et affine par triangle) vaut 1 en un sommet du maillage, et 0 sur les autres. On note $\mu_1, \dots, \mu_N$ une telle base, et le système linéaire $A\mu = b$ analogue au système (2.2.17)-(2.2.18) s'écrit alors :

$$\left\{ \begin{array}{ll} A_{ij} = \dfrac{1}{i\kappa} \displaystyle\int_{\Gamma\times\Gamma} G(|x-y|)\boldsymbol{\nabla}_\Gamma \wedge \mu_i(x) \cdot \boldsymbol{\nabla}_\Gamma \wedge \mu_j(y)dxdy & (6.1.1) \\ \qquad +i\kappa \displaystyle\int_{\Gamma\times\Gamma} G(|x-y|)(\mu_i \boldsymbol{n})(x) \cdot (\mu_j \boldsymbol{n})(y)dxdy & \\ b_j = -\dfrac{1}{i\kappa} \displaystyle\int_\Gamma \dfrac{\partial p^{inc}}{\partial n}(x)\mu_i(x)dx & (6.1.2) \end{array} \right.$$

avec $\boldsymbol{n}$ la normale à la surface Γ.

Application L'application que nous présentons ici correspond à la modélisation du bruit généré par le silencieux d'une voiture. On considère une voiture disposée sur un plan rigide infini, représentée sur la figure 3.43 page 133. La source de bruit est une superposition de monopôles acoustiques localisés au niveau du silencieux, sous la caisse. On souhaite connaître le niveau de bruit rayonné par ces sources de bruit autour de la voiture. La fréquence d'étude est ici de 800Hz, le problème comporte 7 seconds membres et 310.208 inconnues. La taille moyenne des arêtes est de $\lambda/30$ pour cette fréquence.

Les temps de calcul sur la machine Curie-16 sont donnés par :

Assemblage/Factorisation	27mn47s
Résolution	4.5s (0.6s par second membre)
Taux de compression	1.33%

Le solveur FMM a convergé en 127 itérations sur la même machine, pour un temps de calcul de 56mn53s, et l'accord entre les deux solveurs est excellent.

Le temps de calcul du solveur $\mathcal{H}$-Matrice est supérieur à celui pour un cas similaire en électromagnétisme à nombre d'inconnues égal. Ceci est lié à la prépondérance du temps d'assemblage dans le temps de calcul, lequel est plus lent en acoustique. En effet, on rappelle que les fonctions de base ne sont pas, en acoustique, des éléments de Raviart-Thomas. Les degrés de liberté sont portés par les sommets du maillage et non les arêtes. Le nombre d'éléments dans les sommes sur les indices l et m de l'expression analogue à l'équation 5.2.2 est donc dans ce cas de 6, et le nombre d'évaluations du noyau de Green par élément de la matrice est de $6 \times 6 \times 3^2 = 324$, contre 36 en électromagnétisme.

6.1.8 Conclusions et perspectives

Dans cette section, nous avons montré un sous-ensemble des cas d'application du solveur $\mathcal{H}$-Matrice à des problèmes industriels en électromagnétisme et acoustique. Ces exemples couvrent un large spectre de configurations, et mettent en évidence les fonctionnalités du solveur et la polyvalence de la méthode. Du point de vue des fonctionnalités, le caractère « boîte noire » de l'algorithme permet d'avoir les fonctionnalités suivantes :

— Matériaux diélectriques en électromagnétisme, rigides et souples en acoustique ;

— Illuminations variées : onde plane, générateur, sources ponctuelles et leurs équivalents acoustiques, *etc* ;

— Symétries et plans de masse.

Du point de vue des configurations, ont été testées :

— Cavités résonnantes ;

— Grand nombre de seconds membres, avec un coût par second membre très faible ;

— Objets composés de matériaux d'indices différents, avec des détails plus petits que la longueur d'onde ;

Dans les applications de furtivité radar, le nombre de seconds membres visé est bien plus grand que celui possible aujourd'hui. En effet, le missile de la section 6.1.3 a des seconds membres échantillonnés uniquement en θ, alors que l'objectif est de le faire en θ et en ϕ. Une approche prometteuse repose sur la compression des seconds membres, avec l'algorithme ACA. Ceci est fait dans [90] ; l'exploration de cette approche fait partie des travaux futurs. Par ailleurs, il faut noter que tous les calculs présentés dans cette section utilisent la formulation EFIE, et la matrice du système est symétrique. Un gain en mémoire et en temps de calcul d'un facteur 2 est attendu avec l'utilisation de la décomposition LDL^T au lieu de LU.

6.2 Autres applications

6.2.1 Guides d'ondes et décomposition de domaine

Nous présentons ici deux extensions du domaine d'application du solveur $\mathcal{H}$-Matrice. La première concerne la gestion d'une configuration importante en électromagnétisme et acoustique : la propagation d'ondes en sortie d'un guide. La seconde est d'un grand intérêt pratique, permettant de significativement réduire le temps de calcul lorsque plusieurs configurations géométriquement proches sont considérées.

6.2.1.1 Guide d'ondes

On s'intéresse à la propagation d'ondes en sortie d'un guide d'onde. Cette modélisation est courante dans les applications industrielles. En électromagnétisme, on peut citer l'alimentation d'une antenne par un câble coaxial, et en acoustique la modélisation de la nacelle d'un moteur d'avion. Le second cas est plus dimensionnant : la taille de la section du guide par rapport à la longueur d'onde d'étude est grande, et le nombre de modes propagatifs devient important. On note que la présentation de la modélisation correspond à celle de la suite logicielle Aseris, et provient du manuel utilisateur de ce logiciel et de rapports internes (non publiés). Pour plus de détails sur la modélisation dans le cas acoustique, on pourra se reporter à [94].

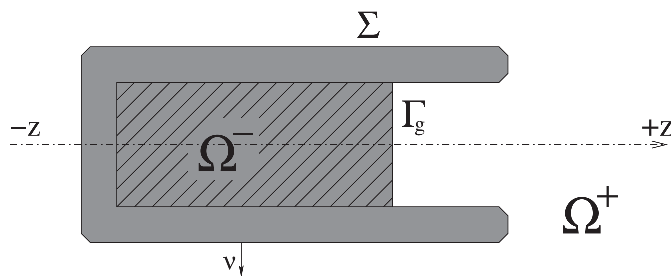


Figure 6.14 – Problème de guide d'onde

6.2.1.1.1 Modélisation On considère la situation représentée par la figure 6.14. Les faces du guide d'onde sont parfaitement conductrices, et séparées par un milieu diélectrique de permittivité et perméabilité (ε, μ). On note Ω^- l'ensemble de la nacelle et du guide et Ω^+ le domaine extérieur infini. Le guide d'onde est représenté en grisé hachuré et la nacelle en blanc. Une frontière fictive Γ_g sépare le guide d'onde du milieu extérieur Ω^+. On note de plus S la frontière de Ω^- et Σ le reste de la frontière de Ω^-, c'est-à-dire $S := \partial\Omega^- = \Sigma \cup \Gamma_g$.

On écrit les équations de Maxwell dans le domaine intérieur :

$$\begin{cases} \mathbf{rot}\boldsymbol{E} - i\omega\mu_0\boldsymbol{H} = 0 & \text{dans } \mathbb{R}^3\backslash\Omega^+ \\ \mathbf{rot}\boldsymbol{H} + i\omega\epsilon_0\boldsymbol{E} = 0 & \text{dans } \mathbb{R}^3\backslash\Omega^+ \\ \boldsymbol{E}\wedge\boldsymbol{n}|_\Gamma = -\boldsymbol{E}_{\text{inc}}\wedge\boldsymbol{n}|_\Gamma & \text{sur } \Gamma, \\ \boldsymbol{j} = \boldsymbol{j}_{\text{inc}} + \boldsymbol{j}_{\text{diff}} = \left[\sum_{m,n}\alpha_{m,n}\boldsymbol{H}^{\text{inc}}_{m,n} + \sum_{m',n'}\beta_{m',n'}\boldsymbol{H}^{\text{diff}}_{m',n'}\right]\wedge\boldsymbol{n} & \text{sur } \Gamma_g \\ \boldsymbol{m} = \boldsymbol{m}_{\text{inc}} + \boldsymbol{m}_{\text{diff}} = -\left[\sum_{m,n}\alpha_{m,n}\boldsymbol{E}^{\text{inc}}_{m,n} + \sum_{m',n'}\beta_{m',n'}\boldsymbol{E}^{\text{diff}}_{m',n'}\right]\wedge\boldsymbol{n} & \text{sur } \Gamma_g \\ \lim_{r\to+\infty} r\left|\sqrt{\varepsilon_0}\boldsymbol{E} - \sqrt{\mu_0}\boldsymbol{H}\wedge\frac{\boldsymbol{r}}{r}\right| = 0 \end{cases} \tag{6.2.1}$$

Les trois premières équations sont les équations de Maxwell dans un milieu caractérisé par (ε_0, μ_0), identiques au système (2.2.4), et la dernière la condition de radiation à l'infini. Les équations 4 et 5 de ce système sont la décomposition des courants électriques et magnétiques sur la base des modes incidents et diffractés.

Précisons ceci. Les champs totaux $(\boldsymbol{E}, \boldsymbol{H})$ s'écrivent sur Γ_g comme une somme de champs incidents $(\boldsymbol{E}_{inc}, \boldsymbol{H}_{inc})$ et diffractés $(\boldsymbol{E}_{diff}, \boldsymbol{H}_{diff})$. Les champs incidents sont une donnée du problème, et s'écrivent sous la forme d'une somme de modes, dits incidents avec les coefficients connus $\alpha_{m,n}$. En effet, la solution des équations de Maxwell dans un guide d'onde de dimension infinie orthogonalement à Γ_g s'écrit sous la forme d'une somme finie de modes propagatifs et d'une somme dénombrable de modes évanescents (au-dessus de la fréquence de coupure). Dans le cadre de cette modélisation, nous ne conservons qu'un nombre fini M_{inc} de modes incidents, et on a donc la décomposition suivante du champ incident sur les modes incidents :

$$\begin{cases} \boldsymbol{E}_{inc} = \sum_{m,n}\alpha_{m,n}\boldsymbol{E}^{m,n}_{inc} \\ \boldsymbol{H}_{inc} = \sum_{m,n}\alpha_{m,n}\boldsymbol{H}^{m,n}_{inc} \end{cases}$$

Les champs diffractés $(\boldsymbol{E}_{diff}, \boldsymbol{H}_{diff})$ s'écrivent de manière analogue sous la forme d'une somme de modes réfléchis avec les coefficients $\beta_{m,n}$:

$$\begin{cases} \boldsymbol{E}_{diff} = \sum_{m,n}\beta_{m,n}\boldsymbol{E}^{m,n}_{diff} \\ \boldsymbol{H}_{diff} = \sum_{m,n}\beta_{m,n}\boldsymbol{H}^{m,n}_{diff} \end{cases}$$

Les conditions aux limites sur Γ_g

$$\begin{cases} \boldsymbol{j} = \boldsymbol{j}_{inc} + \boldsymbol{j}_{diff} = \boldsymbol{H}\wedge\boldsymbol{n} \\ \boldsymbol{m} = \boldsymbol{m}_{inc} + \boldsymbol{m}_{diff} = -\boldsymbol{E}\wedge\boldsymbol{n} \end{cases}$$

donnent alors les conditions aux limites sur Γ_g dans le système (6.2.1).

À ce problème est associé un problème extérieur dans le domaine $\Omega^+ := \mathbb{R}^3 \backslash \Omega^-$:

$$\begin{cases} \mathbf{rot}\boldsymbol{E} - i\omega\mu_0\boldsymbol{H} = 0 & \text{dans } \mathbb{R}^3\backslash\Omega^- \\ \mathbf{rot}\boldsymbol{H} + i\omega\epsilon_0\boldsymbol{E} = 0 & \text{dans } \mathbb{R}^3\backslash\Omega^- \\ \boldsymbol{E} \wedge \boldsymbol{n}|_\Gamma = -\boldsymbol{E}_{\text{inc}} \wedge \boldsymbol{n}|_\Gamma & \text{sur } \Gamma, \\ \boldsymbol{E} \wedge \boldsymbol{n}|_{\Gamma_g} = 0 & \text{sur } \Gamma_g \end{cases} \tag{6.2.2}$$

En utilisant le théorème de représentation intégrale, et en introduisant le courant surfacique normalisé $\boldsymbol{p} = \frac{\boldsymbol{m}}{Z_0}$ on obtient pour tout couple de fonctions tests $(\boldsymbol{j}^t, \boldsymbol{p}^t)$:

$$\begin{cases} -i\kappa Z_0 Z_{\mathrm{r}} \int_\Gamma \boldsymbol{\mathcal{S}_\Gamma j}.\boldsymbol{j}^t d\Gamma - Z_0 \int_\Gamma \boldsymbol{\mathcal{R}_{\Gamma_g} p}.\boldsymbol{j}^t d\Gamma + \frac{Z_0}{2} \int_\Gamma (\boldsymbol{n} \wedge \boldsymbol{p}).\boldsymbol{j}^t d\Gamma = 0 \\ \frac{i\kappa Z_0}{Z_{\mathrm{r}}} \int_{\Gamma_g} \boldsymbol{\mathcal{S}_{\Gamma_g} p}.\boldsymbol{p}^t d\Gamma_g - Z_0 \int_{\Gamma_g} \boldsymbol{\mathcal{R}_\Gamma j}.\boldsymbol{p}^t d\Gamma_g + \frac{Z_0}{2} \int_{\Gamma_g} (\boldsymbol{n} \wedge \boldsymbol{j}).\boldsymbol{p}^t d\Gamma_g = 0 \end{cases} \tag{6.2.3}$$

avec $\mathcal{S}_\Gamma$ et $\mathcal{R}_\Gamma$ les opérateurs de simple et double couche, définis par :

$$\begin{cases} \mathcal{S}_\Gamma \boldsymbol{j}(x) = \int_\Gamma \{G(|x-y|)\boldsymbol{j}(y) + \frac{1}{\kappa^2} \boldsymbol{\nabla_x} G(|x-y|) \mathrm{div}_\Gamma \boldsymbol{j}(y)\} dy & , x \in \mathbb{R}^3 \backslash \Gamma \\ \mathcal{R}_\Gamma \boldsymbol{j}(x) = -\mathbf{rot}_x \int_\Gamma G(|x-y|)\boldsymbol{j}(y) dy = \int_\Gamma \boldsymbol{\nabla_y} G(|x-y|) \wedge \boldsymbol{j}(y) dy & , x \in \mathbb{R}^3 \backslash \Gamma \end{cases} \tag{6.2.4}$$

6.2.1.1.2 Discrétisation On utilise les discrétisations suivantes pour les courants sur Γ :

$$\boldsymbol{j} = \sum_{k=1}^{n_{\mathrm{dl}}} \lambda_k \boldsymbol{\varphi_k} + \left[\sum_{m,n} \alpha_{m,n} \boldsymbol{H}_{m,n}^{\mathrm{inc}} + \sum_{m',n'} \beta_{m',n'} \boldsymbol{H}_{m',n'}^{\mathrm{diff}} \right] \wedge \boldsymbol{n}$$

$$\boldsymbol{p} = -\left(\sum_{m,n} \alpha_{m,n} \boldsymbol{E}_{m,n}^{\mathrm{inc}} + \sum_{m',n'} \beta_{m',n'} \boldsymbol{E}_{m',n'}^{\mathrm{diff}} \right) \wedge \boldsymbol{n}.$$

avec φ_k une fonction de base définie par (2.2.14) et représentée par la figure 2.2. On remarque que les fonction de base φ_k ont un support local (quelques triangles), alors que les fonctions de base modales ont un support global sur la frontière fictive Γ_g. Les inconnues du problème sont $(\lambda_k, \beta_{m,n})$, puisque les coefficients des modes incidents $\alpha_{m,n}$ font partie des données du problème.

La surface Γ_g étant maillée, elle est associée à des fonctions de base $\varphi_k^{\Gamma_g}$. On introduit alors les matrices de projection des modes incidents et diffractés sur ces fonctions de base :

$$X_{inc} := \begin{pmatrix} \boldsymbol{H}_{m',n'}^{inc} \wedge \boldsymbol{n} \\ -\boldsymbol{E}_{m',n'}^{inc} \wedge \boldsymbol{n} \end{pmatrix} \in \mathbb{C}^{n^{\Gamma_g} \times M_{inc}} \quad X_{diff} := \begin{pmatrix} \boldsymbol{H}_{m',n'}^{diff} \wedge \boldsymbol{n} \\ -\boldsymbol{E}_{m',n'}^{diff} \wedge \boldsymbol{n} \end{pmatrix} \in \mathbb{C}^{n^{\Gamma_g} \times M_{diff}}$$

avec n^{Γ_g} le nombre de degrés de liberté associés aux fonctions de base sur Γ_g. On introduit également les vecteurs de coefficients des modes :

$$\beta := \left(\beta_{m',n'}\right) \in \mathbb{C}^{M_{inc}} \quad \alpha := \left(\alpha_{m',n'}\right) \in \mathbb{C}^{M_{diff}}$$

On a bien :

$$X_{diff}\beta = \begin{pmatrix} \boldsymbol{j}_{diff} \\ \boldsymbol{m}_{diff} \end{pmatrix} \quad X_{inc}\alpha = \begin{pmatrix} \boldsymbol{j}_{inc} \\ \boldsymbol{m}_{inc} \end{pmatrix}$$

On définit les matrices S et R correspondant aux opérateurs $\mathcal{S}$ et $\mathcal{R}$, telles que :

$$\begin{cases} S_{ij} := \int_{\Gamma\times\Gamma} G(|x-y|) \left(\varphi_i(x)\cdot\varphi_j(y) - \frac{1}{\kappa^2}\boldsymbol{\nabla}_\Gamma\cdot\varphi_i(x)\boldsymbol{\nabla}_\Gamma\cdot\varphi_j(y) \right) dxdy \\ R_{ij} := \int_{\Gamma\times\Gamma} \boldsymbol{\nabla}_y G(|x-y|) \wedge \varphi_i(y)\cdot\varphi_j(x) dxdy \end{cases}$$

On introduit de plus les matrices suivantes :

$$B := \begin{pmatrix} -i\kappa Z_0 S \\ -Z_0 R \end{pmatrix} \in \mathbb{C}^{n^{\Gamma_g}\times n}$$

et

$$C := \begin{pmatrix} -i\kappa Z_0 S & -Z_0 R \\ -Z_0 R & i\kappa Z_0 S \end{pmatrix} \in \mathbb{C}^{n^{\Gamma_g}\times n^{\Gamma_g}}$$

Le système linéaire à résoudre est :

$$\begin{bmatrix} -i\kappa Z_0 S & {}^t B X_{\text{diff}} \\ {}^t X_{\text{diff}} B & {}^t X_{\text{diff}} C X_{\text{diff}} \end{bmatrix} \times \begin{bmatrix} \lambda^\Sigma \\ \beta \end{bmatrix} = - \begin{bmatrix} {}^t B X_{\text{inc}} \\ {}^t X_{\text{diff}}(C + I_g) X_{\text{inc}} \end{bmatrix} .\alpha \qquad (6.2.5)$$

Ce système est inversible, car la matrice

$$\begin{bmatrix} S & B^T \\ B & C \end{bmatrix}$$

est inversible, et la base des modes est orthogonale. En effet, cette matrice est celle issue d'une formulation EFIE appliquée à la surface $\Sigma \cup \Gamma_g$.

La matrice de ce système est ici décomposée sous la forme d'une matrice bloc 2×2, dont le terme $-i\kappa Z_0 S$ est similaire au cas sans guide d'onde. On rappelle les tailles des diverses matrices :

Nom	**Dimension**
S	$n^\Sigma \times n^\Sigma$
C	$n^{\Gamma_g} \times n^{\Gamma_g}$
I_g	$n^{\Gamma_g} \times n^{\Gamma_g}$
B	$n^{\Gamma_g} \times n^\Sigma$
X_{inc}	$n^{\Gamma_g} \times M_{inc}$
X_{diff}	$n^{\Gamma_g} \times M_{diff}$
λ^Σ	$n^\Sigma \times n_{rhs}$
α	$M_{inc} \times n_{rhs}$
β	$M_{diff} \times n_{rhs}$

Le bloc supérieur gauche de la matrice d'interaction est de taille $n^\Sigma \times n^\Sigma$, et le bloc inférieur droit de taille $M_{\text{diff}} \times M_{\text{diff}}$. N^Σ est du même ordre de grandeur que pour les

autres calculs par la méthode des éléments finis de frontière, soit $10^6 \to 10^8$, et M_{diff} est de l'ordre de $10^2 \to 10^3$.

6.2.1.1.3 Complément de Schur Le système à résoudre est de la forme :

$$\begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \tag{6.2.6}$$

On pose alors $S := M_{22} - M_{21} M_{11}^{-1} M_{12}$ le complément de Schur, et la résolution du système peut s'écrire sous la forme :

$$\begin{cases} M_{11} = LU \\ Sx_2 = y_2 - M_{21} z_1 \\ M_{11} x_1 = y_1 - M_{12} x_2 \end{cases} \tag{6.2.7}$$

La définition du complément de Schur comprend l'inverse de la matrice M_{11}, mais il n'est pas nécessaire de calculer cette inverse. Il est en effet possible de former le complément de Schur S par les opérations suivantes :

1. Décomposition LU de M_{11}
2. Résolution de $T_1 U = M_{21}$
3. Résolution de $LT_2 = M_{12}$
4. $S \leftarrow M_{22} - T_1 T_2$

En supposant que la matrice M_{11} soit une $\mathcal{H}$-Matrice de grande dimension, et que les autres matrices sont des matrices pleines avec M_{22} de petite taille, les opérations nécessaires pour former S sont :

- Décomposition LU d'une $\mathcal{H}$-Matrice ;
- Résolution d'un système linéaire triangulaire inférieur et supérieur ;
- Opérations sur les matrices denses.

Ces opérations ont été décrites dans le chapitre 3, et sont donc disponibles. De plus, la résolution du premier et du dernier système linéaire de l'équation (6.2.7) a également été décrite dans le chapitre précédent.

Il est donc possible de résoudre efficacement un système de la forme (6.2.6) lorsque la matrice M_{11} est une $\mathcal{H}$-Matrice. La section suivante donne une application en acoustique de cette méthode de résolution.

Remarque 6.1 (Symétrie)**.** *Dans le cas où les matrices M_{11} et M_{22} sont symétriques (formulation EFIE par exemple), pour le guide d'onde de la section précédente, $M_{21} = M_{12}^T$, et la résolution du système est simplifiée en utilisant une décomposition LDL^T de M_{11}.*

6.2.1.1.4 Application L'équation matricielle (6.2.5) a la même forme que l'équation (6.2.6), avec :

$$\begin{cases} M_{11} = -i\kappa Z_0 S & \in \mathbb{C}^{n^\Sigma \times n^\Sigma} \\ M_{12} = B^T X_{\text{diff}} & \in \mathbb{C}^{n^\Sigma \times M_{diff}} \\ M_{21} = X_{\text{diff}}^T B & \in \mathbb{C}^{M_{diff} \times n^\Sigma} \\ M_{22} = X_{\text{diff}}^T C X_{\text{diff}} & \in \mathbb{C}^{M_{diff} \times M_{diff}} \end{cases}$$

La matrice M_{11} est de grande dimension, et peut être approchée par une $\mathcal{H}$-Matrice, de même que les matrices B et C. Ce sont en effet des matrices d'interaction semblables à la matrice donnée par l'expression (2.2.17). La matrice M_{22} a pour dimension le nombre de modes diffractés M_{diff} qui est petit (contrairement à B et C), donc la résolution procède comme décrit précédemment.

Les équations de la section précédente sont données pour l'électromagnétisme par souci de cohérence avec le reste de ce document, et de concision. Néanmoins, il est possible de modéliser un guide d'onde acoustique de la même manière. Dans les applications d'EADS, et d'Airbus en particulier, la surface modale Γ_g peut être de grande dimension. Par exemple, Γ_g peut représenter la section de la nacelle d'un moteur, dont le diamètre intérieur est de 3m pour un A380 (diamètre de la soufflante). Nous donnons ici un exemple de résolution en acoustique.

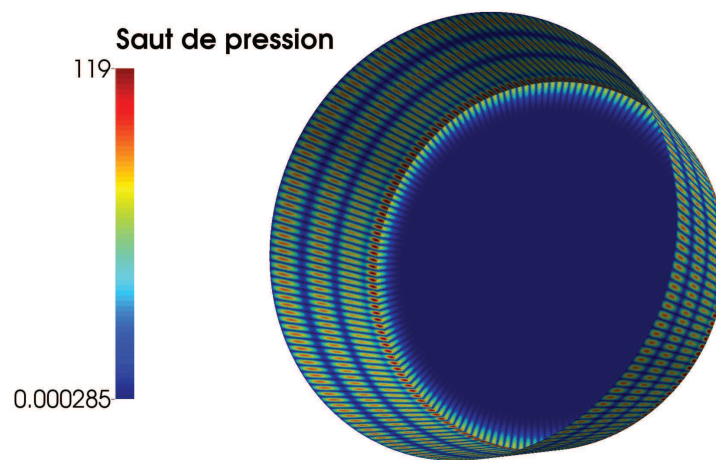


Figure 6.15 – Représentation du saut de pression pour le mode $(79, 1)$.

On considère un cylindre de section circulaire, d'un diamètre de 3m, et d'une longueur de 1.5m, avec un maillage adapté pour la fréquence $f = 3$kHz. Les caractéristiques de ce maillage sont les suivantes :

Quantité	**Valeur**	**Nom**
n^{Σ}	137 226	Degrés de liberté de la surface extérieure
n^{Γ_g}	151 610	Degrés de liberté de la surface modale
M_{inc}	1769	Nombre de modes incidents
M_{diff}	1769	Nombre de modes diffractés

La formulation adoptée par la suite logicielle Actipole mène à des matrices S, B et C non symétriques. La figure 6.15 représente le saut de pression au passage de l'interface sur l'objet pour le mode $(79, 1)$.

Le temps de calcul sur Curie-16 est largement consacré à la lecture et l'écriture des divers fichiers de paramètre et de résultat. Ainsi, avec $\eta = 3$, $\varepsilon = 10^{-4}$ et un calcul en simple précision, le temps total d'assemblage des matrices est de 989s, le temps de factorisation de 113s, et 155s pour les autres opérations (produits matrice-vecteur et résolutions). Le temps de calcul global est de 6254s, le reste du calcul étant consacré aux accès disque.

6.2.2 Décomposition de domaine

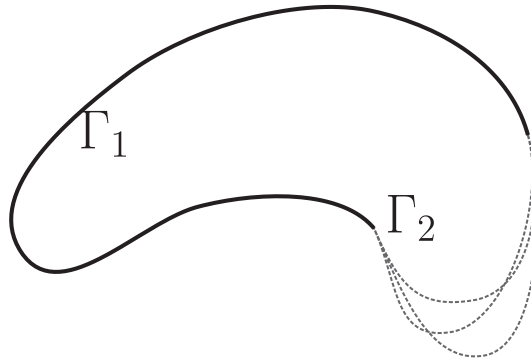


Figure 6.16 – Exemple de partie variable

Dans de nombreuses situations, il est souhaitable d'effectuer plusieurs simulations sur des objets très voisins, différenciés par des variations locales de leur structure :

— Optimisation de la forme d'un sous-système dans une structure de plus grande taille : entrée d'air, nacelle moteur, antenne, *etc.*

— Étude de l'impact de la variabilité des matériaux : variation locale des conditions aux limites ;

— Incertitudes géométriques et/ou sur les conditions aux limites ;

— Impact des différentes configurations : positions des surfaces de contrôles pour un avion, du chargement pour une cavité.

Soit un objet Ω de bord $\Gamma := \Gamma_1 \cup \Gamma_2$. On veut effectuer des calculs de diffraction sur une famille d'objets Γ^i telle que $\Gamma^i = \Gamma_1 \cup \Gamma_2^i$. Cette famille d'objets possède une partie fixe Γ_1, et une partie variable $\Gamma_2 \in \{\Gamma_2^1, \ldots, \Gamma_2^q\}$. Cette situation est illustrée par la figure 6.16. Soit I l'ensemble des indices des degrés de liberté portés par Γ_1, et J^i celle des degrés de liberté portés par Γ_2^i. La matrice d'interaction M^i du problème i s'écrit sous la forme :

$$M^i = \begin{pmatrix} M|_{I\times I} & M|_{I\times J^i} \\ M|_{J^i\times I} & M|_{J^i\times J^i} \end{pmatrix} \tag{6.2.8}$$

Il est possible de résoudre chaque problème classiquement en construisant une $\mathcal{H}$-Matrice approchant M sur l'arbre de groupes $\mathcal{T}_{K^i\times K^i}$, avec $K^i := I \cup J^i$. Il est également possible de construire 4 $\mathcal{H}$-Matrices, construites sur les arbres de groupes $\mathcal{T}_{I\times I}$, $\mathcal{T}_{I\times J^i}$, $\mathcal{T}_{J^i\times I}$ et $\mathcal{T}_{J^i\times J^i}$. En effet, les algorithmes présentés dans le chapitre 3 ne sont pas restreints au cas d'une matrice carrée.

On suppose $|I| \gg |J^i|$, c'est-à-dire que la partie variable est petite devant la partie fixe, en nombre de degrés de liberté. Nous sommes alors dans le cas de la section précédente, et une résolution par complément de Schur est possible, et économique. La construction et factorisation de la $\mathcal{H}$-Matrice approchant $M|_{I\times I}$ n'est faite qu'une fois pour l'ensemble des calculs.

Notons I l'ensemble des indices de la partie fixe, et J celui de la partie mobile. Il est alors possible de construire les 4 sous-matrices sous la forme de $\mathcal{H}$-Matrice basée sur les arbres de groupes $\mathcal{T}_I$ et $\mathcal{T}_J$, et donc de compresser l'ensemble de la matrice. Les opérations

nécessaires à une résolution par complément de Schur sont de plus toutes décrites dans le chapitre 3.

Algorithme 6.1 Algorithme de résolution.

ASSEMBLY($M_{I\times I}$)
LUDECOMPOSITION($M_{I\times I}$)
for all $i \in \{1, \ldots, q\}$ **do**
  ASSEMBLY($M|_{I\times J^i}$) ASSEMBLY($M|_{J^i\times I}$) ASSEMBLY($M|_{J^i\times J^i}$)
  $S \leftarrow$ SCHURCOMPLEMENT(M^i)
  SOLVEMETASYSTEM(M^i, S, b^i)
end for

Les étapes du calcul sont présentées par l'algorithme 6.1, avec SCHURCOMPLEMENT la construction du complément de Schur décrite dans la section précédente, et SOLVEMETASYSTEM la résolution du système (6.2.6) par la méthode (6.2.7).

Remarque 6.2 (Fonctions de base)**.** *En électromagnétisme, les degrés de liberté sont portés par les arêtes. Bien que la frontière $\Gamma_1 \cap \Gamma_2^i$ de la partie variable soit constante dans tous les calculs, les degrés de liberté qui y sont rattachés font partie du bloc variable. En effet, la fonction de base φ_i (équation* (2.2.14) *page 45) associée à ce degré de liberté est modifiée à chaque calcul.*

Remarque 6.3 (Découpage artificiel)**.** *Il est bien entendu possible de découper arbitrairement un objet Γ en deux parties et d'appliquer la méthode décrite ici même pour un unique calcul. Ce découpage revient à imposer un découpage particulier au premier niveau de l'arbre de blocs.*

6.2.3 Application aux statistiques : exemple du Krigeage

Nous présentons dans cette section un domaine d'application très prometteur des algorithmes opérant sur les $\mathcal{H}$-Matrices. En effet, de nombreuses applications en statistiques font apparaître des matrices de corrélation. Celles-ci sont souvent exprimées sous la forme de matrices de convolution d'un noyau asymptotiquement lisse, au sens de la définition 3.2 page 73, ce qui permet de les traiter avec les algorithmes décrits dans le reste de ce document.

Nous prenons comme exemple ici la méthode du krigeage (*kriging* en anglais) [85, 89]. Il s'agit d'une méthode d'interpolation d'une variable présentant une structure de dépendance spatiale par le calcul de l'espérance d'une variable aléatoire. Sous hypothèses, il s'agit du meilleur estimateur linéaire sans biais. Nous présentons ici très brièvement la version la plus simple de cette famille de méthodes, néanmoins le problème de la résolution d'un système linéaire de grande dimension se pose également pour les autres versions. En particulier, le chapitre 8 de [85] est consacré à la discussion de méthodes de résolution approchées plus rapide que $\mathcal{O}(n^3)$. Plus précisément, la section 8.1 aborde la recherche d'une approximation de rang faible de la matrice de covariance, sous forme de $\mathcal{R}k$-Matrice (avec une autre terminologie).

6.2.3.1 Desription du krigeage simple

Soit $Z : \mathbb{R}^d \to \mathbb{R}$ une fonction aléatoire stationnaire d'ordre 2, c'est-à-dire de moyenne constante et de covariance invariante par translation. On dispose de n points d'observation $\{x_i \in \mathbb{R}^d | i = 1, \ldots, n\}$ et des valeurs $Z(x_i)$ associées. On suppose de plus que la covariance pour ces points d'observation est connue, et représentée par une matrice $K \in \mathbb{R}^{n \times n}$, telle que

$$K_{ij} = Cov(Z(x_i), Z(x_j))$$

L'interpolation linéaire $\tilde{Z}(x_0)$ de Z au point x_0 s'écrit comme une combinaison linéaire des valeurs $Z(x_i)$:

$$\tilde{Z}(x_0) = \sum_{i=1}^{n} \alpha_i(x_0) Z(x_i)$$

et se résume donc à la détermination des poids α_i.

La détermination de ces poids est faite par la résolution du système linéaire :

$$K\alpha = K_0 \tag{6.2.9}$$

avec $K_0 \in \mathbb{R}^n$ le vecteur défini par :

$$(K_0)_i := Cov(Z(x_i), Z(x_0))$$

L'évaluation de $\tilde{Z}(x_0)$ nécessite de résoudre un système linéaire de taille $n \times n$ pour chaque point, avec une dépendance en x_0 dans le second membre uniquement. Cette méthode mène donc naturellement à la résolution d'un grand nombre de systèmes linéaires pleins. Par ailleurs, du fait des propriétés de la covariance, la matrice K est symétrique définie positive. Dans ce cas, la méthode de choix pour résoudre les systèmes linéaires du krigeage est d'effectuer une décomposition de Cholesky $K = LL^T$ de la matrice de covariance pour un coût en $\mathcal{O}(n^3)$, et de résoudre les systèmes linéaires avec un coût en $\mathcal{O}(n^2)$ par système à résoudre.

6.2.3.2 Exemples de fonctions de covariance

Fonction de covariance	**Expression**
Gaussienne	$\exp\left(-\frac{r^2}{2l^2}\right)$
Exponentielle	$\exp\left(-\frac{r}{l}\right)$
$\gamma-$exponentielle	$\exp\left(-\left(\frac{r}{l}\right)^\gamma\right)$
Quadratique rationnelle	$\left(1 + \frac{r^2}{2\alpha l^2}\right)^{-\alpha}$

TABLE 6.1 – Fonctions de covariance.

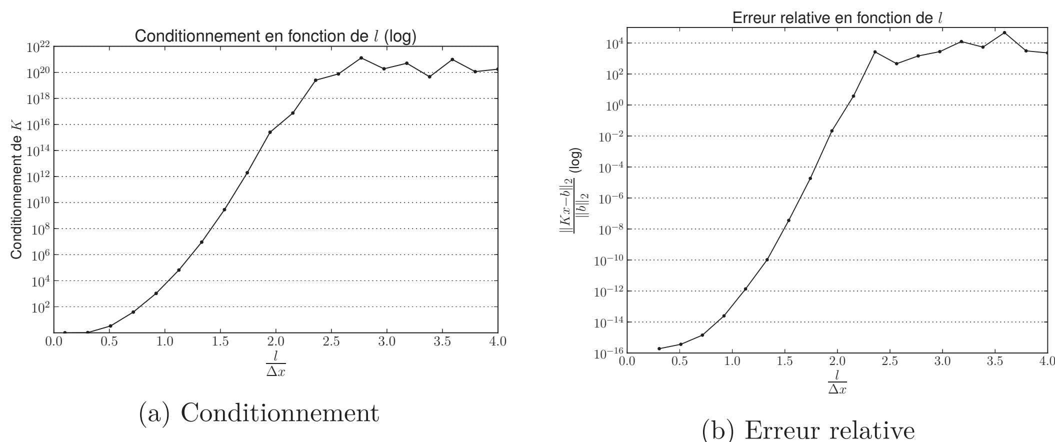
Une fonction quelconque n'est pas en général une fonction de covariance. En effet, la matrice de covariance K (également appelée matrice de Gram) doit être symétrique définie positive, ce qui donne des contraintes d'admissibilité pour les fonctions de covariance.

Dans [85], des exemples de fonctions couramment employées est donné. La table 6.1 reproduit cette liste, en ne conservant que les fonctions stationnaires non dégénérées. On note $r := |x - y|$, et l une longueur de corrélation à déterminer, K_ν est une fonction de Bessel modifiée [10]. Les autres grandeurs inconnues sont des paramètres des familles à déterminer. Dans la suite, nous considérons les fonctions de la table 6.1, à l'exception de la classe des fonctions de Matérn, celle-ci ne présentant néanmoins pas de difficulté particulière.

6.2.3.3 Approximation de matrices de covariances par $\mathcal{H}$-Matrices

Les fonctions de covariance introduites dans la section précédentes sont des noyaux asymptotiquement lisses. Il est donc possible d'approcher les matrices de corrélations sous forme de $\mathcal{H}$-Matrice, à partir de la connaissance de la fonction de corrélation et du nuage de points des x_i. Un avantage des $\mathcal{H}$-Matrices est dans ce cas de donner une indépendance du noyau à la représentation compressée. En effet, les fonctions de la table 6.1 ne sont pas les seules fonctions possibles, et l'existence d'une méthode d'approximation indépendante du noyau est donc un atout important.

Remarque 6.4 (Choix de l)**.** *Il est connu que la valeur de l doit être au plus de l'ordre du pas d'échantillonage [9]. Dans le cas contraire, le conditionnement de la matrice de corrélation grandit de manière très importante, et une résolution précise n'est plus possible, particulièrement pour une fonction de corrélation gaussienne. Dans le cas d'un échantillonnage suivant une grille rectangulaire régulière de pas Δx, pour $l = 2\Delta x$, le conditionnement de la matrice de corrélation est de l'ordre de 10^{17}. La figure 6.17 illustre l'augmentation du conditionnement et de l'erreur pour une grille carrée régulière comprenant 4000 points, pour un noyau gaussien.*



(a) Conditionnement

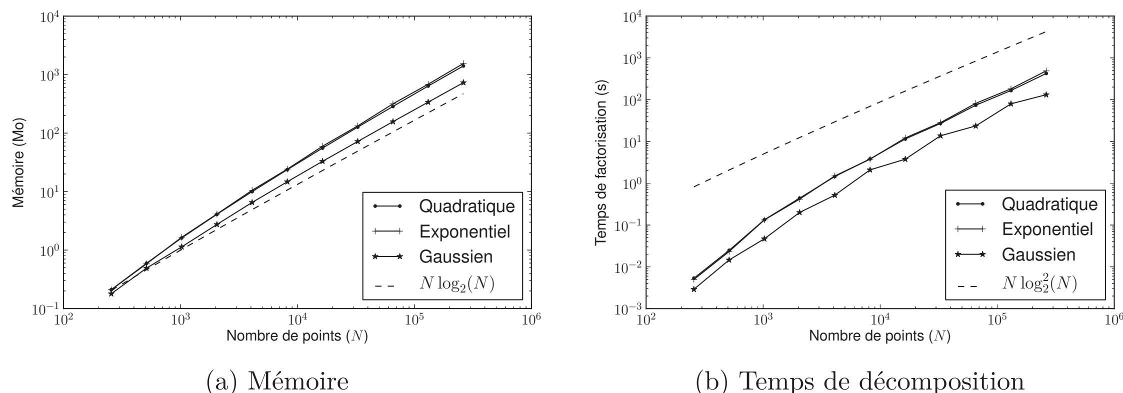
(b) Erreur relative

Figure 6.17 – Croissance du conditionnement et de l'erreur relative pour un noyau gaussien.

Nous considérons ici comme distribution de points une grille régulière carrée en dimension 2. La méthode n'est néanmoins en rien spécifique à une géométrie particulière, comme ceci a été montré dans le reste de ce document. La longueur de corrélation l est

choisie égale au pas de la grille. Le calcul est effectué sur un seul processeur Intel Xeon « Sandy Bridge » à 2.3GHz, avec le même environnement logiciel que la machine Curie-16. Il est néanmoins possible d'utiliser le code parallèle présenté dans les chapitres précédents sans aucune modification. Le choix de la version séquentielle permet d'illustrer le gain à attendre de ces algorithmes sur des machines peu puissantes. Le reste des paramètres de calculs sont fixés comme dans la table 3.2 (ACA+, $\varepsilon = 10^{-4}$, LDL^T), à l'exception de la précision. Les calculs sont ici effectués en double précision réelle, contrairement à simple précision complexe dans le reste de ce document. Un calcul en simple précision serait presque deux fois plus rapide, puisque le temps d'assemblage est ici bien plus faible que pour les éléments finis de frontière.

La figure 6.18 donne l'évolution de la quantité de mémoire requise et du temps de factorisation de la matrice de corrélation, pour divers noyaux. La croissance de ces quantités est conforme à ce qui est attendu pour des noyaux asymptotiquement lisses, et montre tout l'intérêt des $\mathcal{H}$-Matrices pour traiter ces problèmes, avec en particulier une consommation mémoire très faible.



(a) Mémoire

(b) Temps de décomposition

FIGURE 6.18 – Mémoire requise et temps de décomposition pour divers noyaux.

Le comportement de l'erreur est tracé sur la figure 6.19 pour les mêmes noyaux. Dans tous les cas, l'erreur reste maîtrisée pour tous les noyaux. Les variations entre les noyaux sont probablement liées aux différences dans le conditionnement de la matrice pour les divers noyaux. En particulier, le comportement erratique de l'erreur pour le noyau gaussien est à relier aux difficultés de conditionnement que celui-ci pose en général. Par ailleurs, puisque les matrices sont ici symétriques et définies positives, l'utilisation d'une décomposition de Cholesky serait plus indiquée, en terme de performance et de stabilité numérique. Enfin, la figure 6.20 montre la répartition des rangs pour le noyau quadratique rationnel pour $k = 2^{18}$.

Ces résultats montrent le grand intérêt des $\mathcal{H}$-Matrices pour la résolution du système du kriegeage. L'utilisation du solveur $\mathcal{H}$-Matrice permet de factoriser des matrices de grande taille avec une précision maîtrisée. Son efficacité pour le traitement de nombreux seconds membres est importante, puisque le nombre de seconds membres est égal au nombre de points (x_0), souvent grand.

Il faut par ailleurs noter que les matrices de covariance apparaissent dans de nombreux algorithmes en statistiques, et l'application des $\mathcal{H}$-Matrices aux statistiques est donc large.

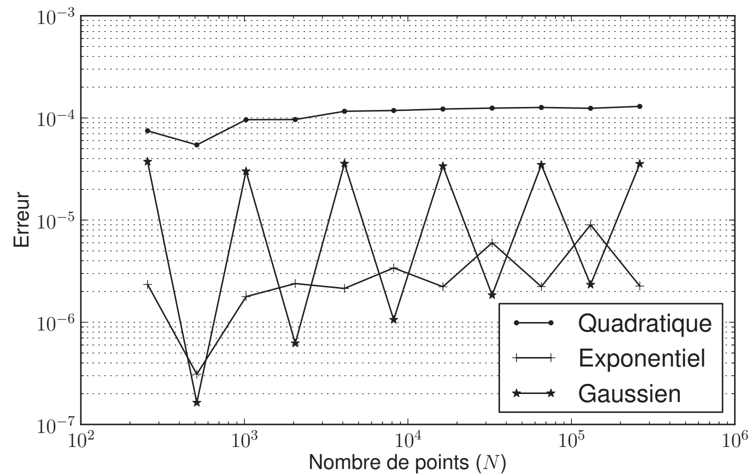


FIGURE 6.19 – Erreur de résolution pour divers noyaux.

6.3 Conclusion

Ce chapitre a donné des exemples d'application des $\mathcal{H}$-Matrices à divers problèmes, liés à la physique des ondes, ou aux statistiques. Pour l'électromagnétisme et l'acoustique, les exemple proviennent d'applications industrielles, et montrent la pertinence des $\mathcal{H}$-Matrices pour une large gamme de configurations. Dans les situations présentées ici, le solveur $\mathcal{H}$-Matrice est comparable, en précision et temps de calcul, à la meilleure méthode disponible industriellement, c'est-à-dire la méthode multipôle rapide dans les situations favorables à ce dernier (convergence en un faible nombre d'itérations). Ce solveur est par ailleurs plus « robuste » qu'un solveur itératif accéléré par FMM. En effet, ce dernier rencontre des difficultés dans les environnements réfléchissants, ou lorsque le nombre de seconds membres devient important. Sa vitesse de convergence est de plus dépendante de l'illumination. Il a été en effet observé empiriquement sur les applications aéronautiques que la convergence tend à être plus lente lorsque la source est localisée (ponctuelle ou concentrée), près de la géométrie. Ces configurations sont cependant fréquentes (pour les antennes, par exemple), et posent de réelles difficultés pratiques.

Le solveur $\mathcal{H}$-Matrice est significativement plus rapide que le solveur FMM dans les configurations problématiques pour ce dernier, et toujours avantageux par rapport à un solveur direct classique. Les extensions du solveur $\mathcal{H}$-Matrices aux guides d'onde et à la décomposition de domaine mettent par ailleurs en évidence un avantage significatif du solveur $\mathcal{H}$-Matrice, sa flexibilité. Celui-ci est en effet adaptable à faible coût à de nouvelles configurations, du fait de la richesse de l'ensemble d'opérations disponibles et de l'indépendance du solveur par rapport à la formulation sous-jacente.

Il reste néanmoins deux aspects non explorés du comportement du solveur $\mathcal{H}$-Matrice :

1. Le comportement (précision, temps de calcul) pour les problèmes ayant un très grand nombre de degrés de liberté ($10^7 - 10^8$) ;
2. L'absence de garantie de qualité de l'approximation par l'algorithme ACA.

Dans les deux cas, une solution attractive à ce problème réside dans le couplage du solveur $\mathcal{H}$-Matrice et du produit multipôle (approximation par FMM du produit matrice-

vecteur Ax). Le comportement de celui-ci a été validé expérimentalement pour des problèmes de très grande taille, et il existe des estimations d'erreur [43] ainsi qu'une décennie d'expérience sur son comportement.

Pour un problème de grande dimension, une factorisation avec un rang k fixe ou une grande tolérance ε fournit un préconditionneur de bonne qualité pour une résolution FMM. Pour la précision, un calcul du résidu par un produit FMM donne une estimation de la qualité du résultat. Dans les deux cas, le solveur $\mathcal{H}$-Matrice est effectivement utilisé en tant que préconditionneur d'une résolution itérative accélérée par FMM. La précision finale du résultat est ainsi au moins égale à celle d'une résolution par le solveur FMM, à un coût réduit. Dans le premier, le solveur $\mathcal{H}$-Matrice sert effectivement de préconditionneur ; dans le second, un seul produit sera nécessaire si la précision demandée est atteinte.

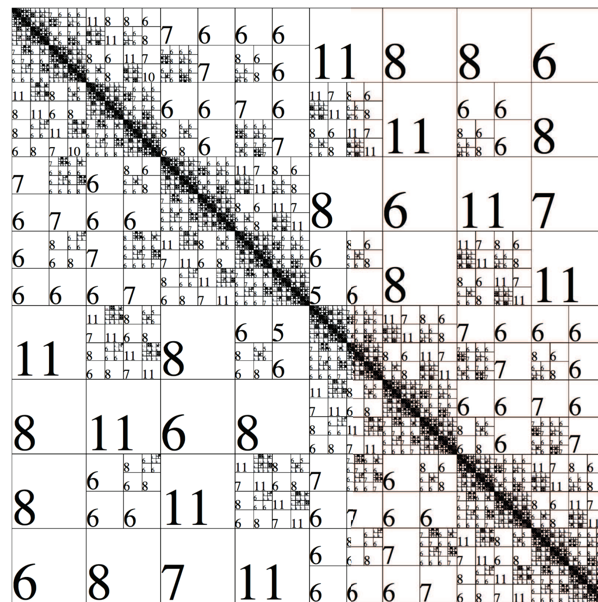


FIGURE 6.20 – $\mathcal{H}$-Matrice pour $N = 2^{18}$ pour le noyau quadratique rationnel.

Conclusion

L'objectif de cette thèse est d'élargir le domaine d'application de la méthode des éléments finis de frontière, en particulier pour les applications à grand nombre de seconds membres, et pénalisant la convergence des solveurs itératifs.

Pour ce faire, nous avons proposé une approche reposant sur la réalisation et la parallélisation efficace d'un solveur direct rapide, basé sur les $\mathcal{H}$-Matrices. Ce solveur a été utilisé avec succès pour les éléments finis de frontière, en particulier pour les équations des ondes. Nous avons effectué une analyse détaillée de la précision et performance de ce solveur sur des géométries canoniques, et démontré sa pertinence pour la résolution de problèmes industriels réalistes. Nous avons effectué une implémentation parallèle efficace de ce solveur, reposant sur un formalisme de graphes de tâches au-dessus d'un moteur d'exécution. Cette implémentation présente un passage à l'échelle quasi-idéal en mémoire partagée, bien meilleur que ce qui est présenté dans la littérature. L'extension à la mémoire distribuée de ces résultats est à notre connaissance nouvelle, et le niveau de performance est satisfaisant, bien que les résultats ne soient que préliminaires. Ces travaux ouvrent de nombreuses perspectives, signalées tout au long de ce document. Elles se décomposent en diverses catégories : perspectives d'analyse numérique, algorithmiques, optimisation du parallèlisme, architectures hybrides, applications.

Du point de vue de l'analyse numérique, de nombreuses questions ont été soulevées par cette thèse. En particulier, l'efficacité meilleure qu'attendue des $\mathcal{H}$-Matrices pour les noyaux oscillants (ne vérifiant pas les hypothèses du théorème connu actuellement de convergence), dont l'analyse mathématique rigoureuse est nécessaire. Une piste de recherche serait de rapprocher le critère d'admissibilité de celui de la FMM directionnelle, afin d'établir certaines configurations dans lesquelles le rang d'approximation ne croît pas avec la taille du bloc. En particulier, l'élaboration d'un critère d'admissibilité pertinent pour les noyaux oscillants permettait d'éliminer les blocs de rang élevé, qui pénalisent fortement le temps de calcul et la consommation mémoire. Un travail mathématique certain est également nécessaire sur la justification, au moins partielle, de l'algorithme ACA, dont l'insolente efficacité n'est à ce jour pas soutenue par un théorème.

D'un point de vue algorithmique, l'implémentation d'autres opérations est une extension aisée et relativement immédiate des ces travaux. Nous pouvons ici citer la décomposition de Cholesky ou la prise en charge des matrices creuses. Le cas des matrices creuses issues d'une dissection emboîtée est particulièrement intéressant, et est une extension di-

recte des travaux présentés ici. Ceci permettra l'élargissement du champ d'application de ce solveur aux matrices issues d'une méthode d'éléments finis volumiques, très largement utilisés dans de nombreuses disciplines, y compris pour les équations des ondes dans ASERIS et ACTIPOLE.

Du point de vue du parallélisme, des travaux portant sur le choix pertinent de la coupe L_0 ou de la fonction $h(d)$ d'affectation des données aux nœuds apporteraient certainement des gains, tout particulièrement pour un grand nombre de processeurs en mémoire partagé, et pour la mémoire distribuée. L'introduction d'un parallélisme à plusieurs niveaux, avec une coupe L_1 est une extension importante pour le passage à l'échelle sur un grand nombre de nœuds du formalisme présenté dans cette thèse.

La gestion des architectures hybrides n'a été qu'esquissée dans ce document, mais représente un sujet important à plusieurs égards. En premier lieu, l'optimisation substantielle de l'assemblage rendue possible par la performance des fonctions trigonométriques sur les cartes graphiques est un atout significatif. Ensuite, l'évolution des machines de calcul mène vers une plus grande hétérogénéité, avec pour motivation l'efficacité énergétique supérieure des accélérateurs, alors que les calculateurs sont de plus en plus fortement contraints par la dissipation thermique. Ces deux facteurs, couplés à la prédominance du temps d'assemblage dans de nombreux calculs (et plus particulièrement en acoustique) invitent à porter une attention particulière à une implémentation sur accélérateur.

Enfin, les configurations présentées dans cette thèse sont loin de représenter l'ensemble des applications possibles des $\mathcal{H}$-Matrices. L'utilisation des $\mathcal{H}$-Matrices pour d'autres équations est facilitée par le caractère « boîte noire » de l'algorithme ACA. Parmi les domaines d'intérêt, les éléments finis de frontière pour l'électrostatique est une cible immédiate, d'autant plus que le noyau est ici asymptotiquement lisse (de la forme $\frac{1}{r}$). L'utilisation des $\mathcal{H}$-Matrices pour les statistiques, brièvement évoquée ici, est un domaine prometteur. En effet, de nombreuses situations mènent à la manipulation de matrices de corrélation, denses et souvent issues d'un noyau asymptotiquement lisse.

Bibliographie

[1] Another software library on hierarchical matrices for elliptic differential equations (ahmed). `http://bebendorf.ins.uni-bonn.de/AHMED.html`.

[2] Calculateur curie. `http://www-hpc.cea.fr/fr/complexe/tgcc-curie.htm`.

[3] Hierarchical matrices. `http://hlib.org`.

[4] $\mathcal{H}$-lib pro. `http://www.hlibpro.com/`.

[5] Plafrim. `https://plafrim.bordeaux.inria.fr/`.

[6] Top 500. `http://www.top500.org/system/177975`. Accessed : 6/3/2013.

[7] Workshop em isae 2012. `http://websites.isae.fr/workshop-em-isae-2012`.

[8] *Équations du transfert radiatif pour l'étude de la diffraction acoustique*. 18e congrès français de mécanique, 2007.

[9] Rachid Ababou, Amvrossios C Bagtzoglou et Eric F Wood : On the condition number of covariance matrices in kriging, estimation, and simulation of random fields. *Mathematical Geology*, 26(1):99–133, 1994.

[10] M. Abramovitz et I.A. Stegun : *Handbook of mathematical functions.* Dover, New York, 1972.

[11] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief et Stanimire Tomov : LU Factorization for Accelerator-based Systems. *In 9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11)*, Sharm El-Sheikh, Égypte, juin 2011.

[12] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault et Stanimire Tomov : QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. *In 25th IEEE International Parallel & Distributed Processing Symposium*, Anchorage, États-Unis, mai 2011.

[13] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Jean Roman, Samuel Thibault et Stanimire Tomov : Dynamically scheduled Cholesky factorization on multicore architectures with GPU accelerators. *In Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Knoxville, États-Unis, juillet 2010.

[14] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner et Toru Takahashi : Pipelining the Fast Multipole Method over a Runtime System. Rapport de recherche RR-7981, INRIA, mai 2012.

[15] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek et Stanimire Tomov : Numerical linear algebra on emerging architectures : The plasma and magma projects. *Journal of Physics : Conference Series*, 180(1):012037, 2009.

[16] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek et Asim YarKhan : Plasma users' guide : Parallel linear algebra software for multicore architectures. Rapport technique, Innovative Computing Laboratory, University of Tennessee, 2011.

[17] G.M. Amdahl : Validity of the single processor approach to achieving large scale computing capabilities. *In Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[18] Cédric Augonnet : *Scheduling Tasks over Multicore machines enhanced with Accelerators : a Runtime System's Perspective.* Thèse de doctorat, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, décembre 2011.

[19] Cédric Augonnet, Samuel Thibault et Raymond Namyst : StarPU : a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Rapport de recherche RR-7240, INRIA, mars 2010.

[20] Nolwenn Balin : Hf rays methods. Rapport technique, EADS IW, 2010.

[21] L. Banjai et W. Hackbusch : Hierarchical matrix techniques for low-and high-frequency helmholtz problems. *IMA journal of numerical analysis*, 28(1):46–79, 2008.

[22] M. Bebendorf : Approximation of boundary element matrices. *Numer. Math.*, 86(4):565–589, 2000.

[23] M. Bebendorf et R. Kriemann : Fast parallel solution of boundary integral equations and related problems. *Comput. Vis. Sci.*, 8(3-4):121–135, décembre 2005.

[24] M. Bebendorf et S. Rjasanow : Adaptive low-rank approximation of collocation matrices. *Computing*, 70:1–24, 2003.

[25] Susan Blackford et Jack J. Dongarra : Installation guide for LAPACK. Rapport technique 41, LAPACK Working Note, juin 1999. originally released March 1992.

[26] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier et Jack Dongarra : Dague : A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1-2):37 – 51, 2012. Extensions for Next-Generation Parallel Programming Models.

[27] A. Le Bot : A vibroacoustic model for high frequency analysis. *Journal of Sound and Vibration*, 211(4):537 – 554, 1998.

[28] A. Le Bot : Derivation of statistical energy analysis from radiative exchanges. *Journal of Sound and Vibration*, 300(3-5):763 – 779, 2007.

[29] A. Le Bot et A. Bocquillet : Comparison of an integral equation on energy and the ray-tracing technique in room acoustics. *The Journal of the Acoustical Society of America*, 108(4):1732–1740, 2000.

[30] Daniel Bouche et Frédéric Molinet : *Méthodes asymptotiques en électromagnétisme.* Springer-Verlag, 1994.

[31] J.J. Bowman, T.B.A. Senior, P.L.E. Uslenghi et J.S. Asvestas : *Electromagnetic and acoustic scattering by simple shapes.* North-Holland Pub. Co., 1970.

[32] Steffen BÖRM et Lars GRASEDYCK : Low-rank approximation of integral operators by interpolation. *Computing*, 72:325–332, 2004.

[33] Steffen BÖRM et Lars GRASEDYCK : Hybrid cross approximation of integral operators. *Numerische Mathematik*, 101:221–249, 2005.

[34] Steffen BÖRM, Lars GRASEDYCK et Wolfgang HACKBUSCH : Introduction to hierarchical matrices with applications. *Engineering Analysis with Boundary Elements*, 27(5):405 – 422, 2003. Large scale problems using BEM.

[35] Steffen BÖRM, Lars GRASEDYCK et Wolfgang HACKBUSCH : Hierarchical matrices. Rapport technique, Max Planck Institut für Mathematik, 2004.

[36] A. CARCATERRA et A. SESTIERI : Energy density equations and power flow in structures. *Journal of Sound and Vibration*, 188(2):269 – 282, 1995.

[37] Henri CASANOVA, Arnaud LEGRAND, Yves ROBERT *et al.* : *Parallel Algorithms*. Chapman & Hall/CRC, 2009.

[38] L. CHAI : *High Performance and Scalable MPI Intra-node Communication Middleware for Multi-core Clusters*. Thèse de doctorat, The Ohio State University, 2009.

[39] Subrahmanyan CHANDRASEKHAR : *Radiative transfer*. Courier Dover Publications, 1960.

[40] Frederica DAREMA : The spmd model : Past, present and future. *In* Yiannis COTRONIS et Jack DONGARRA, éditeurs : *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 de *Lecture Notes in Computer Science*, pages 1–1. Springer Berlin Heidelberg, 2001.

[41] E. DARVE : The fast multipole method : numerical implementation. *Journal of Computational Physics*, 160(1):195–240, 2000.

[42] Eric DARVE : *Méthodes multipôles rapides : Résolution des équations de Maxwell par formulations intégrales*. Thèse de doctorat, 1999.

[43] Eric DARVE : The fast multipole method i : Error analysis and asymptotic complexity. *SIAM Journal on Numerical Analysis*, 38(1):98–128, 2000.

[44] James W. DEMMEL, Nicholas J. HIGHAM et Robert S. SCHREIBER : Block lu factorization. Rapport technique 40, Lapack Working Notes, February 1992.

[45] James W. DEMMEL, Nicholas J. HIGHAM et Robert S. SCHREIBER : Stability of block lu factorization. *Numerical Linear Algebra with Applications*, 2(2):173 – 190, 1995.

[46] J. DONGARRA et Piotr LUSZCZEK : How elegant code evolves with hardware : the case of gaussian elimination. *In* Andy ORAM et Greg WILSON, éditeurs : *Beautiful Code*, chapitre 14. O'Reilly Media, 2007.

[47] Jack J DONGARRA, Jeremy DU CROZ, Sven HAMMARLING et Iain S DUFF : A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[48] Philip DUTRÉ, Kavita BALA et Pilippe BEKAERT : *Advanced Global Illumination*. A K Peters Ltd., 2 édition, 2006.

[49] Katrijn FREDERIX et Marc Van BAREL : Solving a large dense linear system by adaptive cross approximation. *Journal of Computational and Applied Mathematics*, 234(11):3181 – 3195, 2010. Numerical Linear Algebra, Internet and Large Scale Applications.

[50] David GOLDBERG : What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mars 1991.

[51] G.H. GOLUB et C.F. VAN LOAN : *Matrix computations*, volume 3. Johns Hopkins University Press, 1996.

[52] Cindy M. GORAL, Kenneth E. TORRANCE, Donald P. GREENBERG et Bennett BATTAILE : Modeling the interaction of light between diffuse surfaces. *In SIGGRAPH '84 : Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 213–222, New York, NY, USA, 1984. ACM.

[53] S.A. GOREINOV, E.E. TYRTYSHNIKOV et N.L. ZAMARASHKIN : A theory of pseudoskeleton approximations. *Linear Algebra and its Applications*, 261(1-3):1 – 21, 1997.

[54] Ronald L. GRAHAM : Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[55] L. GRASEDYCK : *Theorie und Anwendungen hierarchischer Matrizen*. Thèse de doctorat, UB Kiel, 2001.

[56] L. GRASEDYCK : Adaptive recompression of h-matrices for bem. *Computing*, 74:205–223, 2005.

[57] Lars GRASEDYCK et Wolfgang HACKBUSCH : Construction and arithmetics of h-matrices. *Computing*, 70:295–334, 2003. 10.1007/s00607-003-0019-1.

[58] Lars GRASEDYCK, Ronald KRIEMANN et Sabine LE BORNE : Parallel black box h-lu preconditioning for elliptic boundary value problems. *Computing and Visualization in Science*, 11:273–291, 2008. 10.1007/s00791-008-0098-9.

[59] Leslie GREENGARD et Vladimir ROKHLIN : A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[60] W. GROPP : *MPI : The Complete Reference*. Numéro v. 2 de Scientific and Engineering Computation. MIT Press, 1998.

[61] W. GROPP, E. LUSK et A. SKJELLUM : *Using MPI : portable parallel programming with the message-passing interface*. Scientific and engineering computation. MIT Press, 1999.

[62] W. HACKBUSCH : A sparse matrix arithmetic based on h-matrices. part i : Introduction to h-matrices. *Computing*, 62:89–108, 1999. 10.1007/s006070050015.

[63] W. HACKBUSCH et B. N. KHOROMSKIJ : A sparse h-matrix arithmetic. *Computing*, 64:21–47, 2000. 10.1007/s006070050002.

[64] Nicholas J HIGHAM : *Accuracy and Stability of Numberical Algorithms*. Numéro 48. Siam, 1996.

[65] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2012.

[66] JOSEPH B. KELLER : Geometrical theory of diffraction. *J. Opt. Soc. Am.*, 52(2):116–130, Feb 1962.

[67] Donald E. KNUTH : *The art of computer programming, volume 1 (3rd ed.) : fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[68] N. KORANY, J. BLAUERT et O. Abdel ALIM : Acoustic simulation of rooms with boundaries of partially specular reflectivity. *Applied Acoustics*, 62(7):875 – 887, 2001.

[69] R. Kriemann : Parallel h-matrix arithmetics on shared memory systems. *Computing*, 74:273–297, 2005. 10.1007/s00607-004-0102-2.

[70] Ronald Kriemann : *Parallele Algorithmen für H-Matrizen*. Dissertation, Universität Kiel, 2004.

[71] Christoph Lameter : Numa (non-uniform memory access) : An overview. *Queue*, 11(7):40 :40–40 :51, juillet 2013.

[72] Leslie Lamport : How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[73] Julien Langou : *Résolution de systèmes linéaires de grande taille avec plusieurs seconds membres*. Thèse de doctorat, INSA Toulouse, 2003.

[74] Kai Li et Paul Hudak : Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[75] Rémy Lopez : *Adaptation des méthodes Statistical Energy Analysis (SEA) aux problèmes d'electromagnétisme en cavités*. Thèse de doctorat, Université Paul Sabatier, 2006.

[76] Richard H Lyon : *Statistical energy analysis of dynamical systems : theory and applications*. MIT press Cambridge, MA, 1975.

[77] Matthias Messner : *Fast Boundary Element Methods in Acoustics*. Thèse de doctorat, Université Bordeaux I, 2012.

[78] Matthias Messner, Martin Schanz et Eric Darve : Fast directional multilevel summation for oscillatory kernels based on chebyshev interpolation. *Journal of Computational Physics*, 231(4):1175–1196, 2012.

[79] Jacques Morice : *Méthode Multipôle Rapide pour la résolution de l'équation de la radiosité en transfert radiatif*. Thèse de doctorat, Université Bordeaux I, 2009.

[80] J.C. Nedelec : *Acoustic and Electromagnetic Equations : Integral Representations for Harmonic Problems*. Numéro v. 144 de Acoustic and electromagnetic equations : integral representations for harmonic problems. Springer, 2001.

[81] Youness Noumir : *Une analyse haute fréquence des équations de l'aéroacoustique : étude mathématique et simulations numériques*. Thèse de doctorat, Université Paris 13, 2011.

[82] Nvidia : NVIDIA's Next Generation CUDA Compute Architecture : Kepler TM GK110, Whitepaper. Rapport technique, 2012.

[83] nVIDIA Corporation : Cublas. `https://developer.nvidia.com/cuBLAS`.

[84] Naji Qatanani et Monika Schulz : Analytical and numerical investigation of the fredholm integral equation for the heat radiation problem. *Applied Mathematics and Computation*, 175(1):149 – 170, 2006.

[85] C.E. Rasmussen et C.K.I. Williams : *Gaussian Processes for Machine Learning*. Adaptative computation and machine learning series. University Press Group Limited, 2006.

[86] P.A. Raviart et J.M. Thomas : A mixed finite element method for 2-nd order elliptic problems. *In* Ilio Galligani et Enrico Magenes, éditeurs : *Mathematical Aspects of Finite Element Methods*, volume 606 de *Lecture Notes in Mathematics*, pages 292–315. Springer Berlin Heidelberg, 1977.

[87] Ran Raz : On the complexity of matrix product. *In Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 144–151, New York, NY, USA, 2002. ACM.

[88] Gilles Rougeron, François Gaudaire, Yannick Gabillet et Kadi Bouatouch : Simulation of the indoor propagation of a 60ghz electromagnetic wave with a time-dependent radiosity algorithm. *Computers & Graphics*, 26(1):125 – 141, 2002.

[89] T.J. Santner, B.J. Williams et W. Notz : *The Design and Analysis of Computer Experiments*. Springer Series in Statistics. Springer, 2003.

[90] J. Shaeffer : Direct solve of electrically large integral equations for problem sizes to 1 m unknowns. *Antennas and Propagation, IEEE Transactions on*, 56(8):2306–2313, 2008.

[91] Peter F. M. Smulders : Statistical characterization of 60-ghz indoor radio channel. *IEEE Transactions on Antennas and Propagation*, 57(10):2820 – 2829, October 2009.

[92] Quentin H Spencer, Brian D Jeffs, Michael A Jensen et A Lee Swindlehurst : Modeling the statistical time and angle of arrival characteristics of an indoor multipath channel. *Selected Areas in Communications, IEEE Journal on*, 18(3):347–360, 2000.

[93] Herb Sutter et James Larus : Software and the concurrency revolution. *Queue*, 3(7):54–62, septembre 2005.

[94] Guillaume Sylvand : *La Méthode Multipôle Rapide en Électromagnétisme. Performances, Parallélisation, Applications*. Thèse de doctorat, ENPC / CERMICS, 2002.

[95] I. Terrasse et T. Abboud : *Modélisation des phénomènes de propagation d'ondes*. Ecole Polytechnique, 2007.

[96] H. Topcuoglu, S. Hariri et Min-You Wu : Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[97] L.N. Trefethen et D. Bau : *Numerical linear algebra*. Miscellaneous Bks. Society for Industrial and Applied Mathematics, 1997.

[98] Leslie G. Valiant : A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, août 1990.

[99] Kurzak J. Dongarra J. YarKhan, A. : Quark users' guide : Queueing and runtime for kernels. Rapport technique, Innovative Computing Laboratory, University of Tennessee.

[100] K. Zhao, M.N. Vouvakis et J.F. Lee : The adaptive cross approximation algorithm for accelerated method of moments computations of emc problems. *Electromagnetic Compatibility, IEEE Transactions on*, 47(4):763–773, 2005.

Résumé

La méthode des éléments finis de frontière (BEM) requiert la résolution de systèmes linéaires pleins, mal conditionnés de grande dimension avec un nombre important de seconds membres ; la difficulté de résolution est le principal obstacle pratique à son utilisation. Une matrice hiérarchique ($\mathcal{H}$-Matrice) est un format de stockage hiérarchique, creux et approché de matrices dont la manipulation permet la réalisation d'un solveur linéaire direct avec une complexité asymptotique en $\mathcal{O}(N \log_2^\alpha(N))$ en espace et en temps.
On s'intéresse aux $\mathcal{H}$-Matrices pour les BEM, avec l'évaluation et la mise en œuvre des $\mathcal{H}$-Matrices sur des cas académiques et industriels complexes d'une part ; et la parallélisation efficace des ces algorithmes d'autre part. Une étude paramétrique rigoureuse et détaillée sur des cas modèles et industriels est présentée. On précise le domaine d'application des $\mathcal{H}$-Matrices pour les BEM, donne des recommendations sur les paramètres des algorithmes, et décrit des optimisations réalisables. Nous montrons la pertinence des $\mathcal{H}$-Matrices en termes de précision et temps de calcul, en comparaison avec un solveur direct classique et avec un solveur itératif basé sur la méthode multipôle rapide.
La parallélisation des algorithmes, en mémoire partagée puis distribuée, repose sur l'expression des calculs sous forme de graphes de tâches composables, dont l'ordonnancement est effectué dynamiquement à l'aide d'un moteur d'exécution. Nous donnons une expression permettant de contrôler la granularité des opérations au travers d'une coupe paramétrable des $\mathcal{H}$-Matrices, et exposons diverses optimisations et extensions de ce formalisme. Une efficacité parallèle quasi-optimale est obtenue en mémoire partagée, ainsi que des résultats prometteurs en mémoire distribuée.

Mots-clefs BEM, Solveur Direct Rapide, H-Matrice, Parallélisme, Électromagnétisme, Acoustique

---

Fast Direct Solver for the Boundary Element Method in Electromagnetism and Acoustics: $\mathcal{H}$-Matrices. Parallelism and Industrial Applications.

Abstract

The Boundary Element Method (BEM) requires the solving of large, ill-conditionned, dense linear systems with a large number of righ-hand sides ; the solving difficulty is the main practical obstacle to its use. A hierarchical matrix ($\mathcal{H}-$Matrix) is a hierachical, approximate, data-sparse storage format for matrices that can be manipulated to produce a direct linear solver with an asymptotic space and time complexity of $\mathcal{O}(N \log_2^\alpha(N))$.
We consider the $\mathcal{H}-$Matrices for the BEM, with the assessment and application of $\mathcal{H}-$Matrices to complex academic and industrial test cases on the one hand ; and the efficient parallelization of these algorithms on the other hand. A rigorous and detailed parametric study on model and industrial test cases is shown. We define the working range of $\mathcal{H}$-Matrices for the BEM, provide recommendations for the algorithm parameters, and describe some practical optimizations. We show the relevance of the $\mathcal{H}$-Matrices in terms of precision and computation time, in comparison with a classical direct solver and with an iterative solver based on the Fast Multipole Method.
The parallelization of the algorithms, in shared and distributed memory, relies on the expression of the computations with composable task graphs, dynamically scheduled with the help of a runtime system. We give an expression controlling the operations granularity through a configurable cut of the $\mathcal{H}$-Matrices, and present various optimizations and extensions of this expression. An almost optimal parallel efficiency is achieved in shared memory, along with promising results in distributed memory.

Keywords BEM, Fast Direct Solver, H-Matrix, Parallelism, Electromagnetism, Acoustics