Optimisation de la précision numérique des codes parallèles

Farah Benmouhoub

▶ **To cite this version:**

Farah Benmouhoub. Optimisation de la précision numérique des codes parallèles. Arithmétique des ordinateurs. Université de Perpignan, 2022. Français. NNT : 2022PERP0009 . tel-03783734

HAL Id: tel-03783734

https://theses.hal.science/tel-03783734v1

Submitted on 22 Sep 2022

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.

THÈSE
Pour obtenir le grade de Docteur

Délivrée par
UNIVERSITE DE PERPIGNAN VIA DOMITIA

Préparée au sein de l'école doctorale **ED305**
Et de l'unité de recherche **LAMPS**

Spécialité : **Informatique**

Présentée par

Farah BENMOUHOUB

TITRE DE LA THESE

Optimisation de la précision numérique des codes parallèles

Soutenue le 17 Juin 2022 devant le jury composé de

Mme. **Fabienne Jézéquel,** Maître de conférences, Université Paris 2	Rapportrice
Mme. **Yassamine Seladji,** Maître de conférences, Université de Tlemcen	Rapportrice
Mr. **Philippe Langlois,** Professeur, Université de Perpignan	Examinateur
Mr. **Nicolas Louvet,** Maître de conférences, Université Lyon 1	Examinateur
Mr. **Matthieu MARTEL,** Professeur, Université de Perpignan	Directeur
Mr. **Pierre-Loic Garoche,** Professeur, École nationale de l'aviation civile	Co-Directeur

Remerciements

Je tiens tout d'abord à remercier mes directeurs de thèse Matthieu Martel et Pierre-Loic Garoche pour leurs encadrements et leurs nombreux encouragements. Contente et honorée de travailler sous votre direction, en trouvant auprès de vous les meilleurs conseils pour bien munir ce travail de thèse.

Je remercie également les membres du jury pour l'intérêt qu'ils ont porté à ces travaux de recherche en acceptant de les examiner et de les enrichir par leurs remarques. Merci à mes rapportrices pour le temps consacré aux multiples relectures de ce manuscrit.

Je remercie mon comité de suivi de thèse Alexandre Chapoutout, Xavier Thirioux et Mikael Barboteu pour leurs conseils et propositions qui ont contribué à l'amélioration de ces travaux.

Je tiens à remercier le directeur du laboratoire LAMPS Robert Brouzet ainsi que tous les membres, en particulier, un grand merci à toi Sylvia pour le travail que tu fais et pour la main que tu nous tends à chaque fois qu'on te sollicite. Je ne t'oublierai jamais.

Merci à toi Nasrine, de mon encadrante de stage à collègue à amie à soeur à bien plus que ça. Je te remercierai jamais assez pour tes encouragements, tes séances de psychologie ;) et ton soutien dans mes moments les plus difficiles. Sahit à wletma.

Merci à ma meilleure amie pour tous les moments agréables passés ensemble. Je te souhaite beaucoup d'excellence dans ta carrière et saches que je serai toujours là pour toi.

À la source de mes efforts, mon soutien moral, ma vie et mon bonheur. Maman et Papa, malgré la distance qui nous sépare vous étiez et vous êtes toujours présents à mes cotés. Sans vous, je ne serai pas là où j'en suis aujourd'hui. Je vous aime.

À toi Abdeslam, merci d'être là, merci pour tes encouragements, ton soutien, ta patience et pour tous ces très bons moments passés ensemble. Je te serai éternellement reconnaissante.

À mes deux frères, Amine & Hassen, ma vie sans vous n'aurait aucun sens, j'implore dieu qu'il vous apporte bonheur et vous aide à réaliser tous vos Vœux.

À mes deux neveux, mes petits loulous Anaëlle et Aksil qui ont changé le courant de nos vies depuis leurs arrivées. Amatou Farah vous aime plus que tout.

Table des matières

I État de l'art **7**

1 Arithmétique flottante IEEE−754 **9**

1.1 Introduction . . . 9
1.2 Le standard IEEE−754 . . . 10
1.3 Les nombres . . . 10
1.4 Les formats . . . 11
1.5 Les modes d'arrondi . . . 12
1.6 Les fonctions ufp et ulp . . . 12
1.7 Sources d'erreurs . . . 13
1.7.1 Les erreurs d'arrondi . . . 13
1.7.2 Absorption . . . 13
1.7.3 Élimination catastrophique . . . 14
1.8 Estimation d'erreurs en précision finie . . . 14
1.9 Conclusion . . . 14

2 Algorithmes de sommation existants **15**

2.1 Introduction . . . 15
2.2 Somme récursive . . . 16
2.3 Somme précise . . . 16
2.3.1 Les transformations sans erreur . . . 17
2.3.2 Somme compensée . . . 18
2.3.3 Somme en cascade . . . 19
2.3.4 Algorithmes de Demmel et Hida . . . 20
2.4 Somme reproductible . . . 21
2.5 Conclusion . . . 22

II Vers l'amélioration de la précision numérique de la somme **23**

3 Application de transformation de programme **25**

3.1 Introduction . . . 25
3.2 Transformation de programmes . . . 26
3.2.1 Transformation des expressions arithmétiques . . . 26
3.2.2 Outil de transformation automatique de programmes . . . 27

3.3 Résultats expérimentaux . . . 30
3.3.1 EDP de la propagation de chaleur . . . 30
3.3.2 Précision numérique . . . 32
3.3.3 Temps d'exécution . . . 35
3.4 Conclusion . . . 37

4 Technique de placement de tâches **38**
4.1 Introduction . . . 38
4.2 Résolution de systèmes linéaires . . . 39
4.3 Notions utilisées . . . 40
4.4 Domaines abstraits . . . 41
4.4.1 Exp^n : Ordre de grandeur des éléments de la matrice . . . 42
4.4.2 $\mathsf{Exp}^{\#n}$: Abstraction des exposants en scalaires . . . 44
4.4.3 Abstraction en gradients . . . 45
4.5 Partitionnement des données . . . 48
4.5.1 Algorithme glouton . . . 48
4.5.2 Exemple . . . 50
4.6 Expérimentations . . . 52
4.6.1 Poutre en flexion . . . 52
4.6.2 Résultats expérimentaux . . . 53
4.7 Conclusion . . . 55

5 Algorithmes de somme précis, reproductibles **57**
5.1 Introduction . . . 57
5.2 Somme précise en séquentiel . . . 58
5.3 Somme précise en parallèle . . . 60
5.3.1 Sommes partielles sur chaque processeur . . . 60
5.3.2 Calcul d'une seule somme globale . . . 61
5.4 Expériences numériques . . . 62
5.4.1 Résultats expérimentaux liés à la précision numérique . . . 63
5.4.2 Résultats expérimentaux liés à la reproductibilité . . . 70
5.4.3 Notre algorithme séquentiel [8] versus les deux algorithmes de Demmel et Hida [30] . . . 73
5.5 Conclusion . . . 77

6 Performances des algorithmes proposés **78**
6.1 Introduction . . . 78
6.2 Précision numérique . . . 79
6.2.1 Méthode de Simpson . . . 79
6.2.2 Méthode de Factorisation LU . . . 82

6.3 Convergence des méthodes itératives 84
6.3.1 Méthode de Jacobi 84
6.3.2 Méthode de la puissance itérée 86
6.4 Reproductibilité 86
6.4.1 Méthode de Simpson 87
6.4.2 Multiplication de matrices 88
6.5 Conclusion 89

7 Application aux réseaux de neurones **90**
7.1 Introduction 90
7.2 État de l'art 91
7.3 Réseaux de neurones 92
7.3.1 Modèle mathématique 92
7.3.2 Types des réseaux de neurones 94
7.4 L'apprentissage des réseaux de neurones 95
7.4.1 L'apprentissage supervisé 95
7.4.2 L'apprentissage non supervisé 96
7.4.3 Contribution 97
7.5 Exemple 97
7.5.1 Mécanisme de l'algorithme 97
7.5.2 Résultats préliminaires 98
7.6 Conclusion 99

8 CONCLUSION ET PERSPECTIVES **101**
8.1 Conclusion 101
8.2 Perspectives 102

Bibliographie **104**

Liste des figures

1 Calcul de somme en cascade. 20

2 APEG correspondant à l'expression $((a + a) + b) \times c$ (figure 1 dans [24]). . 27
3 Schéma récapitulatif des entrées et sorties de `Salsa`. 28
4 Programme original. 29
5 Premier programme intermédiaire. 29
6 Deuxième programme intermédiaire. 30
7 Programme final avant transformation des expressions. 30
8 Programme Final après Transformation des Expressions. 31
9 Aperçu d'une grille 800×800 partitionnée suivant le nombre de processeurs dans ce cas 4 processeurs (à gauche). Exemple de points voisins nécessaires pour le calcul de la valeur de la chaleur en un point courant (à droite). . . . 31
10 Aperçu de la grille chauffée en deux coins, nombre d'itérations : 3600, stencil : 1 (figure gauche), stencil : 2 (figure droite). 33
11 Processus de spécialisation des calculs en fonction des données d'entrée. . . 34
12 Erreurs absolues entre le programme original et celui optimisé, température initiale : 10106.6 ˚ F, nombre d'itérations : 1800 (en haut à gauche), 3600 (en haut à droite), 7200 (en bas à gauche), 10800 (en bas à droite). 35
13 Les erreurs absolues et relatives respectivement entre le programme original et celui transformé, température initiale : 1547.6 ˚ F (en haut à gauche, en bas à gauche), pour des nombres d'itérations différents, température initiale : 10106.6 ˚ F (en haut à droite, en bas à droite). 36

14 Schéma général des abstractions. 42
15 Treillis des gradients. 45
16 Algorithme glouton pour le partitionnement d'un vecteur de taille n en gradients. 49
17 Fonction de comparaison entre x et y. 50
18 Les différentes configurations de parallélisation possibles. 50
19 Représentation d'une poutre $1D$ en flexion. 52
20 Système linéaire à matrice tridiagonale. 53
21 le titre 54

22 Schéma du calcul de la somme. 62

23 Erreurs absolues entre les résultats de sommation du Dataset1 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision. 63
24 Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset1. 64
25 Erreurs absolues entre les résultats de sommation du Dataset3 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision. 64
26 Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset3. 65
27 Erreurs absolues entre les résultats de sommation du Dataset4 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision. 66
28 Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset4. 66
29 Erreurs absolues entre les résultats de sommation du Dataset5 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision. 67
30 Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset5. 67
31 Erreurs absolues entre les résultats de sommation du Dataset7 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision. 68
32 Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset7. 68

33 Erreurs absolues entre les résultats de sommation du Dataset8 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision. 69
34 Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset8. 69
35 Reproductibilité des résultats de somme de l'algorithme 10 et de l'algorithme de somme récursive en simple précision pour le Dataset3 en fonction du nombre de processeurs. 71
36 Reproductibilité des résultats de somme de l'algorithme 10 et de l'algorithme de somme récursive en simple précision pour le Dataset4 en fonction du nombre de processeurs. 72
37 Reproductibilité des résultats de somme de l'algorithme 11 et de l'algorithme de somme récursive en simple précision pour le Dataset1 en fonction du nombre de processeurs. 72
38 Reproductibilité des résultats de sommation de l'algorithme 11 et de l'algorithme de somme récursive en simple précision pour le Dataset2 en fonction du nombre de processeurs. 73
39 Erreurs absolues entre les résultats de sommation du Dataset1 à l'aide de trois algorithmes : l'algorithme 9, l'algorithme 7 et l'algorithme de somme récursive en simple précision avec les résultats de sommation du Dataset1 obtenus par l'algorithme de somme récursive en double précision. 74
40 Zoom sur les erreurs absolues de notre algorithme séquentiel et de l'algorithme de Demmel et Hida. 74
41 Temps d'exécution de l'algorithme de somme récursive, l'algorithme 9 et l'algorithme 7. 75
42 Erreurs absolues entre les résultats de sommations du Dataset1 à l'aide de l'algorithme 9 et l'algorithme 8. 76
43 Temps d'exécution de l'algorithme 9 et de l'algorithme 8. 76

44 Les erreurs absolues entre le programme original et le programme précis pour le calcul intégral de trois fonctions différentes ($C \times \cos(x)$, $C \times (1/x^2 + 1)$ et $C \times \tanh(x)$) avec le résultat analytique correspondant en faisant varier la borne supérieure de l'intégrale $b = 2, 3, 4, 5$. 80
45 Temps d'exécution du programme original, programme précis et le programme de sommation par tri pour le calcul intégral de la fonction $C \times \tanh(x)$. 81

46 Les erreurs absolues de la factorisation LU par l'algorithme original et précis pour des matrices de différentes tailles. 82
47 Temps d'exécution du programme original, programme précis et le programme de sommation par tri de la factorisation LU. 83
48 Différence entre le nombre d'itérations nécessaires pour le programme original et le programme précis pour atteindre la convergence de la méthode de Jacobi. 85
49 Différence entre le nombre d'itérations de la méthode de puissance itérée originale et précise ($n = 100$, $d \in [300, 500]$ avec incrément de 20). 87
50 La reproductibilité des calculs d'intégrales utilisant la méthode de Simpson du programme original et de celui précis en fonction du nombre de processeurs. 88
51 La reproductibilité de la multiplication matrice-matrice pour différentes tailles de matrices. 89

52 Réseau de neurones artificiels. 93
53 modèle d'un neurone. 93
54 Aperçu des images de la base de données *fashion_mnist*. 96
55 Erreurs entre les sorties propagées en avant et les sorties attendues. 98
56 Pourcentage de reconnaissance en fonction du nombre de cycles. 99

57 Schéma du calcul de la somme avec analyse statique. 103

Liste des tableaux

1.1 *Formats de représentation de la norme IEEE–754* 11

3.1 Estimations du nombre d'itérations et du temps d'exécution pour le programme de la propagation de chaleur. 37

5.1 Signe et proportion des grandes valeurs parmi les petites et moyennes valeurs pour chaque ensemble de données [Thévenoux et al. [58]]. 63

5.2 Comparaison des temps d'exécution de trois algorithmes : algorithme de somme récursive, algorithme 9 et l'algorithme 7. 75

6.1 Nombre d'itérations des deux programmes original et précis de Jacobi. . . 85

7.1 Fonctions de transfert $R = f(S)$. 94

INTRODUCTION

Motivation

L'informatique a pris une place prépondérante dans notre vie quotidienne ainsi que dans divers domaines d'application dits critiques, comme l'aéronautique, automobile, l'industrie spatiale, les équipements médicaux, le nucléaire, etc. Nous avons tendance à croire aux calculs effectués par ordinateur, or un problème lié à la fiabilité se pose dû à l'arithmétique flottante utilisée par les machines. Notamment, les calculs massifs que nous appelons simulations dans le cas du calcul haute performance, ces calculs sont basés sur l'arithmétique flottante qui fait qu'une valeur ne peut être représentée exactement en mémoire. Plus précisément, toute valeur doit être arrondie et, par conséquent, de nombreuses erreurs de calcul peuvent être générées. En général, ces erreurs sont négligeables, car elles sont petites. Par ailleurs, dans un scénario critique, ces erreurs peuvent s'accumuler et se propager provoquant ainsi des dégâts considérables sur le plan industriel, financier, humain. Citons par exemple `Ariane 5` et le missile `Patriot` comme fameux exemples de désastres causés par les erreurs de calcul :

- Le crash `d'Ariane 5` qui est dû à une erreur de conversion d'un nombre flottant représenté en 64 bits vers un entier de 16 bits.
- Le problème de l'antimissile `Patriot`, dû a une erreur d'arrondi de l'ordre de 10^7.

À côté des erreurs d'arrondi dues aux calculs basés sur l'arithmétique flottante, d'autres types d'erreurs d'origine humaine peuvent survenir. Par exemple, dans un contexte physique, suite à des erreurs de prélèvements de mesures ou encore à cause du choix du modèle. C'est-à-dire, lors de la représentation d'un modèle physique par un modèle mathématique pas nécessairement approprié.

La précision des calculs en nombres flottants dépend de nombreux facteurs : types d'opérations effectuées, valeurs et types de données employés, ordinateurs utilisés. Par ailleurs, la précision des calculs dépend fortement de l'ordre dans lequel sont effectuées les opérations et le parallélisme modifie ce dernier, voire rend cet ordre aléatoire. Les résultats obtenus peuvent alors être non reproductibles (plusieurs simulations identiques donnant des résultats différents). Dans le cadre de cette thèse, nous nous intéressons aux erreurs d'arrondi et à leurs impact sur la précision numérique et la reproductibilité des calculs reposant sur l'arithmétique à virgule flottante spécifiée par la norme IEEE754 [2, 63]. La problématique à laquelle nous voulons répondre est alors la suivante : comment améliorer la précision numérique des codes parallèles tout en étant reproductible ?

Pour répondre à cette problématique nous nous somme concentrés particulièrement sur une brique élémentaire du calcul, largement utilisée, qui est la somme d'une suite de nombres. Cette somme est souvent sujet à de nombreuses imprécisions causées par les erreurs d'arrondi provoquées par la non-associativité de cet opérateur ou encore par les absorptions ou éliminations qui peuvent survenir lors du calcul.

Contributions

Les travaux réalisés durant cette thèse ont abouti aux contributions détaillées ci-dessous dans le cadre de l'amélioration de la précision numérique et de la reproductibilité des calculs flottants.

1. **Étendre une méthode de transformation automatique de programmes pour la précision numérique**

   Plusieurs outils ont été développés afin d'améliorer la précision numérique des calculs. Certains prennent uniquement en considération les expressions arithmétiques, comme les deux outils `Herbie` et `Sardana`. La différence entre ces deux outils réside dans la façon dont le choix de la meilleure expression parmi plusieurs expressions proposées est fait. D'autre part, nous trouvons des outils qui améliorent la précision numérique d'un code entier par des transformations automatiques source à source comme l'outil `Salsa`. Dans ce même contexte, nous allons étendre une technique de transformations de programmes du cas séquentiel au cas parallèle, par le biais de l'outil `Salsa`, pour améliorer automatiquement la précision numérique des calculs parallèles. Au cours de ce travail, en plus des transformations automatiques de codes, nous nous sommes intéressés à la spécialisation du code de chaque processeur. Plus précisément, avant d'appliquer les transformations automatiques à l'aide de l'outil `Salsa`, nous spécialisons le code de chaque processeur en fonction de ses données et nous obtenons en sortie un code mathématiquement équivalent mais plus précis.

   Nous mesurons les performances de notre approche en calculant à la fois les erreurs absolues et relatives entre le programme original et celui transformé par `Salsa`. Nous allons également étudier l'impact de l'optimisation de la précision numérique des résultats sur la convergence et le temps d'exécution.

2. **Proposer une technique de placement de tâches**

   Dans cette partie nous proposons une nouvelle technique basée sur l'analyse statique par interprétation abstraite. Cette technique est considérée comme une phase

préliminaire qui est appliquée aux données avant d'entamer le calcul. Nous voulons à travers cette méthode déterminer le meilleur partitionnement des données qui pourrait favoriser l'optimisation de la précision numérique. Pour mieux illustrer ceci, prenons un cas pratique de la multiplication de matrices. Notre objectif est de détecter l'ordre de grandeur de ses coefficients qui nous permettrons d'ordonner le calcul. Plus en détails, la technique proposée permet de représenter les propriétés appelées gradients des différentes séquences de valeurs $s = x_1.x_2.....x_n$. Les informations provenant des gradients calculés indiquent l'ordre dans lequel les éléments de s sont représentés et à partir de ces informations nous créons des partitions d'un même signe et ordre de grandeur. Une fois l'ordre calculé, nous associons un algorithme de somme approprié à chaque partition créé.

Nous avons testé cette technique sur un exemple issu de la mécanique. Les résultats obtenus ont montré l'efficacité de notre méthode pour identifier les blocs de même ordre de grandeur. L'étape d'après consiste à déterminer comment exploiter ces informations pour optimiser la précision numérique. La réponse à cette question est détaillée dans la contribution suivante.

3. Proposer des algorithmes de somme précis, reproductibles

Nous proposons un nouveau algorithme séquentiel pour sommer n nombres flottants avec précision. Les avantages de cet algorithme sont multiples. Tout d'abord, l'algorithme proposé n'augmente pas la complexité linéaire par rapport à l'algorithme de somme récursive. Ajoutons à cela le fait que les sommes sont calculées en précision de travail sans nécessiter des accumulateurs en haute précision. Le principe de cet algorithme est comme suit. Soient exp_{min} et exp_{max} les plus petits et plus grands exposants en précision de travail. Les sommes sont calculées en fonction des exposants des valeurs à sommer. L'algorithme nécessite un tableau ayant $exp_{max} - exp_{min} + 1$ éléments, dans ce tableau chaque valeur est additionnée selon son exposant et à la fin nous parcourons le tableau pour en accumuler les éléments et renvoyer le résultat final.

Deux différents schémas de parallélisation sont appliqués à cet algorithme, ce qui donne deux algorithmes parallèles. Le premier algorithme parallèle effectue une somme locale au niveau de chaque processeur, i.e., après avoir additionné les valeurs selon leurs exposants, une somme locale est calculée au niveau de chaque processeur. Par la suite, ces sommes sont additionnées par le processeur maître afin de renvoyer le résultat final. Quant au deuxième algorithme, au lieu de calculer des sommes locales au niveau de chaque processeur, le résultat final est calculé à l'aide d'une opération de réduction. De ce fait, le deuxième algorithme est le plus efficace en terme de précision et plus coûteux en terme de complexité tout en étant linéaire.

Nos algorithmes ont été testés sur des ensembles générés aléatoirement avec différentes proportions de grandes valeurs parmi des moyennes et des petites selon le format simple précision. Ce choix dans la variété des valeurs est fait pour introduire différentes absorptions et éliminations. De plus des résultats liés à la précision numérique cités précédemment, nos algorithmes parallèles ont la particularité d'améliorer la reproductibilité des résultats de sommations comparés aux résultats calculés par un algorithme de somme récursive parallèle.

Nous établissons également une comparaison en précision numérique et en temps d'exécution entre notre algorithme séquentiel et celui qui a motivé notre contribution [28] :l'algorithme de Demmel et Hida.

4. Mesurer les performances des algorithmes proposés

La contribution que nous allons détailler ci-après est liée à celle introduite précédemment qui concerne les nouveaux algorithmes de sommation. En effet, après avoir introduit nos algorithmes et les avoir testés sur des ensembles de données générés aléatoirement, nous voulons mettre en évidence l'efficacité de l'un des deux sur des méthodes de calcul numérique.

Nous nous sommes intéressés à la méthode de factorisation LU, la méthode de Simpson, la méthode de Jacobi, la méthode de puissance itérée et la multiplication de matrices. Pour chaque méthode numérique, nous comparons les résultats du programme original c'est-à-dire celui basé sur la somme récursive et les résultats du programme précis basé sur l'algorithme de somme que nous proposons.

Nous avons mesuré l'efficacité de notre algorithme sur ces méthodes en terme de précision numérique et de reproductibilité. De plus, nous allons montrer que notre algorithme accélère la convergence des méthodes itératives, i.e., il réduit le nombre d'itérations nécessaires aux algorithmes itératifs pour converger vers la solution.

5. Application aux réseaux de neurones

Nous allons introduire dans ce qui suit des travaux qui sont en cours d'élaboration et qui visent à étendre les résultats obtenus par nos algorithmes de sommation.

Nous voulons par cette contribution élargir le domaine d'application de la précision numérique. L'idée consiste à explorer un nouveau domaine auquel nous voulons intégrer l'un de nos algorithmes de sommation. Il s'agit de l'apprentissage des réseaux de neurones pour la résolution des problèmes de classification.

Nous allons réaliser des expérimentations afin de justifier l'intérêt pratique d'une somme précise dans le cas d'un apprentissage de réseaux de neurones par rapport à l'utilisation d'une somme récursive. Nous considérons à cet effet l'algorithme de

rétropropagation utilisé pour l'ajustement des poids de neurones lors de l'entraînement d'un réseau. Dans ce même contexte, nous mesurons l'efficacité de notre algorithme en comparant les erreurs générées à chaque cycle et le taux de reconnaissance dû à l'entraînent par deux versions de l'algorithme de rétropropagation (la première basée sur une somme récursive et une autre basée sur une somme précise).

Plan du document

La première partie de cette thèse est consacrée à l'état de l'art des différentes notions et aspects qui nous aiderons à comprendre le reste de ce manuscrit. Le chapitre 1 présente les concepts de base de l'arithmétique flottante, à savoir, la représentation des nombres, les différentes propriétés liées à la norme `IEEE-754`. Nous nous intéressons plus particulièrement aux principales sources d'erreurs qui nuisent à la précision numérique des calculs flottants afin de proposer à l'issue de notre travail des solutions qui remédient à ces problèmes par l'optimisation de la précision numérique. Au chapitre 2, nous rappelons d'une part quelques algorithmes de somme existants qui visent à additionner n nombres flottants avec plus de précision. D'une autre part, nous présentons quelques travaux liés à la reproductibilité des calculs impliquant des sommations. Nous mettons l'accent dans ce chapitre sur les algorithmes qui ont motivé nos contributions 3 et 4.

Quant à la deuxième partie, elle regroupe les travaux distincts réalisés durant cette thèse dans le cadre de l'amélioration de la précision numérique. La première méthode employée au chapitre 3 est la transformation automatique de code. En effet, notre objectif est d'étendre cette méthode développée dans l'outil `Salsa`, qui a montré son efficacité dans le cas séquentiel au cas parallèle en lui rajoutant une spécialisation du code de chaque processeur en fonction de ses données. Au chapitre 4, nous proposons une nouvelle technique basée sur l'analyse statique par interprétation abstraite. Cette technique permet de détecter et créer des partitions de même signe et ordre de grandeur. Par ordre de grandeur, nous désignons des valeurs d'un même exposant en base 10. Une fois les partitions créées, nous associons à chacune d'entre elles un algorithme de somme approprié. Dans ce contexte, nous proposons au chapitre 5 deux nouveaux algorithmes de somme, qui à la fois, améliorent la précision numérique des calculs flottants et assurent la reproductibilité. Ces algorithmes ont été testés aux travers des ensembles de données générés aléatoirement avec différentes proportions de grandes valeurs parmi moyennes et petites. De plus des résultats obtenus au chapitre 5, le chapitre 6 montre l'efficacité de l'un de nos algorithmes proposés sur des méthodes de calcul numérique. Pour aller plus loin, nous proposons d'étudier l'impact de notre algorithme précis dans le contexte de l'apprentissage des réseaux de neurones. Une étude préliminaire et ses résultats est présentée au chapitre 7. Nous concluons cette thèse en rappelant les principaux travaux réalisés et en proposant quelques perspectives pour des travaux futurs.

Publications

Les travaux réalisés dans cette thèse ont donné lieu aux publications suivantes :

— **International Minisymposium on Trusted Numerical Computations (TNC), 2018** [5].
Cet article représente une application de l'outil *Salsa* pour optimiser un programme parallèle représentatif du domaine du calcul haute performance. Le principe de cet outil consiste à transformer automatiquement les programmes de manière source à source en se reposant sur une analyse statique par interprétation abstraite. L'article montre l'approche que nous avons suivi pour spécialiser le code de chaque processeur en fonction des données d'entrée avant d'appliquer les transformations source à source à l'aide de *Salsa*.

— **International Workshop on Numerical and Symbolic Abstract Domains (NSAD), 2019** [6]
Dans cet article, nous décrivons une nouvelle technique qui se repose sur l'analyse statique par interprétation abstraite, et qui vise à améliorer la précision des calculs. Plutôt que de nous concentrer sur l'équilibrage de charge entre les processeurs, nous nous concentrons ici sur le partitionnement des données selon leur ordre de grandeur, afin d'éviter les éventuelles absorptions ou éliminations catastrophiques qui peuvent survenir au cours du calcul.

— **International Conference on Computing (SAI), 2021** [8]
Nous proposons dans cet article deux algorithmes parallèles efficaces pour le calcul de la somme de n nombres à virgule flottante. Le premier objectif de nos algorithmes est d'obtenir un résultat précis sans augmenter la complexité linéaire par rapport à l'algorithme de somme récursive. Le deuxième objectif est d'améliorer la reproductibilité des sommations par rapport à celles calculées par l'algorithme récursive et ce quel que soit le nombre de processeurs utilisés pour les calculs.

— **International Workshop on Numerical Software Verification (NSV), 2021** [7]
Nous avons introduit un nouvel algorithme parallèle pour sommer une séquence de nombres à virgule flottante. Dans cet article, notre principale contribution est une analyse approfondie de son efficacité par rapport à plusieurs propriétés : précision, convergence et reproductibilité. Pour montrer l'efficacité de notre algorithme, nous avons choisi un ensemble de méthodes numériques qui sont la factorisation LU, la méthode de Simpson, la méthode de Jacobi, et la méthode de puissance itérée.

Première partie

État de l'art

Chapitre 1

Arithmétique flottante IEEE−754

Contents

1.1 Introduction 9
1.2 Le standard IEEE−754 10
1.3 Les nombres 10
1.4 Les formats 11
1.5 Les modes d'arrondi 12
1.6 Les fonctions ufp et ulp 12
1.7 Sources d'erreurs 13
1.7.1 Les erreurs d'arrondi 13
1.7.2 Absorption 13
1.7.3 Élimination catastrophique 14
1.8 Estimation d'erreurs en précision finie 14
1.9 Conclusion 14

1.1 Introduction

Jusqu'aux années 80, chaque constructeur de processeurs avait sa propre implémentation de l'arithmétique flottante, et ceci en fonction de la valeur de la base β utilisée et de l'intervalle de l'exposant $[exp_{min}, exp_{max}]$. La variation de ces deux paramètres cause la non-portabilité des programmes d'une architecture à une autre, en d'autres termes l'exécution d'un même programme sur plusieurs machines aboutit à des résultats différents. D'où la nécessité de standardiser et d'homogénéiser l'implémentation de l'arithmétique à virgule flottante en base 2, ce qui a donné naissance à la norme IEEE−754 [2, 33, 63].
La norme IEEE−754 est le standard scientifique permettant de spécifier l'arithmétique à virgule flottante. Cette norme a permis de :

— Fixer les formats de représentation de données ainsi que leurs encodages en machine,

— Définir les modes d'arrondi ;

— Définir les valeurs spéciales ainsi que les exceptions qui peuvent survenir lors d'un calcul ;

— Définir la précision des quatre opérations de base $(+, -, *, /)$, etc.

Dans ce chapitre, nous allons introduire les concepts de base de l'arithmétique flottante. Nous commençons par définir les nombres flottants ainsi que leurs représentations en machine. Nous continuerons par quelques propriétés fixées par le standard IEEE−754 sur les flottants à savoir : les formats de représentation, les modes d'arrondi, les valeurs spéciales. À la fin, une partie sera dédiée aux principales sources d'erreurs numériques et à la façon dont on peut les estimer.

1.2 Le standard IEEE−754

L'arithmétique des ordinateurs est basée sur l'utilisation des nombres à virgule flottante, qui ne sont pas représentables d'une manière exacte en mémoire sur machine. Les nombres flottants sont donc utilisés pour représenter les nombres réels d'une manière approximative. Ces approximations font que le résultat de chaque opération arithmétique n'est pas exact, ce qui engendre une perte d'information causée par les erreurs d'arrondi. La représentation d'un nombre réel x en virgule flottante, en base β (généralement égale à 2 sur ordinateur), est de la forme :

$$x = s \cdot m \cdot \beta^{exp-f+1} \tag{1.1}$$

où,
s représente le **signe** de x avec $s \in \{-1, 1\}$,
La **mantisse** m en base β est représentée par une suite de chiffres entiers $m = d_0.d_1.d_2....d_{f-1}$ tels que

- $0 \leq d_i < \beta$,
- f la précision (nombre de chiffres significatifs).

L'exposant exp est un entier signé défini par

$$exp_{min} \leq exp \leq exp_{max} \tag{1.2}$$

avec les deux entiers relatifs exp_{min} et exp_{max} donnés pour chaque format de représentation défini par la norme IEEE−754 (voir tableau 1.1).

Remarque 1.1 *Le bit de poids fort d_0 est un bit implicite (bit caché) dont la valeur se déduit de celle de l'exposant exp.*

1.3 Les nombres

À partir d'un ensemble de nombres flottants donné, nous pouvons faire la distinction entre les nombres flottants normalisés et les nombres flottants dénormalisés, selon les

caractéristiques décrites par la norme IEEE−754.

Définition 1.1 *Un nombre à virgule flottante est dit normalisé si* $d_0 \neq 0$*. La normalisation évite les représentations multiples du même nombre flottant.*

Exemple 1.1 *Soit* $x = 0,625$ *un nombre flottant dans le format binary32. La représentation du nombre* x *en binaire est donnée par* $x = (0,101)_2$*, après normalisation* $x = 1,01 \times 2^{-1}$*.*
Par conséquent, $exp = -1 + 127 = 126$*, alors*
$(exp)_2 = 011111110$
$m = 010 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0000$

Définition 1.2 *Dans le cas des nombres dénormalisés, le premier chiffre* d_0 *de la mantisse d'un nombre flottant* x *est nulle avec un plus petit exposant possible* $exp = exp_{min}$*.*

Exemple 1.2 *Soit* $x = 2^{-149}$ *le plus petit nombre positif dénormalisé du format binary32. Les valeurs de l'exposant et de la mantisse sont données par*
$exp = 00000000$
$m = 000 \quad 0000 \quad 0000 \quad 0000 \quad 0000 \quad 0001$

Enfin, les valeurs spéciales suivantes sont également définies :

- NaN (de l'anglais Not a Number) pour désigner le résultat d'une opération invalide : $x \div 0$, $\sqrt{-x}$, ... ;
- Les valeurs d'infinis positifs et négatifs $\pm\infty$ correspondant à un overflow ;
- les valeurs $+0$ et -0 (zéros signés).

1.4 Les formats

La norme IEEE−754 définit multiples formats de représentation des nombres à virgule flottante en base 2. Cependant, chaque format de représentation dépend des valeurs de f, β, exp_{min}, exp_{max}, comme le montre le tableau 1.1 ci-dessus :

Format	# bits total	Précision(bits)	Exposant(bits)	exp_{min}	exp_{max}
Simple/ binary32	32 bits	24	8	-126	+127
Double/binary64	64 bits	53	11	-1022	+1023
Quadruple/ binary128	128 bits	113	15	-16382	+16383

TABLE 1.1 – *Formats de représentation de la norme IEEE−754*

Remarque 1.2 *Nous ne considérons pas les formats simple étendu et double étendu, également définis par la norme IEEE−754.*

Exemple 1.3 *Quelques exemples de représentation de nombres flottants codés dans le format simple précision sont donnés ci-dessous*

x	s	exp	mantisse m					
$1.18 \times 10^{-38} \approx 2^{-126}$	1	00000001	000	0000	0000	0000	0000	0000
3.4×10^{38}	1	11111110	111	1111	1111	1111	1111	1111
-0.0	-1	00000000	000	0000	0000	0000	0000	0000
$-\infty$	-1	11111111	000	0000	0000	0000	0000	0000
NaN	$-1 \vee 1$	11111111	010	0000	0000	0000	0000	0000

1.5 Les modes d'arrondi

Pour traiter les nombres flottants, où le résultat d'une opération sur les flottants n'est pas forcément un flottant, le résultat doit être arrondi vers l'un des deux flottants les plus proches du résultat réel, en fonction du mode d'arrondi choisi parmi ceux définis par le standard IEEE−754. Quatre modes d'arrondi ont été définis pour les opérations élémentaires sur les nombres à virgule flottante. Ces modes sont vers $+\infty$, vers $-\infty$, vers zéro et au plus près notés par $\circ_{+\infty}$, $\circ_{-\infty}$, $\circ_0$ et $\circ_\sim$, respectivement. Le comportement des opérations élémentaires $\diamond \in \{+, -, \times, \div\}$ entre les nombres à virgule flottante est donné par l'équation (1.3)

$$v_1 \diamond_\circ v_2 = \circ(v_1 \diamond v_2) \tag{1.3}$$

où $\circ$ est le mode d'arrondi tel que $\circ \in \{\circ_{+\infty}, \circ_{-\infty}, \circ_0, \circ_\sim\}$. Par l'équation (1.3), nous illustrons que dans les calculs en virgule flottante, effectuer une opération élémentaire $\diamond_\circ$ avec le mode d'arrondi $\circ$ renvoie le même résultat que celui obtenu par une opération exacte $\diamond$, puis arrondi avec $\circ$.

Propriété 1.1 *Pour une précision f et un mode d'arrondi au plus près,*

$$\circ_\sim (v_1 \diamond v_2) < 2^{exp-f+1} \tag{1.4}$$

avec, $exp = ufp(v_1 \diamond v_2)$

La norme IEEE−754 ne spécifie rien concernant les fonctions telles que sin, log, etc. Contrairement à la fonction racine carrée qui est incluse dans les opérations de base.

1.6 Les fonctions ufp et ulp

Les équations (1.6) et (1.5) représentent les deux fonctions notées par ufp (de l'anglais the Unit in the First Place) et ulp (de l'anglais the Unit in the Last Place). $L'ufp$ désigne le poids du premier bit de la mantisse d'un nombre flottant. Il est à noter que $l'ufp$ est

indépendant du nombre de bits de la mantisse. Quant à $l'ulp$ c'est le poids du dernier bit de la mantisse. En effet, il existe plusieurs définitions de la fonction $d'ulp$ [49]. Nous considérons dans ce qui suit que $l'ulp$ d'un nombre flottant x avec f bits significatifs est donné par

$$ulp(x) = \begin{cases} \beta^{\lfloor \log_\beta |x| \rfloor - f + 1} & si \quad |x| \geq \beta^{e_{min}}, \\ \beta^{e_{min} - f + 1} & sinon. \end{cases} \tag{1.5}$$

$L'ufp$ d'un nombre à virgule flottante x est défini par

$$ufp(x) = \begin{cases} 2^{\lfloor \log_2 |x| \rfloor} & si \quad x \neq 0, \\ 0 & si \quad x = 0. \end{cases} \tag{1.6}$$

Ces fonctions seront utilisées plus loin (voir chapitre 5) pour calculer les erreurs d'arrondi causées par les algorithmes de sommations.

1.7 Sources d'erreurs

Les calculs basés sur l'arithmétique flottante peuvent renvoyer dans certains cas des résultats peu intuitifs à cause des erreurs survenant au cours de calcul. Les sources de ces erreurs sont multiples, commençons par la source principale qui est les erreurs d'arrondi. Nous citons également, les deux phénomènes d'absorption et d'élimination catastrophique.

1.7.1 Les erreurs d'arrondi

Les erreurs d'arrondi surviennent soit au moment de la représentation d'un nombre réel par un flottant selon un mode d'arrondi parmi les quatre décrits en section 1.5, soit lors d'un calcul impliquant plusieurs opérations arithmétiques où chaque opération introduit une erreur d'arrondi qui se propage tout au long des calculs.

1.7.2 Absorption

Une absorption se produit lors de l'addition de deux nombres à virgule flottante avec des ordres de grandeur différents. La petite valeur est alors absorbée par la plus grande.

Exemple 1.4 *Considérons la somme* $s = ((x + y) + z)$ *tels que* x, y *et* z *sont trois nombres à virgule flottante stockés au format binary32, où* $x = 1.2 \times 2^0$, $y = 1 \times 2^{24}$ *et* $z = -1 \times 2^{24}$, *respectivement. Le calcul de* s *produit le résultat* $s = 0.0$ *qui n'a pas de bits de mantisse corrects, car le résultat réel correct devrait être* $s = 1,2 \times 2^0$.

1.7.3 Élimination catastrophique

Une annulation se produit lorsque deux nombres presque égaux sont soustraits et que les chiffres les plus significatifs s'annulent. La gravité de l'annulation diffère d'un calcul à l'autre en fonction du nombre de bits perdus.

Exemple 1.5 *Soit* $x = 1.5 \times 2^0$ *et* $y = 1 \times 2^{24}$ *deux nombres à virgule flottante en simple précision. Le calcul de la séquence d'opérations* $s = ((x+y)-y)$ *produit le résultat* $s = 0.0$ *alors que le résultat réel devrait être* 1.5×2^0*.*

1.8 Estimation d'erreurs en précision finie

Comme vu précédemment, le problème majeur du calcul flottant est la propagation des erreurs d'arrondi. En effet, pour représenter un réel, on doit l'arrondir. Cette opération d'arrondi introduit une erreur même dans le cas où la valeur du réel est connue (par exemple $\frac{1}{3}$).

Si on note x une valeur réelle, sous la représentation flottante en base 2, voir l'équation (1.1), et soit $\hat{x}$, la valeur approximée de x. Deux types d'erreurs peuvent être estimées, soit l'erreur absolue et l'erreur relative.

Définition 1.3 *(Erreur absolue). L'erreur absolue* $Erreur_{abs}$ *entre un nombre réel* x *et sa valeur approchée* $\hat{x}$ *est calculée comme suit :*
$Erreur_{abs} = |x - \hat{x}|$

Définition 1.4 *(Erreur relative). L'erreur relative* $Erreur_{rel}$ *est le résultat de la division de l'erreur absolue par la valeur exacte* x *:*
$Erreur_{rel} = \frac{|x-\hat{x}|}{|x|}$

Remarque 1.3 *La valeur des erreurs absolues et relatives dépend à la fois de la précision du format* f *et du mode d'arrondi* $\circ$ *utilisé pour effectuer le calcul.*

1.9 Conclusion

Nous avons consacré ce chapitre à l'introduction des nombres à virgule flottante, ainsi que leurs propriétés telles qu'elles sont décrites par le standard IEEE−754. Nous rappelons que notre objectif principal est l'amélioration de la précision numérique des calculs flottants. De ce fait, nous avons abordé le sujet des sources d'erreurs numériques qui rendent dans certains cas nos résultats incorrects. De plus, nous avons présenté les moyens de mesurer ces erreurs que nous utiliserons par la suite pour mesurer la précision de nos calculs.

Chapitre 2

Algorithmes de sommation existants

Contents

2.1 Introduction **15**

2.2 Somme récursive **16**

2.3 Somme précise **16**

2.3.1 Les transformations sans erreur 17

2.3.2 Somme compensée 18

2.3.3 Somme en cascade 19

2.3.4 Algorithmes de Demmel et Hida 20

2.4 Somme reproductible **21**

2.5 Conclusion **22**

2.1 Introduction

Dans ce chapitre, nous allons rappeler quelques algorithmes existants conçus pour le calcul de la somme de n nombres flottants et les solutions proposées pour prévenir ou pour corriger les comportements indésirables de l'arithmétique flottante qui mènent à des résultats de sommations imprécis et/ou non reproductibles. En effet, la somme de nombres à virgule flottante est l'une des tâches les plus élémentaires de l'analyse numérique. En arithmétique à virgule flottante, la somme entraîne des erreurs d'arrondi. Ces erreurs d'arrondi peuvent s'accumuler et se propager au fil du calcul, par conséquent, le résultat final peut devenir peu intuitif.

Dans ce même contexte, plusieurs travaux de recherche se sont concentrés sur l'amélioration de la précision numérique [27, 28, 54, 62, 65, 74] ou la reproductibilité [1, 29, 30, 46] des calculs impliquant des sommations. Il existe de nombreux algorithmes séquentiels et parallèles pour cette tâche, un bon aperçu de ces travaux est présenté dans [43, 44]. En plus des algorithmes séquentiels existants, de nombreux autres algorithmes parallèles ont été proposés. Leuprecht et Oberaigner [60] décrivent des algorithmes parallèles pour additionner des nombres à virgule flottante. Les auteurs proposent une version pipeline d'algorithmes séquentiels [10, 71] dédiés au calcul de l'arrondi exact de la somme de nombres à virgule flottante.

Ce chapitre commence par rappeler brièvement en section 2.2 l'algorithme de somme récursive en précisant l'erreur numérique qui découle de ce calcul. Nous présentons ensuite les solutions existantes pour remédier aux problèmes de précision à la section respectives 2.3 et aux problèmes de reproductibilité à la section 2.4.

2.2 Somme récursive

L'algorithme le plus basique utilisé pour le calcul de la somme est celui appelé algorithme de somme récursive. En effet, pour calculer la somme S de n nombres à virgule flottante tels que $S = \sum_{i=0}^{n-1} s_i$ avec l'algorithme de somme récursive [50] donné par l'équation (2.1),

$$S = (((s_1 + s_2) + s_3) + ... + s_n) \tag{2.1}$$

nous constatons qu'une erreur absolue est générée entre le résultat de la somme exacte et le résultat de la somme approchée. Cette erreur peut être majorée par l'équation (2.2) [50]. Cette borne sera utilisée dans la suite de ce manuscrit pour comparer l'erreur d'arrondi dans le pire des cas entre l'algorithme récursive et nos algorithmes détaillés dans le chapitre 5.

$$\mid \hat{S} - \sum_{i=0}^{n-1} s_i \mid \leq \sum_{i=0}^{n} \mid e_i \mid \leq \frac{(n-1)u}{1+u} \sum_{i=0}^{n-1} \mid s_i \mid, \tag{2.2}$$

où

— $\hat{S}$ représente la solution approchée,

— $\mid e_i \mid$ désigne l'erreur absolue de la $i^{ème}$ addition à virgule flottante,

— $u = \frac{1}{2}\beta^{1-p} = \frac{1}{2}ulp(1)$.

2.3 Somme précise

La sommation à virgule flottante est souvent améliorée par les méthodes de sommations compensées [54, 62, 65, 74, 75, 76] avec ou sans l'utilisation des transformations sans erreur pour calculer l'erreur introduite par chaque accumulation. Nous allons détailler dans cette section quelques algorithmes de somme compensée. La précision des algorithmes de sommation peut également être améliorée en manipulant l'exposant et la mantisse des nombres à virgule flottante afin de diviser les données avant de commencer les calculs [27, 28]. Cette approche est celle employée dans les chapitres 5 et 6 et elle est expliquée en détail dans la section 2.3.4.

2.3.1 Les transformations sans erreur

Il est bien connu que le calcul avec une précision finie implique des erreurs d'arrondi. Ces erreurs conduisent souvent à des résultats inexacts pour un calcul donné. Un outil important pour essayer d'éviter ces erreurs d'arrondi est les transformations sans erreur qui peuvent être vues comme des nombres à virgule flottante double-double mais sans l'étape de renormalisation. En d'autre terme, les transformations sans erreur permettent d'effectuer les calcul sur deux fois plus de précision que celle autorisée par le format choisi. Plusieurs algorithmes de transformations sans erreur ont été proposés pour les quatre opérations élémentaires $\{+, -, \times, \div\}$, nous nous intéressons dans ce qui suit à l'addition (voir les algorithmes 1 et 2) et à la multiplication (voir les algorithmes 4 et 5).

Soit a et b deux nombres flottants encodés dans l'un des formats binaires spécifiés par la norme IEEE−754. Supposons que l'on travaille dans le mode d'arrondi au plus proche. L'algorithme le plus utilisé dans le contexte des transformations sans erreur pour l'addition est l'algorithme `TwoSum` proposé par Knuth en 1969 [56] et détaillé ci-dessous.

Algorithme 1 Transformation sans erreur pour le calcul d'une somme de deux nombres flottants [56]

Fonction$[r, s]$ = `TwoSum`(a, b)
$r \longleftarrow a + b$
$z \longleftarrow r - a$
$s \longleftarrow (a - (r - z)) + (b - z)$

L'algorithme 1 illustre que $a + b = r + s$ avec $r = a + b$. La valeur de s, quant à elle, représente le nombre flottant correspondant à l'erreur d'arrondi calculée.

Citons également à titre d'information que Dekker a proposé en 1971 un algorithme [26] appelé `FastTwoSum` qui calcule le résultat de l'opération d'addition flottante plus le terme d'erreur d'arrondi dû à ce calcul. La particularité de l'algorithme 2 réside dans le fait que ce dernier calcule la somme exacte que dans le cas où la condition $(|a| \geq |b|)$ sur les deux valeurs a et b est vérifiée contrairement à l'algorithme 1 qui ne met aucune condition sur les valeurs d'entrées.

Algorithme 2 Transformation sans erreur pour le calcul d'une somme de deux nombres flottants [26]

Fonction$[r, s]$ = `FastTwoSum`(a, b)
Si $|a| \geq |b|$
  $r \longleftarrow a + b$
  $s \longleftarrow (a - r) + b$

Il est possible de calculer la transformation exacte de la multiplication de deux nombres

à virgule flottante à l'aide de la fonction `TwoProd` [26]. Pour calculer l'erreur commise lors du calcul d'un produit, l'algorithme 4 fait appel à une fonction de découpage d'un nombre flottant a en deux nombres flottants a_h et a_l avec des mantisses de même longueur, tels que : $a = a_h + a_l$. Considérons l'algorithme 3 de découpage de Dekker et Veltkamp [26]. Soient $s = \lceil f/2 \rceil$, avec f le nombre de bits de la mantisse et C une constante.

Algorithme 3 Découpage d'un nombre flottant [26]

Fonction$[a_h, a_l] = \texttt{Split}(a)$

$C \longleftarrow 2^s + 1$

$\alpha \longleftarrow C \times a$

$a_h \longleftarrow \alpha - (\alpha - a)$

$a_l \longleftarrow a - a_h$

Comme nous pouvons constater, l'algorithme 4 nécessite plusieurs opérations flottantes pour calculer cette transformation exacte.

Algorithme 4 Transformation sans erreur pour le calcul d'un produit de deux nombres flottants [26]

Fonction$[r, s] = \texttt{TwoProd}(a, b)$

$r \longleftarrow a \times b$

$[a_h, a_l] \longleftarrow Split(a)$

$[b_h, b_l] \longleftarrow Split(b)$

$s \longleftarrow a_l \times b_l - (((r - a_h \times b_h) - a_l \times b_h) - a_h \times b_l)$

En outre, la fonction `TwoProdFMA` est proposée pour les processeurs disposant des FMA (Fused-Multiply Add). Cependant, l'algorithme 5 calcule l'erreur sur la multiplication à l'aide du FMA en seulement deux opérations flottantes.

Algorithme 5 Transformation sans erreur pour le calcul d'un produit de deux nombres flottants en présence du FMA [26]

Fonction$[r, s] = \texttt{TwoProdFMA}(a, b)$

$r \longleftarrow a \times b$

$s \longleftarrow FMA(a, b, -r)$

2.3.2 Somme compensée

Comme nous l'avons déjà vu au chapitre 1, l'erreur dans une addition à virgule flottante est elle-même un nombre à virgule flottante. En d'autres termes, pour tout couple (a, b)

de nombres à virgule flottante et leur somme calculée $\hat{S} = a \oplus b$, il existe un nombre flottant e tel que

$$a + b = \hat{S} + e. \tag{2.3}$$

Le principe des algorithmes de somme compensée est alors d'approximer ce terme d'erreur à chaque étape de sommation récursive afin de corriger la somme calculée.

Nous nous concentrons dans ce qui suit sur l'algorithme 6 pour illustrer le calcul de l'erreur d'arrondi exacte sur une addition [53, 64]. Notons que ce processus de compensation peut être appliqué de manière récursive produisant un algorithme compensé en cascade (voir section 2.3.3).

Algorithme 6 Somme compensée

Somme de n nombres à virgule flottante $S = \sum_{i=1}^{n} x_i$

1: S=0, e=0
2: **for** i=1 *to* n **do**
3: z=S
4: y=x_i+e
5: S=z+y
6: e=(z-S)+y

De nombreux algorithmes de ce type ont été conçus et analysés au fil des ans. Rump et al. [65, 74, 75] ont proposé plusieurs algorithmes pour la sommation et le produit scalaire de nombres à virgule flottante. Ces algorithmes compensés sont fondés sur l'application itérative des transformations sans erreur (EFT) qui sont utilisées pour compenser les erreurs d'arrondi accumulées au fil des calculs. La technique de compensation des erreurs d'arrondi a été abordée dans plusieurs travaux afin d'améliorer la précision du résultat de calcul [39, 40]. Thévenoux et al. [80] ont implémenté une transformation de code automatique pour dériver un programme compensé.

2.3.3 Somme en cascade

La sommation par paires également appelée sommation en cascade est une technique permettant de sommer une séquence de nombres à virgule flottante de précision finie qui réduit considérablement l'erreur d'arrondi accumulée par rapport à l'accumulation dans le cas de la somme récursive 2.2. L'idée consiste à calculer des sommes locales par paire de valeurs jusqu'à ce que la somme de deux termes soient obtenues comme le montre la figure 1. Il y a donc au plus $\lceil \log_2 n \rceil$ niveaux de récursion.

Parmi les travaux liés à cette classe d'algorithmes, nous trouvons Malcolm [62] qui a décrit des méthodes en cascade basées sur la plage limitée des exposants des nombres à virgule flottante. Il définit des accumulateurs en extra précision où chaque composante e_i correspond à un exposant. Pour extraire et mettre à l'échelle l'exposant, Malcolm utilise une division entière, sans exiger que la division soit une puissance de 2. Pour le calcul

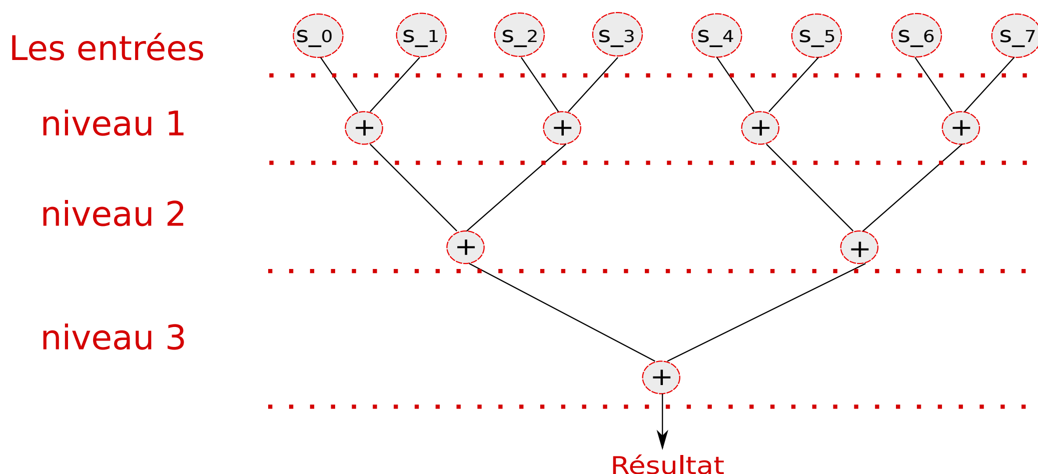

FIGURE 1 – Calcul de somme en cascade.

d'une somme de n nombres flottants $S = \sum_{i=1}^{n} s_i$, l'idée consiste à décomposer chaque nombre s_i en q parties formant ainsi $n \times q$ valeurs à sommer dans des cellules appropriées selon leurs exposants. À la fin, ces valeurs sont additionnées dans l'ordre décroissant.

2.3.4 Algorithmes de Demmel et Hida

Dans cette section, nous décrivons d'autres algorithmes dédiés au calcul précis de la somme à virgule flottante. Il s'agit des algorithmes introduits par Demmel et Hida [28] qui ont motivé nos contributions aux chapitres 5 et 6. Ces algorithmes partagent quelques idées communes avec nos algorithmes introduits en détails dans le chapitre 5. Étant données deux précisions f et F avec $F > f$, les algorithmes de Demmel et Hida utilisent un accumulateur de précision F pour additionner n nombres à virgule flottante de précision f dans l'ordre décroissant. Dans l'algorithme 7, n nombres à virgule flottante donnés en précision f sont triés dans l'ordre décroissant de leurs valeurs absolues, par la suite, ces nombres sont additionnés dans un accumulateur en extra précision F afin de calculer $S = \sum_{i=1}^{n} s_i$.

Algorithme 7 Somme précise avec tri d'exposants (Algorithme 2 dans [28])

1: Sort the s_i such that $exp(s_1) \geq exp(s_2) \geq \cdots \geq exp(s_n)$
2: $S \leftarrow 0$ //S has F bits of precision.
3: **for** $i \leftarrow 1$ to n **do**
4: S $\leftarrow S + s_i$ //s_i has f bits of precision
5: $s \leftarrow$ round(S)

D'autre part, l'algorithme 8 nécessite un tableau fixe d'accumulateurs $A_0, ..., A_N$ de précision F, chaque A_j accumule la somme d'un sous-ensemble des s_i. Ensuite, ces A_j

seront ajoutés à l'aide de l'algorithme 7. Cet algorithme nécessite d'accéder à l'exposant

Algorithme 8 Somme précise avec N accumulateurs (Algorithme 3 dans [28])

1: Choose parameter e (number of leading exponent bits of each s_i to extract).
2: Initialize $A_k = 0$ for $k = 0, 1, ..., N$, where $N = 2^e$.
3: **for** i=1 to n **do**
4: $j \leftarrow$ leading e bits of the exponent of s_i
5: $A_j \leftarrow A_j + s_i$ //from N to 0
6: Add the A_j yielding S of precision F
7: $s \leftarrow$ round(S) //round S back to f-bits

de chaque s_i pour décider à quel accumulateur A_j l'ajouter. En supposant que j sont les premiers bits de l'exposant de s_i, l'algorithme 8 divise la plage d'exposants de s_i en groupes de puissances de deux. Demmel et Hida [28] donnent des détails sur la précision du résultat calculé en fonction de f, F et du nombre de sommations effectuées. Contrairement à notre travail qui n'utilise que la précision du calcul avec un accès aux exposants, les algorithmes de Demmel et Hida ont besoin dans tous les cas d'un format à virgule flottante de précision supplémentaire et également d'un accès à la mantisse. Notons également que ces algorithmes ont une complexité de $O(n \log n)$ suite à l'étape de tri, par rapport à un algorithme de somme récursive de complexité $O(n)$ où n est le nombre de sommations.

2.4 Somme reproductible

Avoir des résultats reproductibles, i.e. identiques à partir de plusieurs exécutions du même programme sur des architectures parallèles différentes ou même similaires est important pour plusieurs raisons. Par exemple, pour déboguer et tester un programme, pour reproduire des résultats de recherche publiés précédemment, etc. Pour remédier à ces problèmes, plusieurs solutions ont été proposées.

Une première consiste à définir un ordre déterministe des calculs et comment ces calculs sont affectés aux processeurs. Cette approche a été considérée comme peu pratique et ne s'adapte pas bien au nombre de processeurs. À titre d'exemple, les routines CuBLAS [1] de NVIDIA.

Une deuxième solution vise à réduire ou à éliminer les erreurs d'arrondi, elle est basée principalement sur l'utilisation d'accumulateurs reproductibles [14, 15, 29, 30, 46, 57]. Dans ce même contexte, citons quelques techniques.

— La sommation avec pré-arrondi, l'idée principale est de pré-arrondir les valeurs à virgule flottante à une précision fixe selon une certaine limite. Cependant, la taille des accumulateurs est réduite et l'erreur ne dépend que des valeurs d'entrée et de la limite, contrairement aux résultats intermédiaires qui dépendent de l'ordre des

1. http ://docs.nvidia.com/cuda/cublas/index.html

calculs. Demmel et Nguyen ont introduit une famille d'algorithmes [1, 29, 30] qui s'articulent autour de cette technique. Pour le calcul d'une somme parallèle à virgule flottante indépendamment de l'ordre de sommation, Demmel et Nguyen proposent de s'appuyer sur le pré-arrondi pour obtenir des erreurs d'arrondi déterministes et d'utiliser l'algorithme de Rump [75] pour une transformation vectorielle sans erreur. Notons que la méthode proposée par Demmel et Nguyen est similaire à celle décrite par Rump et al. [74, 75].

— La sommation avec arrondi fidèle, pour un nombre machine donné, son arrondi fidèle correspond à l'un des deux nombres qui l'encadrent, et en cas d'égalité c'est le nombre machine lui même (voir équation 1.3, chapitre 1). L'arrondi fidèle rend les opérations élémentaires déterministes, par conséquent, cette technique est reproductible. De plus des accumulateurs nécessaires pour accumuler toutes les sommes partielles [14, 15, 57], cette technique nécessite dans certains cas plusieurs passages aux données en les réécrivant afin d'éliminer les annulations catastrophiques [14, 75].

2.5 Conclusion

Il existe plusieurs moyens d'améliorer la précision numérique des calculs. Une première solution est d'utiliser des algorithmes qui ont été conçus à ce propos et qui sont basés soit sur des méthodes de réécriture de code ou d'extension de précision de calcul. Une deuxième solution consiste à utiliser des bibliothèques de calcul scientifique comme par exemple `MPFR` [31] la bibliothèque d'arithmétique flottante binaire en multiprécision avec arrondi correct ou `CADNA` [51] qui est basé sur des approches stochastique pour estimer et contrôler les erreurs d'arrondi commises durant le calcul.

Dans ce chapitre, nous avons mis l'accent sur quelques algorithmes de somme existants dans la littérature. Nous avons commencé par rappeler l'algorithme de somme récursive puis citer les principales solutions proposées à savoir : les transformations sans erreur et les compensations.

Nous avons également cité une autre méthode d'optimisation de la précision numérique au travers des algorithmes proposés par Demmel et Hida. Ces algorithmes s'appuient sur une extension de la précision de calcul tout en manipulant les exposants et mantisses des valeurs à sommer. Notons que l'approche suivie dans le cadre de notre travail est similaire à celle-ci, nous développerons notre technique au cours des chapitres suivants.

Deuxième partie

Vers l'amélioration de la précision numérique de la somme

Chapitre 3

Application de transformation de programme

Contents

3.1 Introduction 25

3.2 Transformation de programmes 26

3.2.1 Transformation des expressions arithmétiques 26

3.2.2 Outil de transformation automatique de programmes 27

3.3 Résultats expérimentaux 30

3.3.1 EDP de la propagation de chaleur 30

3.3.2 Précision numérique 32

3.3.3 Temps d'exécution 35

3.4 Conclusion 37

3.1 Introduction

Dans ce chapitre nous allons nous intéresser à quelques techniques existantes d'analyse et d'amélioration de la précision numérique des calculs basés sur l'arithmétique flottante. En effet, plusieurs méthodes d'analyse de la précision numérique des calculs flottants ont été proposées. Souvent, ces méthodes nous donnent une surestimation de la pire erreur numérique qui peut survenir lors d'un calcul. On peut citer l'outil `Fluctuat` [36, 37] basé sur l'analyse statique par interprétation abstraite, utilisé dans le domaine de l'industrie. L'avantage de ces méthodes est qu'elles sont capables de calculer les bornes sur les erreurs d'arrondis et de fournir à l'utilisateur une idée sur l'origine de l'erreur, ce qui va guider par la suite le processus d'amélioration de la précision numérique. Darulova et Kuncak ont proposé un outil, `Rosa`, qui utilise à la fois l'analyse statique et un solveur SMT pour calculer la propagation d'erreurs [25]. Solovyev et al. ont proposé un autre outil, `FP-Taylor` [77]. Il est à noter qu'aucune des techniques mentionnées ci-dessus n'optimise la précision numérique des programmes.

Par ailleurs, plusieurs outils ont été développés pour améliorer la précision numérique des calculs, tel que `Sardana` [47] qui améliore la précision numérique des expressions arithmétiques en appliquant une transformation source à source. Ensuite, à l'aide d'une analyse

statique, il effectue un choix de la meilleure expression. Contrairement à `Herbie` [66] qui est également un outil pour l'amélioration de la précision numérique des expressions arithmétiques mais le choix de la meilleure expression se fait à l'aide d'une analyse dynamique sur des données aléatoires. La limitation de ces outils consiste au fait qu'ils ne sont applicables que pour les expressions arithmétiques. En revanche, nous trouvons `Salsa` [18], qui est un outil de transformation automatique de code qui comporte non seulement des expressions arithmétiques mais aussi des codes avec affectations, boucles, conditionnelles, fonctions, etc. en un autre code source plus précis.

Notre objectif principal est d'étendre ces méthodes de transformation automatique de programmes, afin de résoudre les problèmes numériques liés au calcul parallèle. Plus précisément, nous avons amélioré la précision numérique, par des transformations automatiques de codes, tout en spécialisant le code de chaque processeur. En d'autres termes, par des transformations de codes SPMD (Single Program Multiple Data) en codes MIMD (Multiple Instructions Multiple Data) [5].

Ce chapitre est organisé comme suit. Nous donnons à la section 3.2 un aperçu de la transformation d'expressions ainsi que de codes. À la section 3.3 nous détaillons notre cas d'étude et l'approche appliquée pour améliorer la précision numérique de ses calculs suivi des différents résultats expérimentaux obtenus.

3.2 Transformation de programmes

Dans cette section, nous commençons par présenter une des techniques (thèse de A. Ioualalen) de transformation automatique d'expressions arithmétiques dans le but d'améliorer leur précision numérique. Ensuite nous présentons l'outil `Salsa` (thèse de N. Damouche) qui a étendu l'optimisation de la précision numérique sur des codes contenant en plus des expressions arithmétiques, des boucles, des conditions, des fonctions, etc.

3.2.1 Transformation des expressions arithmétiques

Nous présentons ici brièvement les travaux de thèse [47] de A. Ioualalen qui portent sur la transformation [26] des expressions arithmétiques en utilisant les `APEGs` [47, 23, 79]. Les `APEGs`, abréviation de Abstract Program Equivalent Graph, est une représentation permettant de regrouper un nombre exponentiel d'expressions arithmétiques équivalentes dans une structure polynomiale. Cette représentation permet de réduire la complexité de la transformation à la fois en temps et en taille. Comme le montre la figure 2, un `APEG` se compose de classes d'équivalence représentées par des ellipses (en pointillés sur la figure) qui contiennent des opérations, et des boîtes. Pour pallier aux problèmes d'explosion combinatoire, ces boîtes contiennent plusieurs expressions arithmétiques équivalentes à base de commutativité, d'associativité et de distributivité, autrement dit ce sont dif-

férentes façons de combiner les opérandes et les opérations pour avoir des expressions mathématiquement équivalentes. Afin de construire un `APEG`, deux algorithmes sont utili-

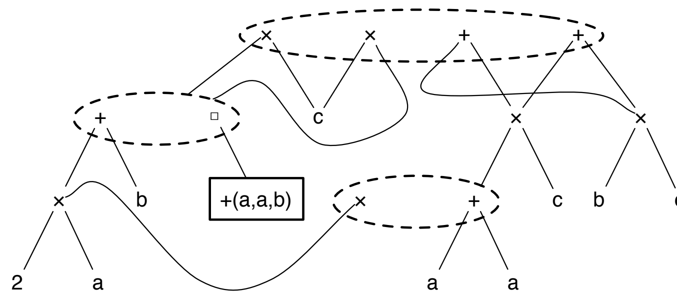

FIGURE 2 – APEG correspondant à l'expression $((a+a)+b)\times c$ (figure 1 dans [24]).

sés (algorithmes de propagation et d'expansion). Le premier cherche récursivement dans `l'APEG` les opérateurs binaires symétriques, qui seront mis par la suite dans les boîtes abstraites. Le deuxième cherche dans `l'APEG` une expression plus précise parmi toutes les expressions équivalentes. À la fin nous recherchons l'expression arithmétique la plus précise à l'aide de l'interprétation abstraite.

Un exemple de construction `d'APEG` est donné à la figure 2. Chaque classe d'équivalence, représentée par des ellipses en pointillées contient plusieurs `APEGs` $p_i......p_n$. Pour former une expression, on doit effectuer un choix d'une seule expression p_i tel que : $1 \leq i \leq n$ dans chaque classe d'équivalence. Une boîte $\boxed{*(p_1,\ldots,p_n)}$ regroupe tous les parenthésages possibles de l'opération $* \in \{+,\times\}$ pour les opérandes $p_i......p_n$.
Une représentation de toutes les expressions équivalentes correspondant à `l'APEG` de l'expression $((a+a)+b)\times c$ de la figure 2 est donné par l'équation 3.1.

$$\mathcal{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big)\times c,\ \big((a+b)+a\big)\times c,\ \big((b+a)+a\big)\times c,\ \big((2\times a)+b\big)\times c, \\ c\times\big((a+a)+b\big),\ c\times\big((a+b)+a\big),\ c\times\big((b+a)+a\big),\ c\times\big((2\times a)+b\big), \\ (a+a)\times c+b\times c,\ (2\times a)\times c+b\times c,\ b\times c+(a+a)\times c,\ b\times c+(2\times a)\times c \end{array} \right\}. \tag{3.1}$$

3.2.2 Outil de transformation automatique de programmes

Dans cette section, nous allons présenter l'outil `Salsa` (Travaux de thèse N. Damouche) [20]. `Salsa` est un outil de transformation automatique de programmes, il est utilisé pour améliorer la précision numérique des calculs flottants. `Salsa` prend en entrée un programme initialement écrit dans un langage impératif, auquel il va appliquer un ensemble de règles interprocédurale et intraprocédurale et retourne en sortie un programme écrit dans le même langage et numériquement plus précis. `Salsa` est basé sur l'analyse statique par interprétation abstraite ; il prend également en entrée les intervalles des variables d'entrées et retourne en sortie l'intervalle des erreurs d'arrondi (voir figure 3). La

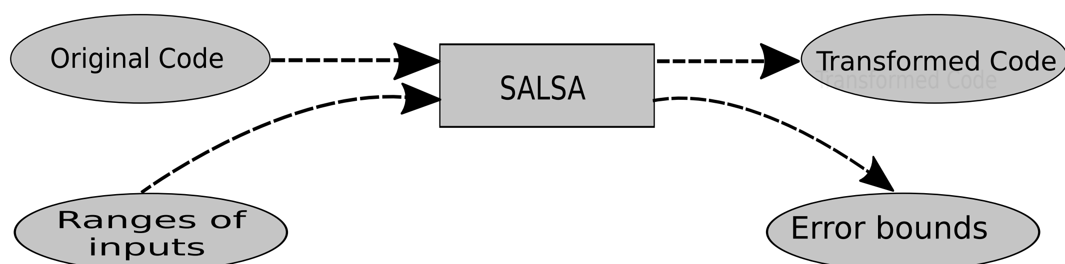

Figure 3 – Schéma récapitulatif des entrées et sorties de `Salsa`.

transformation automatique d'un programme par `Salsa` nécessite plusieurs étapes décrites ci-dessous :

— *La première étape appelée analyse syntaxique*, consiste à réécrire le programme en question sous forme `SSA`, abréviation de Static Single Assignment, d'une façon à ce que chaque variable est écrite une seule fois dans le programme pour éviter les conflits au moment de la lecture et de l'écriture d'une même variable. À la fin de cette étape un arbre syntaxique est généré.

— *La deuxième étape est une analyse statique par interprétation abstraite.* Elle est appliquée pour calculer les intervalles sur les erreurs d'arrondi en faisant appel à un ensemble de règles interprocédurale et intraprocédurale qui sont déjà implémentées dans `Salsa`.

— *La dernière étape est la construction de grosses expressions arithmétiques pour le regroupement de celles présentées dans le code et un appel à* ***`Sardana`*** qui à son tour va procéder à l'amélioration de la précision numérique de ces dernières expressions arithmétiques.

Afin d'illustrer ce fonctionnement, nous appliquons `Salsa` sur un code contenant une conditionnelle (voir figure 4) dans le but de l'amélioration de la précision numérique de ses calculs.

On note que, l'optimisation de la précision numérique d'un programme à l'aide de `Salsa` nécessite :

— Un environnement formel δ,

— Une liste notée β qui contient les variables que nous ne devons pas supprimer du programme, initialement cette liste contient la variable de référence notée ν qui devra être optimisée,

— Le contexte C qui contient la commande qui va être transformée par la suite.

On suppose que z est la variable qu'on veut optimiser, donc on a $\nu = z$. Au départ, l'environnement δ et le contexte C sont vides, et la liste β contient la variable z qu'on doit garder tout au long de l'exécution du programme. Une première étape de transformation consiste à appliquer une des règles intraprocédurales implémentées dans `Salsa` qui permet

$\delta = \emptyset$
$C = []$
$\beta = \{z\}$

```
 x = a + a + b ;
 y = b ;
if (x > 3.) then
  z = x + y ;
else
   z = (x - y)/2 ;
```

FIGURE 4 – Programme original.

de supprimer les affectations du programme source (la première et deuxième ligne du programme de la figure 4) et de sauvegarder dans la mémoire β sous certaines conditions.

$\delta' = \delta[x \mapsto a + a + b; y \mapsto b]$
$C = nop; nop; []$
$\beta = \{z\}$

```
if (x > 3.) then
  z = x + y  ;
else
  z = (x - y)/2  ;
```

FIGURE 5 – Premier programme intermédiaire.

Le premier programme intermédiaire (figure 5), est donné par la suppression de l'expression $a+a+b$ et de la variable b du programme original (figure 4) et de les mémoriser dans δ, par conséquent, les lignes correspondantes seront supprimées et remplacées par des nop, et le nouvel environnement δ va contenir $x = a + a + b$ et $y = b$.
Par la suite on applique des règles définies sur les séquences de commandes et on élimine également les nop du premier programme intermédiaire. L'étape d'après consiste à analyser le programme. Notons que la variable x n'est pas définie dans le premier programme intermédiaire, c'est pourquoi, elle sera à la fois réinsérée dans le programme mais aussi ajoutée à la liste β, comme le montre le nouveau programme intermédiaire de la figure 6.
Pour la transformation des commandes qui concernent les conditionnelles, on peut être confronté à deux cas. Si on connaît la valeur statique de la condition, dans le cas où la condition est vraie nous transformons la partie `then` et quand elle vaut faux, nous transformons la branche `else`. Par ailleurs, si on ne connaît pas statiquement la valeur de la

$\delta' = \delta[y \mapsto b]$
$C = nop; nop; []$
$\beta = \{x, z\}$

```
 x = a + a + b ;
if (x > 3.) then
  z = x + y ;
else
  z = (x - y)/2  ;
```

FIGURE 6 – Deuxième programme intermédiaire.

$\delta' = \emptyset$
$C = [];\ if(x > 3)\ then\ []\ else\ []$
$\beta = \{x, z\}$

```
 x = a + a + b ;
 y = b ;
if (x > 3.) then
  z = a + a + b + b ;
else
  z = (a + a + b - b )/2 ;
```

FIGURE 7 – Programme final avant transformation des expressions.

condition, nous transformons les deux branches de la conditionnelle. Dans notre exemple on évalue les deux branches `then` et `else` (voir figure 7 et figure 8).

3.3 Résultats expérimentaux

Dans cette section, nous présentons tout d'abord notre exemple d'application qui modélise `la propagation de la chaleur sur une grille`, suivi des résultats obtenus qui montrent l'impact qu'a l'outil `Salsa` sur l'amélioration de la précision numérique des calculs flottants dans le cas parallèle. Par la suite, nous allons montrer l'impact de cette amélioration de précision sur le temps d'exécution.

3.3.1 EDP de la propagation de chaleur

Dans le but d'améliorer la précision des calculs numériques nous avons utilisé l'outil `Salsa` [18] sur un code parallèle. Notre application modélise la propagation de la chaleur sur une grille chauffée en deux coins. Pour nos simulations, on considère une grille carrée

$\delta' = \emptyset$
$C = [];\ if(x > 3)\ then\ []\ else\ []$
$\beta = \{x, z\}$

```
  x = a + a + b ;
  y = b ;
 if (x > 3.) then
   z = 2(a + b) ;
 else
   z = a ;
```

FIGURE 8 – Programme Final après Transformation des Expressions.

d'une dimension 800×800, la grille est initialisée à la valeur 0.1 en tout point. De plus, elle est chauffée à deux angles (en haut à gauche, en bas à droite). Nos expérimentations consistent à faire des calculs précis sur la propagation de chaleur et ceci en un minimum de temps possible. On utilise un code parallèle dans lequel on divise notre grille en plusieurs bandes, tel que chaque bande sera traitée par un nœud de calcul dédié. À la fin, un nœud se charge de relier les résultats provenant de tous les autres nœuds. La parallélisation dans notre cas est effectuée en partageant le tableau de données représentant la grille entre les différents nœuds comme le montre la figure 9, le partage de données entre ces différentes nœuds de calculs se fait d'une manière équitable suivant le nombre de nœuds de calculs dont on dispose. Étant donné que chaque point a besoin des valeurs des points qui l'entourent, les valeurs aux bords des bandes doivent être communiquées entre les différents nœuds.

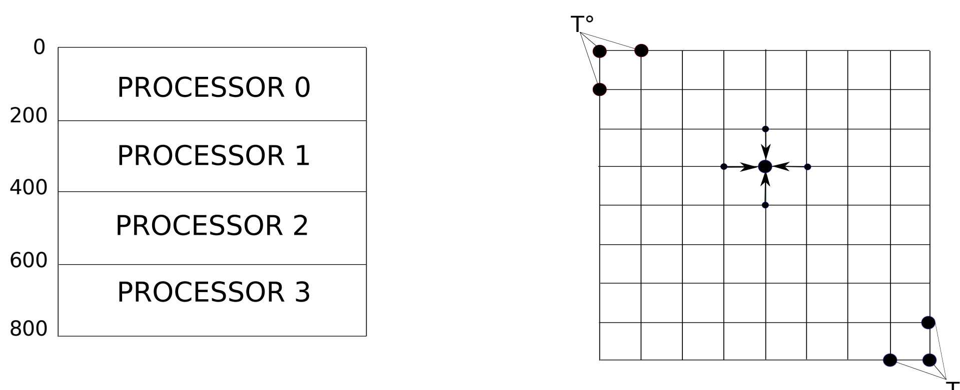

FIGURE 9 – Aperçu d'une grille 800×800 partitionnée suivant le nombre de processeurs dans ce cas 4 processeurs (à gauche). Exemple de points voisins nécessaires pour le calcul de la valeur de la chaleur en un point courant (à droite).

Indépendamment de la solution analytique, une solution numérique est apportée à ce

problème. Elle consiste en un calcul itératif, en parcourant tous les points de la grille et en faisant la moyenne des valeurs des points voisins. Plus précisément, la propagation de la chaleur en tout point sur la grille est calculée par la moyenne des points qui entourent le point courant, cette moyenne sera la nouvelle valeur de la chaleur en ce point. Si on prend l'exemple d'un point au centre de la grille T avec un stencil égal à 1, le calcul est montré par l'équation(3.2).

$$T_{new}[i][j] = \frac{T[i-1][j] + T[i+1][j] + T[i][j-1] + T[i][j+1]}{4}. \tag{3.2}$$

Notre étude se base sur plusieurs paramètres à savoir :

- L'augmentation du stencil qui est la distance entre un point et ses voisins, puis l'étude de l'impact de cette augmentation sur la propagation de la chaleur sur notre grille.
- La variation des valeurs de chauffe initiale aux coins de la grille en fonction des températures de fusion significatives des éléments chimiques, tels que le fer qui est égal 2800.4 $^{\circ}$F, le platine qui est égal à 3214.76 $^{\circ}$F, le cuivre qui est égal à 1984.316 $^{\circ}$F et le calcium qui est égal à 1547.6 $^{\circ}$F et d'étudier par la suite l'impact de cette variation sur la propagation de la chaleur sur la grille.
- Le changement du nombre d'itérations dans le but de savoir si la propagation de la valeur de chauffe initiale aux coins de la grille est influencée par le nombre d'itérations effectuées, en d'autres termes, on fixe une valeur de chaleur en un point de la grille (dans notre cas on a choisi le milieu) puis on compare le nombre d'itérations nécessaires pour atteindre cette valeur fixée à chaque fois qu'on change la valeur de chauffe initiale aux coins de la grille.

3.3.2 Précision numérique

Dans cette section, nous présentons nos résultats expérimentaux liés à la précision numérique des calculs. En un premier temps, on étudie l'impact de la propagation de chaleur après augmentation du stencil. Pour ce faire, considérons le programme pour la propagation de la chaleur développé avec MPI [67]. Les résultats de la simulation sont présentés à la figure 10. On remarque sur la figure des effets de lumière aux coins situés en haut à gauche et en bas à droite, qui correspondent aux coins de la grille chauffés initialement. Intuitivement, si on itère suffisamment (en augmentant le nombre d'itérations) la température initiale aux coins va se propager sur tous les points de la grille. Dans notre cas, pour une même température initiale et pour un même nombre d'itérations on remarque une propagation de la chaleur plus importante dans le cas du stencil égal à 2 voir figure 10 (à droite), ce qui montre que l'augmentation du stencil a un impact sur la propagation de la chaleur sur la grille. Deuxièmement, il est clair que l'objectif principal

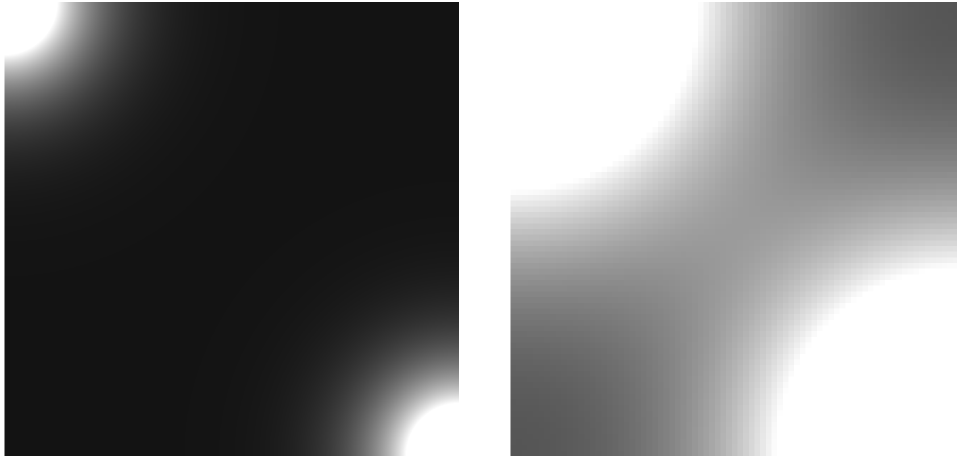

FIGURE 10 – Aperçu de la grille chauffée en deux coins, nombre d'itérations : 3600, stencil : 1 (figure gauche), stencil : 2 (figure droite).

du parallélisme est de traiter de longs codes de calcul en un minimum de temps possible. En tenant compte des problèmes de précision qui se posent à la fin de ces longs calculs, on souhaite mesurer les performances de `Salsa` sur l'amélioration de la précision numérique de ces calculs qui modélisent la propagation de la chaleur sur une grille, ce qui nous permettra par la suite d'établir une comparaison entre les résultats du programme initial et celui transformé. Notre approche se base sur le calcul des erreurs relatives et absolues entre le programme initial et celui transformé afin de mesurer les performances de `Salsa` sur l'amélioration de la précision numérique des calculs effectués. Comme vu précédemment, on chauffe à différents endroits de la grille (en haut à gauche, en bas à droite) voir figure 11, ce qui fait que la distribution de la chaleur est différente sur les points de la grille. Notre contribution consiste à spécialiser le code de chaque processeur en fonction de ses données d'entrée, plus précisément, l'ordre de calcul en chaque processeur dépend de l'ordre de grandeur de ses données. Dans ce même contexte, nous utilisons `Salsa` pour transformer le code de chaque processeur en fonction de ses données d'entrée. Dans le cas du `Processeur 0`, l'équation(3.2) devient l'équation(3.3).

$$T_{new}[i][j] = \frac{(((T[i][j+1] + T[i+1][j]) + T[i-1][j]) + T[i][j-1])}{4}. \qquad (3.3)$$

Le nouveau parenthésage résulte du fait qu'on chauffe au coin haut gauche de la grille. Pour mieux illustrer ceci, prenons l'exemple d'un point au milieu de la grille avec les coordonnées (i, j), les deux points avec les coordonnées $(i+1, j)$ et $(j-1, i)$ auront deux grandes valeurs de la chaleur supérieures à celles des deux autres points avec les coordonnées $(i, j+1)$ et $(i+1, j)$. Nous rappelons que nos calculs se basent sur l'arithmétique flottante, donc il est préférable de sommer d'abord les petites valeurs entre elles ensuite les grandes

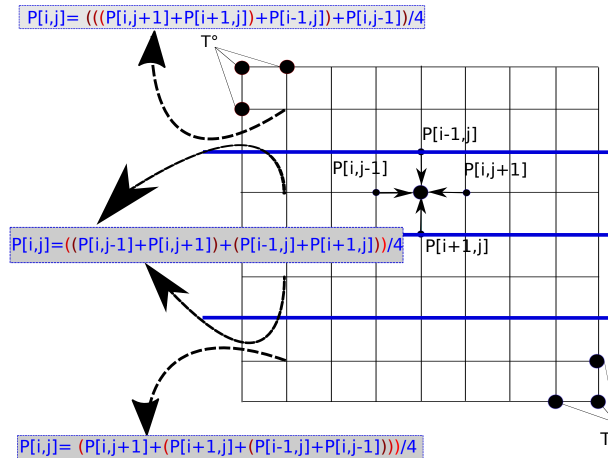

FIGURE 11 – Processus de spécialisation des calculs en fonction des données d'entrée.

pour éviter ce que l'on appelle une absorption définit dans le chapitre 1. À l'opposé, si on prend le `Processeur 3` où on chauffe au coin en bas à droite, et pour le même point avec les coordonnées (i, j) les deux points qui auront les plus grandes valeurs de la chaleur seront $(i, j+1)$ et $(i+1, j)$, donc l'équation(3.2) aura la forme de l'équation(3.4).

$$T_{new}[i][j] = \frac{(T[i][j+1] + (T[i+1][j] + (T[i][j-1] + T[i-1][j])))}{4}. \tag{3.4}$$

La figure 12, représente les erreurs absolues dues au calcul de la propagation de la chaleur sur une grille de dimension 800×800, chauffée aux coins en haut à gauche et en bas à droite. Les coins sont à une température qui est égale à celle de la fusion du platine. Pour ce calcul d'erreurs, on convient que les résultats obtenus par le programme transformé par `salsa` sont des résultats exacts et ceux obtenus par le programme original sont des résultats approchés. Nous remarquons sur la figure que les erreurs absolues sont négligeables au milieu de la grille, puisque nous n'avons pas réitéré assez sur les calculs de la propagation de chaleur, pour que les erreurs atteignent tous les points de la grille. On note également que les erreurs absolues autour des coins correspondant aux points chauffés apparaissent petites au début pour un nombre d'itérations égale à 1800 voir la figure 12 (en haut à gauche). Ces erreurs seront accumulées et propagées en augmentant le nombre d'itérations tel qu'il est présenté dans la figure 12 (3600 itérations en haut à droite, 7200 itérations en bas à gauche, 10800 itérations en bas à droite). De plus, quand nous mesurons le temps d'exécution correspondant à chaque cas (suivant le nombre d'itérations) précédemment présenté dans la figure 12, nous notons que l'exécution est réalisée en seulement 0.52 secondes pour un nombre d'itérations égales à 1800 et en 1.14 secondes pour atteindre 10800 itérations.

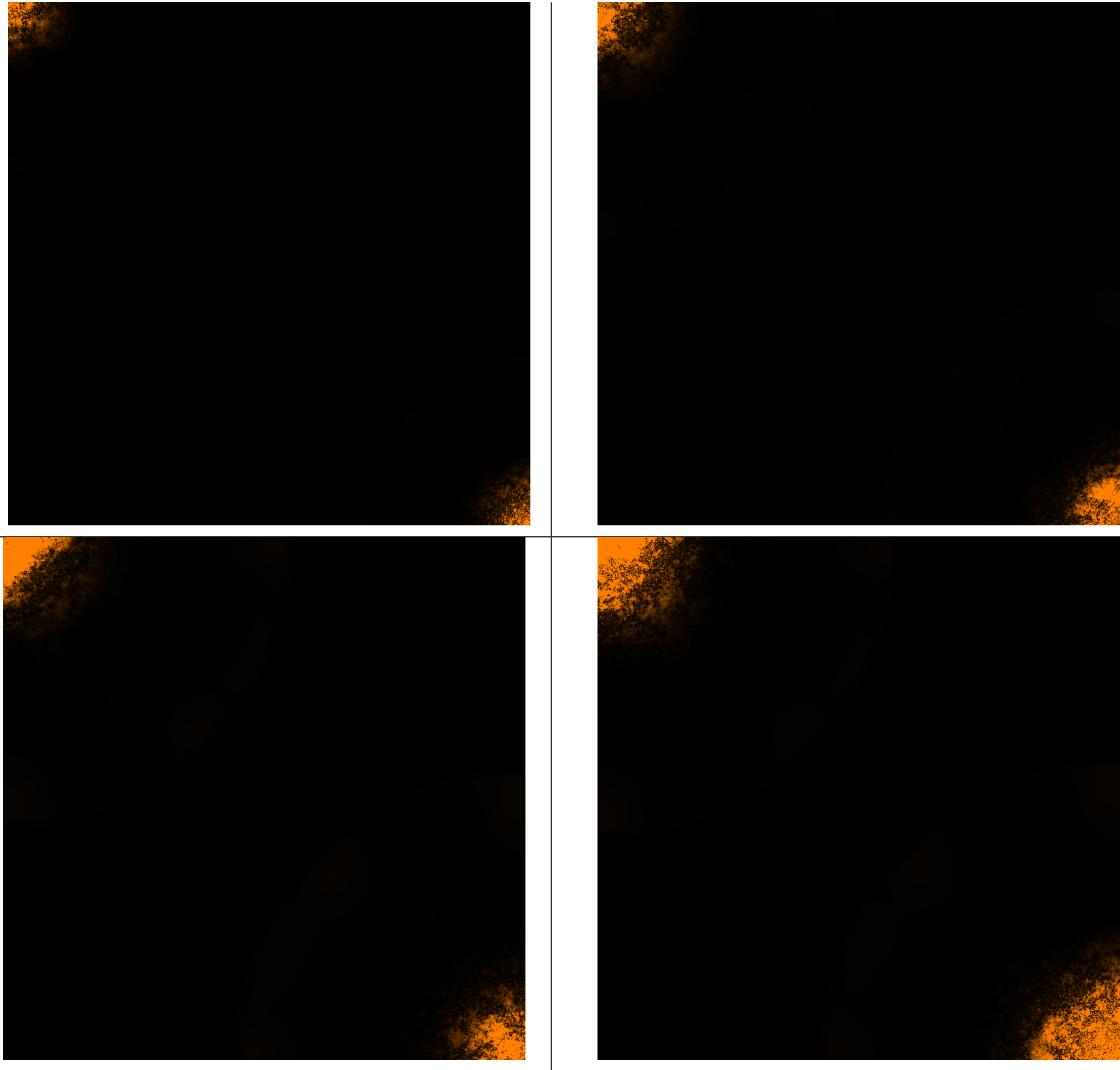

FIGURE 12 – Erreurs absolues entre le programme original et celui optimisé, température initiale : 10106.6 ˚F, nombre d'itérations : 1800 (en haut à gauche), 3600 (en haut à droite), 7200 (en bas à gauche), 10800 (en bas à droite).

Une autre expérimentation est consacrée aux calculs d'erreurs relatives et absolues (voir figure 13) des mêmes exemples précédemment introduits. C'est-à-dire, pour différentes températures initiales et en variant le nombre d'itérations de 1800 à 10800. D'après la figure 13. nous remarquons la présence d'erreurs autour des coins chauffés. Ces erreurs dépendent du nombre d'itérations aussi bien que la valeur de chauffe initiale des deux coins.

3.3.3 Temps d'exécution

Dans cette section, nous allons étendre le concept d'accélération de convergence [21] des programmes séquentiels aux programmes parallèles observé expérimentalement avec `Salsa`. Nous voulons montrer à travers cette expérimentation qu'en optimisant les programmes pour qu'ils soient plus précis, nous accélérons leur vitesse de convergence, c'est-

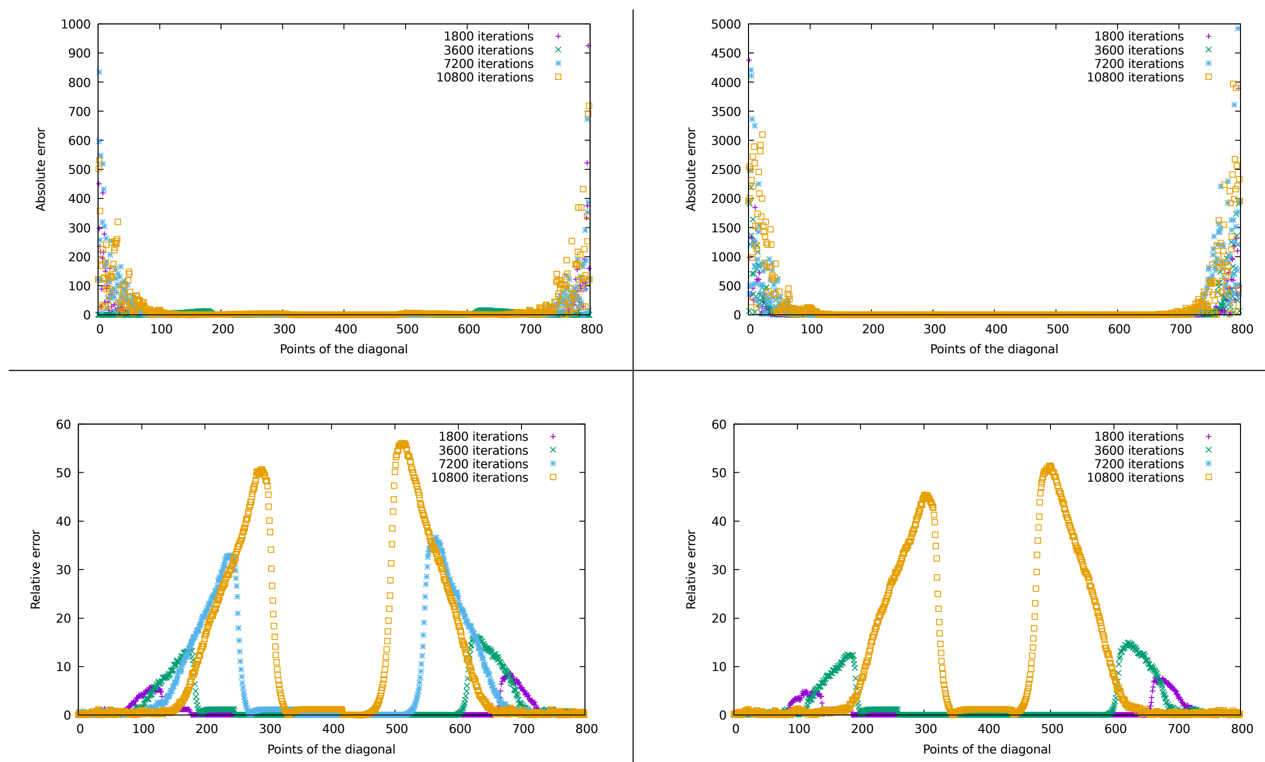

FIGURE 13 – Les erreurs absolues et relatives respectivement entre le programme original et celui transformé, température initiale : 1547.6 ° F (en haut à gauche, en bas à gauche), pour des nombres d'itérations différents, température initiale : 10106.6 ° F (en haut à droite, en bas à droite).

à-dire nous réduisons leur nombre d'itérations nécessaires à cet effet. En pratique, nous avons appliqué cette étude sur le programme de propagation de chaleur précédemment décrit. Les résultats obtenus en transformant ce programme avec `Salsa` montrent une amélioration de la précision numérique et du temps d'exécution du programme, de plus, la vitesse de convergence est accélérée ; plus précisément, le nombre d'itérations nécessaires pour converger à une valeur donnée est réduit. Le principe est le suivant. Nous fixons une valeur de chaleur souhaitée à atteindre au centre de la grille et en variant la valeur de chauffe initiale aux coins et à chaque fois on récupère le nombre d'itérations effectuées pour atteindre la valeur fixée. La figure 3.1 récapitule les différents résultats en termes de nombre d'itérations et en temps d'exécution tout en variant la température initiale.

Si nous prenons l'exemple d'une température égale à 1010.6 ° F, et nous souhaitons calculer le nombre d'itérations nécessaires pour atteindre la valeur de 357 ° F au milieu de la grille. Nos résultats montrent que nous réduisons le nombre d'itérations de 33 itérations dans le code transformé comparé au code initial. On remarque également que nous n'améliorons pas le temps d'exécution d'une façon remarquable mais notre objectif majeur est d'avoir des résultats précis sans être plus lent.

Température initiale	Élément chimique	Nbr d'itérations avant transfo.	Nbr d'itération après transfo.	Temps d'exec. avant transfo.	Temps d'exec. après transfo.
1010.6 ˚ F	Tellure	10945	10912	1.110s	1.080s
1547.6 ˚ F	Calcium	6712	6693	0.916s	0.901s
1984.316 ˚ F	Cuivre	5426	5411	0.876s	0.867s
2800.4 ˚ F	Fer	3652	3643	0.797s	0.814s
3214.6 ˚ F	Platine	3202	3190	0.789s	0.778s
10106.6 ˚ F	Carbone	1303	1290	0.716s	0.705s

Table 3.1 – Estimations du nombre d'itérations et du temps d'exécution pour le programme de la propagation de chaleur.

3.4 Conclusion

Nous avons abordé dans ce chapitre l'amélioration automatique de la précision, via la transformation complète d'un programme. Notre objectif a été d'appliquer une technique d'amélioration de la précision numérique sur un code parallèle développé en MPI, plus précisément, un programme modélisant la propagation de chaleur sur une grille.

L'approche suivie au cours de ce travail consiste en une spécialisation du code de chaque processeur avant d'appliquer les transformations automatiques à l'aide de l'outil appelé `Salsa`. Cet outil prend en entrée le code de chaque processeur, lui applique un certain nombre de règles pour retourner en sortie un code mathématiquement équivalent mais plus précis. Nous avons testé les performances de notre approche en calculant à la fois les erreurs absolues et relatives entre le programme original et transformé par `Salsa`, on a également étudié l'impact de l'optimisation de la précision numérique des résultats sur la convergence et sur le temps d'exécution. Dans ce même contexte, une perspective intéressante est d'étendre les techniques de transformation qui ont prouvé leur efficacité dans le cas séquentiel sur les programmes parallèles correspondant dans le but de l'amélioration de la convergence des méthodes itératives [21, 7], ce que nous allons voir au chapitre 6.

Dans le chapitre suivant, nous allons proposer une nouvelle technique d'amélioration de la précision numérique basée sur l'analyse statique. L'idée générale consiste à mettre en place un placement de tâches, en d'autres termes on va chercher la meilleure configuration de la distribution du calcul sur l'ensemble des processeurs de façon à être à la fois précis et rapide [6].

Chapitre 4

Technique de placement de tâches

Contents

4.1 Introduction **38**

4.2 Résolution de systèmes linéaires **39**

4.3 Notions utilisées **40**

4.4 Domaines abstraits **41**

4.4.1 Exp^n : Ordre de grandeur des éléments de la matrice 42

4.4.2 $\mathsf{Exp}^{\#n}$: Abstraction des exposants en scalaires 44

4.4.3 Abstraction en gradients 45

4.5 Partitionnement des données **48**

4.5.1 Algorithme glouton 48

4.5.2 Exemple 50

4.6 Expérimentations **52**

4.6.1 Poutre en flexion 52

4.6.2 Résultats expérimentaux 53

4.7 Conclusion **55**

4.1 Introduction

Afin d'améliorer la précision numérique des calculs flottants, nous introduisons dans ce chapitre une nouvelle technique basée sur l'analyse statique par interprétation abstraite. Notre objectif principal est de détecter le meilleur ordonnancement qui conduit à une optimisation de la précision numérique du calcul. Il est bien connu que les causes de pertes de précision sont multiples. Citons le cas d'un calcul impliquant des valeurs scalaires de différents ordres de grandeurs. Plus en détails, pour que le calcul soit précis, il est important de savoir si les valeurs peuvent être ordonnées par rapport à leur ordre de grandeur, en commençant les sommations avec les plus petits valeurs. Pour ce faire, nous proposons de nous appuyer sur l'analyse statique pour détecter de tels arrangements de coefficients matriciels : détection de l'ordre de grandeur de chaque scalaire impliqué dans le calcul et l'ordonnancement (croissant, décroissant, équilibré) des séquences de celui-ci. Une fois cet ordre calculé avec précision, nous pouvons choisir un algorithme de

sommation approprié (gauche à droite, droite à gauche, équilibré), par conséquent, obtenir des résultats en virgule flottante plus précis.

Concernant le cas des programmes parallèles [81], nous visons à spécialiser le code de chaque processeur en fonction de ses données, au lieu de se concentrer uniquement sur l'équilibrage de charge entre les noeuds de calcul, ce qui est fait habituellement. Cette spécialisation est basée sur le placement de tâches [32, 45]. L'idée est d'affecter à chaque processeur des ensembles de données qui peuvent être additionnés avec précision. Notre contribution est donc de s'appuyer sur une méthode d'analyse statique pour construire ces partitions avant de répartir les données entre les processeurs. Nous avons appliqué cette approche à une méthode itérative pour la résolution des systèmes linéaires de la forme $Ax = b$. Comme exemple introductif, nous avons utilisé le schéma itératif représentant la méthode de Jacobi.

Dans ce chapitre, nous décrivons les différentes phases de notre technique. Nous nous appuyons d'abord sur l'interprétation abstraite pour représenter les propriétés des séquences de valeurs $s = x_1, x_2, \dots x_n$. Les propriétés calculées, appelées *gradients*, indiquent comment les éléments de s sont ordonnés. Soit $grad(s) \in \{\nearrow, \searrow, \rightarrow\}$ une telle propriété, désignant respectivement des séquences de valeurs croissantes, décroissantes ou équilibrées. Par séquences équilibrées, nous entendons des valeurs de même ordre de grandeur (même exposant en base 10.) plutôt que des séquences constantes. En considérant les plus grandes séquences avec la même propriété de gradient, nous pouvons construire un partitionnement grâce à un algorithme glouton.

Nous présentons brièvement la méthode de Jacobi à la section 4.2. La section 4.4 présente notre contribution : nous détaillons notre technique avec ses différentes étapes. La section 4.5 décrit l'algorithme glouton utilisé pour construire nos partitions selon les informations du gradient. La section 4.6 détaille notre exemple. Enfin, avant de conclure, nous présentons quelques résultats expérimentaux à la section 4.7.

4.2 Résolution de systèmes linéaires

La méthode de Jacobi est une méthode numérique bien connue, elle est utilisée pour résoudre des systèmes linéaires de n équations et n inconnues. nous l'avons choisie pour sa simplicité, et comme premier algorithme sur lequel nous appliquons notre méthodologie. Dans cette méthode, une estimation initiale aussi appelée solution approximative x^0, est sélectionnée et est mise à jour de manière itérative jusqu'à trouver la solution exacte x.

Afin d'expliquer l'idée de l'algorithme, considérons le système linéaire suivant de n équations $Ax = b$, où :

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \text{ and } \quad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

Le calcul de la solution à chaque itération est donné par l'équation (5.2) ci-dessous :

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1,\ i \neq j}^{n} a_{ij} x_j^k \right). \qquad i = 1, \ldots, n, \quad a_{ii} \neq 0. \tag{4.1}$$

Notez que la méthode de Jacobi est stable lorsque la matrice A est strictement à diagonale dominante (voir équation 4.2), c'est-à-dire sur chaque ligne, la valeur absolue du terme diagonal est supérieur à la somme des valeurs absolues des autres termes :

$$\forall i \in 1, \ldots, n, \qquad |a_{ii}| > \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|. \tag{4.2}$$

Notre choix de cette méthode est lié au fait qu'elle utilise un opérateur de sommation, c'est-à-dire qu'une somme est effectuée à chaque itération de l'algorithme. En raison de l'utilisation des nombres à virgule flottante, cette somme peut être incorrecte à cause des erreurs accumulées. Par exemple, il est bien connu que si nous additionnons les petites valeurs avec les grandes, une absorption peut se produire et, par conséquent, une éventuelle perte de précision. Afin de résoudre ce problème, nous préconisons d'additionner les valeurs en commençant par les plus petites. Il est à noter que de nombreux travaux liés à la précision numérique existent dans la bibliographie par transformation de code (voir section 3.2).

4.3 Notions utilisées

Dans cette section, nous commençons d'abord par rappeler quelques définitions standard afin d'introduire les notions de l'interprétation abstraite [18, 19, 42] qui nous seront utiles pour la suite.

Définition 4.1 *(Ensemble partiellement ordonné). Soit E un ensemble muni d'une relation d'ordre $\sqsubseteq$. L'ensemble $(E, \sqsubseteq)$ est dit partiellement ordonné si :*

- $\forall x \in E, x \sqsubseteq x$ *(réflexive),*
- $\forall (x, y) \in E^2, x \sqsubseteq y \land y \sqsubseteq x \Longrightarrow x = y$ *(antisymétrique),*
- $\forall (x, y, z) \in E^3, x \sqsubseteq y \land y \sqsubseteq z \Longrightarrow x \sqsubseteq z$ *(transitive).*

Définition 4.2 *(Bornes supérieure et inférieure). Soit un ensemble partiellement ordonné* $(E, \sqsubseteq)$ *et* $S \subseteq E$*. Un élément de l'ensemble* E *noté par* $\sqcup S$ *est dit borne supérieure de* S *si* $\forall s \in S, s \sqsubseteq \sqcup S$*. L'élément* $\sqcup S$ *est le plus petit majorant de* S*, tel que* $\forall y \in E, (\forall s \in S, s \sqsubseteq y) \Longrightarrow \sqcup S \sqsubseteq y$*. De la même façon, nous définissons la borne inférieure de* S *par* $\sqcap S$ *si* $\forall s \in S, \sqcap S \sqsubseteq s$*. L'élément* $\sqcap S$ *est le plus grand minorant de* S *tel que* $\sqcap S = \sqcup\{y | \forall x \in S, y \sqsubseteq x\}$

Définition 4.3 *(Treillis complet). Un treillis complet est un ensemble* E *muni d'un ordre partiel* $\sqsubseteq$ *(relation réflexive, antisymétrique et transitive) et d'une borne supérieur* $\sqcup$ *pour toute partie* S *de* E*.*
Un treillis complet est muni d'un plus petit élément $\bot = \sqcup\emptyset = \sqcap S$ *et d'un plus grand élément* $\top = \sqcup S = \sqcap\emptyset$*.*

Exemple 4.1 *Soit l'ensemble* $\mathbb{R}$ *et l'ensemble de ses parties* $\wp(\mathbb{R})$ *muni de l'ordre inclusion* $\subseteq$ *et de la borne supérieure union* $\cup$ *est un treillis complet. La borne inférieure de ce treillis est l'intersection* $\cap$*, son plus petit élément est représenté par l'ensemble vide* $\emptyset$ *et son plus grand élément est la totalité de l'ensemble* $(\mathbb{R})$*.*

Définition 4.4 *(Fonction de concrétisation). Soit* $(S, \sqsubseteq, \sqcup)$ *le treillis complet représentant les états concrets (possibles) d'un programme et le treillis complet* $(S^{\#}, \sqsubseteq^{\#}, \sqcup^{\#})$ *des états abstraits. Le lien entre le domaine concret* $(S, \sqsubseteq)$ *et le domaine abstrait* $(S^{\#}, \sqsubseteq^{\#})$ *est donné par la fonction dite de concrétisation :* $\gamma : S^{\#} \longrightarrow S$*. Ainsi, l'état concret* $s \in S$ *est dit abstrait par* $s^{\#} \in S^{\#}$ *lorsque* $s \sqsubseteq \gamma(s^{\#})$*.*

Définition 4.5 *(Correspondance de Galois). Soient* $(S, \sqsubseteq, \sqcup)$ *et* $(S^{\#}, \sqsubseteq^{\#}, \sqcup^{\#})$ *deux treillis complets. La paire de fonctions* $\alpha : S \longrightarrow S^{\#}$ *et* $\gamma : S^{\#} \longrightarrow S$ *forme une correspondance de Galois si :*

- $\alpha : S \longrightarrow S^{\#}$ *est monotone,*
- $\gamma : S^{\#} \longrightarrow S$ *est monotone,*
- $\forall s \in S$*,* $s : \gamma \circ \alpha(s) \sqsupseteq s$ *est extensive,*
- $\forall s^{\#} \in S^{\#}$*,* $s^{\#} : \alpha \circ \gamma(s^{\#}) \sqsubseteq^{\#} s$ *est réductive.*

4.4 Domaines abstraits

Dans cette section, nous présentons la première étape de notre contribution qui consiste à détecter les séquences de valeurs croissantes, décroissantes ou équilibrées dans des vecteurs. En se concentrant sur les vecteurs et leurs séquences croissantes, décroissantes ou équilibrées associées, nous introduisons des domaines abstraits pour représenter et calculer des propriétés sur ces ensembles de vecteurs. Soit $\mathbb{Z}^n$ l'ensemble des vecteurs de taille

$$\langle \wp(\mathbb{R}^n), \subseteq \rangle \underset{\alpha^n_{\mathsf{Exp}}}{\overset{\gamma^n_{\mathsf{Exp}}}{\leftrightarrows}} \langle \mathsf{Exp}^n, \subseteq \rangle \underset{\alpha^n_D}{\overset{\gamma^n_D}{\leftrightarrows}} \langle \mathsf{Exp}^{\#n}_D, \sqsubseteq^n_D \rangle$$

$$\alpha_S \downarrow\uparrow \gamma_S \qquad\qquad \alpha_G \downarrow\uparrow \gamma_G$$

$$\langle \mathtt{Sign}, \sqsubseteq \rangle \quad \times \quad \langle \mathsf{Grad}, \sqsubseteq \rangle$$

FIGURE 14 – Schéma général des abstractions.

n et $\wp(\mathbb{Z}^n)$ l'ensemble des parties de $\mathbb{Z}^n$. L'ensemble $\wp(\mathbb{Z}^n)$ est muni d'un ordre partiel, l'inclusion. C'est un treillis complet.

Grâce aux techniques d'interprétation abstraite [18, 19], nous pouvons définir différentes abstractions de ce treillis, et les combiner pour calculer les propriétés sur les ensembles de n-vecteurs. Dans ce contexte, nous présentons les différentes correspondances de Galois que nous effectuons pour obtenir les gradients. Un aperçu de notre séquence de correspondances est donné à la figure 14. L'ensemble concret (donc le plus précis) partiellement ordonné est l'ensemble des n-vecteurs $\langle \wp(\mathbb{R}^n), \subseteq \rangle$. Une première abstraction est appelée l'abstraction *Signe*, elle s'appuie sur le domaine des signes classique pour détecter si tous les éléments du vecteur ont le même signe. Par conséquent, un élément abstrait positif dénote l'ensemble des n-vecteurs avec seulement des valeurs positives. L'autre abstraction principale est l'abstraction *Grad*(ient), représentant la nature croissante, décroissante ou équilibrée des scalaires des vecteurs représentés. À titre d'exemple, un élément abstrait croissant désigne l'ensemble des n-vecteurs avec des scalaires croissants. Ce domaine abstrait de gradients est défini grâce à l'introduction de domaines abstraits plus simples. Tout d'abord, les ensembles de n-vecteurs sont abstraits sous forme de n-vecteurs d'ordre de grandeur, en associant à chaque scalaire son exposant à virgule flottante. Puis ces vecteurs d'ordre de grandeur sont abstraits encore une fois pour produire l'abstraction en gradient. Les ensembles partiellement ordonnés Exp^n, $\mathsf{Exp}^{\#n}_D$, Grad sont définis en détail dans les sections 4.4.1, 4.4.2, 4.4.3, respectivement.

4.4.1 Exp^n : Ordre de grandeur des éléments de la matrice

Les nombres à virgule flottante n'ont pas besoin d'être exactement ordonnés pour éviter l'absorption. Nous avons seulement besoin d'avoir des valeurs similaires ou de *même ordre de grandeur*. Un premier domaine abstrait représente un ensemble de n-vecteurs $\wp(\mathbb{R}^n)$ par un vecteur d'un ensemble d'exposants $\mathsf{Exp}^n = (\wp(\mathbb{Z}))^n$. Les exposants sont définis comme des entiers signés où l'entier n correspond à l'exposant du nombre réel qu'il représente, c'est-à-dire $\lfloor \log_{10}(x) \rfloor$ la valeur entière de $\log_{10}(x)$ arrondie vers $-\infty$. Forma-

lisons d'abord cette abstraction : le treillis complet $\langle \wp(\mathbb{R}), \subseteq, \cup, \cap, \emptyset, \mathbb{R} \rangle$ est abstrait par le treillis complet $\langle \wp(\mathbb{Z}), \subseteq, \cup, \cap, \emptyset, \mathbb{Z} \rangle$. Soit $(\alpha_{\mathsf{Exp}}, \gamma_{\mathsf{Exp}})$ la paire de fonctions d'abstraction et de concrétisation définies par :

$$\begin{cases} \alpha_{\mathsf{Exp}} : \wp(\mathbb{R}) & \rightarrow & \wp(\mathbb{Z}) \\ X & \mapsto & \{\lfloor \log_{10}(x) \rfloor : x \in X)\} \\ \gamma_{\mathsf{Exp}} : \wp(\mathbb{Z}) & \rightarrow & \wp(\mathbb{R}) \\ Y & \mapsto & \{x \in \mathbb{R} | \lfloor \log_{10}(x) \rfloor \in Y)\} \end{cases} \tag{4.3}$$

Propriété 1 (correspondance de Galois Exp) *La paire $(\alpha_{\mathsf{Exp}}, \gamma_{\mathsf{Exp}})$ est une correspondance de Galois.*

$$\langle \wp(\mathbb{R}), \subseteq \rangle \underset{\alpha_{\mathsf{Exp}}}{\overset{\gamma_{\mathsf{Exp}}}{\leftrightarrows}} \langle Exp, \subseteq \rangle$$

Preuve. Les deux domaines sont des treillis complets et les fonctions α_{Exp} et γ_{Exp} sont définies élément par élément : elles sont monotones pour l'inclusion.

$\alpha_{\mathsf{Exp}} \circ \gamma_{\mathsf{Exp}}(Y) = Y$ est réductive alors que $\gamma_{\mathsf{Exp}} \circ \alpha_{\mathsf{Exp}}(X) = \{x \in \mathbb{R} | \exists x' \in X, \log_{10}(x) = \log_{10}(x')\} \supseteq X$ est extensive.

□

La fonction d'abstraction représente un ensemble de valeurs par ordre de grandeur obtenu avec une fonction $\log_{10}$. La fonction de concrétisation est l'opération associée pour obtenir une correspondance de Galois. Par exemple, $\alpha_{\mathsf{Exp}}(\{1.10^4, 2.10^4, 3.10^4\}) = \{4\}$ puisque toutes ces valeurs partagent le même exposant 4 [1].

Nous pouvons maintenant élargir cette abstraction aux ensembles de n-vecteurs dans $\wp(\mathbb{R}^n)$. Le treillis $\langle \wp(\mathbb{R}^n), \subseteq, \cup, \cap, \emptyset, \mathbb{R} \rangle$ est abstrait par le treillis $\mathsf{Exp}^n = \langle (\wp\mathbb{Z})^n, \subseteq^n, \cup^n, \cap^n, \emptyset^n, \mathbb{Z}^n \rangle$ où $\subseteq^n$, $\cup^n$, et $\cap^n$ dénotent les opérateurs ensemblistes classiques pour n-vecteurs. Par exemple. $\forall x, y \in (\wp\mathbb{Z})^n, x \subseteq^n y$ tq $\forall i \in [1, n], x_i \subseteq y_i$. De la même manière, $\forall x, y \in (\wp\mathbb{Z})^n, \exists z \in (\wp\mathbb{Z})^n$, alors $z = x \cup^n y$ et $\forall i \in [1, n], z_i = x_i \cup y_i$.

Introduisons les deux fonctions $(\alpha^n_{\mathsf{Exp}}, \gamma^n_{\mathsf{Exp}})$:

$$\begin{cases} \alpha^n_{\mathsf{Exp}} : \wp(\mathbb{R}^n) & \rightarrow & (\wp\mathbb{Z})^n \\ X & \mapsto & z \in (\wp\mathbb{Z})^n \text{ s.t. } \forall i \in [1, n], z_i = \alpha_{\mathsf{Exp}}(\{x_i | x \in X\}) \\ \gamma^n_{\mathsf{Exp}} : (\wp\mathbb{Z})^n & \rightarrow & \wp(\mathbb{R}^n) \\ z & \mapsto & \{x \in \mathbb{R}^n | \forall i \in [1, n], x_i \in \gamma_{\mathsf{Exp}}(z_i)\} \end{cases} \tag{4.4}$$

Théorème 1 (correspondance de Galois Exp^n) *Les deux fonctions $(\alpha^n_{\mathsf{Exp}}, \gamma^n_{\mathsf{Exp}})$ forment une correspondance de Galois.*

$$\langle \wp(\mathbb{R}^n), \subseteq \rangle \underset{\alpha^n_{\mathsf{Exp}}}{\overset{\gamma^n_{\mathsf{Exp}}}{\leftrightarrows}} \langle Exp^n, \subseteq^n \rangle$$

1. En pratique, puisque les valeurs sont représentées par des sommes de puissances de 2, on aurait pu utiliser le log binaire log 2.

PREUVE. Les deux domaines $\wp(\mathbb{R}^n)$ et $\wp(\mathbb{Z})^n$ sont des treillis complets et les fonctions α^n_{Exp} et γ^n_{Exp} sont définies élément par élément : elles sont monotones pour l'inclusion.

$\alpha^n_{\mathsf{Exp}} \circ \gamma^n_{\mathsf{Exp}}(z) = z$ est réductive et $\gamma^n_{\mathsf{Exp}} \circ \alpha^n_{\mathsf{Exp}}(X) = \{x \in \mathbb{R}^n | \exists x' \in X \quad \forall i \in [1,n], \alpha_{\mathsf{Exp}}(x'_i) = \alpha_{\mathsf{Exp}}(x_i)\} \supseteq X$ est extensive.

□

L'exemple ci-dessous représente l'abstraction d'un ensemble de vecteurs $\wp(\mathbb{R}^n)$ par un vecteur d'ensembles d'exposants $\wp(\mathbb{Z})^n$ en utilisant la fonction abstraite α^n_{Exp}.

$$\left\{\begin{pmatrix}1000\\100\\10\\1\end{pmatrix}, \begin{pmatrix}0.001\\1\\0.1\\8\end{pmatrix}\right\} \overset{\alpha^n_{\mathsf{Exp}}}{\rightarrow} \begin{pmatrix}\{-3,3\}\\\{0,2\}\\\{-1,1\}\\\{1\}\end{pmatrix} \in \wp(\mathbb{Z})^n$$

4.4.2 $\mathsf{Exp}^{\#n}$: Abstraction des exposants en scalaires

Dans cette partie, nous allons encore faire abstraction de ces vecteurs d'ensemble d'exposants $\mathsf{Exp}^n = (\wp\mathbb{Z})^n$ qui résultent de la première abstraction. Puisque chaque élément de ces vecteurs est un ensemble d'entiers, on peut s'appuyer sur l'état de l'art des domaines abstraits pour représenter ces ensembles. Par exemple reprenons le vecteur d'ensembles $v = \begin{pmatrix}\{-3,3\}\\\{0,2\}\\\{-1,1\}\\\{1\}\end{pmatrix}$ de la section 4.4.1. Nous allons l'abstraire en $v^{\#}$ un vecteur abstrait. Ensuite, nous devons faire abstraction des ensembles $\{-3,3\}, \{0,2\}, \{-1,1\}$ et $\{1\}$. Pour cela, nous allons utiliser le domaine des constantes de Kildall ou encore le domaine des intervalles.

Soit $\langle D, \sqsubseteq_D\rangle$ une abstraction de $\langle \wp(\mathbb{Z}), \subseteq\rangle$ associée à une correspondance de Galois (α_D, γ_D). On définit l'ensemble $\mathsf{Exp}^{\#n}_D = D^n$ comme n-vecteurs de D éléments. Par exemple, $\forall x, y \in \mathsf{Exp}^{\#n}_D, x \sqsubseteq^n_D y$ ssi $\forall i \in [1,n], x_i \sqsubseteq_D y_i$.

Nous introduisons l'abstraction suivante :

$$\langle \mathsf{Exp}^n, \subseteq^n\rangle \underset{\alpha^n_D}{\overset{\gamma^n_D}{\leftrightarrows}} \langle \mathsf{Exp}^{\#n}_D, \sqsubseteq^n_D\rangle \tag{4.5}$$

où $\alpha^n_D(x) = z$ avec $\forall i \in [1,n], z_i = \alpha_D(x)$. De la même façon, $\gamma^n_D(z) = x \in \mathsf{Exp}^n$ avec $\forall i \in [1,n], x_i \subseteq \gamma_D(z_i)$.

Instanciation. Nous proposons d'instancier cette abstraction par deux domaines abstraits de base :

1. $\langle K, \sqsubseteq_K\rangle$ le domaine des constantes de Kildall [55] avec $K = \mathbb{Z} \cup \{\bot, \top\}$
2. $\langle I, \sqsubseteq_I\rangle$ le domaine des intervalles avec $I = ((\mathbb{Z} \cup \{-\infty\}) \times (\mathbb{Z} \cup \{+\infty\})) \cup \{\bot\}$

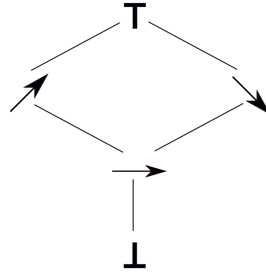

FIGURE 15 – Treillis des gradients.

Appliqué à l'exemple ci-dessus, nous obtiendrons :

$$\begin{pmatrix}\{-3,3\}\\\{0,2\}\\\{-1,1\}\\\{1\}\end{pmatrix} \overset{\alpha^n_{\mathsf{K}}}{\rightarrow} \begin{pmatrix}\top\\\top\\\top\\1\end{pmatrix} \qquad \begin{pmatrix}\{-3,3\}\\\{0,2\}\\\{-1,1\}\\\{1\}\end{pmatrix} \overset{\alpha^n_{\mathsf{I}}}{\rightarrow} \begin{pmatrix}[-3,3]\\ [0,2]\\ [-1,1]\\ [1,1]\end{pmatrix}$$

4.4.3 Abstraction en gradients

Pour abstraire le gradient à partir des données, nous définissons une nouvelle relation de comparaison selon l'ordre de grandeur des données noté $\leq^\#$, tel que l'on associe à un ensemble de valeurs $(x_1 \cdot x_2 \cdot \ldots \cdot x_n)$ l'une des cinq valeurs suivantes $\bot, \nearrow, \searrow, \rightarrow$ et $\top$. Nous devons être capables de calculer un *gradient* abstrait dénotant la caractéristique monotone d'une séquence d'éléments D, c'est-à-dire $\mathsf{Exp}_D^{\#n}$. Afin de détecter les propriétés croissantes entre les valeurs abstraites, nous devons introduire l'opérateur abstrait $\leq^\#_D$: $D \times D \rightarrow \mathbb{B}$. Avant d'introduire les instanciations de $\leq^\#_D$ dans les équations (4.7) et (4.8) pour les domaines définis à la section 4.4.2 nous détaillons les propriétés générales que $\leq^\#_D$ doit satisfaire. Il est à noter que, dans la propriété 2, l'ordre $\leq$ est l'ordre habituel sur les entiers.

Propriété 2 *Un opérateur binaire abstrait* $\leq^\#$: $D \times D \rightarrow \mathbb{B}$ *doit satisfaire* $\forall x^\# \in D, \bot \leq^\# x^\#$ *et :*

$$\begin{gathered}\forall x^\#, y^\# \in D \setminus \{\bot\}, \forall x \in \gamma_D(x^\#), y \in \gamma_D(y^\#),\\ x^\# \leq^\#_D y^\# \implies x \leq y\end{gathered}$$

Soit $G = \{\bot, \nearrow, \searrow, \rightarrow, \top\}$ un ensemble de gradients abstraits, muni de l'ordre partiel $\sqsubseteq_G$ avec $\forall g \in G, \bot \sqsubseteq_G g, g \sqsubseteq_G \top$ comme illustré dans la figure (15).

En utilisant $\leq^\#_D$, nous définissons le gradient abstrait $g \in G$, associé à la séquence de valeurs $(x_1 \cdot x_2 \cdot \ldots \cdot x_n) \in D^n$. On introduit la dernière abstraction :

$$\langle \mathsf{Exp}_D^{\#n}, \subseteq_D^n \rangle \xrightleftharpoons[\alpha_G]{\gamma_G} \langle G, \sqsubseteq_G \rangle \tag{4.6}$$

où

$$\alpha_G(x_1 \cdot x_2 \ldots \cdot x_n) = \begin{cases} \nearrow & \text{si } x_1 \leq_D^\# x_2 \cdot \ldots \cdot x_{n-1} \leq_D^\# x_n, \\ \searrow & \text{si } x_n \leq_D^\# x_{n-1} \cdot \ldots \cdot x_2 \leq_D^\# x_1, \\ \rightarrow & \text{si } x_1 = x_2 \cdot \ldots \cdot x_{n-1} = x_n, \\ \top & \text{sinon.} \end{cases}$$

et,

— $\gamma_G(\nearrow) = \{(x_1, \ldots, x_n) \in D^n \mid \forall i \in [1, n[, x_i \leq_D^\# x_{i+1}\}$.

— $\gamma_G(\searrow) = \{(x_1, \ldots, x_n) \in D^n \mid \forall i \in [1, n[, x_{i+1} \leq_D^\# x_i\}$.

— $\gamma_G(\rightarrow) = \{(x_1, \ldots, x_n) \in D^n \mid \forall i \in [1, n[, x_i = x_{i+1}\}$.

— $\gamma_G(\top) = \{(x_1, \ldots, x_n) \in D^n\}$.

Théorème 2 (Correspondance de Galois Exp) *On a bien une correspondance de Galois avec les deux fonctions* (α_G, γ_G).

$$\langle \mathit{Exp}_D^{\#n}, \sqsubseteq_D^n \rangle \xrightleftharpoons[\alpha_G]{\gamma_G} \langle G, \subseteq_G \rangle$$

Preuve. $\mathsf{Exp}_D^{\#n}$ et G sont des treillis complets et les fonctions γ_G et α_G sont définies composante par composante pour chaque vecteur : elles sont monotones par construction. La composition $\alpha_G \circ \gamma_G(g) = g$ est réductive alors que $\gamma_G \circ \alpha_G(x_1, \ldots, x_n) = \{(x_1, \ldots, x_n) \in D^n | \exists (x'_1, \ldots, x'_n) \in \mathsf{Exp}_D^{\#n} : \quad \forall i \in [1, n],\ \alpha_{\mathsf{Exp}_D^{\#n}}(x'_i) = \alpha_{\mathsf{Exp}_D^{\#n}}(x_i)\} \supseteq_D^n \mathsf{Exp}_D^{\#n}$ est extensive puisque $\alpha_{\mathsf{Exp}_D^{\#n}}$ est aussi extensive. □

Instanciation. Nous devons définir l'opérateur $\leq_D^\#$ pour les deux domaines $\langle K, \sqsubseteq_K \rangle$ et $\langle I, \sqsubseteq_I \rangle$.

— Pour le domaine des constantes Kildall,

$$x^\# \leq_K^\# y^\# \triangleq \begin{cases} \text{vrai} & \text{si } x^\# = \bot, \\ \text{vrai} & \text{si } x^\#, y^\# \in \mathbb{Z}, \gamma_K(x^\#) \leq \gamma_K(y^\#), \\ \text{faux} & \text{sinon} \end{cases} \tag{4.7}$$

— Pour les intervalles,

$$x \leq^\# y \begin{cases} x = [a, b] \quad y = [c, d] \qquad \text{et} \quad b \leq c \\ \text{ou} \quad x = \bot \end{cases} \tag{4.8}$$

Propriété 3 *L'opérateur binaire* $\leq_K^{\#}: K \times K \to \mathbb{B}$ *satisfait :*

$$\forall x^{\#}, y^{\#} \in K, \forall x \in \gamma_K(x^{\#}), y \in \gamma_K(y^{\#}),$$
$$x^{\#} \leq_K^{\#} y^{\#} \implies x \leq y$$

Preuve. Soit $x^{\#}, y^{\#} \in K$ tel que $x^{\#} \leq_K^{\#} y^{\#}$, seuls trois cas sont possibles :

1. $x^{\#} = \bot$, $y^{\#} = i$ pour certains $i \in \mathbb{Z}$

$$\gamma_K(\bot) = \bot, \gamma_K(i) = \{i\} \quad et \quad \bot \subseteq \{i\}.$$

2. $x^{\#} = i$ pour $i \in \mathbb{Z}$, $y^{\#} = \top$. Dans ce cas :

$$\gamma_K(i) = \{i\}, \gamma_K(\top) = \top \quad et \quad \{i\} \subseteq \top.$$

3. $x^{\#} = \bot$, $y^{\#} = \top$. Alors,

$$\gamma_K(\bot) = \bot, \gamma_K(\top) = \top \quad et \quad \bot \subseteq \top.$$

□

Propriété 4 *L'opérateur binaire* $\leq_I^{\#}: I \times I \to \mathbb{B}$ *satisfait :*

$$\forall x^{\#}, y^{\#} \in I, \forall x \in \gamma_I(x^{\#}), y \in \gamma_I(y^{\#}),$$
$$x^{\#} \leq_I^{\#} y^{\#} \implies x \leq y$$

Preuve. Les trois cas suivants sont similaires aux cas correspondants dans la preuve de la propriété 2 :

1. $x^{\#} = \bot, y^{\#} = \top$,
2. $x^{\#} = \bot, y^{\#} = [a, b]$ pour $a, b \in \mathbb{Z}$,
3. $x^{\#} = [a, b]$ pour $a, b \in \mathbb{Z}, y^{\#} = \top$.

Le dernier cas à considérer est lorsque $x^{\#} = [a, b]$ et $y^{\#} = [c, d]$. Par hypothèse $[a, b] \leq_I^{\#} [c, d]$, et par définition de $\leq_I^{\#}$ dans l'équation (4.8) :

$$b \leq c. \tag{4.9}$$

Soit $x \in [a, b], y \in [c, d]$. Par conséquent, $x \leq b$ et $c \leq y$. En plus de l'équation (4.9) nous savons que $b \leq c$. Alors, nous concluons que $x \leq b \leq c \leq y$. □

Nous terminons cette section par deux exemples d'un ensemble de deux vecteurs de taille 3 contenant toutes les abstractions. La différence entre les deux exemples est l'abs-

traction des exposants en scalaires, de sorte que le premier exemple utilise la fonction abstraite α_{K}^{n} tandis que le second utilise la fonction abstraite α_{I}^{n}.

$$\left\{ \begin{pmatrix} 100 \\ 5 \\ 0.001 \end{pmatrix}, \begin{pmatrix} 0.1 \\ 8 \\ 100 \end{pmatrix} \right\} \overset{\alpha_{\mathsf{Exp}}^{n}}{\rightarrow} \begin{pmatrix} \{-1, 2\} \\ \{1\} \\ \{-3, 2\} \end{pmatrix} \overset{\alpha_{\mathsf{K}}^{n}}{\rightarrow} \begin{pmatrix} \top \\ 1 \\ \top \end{pmatrix} \overset{\alpha_{\mathsf{G}}^{n}}{\rightarrow} \quad \top$$

$$\left\{ \begin{pmatrix} 10 \\ 100 \\ 1000 \end{pmatrix}, \begin{pmatrix} 1 \\ 1000 \\ 10000 \end{pmatrix} \right\} \overset{\alpha_{\mathsf{Exp}}^{n}}{\rightarrow} \begin{pmatrix} \{0, 1\} \\ \{2, 3\} \\ \{3, 5\} \end{pmatrix} \overset{\alpha_{\mathsf{I}}^{n}}{\rightarrow} \begin{pmatrix} [0, 1] \\ [2, 3] \\ [3, 5] \end{pmatrix} \overset{\alpha_{\mathsf{G}}^{n}}{\rightarrow} \quad \nearrow$$

4.5 Partitionnement des données

Nous présentons maintenant la prochaine étape de notre contribution, (1) à partir des résultats fournis par l'analyse statique de la section 4.4, nous construisons une partition pour laquelle nous sélectionnons l'ordre de sommation approprié. Puis (2) nous appliquons notre technique sur un exemple provenant de problèmes réalistes en mécanique.

4.5.1 Algorithme glouton

Comme mentionné dans la section 4.1, nous visons à utiliser le domaine G pour abstraire les matrices utilisées comme argument d'algorithmes numériques. Notre objectif est d'utiliser les informations fournies par G pour déterminer la meilleure façon d'effectuer le calcul de la somme. Plus précisément, nous voulons détecter des lignes, des colonnes ou blocs de la matrice qui pourraient être optimisés efficacement en ce qui concerne la précision numérique. Comme décrit dans la section 4.4, en utilisant G, nous partitionnons les données $x = x_1, x_2, \dots x_n$ en fonction de leurs signes et ordres de grandeurs. Évidemment, il existe de nombreuses façons de partitionner une matrice en blocs afin d'attribuer un gradient à chaque bloc. Par contre, en essayant d'attribuer ou de calculer un seul gradient pour ces blocs on peut entraîner un résultat égal au gradient imprécis $\top$. Par conséquent, nous visons à partitionner la matrice, puis attribuer une séquence de gradients aux lignes ou aux colonnes résultantes du partitionnement. Par exemple, considérons le vecteur $x = 1, 2, 3, 4, 4, 3, 2, 1$. Alors qu'un gradient commun sera $\top$, nous choisissons de l'affecter à la séquence de gradients suivante : $\{\nearrow, \rightarrow, \searrow\}$.

Notre objectif pour un algorithme efficace, est de construire de gros blocs (c'est à dire un ensemble de vecteurs) avec des gradients homogènes. Le pire des cas, le cas le plus coûteux étant une partition complète de la matrice ou vecteur avec autant d'éléments abstraits que d'éléments concrets.

Pour construire une partition efficace, nous introduisons un algorithme glouton. Nous verrons dans nos résultats expérimentaux de la section 4.6 que cet algorithme fait abs-

```
Grad_position = [];
position = 0;
while (position < n) do
   Grad_position = append(Grad_position, position);
   Grad_x = compare(x[position], x[position + 1]);
   position = position + 1;
   New_Grad_x = compare(x[position], x[position + 1]);
   while ((Grad_x = New_Grad_x)  and  (position < n)) do
      position = position + 1;
      Grad_x = New_Grad_x;
      New_Grad_x = compare(x[position], x[position + 1]);
   end while
   position = position + 1;
end while
return Grad_position
```

FIGURE 16 – Algorithme glouton pour le partitionnement d'un vecteur de taille n en gradients.

traction d'un vecteur de taille n par quelques gradients différents. Notez que, dans les travaux futurs, pour la parallélisation, nous allons avoir besoin de blocs de même taille. Pour ce faire, nous subdivisons les blocs trouvés à l'aide de notre algorithme par le plus grand diviseur commun des tailles calculées pour chacun d'entre eux afin d'obtenir notre partition finale. Mentionnons également que notre algorithme glouton n'est pas forcément optimal. Une perspective pour des travaux futurs est d'explorer ce point.

Dans notre étude, pour détecter les blocs de même signe et de même ordre de grandeur nous utilisons un algorithme glouton décrit dans la figure 16.

L'idée est de comparer les valeurs du vecteur d'entrée par composante. Ces valeurs peuvent être par ordre décroissant, croissant ou équilibré. Le nombre de blocs est déterminé par le nombre de gradients calculés par cet algorithme. Le principe est le suivant, l'algorithme prend en entrée le vecteur solution $x = x_1, x_2.....x_n$ à chaque itération. Les sorties de l'algorithme représentent le gradient de chaque bloc noté $Grad_x$ et l'indice de la composante par laquelle le bloc débute noté par $Grad_position$. L'algorithme de la Figure 16 partitionne un vecteur x de taille n en gradients. Cet algorithme appelle la fonction *compare* définie dans la figure 17.

L'algorithme glouton compare les éléments du vecteur x du premier élément jusqu'au dernier (première boucle while), la position actuelle est notée par $position$. Tant que les valeurs sont ordonnées dans le même ordre suivant $\leq_I^\#$ ou $\leq_K^\#$, on garde les éléments dans le même bloc (deuxième boucle while). Lorsqu'un changement de gradient est détecté, nous commençons un nouveau bloc en initiant une nouvelle itération de la première boucle (première boucle while). Si l'on considère l'exemple présenté précédemment $x = 1, 2, 3, 4, 4, 3, 2, 1$, on obtient en sortie les deux informations suivantes pour chaque bloc, $[Grad_position = 1 : Grad_x = \nearrow]$, $[Grad_position = 4 : Grad_x = \rightarrow]$,

function $compare(x, y)$
if $x \leq^{\#}_{D} y$ **then**
  $result = \nearrow;$
end if
if $x \geq^{\#}_{D} y$ **then**
  $result = \searrow;$
end if
if $x =^{\#}_{D} y$ **then**
  $result = \rightarrow;$
else
  $result = \top$
end if

FIGURE 17 – Fonction de comparaison entre x et y.

$[Grad_position = 5 : Grad_x = \searrow]$.

Ce travail reste préliminaire et notre perspective est de s'appuyer sur les gradients abstraits pour déterminer comment partitionner les matrices en blocs, de telle manière que l'on puisse affecter chaque bloc à un processeur donné d'une machine parallèle. La finalité est de spécialiser l'algorithme de sommation de chaque processeur en fonction de ses données afin d'optimiser la précision numérique sans augmenter le temps d'exécution. Les placements de tâches habituels sont représentés dans la figure 18, notre objectif est de sélectionner l'une de ces configurations en fonction des informations du gradient.

$$\begin{pmatrix} \boxed{a_{11}x_1 \quad ..... \quad a_{1n}x_n} \\ \boxed{a_{21}x_1 \quad ..... \quad a_{2n}x_n} \\ \boxed{a_{31}x_1 \quad ..... \quad a_{3n}x_n} \\ \boxed{a_{m1}x_1 \quad ..... \quad a_{mn}x_n} \end{pmatrix} \qquad \begin{pmatrix} a_{11}x_1 & ..... & a_{1n}x_n \\ a_{21}x_1 & ..... & a_{2n}x_n \\ a_{31}x_1 & ..... & a_{3n}x_n \\ a_{m1}x_1 & ..... & a_{mn}x_n \end{pmatrix} \qquad \begin{pmatrix} a_{11}x_1 & ..... & a_{1n}x_n \\ a_{21}x_1 & ..... & a_{2n}x_n \\ a_{31}x_1 & ..... & a_{3n}x_n \\ a_{m1}x_1 & ..... & a_{mn}x_n \end{pmatrix}$$

FIGURE 18 – Les différentes configurations de parallélisation possibles.

4.5.2 Exemple

Illustrons la méthode sur un petit exemple, soit une matrice avec $n = 2$. Dans un premier temps, nous générons le système correspondant à l'exemple d'une poutre unidimensionnelle en flexion introduit dans un contexte plus général dans la section 4.6. Nous calculons ensuite la solution de ce système linéaire par la méthode de Jacobi, et nous associons à chaque valeur du vecteur solution son exposant à virgule flottante en utilisant

la fonction abstraite α^n_{Exp} présentée dans la section 4.4.

$$A = \begin{pmatrix} 11778.279410 & -1778.279410 \\ -1778.279410 & 3556.558820 \end{pmatrix}, \quad b = \begin{pmatrix} 0.000662 \\ 0.001125 \end{pmatrix}$$

$$x^1 = \begin{pmatrix} \text{-5.623062e-07} \Rightarrow Exp^1 = \text{-7} \\ \text{-3.162326e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix} \quad x^2 = \begin{pmatrix} \text{-1.039753e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.443479e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix}$$

$$x^3 = \begin{pmatrix} \text{-1.082201e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.682203e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix} \quad x^4 = \begin{pmatrix} \text{-1.118244e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.703427e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix}$$

$$x^5 = \begin{pmatrix} \text{-1.121448e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.721448e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix} \quad x^6 = \begin{pmatrix} \text{-1.124169e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.723050e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix}$$

$$x^7 = \begin{pmatrix} \text{-1.124169e-06} \Rightarrow Exp^1 = \text{-6} \\ \text{-3.723050e-06} \Rightarrow Exp^2 = \text{-6} \end{pmatrix}$$

Une fois cette étape terminée, nous appliquons la fonction abstraite notée par α_G sur les exposants générés par la première abstraction afin de représenter le nature croissante, décroissante ou équilibrée des scalaires vectoriels. Les résultats de cette abstraction sont donnés ci-dessous :

$$x^1 = \begin{pmatrix} Exp^1 = \text{-7} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \nearrow \quad x^2 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow$$

$$x^3 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow \quad x^4 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow$$

$$x^5 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow \quad x^6 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow$$

$$x^7 = \begin{pmatrix} Exp^1 = \text{-6} \\ Exp^2 = \text{-6} \end{pmatrix} \Rightarrow grad = \rightarrow$$

Comme nous pouvons l'observer, les itérations calculent le même gradient à partir de x^2. On peut en déduire deux points. Tout d'abord pour cet exemple précis, un point fixe différent du supremum du treillis abstrait pourrait être trouvé par une analyse statique. Deuxièmement, nous pouvons choisir un algorithme de sommation efficace pour cette séquence (somme équilibrée). Nous pourrions également affecter ce bloc de calcul aux processeurs en utilisant un seul algorithme de sommation.

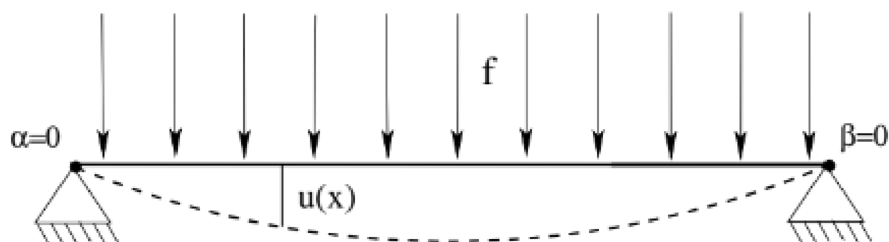

FIGURE 19 – Représentation d'une poutre $1D$ en flexion.

4.6 Expérimentations

Dans cette section, nous présentons quelques résultats expérimentaux sur un problème tiré de la mécanique. La section 4.6.1 introduit le problème et nos résultats sont présentés dans la section 5.4. Pour nos expériences, notre implémentation est en cours d'amélioration, nous n'utilisons que des matrices simples au lieu d'ensembles de matrices. Les abstractions $\leq_I^\#$ et $\leq_K^\#$ introduites dans la section 4.4.2 sont alors inutilisables dans ce contexte (elles se réduisent à des singletons) et on abstrait directement des matrices d'éléments concrets en gradients. Évidemment, nous visons à améliorer ce point dans nos prochains développements.

4.6.1 Poutre en flexion

Cet exemple consiste en un problème posé en Mécanique : la flexion d'une poutre élastique $1D$ avec des conditions aux limites de Dirichlet sur ses extrémités [4]. La discrétisation de ce type de problème est basée sur la méthode des éléments finis (MEF), typiquement utilisée en ingénierie. La résolution de la plupart des problèmes d'éléments finis nécessite de résoudre un système d'équations linéaires, par exemple. en utilisant l'algorithme de Jacobi. Le schéma représentatif de notre étude de cas est présenté par la figure 19. Dans la figure 19, u est discrétisé et représente le déplacement tel que $u_1 = \alpha$ et $u_{N+1} = \beta$, α et β les extrémités où la poutre est fixée telles que $(\alpha = \beta = 0)$. f est une force verticale constante agissant sur l'intervalle de domaine $\Omega = [0, 1]$. Le problème est formalisé comme suit :

$$\begin{cases} u''(x) = f & \forall x \in]0,1] \\ u(0) = \alpha \quad \text{et} & u(1) = \beta \end{cases} \tag{4.10}$$

La discrétisation du maillage produit un système linéaire à résoudre. Nous introduisons d'abord le maillage du domaine $\Omega = [0, 1]$. Nous considérons $N + 1$ nœuds $\{x_i, i = 1, \dots, N + 1\}$ de l'intervalle $[0, 1]$ avec $x_1 = 0$, $x_{N+1} = 1$ et $x_{i+1} = x_i + h_i, \forall i =$

$$\begin{pmatrix} \frac{1}{h_1}+C & \frac{-1}{h_1} & & & \\ \frac{-1}{h_1} & \frac{1}{h_1}+\frac{1}{h_2} & \frac{-1}{h_2} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{-1}{h_{N-1}} & \frac{1}{h_{N-1}}+\frac{1}{h_N} & \frac{-1}{h_N} \\ & & & \frac{-1}{h_N} & \frac{1}{h_N}+C \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \\ u_{N+1} \end{pmatrix} = \frac{f}{2} \begin{pmatrix} h_1 + C\alpha \\ h_1 + h_2 \\ \vdots \\ h_{N-1} + h_N \\ h_N + C\beta \end{pmatrix}$$

FIGURE 20 – Système linéaire à matrice tridiagonale.

$1, \ldots, N$. Ensuite, le domaine $[0, 1]$ est discrétisé en N intervalles (x_i, x_{i+1}) de taille h_i. Enfin, par substitution des valeurs connues (u_1, u_{N+1}) nous obtenons le système à matrice tridiagonale représenté sur la figure 20 que nous allons résoudre par la méthode de Jacobi. Ce système est composé de trois blocs, le premier bloc est une matrice tridiagonale correspondant à la poutre, le second (vecteur u) est le déplacement et le dernier bloc est un vecteur donnant la taille des éléments finis.

Pour nos expérimentations, les valeurs de h sont calculées automatiquement avec la spécificité d'obtenir des valeurs symétriques. Plus précisément, nous initialisons la première et la dernière valeur de h et on calcule les autres valeurs de telle sorte que $h[n-i-1] = h[i]$. Enfin, nous fixons les autres paramètres : la pénalisation $C = 10^6$ et la force verticale $f = -20N/m^2$.

Nous rappelons que notre motivation est de générer des systèmes linéaires de différentes taille N modélisant la flexion d'une poutre $1D$. Ensuite, de calculer la solution réelle X en utilisant la méthode de Jacobi. Une fois la solution calculée, nous faisons abstraction des valeurs des vecteurs solutions par leurs exposants respectifs (voir section 4.4.1). Après abstraction, nous appliquons l'algorithme glouton pour trouver le meilleur partitionnement de données, plus précisément, des blocs de valeurs de même signe et gradient.

4.6.2 Résultats expérimentaux

Dans cette section, nous présentons les résultats expérimentaux de notre étude, en se concentrant sur la précision numérique des calculs en utilisant nos algorithmes de partitionnement et de sommation.

Premièrement, nous voulons améliorer l'efficacité de notre technique dans la détection de scalaires de même signe et gradient qui peuvent être regroupés en blocs et qui pourraient éventuellement conduire à une optimisation de la précision numérique des calculs. Le cas le plus critique est d'avoir autant de blocs que de valeurs dans les vecteurs. En d'autres termes, le rapport entre le nombre total de blocs et la taille de la matrice est égal à 1. Pour tester notre méthode, nous générons différents systèmes linéaires de la forme $Ax = b$ qui modélisent l'exemple précédent décrit dans la section 4.6.1. Par la suite, nous résolvons ces systèmes en utilisant la méthode de Jacobi, puis nous appliquons notre

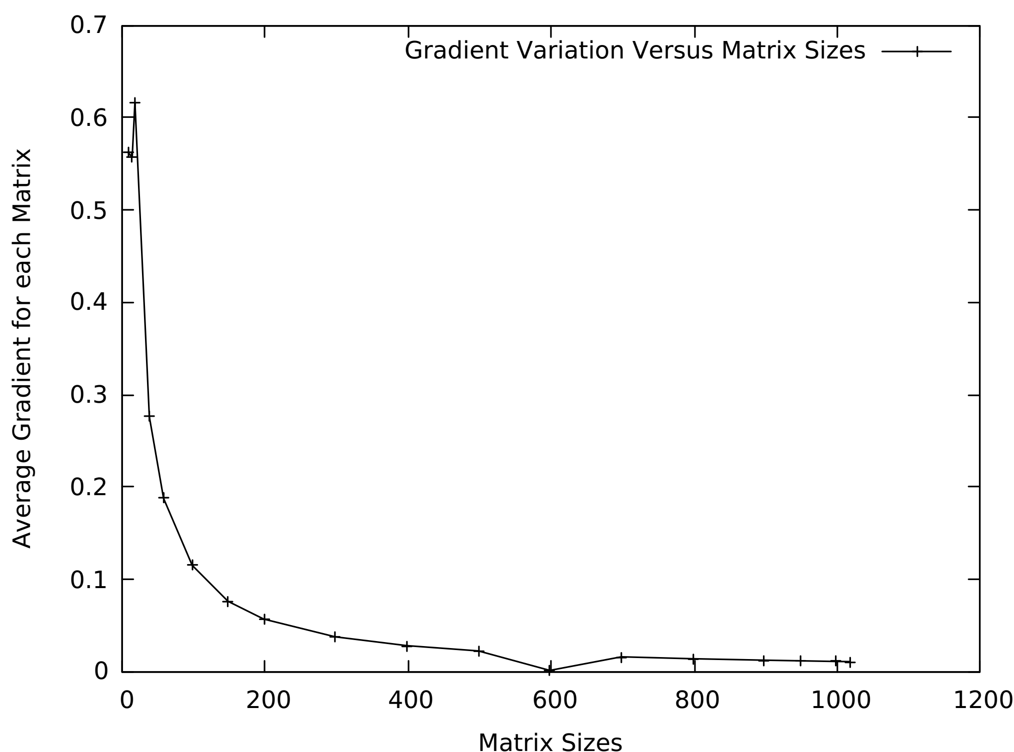

FIGURE 21 – Moyenne des gradients pour différentes tailles de matrices de 10 à 1000.

technique sur le vecteur solution x. Pour mesurer l'efficacité de notre technique nous calculons la moyenne des gradients d'un ensemble de matrices pour comparer la variation de ces moyennes avec celles des tailles de matrices. La figure 21 représente la moyenne des gradients correspondant à chaque taille de matrice de 10 à 1000. Nous remarquons que pour les petites matrices, par exemple $N = 100$ la moyenne est d'environ 0.6, nous concluons donc que le nombre de blocs représente 60% de la taille de la matrice. Nous remarquons également que lorsque la taille de la matrice augmente, la moyenne diminue. De ces deux remarques, nous déduisons que l'efficacité de notre technique est atteinte lorsque nous manipulons de grandes matrices.

Deuxièmement, nous voulons savoir si pour une matrice donnée nous pouvons généraliser la division des scalaires en lignes, colonnes ou blocs. Comme mentionné dans la section 4.5, pour un système linéaire de la forme $Ax = b$ nous appliquons notre technique à un vecteur solution $x = x_0, x_1, ...x_{n-1}$ à chaque itération afin de regrouper les données en fonction de leurs signes et ordres grandeurs. Nous associons à chaque séquence de données une séquence de gradients correspondante, et chaque bloc est représenté par deux valeurs : l'indice i de la composante $x_i, \forall i \in [0, n-1]$ par laquelle le bloc débute et le gradient associé à sa séquence de scalaires. Pour des raisons de simplicité, nous considérons une matrice de 16×16 avec des coefficients tirés d'un exemple réaliste correspondant à la flexion d'une poutre développée dans la section 4.6.1. Les résultats de notre étude sont donnés ci-dessous :

$iteration1$: [0 :→][2 :↘][3 :→][4 :↘][5 :→][9 :↗][10 :→][11 :↗][12 :→]
$iteration2$: [0 :→][1 :↘][2 :↗][3 :↘][5 :→][6 :↗][7 :↘][8 :→][10 :↗][11 :→][12 :↗]
$iteration3$: [0 :→][1 :↘][2 :↗][3 :→][4 :↘][5 :↗][6 :↘][7 :↗][8 :↘][10 :↗][11 :→]
$iteration4$: [0 :↘][1 :↗][2 :↘][4 :↗][5 :↘][7 :↗][9 :↘][12 :→]
$iteration5$: [0 :→][1 :↗][2 :↘][3 :↗][4 :↘][6 :↗][7 :↘][8 :↗][10 :↘]
$iteration6$: [0 :→][1 :↘][2 :↗][3 :↘][4 :→][6 :↗][7 :↘][9 :↗][12 :↘]
$iteration7$: [0 :→][1 :↘][2 :→][3 :↘][4 :↗][6 :↘][7 :↗][8 :↘][10 :↗][11 :↘][12 :↗]
$iteration8$: [0 :→][1 :↗][2 :↘][4 :↗][5 :↘][7 :↗][9 :↘]
$iteration9$: [0 :→][2 :↘][3 :↗][4 :↘][6 :↗][7 :↘][8 :↗][10 :↘]
$iteration10$: [0 :→][1 :↘][2 :↗][3 :↘][4 :→][5 :↗][7 :↘][9 :→][10 :↗][12 :↘]
$iteration11$: [0 :→][1 :↘][2 :↗][3 :↘][4 :↗][6 :↘][7 :↗][8 :↘][10 :↗][11 :↘][12 :↗]
$iteration12$: [0 :→][1 :↗][2 :↘][4 :↗][5 :↘][6 :→][8 :↗][9 :↘]
$iteration13$: [0 :↘][1 :↗][2 :↘][3 :↗][4 :↘][6 :↗][7 :↘][8 :↗][10 :↘]
$iteration14$: [0 :↗][1 :↘][2 :↗][3 :↘][5 :↗][7 :↘][9 :↗][12 :↘]
$iteration15$: [0 :→][1 :↘][2 :↗][3 :↘][4 :↗][6 :↘][7 :↗][8 :↘][10 :↗][11 :↘][12 :↗]
$iteration16$: [0 :→][1 :↗][2 :↘][4 :↗][5 :↘][7 :↗][9 :↘]
$iteration17$: [0 :→][1 :↗][2 :↘][3 :↗][4 :↘][6 :↗][7 :↘][8 :↗][10 :↘]
$iteration18$: [0 :→][1 :↘][2 :↗][3 :↘][5 :↗][7 :↘][9 :↗][12 :↘]
$iteration19$: [0 :↗][1 :↘][2 :↗][3 :↘][4 :↗][6 :↘][7 :↗][8 :↘][10 :↗][11 :↘][12 :↗]
$iteration20$: [0 :↘][1 :↗][2 :↘][4 :↗][5 :↘][7 :↗][9 :↘]
$iteration21$: [0 :→][1 :↗][2 :↘][3 :↗][4 :↘][5 :→][6 :↗][7 :↘][8 :→][9 :↗][10 :↘]
$iteration22$: [0 :→][1 :↘][2 :↗][3 :↘][5 :↗][7 :↘][9 :↗][12 :↘]
$iteration23$: [0 :→][1 :↘][2 :↗][3 :↘][4 :↗][6 :↘][7 :↗][8 :↘][10 :↗][11 :↘][12 :↗]
$iteration24$: [0 :→][1 :↗][2 :↘][4 :↗][5 :↘][7 :↗][9 :↘]
$iteration25$: [0 :→][2 :↘][3 :↗][4 :↘][6 :↗][7 :↘][8 :↗][10 :↘]

Si nous prenons l'itération 11 comme exemple, nous remarquons qu'après chaque 4 itérations nous trouvons la même séquence de blocs, c'est-à-dire que la même séquence de gradients est associée aux séquences de scalaires telles qu'elles sont représentées par les itérations 15, 19, 23. De la même façon, les itérations 16 et 14 sont répétées après chaque 4 itérations données par les itérations 20, 24 et 18, 22, respectivement. Un analyseur statique dépliant la boucle 4 fois pourrait alors trouver un point fixe pour cet exemple.

4.7 Conclusion

Nous avons présenté notre technique qui s'appuie sur une analyse statique pour améliorer la précision numérique de la résolution des systèmes linéaires. Nous avons détaillé ses différentes étapes et l'algorithme utilisé pour détecter l'ordre potentiel des scalaires. Nous avons testé notre technique à travers des expérimentations réalisées sur un exemple provenant d'un problème mécanique. Bien que les résultats obtenus montrent l'efficacité

de notre technique d'analyse pour identifier différents blocs de données selon leurs signes et ordres grandeurs, ces blocs ne contiennent pas de longues séquences de valeurs. Nous nous intéresserons donc à la question suivante : comment exploiter les informations issues de l'analyse statique ? ou encore, comment choisir les algorithmes de somme adaptés aux blocs de même signe et ordre de grandeur ? En réponse à cette question, nous proposerons deux algorithmes pour sommer n nombres flottants efficacement. Nos algorithmes ont été détaillés et suivis des résultats expérimentaux mesurants leurs performances en terme de précision numérique aux chapitres 5 et 6

Chapitre 5

Algorithmes de somme précis, reproductibles

Contents

5.1 Introduction **57**

5.2 Somme précise en séquentiel **58**

5.3 Somme précise en parallèle **60**

5.3.1 Sommes partielles sur chaque processeur 60

5.3.2 Calcul d'une seule somme globale 61

5.4 Expériences numériques **62**

5.4.1 Résultats expérimentaux liés à la précision numérique 63

5.4.2 Résultats expérimentaux liés à la reproductibilité 70

5.4.3 Notre algorithme séquentiel [8] versus les deux algorithmes de Demmel et Hida [30] 73

5.5 Conclusion **77**

5.1 Introduction

Au chapitre 2, nous avons passé en revue quelques travaux de recherche existants dans la littérature concernant l'amélioration de la précision numérique [27, 28, 54, 62, 65, 74] ou de la reproductibilité [1, 29, 30, 46] des calculs impliquant des sommes. En effet, il y a eu de nombreux efforts de recherche liés à la précision numérique de la somme et du produit scalaire des nombres à virgule flottante [27, 28, 54, 62, 65, 74, 75, 76]. Dans ce qui suit, nous nous concentrons sur la somme à virgule flottante. Certains travaux sont basés sur des méthodes dites compensées [54, 62, 65, 74, 75, 76]. D'autres utilisent des accumulateurs de haute précision et manipulent la mantisse et les exposants des nombres flottants à sommer [27, 28].

Dans le présent chapitre, nous proposons deux algorithmes parallèles efficaces pour sommer n nombres à virgule flottante [8]. Le premier objectif de nos algorithmes est d'obtenir un résultat précis sans augmenter significativement la complexité linéaire de l'algorithme de somme récursive. Le deuxième objectif est d'améliorer la reproductibilité

des résultats de sommations par rapport à ceux calculés par l'algorithme de somme récursive, lorsque la somme est calculée par une machine parallèle. En effet, il est difficile de déboguer des programmes lorsque les résultats de plusieurs exécutions du même code sur des architectures parallèles différentes ou même similaires sont différents. On note que reproductibilité et précision ne sont pas synonymes mais l'amélioration de la précision améliore souvent la reproductibilité. Dans notre contexte, nous nous concentrons dans ce qui suit sur une solution qui vise à réduire ou éliminer les erreurs d'arrondi, i.e. en améliorant la précision numérique des calculs que nous verrons plus loin dans ce chapitre. Pour remédier à ces problèmes, nous proposons deux algorithmes de sommation avec les propriétés suivantes :

1. Ils améliorent la précision numérique du calcul de la somme à virgule flottante sans augmenter la précision de travail, en séquentiel ou en parallèle.
2. Ils améliorent la reproductibilité des résultats de somme indépendamment de la manière dont les sommations sont affectées aux processeurs.
3. Ils n'augmentent pas la complexité par rapport à l'algorithme de somme récursive.

L'algorithme séquentiel est introduit dans la Section 5.2. Deux versions parallèles de l'algorithme séquentiel sont présentées dans les sections 5.3.1 et 5.3.2, la première effectue des sommes locales de ses accumulations tandis que la seconde concentre toute l'accumulation finale sur un seul processeur. La Section 5.4 est dédiée aux expérimentations numériques, visant à montrer l'efficacité de nos algorithmes sur l'amélioration de la précision numérique ainsi que la reproductibilité des calculs de somme.

5.2 Somme précise en séquentiel

Comme mentionné dans la Section 5.1, nous fournissons un algorithme séquentiel pour additionner avec précision n nombres à virgule flottante. Cet algorithme offre plusieurs avantages par rapport aux algorithmes cités dans le chapitre 2. Premièrement, il améliore la précision numérique de la même manière que les algorithmes de Demmel et Hida [27], tout en effectuant les calculs dans la précision de travail sans utiliser d'accumulateurs de haute précision. Deuxièmement, il améliore la reproductibilité par rapport à la version parallèle des algorithmes de somme récursive. De plus, cet algorithme peut être facilement parallélisé indépendemment du nombre de processeurs. Quant à la complexité, elle est linéaire tout comme les algorithmes de somme récursive. Le principe de l'algorithme 9 de somme précise que nous formulons dans cette section est le suivant : avant de commencer la sommation, l'algorithme alloue un tableau appelé sum_by_exp qui est initialisé à 0 pour tous ses éléments. Supposons que les exposants varient de exp_min à exp_max. Le nombre d'éléments du tableau sum_by_exp est $exp_max - exp_min + 1$ éléments (notons que ce nombre ne dépend pas du nombre de valeurs à sommer n).

Algorithme 9 Somme précise en séquentiel

```
1: Initialization of the array   sum_by_exp
2: total_sum=0.0
3: for i=0 to n-1 do
4:     exp_s_i=getExp(s[i])+bias
5:     sum_by_exp[exp_s_i]=sum_by_exp[exp_s_i]+s[i]
6: for i=0 to exp_max-exp_min do
7:     total_sum=total_sum+sum_by_exp[i]
```

En utilisant un biais tel que $bias = -exp_min$, l'idée est de sommer toutes les valeurs d'exposant i en base 2 dans la cellule $sum_by_exp[i + bias]$. Cela évite la plupart des absorptions tout en évitant de trier explicitement le tableau. Soit $exp_(s_i)$ l'exposant de s_i en base 2. Pour calculer la somme $S = \sum_{i=0}^{n-1} s_i$, chaque valeur s_i est additionnée à la cellule appropriée $sum_by_exp[exp_s_i + bias]$ comme décrit par la première boucle. Par la suite, une somme est effectuée entre les éléments du tableau sum_by_exp pour calculer la somme totale (deuxième boucle). La complexité de l'algorithme 9 est de $O(n)$ comme l'algorithme de somme récursive. Nous verrons dans la Section 5.4.3 une comparaison en terme de précision numérique et temps d'exécution entre l'algorithme 9 et l'algorithme 7 de Demmel et Hida.

Dans le reste de cette section, nous évaluons la précision de l'algorithme 9. Dans notre contexte, nous avons privilégié l'approche consistant à n'utiliser que le mode d'arrondi au plus près, afin de valider nos algorithmes. Commençons par le lemme suivant.

Lemme 1 *On suppose que l'on travaille dans le mode d'arrondi au plus près. Soit* $S_n = \sum_{i=0}^{i<n} s_i$ *la somme en précision* f *de* n *nombres flottants de même exposant* exp*, i.e.* $\forall i,\ 0 \leq i < n,\ 2^{exp} \leq s_i < 2^{exp+1}$. *Alors l'erreur* ε_n *sur* S_n *est majorée par*

$$\varepsilon_n < 2^{exp+2\lceil \log_2 n \rceil - f + 1} \quad . \tag{5.1}$$

Preuve. D'après l'équation (2.2), on a

$$\varepsilon_n \leq \frac{(n-1)u}{u+1} \sum_{i=0}^{n-1} \mid s_i \mid$$

avec $u = 2^{-f}$ et $|s_i| < 2^{exp+1}$.
C'est-à-dire,

$$\varepsilon_n < \frac{(n-1)2^{-f}}{2^{-f}+1} \times n \times 2^{exp+1}$$

Puisque, $2^{-f} + 1 > 1$ alors on a,

$$\varepsilon_n < (n-1) \times n \times 2^{-f} \times 2^{exp+1},$$

$$\varepsilon_n < n^2 \times 2^{exp-f+1},$$

$$\varepsilon_n < 2^{exp+\log_2 n^2-f+1},$$

$$\varepsilon_n < 2^{exp+2\lceil\log_2 n\rceil-f+1}.$$

□

5.3 Somme précise en parallèle

Dans cette section, nous décrivons nos algorithmes parallèles qui visent à améliorer la précision numérique et la reproductibilité des résultats de sommation. Dans la Section 5.3.1 nous donnons le premier algorithme qui est caractérisé par le calcul d'une somme locale sur chaque processeur avant de calculer le résultat total tandis que la Section 5.3.2 présente le deuxième algorithme qui calcule la somme totale uniquement en appliquant une réduction.

5.3.1 Sommes partielles sur chaque processeur

Dans cette section, nous présentons notre premier algorithme parallèle (voir algorithme 10 ci-dessous). Plus précisément, supposons que nous ayons P processeurs et n

Algorithme 10 Somme précise avec sommes partielles sur chaque processeur

```
1: Initialization of the array   sum_by_exp
2: total_sum=0.0
3: for i=0 to n/P do
4:     exp_s_i=getExp(s[i])+bias
5:     sum_by_exp[exp_s_i]=sum_by_exp[exp_s_i]+s[i]
6: local_sum=0.0
7: for i=0 to exp_max-exp_min do                          ▷ Local summations
8:     local_sum=local_sum+sum_by_exp[i]
                                               ▷ Total sum by processor 0
9: MPI_Reduce(&local_sum, &total_sum,1, MPI_float, MPI_sum,0.)
```

valeurs à sommer (nous supposons $n \gg P$ et que P divise n). Nous attribuons n/P valeurs à chaque processeur. Le processeur i, $0 \le i < P$ calcule $\sum_{j=i\times n/P}^{(i+1)\times n/P} s_j$ par la méthode détaillée ci-après. Pour calculer le résultat final, nous appliquons une réduction entre les différentes sommes locales calculées par l'ensemble des processeurs P. Les valeurs s_i sont additionnées à la cellule appropriée $sum_by_exp[exp_s_i + bias]$ en fonction de leurs exposants de la même manière que celles réalisées à la section 5.2 pour l'algorithme 9. Notre algorithme est parallèle et, par conséquent, chaque processeur a un tableau sum_by_exp

à additionner localement. Au niveau des processeurs, les valeurs à sommer sont additionnées dans l'ordre croissant afin d'obtenir le résultat local propre à chacun. Une fois les sommes locales finales calculées, une réduction est effectuée par un processeur, qui à la fin nous renvoie la somme totale. Le coût de l'algorithme 10 est de $O(n/P)$ en utilisant P processeurs, aucun tri n'étant effectué et l'accès aux données à sommer se fait une seule fois.

5.3.2 Calcul d'une seule somme globale

Le deuxième algorithme parallèle diffère de l'algorithme 10 dans la manière dont la somme finale est calculée. Comme nous l'avons souligné dès l'introduction de ce chapitre,

Algorithme 11 Somme précise avec une seule somme globale

```
 1: Initialization of the array elements   sum_by_exp
 2: total_sum=0.0
 3: for i=0 to n/P do
 4:     exp_s_i=getExp(s[i])+bias
 5:     sum_by_exp[exp_s_i]=sum_by_exp[exp_s_i]+s[i]
 6: MPI_Gather(sum_by_exp,exp_max,,&sum_by_exp_proc0[rank*exp_max],exp_max,,0,)
    ▷ Summing the values by column before total_sum by processor 0
 7: for j=0 to exp_max-exp_min do
 8:     sum_by_column=0.0 ;
 9:     for i=0 to exp_max-exp_min do
10:         sum_by_column=sum_by_column+sum_by_exp_proc0[i*exp_max+j] ;
11:     total_sum=total_sum+sum_by_column
```

les valeurs s_i sont additionnées dans le tableau sum_by_exp en fonction de leur exposant exp_s_i. Mais plutôt que de calculer une somme locale dans chaque processeur, chaque tableau local sum_by_exp est envoyé au processeur 0 en utilisant la fonction MPI_Gather. Une fois cette étape terminée, le processeur 0 additionne les éléments de tous les tableaux (correspondant à tous les processeurs) par exposant pour calculer la somme totale donnée par l'équation (5.2) :

$$S = \sum_{j=0}^{E} \sum_{i=0}^{P-1} sum_by_exp[i, j] \tag{5.2}$$

où $E = exp_max - exp_min + 1$.

À titre d'illustration, la figure 22 résume le processus du calcul de la somme par l'algorithme 11. exp_min et exp_max étant des constantes, la complexité de l'algorithme 11

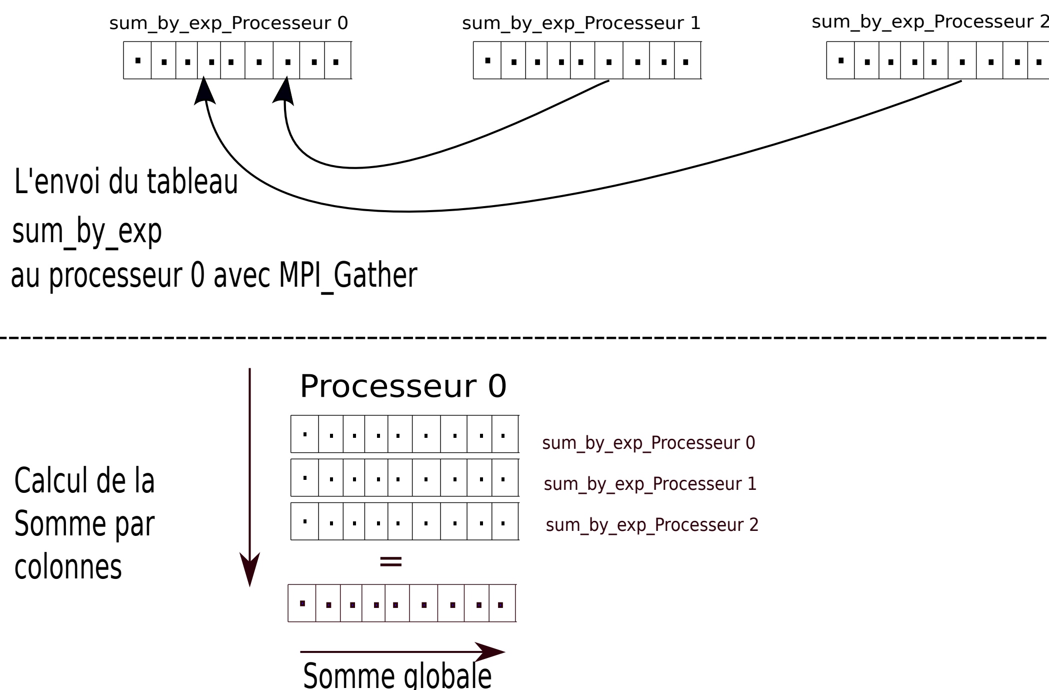

FIGURE 22 – Schéma du calcul de la somme.

reste linéaire en nombre de sommations n (le nombre d'exposants est constant). Par rapport à l'algorithme 10, l'algorithme 11 présente un ensemble d'avantages et d'inconvénients. L'algorithme 11 est le plus précis, en tenant compte de la somme par exposants faite globalement au lieu de calculer des sommes locales par chaque processeur comme le cas de l'algorithme 10. Il effectue plus de communications puisque $E \times P$ valeurs sont communiquées au lieu de P dans l'algorithme 10. Tout en étant linéaire, la constante de complexité $O(n)$ est supérieure à celle de l'algorithme 10 en raison des $E \times P$ sommes effectuées à l'étape finale au lieu de P dans l'algorithme 10. Notons que, dans le pire des cas, la borne d'erreur de l'algorithme 11 est identique à la borne donnée par l'équation (**??**) pour l'algorithme 10 (le pire des cas étant lorsque tous les nombres à virgule flottante ont le même exposant).

5.4 Expériences numériques

Dans cette section, nous évaluons d'abord l'efficacité de nos algorithmes décrits dans les sections 5.2 et 5.3 sur l'amélioration de la précision numérique 5.4.1 et la reproductibilité (voir Section 5.4.2). Deuxièmement, une comparaison est faite entre notre algorithme séquentiel (Algorithme 9) et l'algorithme 7 de Demmel et Hida dans la Section 5.4.3. Pour nos expériences, nous utilisons des ensembles de données qui varient de 10000 à 100000 valeurs générées aléatoirement de façon à introduire des sommes mal conditionnées. Les valeurs à virgule flottante peuvent être petites, moyennes ou grandes, ce qui signifie, respectivement, de l'ordre de 10^{-7} , 10^{0} et 10^{7}. Ceci est motivé par le format simple précision

Datasets	Dataset1	Dataset2	Dataset3	Dataset4	Dataset5	Dataset6	Dataset7	Dataset8
Signe	+	+	+	+	+ and −	+ and −	+ and −	+ and −
% grandes valeurs	10%	30%	50%	70%	10%	30%	50%	70%

TABLE 5.1 – Signe et proportion des grandes valeurs parmi les petites et moyennes valeurs pour chaque ensemble de données [Thévenoux et al. [58]].

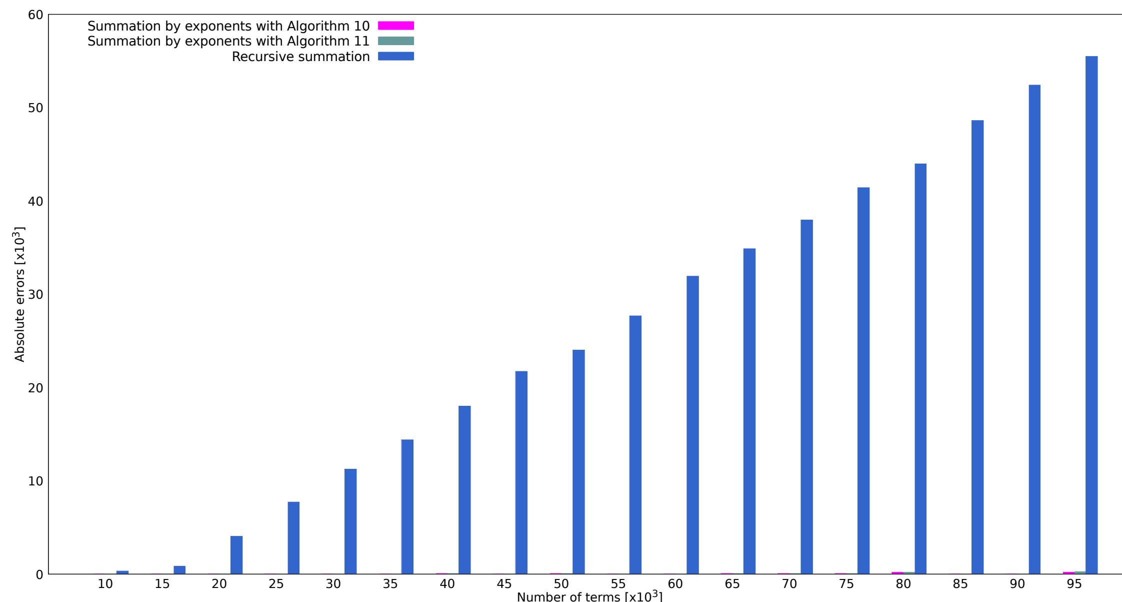

FIGURE 23 – Erreurs absolues entre les résultats de sommation du Dataset1 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision.

de l'arithmétique IEEE754. Afin de générer divers types d'absorptions et des éliminations catastrophiques, nous introduisons huit ensembles de données avec différentes proportions de grandes valeurs parmi des petites et des moyennes valeurs suivant la méthodologie de Thévenoux et al. [58] comme spécifié par le tableau 5.1.

5.4.1 Résultats expérimentaux liés à la précision numérique

Dans cette section, nous évaluons comment nos algorithmes améliorent la précision numérique des sommes à virgule flottante. Notons que les performances d'un algorithme peuvent être mesurées sur plusieurs aspects. Dans la suite, nous avons choisi d'évaluer la précision des algorithmes 10 et 11 en simple précision en comparant leurs résultats à une somme effectuée en double précision de l'arithmétique flottante, que nous considérons comme solution de référence (nous supposons que le résultat d'une somme en double précision est correct même si de façon générale, si la somme est mal conditionnée, le résultat peut être erroné).

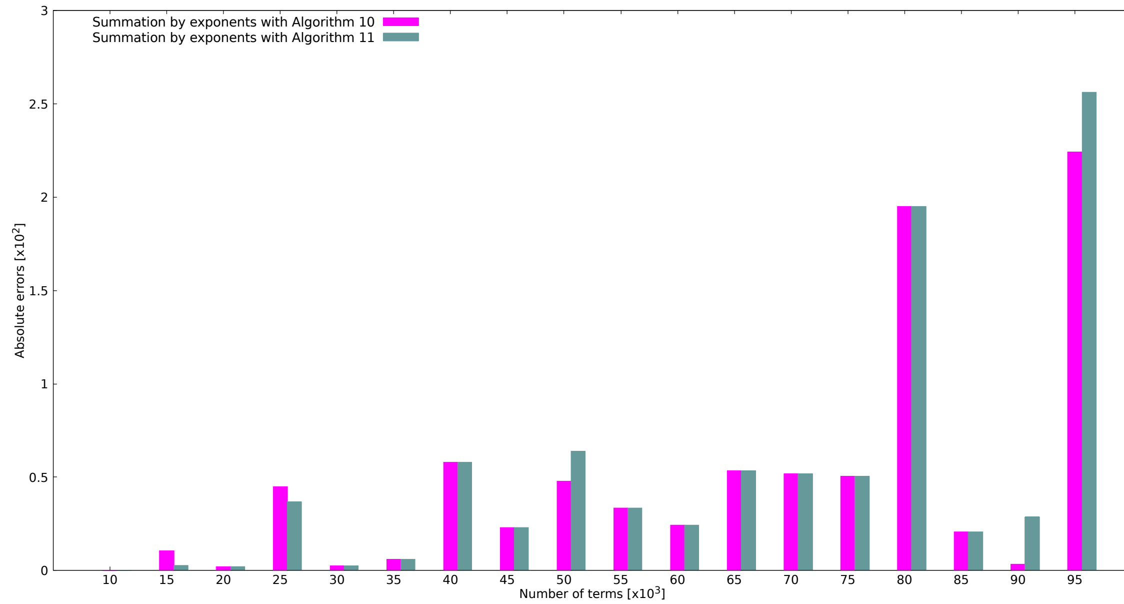

FIGURE 24 – Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset1.

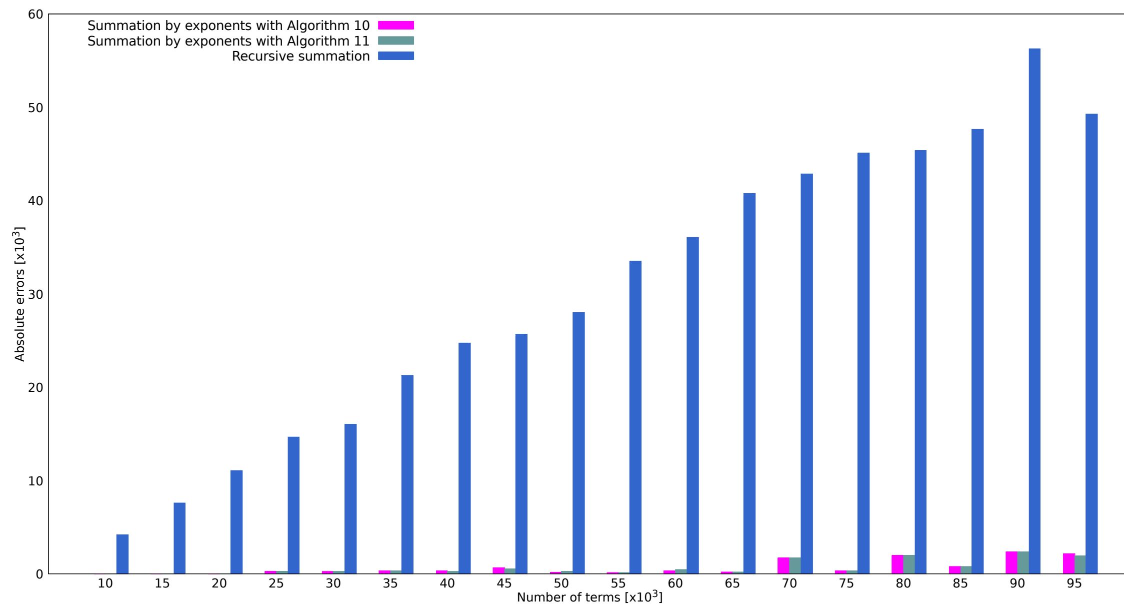

FIGURE 25 – Erreurs absolues entre les résultats de sommation du Dataset3 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision.

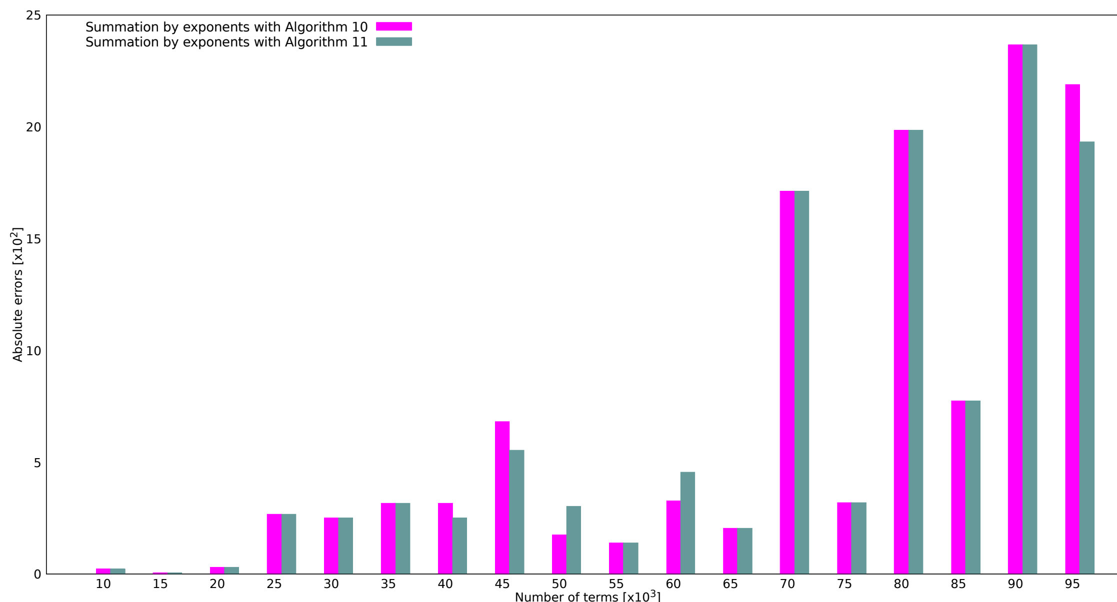

FIGURE 26 – Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset3.

Les quatre premiers ensembles de données contiennent des valeurs du même signe. Les figures 23, 25 et 27 représentent les erreurs absolues calculées pour ces quatre premiers ensembles de données. Ces ensembles de données ont des proportions différentes (de 10 % à 70 %) de grandes valeurs parmi les valeurs moyennes et petites. Pour nos expérimentations, pour chaque ensemble de données nous générons des ensembles de 10000 à 100000 valeurs qui seront additionnées à l'aide d'un des algorithmes étudiés. Par exemple, les résultats du `Dataset1` montrent que nos algorithmes de sommation améliorent la précision de la somme à virgule flottante par rapport à l'algorithme de somme récursive en simple précision (voir figures 23, 24). Nous remarquons que les erreurs absolues de l'algorithme de somme récursive sont beaucoup plus grandes que celles de nos algorithmes de somme pour chaque ensemble de données.

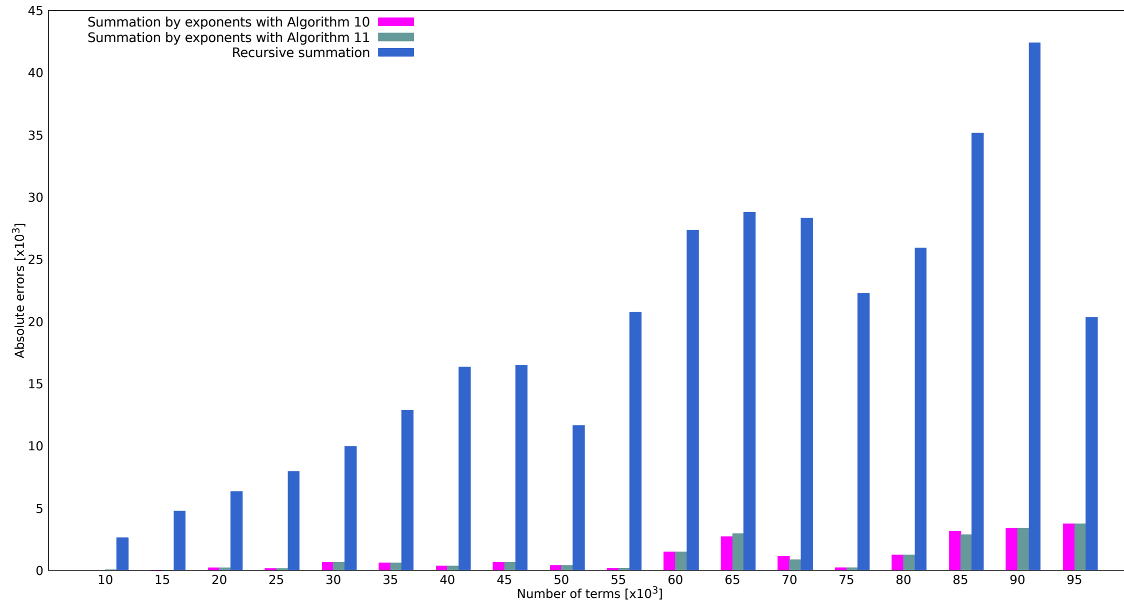

FIGURE 27 – Erreurs absolues entre les résultats de sommation du Dataset4 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision.

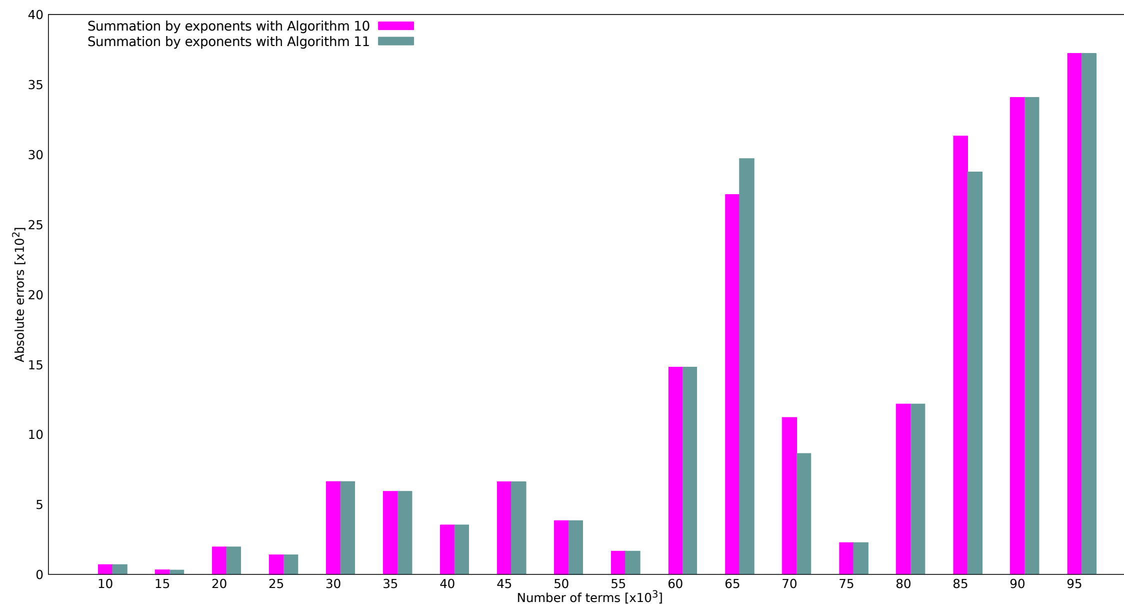

FIGURE 28 – Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset4.

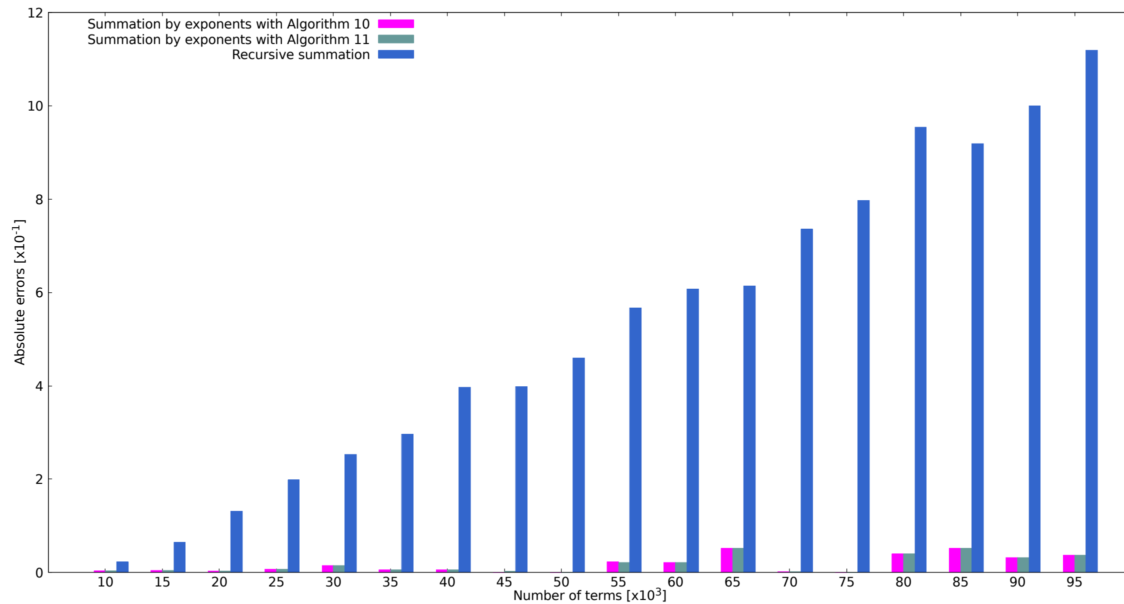

FIGURE 29 – Erreurs absolues entre les résultats de sommation du Dataset5 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision.

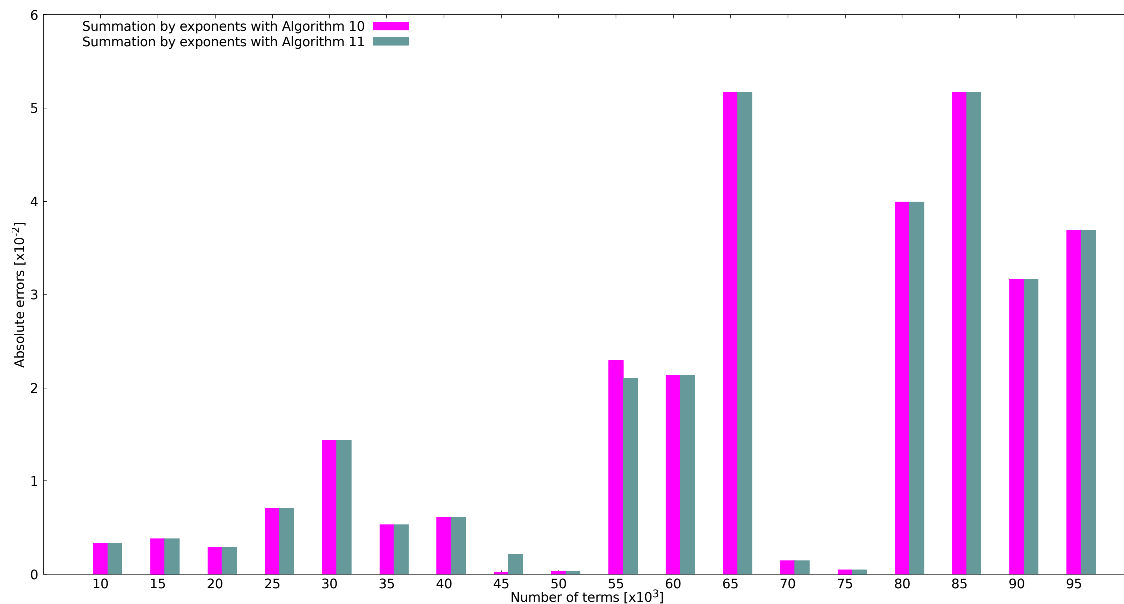

FIGURE 30 – Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset5.

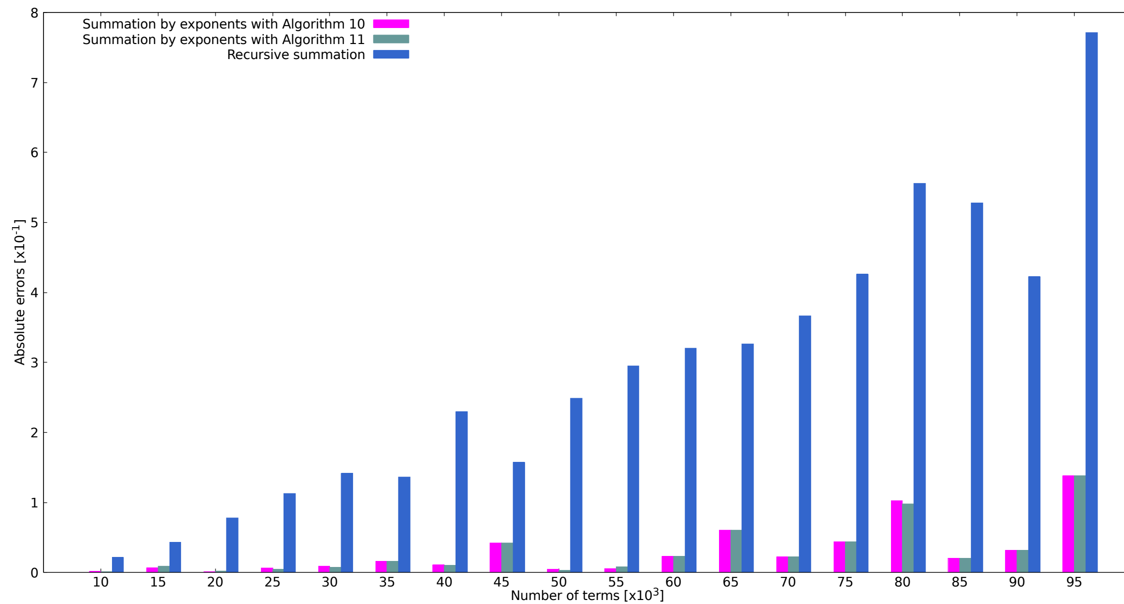

FIGURE 31 – Erreurs absolues entre les résultats de sommation du Dataset7 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision.

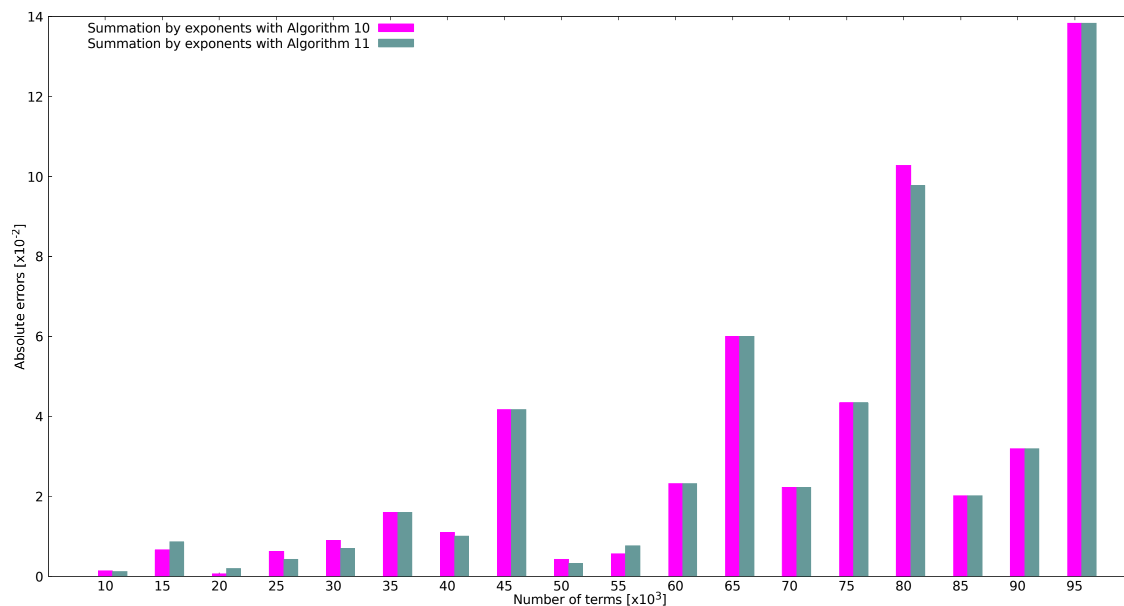

FIGURE 32 – Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset7.

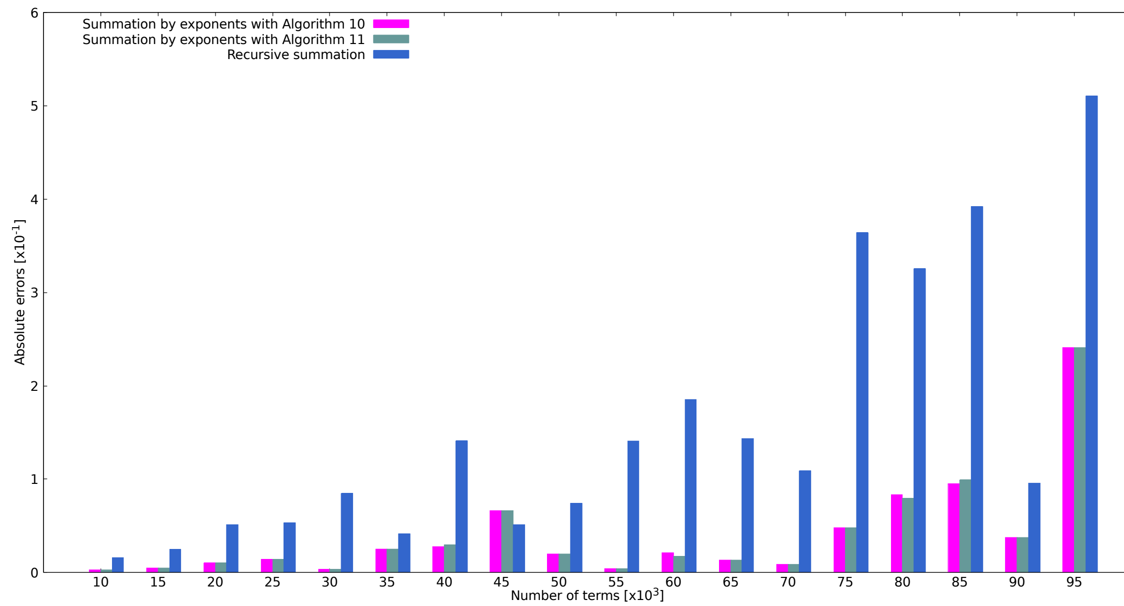

FIGURE 33 – Erreurs absolues entre les résultats de sommation du Dataset8 par les trois algorithmes : Algorithme 10, Algorithme 11 et l'algorithme de somme récursive en simple précision avec les résultats de sommation des mêmes ensembles de données utilisant un algorithme de somme récursive en double précision.

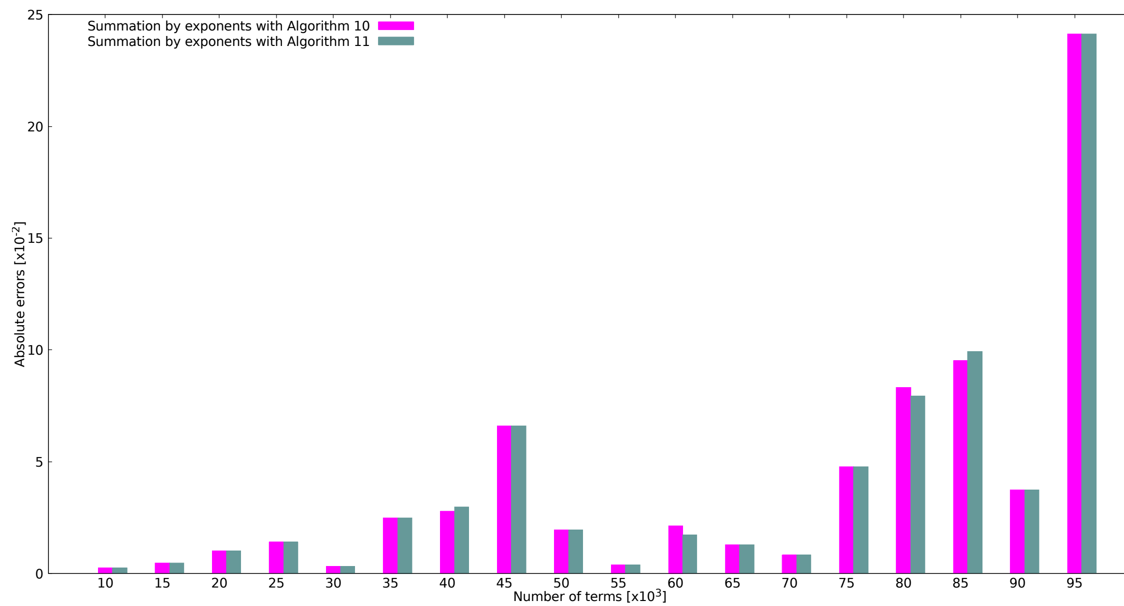

FIGURE 34 – Zoom sur les erreurs absolues de l'algorithme 10 et l'algorithme 11 pour le calcul de la somme du Dataset8.

Les erreurs absolues varient de 300 à 55000 pour l'algorithme de somme récursive contrairement à nos algorithmes où les erreurs sont comprises entre 0 et 270. Concernant nos algorithmes, trois cas se présentent :

— Les résultats des sommations des deux algorithmes (algorithmes 10 et 11) sont similaires, on obtient donc les mêmes erreurs absolues,

— Les résultats de sommation par l'algorithme 10 sont meilleurs que ceux de l'algorithme 11, ce qui se justifie par le fait que les sommes intermédiaires par colonne calculées par l'algorithme 11 ne sont pas en ordre croissant des exposants, donc une perte de précision est observée. En conséquence, les erreurs absolues calculées pour cet algorithme seront supérieures à celles de l'algorithme 10,

— Les résultats de somme de l'algorithme 11 sont meilleurs que ceux de l'algorithme 10 en raison du gain en précision obtenu grâce à l'ordre croissant des sommes intermédiaires calculées par l'algorithme 11, donc les erreurs absolues de l'algorithme 10 sont supérieures à celles de l'algorithme 11.

De la même manière, on note que les résultats du `Dataset2` et `Dataset3` sont similaires à ceux du `Dataset1`. Cependant, les résultats du `Dataset4` (figures 27 et 28) montrent que les erreurs absolues de l'algorithme de somme récursive sont inférieures à celles calculées pour d'autres ensembles de données, ce qui est dû à la proportion de grandes valeurs contenues dans chacun d'eux. Plus précisément, dans le `Dataset4` nous avons 70% de grandes valeurs parmi les petites et moyennes, donc les erreurs accumulées de cet ensemble de données causées par les absorptions ne seront pas plus grandes que celles générées par le `Dataset1` ou même le `Dataset2` contenant respectivement 10% et 30% de grandes valeurs parmi les petites et moyennes valeurs. En fait, les erreurs absolues de l'algorithme de somme récursive passent de 55000 à 3700.

Dans la deuxième partie de nos expériences, nous examinons les ensembles de données contenant les valeurs à virgule flottante avec des signes positifs et négatifs. Les figures 29, 31 et 33 résument les résultats obtenus pour les ensembles de données correspondants. Le but de ces expériences est de mesurer l'impact des éliminations catastrophiques sur la précision du calcul de la somme. Comme nous pouvons l'observer sur les figures 29, 31 et 33, nos algorithmes de sommation améliorent également la précision numérique des calculs effectués entre des valeurs de signes différents, par rapport à l'algorithme de somme récursive qui représente beaucoup plus d'erreurs absolues. En effet, la perte de précision a été causée d'une part par des calculs impliquant des valeurs à virgule flottante de différents signes et d'autre part par l'ordre de grandeur des valeurs, (i.e. petites, moyennes et grandes).

5.4.2 Résultats expérimentaux liés à la reproductibilité

Dans cette section, nous évaluons l'efficacité des algorithmes 10 et 11 sur l'amélioration de

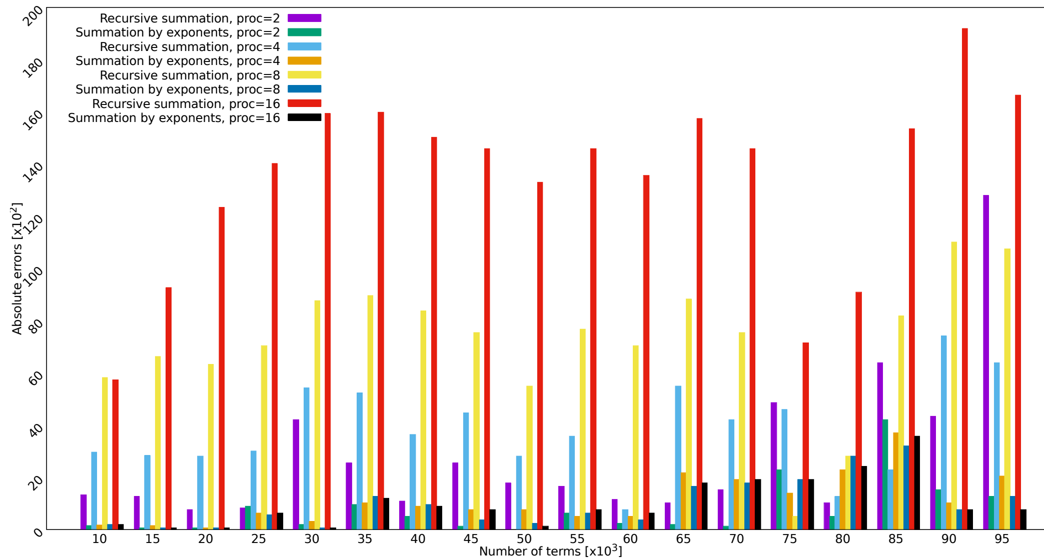

FIGURE 35 – Reproductibilité des résultats de somme de l'algorithme 10 et de l'algorithme de somme récursive en simple précision pour le Dataset3 en fonction du nombre de processeurs.

la reproductibilité. En effet, la combinaison de la non-associativité des opérations en virgule flottante comme l'addition et les calculs basés sur une architecture parallèle peut affecter la reproductibilité. Les figures 35 et 37 donnent les résultats de reproductibilité pour le Dataset3 et le Dataset4 en utilisant respectivement l'algorithme 10 et l'algorithme 11 avec un nombre P de processeurs variant de 2 à 16. Bien que les figures 36 et 38 donnent les résultats de reproductibilité pour le Dataset1 et le Dataset2 en utilisant respectivement l'algorithme 10 et l'algorithme 11 également en faisant varier le nombre P de processeurs de 2 à 16. Au cours de nos expériences, les erreurs absolues sont calculées entre les résultats de sommation de l'algorithme 10, de l'algorithme 11 et de l'algorithme de somme récursive en simple précision en faisant varier le nombre de processeurs et leur algorithme respectif avec un seul processeur. Premièrement, les résultats montrent que les erreurs absolues de l'algorithme de somme récursive sont plus importantes (entre 2368 et 14144) que celles des deux nouveaux algorithmes de sommation (entre 0 et 488). Deuxièmement, les résultats des sommations calculées par l'algorithme de somme récursive effectué sur un grand nombre de processeurs sont ceux qui ont une erreur plus importante,. Par exemple, pour un jeu de données de 100000 termes, l'erreur est estimée à 14144. En d'autres termes, les résultats des sommations obtenues en utilisant l'algorithme de somme récursive avec un nombre de processeur égale à 2 sont plus reproductibles que ceux obtenus avec 4, 8 et 16 processeurs. Contrairement à l'algorithme de somme récursive , les erreurs absolues liées aux deux algorithmes décrits dans la Section 5.3 sont négligeables quel que soit le nombre de processeur utilisé pour effectuer les calculs de sommation.

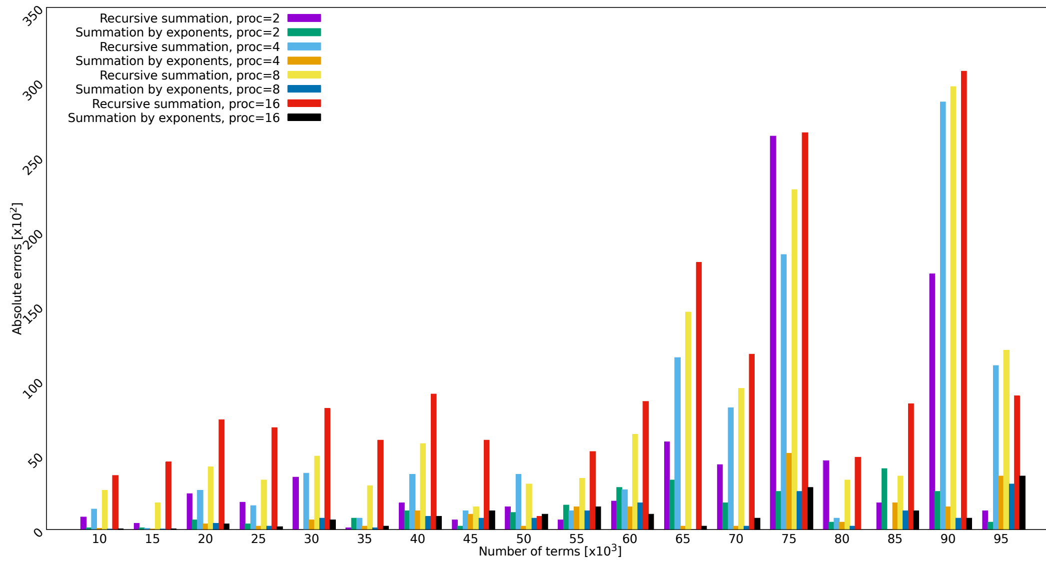

FIGURE 36 – Reproductibilité des résultats de somme de l'algorithme 10 et de l'algorithme de somme récursive en simple précision pour le Dataset4 en fonction du nombre de processeurs.

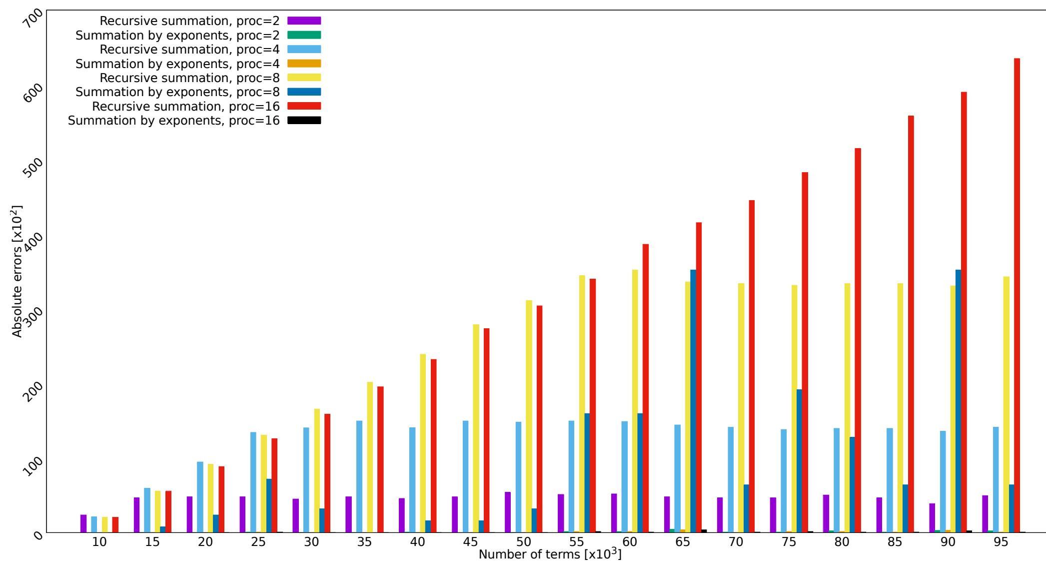

FIGURE 37 – Reproductibilité des résultats de somme de l'algorithme 11 et de l'algorithme de somme récursive en simple précision pour le Dataset1 en fonction du nombre de processeurs.

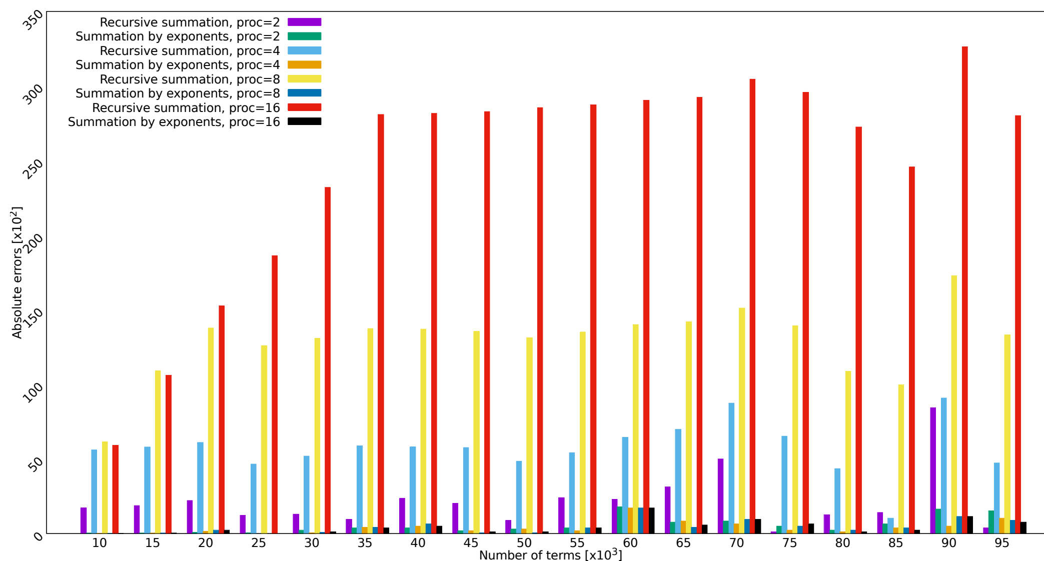

FIGURE 38 – Reproductibilité des résultats de sommation de l'algorithme 11 et de l'algorithme de somme récursive en simple précision pour le Dataset2 en fonction du nombre de processeurs.

5.4.3 Notre algorithme séquentiel [8] versus les deux algorithmes de Demmel et Hida [30]

Dans cette section, nous établissons une comparaison plus poussée entre notre algorithme séquentiel 9 déjà décrit dans la section 9 et les algorithmes 7 et 8 de Demmel et Hida donnés dans la section 2.5.

Commençons par l'algorithme 7. La première expérimentation consiste à comparer la précision des résultats de sommations effectuées en utilisant notre algorithme séquentiel (algorithme 9), l'algorithme de Demmel et Hida (algorithme 7) et l'algorithme de somme récursive. Prenons l'exemple du Dataset1 : nous remarquons que les erreurs absolues de l'algorithme de somme récursive sont beaucoup plus importantes que celles de notre algorithme de sommation et celles de l'algorithme de Demmel et Hida comme le montre la figure 39. Pour mieux illustrer cela, considérons la valeur 30000 de l'axe des abscisses. Nous remarquons que les erreurs absolues sont de l'ordre de 10^4, 10^1 et 10^0 pour l'algorithme de somme récursive, l'algorithme 9 et l'algorithme 7, respectivement. D'autre part, la figure 40 montre que les résultats de sommation de l'algorithme de Demmel et Hida sont plus précis par rapport à nos résultats en raison du tri effectué sur leurs valeurs avant les sommations contrairement à notre algorithme où aucun tri n'est effectué.

La deuxième expérimentation mesure le temps d'exécution (en secondes) de chaque algorithme. Par cette expérience, nous voulons montrer le compromis entre la précision et le temps d'exécution de notre algorithme (algorithme 9). La tableau 5.2 présente le temps en secondes pris par chaque programme (notre algorithme, l'algorithme de Demmel et Hida et l'algorithme de somme récursive) pour calculer les résultats de sommations de

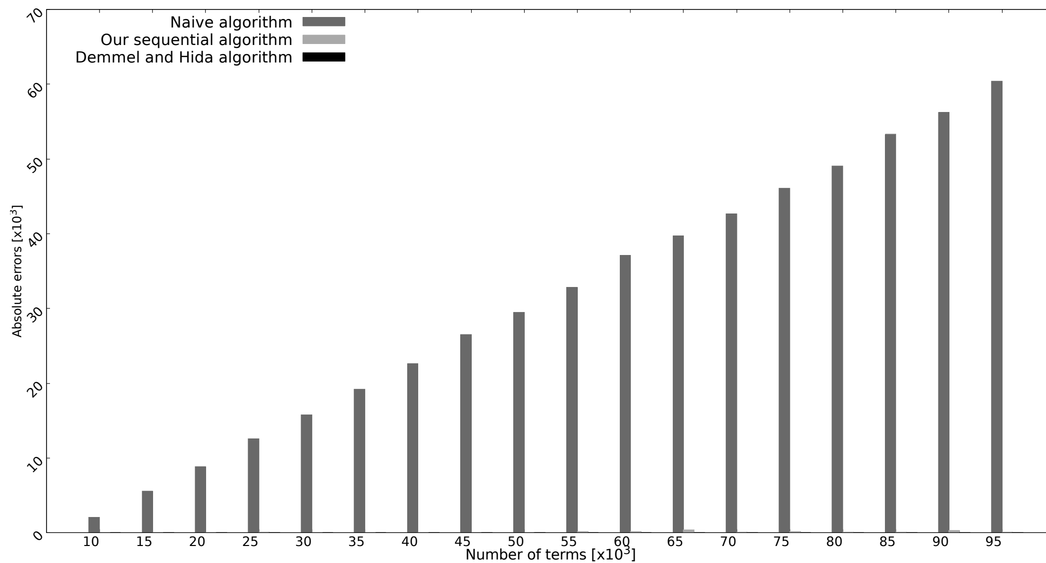

FIGURE 39 – Erreurs absolues entre les résultats de sommation du Dataset1 à l'aide de trois algorithmes : l'algorithme 9, l'algorithme 7 et l'algorithme de somme récursive en simple précision avec les résultats de sommation du Dataset1 obtenus par l'algorithme de somme récursive en double précision.

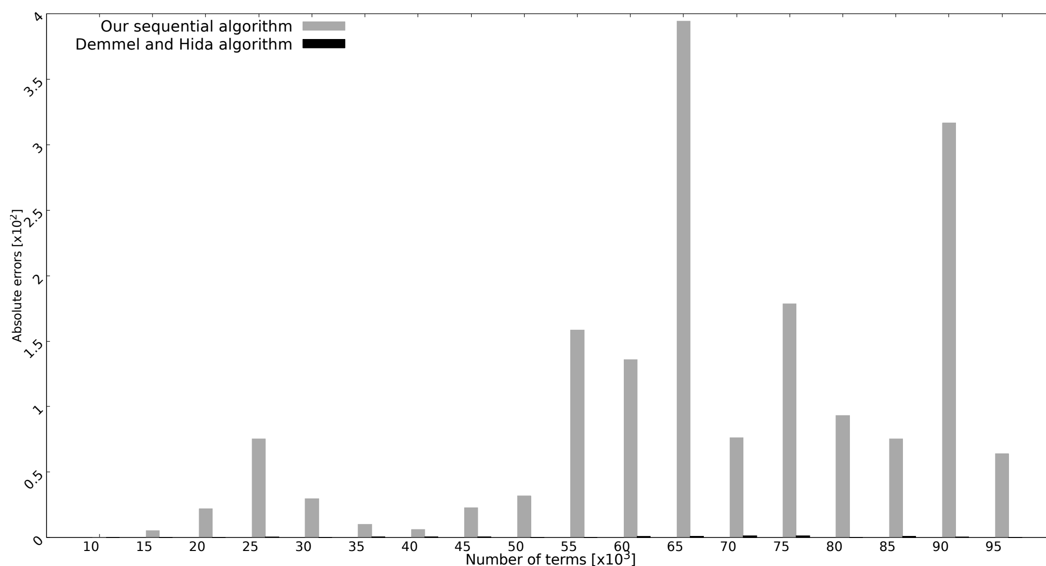

FIGURE 40 – Zoom sur les erreurs absolues de notre algorithme séquentiel et de l'algorithme de Demmel et Hida.

quelques sous-ensembles du Dataset1. Les résultats présentés par la figure 41 et le tableau 5.2 montrent que l'algorithme de Demmel et Hida a besoin de plus de temps pour calculer ces sommations. En outre, notre algorithme nécessite beaucoup moins de temps que l'algorithme de Demmel et Hida et un peu plus que l'algorithme de somme récursive. Par exemple : pour additionner un jeu de données de 10000 termes, il ne faut que $0,0001s$ et $0,001s$ en utilisant respectivement l'algorithme de somme récursive et l'algorithme 9, tandis que cette somme nécessite $7s$ par l'algorithme 7.

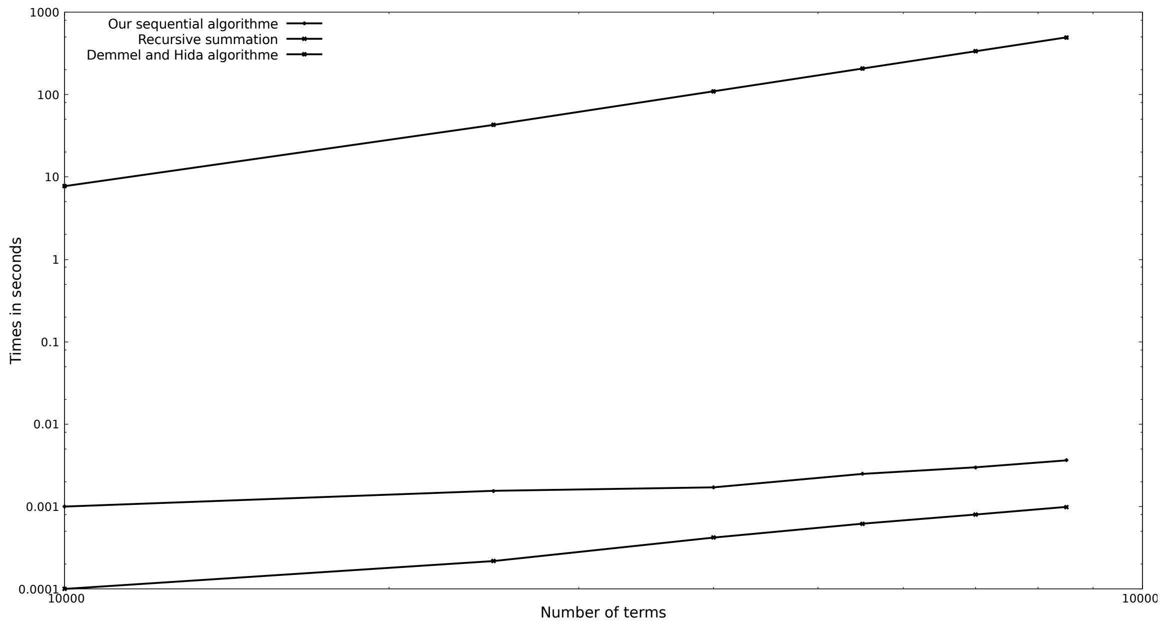

FIGURE 41 – Temps d'exécution de l'algorithme de somme récursive, l'algorithme 9 et l'algorithme 7.

Nbr d'éléments Dataset1	Algorithme de somme récursive	Algorithme 9	Algorithme 7
10000	1×10^{-4}s	1×10^{-3}s	7×10^{0}s
25000	2.18×10^{-4} s	1.55×10^{-3}s	4.29×10^{1}s
40000	4.2×10^{-4}s	1.71×10^{-3}s	1.09×10^{2}s
55000	6.2×10^{-4}s	2.49×10^{-3}s	2.07×10^{2}s
70000	8.0×10^{-4}s	2.99×10^{-3}s	3.35×10^{2}s
85000	9.9×10^{-4}s	3.63×10^{-3}s	4.95×10^{2}s

TABLE 5.2 – Comparaison des temps d'exécution de trois algorithmes : algorithme de somme récursive, algorithme 9 et l'algorithme 7.

Pour aller plus loin, nous avons établi une deuxième comparaison avec les algorithmes de Demmel et Hida. Cette fois, nous nous sommes concentrés sur l'algorithme 8. Premièrement, une comparaison en terme de précision numérique entre notre algorithme séquentiel 9 et l'algorithme 8 de Demmel et Hida est donnée à la figure 42. La figure 42 montre les erreurs absolues dues au calcul de la somme de différents ensembles de données par l'algorithme 9 et l'algorithme 8 avec leurs résultats respectifs en double précision. Les résultats montrent que l'algorithme proposé par Demmel et Hida 8 est plus précis que l'algorithme 9 ce qui est justifié par leur utilisation des accumulateurs en extra précision pour le calcul des sommes partielles puis le tri de ces dernières avant de calculer la somme finale, contrairement à notre cas où les accumulateurs utilisés sont en précision de travail et la somme finale est calculée sans tri préalable. Deuxièmement, nous avons

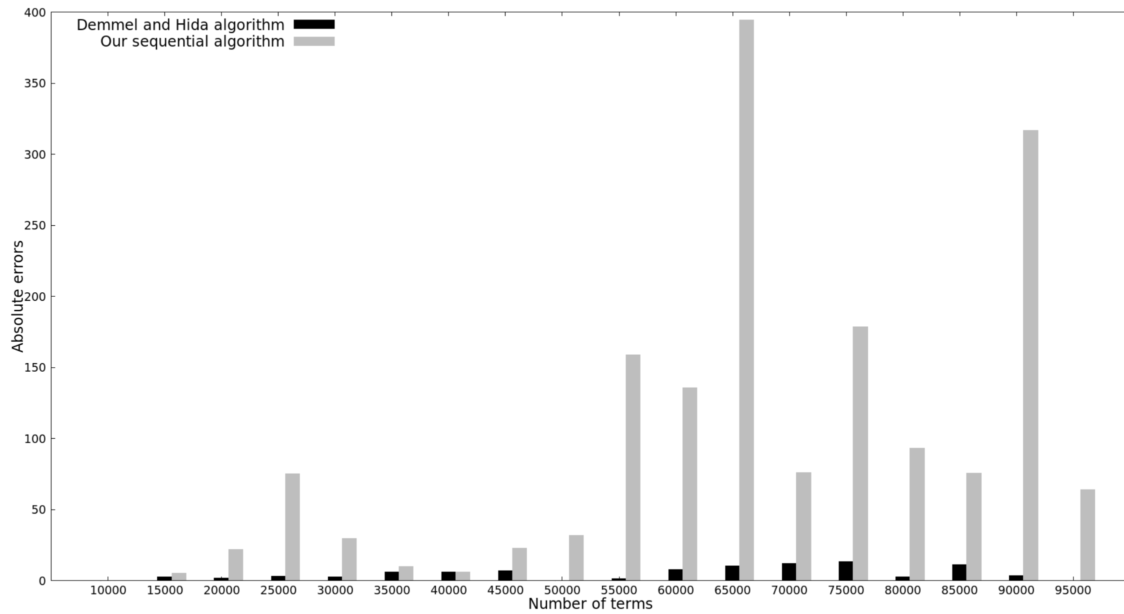

FIGURE 42 – Erreurs absolues entre les résultats de sommations du Dataset1 à l'aide de l'algorithme 9 et l'algorithme 8.

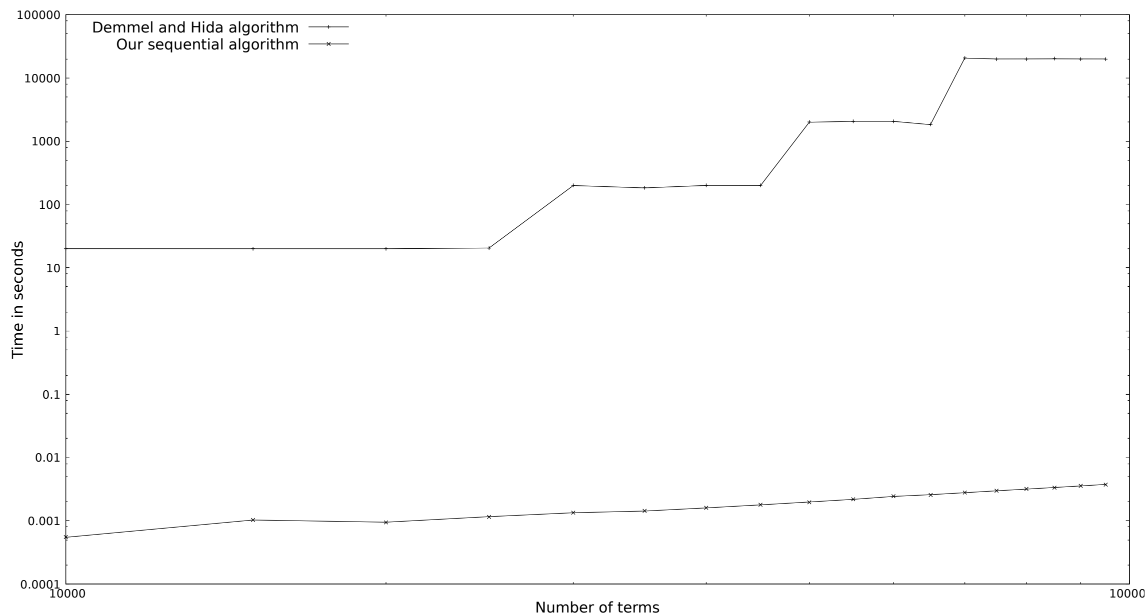

FIGURE 43 – Temps d'exécution de l'algorithme 9 et de l'algorithme 8.

mesuré le temps d'exécution des deux algorithmes : l'algorithme de Demmel et Hida 8 et notre algorithme séquentiel 9 (voir figure 43). La figure 43 montre que l'algorithme 8 de Demmel et Hida nécessite plus de temps par rapport à notre algorithme séquentiel 9 pour le calcul des sommes des mêmes ensembles de données. Ainsi, nous concluons des expériences montrées dans les figures 39, 40 et 42 que les sommations calculées par notre algorithme (algorithme 9) sont plus précises que celles de l'algorithme de somme récursive même s'il s'exécute plus rapidement que notre algorithme. Concernant la comparaison avec les algorithmes 7 et 8 de Demmel et Hida, leurs sommes sont plus précises en raison

des accumulateurs en extra précision et du tri effectué sur les valeurs à sommer avant les calculs mais leur inconvénient majeur est le temps nécessaire pour effectuer ces calculs ($207s$ pour un ensemble de données égal à 55000) par rapport à notre algorithme ($0,002s$ et ceci pour le même ensemble de données égal à 55000).

5.5 Conclusion

Nous avons présenté dans ce chapitre deux nouvelles approches parallèles pour le calcul de la somme de n nombres à virgule flottante. De plus, au travers diverses expérimentations, nous avons montré l'efficacité de nos algorithmes sur l'amélioration de la précision numérique et de la reproductibilité. L'amélioration a été mesurée en comparant nos résultats de sommation avec ceux de l'algorithme de somme récursive. Par la suite, une comparaison a été établie entre nos algorithmes et ceux proposés par Demmel et Hida [28].

Nos algorithmes ont été testés sur huit ensembles de données de différentes proportions de grandes valeurs parmi les petites et moyennes valeurs de signe positif ou des deux signes, ils effectuent tous les calculs dans la précision de travail sans avoir besoin d'augmenter la précision où d'utiliser des accumulateurs de plus haute précision le cas des algorithmes de Demmel et Hida [28]. De plus des résultats obtenus, nos algorithmes n'augmentent pas le coût de la complexité.

Nous allons voir dans le chapitre suivant, de nouvelles expérimentations sur l'algorithme 10 appliquées à des méthodes de calcul numériques. Nous allons également étudier l'impact de l'amélioration de la précision numérique sur l'accélération de convergence des méthodes itératives.

Chapitre 6

Performances des algorithmes proposés

Contents

6.1 Introduction **78**

6.2 Précision numérique **79**

6.2.1 Méthode de Simpson 79

6.2.2 Méthode de Factorisation LU 82

6.3 Convergence des méthodes itératives **84**

6.3.1 Méthode de Jacobi 84

6.3.2 Méthode de la puissance itérée 86

6.4 Reproductibilité **86**

6.4.1 Méthode de Simpson 87

6.4.2 Multiplication de matrices 88

6.5 Conclusion **89**

6.1 Introduction

Nous avons vu au chapitre précédent, deux nouveaux algorithmes pour additionner n nombres en virgule flottante. Comme évoqué à la Section 5.3, les algorithmes 10 et 11 effectuent les calculs uniquement dans la précision de travail, ne nécessitant qu'un accès aux exposants des valeurs à sommer. L'idée est de calculer les sommes en fonction de leurs exposants sans augmenter la complexité. Plus précisément, la complexité de l'algorithme est linéaire par rapport au nombre d'éléments à sommer, tout comme les algorithmes de sommation naïfs. Ces algorithmes ont été testés et ont montré leur efficacité sur des ensembles de données générés aléatoirement. Dans les sections suivantes, nous évaluons la version la plus simple et la plus rentable de l'algorithme 9 en ce qui concerne la précision ; la vitesse de convergence des schémas itératifs et la reproductibilité.

Le but de ce chapitre est de mettre en évidence les performances de cet algorithme sur des méthodes de calcul numérique. À l'issue de ce chapitre nous allons montrer que cet algorithme améliore simultanément le temps d'exécution, la reproductibilité et la convergence des calculs par l'amélioration de leur précision numérique comme suit :

1. L'amélioration de la précision numérique est illustrée sur la méthode de Simpson et la méthode de factorisation LU toutes les deux implémentées en parallèle, à la Section 6.2.
2. l'accélération de la convergence est évaluée sur la méthode de Jacobi et la méthode de puissance itérée qui utilisent l'algorithme que nous proposons par rapport aux versions de ces méthodes qui utilisent un algorithme de somme naïve ; les résultats seront présentés à la Section 6.3. Les résultats antérieurs [21] montrent que l'amélioration de la précision du calcul conduit également à accélérer la convergence des algorithmes séquentiels itératifs. Notre motivation est donc de paralléliser ces deux méthodes en nous concentrant d'abord sur la précision et en obtenant, comme résultat secondaire, une meilleure convergence.

Une dernière contribution est une comparaison du temps d'exécution de notre algorithme par rapport à un algorithme basé sur une approche similaire à celle proposée par Demmel et Hida [27]. En effet, l'algorithme de Demmel et Hida [27] a une complexité de $O(n \log n)$ dû à une étape de tri supplémentaire, par rapport à notre algorithme de sommation qui n'implique aucun tri explicite et a une complexité en $O(n)$. Les temps d'exécution des deux approches sont comparés dans les sections 6.2.1 et 6.2.2.

Nous notons que nos implémentations sont faites dans le langage de programmation C avec MPI, compilé avec MPICC 3.2, et sur un Intel i5 avec 7,7 Go de mémoire. Notons également que pour nos expériences, nous rapportons des valeurs numériques reposant sur des séparateurs de milliers. Par exemple, la valeur 1234567.89 est représentée par $1,234,567.89$.

6.2 Précision numérique

Dans cette section, nous évaluons d'abord la précision numérique de notre algorithme 10 introduit dans le chapitre 5. Dans un second temps, nous abordons le compromis entre précision numérique et temps d'exécution des algorithmes étudiés. Nous prenons en considération deux exemples, à savoir la méthode de Simpson et la méthode de factorisation LU. Pour chaque exemple, nous avons implémenté deux versions parallèles, en utilisant MPI [68]. Le premier, appelé programme original utilise des sommes naïves : pour additionner n valeurs $x_1, ..., x_n$, il calcule $(((x_1 + x_2) + x_3) + ... + x_n)$. Le deuxième programme est appelé programme précis et il est basé sur l'algorithme 10. Les expérimentations sont réalisées sur plusieurs configurations selon le nombre de processeurs.

6.2.1 Méthode de Simpson

Notre premier exemple calcule l'intégrale $\int_0^b f(x)dx$ de fonctions mathématiques f en utilisant la méthode de Simpson. La méthode de Simpson est une méthode numérique

qui approxime la valeur d'une intégrale définie d'une fonction f à l'aide de polynômes quadratiques. Nous mesurons l'efficacité de notre algorithme sur cet exemple en calculant les erreurs absolues entre les résultats des deux programmes l'original et le précis par rapport à la solution analytique de l'intégrale, comme le montre la figure 44. Nous avons intégré les fonctions suivantes $C \times \cos(x)$, $C \times (1/x^2 + 1)$ et $C \times \tanh(x)$ avec $C = 10^6$, et b compris entre $[2; 5]$. Le nombre de processeurs P est compris entre $[2; 8]$. Chaque processeur calcule une partie de l'intégrale.

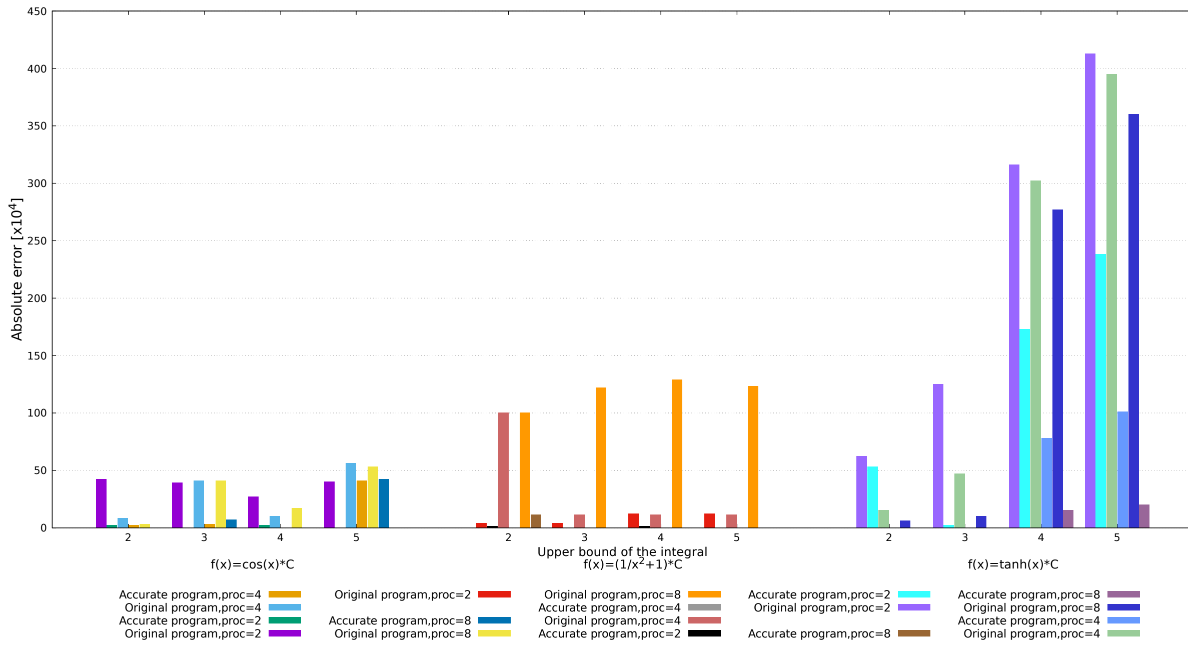

FIGURE 44 – Les erreurs absolues entre le programme original et le programme précis pour le calcul intégral de trois fonctions différentes ($C \times \cos(x)$, $C \times (1/x^2 + 1)$ et $C \times \tanh(x)$) avec le résultat analytique correspondant en faisant varier la borne supérieure de l'intégrale $b = 2, 3, 4, 5$.

Pour la première expérimentation, prenons la fonction $C \times \cos(x)$ comme exemple. Comme il est observé sur la figure 44, les erreurs absolues du programme original sont plus grandes que celles du programme précis de plusieurs ordres de grandeur. Pour mieux illustrer, considérons la valeur 3 de l'axe des abscisses x correspondant à la borne supérieure de l'intégrale. Nous remarquons que les erreurs absolues du programme original sont de $392,700.198$; $411,541.22875$ et $414,048.5725$ pour $P = 2$, $P = 4$ et $P = 8$, respectivement. Contrairement au programme précis où les erreurs absolues calculées pour le même exemple sont de $3,238.3225$; $32,419.6975$ et $77,805.5725$, respectivement. De la même manière, on note que les résultats de la deuxième fonction $C \times (1/x^2 + 1)$ et de la troisième fonction $C \times \tanh(x)$ sont similaires à ceux de la première fonction $C \times \cos(x)$. De plus, les résultats de la fonction $C \times \tanh(x)$ montrent que les résultats calculés par le programme original de Simpson avec une grande borne supérieure de l'intégrale sont ceux

qui ont des erreurs absolues plus importantes. En effet, pour $P = 2$ les erreurs absolues calculées pour la borne supérieure égale à 2,3,4 et 5 sont respectivement $621, 315.3125$; $1, 253, 797.375$; $3, 166, 450.745625$ et $4, 130, 976.360625$.

La deuxième expérience mesure le temps d'exécution (en secondes) du programme original, du programme précis et d'un autre programme basé sur le tri. Le choix de ce dernier programme est motivé par l'idée principale des algorithmes de Demmel et Hida [27]. Considérons la troisième fonction $C \times \tanh(x)$ pour $P = 8$, la figure 45 affiche le temps d'exécution en secondes pris par chaque programme (programme original, programme précis et sommation par tri) pour calculer l'intégrale de cette fonction. Les résultats montrent que le programme de sommation basé sur le tri comme les algorithmes de Demmel et Hida [27] a besoin de plus de temps pour calculer l'intégrale de la fonction $C \times \tanh(x)$. Contrairement au programme de sommation par tri, notre algorithme appelé programme précis nécessite moins de temps et un peu plus que le programme original. Par exemple, pour calculer l'intégrale de la fonction $C \times \tanh(x)$ pour $b = 2$ il faut $24s$ en utilisant le programme de sommation par tri, alors que ce calcul ne prend que $0.37s$ et $1.08s$ en utilisant respectivement le programme original et le programme précis. Il est bien connu que les sommations basées sur le tri effectué sur les valeurs à sommer sont plus précises. Par contre, ces calculs prennent plus de temps ($49s$ pour $b = 5$) comparé à notre algorithme précis où aucun tri n'est effectué ($1, 15s$ pour la même valeur de b).

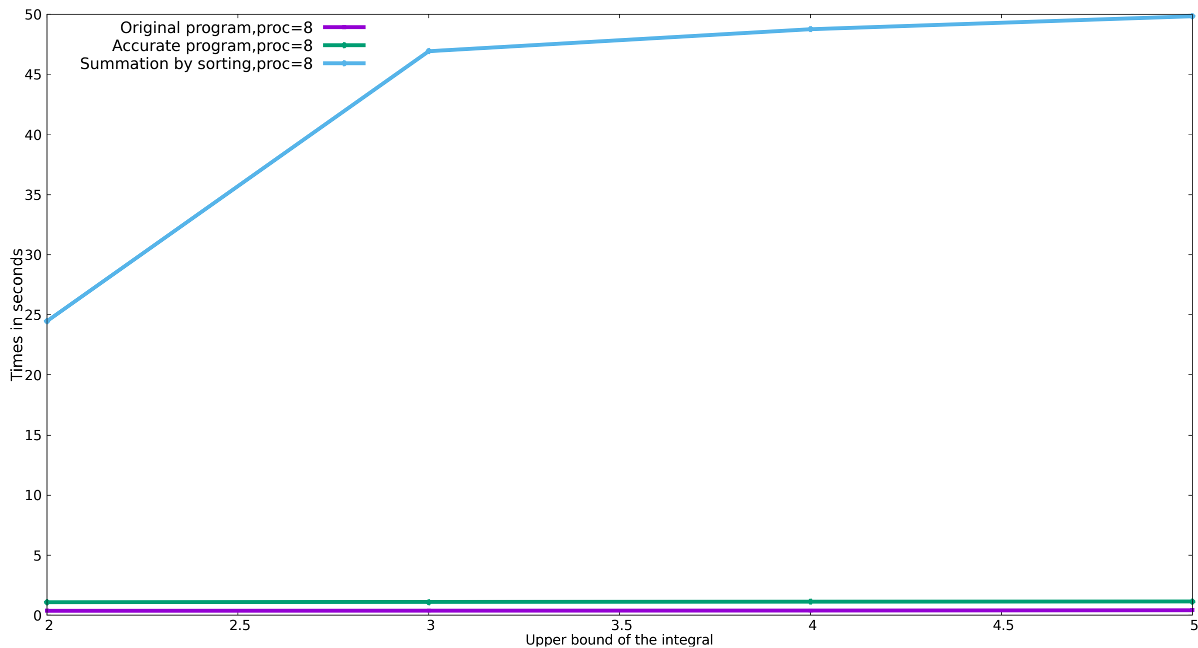

FIGURE 45 – Temps d'exécution du programme original, programme précis et le programme de sommation par tri pour le calcul intégral de la fonction $C \times \tanh(x)$.

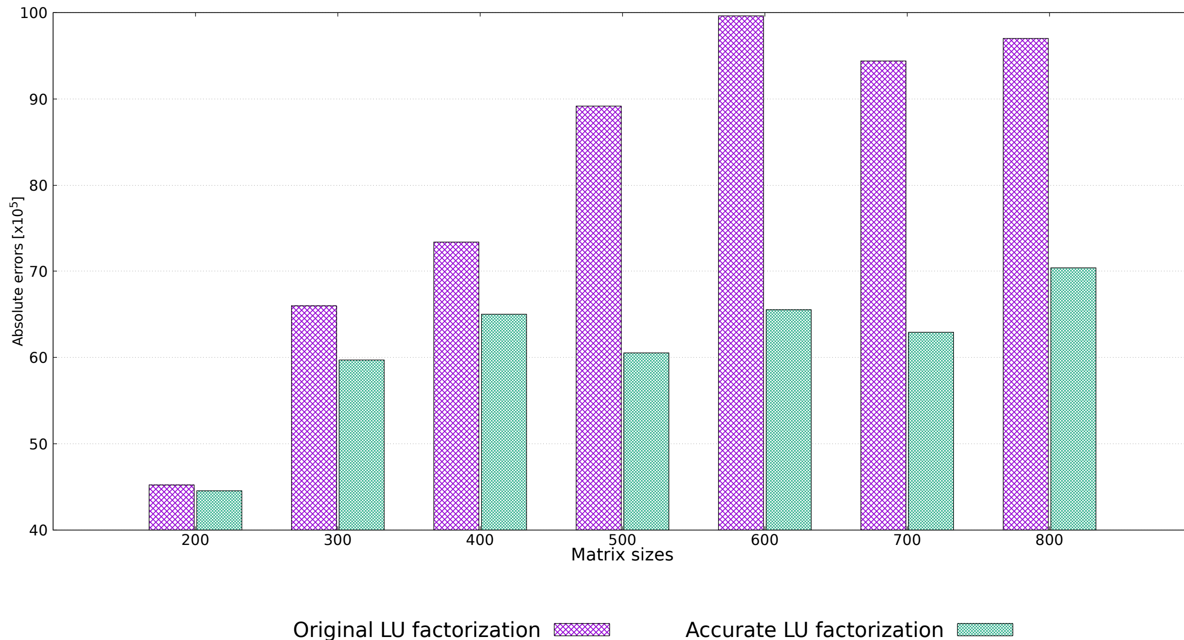

FIGURE 46 – Les erreurs absolues de la factorisation LU par l'algorithme original et précis pour des matrices de différentes tailles.

6.2.2 Méthode de Factorisation LU

Notre deuxième exemple est la méthode de factorisation LU parallèle [78]. Cette méthode consiste à réécrire une matrice A comme le produit d'une matrice triangulaire inférieure L et d'une matrice triangulaire supérieure U telle que $A = LU$. La méthode de factorisation LU est un algorithme très courant qui peut être utilisé par exemple pour résoudre des systèmes linéaires ou pour calculer le déterminant d'une matrice. Dans le cas parallèle, la matrice A est divisée en blocs de lignes et chaque processeur effectue ses calculs sur un bloc donné. Pour nos expériences, nous avons généré des matrices carrées de différentes dimensions $n \in [200, 800]$ avec un incrément de 100. Ces matrices contiennent des valeurs choisies pour introduire des sommes mal conditionnées [58]. Dans notre cas, nous considérons 30% de grandes valeurs parmi les petites et moyennes. Par petites, moyennes et grandes valeurs on entend respectivement, de l'ordre de 10^{-7}, 10^0 et 10^7. Ceci est motivé par le format simple précision de l'arithmétique flottante. Aussi, on prend des vecteurs x avec les mêmes proportions de grandes valeurs parmi les petites et moyennes que pour les matrices.

La première expérimentation consiste à comparer la précision numérique de la factorisation LU réalisée à l'aide des deux programmes l'original et le précis. Considérons une matrice A et un vecteur x. On commence par calculer le vecteur solution donné par $Ax = b$. Ce vecteur est considéré comme notre solution de référence. Ensuite, nous appliquons le programme de factorisation LU original à la matrice A avec $P = 16$ processeurs

afin d'obtenir L_{orig} et U_{orig}. De la même manière, nous factorisons la matrice A en utilisant la factorisation LU précise en L_{acc} et U_{acc}. Nous comparons les nouveaux vecteurs solutions $b_{orig} = L_{orig} \times U_{orig} \times x$ et $b_{acc} = L_{acc} \times U_{acc} \times x$ avec la solution de référence b. La figure 46 représente les erreurs absolues entre les solutions calculées après factorisation et la solution de référence. Ces expériences montrent des améliorations significatives telle que la différence entre les erreurs absolues du programme original et le programme précis sont déjà de l'ordre de 10^5 pour nos plus petites matrices ($n = 200$), cette différence atteint un ordre de 10^7 pour les matrices de plus grande taille ($n = 600$). Plus précisément, pour $n = 200$, l'erreur absolue calculée est égale à $4,521,984$ pour le programme de la factorisation LU originale et à $4,456,448$ pour le programme de la factorisation LU exacte. De la même manière, pour une matrice de taille $n = 600$, nous obtiendrons un terme d'erreur absolue égale à $9,961,472$ et $6,553,600$ pour les deux programmes de la factorisation Lu originale et exacte, respectivement. Ainsi, nous concluons de cette expérimentation que le programme précis montre son efficacité en termes de précision numérique lorsque nous manipulons de grandes matrices, c'est-à-dire lorsque différents types d'absorptions et d'annulations ont été introduits.

Dans notre deuxième expérimentation, nous allons montrer le temps d'exécution pris par chaque programme de factorisation LU pour un ensemble de matrices de taille variant de 200 à 800 avec $P = 16$. La figure 47 résume le temps d'exécution pris par chaque algorithme (programme original, programme précis et sommation par tri) pour calculer la factorisation LU.

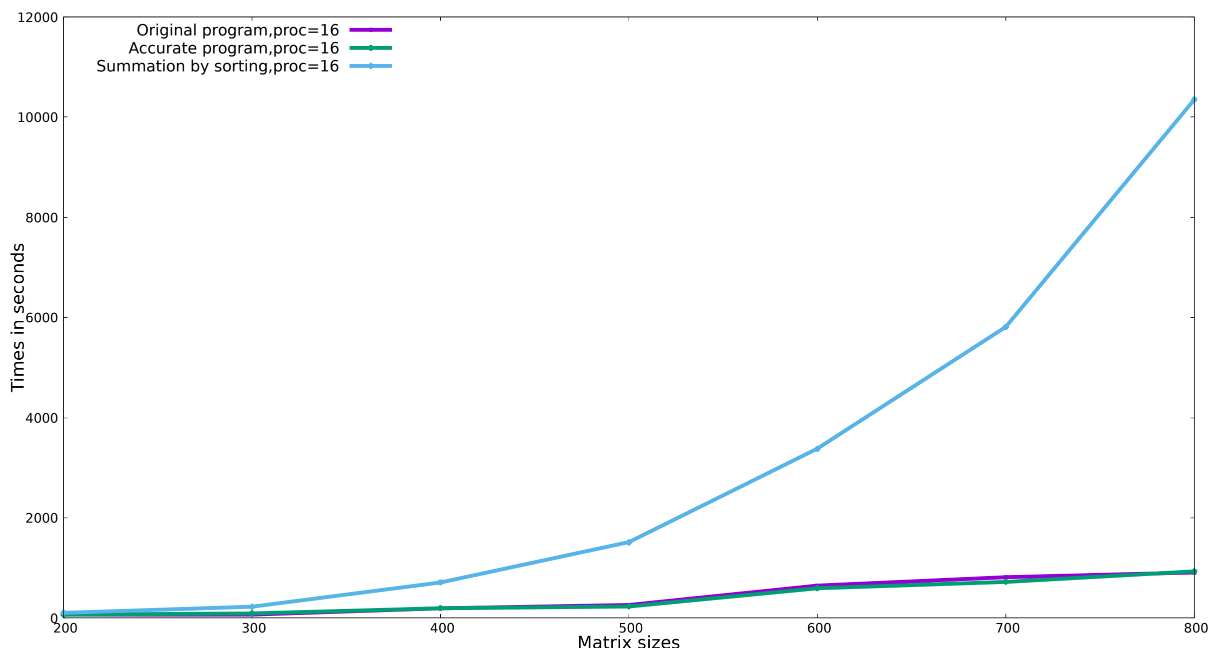

FIGURE 47 – Temps d'exécution du programme original, programme précis et le programme de sommation par tri de la factorisation LU.

On remarque sur la figure 47 que l'algorithme de sommation basé sur le tri nécessite beaucoup plus de temps pour décomposer les matrices. De plus, un programme précis nécessite beaucoup moins de temps que l'algorithme de sommation par tri et un peu plus que l'algorithme d'origine. Sauf pour certains cas, par exemple pour n=700, suite à une erreur de mesure le programme original a pris plus de temps que le programme précis. Pour mieux illustrer, prenons une matrice de taille $n = 300$. On remarque que l'exécution ne prend que $66s$ et $90s$ avec le programme original et le programme précis, respectivement, tandis que la sommation par programme de tri nécessite $224s$ pour le même calcul. Remarquons également que les temps d'exécution obtenus pour le programme de sommation par tri pour les grandes matrices sont beaucoup plus grands que ceux obtenus pour la plus petite. En effet, le temps d'exécution passe de $101s$ à environ $2h$ pour des matrices de tailles respectivement 200 et 800, en utilisant le programme de sommation par tri.

6.3 Convergence des méthodes itératives

Dans cette section, nous nous concentrons sur l'impact de la précision sur le nombre d'itérations requises par les méthodes itératives pour converger. Pour nos expérimentations, nous considérons deux méthodes itératives : la méthode de Jacobi et la méthode de la puissance itérée. Comme dans la section précédente, nous avons implémenté deux versions du même algorithme, l'original et notre version précise. Pour nos expérimentations, nous avons observé l'impact sur la convergence, en comparant leur nombre respectif d'itérations.

6.3.1 Méthode de Jacobi

La méthode de Jacobi est une méthode numérique bien connue utilisée pour résoudre des systèmes linéaires [78] de la forme $Ax = b$. Dans cette méthode, une estimation initiale x^0, est sélectionnée et est mise à jour par un processus itératif jusqu'à trouver la solution x^k du système linéaire. Plus précisément, cette méthode itère jusqu'à $|x_i^{(k+1)} - x_i^k| < \varepsilon$. Dans notre cas, la parallélisation de la méthode de Jacobi se fait selon la distribution par ligne. La méthode de Jacobi est stable lorsque la matrice A est à diagonale strictement dominante, c'est-à-dire qu'elle satisfait la propriété de l'équation (6.1).

$$\forall i \in 1, \ldots, n, \qquad |a_{ii}| > \sum_{j \neq i} |a_{ij}|. \tag{6.1}$$

Nous examinons l'impact de la précision sur la vitesse de convergence pour les systèmes de tailles 10 et 100. Bien que les systèmes choisis sont stables par rapport à l'équation (6.1).

Taille de matrice	epsilon ϵ	Nbr d'itérations du prog original	Nbr d'itérations prog précis
10	10^{-2}	5343399	5343319
	10^{-3}	7665256	7664743
	10^{-4}	9987621	9987189
	10^{-5}	12310629	12308600
100	10^{-2}	5359620	5358310
	10^{-3}	7688730	7686510
	10^{-4}	10017130	10014736
	10^{-5}	12356009	12345063

TABLE 6.1 – Nombre d'itérations des deux programmes original et précis de Jacobi.

Néanmoins, ils sont proches de l'instabilité avec $\forall i \in 1, \ldots, n, |a_{ii}| \approx \sum_{j \neq i} |a_{ij}|$. Le tableau 6.1 donne le nombre d'itérations nécessaires à la convergence des deux programmes de Jacobi (original et précis) pour des tailles de matrices égales à 10 et 100 pour différentes valeurs de ϵ. La figure 48 représente la différence entre le nombre d'itérations du programme original et ceux du programme précis.

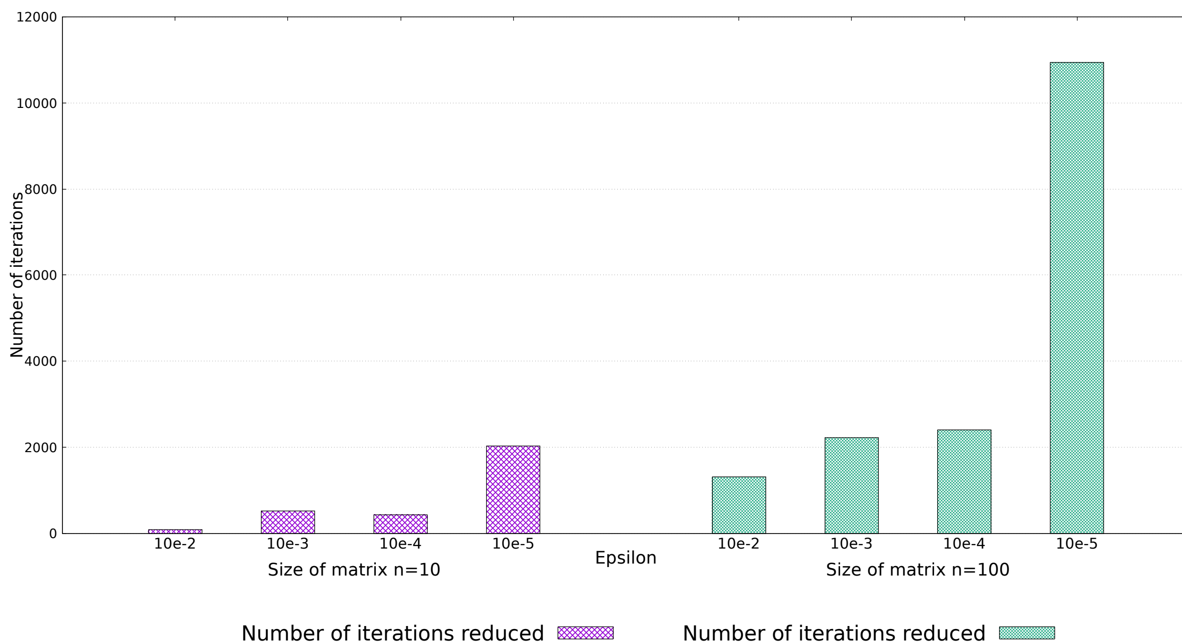

FIGURE 48 – Différence entre le nombre d'itérations nécessaires pour le programme original et le programme précis pour atteindre la convergence de la méthode de Jacobi.

Prenons le premier système de taille $n = 10$. On remarque que pour différentes valeurs de ε variant de 10^{-2} à 10^{-5}, la vitesse de convergence en nombre d'itérations passe de 80 à $2,029$. Pour le second système lorsque $n = 100$, on remarque que les différences entre le nombre d'itérations du programme original et du programme précis sont supérieures

à celles calculées pour le système $n = 10$ pour les mêmes valeurs de ε. Par exemple, la différence en nombre d'itérations pour $\varepsilon = 10^{-2}$ et $\varepsilon = 10^{-5}$ est de $1,310$ et $10,946$, respectivement. De ces deux exemples, nous concluons que les plus petites valeurs de ε sont celles qui ont une grande différence entre le programme original et le programme précis en termes de nombre d'itérations. Aussi, pour une valeur donnée de ε, le programme précis montre son efficacité sur la vitesse de convergence sur les grandes matrices par rapport aux petites.

6.3.2 Méthode de la puissance itérée

La méthode de la puissance itérée est particulièrement utilisée pour estimer numériquement la plus grande valeur propre et son vecteur propre correspondant [34, 78]. L'idée est de fixer un vecteur initial arbitraire $\mathbf{x}^{(0)}$ qui contient un seul élément non nul. Ensuite, nous construisons un vecteur intermédiaire $\mathbf{y}^{(1)}$ tel que $\mathbf{A}\mathbf{x}^{(0)} = \mathbf{y}^{(1)}$. Afin d'obtenir le vecteur $\mathbf{x}^{(1)}$, nous renormalisons $\mathbf{y}^{(1)}$ pour que l'élément maximal soit à nouveau égale à 1. Pour l'itération suivante, nous utilisons $\mathbf{x}^{(1)}$ comme vecteur sélectionné. Le processus itératif est répété jusqu'à convergence. Nous supposons que la parallélisation de la méthode de la puissance itérée se fait selon la distribution par ligne. Prenons une matrice carrée $\mathbf{A}$ de la forme :

$$A = \begin{pmatrix} d & a_{12} & \cdots & a_{1j} \\ a_{21} & d & \cdots & a_{2j} \\ \vdots & & & \vdots \\ a_{i1} & a_{i2} & \cdots & d \end{pmatrix} \tag{6.2}$$

On suppose que $a_{ij} = 0.01$ et $d \in [300.0, 500.0]$ suivant la méthodologie introduite dans [21]. La figure 49 résume la différence entre le nombre d'itérations du programme original et du programme précis de la méthode de puissance itérée. Comme il est observé sur la figure 49, le programme précis accélère la convergence de la méthode de la puissance itérée en réduisant le nombre d'itérations nécessaires pour converger. En effet, pour la matrice de taille $n = 100$ avec différentes valeurs de la diagonale et en utilisant $P = 4$ nous montrons que la différence entre le nombre d'itérations du programme original et du programme précis passe de 205 à 340.

6.4 Reproductibilité

Dans cette section, nous visons à évaluer l'efficacité de l'algorithme 10 sur l'amélioration de la reproductibilité. Les figures 50 et 51 donnent des résultats de reproductibilité pour la méthode de Simpson et la multiplication de matrices, respectivement. Au cours de

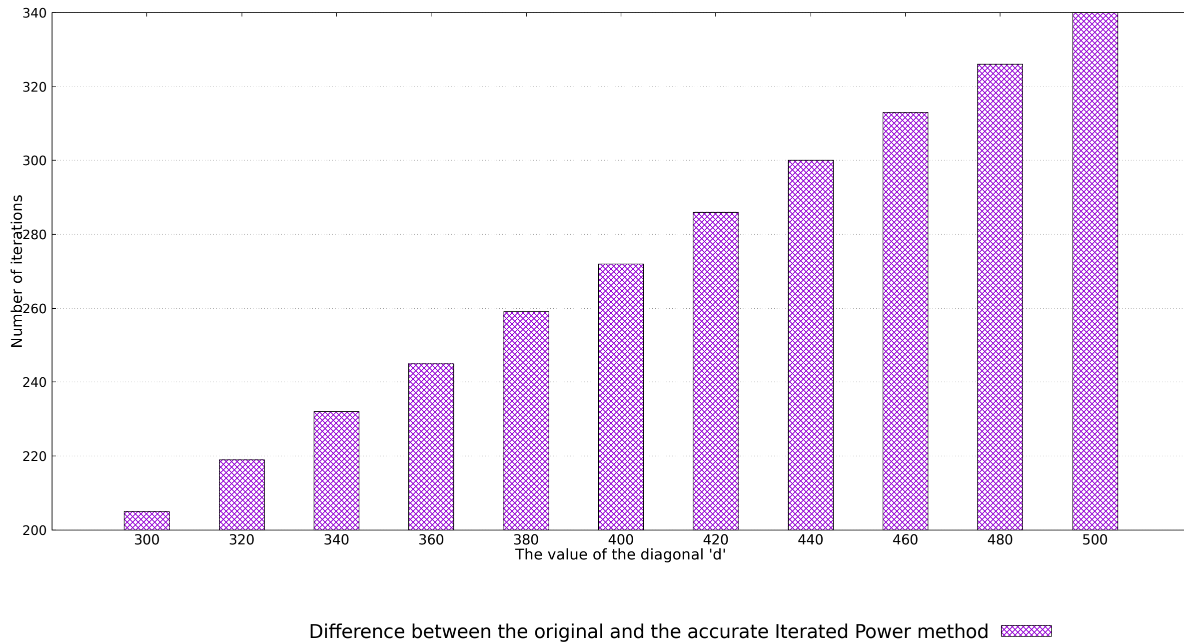

FIGURE 49 – Différence entre le nombre d'itérations de la méthode de puissance itérée originale et précise ($n = 100$, $d \in [300, 500]$ avec incrément de 20).

nos expériences, nous considérons les deux programmes : l'original et le précis de chaque méthode. Nous comparons les résultats de chacun d'eux sur plusieurs processeurs et leurs résultats respectifs avec un seul processeur.

6.4.1 Méthode de Simpson

Reprenons l'exemple de la méthode de Simpson déjà introduite dans la Section 6.2. En pratique, l'amélioration de la précision numérique améliore souvent la reproductibilité. Nous mesurons l'efficacité de notre algorithme sur cet exemple en calculant les erreurs absolues entre les résultats du programme original et du programme précis en faisant varier le nombre de processeurs de 2 à 8 et leur programme respectif avec un seul processeur, comme indiqué dans la figure 50. Considérons plusieurs fonctions mathématiques $C \times \cos(x)$, $C \times (1/x^2 + 1)$ et $C \times \tanh(x)$ avec $C = 10^6$, et b allant de $[2; 5]$. Par exemple, pour $f(x) = C \times \cos(x)$ avec $P = 2$, les résultats montrent que les erreurs absolues du programme original sont plus importantes (entre $105, 553.111787$ et $703, 687.4375$) que celles du programme précis (entre $47, 938.1875$ et $195, 067.296875$) tel qu'il est observé sur la figure 50. Aussi, on peut observer sur la figure 50 que les résultats des intégrales calculées par le programme original et exécutées sur un grand nombre de processeurs $P = 8$ sont ceux qui ont une plus grande erreur absolue. À titre d'exemple, les erreurs absolues calculées pour la fonction $C \times \tanh(x)$ avec $P = 8$ sont comprises entre $23, 122.109375$ et $3, 637, 171.1875$ alors que les erreurs absolues calculées pour le même exemple avec $P = 2$ sont entre $87, 960.921875$ et $1, 221, 541.8750$.

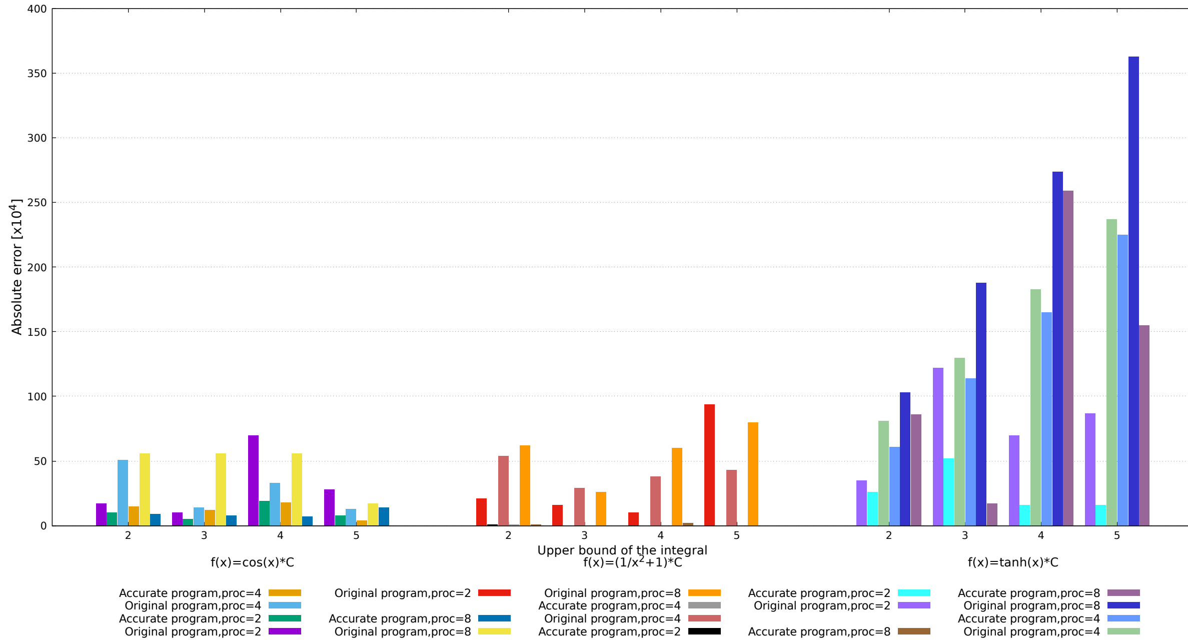

FIGURE 50 – La reproductibilité des calculs d'intégrales utilisant la méthode de Simpson du programme original et de celui précis en fonction du nombre de processeurs.

6.4.2 Multiplication de matrices

Le calcul de la multiplication de matrices basé sur l'addition et la multiplication à virgule flottante qui sont des opérations non-associatives est sujet à des problèmes de précision. De plus, l'ordre d'exécution des opérations arithmétiques sur des architectures parallèles différentes ou même similaires étant différent, la reproductibilité des résultats n'est pas garantie. Dans ce contexte, nous abordons le problème de la reproductibilité dans le cas de la multiplication de matrices. Pour paralléliser cette méthode, nous supposons que chaque matrice est divisée en sous-matrices de taille n/P. Pour nos expériences, nous considérons des matrices carrées de différentes dimensions $n \in [200, 800]$ avec un incrément de 200. Ces matrices contiennent une variété de valeurs à virgule flottante choisies avec une différence de magnitude. Plus précisément, elles sont constituées de 50% de grandes valeurs (de l'ordre de 10^7) parmi petites (de l'ordre de 10^0) et moyennes (de l'ordre de 10^{-7}). La figure 51 représente le pourcentage de précision calculé entre le programme original et le programme précis réalisé à l'aide de $P = 8$ et leur résultat respectif à l'aide d'un seul processeur. Les résultats montrent que pour différentes tailles de matrice, le pourcentage de précision du programme original varie de 3% à 13%. Contrairement au programme original, le pourcentage lié au programme précis est égal à 100% pour chaque matrice ce qui confirme l'utilité de l'algorithme 10.

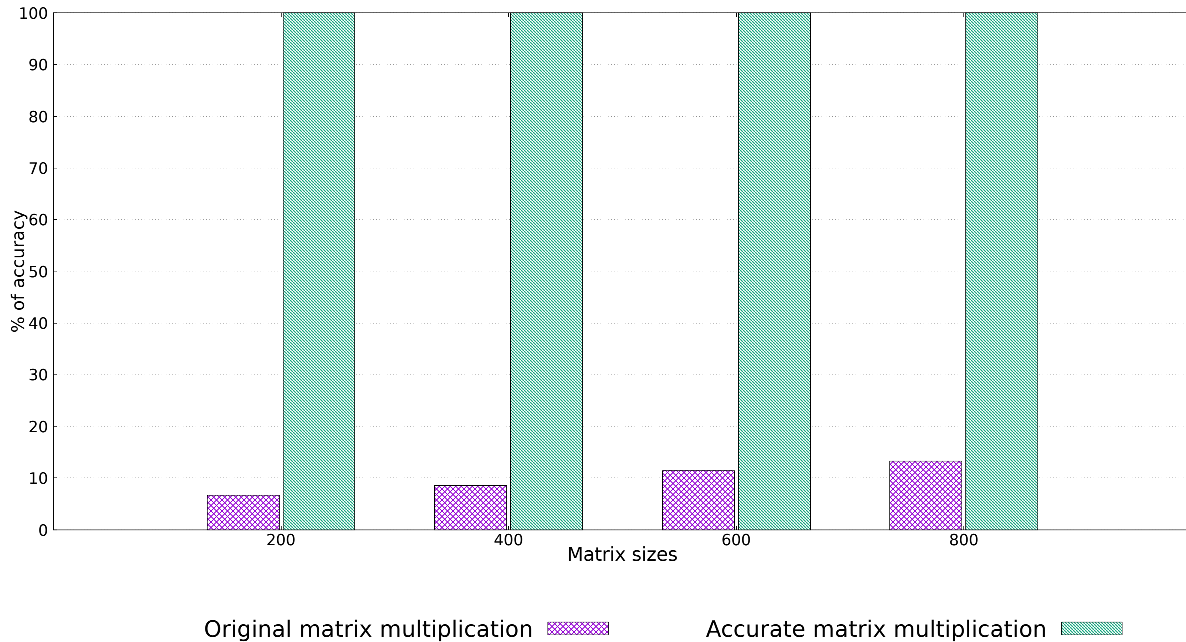

FIGURE 51 – La reproductibilité de la multiplication matrice-matrice pour différentes tailles de matrices.

6.5 Conclusion

Dans ce chapitre, nous nous sommes intéressés à l'impact de la précision numérique à la fois sur la reproductibilité et la vitesse de convergence des méthodes de calculs numériques. L'idée est d'étudier l'impact de l'algorithme 10 sur la précision numérique des calculs faisant intervenir des sommes.

Nous avons expérimenté notre algorithme sur plusieurs méthodes numériques en comparant les programmes original et précis de chacune d'entre elles. Les expériences montrent l'efficacité de notre algorithme sur l'amélioration de la reproductibilité. Aussi, les résultats obtenus montrent l'efficacité de notre algorithme pour réduire le nombre d'itérations nécessaires aux méthodes itératives pour converger. Plus précisément, le programme précis converge plus rapidement que le programme original sans perte de précision.

Bien que, par manque de temps, nous n'ayons pas affiné cet algorithme, il serait judicieux d'ajouter une phase de test après la ligne 4 de l'algorithme 10, c'est-à-dire avant d'additionner la valeur s_i à la cellule appropriée du tableau sum_by_exp. En effet, si nous avons un grand ensemble de valeurs à additionner avec le même exposant, le résultat produit peut avoir un exposant plus grand que celui initial. Par conséquent, une légère perte de précision peut être provoquée lors des calculs des sommes locales.

Chapitre 7

Application aux réseaux de neurones

Contents

7.1 Introduction **90**
7.2 État de l'art **91**
7.3 Réseaux de neurones **92**
7.3.1 Modèle mathématique 92
7.3.2 Types des réseaux de neurones 94
7.4 L'apprentissage des réseaux de neurones **95**
7.4.1 L'apprentissage supervisé 95
7.4.2 L'apprentissage non supervisé 96
7.4.3 Contribution 97
7.5 Exemple **97**
7.5.1 Mécanisme de l'algorithme 97
7.5.2 Résultats préliminaires 98
7.6 Conclusion **99**

7.1 Introduction

Plusieurs applications liées à la précision numérique existent dans de nombreux domaines tels que le HPC [22, 69] ou les systèmes embarqués (robotique, l'électronique, voitures autonomes) [38, 9]. Dans ce chapitre, nous allons montrer son utilité dans un nouveau domaine peu exploré, à savoir l'apprentissage des réseaux de neurones utilisés par exemple dans le domaine médical pour diagnostiquer des maladies. Dans ce chapitre, nous nous intéressons plus particulièrement à la classification d'images où la précision numérique sera mesurée par le taux de reconnaissance obtenu. L'objectif de notre étude est de montrer à quel point une somme précise peut impacter les résultats d'un algorithme d'apprentissage par rapport à ce même algorithme basé sur une somme récursive. Ce chapitre est organisé de la manière suivante. La section 7.2 Cette section constitue un aperçu de quelques travaux reliant la précision numérique aux réseaux de neurones. La section 7.3 est consacrée aux réseaux de neurones et en présente les éléments essentiels : types de réseaux, fonctions de transfert, etc. À la section 7.4, nous présentons les deux ap-

proches d'apprentissage et nous expliquons l'intérêt de notre contribution. Nous détaillons l'algorithme d'apprentissage étudié ainsi que les résultats obtenus à la section 7.5.

7.2 État de l'art

Dans l'apprentissage automatique des réseaux de neurones, les poids des neurones sont souvent représentés avec plus de précision de ce qui est nécessaire à leurs entraînements. Cependant, ces surestimations en précision numérique peuvent avoir plusieurs impacts notamment sur l'empreinte mémoire, le temps de calcul, le déploiement sur du matériel embarqué tels que les FPGA et les microcontrôleurs, etc. Une solution proposée pour pallier ces problèmes est la compression de ces réseaux de neurones. En effet, les réseaux de neurones sont souvent compressés à l'aide d'outils conçus à cet effet ou par le biais de méthodes qui ont été mises en place que nous allons détailler ci-après.

Commençons par un travail récent [52] qui décrit l'outil *Condensa*. Pour réduire l'empreinte mémoire, *Condensa* propose de mettre à zéro les poids individuels et à réduire la latence d'inférence en élaguant les blocs $2D$ de poids non nuls pour le cas d'un réseau de traduction de langue.

La compression des réseaux de neurones a été également réalisée à l'aide des deux méthodes de quantization et d'élagage par poids, qui réduisent à la fois la taille du modèle et le temps d'inférence avec une précision comparable. La quantization d'un modèle consiste à représenter les paramètres d'un modèle avec des valeurs ayant moins de bits. Citons [35, 83] qui portent sur la quantization et qui visent à réduire la complexité des réseaux de neurones et à faciliter leur déploiement potentiel sur du matériel embarqué le cas des FPGA. Quant à l'élagage d'un modèle [3, 13], il consiste à effacer les paramètres redondants, ainsi l'empreinte mémoire et le temps de calcul peuvent être considérablement réduits.

Récemment, plusieurs travaux de recherche ont porté sur une méthode de compression basée sur l'utilisation de nouvelles arithmétiques telle que l'arithmétique à virgule fixe [82]. Cette méthode est basée sur le réglage de la précision numérique [16, 41] qui permet aux compilateurs de sélectionner les formats les plus appropriés au poids des neurones. Ainsi, il est possible de réduire l'empreinte mémoire et d'utiliser moins de bande passante lors des communications dans le cas d'applications distribuées.

Le réglage de précision est également d'un grand intérêt pour l'arithmétique en virgule flottante [2, 63]. Le principe consiste à optimiser les types de données en associant un nouveau type approprié à chaque poids (simple, double ou quadruple selon la norme IEEE-754). Cette méthode effectue une analyse en avant et en arrière pour définir le type optimal de chaque neurone [48]. Le réseau de neurones obtenu est alors dit optimisé, i.e. avec des types de données de tailles plus petites mais qui sera équivalent au réseau de départ.

Nous finissons par une autre méthode qui permet non seulement de réduire les erreurs de calcul mais également d'accélérer le temps de calcul. Il s'agit de faire appel aux algorithmes de calcul précis soit pour le cas de la multiplication [59] ou de la sommation [8, 7]. C'est ce que nous voulons montrer à l'issue de ce chapitre en appliquant nos algorithmes de sommes sur un exemple d'apprentissage d'un réseau de neurones.

7.3 Réseaux de neurones

Les réseaux de neurones sont très utilisés en apprentissage automatique. Ils constituent une famille d'algorithmes qui visent à résoudre les problèmes de l'intelligence artificielle. Leur principe est de reconnaître les relations sous-jacentes dans un ensemble de données grâce à un processus qui imite le fonctionnement du cerveau humain. En ce sens, ces réseaux de neurones également appelés réseaux de neurones artificiels *(ANN)* ou réseaux de neurones simulés *(SNN)* font référence à des systèmes de neurones de nature organique. Dans cette section, nous présentons dans un premier temps l'architecture d'un réseau de neurones et son modèle mathématique puis nous allons classer les réseaux de neurones existants en plusieurs types.

Architecture d'un réseau de neurones

Un réseau de neurones est un graphe orienté de plusieurs neurones, généralement organisés sous forme de couches de plusieurs noeuds : une couche d'entrée, une ou plusieurs couches cachées et une couche de sortie comme illustré à la figure 52.

Chaque noeud est connecté à un autre et possède un poids w (de l'anglais weight), de telle manière que chaque sortie de neurone sert d'entrée à un autre neurone. La propagation du signal et le traitement de l'information le long d'un réseau dépendent du type de réseau étudié. Le nombre de neurones et la façon dont ils sont interconnectés déterminent l'architecture (topologie) du réseau de neurones, c'est ce que nous allons voir dans la section 7.3.2.

7.3.1 Modèle mathématique

Soient $e_i \in \mathbb{R}$, $1 \leq i \leq n$ les entrées d'un neurone, $w_{j,i} \in \mathbb{R}$, $1 \leq i \leq n$ ses poids et b_j un biais. D'un point de vue mathématique, un neurone est essentiellement constitué d'un intégrateur [70] qui effectue la somme pondérée donnée par $\sum_{i=1}^{n} w_{j,i} \times e_i + b_j$ des valeurs reçues en entrées. Le résultat de cette somme est ensuite transformé par une fonction de transfert appelée f qui produit la sortie R du neurone comme le décrit la figure 53. Le vecteur $e = (e_1 e_2 ... e_n)^T$ correspond aux entrées du neurone, quant aux poids des neurones, ils sont représentés par le vecteur $w = (w_{j,1} w_{j,2} ... w_{j,n})^T$. La fonction qui se charge de transférer l'activation et de produire la sortie du neurone est essentielle lors de

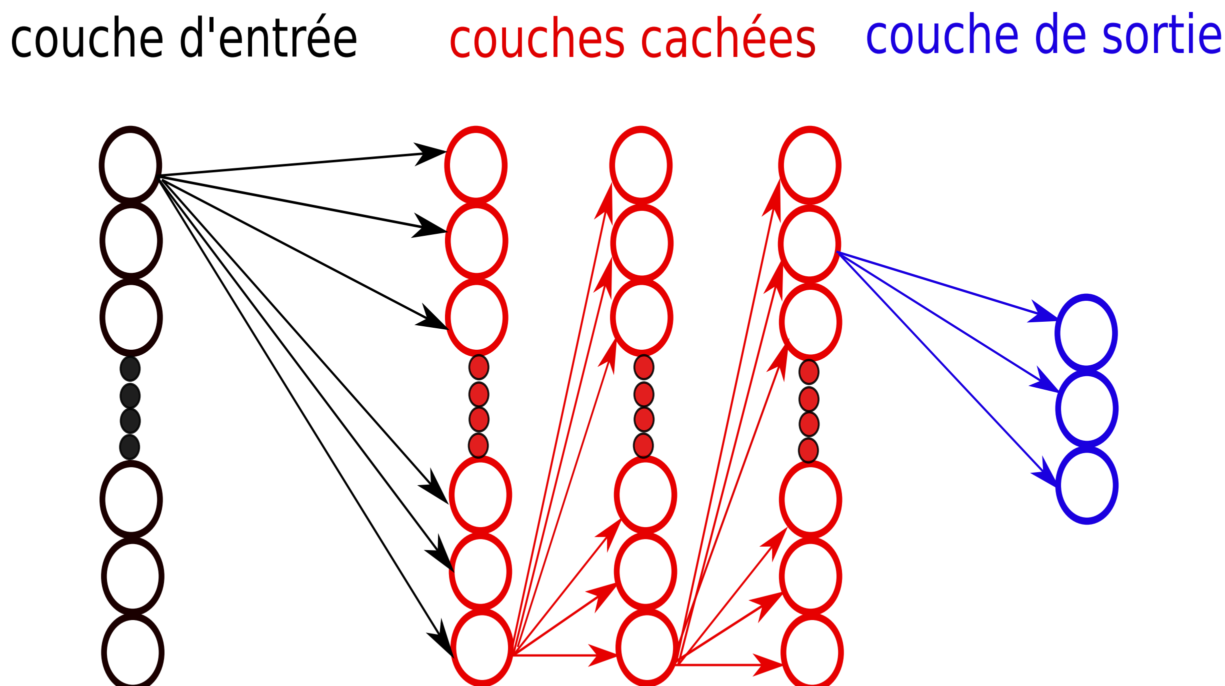

FIGURE 52 – Réseau de neurones artificiels.

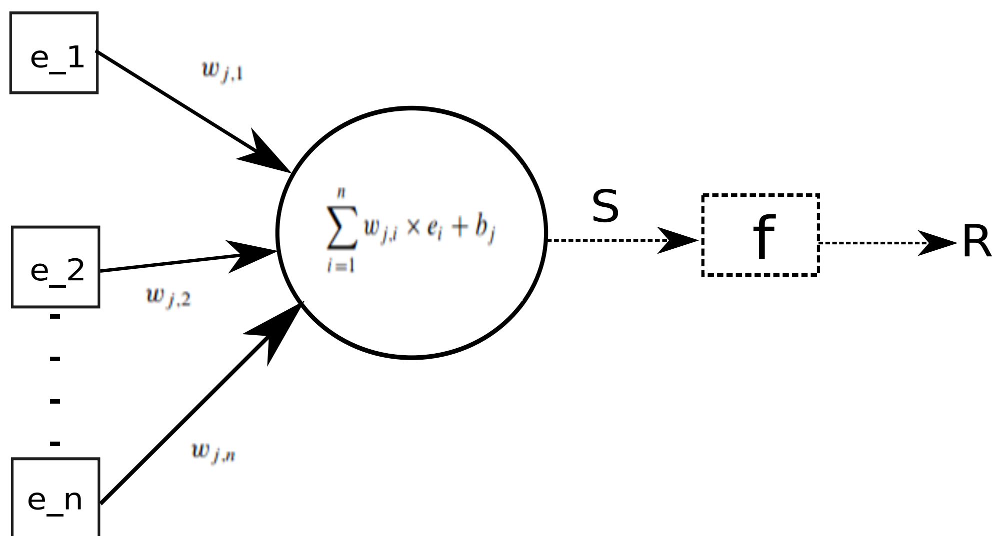

FIGURE 53 – modèle d'un neurone.

la conception d'un réseau de neurones. Plusieurs fonctions de transfert existent et il est important de choisir soigneusement cette fonction afin d'avoir un réseau ayant un taux de reconnaissance élevé. La fonction peut varier selon le domaine d'application, comme les classifications de textes, la reconnaissance d'images, etc. Le tableau 7.1, regroupe les définitions et les processus de fonctionnement d'un ensemble de fonctions de transfert.

Fonction de transfert	Définition	Description
Rectified Linear Unit Relu	R = max(0,S)	• Chaque valeur négative reçue en entrée est remplacée par un zéro. • L'une des fonctions les plus utilisées dans les algorithmes de rétropropagation.
Linéaire	$R = S$	• Chaque entrée est affectée en sortie.
Sigmoid	$R = \frac{1}{1+exp^{-n}}$	• Elle normalise la sortie de chaque neurone. • Chaque valeur en entrée est remplacée par une valeur comprise entre 0 et 1. • Utilisée pour la prédiction de probabilité.
Tanh	$R = \frac{\exp^{n} - \exp^{-n}}{\exp^{n} + \exp^{-n}}$	• Similaire à la fonction *Sigmoid* en terme de normalisation des sorties de neurones. • Une valeur comprise entre -1 et 1 est générée pour toute valeur en entrée.

Table 7.1 – Fonctions de transfert $R = f(S)$.

7.3.2 Types des réseaux de neurones

Les réseaux de neurones peuvent être classés en des types différents, correspondant à des utilisations différentes. Bien qu'il ne s'agit pas d'une liste complète de ces types, nous allons décrire dans ce qui suit ceux qui sont couramment utilisés.

Les réseaux de neurones feed-forward

Ce sont des réseaux de neurones artificiels où les connexions entre les noeuds sont unidirectionnelles. Les réseaux de neurones feed-forward ont été les premiers types de réseaux de neurones artificiels inventés et sont les plus simples à concevoir. L'information se déplace au sein de ces réseaux dans une seule direction, vers l'avant (pas de boucles), d'abord à travers les noeuds d'entrée, puis à travers les noeuds cachés et enfin à travers les noeuds de sortie. Ce type de réseaux est utilisé par exemple dans la reconnaissance faciale.

Les réseaux de neurones feed-back

Les réseaux de neurones feed-back également appelés réseaux récurrents sont des réseaux qui comportent des connexions cycliques entre les noeuds. Ils sont principalement exploités lors du traitement de données séquentielles [11] (données liées par des relations chronologiques) pour faire des prédictions sur les résultats futurs, tels que les prévisions boursières ou les prévisions de ventes. Ces réseaux sont utilisés dans l'apprentissage profond. Leur motivation principale est de s'appuyer sur des informations d'entrées antérieures pour influencer l'entrée et la sortie actuelles. Plus précisément, ces réseaux mémorisent la sortie d'une couche pour servir à mieux prédire les prochaines entrées. Un exemple

d'application de ces réseaux consiste en la suggestion automatique pour le traitement du texte.

Les réseaux de neurones convolutionnels

Un réseau de neurones convolutionnel ou *CNN* peut être considéré comme un réseau feed-forward avec plus de couches, la puissance de ce type de réseaux provient d'une couche convolutive [61]. Un *CNN* est un réseau de neurones d'apprentissage en profondeur conçu pour traiter des tableaux structurés de pixels, le cas des images. Ces réseaux ont montré des résultats très efficaces dans la reconnaissance d'images, de vidéos et en vision par ordinateur. Les réseaux de neurones convolutionnels sont sensibles aux variations de données. Par exemple, pour une image présentée en entrée, le réseau convolutionnel est capable de détecter toute les caractéristiques liées à cette image (formes, couleurs, etc.).

7.4 L'apprentissage des réseaux de neurones

L'apprentissage d'un réseau de neurones se fait par le biais d'un algorithme itératif [17] ou par l'utilisation de l'une des bibliothèques conçues à cet effet [1]. Le but de l'apprentissage consiste à bien entraîner un réseau donné sur une base de données fournie en entrée. L'idée de ces algorithmes est de corriger les poids des neurones de façon à se rapprocher de la solution attendue. En résumé, à l'aide des algorithmes d'apprentissage, un réseau de neurones est capable de prédire de nouvelles données à partir d'exemples qui ont été analysés rigoureusement. L'apprentissage d'un réseau de neurones nécessite trois étapes :

1. Choisir une architecture du réseau : faire le choix du type de réseau et de la fonction de transfert à utiliser selon le problème à résoudre ;
2. Choisir les meilleurs poids de neurones afin d'accélérer le processus d'entraînement en consommant moins de temps et d'énergie ;
3. Faire appel à un algorithme d'apprentissage selon le mode choisi.

Nous pouvons distinguer deux approches d'apprentissage des réseaux de neurones : l'apprentissage supervisé où l'algorithme s'entraîne sur des données étiquetées et l'apprentissage non supervisé où les données ne sont pas étiquetées.

7.4.1 L'apprentissage supervisé

Les algorithmes d'apprentissage supervisé sont couramment utilisés. Ces algorithmes reçoivent pour chaque entrée une valeur attendue appelée également cible. Ils apprennent à partir de ces données étiquetées fournies en entrée. Prenons l'exemple de la base de données $f$ashion-mnist qui comporte des images de vêtements (voir figure 54). Pour chaque image

1. https ://le-datascientist.fr/machine-learning-open-source

une cible est associée, c'est-à-dire définir si cette image est une sandale ou une veste, etc. Cette approche d'apprentissage nécessite une intervention humaine afin d'étiqueter toutes les images, une tâche qui s'avère difficile dans le cas des bases de données contenant des milliers d'images. Nous nous intéressons dans le reste de ce chapitre à un algorithme basé sur cette approche, appelé l'algorithme de *rétropropagation* [72, 73].

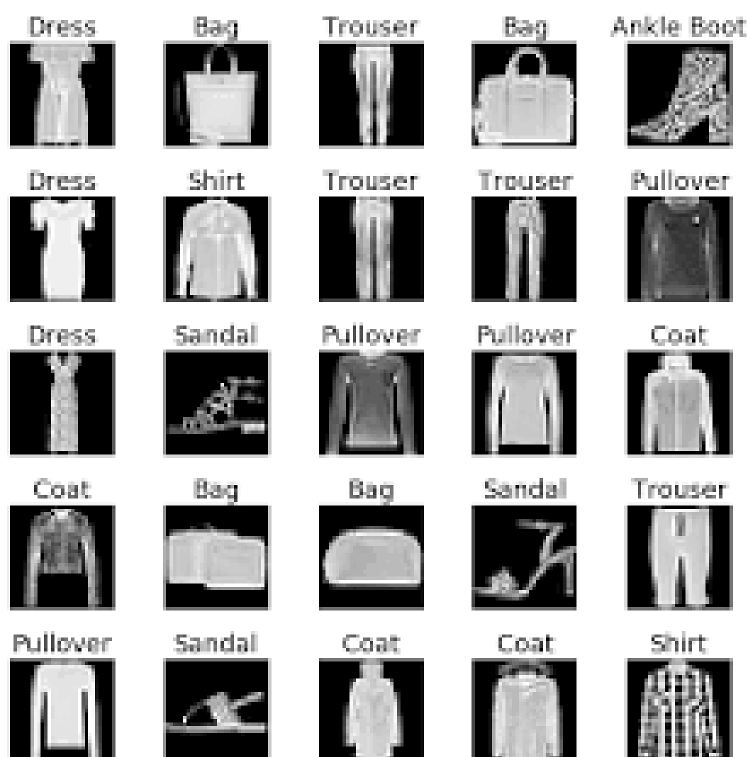

FIGURE 54 – Aperçu des images de la base de données *fashion_mnist*.

7.4.2 L'apprentissage non supervisé

L'apprentissage non supervisé est une approche utilisée généralement dans le domaine médical ou la finance. Les algorithmes d'apprentissage non supervisé s'entraînent sur des données où les cibles ne sont pas spécifiées. Deux méthodes peuvent être utilisées dans ce cas pour regrouper les données. Une première, la méthode de partitionnement, qui à partir d'un ensemble d'entrées, regroupe les données en fonction de leurs similitudes ou différences pour former des groupes adéquats. La deuxième méthode est celle basée sur les relations existantes entre les données avant de créer des partitions.

7.4.3 Contribution

Comme nous l'avons souligné à la section 7.4, l'apprentissage d'un réseau de neurones se fait par appel à un algorithme qui ajuste les poids des neurones. Ainsi, l'opération d'addition intervient à chaque itération afin de mettre à jour les poids des neurones par le terme d'erreur calculé. Dans ce même contexte, nous voulons intégrer notre algorithme de somme [7, 8] pour effectuer cette tâche au lieu d'utiliser un algorithme de somme récursive. Nos objectifs sont les suivants :

- Accélérer le processus d'entraînement : pour une même base de donnée fournie en entrée, l'algorithme d'apprentissage basé sur notre algorithme de somme nécessite moins d'itérations pour être entraîné contrairement à ce même algorithme basé sur une somme récursive qui prend plus d'itérations,
- Le taux de reconnaissance sera plus élevé dans le cas où l'algorithme de somme utilisé est précis 9.

7.5 Exemple

Rappelons que notre objectif est d'améliorer la précision numérique dans le contexte des réseaux de neurones. Nous envisageons d'intervenir dans la phase d'apprentissage, plus précisément au moment d'ajuster des poids des neurones en remplaçant l'algorithme de somme récursive par notre algorithme de somme précise (voir chapitre 5). Considérons l'algorithme de rétropropagation [2] utilisé pour les problèmes de classification. L'idée principale de l'approche de rétropropagation est d'entraîner les poids d'un signal d'entrée dans un réseau de neurones pour produire un signal de sortie attendu. Cette sortie est référencée dans la base de données d'apprentissage fournie en entrée.

7.5.1 Mécanisme de l'algorithme

L'entraînement d'un réseau de neurones par l'algorithme de rétropropagation nécessite la définition de plusieurs fonctions. Dans un premier temps, nous propageons vers l'avant une entrée pour obtenir une sortie. Cette étape est réalisée par la fonction *forward_propagate()*. La fonction *forward_propagate()* regroupe le calcul des sommes pondérées $\sum_{i=1}^{n} w_{j,i} \times e_i + b_j$ pour activer ou désactiver les neurones et l'appel à la fonction de transfert *sigmoid* qui se charge de transférer cette activation pour produire la sortie du neurone.

Dans un second temps, nous calculons l'erreur définie par l'équation (7.1) entre la sortie calculée et une sortie attendue (connue).

2. https ://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

$$erreur = (R - C) * transfert_derivative(R) \tag{7.1}$$

Avec,

— R la sortie du neurone ;

— C la cible ou valeur attendue ;

— *transfert_derivative(R)* : la dérivée de la fonction de transfert *sigmoid* $= R * (1 - R)$ appliquée à la sortie du neurone R.

Enfin, nous rétro-propageons les termes d'erreurs calculés pour modifier les valeurs initiales des poids des neurones par une fonction appelée *backward_propagate_error()*. Dans cette dernière phase, il est bien connu que si nous utilisons l'algorithme de somme récursive pour mettre à jour les poids des neurones, des erreurs d'arrondi peuvent être générées suite à l'opération d'addition. Par conséquent, le taux de reconnaissance sera faible et le processus d'entraînement nécessitera plusieurs itérations pour que les poids des neurones soient ajustés au bonnes valeurs. C'est ce que nous voulons améliorer par notre contribution.

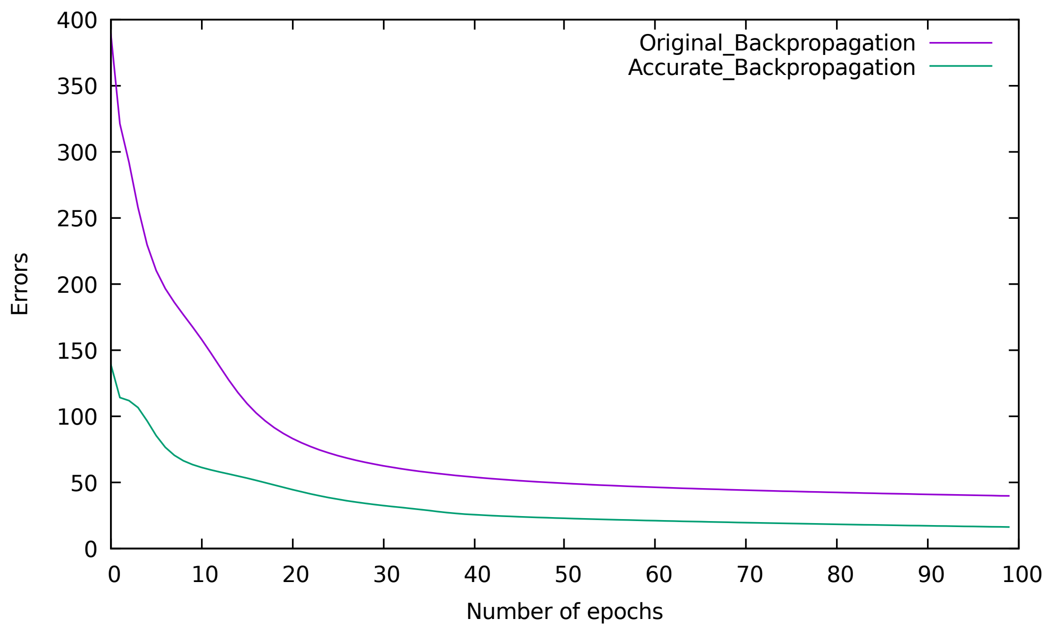

FIGURE 55 – Erreurs entre les sorties propagées en avant et les sorties attendues.

7.5.2 Résultats préliminaires

Pour nos expérimentations, nous avons testé cette technique sur un problème de classification avec 3 classes de sortie. Il s'agit de la classification des graines de blé [12]. La

base de données contient 210 échantillons, à chaque graine une cible est associée sur un ensemble de trois cibles possibles (1 pour valid, 2 pour mismatched et 3 pour missing). Nous avons reproduit les résultats obtenus en utilisant l'algorithme de sommation récursive et en parallèle nous avons refait les mêmes calculs en se basant sur l'algorithme de somme précise (voir algorithme 9, chapitre 5).

La figure 55 est une comparaison en terme d'erreurs calculées entre le programme original de rétropropagation (basé sur une somme récursive) et le programme précis (basé sur une somme précise). Nous constatons que les erreurs calculées entre les entrées d'un réseau et les cibles enregistrées dans la base de données sont plus petites dans le cas où l'algorithme qui se charge d'ajuster les poids des neurones est précis.

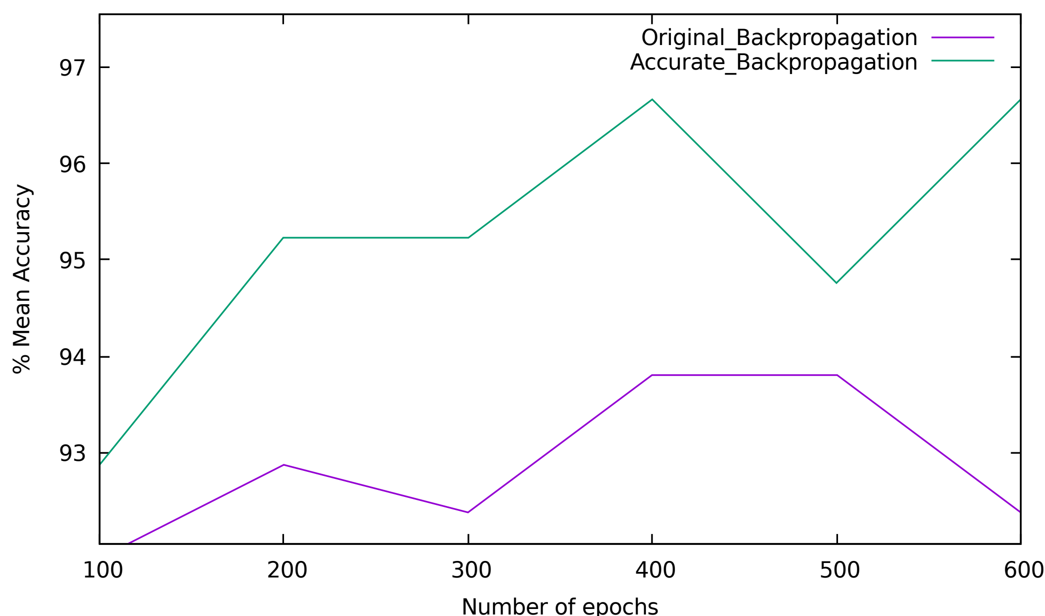

FIGURE 56 – Pourcentage de reconnaissance en fonction du nombre de cycles.

Quant à la figure 56, elle représente le taux de reconnaissance des graines de blé en fonction du nombre de cycles. Nous constatons que pour un même nombre de cycles le taux de reconnaissance obtenu par le programme de rétropropagation précis est plus important (varie de 92% à 96%) que celui obtenu par le programme original (varie de 91% à 93%).

7.6 Conclusion

Nous avons abordé dans ce chapitre quelques notions liées aux réseaux de neurones. Nous nous sommes concentrés sur les techniques d'apprentissage des réseaux de neurones et nous avons expliqué l'intérêt d'intégrer notre algorithme de somme à ce niveau. Nous

avons appliqué l'algorithme de rétropropagation sur un exemple simple de classification de graines de blé pour montrer l'impact que peut avoir une somme précise dans l'entraînement d'un réseau de neurones. Les résultats ont montré que dans le cas de classification d'images et pour une même base de données, l'algorithme de rétropropagation basé sur un l'algorithme de somme précise (algorithme 9, chapitre 5) nécessite moins d'itérations pour entraîner le réseau de neurones sur l'ensemble de la base de données. De plus, le taux de reconnaissance est plus élevé, par conséquent, le réseau de neurones est performant et a une capacité d'analyser plus de données.

Chapitre 8

CONCLUSION ET PERSPECTIVES

8.1 Conclusion

Le travail présenté dans cette thèse décrit des contributions distinctes qui ont un but commun, l'amélioration de la précision numérique et la reproductibilité des codes parallèles.

Nous avons étudié au chapitre 3 l'amélioration de la précision numérique par une transformation complète de programme. La démarche que nous avons suivie consiste à spécialiser le code de chaque processeur en fonction de ses données d'entrée avant d'appliquer les transformations automatiques implémentées dans l'outil appelé `Salsa`. Nous avons expérimenté cette approche sur un code parallèle développé en `MPI`. Ce code modélise la propagation de la chaleur sur une grille. Dans le cadre de nos expérimentations, nous avons comparé le programme original de la propagation de chaleur sur une grille et celui optimisé, c'est-à-dire celui auquel nous avons appliqué la spécialisation de code et les transformations automatiques de l'outil `Salsa`. Les résultats ont confirmé une amélioration de la précision numérique mais aussi un gain en temps d'exécution.

Au chapitre 4, notre objectif était d'améliorer la précision numérique dans le cas de la résolution des systèmes linéaires. Pour ce faire, nous avons développé une nouvelle technique qui s'appuie sur une analyse statique par interprétation abstraite. La technique est composée de plusieurs étapes d'abstractions pour détecter l'ordre des scalaires qui pourrait minimiser les erreurs commises durant un certain calcul, donc optimiser la précision numérique. Une fois l'ordre des scalaires calculé, un algorithme est utilisé pour construire les partitions du même signe et ordre de grandeur. Nous avons testé les performances de cette technique sur un exemple fondamental en mécanique. Bien que la technique proposée parvient à détecter et créer des partitions de même signe et ordre de grandeur, nous avons remarqué que ces partitions ont une limite en terme de longueur. Plus précisément, les partitions créées ne contiennent pas de longues séquences de valeurs. À ce niveau, nous nous sommes demandés comment allons nous choisir un algorithme de somme approprié pour chaque partition.

En réponse à cette question, nous avons proposé au chapitre 5 deux nouveaux algo-

rithmes de somme pour additionner efficacement n nombres flottants. Nous avons prouvé l'efficacité de nos algorithmes par rapport à l'algorithme de somme récursive au travers différentes expérimentations. De plus, nous avons comparé les résultats de nos algorithmes par rapport à ceux obtenus par l'algorithme de Demmel et Hida [27]. Nous avons constaté sur huit ensembles de données, générés différemment selon le format simple précision, que nos algorithmes calculent les sommes, en précision de travail, sans nécessiter des accumulateurs de plus haute précision. Quant à la complexité, nos algorithmes ont une complexité linéaire à la complexité de l'algorithme de somme récursive.

Au chapitre 6, nous avons testé l'efficacité de nos algorithmes sur des méthodes de calcul numérique. Nous avons choisi un ensemble de cinq méthodes représentatives qui sont la méthode de Simpson, la méthode de Factorisation LU, la multiplication des matrices, une méthode pour calculer la plus grande valeur propre et la méthode de Jacobi. En plus des deux propriétés de précision numérique et de reproductibilité, des accélérations importantes sont obtenues en terme de nombre d'itérations pour converger vers les solutions des méthodes itératives.

Nous avons introduit au chapitre 7 des travaux qui sont en cours d'élaboration. Nous avons voulu montrer l'importance d'une somme précise dans le processus d'apprentissage d'un réseau de neurones. Comme nous l'avons vu, les résultats obtenus ont montré que l'amélioration de la précision numérique dans ce contexte a un impact sur le taux de reconnaissance pour le cas d'un problème de classification.

8.2 Perspectives

Les travaux réalisés durant cette thèse ouvrent la voie à de nombreuses perspectives. Nous détaillons ci-dessous quelques-unes.

Dans un futur travail, nous aimerions affiner notre algorithme en ajoutant une phase de test après la ligne 4 de l'algorithme 10, c'est-à-dire avant d'additionner la valeur s_i à la cellule appropriée du tableau sum_by_exp. Plus précisément, on vérifie avant toute sommation si l'exposant calculé $exp_(s_i)$ est égal à l'exposant de sa cellule appropriée donné par $sum_by_exp[exp_s_i]$. En effet, si nous avons un grand ensemble de valeurs à additionner avec le même exposant, le résultat produit peut avoir un exposant plus grand ou plus petit que l'exposant initial. Par conséquent, une perte de précision peut être causée lors des calculs des sommes locales.

Une autre perspective intéressante consiste à alimenter notre algorithme 10 d'une analyse statique. Notre idée est de s'appuyer sur l'analyse statique pour détecter la plage des exposants d'un ensemble donné de valeurs à sommer. Cette étape nous permettra d'avoir la taille du tableau sum_by_exp alloué par l'algorithme 10, donc au lieu d'avoir $exp_max - exp_min + 1$ éléments nous aurons $max - min + 1$ éléments comme indiqué dans la figure 57. Une fois cette plage calculée avec précision, nous pourrons utiliser notre

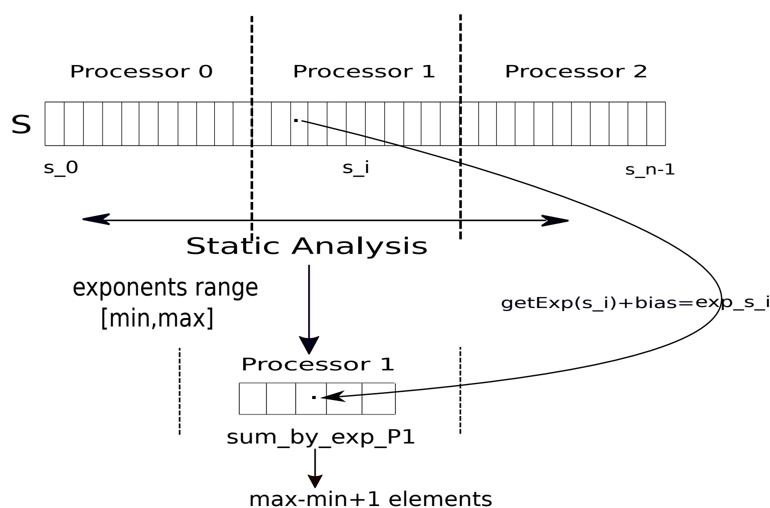

Figure 57 – Schéma du calcul de la somme avec analyse statique.

algorithme de somme afin d'obtenir des résultats en virgule flottante plus précis.

Pour le domaine des réseaux de neurones, il sera intéressant de tester la nouvelle version (basée sur l'algorithme de somme précise) de l'algorithme de rétro-propagation sur d'autres exemples et bases de données. Il convient également de poursuivre l'étude commencée sur les librairies open source existantes telle que *tensorflow* afin d'intégrer nos algorithmes de sommations.

Bibliographie

[1] P. Ahrens, J. Demmel, and H.D. Nguyen. Algorithms for efficient reproducible floating point summation. *ACM Trans. Math. Softw.*, 46(3) :22 :1–22 :49, 2020.

[2] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[3] S. Anwar, K. Hwang, and W. Sung. Structured pruning of deep convolutional neural networks. *J. Emerg. Technol. Comput. Syst.*, 13(3), feb 2017.

[4] M. Barboteu, N. Djehaf, and M. Martel. Numerically accurate code synthesis for gauss pivoting method to solve linear systems coming from mechanics. *Computers & Mathematics with Applications*, 77(11) :2883–2893, 2019.

[5] F. Benmouhoub, N. Damouche, and M. Martel. Improving the numerical accuracy of high performance computing programs by process specialization. In *Emerging Trends In Applied Mathematics And Mechanics*, 2018.

[6] F. Benmouhoub, P.L. Garoche, and M. Martel. Improving the Numerical Accuracy of Parallel Programs by Data Mapping. In *Numerical and Symbolic Abstract Domains*, Porto, Portugal, 2019. .

[7] Farah Benmouhoub, Pierre-Loïc Garoche, and Matthieu Martel. An efficient summation algorithm for the accuracy, convergence and reproducibility of parallel numerical methods. In *Software Verification - 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers*, volume 13124 of *Lecture Notes in Computer Science*, pages 165–181. Springer, 2021.

[8] Farah Benmouhoub, Pierre-Loic Garoche, and Matthieu Martel. Parallel accurate and reproducible summation. In Kohei Arai, editor, *Intelligent Computing*, pages 363–382, Cham, 2022. Springer International Publishing.

[9] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *SIGSOFT Softw. Eng. Notes*, 36(1) :1–8, jan 2011.

[10] G. Bohlender. Floating-point computation of functions with maximum accuracy. *IEEE Trans. Comput.*, 26(7) :621–632, July 1977.

[11] M. Bouaziz. *Réseaux de neurones récurrents pour la classification de séquences dans des flux audiovisuels parallèles.* Theses, December 2017.

[12] J. Brownlee. *Machine Learning Algorithms from Scratch : With Python.* Jason Brownlee, 2017.

[13] J. Cheng, P. Wang, G. Li, Q. Hu, and H. Lu. Recent advances in efficient computation of deep convolutional neural networks. *Frontiers of Information Technology and Electronic Engineering*, 19 :64–77, 01 2018.

[14] C. Chohra, P. Langlois, and D. Parello. Efficiency of Reproducible Level 1 BLAS. In *SCAN : Scientific Computing, Computer Arithmetic, and Validated Numerics*, volume LNCS of *Scientific Computing, Computer Arithmetic, and Validated Numerics*, pages 99–108, Würzburg, Germany, September 2014.

[15] C. Collange, D. Defour, S. Graillat, and R. Iakymchuk. Numerical Reproducibility for the Parallel Reduction on Multi- and Many-Core Architectures. working paper or preprint, September 2015.

[16] M. Courbariaux, Y. Bengio, and J.P. David. Low precision arithmetic for deep learning. *3rd International Conference on Learning Representations (ICLR2015)*, 12 2014.

[17] L. Courtrai, M.T. Pham, J.C. Burnel, and S. Lefèvre. Apprentissage de réseaux de neurones de super-résolution pour la détection d'objets de petite taille dans les images de télédétection. *Congrès Reconnaissance des Formes, Image, Apprentissage et Perception*, 2020.

[18] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[19] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4) :511–547, 1992.

[20] N. Damouche and M. Martel. Salsa : An automatic tool improve the accuracy of programs. In *6th International Workshop on Automated Formal Methods, AFM*, 2017.

[21] N. Damouche, M. Martel, and A. Chapoutot. Impact of accuracy optimization on the convergence of numerical iterative methods. In M. Falaschi, editor, *LOPSTR 2015*, volume 9527 of *Lecture Notes in Computer Science*, pages 143–160. Springer, 2015.

[22] N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In *FMICS*, 2015.

[23] N. Damouche, M. Martel, and A. Chapoutot. Transformation of a PID controller for numerical accuracy. *Electronic Notes in Theoretical Computer Science*, 317 :47–54, 2015.

[24] N. Damouche, M. Martel, and A. Chapoutot. Numerical accuracy improvement by interprocedural program transformation. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '17, page 1–10, New York, NY, USA, 2017. Association for Computing Machinery.

[25] E. Darulova and V. Kuncak. Sound compilation of reals. In S. Jagannathan and P. Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.

[26] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18 :224–242, 1971/72.

[27] J. Demmel and Y. Hida. Accurate floating point summation. 06 2002.

[28] J. Demmel and Y. Hida. Accurate and efficient floating point summation. *SIAM J. Sci. Comput.*, 25(4) :1214–1248, April 2003.

[29] J. Demmel and H.D. Nguyen. Fast reproducible floating-point summation. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, pages 163–172. IEEE Computer Society, 2013.

[30] J. Demmel and H.D. Nguyen. Parallel reproducible summation. *IEEE Transactions on Computers*, 64(7) :2060–2070, 2015.

[31] L. Fousse, G. Hanrot, V. Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR : A Multiple-Precision Binary Floating-Point Library With Correct Rounding. Research Report RR-5753, INRIA, 2005.

[32] Y. Georgiou, E. Jeannot, G. Mercier, and A. Villiermet. Topology-aware job mapping. *IJHPCA*, 32(1) :14–27, 2018.

[33] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1) :5–48, March 1991.

[34] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, third edition, 1996.

[35] Y. Gong, L. Liu, M. Yang, and L.D. Bourdev. Compressing deep convolutional networks using vector quantization. *ArXiv*, abs/1412.6115, 2014.

[36] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations : A simple abstract interpreter. In D. Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212. Springer, 2002.

[37] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *European Congress on Embedded Real Time Software, ERTS 2006, Proceedings*, 2006.

[38] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *Conference ERTS'06*, Toulouse, France, January 2006.

[39] S. Graillat, P. Langlois, and N. Louvet. Algorithms for Accurate, Validated and Fast Polynomial Evaluation. *Japan Journal of Industrial and Applied Mathematics*, 26(2-3) :191–214, 2009.

[40] S. Graillat and V. Ménissier-Morain. Compensated Horner scheme in complex floating point arithmetic. In *Proceedings, 8th Conference on Real Numbers and Computers*, pages 133–146, Santiago de Compostela, Spain, July 2008.

[41] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1737–1746. JMLR.org, 2015.

[42] J. HENRY. *Analyse statique par interprétation abstraite et procédures de décision.* Theses, Université de Grenoble, October 2014.

[43] N. Higham. The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4) :783–799, 1993.

[44] N. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, USA, 1996.

[45] T. Hoefler, E. Jeannot, and G. Mercier. An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing. In Emmanuel Jeannot and Julius Zilinskas, editors, *High Performance Computing on Complex Environments*, pages 75–94. Wiley, June 2014.

[46] Roman Iakymchuk, Caroline Collange, David Defour, and Stef Graillat. ExBLAS : Reproducible and Accurate BLAS Library. In *NRE : Numerical Reproducibility at Exascale*, Austin, TX, United States, November 2015. Numerical Reproducibility at Exascale (NRE2015) workshop held as part of the Supercomputing Conference (SC15).

[47] A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *SAS'12*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.

[48] A. Ioualalen and M. Martel. Neural network precision tuning. In David Parker and Verena Wolf, editors, *Quantitative Evaluation of Systems*, pages 129–143, Cham, 2019. Springer International Publishing.

[49] Claude-Pierre Jeannerod, Jean-Michel Muller, and Paul Zimmermann. On various ways to split a floating-point number. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 53–60, 2018.

[50] C.P. Jeannerod and S. Rump. On relative errors of floating-point operations : Optimal bounds and applications. *Mathematics of Computation*, 87, 01 2014.

[51] F. Jézéquel and J.M. Chesneaux. CADNA : a library for estimating round-off error propagation. *Computer Physics Communications*, 178(12) :933–955, June 2008.

[52] V. Joseph, G.L Gopalakrishnan, S. Muralidharan, M. Garland, and A. Garg. A programmable approach to neural network compression. *IEEE Micro*, 40(5) :17–25, 2020.

[53] W. Kahan. Pracniques : Further remarks on reducing truncation errors. *Commun. ACM*, 8(1) :40, January 1965.

[54] W. Kahan. A survey of error analysis. In *IFIP Congress*, 1971.

[55] G.A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[56] Donald E. Knuth. *The Art of Computer Programming, Volume 2 : Seminumerical Algorithms.* Addison-Wesley, Boston, third edition, 1997.

[57] U. Kulisch. *Computer Arithmetic and Validity : Theory, Implementation, and Applications.* De Gruyter, 2008.

[58] P. Langlois, M. Martel, and L. Thévenoux. Accuracy versus time : A case study with summation algorithms. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, page 121–130, New York, NY, USA, 2010. Association for Computing Machinery.

[59] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3) :308–323, sep 1979.

[60] H. Leuprecht and W. Oberaigner. Parallel algorithms for the rounding exact summation of floating point numbers. *Computing*, 28(2) :89–104, 1982.

[61] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou. A survey of convolutional neural networks : Analysis, applications, and prospects. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–21, 2021.

[62] M. Malcolm. On accurate floating-point summation. *Commun. ACM*, 14(11) :731–736, November 1971.

[63] J.M. Muller, N. Brisebarre, F. De Dinechin, C.P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[64] O. Møller. Quasi double-precision in floating point addition. *BIT Numerical Mathematics*, 5, 1965.

[65] T. Ogita, S. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6) :1955–1988, 2005.

[66] J. R. Wilcox P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI'15*, pages 1–11. ACM, 2015.

[67] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[68] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[69] P. Panchekha, A. Sanchez-Stern, J.R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6) :1–11, jun 2015.

[70] M. Parizeau. Réseaux de neurones. *GIF-21140 et GIF-64326*, 124, 2004.

[71] M. Pichat. Correction d'une somme en arithmetique a virgule flottante. *Numer. Math.*, 19(5) :400–406, October 1972.

[72] W. Rawat and Z. Wang. Deep Convolutional Neural Networks for Image Classification : A Comprehensive Review. *Neural Computation*, 29(9) :2352–2449, 09 2017.

[73] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323 :533–536, 1986.

[74] S. Rump. Ultimately fast accurate summation. *SIAM Journal on Scientific Computing*, 31(5) :3466–3502, 2009.

[75] S. Rump and T. Ogita. Fast high precision summation. *Nonlinear Theory and Its Applications, IEICE*, 1, 01 2010.

[76] S. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I : faithful rounding. *SIAM J. Scientific Computing*, 31(1) :189–224, 2008.

[77] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM'15*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.

[78] J. Stoer and R Bulirsch. *Introduction to numerical analysis.* Texts in applied mathematics. Springer, 2002.

[79] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation : A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[80] L. Thévenoux, P. Langlois, and M. Martel. Automatic source-to-source error compensation of floating-point programs : code synthesis to optimize accuracy and time. *Concurrency and Computation : Practice and Experience*, 29(7) :e3953, 2017.

[81] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás. Trends in data locality abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(10) :3007–3020, Oct 2017.

[82] R. Yates. Fixed-point arithmetic : An introduction. *Digital Signal Labs*, 81(83) :198, 2009.

[83] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun. Efficient and accurate approximations of nonlinear convolutional networks. pages 1984–1992, 06 2015.