

Squelettes algorithmiques pour la programmation et l'exécution efficaces de codes parallèles

Joeffrey Legaux

▶ **To cite this version:**

Joeffrey Legaux. Squelettes algorithmiques pour la programmation et l'exécution efficaces de codes parallèles. Autre [cs.OH]. Université d'Orléans, 2013. Français. NNT : 2013ORLE2073 . tel-00990852v2

HAL Id: tel-00990852

https://theses.hal.science/tel-00990852v2

Submitted on 13 Apr 2015

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

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



UNIVERSITÉ D'ORLÉANS



ÉCOLE DOCTORALE MATHEMATIQUES, INFORMATIQUE, PHYSIQUE THEORIQUE et INGENIERIE DES SYSTEMES

Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE présentée par :

Joeffrey LEGAUX

soutenue le : **13 décembre 2013**

pour obtenir le grade de : **Docteur de l'université d'Orléans**

Discipline/ Spécialité : **Informatique**

Squelettes algorithmiques pour la programmation et l'exécution efficaces de codes parallèles

THÈSE dirigée par :

Frédéric LOULERGUE	Professeur, Université d'Orléans

RAPPORTEURS :

Marco DANELUTTO	Associate Professor, Università di Pisa
Herbert KUCHEN	Professeur, Westfälische Wilhelms-Universistät Münster

JURY :

Joël FALCOU	Maître de conférence, Université Paris-Sud 11
Sylvain JUBERTIE	Maître de conférence, Université d'Orléans
Sébastien LIMET	Professeur, Université d'Orléans
Stéphane VIALLE	Professeur, Supélec Metz

Remerciemements

Je tiens tout d'abord à remercier Stéphane Vialle de m'avoir fait l'honneur de présider mon jury de thèse, Herbert Kuchen et Marco Danelluto d'avoir accepté de rapporter ma thèse et Sébastien Limet et Joël Falcou d'avoir bien voulu participer à mon jury.

Je remercie particulièrement Frédéric Loulergue qui a toujours su m'orienter dans les bonnes directions afin de faire progresser et valoriser mes recherches, et Sylvain Jubertie pour m'avoir apporté un soutien conséquent face aux difficultés techniques notamment de programmation.

Merci à Emmanuel Melin qui m'a permis de reprendre pied dans le monde de la recherche.

Un merci général à l'ensemble du LIFO et de l'IUT d'Orlénas qui ont toujours su donner une place importante aux doctorants et les intégrer dans leurs activités de recherche ou d'enseignement.

Last but not least, un immense merci à Caty pour m'avoir soutenu et supporté dans les moments les plus difficiles...

Table des matières

Table des matières iv

Liste des figures vi

1 Introduction 1
1.1 Contexte . . . 1
1.2 Contribution . . . 3
1.3 Organisation du manuscrit . . . 4

2 État de l'art 7
2.1 Évolution des architectures parallèles . . . 8
2.1.1 Les origines . . . 8
2.1.2 Microprocesseurs . . . 9
2.1.3 Processeurs spécialisés . . . 12
2.2 Modèles de programmation parallèle . . . 14
2.2.1 Modèles architecturaux . . . 14
2.2.2 Modèles algorithmiques . . . 18
2.2.3 Modèles de transition logico-matérielle . . . 19
2.2.4 Modèles implicites . . . 22
2.3 Squelettes algorithmiques . . . 23
2.3.1 Classification des squelettes . . . 23
2.3.2 Langages et bibliothèques de squelettes . . . 25
2.4 Performance et complexité de développement . . . 31
2.4.1 Mesures de performance . . . 31
2.4.2 Effort de programmation . . . 37
2.5 Conclusion . . . 39

3 Cas d'étude : optimisation d'un calcul de multiplication de matrices 41
3.1 Programme séquentiel . . . 42
3.1.1 Optimisation des accès mémoire . . . 43
3.1.2 Utilisation de la mémoire cache : algorithme par blocs . . . 44
3.2 Vectorisation . . . 45
3.2.1 Alignement mémoire . . . 46

3.2.2 Transposition . . . 47
3.2.3 Algorithme . . . 48
3.3 Memoire partagée . . . 50
3.4 Analyse de l'effort de programmation . . . 52
3.5 Conclusion . . . 55

4 OSL **59**
4.1 OSL pour les utilisateurs . . . 60
4.1.1 Aperçu . . . 60
4.1.2 Exemples . . . 65
4.1.3 Sérialisation des données . . . 67
4.2 OSL pour les développeurs . . . 69
4.2.1 Principes généraux de métaprogrammation . . . 69
4.2.2 Expression templates . . . 76
4.2.3 Création de nouveaux squelettes . . . 82
4.3 Conclusion . . . 85

5 Manipulations des distributions **87**
5.1 Problématique . . . 88
5.2 Squelettes agissant sur la distribution . . . 88
5.2.1 Redistribute . . . 89
5.2.2 GetPartition . . . 90
5.2.3 Flatten . . . 90
5.2.4 Implémentation du filtrage . . . 91
5.3 Tri par échantillonnage régulier en parallèle . . . 91
5.3.1 Implémentation . . . 93
5.3.2 Coût BSP . . . 93
5.3.3 Expérimentations . . . 94
5.4 Conclusion . . . 95

6 Gestion des exceptions parallèles **99**
6.1 Support de la gestion des exceptions en parallèle . . . 100
6.1.1 Sérialisation grâce à Boost . . . 105
6.1.2 Relancer les exceptions . . . 106
6.2 Implémentation dans OSL . . . 108
6.2.1 Le squelette forwardExceptions . . . 108
6.2.2 Retour sur trace avec exceptions en parallèle . . . 109
6.2.3 Mesures expérimentales . . . 110
6.3 Conclusion . . . 112

7 Parallélisation correcte avec l'homomorphisme BSP **115**
7.1 Théorie de *BH* . . . 116
7.1.1 Définitions . . . 116
7.1.2 Transformation d'une fonction *mapAround* en *BH* . . . 118

7.2 Intégration à OSL . . . 119
7.2.1 Utilisation . . . 119
7.2.2 Implémentation . . . 121
7.3 Dérivation de deux problèmes . . . 122
7.3.1 Valeurs inférieures les plus proches . . . 122
7.3.2 Multiplication matrice creuse-vecteur . . . 125
7.3.3 Mesures expérimentales . . . 127
7.4 Comparaison avec un algorithme de référence . . . 129
7.4.1 Description . . . 129
7.4.2 Performances . . . 132
7.4.3 Effort de programmation . . . 133
7.5 Conclusion . . . 134

8 Conclusion **135**
8.1 Bilan . . . 135
8.2 Perspectives . . . 136

A Les systèmes SPEED et Mirev **141**
A.1 SPEED . . . 141
A.2 Mirev . . . 141

B Exemple détaillé du calcul des valeurs inférieures les plus proches **143**

Bibliographie **145**

Liste des figures

2.1 Modèle d'exécution BSP . . . 21

3.1 Découpage par blocs . . . 45
3.2 Mécanisme général en SSE . . . 47
3.3 Gains des différentes optimisations . . . 50
3.4 Gains de la parallélisation par OpenMP . . . 52
3.5 Comparaison performance - effort de développement . . . 54

4.1 OSL Skeletons . . . 62
4.2 Pile d'appels . . . 80
4.3 Construction d'une expression . . . 80

5.1 Filtrage de données 87
5.2 Filtrage de donnée par un mécanisme de haut niveau 89
5.3 Parallel Regular Sampling Sort 92
5.4 Tri par échantillonnage régulier en parallèle dans OSL 94
5.5 Passage à l'échelle de la parallélisation du tri par échantillonage régulier 95

6.1 Problématique de la récupération locale 101
6.2 Solution envisagée 104
6.3 Surcoût des exceptions dans une simulation 111
6.4 Solveur de sudoku par retour sur trace 112

7.1 BH local 117
7.2 Calcul de BH en parallèle 118
7.3 Calcul des ANSV pour la valeur centrale 3 (en blanc), les flèches pointent vers les deux solutions. 122
7.4 Les candidats d'une sous-liste à droite. Les valeurs se succédant en conservant un ordre décroissant sont conservées comme candidats (en noir) tandis que les autres valeurs sont éliminées (en gris). 124
7.5 Parallélisation en mémoire distribuée 128
7.6 Parallélisation en mémoire partagée 129
7.7 Calcul des séquences Seq1i et Seq2j. Les flèches représentent la relation « valeur inférieure la plus proche » dans la direction correspondante. Le premier tableau ne contient que les minima, le tableaux du bas représentent la totalité des données sur les processus i, j et k. 131

Introduction

1

Sommaire

1.1 Contexte 1
1.2 Contribution 3
1.3 Organisation du manuscrit 4

1.1 Contexte

De nos jours, les architectures parallèles se sont progressivement généralisées au sein de tous les matériels informatiques. À l'origine, ces architectures étaient réservées à des utilisations très spécifiques de calcul haute performance et leurs développeurs étaient une niche de spécialistes, tandis que la majorité des développements avaient lieu sur les architectures séquentielles. Ces dernières ont offert durant de nombreuses années une amélioration des performances sans qu'il soit nécessaire de remettre en question leur mode de programmation grâce à l'augmentation rapide des fréquences de fonctionnement. Certains formes de parallélisme ont été intégrées dans les processeurs pour en améliorer l'efficacité (*pipelining* des instructions, calcul simultané dans des unités de calcul séparées, ...) mais celles-ci restaient implicites et cachées aux programmeurs. Cependant, l'augmentation de la fréquence des processeurs a atteint il y a quelques années un pic difficilement franchissable à cause de limitations physiques. Les architectures parallèles se sont donc immiscées dans les utilisations courantes afin de pouvoir continuer à fournir un accroissement des performances. Le parallélisme visible par les développeurs est désormais présent dans toutes les architectures, des super-calculateurs jusqu'aux aux appareils embarqués tels que les téléphones portables. Exploiter explicitement cette caractéristique est nécessaire si l'on souhaite profiter réellement des performances offertes par le matériel.

La programmation parallèle amène des difficultés de programmation spécifiques, notamment les risques d'indéterminisme et d'interblocage. La généralisation des architectures parallèles amène ces complexités à l'ensemble des développeurs, qui, endehors d'une niche de spécialistes, ne sont pas formés pour cela. On ne peut pas s'attendre à ce que tous les développeurs se spécialisent dans la programmation parallèle,

les exigences dans les applications générales étant beaucoup moins élevées que dans le calcul scientifique. Les développeurs cherchent à obtenir des performances correctes pour un ensemble de matériels et d'applications, et pas la meilleure performance possible pour une application spécifique exécutée sur un matériel particulier. La question qui se pose est plutôt celle de la productivité : les développeurs souhaitent créer des programmes parallèles exploitant suffisamment les capacités du matériel (autrement dit, le programme parallèle doit être sensiblement plus rapide que son équivalent séquentiel), mais veulent également conserver des temps de développement raisonnables.

De nombreux modèles de programmation parallèle existent, on retrouve notamment parmi les modèles de références MPI pour la gestion du parallélisme distribué et les *pthreads* pour le parallélisme en mémoire partagée. Ces modèles de bas niveau permettent d'exploiter au mieux les capacités des machines parallèles en laissant l'utilisateur gérer de manière très fine le comportement de ses applications. La contrepartie est qu'il faut développer les applications spécifiquement pour ces modèles et qu'il peut être très difficile d'en tirer de bonnes performances sans une connaissance pointue du fonctionnement du matériel.

Nous souhaitons nous orienter vers des modèles permettant une meilleure productivité, nous étudions donc les approches de haut niveau. Elles sont toutes indiquées pour cela, car leur objectif principal est de cacher, au moins partiellement, les détails de l'architecture matérielle pour permettre à l'utilisateur de se concentrer sur l'expression du comportement algorithmique de son application. Nous effectuons plus loin un tour d'horizon des nombreux modèles existants, parmi ceux-ci le modèle des squelettes algorithmiques nous intéresse plus particulièrement.

Les squelettes algorithmiques furent définis pour la première fois par Cole [34] comme étant « *des fonctions d'ordre supérieur spécialisées parmi lesquelles l'une d'entre elles doit être choisie afin d'exprimer le but du programme au sens le plus large possible* ». En pratique ceux-ci sont donc des abstractions de schémas utilisés de manière récurrente dans la programmation parallèle afin d'effectuer des calculs ou des communications.

Dans la programmation par squelettes algorithmiques, le programmeur doit donc appréhender l'algorithme qu'il souhaite implémenter de façon abstraite, en déterminant quelle est l'action la plus générale effectuée : par exemple « cet algorithme effectue un calcul indépendant sur chacune des données », « cet algorithme doit effectuer successivement plusieurs calculs indépendants sur l'ensemble des données » ou encore « cet algorithme divise l'ensemble des données en sous-groupes puis les traite séparément ». Ce comportement général de l'algorithme constituera donc le *squelette* du programme qu'il faudra ensuite raffiner afin d'exprimer plus en détail les parties plus spécifiques de l'algorithme par la composition d'autres squelettes et/ou d'opérations définies explicitement, généralement sous la forme de fonctions utilisateur qui seront appliquées par les squelettes.

Cette approche « de haut en bas » connue sous le nom de *parallélisme structuré* [96]

offre de nombreux avantages : il est possible de facilement comprendre l'algorithme effectué à partir du programme à squelettes, il est possible d'analyser le programme directement dans sa globalité à partir du squelette principal et le programme est totalement indépendant de l'architecture matérielle : la description du programme se faisant à travers un ensemble de squelettes abstraits, celle-ci n'est donc pas liée à leur implémentation.

On obtient ainsi un modèle où la composante *calcul* du programme, c'est à dire l'ensemble des opérations que l'algorithme doit effectuer, se trouve totalement séparée de la composante *coordination*, l'ensemble des moyens à mettre en place pour synchroniser les processeurs, échanger leurs données. Ce découpage permet également de bien séparer et spécialiser les rôles des développeurs travaillant autour d'un langage à squelettes : les utilisateurs n'ont à se préoccuper que de la traduction de leurs algorithmes en programmes à squelettes, alors que les développeurs du langage n'ont à se préoccuper que de l'implémentation des squelettes.

La programmation parallèle par les squelettes algorithmiques de par son approche de haut niveau semble intuitivement offrir une bonne productivité aux développeurs. Cette intuition pourrait être vérifiée par l'étude du comportement de développeurs mis en situation face à un langage à squelettes, et ainsi obtenir une caractérisation précise de la productivité obtenue par ce modèle de programmation. Cependant un tel protocole de test est très difficile à mettre en place. Une autre façon de caractériser cette productivité de manière plus synthétique serait d'effectuer des mesures sur les codes produits afin de comparer un programme séquentiel avec un programme à squelettes équivalent.

1.2 Contribution

Nous effectuons donc dans cette thèse une étude des modèles de programmation parallèle, en nous plaçant du point de vue du développeur généraliste. Nous nous intéressons donc aux performances qu'il est possible d'obtenir grâce à ces modèles, mais également à leur expressivité et la productivité découlant de leur utilisation. Nous mettons en place un protocole expérimental permettant de donner des mesures représentatives des performances et du temps de développement nécessaire pour un programme donné. L'application de ce protocole pour un ensemble de modèles de programmation parallèle courants sur un algorithme classique corrobore notre intuition selon laquelle les modèles de haut niveau offrent une bonne productivité en permettant de développer dans des temps relativement faibles des applications fournissant des performances parallèles acceptables.

À partir de ce constat, nous orientons nos travaux vers le modèle des squelettes algorithmiques au sein de la bibliothèque *Orléans Skeleton Library* ou OSL. Nos contributions à cette bibliothèque auront consisté en l'apport de nouvelles fonctionnalités

afin d'élargir le spectre des comportements algorithmiques exprimables dans ce modèle. Nos travaux sont toujours observés selon l'angle de la performance obtenue, afin de s'assurer que l'expressivité supplémentaire n'entraîne pas de dégradation importante des temps d'exécution des programmes obtenus.

1.3 Organisation du manuscrit

La présentation de nos travaux se déroule de la manière suivante :

- Nous étudions dans le chapitre 2 les différents architectures et modèles de programmation parallèles existants et leurs évolutions prévues, une attention toute particulière étant portée au modèle des squelettes algorithmiques. Nous étudions également les méthodes permettant de mesurer les performances des architectures et programmes parallèles ainsi que les techniques d'analyses du coût de développement, ce qui nous permet de décrire un protocole expérimental afin de caractériser les programmes parallèles selon ces axes.
- Dans le chapitre 3, nous appliquons ici diverses techniques et modèles de parallélisation afin d'améliorer les performances d'un algorithme classique de multiplication de matrices sur l'architecture parallèle courante du multi-processeur multi-cœurs à mémoire partagée. Le protocole expérimental décrit précédemment est ici appliqué sur un cas concret, et la comparaison des performances obtenues avec les coûts de développement vont dans le sens de notre intuition selon laquelle les modèles de haut niveau offrent un meilleur rapport performance/coût de développement.
- Le chapitre 4 est consacré à une présentation d'OSL, la bibliothèque de squelettes algorithmiques sur laquelle se base la suite de nos travaux. Nous décrivons dans un premier temps ses principes généraux et la sémantique des squelettes existants, afin de donner un aperçu des possibilités de développement de programmes parallèles en utilisant cette bibliothèque. Nous effectuons ensuite une description plus approfondie des mécanismes internes permettant d'obtenir de bonnes performances dans les programmes générés.
- Nous étudions dans le chapitre 5, une des fonctionnalités clé d'OSL : la possibilité pour l'utilisateur de manipuler explicitement la distribution des données au sein de l'architecture parallèle. Nous introduisons le besoin d'une telle fonctionnalité à travers un exemple simple, nous présentons ensuite les squelettes permettant de mettre en place ce mécanisme de manière sûre et facilement manipulable, puis concluons sur un cas complexe de tri en parallèle utilisant fortement cette fonctionnalité.
- Les mécanismes de gestion d'exceptions sont l'objet du chapitre 6. Ils sont un moyen classique permettant d'exprimer sans trop alourdir le code le comportement qu'un programme doit adopter lorsque des évènement irréguliers se produisent durant son exécution. Nous étudions dans ce chapitre les difficultés rencontrées à manipuler les exceptions dans un environnement parallèle, puis nous

décrivons la mise en place d'un mécanisme permettant leur utilisation de manière simple au sein de nos programmes à squelettes.
- Dans le chapitre 7, la théorie des homomorphismes BSP permet la dérivation de programmes parallèles corrects à travers un modèle de très haut niveau. Nous étudions ici ce modèle et décrivons la dérivation des deux algorithmes : le calcul des valeurs inférieures les plus proches dans un tableau, et le produit d'une matrice creuse avec un vecteur, afin de pouvoir leur appliquer ce modèle. Nous détaillons ensuite l'intégration au sein d'OSL de ce modèle de programmation, effectuons une analyse des performances obtenues par les deux algorithmes précédents. Nous terminons par l'implémentation d'une variante du premier algorithme utilisant un modèle de bas niveau, et comparons les deux versions sous l'angle de la productivité.
- Le chapitre 8 dresse un bilan de nos travaux et dégage des pistes de recherche pour le futur.

L'annexe A décrit les machines sur lesquelles nous avons réalisé nos expériences.

Des versions antérieures des travaux présentés ont fait l'objet de publications dans des conférences internationales:

1. Joeffrey Legaux, Zhenjiang Hu, Frédéric Loulergue, Kiminori Matsuzaki, Julien Tesson Programming with BSP Homomorphisms. Dans *19th international conference on Parallel Processing (Euro-Par'13)*, pages 446–457. Springer Berlin Heidelberg, 2013
2. Joeffrey Legaux, Frédéric Loulergue, Sylvain Jubertie Managing Arbitrary Distributions of Arrays in Orléans Skeleton Library Dans 2013 International Conference on High Performance Computing & Simulation(HPCS 2013)
3. Joeffrey Legaux, Frédéric Loulergue, Sylvain Jubertie OSL: An Algorithmic Skeleton Library with Exceptions Dans 2013 International Conference on Computational Science(ICCS2013), Procedia Computer Science, pages 260–269. Elsevier 2012
4. Joeffrey Legaux, Sylvain Jubertie, Frédéric Loulergue Experiments in Parallel Matrix Multiplication on Multi-core Systems Dans 12th international conference on Algorithms and Architectures for Parallel Processing(ICA3PP'12), pages 362–376. Springer Berlin Heidelberg, 2012

État de l'art

2

Sommaire

2.1 Évolution des architectures parallèles . . . 8
2.1.1 Les origines . . . 8
2.1.2 Microprocesseurs . . . 9
2.1.3 Processeurs spécialisés . . . 12
2.2 Modèles de programmation parallèle . . . 14
2.2.1 Modèles architecturaux . . . 14
2.2.2 Modèles algorithmiques . . . 18
2.2.3 Modèles de transition logico-matérielle . . . 19
2.2.4 Modèles implicites . . . 22
2.3 Squelettes algorithmiques . . . 23
2.3.1 Classification des squelettes . . . 23
2.3.2 Langages et bibliothèques de squelettes . . . 25
2.4 Performance et complexité de développement . . . 31
2.4.1 Mesures de performance . . . 31
2.4.2 Effort de programmation . . . 37
2.5 Conclusion . . . 39

Afin de comprendre les problématiques causées par l'utilisation de machines parallèles, nous étudions comment sont apparues et ont évolué ces architectures et leurs caractéristiques. Nous nous intéressons ensuite aux modèles de programmation ayant été conçus afin d'exploiter les diverses capacités de parallélisme des matériels. Parmi ces différents modèles, celui des squelettes algorithmiques a plus particulièrement retenu notre attention et nous en dressons un portrait détaillé. Le développement de programmes parallèles a pour objectif premier d'obtenir de meilleurs performances, mais cela représente un travail supplémentaire par rapport à une implémentation séquentielle. Nous finissons donc par l'étude de méthodes nous permettant de quantifier

précisément les programmes parallèles selon deux axes : le gain de performances et le coût de développement.

2.1 Évolution des architectures parallèles

Le but premier des ordinateurs parallèles est d'obtenir des performances inatteignables par les ordinateurs séquentiels. Le besoin de performances élevées provient principalement de la communauté du calcul scientifique, qui souhaite toujours plus de puissance de calcul afin d'effectuer des simulations de plus en plus précises. De nombreux matériels différents ont été utilisés au cours des années dans ce domaine ; dans l'absolu, toute machine capable d'effectuer des opérations mathématiques peut être utilisée pour le calcul scientifique, cependant pour des raisons de coût, de performance, de facilité d'utilisation ou de disponibilités, on retrouvera fréquemment les mêmes architectures.

2.1.1 Les origines

Au départ, les premiers ordinateurs utilisés pour le calcul scientifique étaient simplement les uniques ordinateurs scalaires existants. Ces machines évoluèrent rapidement en fréquence de fonctionnement grâce aux développements techniques des transistors puis des circuits intégrés. Les premiers super-calculateurs apparurent dans les années 60, leur architecture se focalisant sur une exécution très rapide des instructions, à l'opposée des *mainframes* qui visaient eux à gérer un grand nombre d'utilisateurs simultanés.

2.1.1.1 Multiprocesseurs

Malgré les progrès réguliers dans la rapidité des unités de calcul, il fallait toujours attendre les développements des nouveaux processeurs pour obtenir plus de puissance de calcul. L'idée de faire collaborer plusieurs processeurs afin d'effectuer un unique calcul plus rapidement se concrétisa dans les années 60 dans les architectures SMP (*Symmetric Multi Processor*). Dans celles-ci, un ensemble de processeurs identiques se côtoient au sein d'un même noeud de calcul et accèdent tous à une mémoire commune.

2.1.1.2 Processeurs vectoriels

L'architecture des super-calculateurs évolua et adopta les processeurs vectoriels dans les années 70, les machines conçues sur ce modèle par Cray Research furent les plus puissantes de leur époque et connurent un grand succès et d'autres acteurs encore présents de nos jours dans ce domaine tels que Fujitsu, NEC et Hitachi conçurent des machines similaires bien que globalement moins performantes. Ces processeurs tirent

leur puissance de leurs capacités de parallélisme de données : une même instruction peut être appliquée simultanément à de nombreux éléments de données (modèle SIMD : *Single Instruction Multiple Data*). Suite aux difficultés à augmenter la quantité de données manipulables vectoriellement par un unique processeur, ces machines évoluèrent vers un niveau supérieur de parallélisme en utilisant plusieurs processeurs vectoriels en parallèle afin d'obtenir toujours plus de puissance de calcul.

2.1.2 Microprocesseurs

Les microprocesseurs apparurent dans les années 80. De puissance modeste, leur faible encombrement et consommation électrique ont permis l'apparition des ordinateurs personnels. Ces microprocesseurs évoluèrent rapidement, au point d'atteindre des niveaux de performance d'un ordre proche de celui des processeurs vectoriels malgré leur nature scalaire. Étant beaucoup moins coûteux, ceux-ci furent combinés en grand nombre afin d'obtenir des niveaux de parallélisme et des performances équivalentes aux machines vectorielles.

Les architectures de type SMP furent rapidement adaptées pour les microprocesseurs, cependant ceux-ci entrent alors en compétition pour accéder à la mémoire ce qui limite très fortement le nombre de processeurs utilisables dans une telle architecture : ajouter plus de processeurs sur une même mémoire partagée n'offrira plus de performances que si cette mémoire est capable d'alimenter les processeurs supplémentaires en données, or les débits de la mémoire restent fortement limités. Ce phénomène tend à s'accentuer car la vitesse de calcul des processeurs a toujours évolué plus rapidement que les débits des mémoires, et les microprocesseurs étaient d'ores et déjà beaucoup plus rapides que les mémoires.

Une approche plus simple dans l'exploitation de nombreux microprocesseurs en parallèle est celle de la grille : il s'agit simplement de connecter entre eux plusieurs ordinateurs indépendants à travers un réseau d'interconnexion. Cette architecture est simple, extensible (il suffit d'augmenter le nombre de machines pour augmenter la puissance de calcul), d'un coût faible et très répartissable géographiquement (comme par exemple dans la structure Grid5000 qui est répartie à travers de nombreuses villes françaises). Une différence fondamentale entre une grille de processeurs scalaires et un SMP est que la mémoire de chaque microprocesseur est indépendante des autres, il s'agit donc de mémoire distribuée où les échanges de données et synchronisations doivent se faire de manière explicite à travers le réseau. Cette architecture est encore très présente de nos jours sous la forme plus abstraite du *cloud computing*.

Cependant cette architecture est de son côté limitée lors des calculs intensifs par les performances du réseau, celui-ci étant forcément beaucoup plus lent que les accès à la mémoire, c'est pourquoi les structures de grilles sont plutôt utilisées pour accueillir un grand nombre d'utilisateurs et d'applications légèresen simultané. Une approche plus favorable au calcul haute performance est celle de la grappe, qui est une spécialisation

de la grille où un grand nombre de processeurs identiques sont reliés entre eux par un réseau d'interconnexion à très haute performance. Ces réseaux d'interconnexion très rapides impliquent une distance très faible entre les processeurs, et sont conçus avec une forme particulière (par exemple un tore multidimensionnel) visant à réduire autant que possible les distances physiques parcourues par les données (et donc la latence). Cette architecture est beaucoup plus performante pour le calcul intensif, mais également beaucoup plus coûteuse et peu extensible.

2.1.2.1 Architectures hiérarchiques

Les approches distribuées permettent d'utiliser un très grand nombre de processeurs, cependant la latence d'accès aux données entre processeurs reste un goulot d'étranglement majeur dans ces architectures. Le réutilisation de l'approche à mémoire partagée au sein des architectures à mémoire distribuée permet d'augmenter le nombre total de processeurs sans augmenter la charge sur le réseau : chaque noeud de la grappe ou du cluster n'est plus un simple processeur mais un système SMP au sein duquel les processeurs peuvent s'échanger des données très rapidement grâce à leur mémoire partagée. Cependant, les architectures de type SMP se montrent de plus en plus limitées par l'augmentation de la fréquence de fonctionnement des processeurs et l'augmentation de leur nombre dans un même système à mémoire partagée.

Des architectures dérivant du SMP tentent donc de repousser les limitations dues aux accès mémoire, notamment les architecture à accès mémoire non uniformes *NUMA* (*Non Uniform Memory Access*). La mémoire y est organisée de manière hiérarchique et chaque processeur (ou petit groupe de processeurs) se voit attribué une section de mémoire en accès direct et donc très rapide, et ne pourra accéder aux données des autres mémoires qu'à travers un bus d'interconnexion beaucoup plus lent. Néanmoins le nombre de processeurs exploitables reste faible par rapport à la quantité que l'on souhaite utiliser dans les super-calculateurs actuels, ceux-ci prennent donc généralement la forme hybride d'un ensemble de noeuds à mémoire distribuée, mais dont chaque noeud constitue en lui-même un multiprocesseur *NUMA*.

2.1.2.2 Multi-coeurs

Toujours dans le même but d'augmenter la puissance de calcul des microprocesseurs sans augmenter leur fréquence est apparu le concept des coeurs multiples : il s'agit de regrouper plusieurs ensembles de circuits logiques correspondant chacun à un processeur complet (les *coeurs*) dans un seul composant physique qui sera considéré de l'extérieur comme un unique processeur. Cette architecture peut être vue comme un système multiprocesseur intégré en une seul puce, l'avantage sur ces derniers étant la proximité des coeurs de calcul qui leur permet de communiquer très rapidement et selon les architectures d'avoir accès à des mémoires caches partagées entre eux ce qui permet de limiter les phénomènes d'engorgement lors des accès à la mémoire.

Ce type de processeur est devenu la norme aujourd'hui, et les prévisions actuelles suggèrent que l'augmentation de leurs performances se fera principalement par l'augmentation du nombre de coeurs à l'avenir.

2.1.2.3 Vectorisation

Les microprocesseurs ont vu leurs performances augmenter très rapidement grâce à leur miniaturisation progressive permettant d'atteindre des fréquences de fonctionnement de plus en plus élevées, cependant cet accroissement des fréquences a commencé à ralentir au début des années 2000, pour atteindre une quasi stagnation de nos jours. D'autres voies ont alors été explorées afin de continuer à améliorer la puissance de calcul des microprocesseurs, l'une d'entre elles étant de réintroduire des capacités de vectorisation dans les microprocesseurs scalaires. Introduites dans un premier temps avec les instructions *MMX* (*MultiMedia eXtension*) de la première génération de processeur Intel Pentium, tous les microprocesseurs actuels les intègrent sous diverses formes grâce à des unités de calcul indépendantes permettant d'effectuer les opérations basiques sur plusieurs valeurs à la fois dans un temps identique ou proche de celui nécessaire à effectuer cette même opération sur une seule valeur scalaire avec les instructions standard du processeur, multipliant ainsi la puissance arithmétique théorique du processeur par la quantité de nombres traitables en une instruction par ces unités.

Dans les architectures x86, le premier jeu d'instructions vectorielles MMX permettait de manipuler des registres de 64 bits contenant plusieurs nombres entiers de 8, 16 ou 32 bits. AMD étendit ce mécanisme avec le jeu d'instructions 3DNow ! qui permettait en plus la manipulation de deux nombres flottants 32 bits, cependant ces instructions connurent un succès limité notamment à cause de contraintes matérielles (registres partagés avec l'unité de calcul en virgule flottante standard) les rendant difficilement utilisables efficacement au sein d'un programme complet. Les instructions *SSE* (*Streaming SIMD Extension*) qui leurs succédèrent introduisirent des registres 128 bits spécifiques dans l'architecture ce qui permit d'effectuer des calculs sur 4 nombres à virgule flottant 32 bits (puis 2 nombres 64 bits à partir des instructions SSE2) avec une grande efficacité et une grande souplesse d'utilisation (car n'entrant pas en conflit avec le code séquentiel). Les jeux d'instructions *AVX* (*Advanced Vector Extensions*) récents sont dans la continuité du SSE en proposant des opérations sur des registres de 256 puis 512 bits.

Le jeu d'instructions *AltiVec* fut mis en place par Motorola et IBM dans leurs familles de processeurs *PowerPC*, notamment le *Cell Broadband Engine*. Ce jeu d'instruction, à l'instar SSE, permet de manipuler des registres 128 bits représentant des entiers de diverses tailles, ou des nombres à virgules flottante de 32 bits.

Les processeurs développés par la société *ARM* (*Advanced RISC Machines*) et principalement présents dans les appareils mobiles et les systèmes embarqués disposent

également de leur jeu d'instruction SIMD nommé NEON. Celui-ci permet la manipulation de nombres entiers de 8 à 64 bits et de nombres flottants 32 bits, stockés dans des registres de 64 ou 128 bits selon les processeurs. La différence notable de ce jeu est dans la nature des instructions SIMD, qui sont spécifiquement destinées aux traitements multimédias tels que la décompression de flux audio ou vidéo sur des architectures à faible fréquence de fonctionnement et consommation électrique.

2.1.3 Processeurs spécialisés

Bien qu'étant de plus en plus performants pour le calcul scientifique, les microprocesseurs doivent toujours être capables d'effectuer toutes les tâches d'un système informatique et restent donc généralistes, des unités de calcul spécialisées furent donc développées et introduites dans les microprocesseurs. Un premier exemple est celui de unités de calcul à virgule flottante (*FPU, Floating Point Unit*) dédiées aux opérations arithmétiques élémentaires sur les nombres à virgule flottante. Ces unités furent dans un premier temps disponibles sous la forme d'un coprocesseur additionnel, avant d'être systématiquement intégrées dans les processeurs.

Processeurs de signaux numériques (DSP : Digital Signal Processor) Apparus dans les années 80, les DSP sont des microprocesseurs spécifiquement dédiés au traitement de signaux numériques. Les fonctionnalités apportées par ces processeurs peuvent être obtenues par un microprocesseur standard, mais les DSP apportent une qualité de service supérieure en permettant d'effectuer des traitements sur des flux continus de données avec une latence très faible tout en ayant des coûts et consommations bien plus faibles que les microprocesseurs. Cette efficacité vient de leur architecture, qui possède de nombreuses fonctions complexes directement câblées dans le circuit intégré. Bien qu'étant très performants sur certains type de calculs scientifiques courants (multiplication de matrices, transformées de Fourier, ...) leur utilisation reste restrictive car ils ne peuvent être exploités en-dehors de leurs fonctions fixes ; ils sont de nos jours principalement utilisés dans des systèmes embarqués.

Circuits logiques programmables (FPGA : Field-Programmable Gate Array) Ces processeurs utilisent une approche différente de tous les autres types de circuits intégrés. Contrairement aux processeurs classiques qui sont constitués d'un ensemble d'unités logiques fixes exécutant les instructions des programmes, les FPGA offrent un ensemble de blocs logiques programmables reliés entre eux par des interconnexions reconfigurables. Cette approche permet aux FPGA d'atteindre une efficacité énergétique quasiment maximale et d'excellentes performances de calcul, à ce niveau ils sont très proches des DSP mais offrent des fonctions programmables ce qui les rends utilisables pour n'importe quelle application. En contrepartie leur programmation est très complexe car il faut décrire l'ensemble de l'architecture effectuant le calcul souhaité. Des approches

hybrides utilisant des FPGA au sein de super-calculateurs existent (tel que le Cray XT5h) mais peinent à s'imposer, notamment à cause de la reconfiguration qui est relativement longue et difficile à mettre en place lorsque plusieurs utilisateurs se partagent le calculateur.

Processeurs graphiques (GPU : Graphics Processing Unit) Dans le même ordre d'idée que les DSP, les calculateurs graphiques sont apparus afin d'exécuter de manière performante les calculs nécessaire à l'affichage graphique, notamment le rendu d'image en trois dimensions par rastérisation (projection dans le plan de l'écran de primitives géométriques simples en trois dimensions, et remplissage pixel par pixel). Leur architecture a naturellement évolué vers un parallélisme massif étant donné leurs tâches très indépendantes et répétitives (calculs indépendants sur des millions d'éléments de géométrie et de pixels à chaque seconde). Cantonnés dans un premier temps à des fonctionnalités fixées très spécifiques, ces unités devinrent programmables afin de permettre aux développeurs d'intégrer de nouveaux effets de rendu. Ces possibilités de programmation ont ouvert la voie à un usage détourné de ces processeurs afin d'effectuer des calculs non spécialisés, étant de par leur nature très efficaces pour les calculs sous forme vectorisée. Cette tendance s'est par la suite confirmée, l'architecture et les outils évoluant vers toujours plus de souplesse de programmation, tout en restant sur un modèle massivement parallèle : les processeurs graphique récents les plus performants contiennent plus d'un millier de coeurs au sein d'une unique carte. Une grande partie des super-calculateurs utilisent désormais ce type de carte qui fournit un excellent rapport performance/prix pour les calculs s'y prêtant bien (notamment les calculs d'algèbre linéaire).

Systèmes intégrés D'autres approches existent où un ensemble d'unités de calculs habituellement séparées sont intégrées au sein d'un unique circuit. On peut noter principalement les approches *System On Chip (SOC)* où les composants d'un ordinateur complet sont intégrés dans un unique chipset. Une de leur évolution est celle des *Network on Chip (NOC)*, qui est un SOC où le modèle de communication entre les composants se base sur les architectures de réseaux au lieu des traditionnels bus de données. Cette approche offre une gestion plus fine des communications dans le SOC et permet un meilleur passage à l'échelle.

L'architecture *MIC* (*Many Integrated Cores*) introduite récemment par Intel avec les processeurs Intel Xeon Phi utilise une approche similaire : un ensemble de coeurs de microprocesseurs classiques (une soixantaine pour les Xeon Phi actuels) communiquant à travers un bus en anneau sont rassemblés au sein d'une même carte. Ces processeurs disposent d'unités SIMD avancées permettant de traiter des données vectorielles de 512 bits de large, donnant à l'ensemble de la carte une très grande puissance de calcul vectorielle, tout en gardant la souplesse d'utilisation des processeurs multi-coeurs. Le Xeon Phi est notamment utilisé dans le super-calculateur Tianhe-2 [1].

1. classé en juin 2013 comme étant le plus puissant du monde sur www.top500.org

En conclusion, le parallélisme s'est généralisé dans les architectures informatiques à tous les niveaux, des périphériques embarqués jusqu'aux super-calculateurs. Ce parallélisme se présente à de nombreux niveaux : parallélisme des machines au sein d'une grappe, parallélisme des unités de calcul au sein d'un noeud, parallélisme des coeurs au sein d'un processeur ou encore parallélisme des données au sein des instructions vectorielles.

L'évolution des architectures s'oriente aujourd'hui vers toujours plus de parallélisme à tous les niveaux (instructions SIMD sur des registres plus larges, nombre de coeurs plus élevés au sein des processeurs, ...), ainsi qu'une tendance forte à l'hétérogénéité des unités de calculs utilisées. Il est nécessaire d'exploiter ce parallélisme pour obtenir les meilleures performances possibles du matériel, mais cela rends la programmation très complexe. De nombreux modèles de programmation spécifiques au parallélisme permettent d'effectuer cette tâche à divers niveaux de facilité et d'efficacité, nous allons en effectuer un tour d'horizon dans la section suivante.

2.2 Modèles de programmation parallèle

Les langages et outils de programmation séquentiels ne permettent pas d'exploiter facilement ou efficacement les diverses capacités de parallélisme offertes par le matériel, des librairies ou langages spécifiques ont donc été créés à cet effet. Les différents modèles existant peuvent être classifiés selon divers axes : certains s'articulent autour de l'architecture matérielle, d'autres mettent en avant l'aspect algorithmique de la parallélisation, et certains modèles visent à faire le lien entre l'aspect algorithmique abstrait et l'architecture matérielle concrète.

2.2.1 Modèles architecturaux

Ces modèles mettent en avant le type d'architecture parallèle ciblée et les mécanismes permettant de l'exploiter.

2.2.1.1 Mémoire partagée

Les modèles de programmation en mémoire partagée se basent très souvent sur les fils d'exécution fournis par les systèmes d'exploitation depuis l'avènement des systèmes multi-tâches. Ceux-ci furent conçus au départ de manière à permettre l'exécution de plusieurs tâches ou programmes en concurrence dans un système possédant un unique processeur séquentiel.

Le standard des **threads POSIX** (ou *pthreads*) [92] est très fréquemment rencontré : ces derniers permettent de créer très simplement divers processus légers dont

l'exécution sera gérée par le système selon une politique d'ordonnancement définie (sur un système ne possédant qu'un unique processeur ou coeur de calcul, un unique *thread* peut être physiquement exécuté à un instant donné, le système leur alloue donc des temps d'exécution très courts à tour de rôle afin de créer une illusion de simultanéité). Les bibliothèques de *pthreads* fournissent également des moyens leur permettant de communiquer et se synchroniser entre eux, afin de pouvoir éviter les situations conflictuelles lors des accès mémoire.

Les **pthreads** permettant l'exécution de plusieurs tâches dans un même environnement, ceux-ci sont naturellement adaptés à l'utilisation du parallélisme dans une architecture à mémoire partagée, la seule nuance étant que dans un système multi-coeurs ou multiprocesseurs, plusieurs *threads* peuvent réellement s'exécuter simultanément à condition d'être situés sur des processeurs ou coeurs différents. Des mécanismes d'affinité aux processeurs ont été ajouté par la suite aux systèmes utilisant les **pthreads** : ceux-ci permettent d'associer explicitement les *threads* aux processeurs afin de garantir une exécution parallèle et éviter leurs déplacements qui peuvent être provoqués par le système. Les **pthreads** sont très performants, mais leur programmation est assez complexe car il faut notamment mettre en place soi-même tous les mécanismes nécessaires pour gérer les synchronisations entre les *threads*, les accès concurrents ou les risques d'inter-blocages. De nombreux langages ou bibliothèques visant à apporter une parallélisation plus simple sont basés sur les **pthreads**.

OpenMP [40, 93] est une interface de programmation permettant d'ajouter du *multi-threading* dans des programmes C/C++ et Fortran à travers l'ajout dans le code de directives de compilation et l'utilisation de variables d'environnement, par exemple pour spécifier dynamiquement le nombre de *threads* à utiliser.

Les directives fournies par OpenMP couvrent de nombreux aspects du parallélisme en mémoire partagée tels que le parallélisme de données (répartition des itérations d'une boucle sur plusieurs *threads*), le parallélisme de tâches [6] (affectation de calculs différents à plusieurs threads), la définition de valeurs privées ou partagées entre *threads*, le définition de politiques d'ordonnancement, les synchronisations, les sections critiques ou opérations atomiques. La dernière version du standard intègre également des directives pour la vectorisation SIMD des calculs.

Bien qu'il soit possible de manipuler explicitement les *threads* à travers leurs identifiants, les directives sont plutôt conçues de manière à mettre en place la parallélisation sur des codes C ou Fortran déjà existants ; on peut donc manipuler séparément la définition du calcul à effectuer (par le code source) et la méthode de parallélisation à utiliser (par les directives de compilation).

Cilk [20] est un langage de programmation parallèle général dérivé du C/C++. Ce langage amène de nouveaux mots-clés permettant au programmeur d'identifier explicitement ce qui peut être calculé en parallèle dans son programme, de mettre en place des barrières de synchronisation ou encore des opérations

atomiques. C'est ensuite l'environnement d'exécution qui découpera automatiquement les calculs à effectuer en différentes tâches selon les contraintes définies dans le code ; la parallélisation en elle-même est donc totalement cachée à l'utilisateur, et ses programmes parallèles peuvent s'exécuter de manière transparente sur n'importe quel ordinateur.

Intel Threading Building Blocks (TBB) [98] proposent une approche de haut niveau à base de *templates* C++. L'utilisateur peut mettre en place divers algorithmes parallèles (boucles ou pipelines par exemple) utilisant des structures concurrentes (vecteurs, files, ...) et des éléments de contrôle (opérations en exclusion mutuelle, opérations atomiques, ...). L'environnement d'exécution se charge ensuite de découper en tâches dépendantes les différents calculs à effectuer par les algorithmes parallèles. Ces tâches sont ensuite réparties sur les différents processeurs disponibles en fonction du graphe des dépendances, et leur répartition sera éventuellement modifiée grâce à un mécanisme de vol de tâche si un déséquilibre dans la répartition des calculs apparaît durant l'exécution.

2.2.1.2 Passage de messages

Les modèles de programmation par passage de message se destinent dans un premier temps aux architectures à mémoire distribuée, où le seul moyen d'échanger des informations entre processeurs est l'envoi de messages sur leur réseau d'interconnexion. Ces modèles peuvent être appliqués dans les architectures à mémoire partagée, mais dans ce cas les processeurs seront malgré tout considérés comme totalement indépendants les uns des autres et les possibilités de communications directes par la mémoire commune ne seront pas exploitables.

Modèle d'acteurs Ce modèle définit un système de programmation concurrente dont la primitive sont les *acteurs* qui communiquent entre eux par l'envoi de messages. Lorsqu'un acteur reçoit un message, celui-ci réagit en effectuant des calculs locaux, en créant de nouveaux acteurs ou en envoyant de nouveaux messages. Ce modèle est à la fois utilisé sous forme théorique dans la recherche en informatique fondamentale, mais aussi comme modèle de programmation concret par exemple dans le langage Scala [58], ou les nombreuses librairies disponibles pour d'autres langages courants. Les modèles multi-agents en sont dérivés, et ce modèle se retrouve appliqué notamment dans les services web et le *cloud computing*.

MPI (Message Passing Interface) [103] est la spécification d'un protocole de communication destiné principalement au calcul scientifique sur machines parallèles à mémoire distribuée grâce à des communications de type point-à-point ou collectives, ayant pour buts d'être performant, portable et capable de passer à l'échelle sur des machines de grande taille. Plusieurs implémentations génériques sont disponibles, mais les constructeurs de super-calculateurs peuvent également fournir des implémentations spécifiquement optimisées pour leurs

machines. MPI a connu un grand succès depuis sa création dans les années 90 et s'impose aujourd'hui comme l'outil le plus populaire pour les communications par passage de message.

Cependant, ce modèle de programmation exploite assez mal la mémoire partagée et on voit fréquemment une utilisation hybride dans les super-calculateurs composés de processeurs multi-coeurs : MPI est utilisé pour mettre en place le parallélisme à gros grain entre les différents noeuds de calcul, et chaque processus MPI est à son tour parallélisé à un grain plus fin au sein de sa mémoire partagée (par exemple en utilisant OpenMP [48]). Ce genre d'approche permet également de limiter la quantité de communications, celles-ci devant être effectuées entre l'ensemble des noeuds et non pas entre l'ensemble des coeurs de calcul.

2.2.1.3 Modèles pour architectures hétérogènes

Ces modèles sont conçus afin de permettre la manipulation d'unités de calcul (telles que les GPU ou les FPGA) travaillant en complément des microprocesseurs dans les architectures hybrides. On peut séparer ces modèles en plusieurs catégories : ceux destinés uniquement à la manipulation d'une architecture particulière (tels que CUDA permettant de programmer les GPU de nVidia), les modèles portables exposant des propriétés génériques des architectures hétérogènes (tels que OpenCL), et les modèles généraux abstrayant l'aspect hybride de l'architecture (tels que OpenACC).

CUDA Le modèle *CUDA* (*Computer Unified Device Architecture*) [37], proposé par le constructeur de cartes graphiques nVidia, est spécifiquement dédié à la programmation de calculs généraux sur accélérateurs graphiques, et peut s'utiliser notamment en C/C++ et Fortran grâce à des compilateurs spécifiques ou des interfaces vers de nombreux autres langages (Perl, Java, Ruby, Python). Vu la nature extrêmement vectorisée de l'architecture, les calculs effectués par l'accélérateur sont définis sous forme de fonctions noyaux écrites dans une syntaxe C simplifiée et qui seront exécutées un grand nombre de fois dans des *threads* parallèles regroupés en blocs implicitement synchronisés.

CUDA met fortement en avant les caractéristiques matérielles des accélérateurs utilisés, ainsi il revient à l'utilisateur d'effectuer explicitement les échanges de données entre la mémoire principale et celle de l'accélérateur, de définir l'organisation des *threads* et le découpage des noyaux sur ceux-ci, la gestion des types de mémoire disponibles (mémoire globale, registres locaux à chaque *thread*, mémoire partagée entre plusieurs *threads* d'un même bloc). Conçu par nVidia, les outils de développement CUDA ne sont utilisables que sur les accélérateurs graphiques de ce constructeur.

OpenCL [91] est un environnement de développement visant à permettre l'écriture de programmes portables sur diverses architectures : les processeurs standards,

les accélérateurs graphiques, les DSP et les FPGA. L'écriture des programmes se fait dans un langage dédié dérivé du C permettant d'écrire des noyaux de calcul mettant en place du parallélisme de données ou de tâches. L'architecture utilisée n'étant pas connue à priori, un ensemble de mots-clés spécifiques permet de manipuler différentes zones de mémoire abstraites génériques (globale, constante, locale ou privée) au sein des noyaux. Une interface de programmation permet ensuite d'exécuter ces noyaux sur une architecture donnée où des espaces concrets en mémoire seront associés aux zones mémoire abstraites demandées par le noyau. Des structures de données vectorielles sont également disponibles, afin de permettre l'utilisation d'instructions SIMD si l'architecture ciblée le permet.

Les programmes OpenCL sont totalement portables entre les diverses architectures supportées, cependant les performances ne le sont pas, et il reste nécessaire d'optimiser le code spécifiquement pour chaque architecture si l'on souhaite obtenir de bonnes performances.

OpenACC [2] est un standard de programmation pour le développement sur architectures hétérogènes CPU/GPU utilisant une approche similaire à OpenMP (un projet de fusion visant à intégrer les mécanismes d'OpenACC dans une prochaine version d'OpenMP est à l'étude [3]) : le programmeur peut ajouter des directives de compilation spécifiques dans un code C/C++ ou Fortran existant afin d'identifier quelles parties du code sont susceptibles d'être exécutées en parallèle. Les détails de l'architecture cible sont ici totalement cachés à l'utilisateur, le découpage en *threads* ou les échanges de données entre mémoires sont donc transparents et l'utilisateur doit simplement préciser des propriétés générales dans les directives de compilation.

Les modèles de programmation parallèle basés sur l'architecture permettent d'obtenir de très bonnes performances car ils donnent la possibilité de contrôler l'exécution du programme sur le matériel. Cependant ceci implique de suffisamment connaître le fonctionnement de l'architecture ciblée pour obtenir un programme efficace, et un programme efficace dans une architecture donnée ne le sera pas nécessairement pour une autre, malgré les performances atteignables par ces modèles elles restent donc difficiles à obtenir et peu portables. Des modèles définissant le parallélisme indépendamment de l'architecture évitent ces inconvénients.

2.2.2 Modèles algorithmiques

Ces modèles s'intéressent à l'aspect algorithmique de la parallélisation, et définissent des méthodes abstraites permettant sa mise en place sans s'intéresser directement aux capacités de parallélisation du matériel.

2. http://www.openacc.org
3. http://www.openacc.org/ ?q=node/49

2.2.2.1 Parallélisme de tâches

Le but dans ce modèle est de diviser l'ensemble du programme à effectuer en un sous-ensemble de tâches éventuellement interdépendantes ; l'aspect parallèle de l'exécution des programmes est donc fortement mis en avant. On peut distinguer principalement deux variantes du parallélisme de tâches : premièrement le modèle MISD (Multiple Instructions Single Data) où les tâches effectuent des traitements différents, mais toujours sur le même ensemble de données. La structure de pipeline est un exemple connu de ce modèle qui reste cependant assez peu répandu pour le calcul haute performance.

À contrario, dans le modèle MIMD (Multiple Instructions Multiple Data) rencontré plus fréquemment chaque tâche travaillera sur des données différentes. Les tâches sont alors totalement indépendantes les unes des autres, et il revient donc au programmeur de mettre en place des mécanismes de synchronisation afin d'assurer la cohérence de l'exécution de son programme.

2.2.2.2 Parallélisme de données

Dans ce type de modèle, on se focalise sur la façon dont les données sont réparties entre les différentes unités de calcul, qui par contre effectueront exactement toutes les mêmes opérations sur leurs données respectives. Ce modèle se place donc à l'opposé du parallélisme de tâches ; l'aspect parallèle des programmes est donc implicite et découle de la répartition des données. On peut distinguer deux niveaux dans le parallélisme de données : SIMD et SPMD.

Dans l'approche SIMD (Single Instruction Multiple Data), une même instruction est appliquée sur plusieurs éléments de données à un instant donné. La synchronisation implicite très forte de ce modèle limite son application aux matériels capables de conserver une synchronisation au niveau des instructions et donc en pratique aux processeurs vectoriels, aux unités vectorielles des microprocesseurs ou aux GPUs.

L'approche SPMD (Single Program Multiple Data) est beaucoup plus relâchée : bien que chaque donnée subisse le même traitement, l'exécution de chacune des instructions du programme se déroule de manière totalement indépendante entre les différentes unités de calcul, et c'est au programmeur de mettre en place les synchronisations nécessaires à la cohérence de son calcul.

2.2.3 Modèles de transition logico-matérielle

Ces modèles sont à cheval sur les approches architecturales et les approches algorithmiques, et visent à fournir au programmeur un modèle abstrait mais prenant néanmoins en compte l'implémentation physique du parallélisme.

PRAM Le modèle de Parallel Random Access Machine [52] vise à décrire de manière abstraite une machine à mémoire partagée, et dérive du modèle RAM utilisé pour modéliser les ordinateurs séquentiels. Les processeurs dans ce modèle sont tous synchronisés et peuvent accéder directement à toute la mémoire partagée. Différentes règles d'accès à la mémoire peuvent être utilisées, mettant en place des accès en lecture ou écriture soit exclusifs, soit autorisant la concurrence. Dans le cas où les écritures concurrentes sont permises, diverses politiques peuvent être mises en place pour résoudre les conflits, telles que la validation uniquement en cas d'écriture identique par tous les processeurs, la priorité au premier processeur voulant écrire ou selon une hiérarchie définie, ou encore par réduction des différentes valeurs écrites. Ce modèle ne peut être implémenté concrètement sur les architectures courantes où des CPU accèdent à une mémoire de type DRAM car cette dernière ne supporte pas les accès concurrents, cependant il est possible d'obtenir une architecture compatible avec les FPGA.

PRAM est largement utilisé en informatique théorique afin de calculer la complexité algorithmique de programmes parallèles en fonction du temps et du nombre de processeurs dans le système, cependant ce modèle est très simplifié : les communications et synchronisations sont ignorées, la mémoire est supposée accessible uniformément en temps constant et ne disposer que d'un unique niveau. Ceci fait que les temps de calcul prévus grâce à ce modèle donnent une idée de la complexité générale du programme, mais sont fortement éloignés de la réalité de l'exécution sur les machines parallèles actuelles.

BSP Le modèle *Bulk Synchronous Parallel* [87, 113] décrit des machines parallèles à travers trois modèles : une architecture, un modèle d'exécution et un modèle de coût. Cette séparation en modèles bien distincts permet de simplifier la tâche à la fois pour les concepteurs des machines parallèles qui n'ont pas à se préoccuper des détails des algorithmes qui y seront implémentés, et également pour les développeurs qui n'auront pas à se préoccuper des détails architecturaux pour concevoir leurs programmes.

Modèle architectural La machine est modélisée comme étant à mémoire distribuée, composée d'un ensemble de processeurs possédant chacun sa propre mémoire locale et communicant entre eux à travers un réseau d'interconnexion. Ce modèle est souple et peut s'adapter à de nombreuses architectures concrètes : bien que correspondant naturellement à une machine parallèle distribuée, il peut tout à fait s'appliquer sur une architecture à mémoire partagée, ou des architecture hybrides, homogènes ou non.

Les caractéristiques d'une machine BSP sont définies par quatre paramètres : **p** est le nombre de processeurs de la machine, **r** la puissance de calcul (en FLOPS) d'un processeur, **L** est le temps nécessaire pour effectuer une synchronisation globale, et **g** la vitesse des communication interprocesseurs (par exemple le débit du réseau dans un architecture distribuée, ou le débit de la mémoire centrale dans une architecture à mémoire partagée). On peut facilement récupérer ces caractéristiques pour une machine

réelle par *benchmarking*, ce qui permet d'obtenir directement sa modélisation BSP.

Modèle d'exécution L'élément de base des programmes dans le modèle BSP est la *super-étape (superstep)* qui consiste en trois phases : les processeurs effectuent localement un ensemble de calculs totalement indépendants, suivis d'une deuxième phase de communications où les processeurs échangent des données entre eux de manière libre. La phase de communication se termine par une barrière de synchronisation globale entre tous les processeurs, une nouvelle super-étape peut ensuite commencer. Un programme BSP est ainsi composé d'une succession de super-étapes.

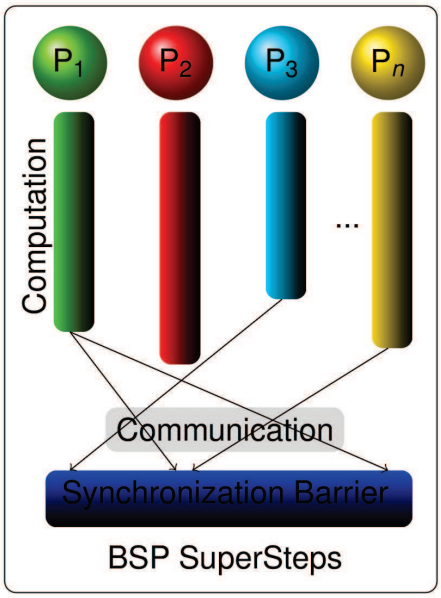


FIGURE 2.1 – *Modèle d'exécution BSP*

Modèle de coût Les trois parties d'une super-étape étant bien découpées, il est facile de donner une borne supérieure sur les temps d'exécution et donc de donner un modèle de coût pour chaque super-étape d'un programme : ce sera la somme de la durée du calcul local le plus long, de la durée de la phase de communication et de la durée de synchronisation. Ce calcul s'écrit généralement sous la forme :

$$\text{Temps}(s) = \max_{0 \leq i < p} w_i^{(s)} + \max_{0 \leq i < p} h_i^{(s)} \times g + L$$

où $w_i^{(s)}$ est le temps nécessaire au processeur i pour effectuer ses calculs locaux dans l'étape s et $h_i^{(s)}$ est le maximum entre les quantités de données recues et envoyées par le processeur i durant l'étape s. Le coût de l'ensemble d'un programme BSP sera donc la somme des coûts des différentes étapes, mais on le retrouve fréquemment exprimé sous la forme de

termes globaux W (la somme des $\max_{0 \leq i < p} w_i^{(s)}$ de toutes les étapes), H (la somme des $\max_{0 \leq i < p} h_i^{(s)}$) et S le nombre d'étapes ce qui permet d'exprimer synthétiquement le coût total par $W + H \times g + S \times L$.

Le modèle BSP a été implémenté sous de nombreuses formes. BSPLib [102] est la librairie C/Fortran de référence, mais on rencontre plus souvent sa variante BSPonMPI [18] qui utilise MPI pour la gestion des communications et est donc beaucoup plus portable et évolutive. Des implémentations visent à donner une utilisation plus abstraite dans des langages de haut niveau, telles que des approches fonctionnelles dans le langage Objective Caml avec la librarie BSML [80].

De par son uniformité, le modèle BSP montre ses limites sur les architectures actuelles, celles-ci n'étant pas uniformes non seulement au niveau des échanges entre processeurs, mais également dans la nature de ces processeurs, et des modèles dérivés ont été engendrés afin de prendre en compte des caractéristiques plus fines dans les architectures matérielles. dBSP (Decomposable BSP) [12] définit une approche à deux niveaux, où une machine BSP peut être vue comme un ensemble de sous-machines BSP indépendantes contenant chacune un nombre p de processeurs, ce qui permet de modéliser des clusters de multiprocesseurs à mémoire partagée en définissant chaque noeud comme une sous-machine BSP. Le modèle BSP++ [60] approfondi cette démarche avec l'apport d'un modèle hiérarchique permettant de décomposer récursivement la machine BSP en un arbre de sous-machines de profondeur arbitraire, chaque niveau possédant son ensemble de paramètres BSP.

2.2.4 Modèles implicites

Certains modèles visent à simplifier au maximum le travail du développeur en lui cachant totalement le parallélisme. Ces modèles de programmation en eux-même ne sont pas parallèles, mais leur implémentation l'est. On trouve dans ces catégories des extensions de langages existant tel que le High Performance Fortran (HPF) [70], mais également des compilateurs ou options de compilation spécifiques, ainsi le compilateur `gcc` pour le langage C permet de mettre en place de la vectorisation automatique de code. Java permet la manipulation d'objets distribués [28,118] se trouvant sur différentes machines virtuelles, moyennant quelques particularités de déclaration ces objets se manipulent de la même manière que les objets habituels. On peut aussi noter la présence de langages dédiés mettant en place des mécanismes de parallélisation afin de résoudre les problèmes de leur domaine d'application [31].

Une catégorie particulière de langages vise explicitement à fournir aux utilisateurs un modèle de programmation structuré exploitant de manière implicite le parallélisme : les squelettes algorithmiques. Ces langages nous intéressent particulièrement car leur approche abstraite de haut niveau permet aux utilisateurs de manipuler très facilement le parallélisme.

2.3 Squelettes algorithmiques

Les squelettes algorithmiques définis par Cole [34] sont des abstractions de schémas apparaissant de manière récurrente dans la programmation parallèle. Nous allons étudier les squelettes rencontrés couramment, avant d'effectuer un tour d'horizon détaillant les principaux langages implémentant ces squelettes.

2.3.1 Classification des squelettes

La définition des squelettes algorithmiques étant très générale, il est possible de définir un squelette à partir de n'importe quel algorithme ou sous-partie d'algorithme. Chaque langage ou bibliothèque à squelette apporte son propre ensemble de squelettes utilisables, et la recherche sur cette thématique étant très active ceux-ci évoluent au cours du temps. Bien qu'aucune norme ne soit définie ni même de consensus informel, un squelette doit répondre à un certain nombre de critères subjectifs pour être considéré utile et implémenté : il doit avoir un champ d'application le plus large possible afin d'avoir une bonne réutilisabilité, sa sémantique doit être simple à comprendre pour l'utilisateur et il ne doit pas être redondant par rapport aux autres squelettes ; c'est pourquoi on retrouvera souvent les mêmes squelettes de base dans la plupart des langages à squelettes. Ceux-ci peuvent être classés en trois grandes familles : les squelettes de parallélisme de données, les squelettes de parallélisme de tâche et les squelettes de résolution.

2.3.1.1 Squelettes de parallélisme de données

Ces squelettes sont conçus afin de permettre la manipulation de structures de données implicitement parallèles. Ceux-ci travaillent à un grain très fin afin d'appliquer à ces données les calculs voulus en prenant en compte la nature parallèle des données, tout en cachant complètement les communications et synchronisations nécessaires à l'utilisateur qui aura uniquement à spécifier les opérations de calcul à effectuer.

Map Il s'agit du squelette le plus connu est probablement le plus simple à comprendre : celui-ci distribue les données au sein de l'architecture parallèle, puis applique à chaque élément une fonction passée en paramètre, produisant ainsi un nouvel ensemble contenant le même nombre d'éléments que l'ensemble de données de départ. La sémantique de Map demande que la fonction puisse s'appliquer indépendamment pour chaque élément, ainsi le squelette peut appliquer cette fonction parallèlement sur de nombreux éléments sans qu'il y ait besoin de communications ; Map peut être ainsi vu comme une abstraction du modèle SIMD.

Zip est une extension de Map, son comportement est identique à ceci près qu'il demande deux ensembles de données de la même taille et une fonction binaire qui

sera appliquée à chacune des paires formées par les éléments de même position dans les deux ensembles. On peut également trouver des formes généralisées appliquant une fonction n-aire à n ensembles.

Fork Ce squelette est similaire à Map mais permet d'appliquer à chaque élément une fonction différente, permettant ainsi d'appliquer un modèle de calcul de type MIMD.

Reduce/Scan Ces squelettes appliquent une fonction binaire associative successivement à tous les éléments d'un ensemble en le parcourant dans son ordre naturel. Les données sont distribuées en parallèle tout comme dans Map, cependant des communications doivent être mises en place étant donné que l'application de la fonction à un nouvel élément demande le résultat des applications précédentes. Dans Reduce seul le résultat final est retourné, alors que Scan renverra l'ensemble des résultats partiels.

Shift/Permute Ces squelettes n'effectuent pas de calcul, mais uniquement des communications ayant pour but de modifier l'ordre des éléments d'un ensemble de données. Permute effectuera l'échange de positions entre plusieurs éléments alors que Shift effectuera un décalage de tout l'ensemble, ce décalage peut être circulaire ou bien introduire de nouveaux éléments pour remplacer ceux ayant été éliminés aux extrémités de l'ensemble.

2.3.1.2 Squelettes de parallélisme de tâches

Ces squelettes modélisent des méthodes pouvant être utilisées pour faire interagir plusieurs tâches, ces squelettes permettent donc de décrire le flux d'exécution des programmes. Ceux-ci peuvent mettre en place un parallélisme à des niveaux variables selon la finesse de modélisation du flux d'exécution.

Pipeline applique un calcul sur un ensemble donné sous la forme d'une série d'étapes successives totalement indépendantes. Le parallélisme se situe dans l'exécution des différentes étapes : une fois qu'un élément de donnée a fini d'être traité dans l'étape e, celui-ci entre donc dans l'étape $e+1$ et l'élément suivant peut simultanément entrer à son tour dans l'étape e, on aura donc un élément en train d'être calculé dans chacune des étapes dès lors que le premier élément aura atteint la dernière étape. Ce modèle est très répandu en programmation séquentielle et est classiquement utilisé dans la programmation des processus systèmes.

Farm décrit un programme sous la forme d'un ensemble de tâches totalement indépendantes dont l'exécution sera parallélisée suivant un modèle maître/esclave dans lequel un maître possède l'ensemble des tâches à effectuer qu'il déléguera aux esclaves en leur fournissant à chacun une tâche dès lors que la précédente lui ayant été affectée aura été terminée.

For/While/If se comportent avec les tâches de manière analogue aux instructions classiques homonymes en permettant de répéter une tâche un certain nombre de fois (défini explicitement ou par l'évaluation d'une condition), ou le choix d'un branchement entre diverses tâches.

2.3.1.3 Squelettes de résolution

Ces squelettes modélisent des méthodes algorithmiques spécialisées apportant la solution à des familles de problèmes spécifiques. Leur nature et leurs mécanismes de parallélisation peuvent donc être très variés. Deux squelettes de ce type sont fréquemment rencontrés :

Divide and conquer peut être vu comme un Map récursif dont le but est de partitionner les éléments tant qu'un critère d'optimisation n'est pas satisfait. Lorsqu'un ensemble de données à traiter arrive, si le critère est satisfait un sous-squelette est appliqué sur les données, sinon celles-ci sont divisées en sous-ensembles par une fonction de subdivision. Lorsque la récursion se termine (par satisfaction du critère, par atteinte d'une profondeur fixée ou par impossibilité de subdiviser à nouveau), une fonction de fusion est appliquée afin de synthétiser en un unique résultat global tous les résultats partiels obtenus au sein des sous-ensembles créés lors des subdivisions.

Branch and bound modélise une approche assez similaire afin de parcourir un espace de solutions, qui est dans un premier temps divisé en sous-espaces (étape *branch*) dans lesquels une fonction de coût se voit appliquée. Les résultats de cette fonction de coût sont comparés afin d'éliminer les sous-espaces ne contenant pas de solutions potentielles, et le squelette est ainsi appliqué récursivement jusqu'à l'obtention d'une solution exacte.

2.3.2 Langages et bibliothèques de squelettes

Les squelettes étant des fonctions de haut niveau, la mise en place d'un environnement de programmation par squelettes algorithmiques nécessite de pouvoir manipuler des fonctions ou leurs équivalents. On les rencontre donc fréquemment sous la forme de bibliothèques ou de langages dédiés permettant d'utiliser les fonctions et donc principalement basés sur les paradigmes de programmation fonctionnelle, impérative ou orientée objet. Une autre approche est celle des langages de coordination, utilisant un langage spécifique pour la manipulation des squelettes et un langage courant pour leur implémentation.

2.3.2.1 Coordination

Dans cette approche, la développement d'un programme à squelettes se fait par le biais de deux langages : un langage de haut niveau permettant de définir le comportement de l'application par le biais des squelettes, et un langage hôte gérant les interactions directes avec l'architecture matérielle. La séparation est donc totale entre la description abstraite de l'application et son implémentation, cependant cela implique l'apprentissage du nouveau langage de coordination et la mise en place d'outils de traduction et de compilation dédiés.

SCL (Structured Coordination Language) [45] est une des premières implémentations concrètes du modèle de langage à squelettes. Ce langage utilise une approche par composant en fournissant à l'utilisateur un ensemble de squelettes utilisés pour coordonner entre eux des modules écrits dans un langage séquentiel de base (typiquement C ou Fortran), la description du programme en tant qu'assemblage de squelettes n'est donc pas liée au langage de base utilisé. SCL met en place des structures de données distribuées manipulables par des squelettes de configuration, permet d'effectuer des calculs grâce à des squelettes de parallélisme de données, qui peuvent ensuite être combinés par l'utilisation de squelettes manipulant les tâches tels que farm ou pipe.

P3l (Pisa Parallel Programming Language) [95] introduit également un langage de coordination dans lequel sont définis l'ensemble des squelettes que l'utilisateur peut manipuler afin de coordonner l'exécution séquentielle ou parallèle d'éléments de code dans un langage hôte (l'implémentation par défaut utilise le C). Un compilateur spécifique convertit le code P3L en programme exécutable par le biais de *templates* permettant de spécifier des implémentations de squelettes variant en fonction de l'architecture ciblée. Lors de la génération des *templates*, un réseau de processus modélisant l'application est construit puis optimisé de manière globale, et ce sont les *templates* correspondant au réseau optimisé qui seront réellement compilés.

SkIE (Skeleton-based Integrated Environment) [7] est construit sur la même base technique que P3L, mais vise une plus grande facilité d'utilisation permettant un développement rapide d'applications. Des outils de développement supplémentaires sont fournis tels qu'un débuggueur, un analyseur de performance ou encore un outil graphique permettant de spécifier la composition des squelettes formant un programme remplaçant donc l'utilisation du langage de coordination. L'utilisation de codes séquentiels en C, Fortran ou Java mais également la définition de modules parallèles en MPI et HPF sont également supportés.

ASSIST [114] reprends une approche similaire à SCL : ce langage de coordination structuré permet de définir une application sous la forme d'un graphe décrivant un ensemble de composants et les flux de données leur permettant d'interagir entre eux. Les modules peuvent soit être séquentiels et écrits en C, C++ ou Fortran, soit être parallèles en instanciant le module *parmod* d'ASSIST. Une autre

composante du langage est l'environnement d'exécution qui est spécifique et fournit des méthodes permettant de manipuler les flux de données. Ciblant principalement les grilles de calcul, l'environnement d'exécution permet de mettre en place automatiquement une allocation dynamique des ressources et des mécanismes d'équilibrage de charge.

2.3.2.2 Impératifs

Ces langages prennent la forme de bibliothèques venant enrichir un langage impératif hôte tel que C ou Fortran, ce qui permet d'introduire facilement leurs mécanismes dans des programmes existant tout en conservant les caractéristiques et performances du langage hôte. Un défaut assez important de ce paradigme est qu'il ne permet pas de mettre en place une véritable vérification de type ce qui est important dans les langages à squelettes si l'on veut pouvoir vérifier que les programmes des utilisateurs combinent les squelettes et utilisent les données générées (qui peuvent rapidement atteindre une complexité importante au niveau du type) de manière cohérente.

eSkel (Edinburgh Skeleton Library) [11, 36] est une bibliothèque C construite sur les primitives de MPI. eSkel donne une grande liberté à l'utilisateur en proposant plusieurs modes d'utilisation des squelettes, notamment à travers deux *modes d'interaction* : dans le mode implicite les flux de données entre les squelettes sont uniquement définis par leur composition, alors que dans le mode explicite l'utilisateur peut intervenir directement sur ces flux de données. La syntaxe des squelettes est donc assez complexe et bas niveau afin de permettre à l'utilisateur de manipuler les échanges de données explicitement à travers les primitives MPI. Des travaux ont été effectués sur la base de ce langage afin de définir des mécanismes de prédiction de performances sur les programmes à squelettes [10, 120].

SKELib [43] reprend les principes développés dans P3L et SkIE mais en éliminant le langage de coordination, l'ensemble des squelettes est accessible directement sous la forme d'une bibliothèque de fonctions en C utilisant TCP/IP pour les communications, et les calculs applicables par les squelettes doivent être eux aussi sous la forme de fonctions C classiques prenant en arguments des pointeurs vers les données. SKELib est un des premiers langages à squelettes totalement inclus dans une bibliothèque C.

Skil (Skeleton Imperative Language) [21–23] étend le langage C afin de mettre en place des mécanismes de programmation fonctionnelle, les squelettes sont donc introduits sous la forme de fonctions polymorphiques d'ordre supérieur supportant l'application partielle grâce à un mécanisme de curryfication. Toutes les parties des programmes utilisant les extensions fonctionnelles sont ensuite converties en code C standard avant d'être compilées. Skil propose des squelettes s'appliquant à des structures de données parallèles, notamment des tableaux de taille fixe ou variable.

2.3.2.3 Orientés objets

Les squelettes sont intégrés sous la forme de classes dans les langages objets tels que C++ ou Java, en profitant généralement des capacités d'abstraction de ces langages permettant d'introduire un surcoût faible à l'utilisation des squelettes.

Lithium [3,44] est une bibliothèque apportant des squelettes de parallélisme de tâches et de données au langage Java, en utilisant la RMI (invocation de méthodes distantes) afin de manipuler plusieurs processus se divisant le travail de calcul. L'implémentation des programmes à squelettes est effectuée avec la méthode des *Macro Data Flow (MDF)* : les squelettes composant le programme sont analysés afin d'extraire un graphe modélisant les flux de données. Les noeuds du graphe sont les *Macro Data Flow instructions (MDFi)* et correspondent à des blocs de code exécutables dès lors que tous les flux de données entrant sont présents. Les MDFi dans Lithium correspondent aux codes séquentiels dont l'exécution est demandée aux squelettes parallèles. L'environnement d'exécution se charge ensuite de répartir l'exécution des MDFi selon l'évolution des flux de données, en les organisant de manière à optimiser les communications.

Muskel [2] est la continuation de Lithium dont il reprends les principes en en étendant les fonctionnalités en permettant la mise en place de nouveaux squelettes, et en introduisant des mécanismes de tolérance aux pannes ou encore d'ajustement des performances.

Muesli (Munster Skeleton Library) [71] est la continuation de Skil et vise à intégrer les nouveautés présentes dans les autres langages à squelettes. Un des premiers soucis est l'intégration au langage, Muesli est donc une bibliothèque C++ utilisant des patrons de fonctions pour mettre en place les mécanismes fonctionnels introduits dans Skil, et implémenter les squelettes sous la forme de fonctions polymorphes d'ordre supérieur. Les squelettes de Muesli permettent de manipuler des structures de tableaux et matrices distribués et également la mise en place de parallélisme de tâches, ainsi que des algorithmes de résolution généraux (*Branch and Bound, Divide and Conquer*) ; la gestion de la distribution des données et des calculs se fait grâce à MPI, et des travaux plus récents apportent un support des processeurs multi-cœurs grâce à OpenMP [33].

Calcium et Skandium Calcium [77] apporte un ensemble de squelettes pour le parallélisme de données et de tâches dans le langage Java ; les programmes écrit avec ces squelettes peuvent ensuite être déployés sur diverses infrastructures telles que les multiprocesseurs à mémoire partagée et les ordonnanceurs sur grilles de calcul. Cette bibliothèque fournit notamment un modèle d'optimisation des performances afin d'aider les développeurs à identifier les appels de squelettes peu performants [29], et des squelettes dédiés à la manipulation parallèle des fichiers [30]. Skandium [79] est la continuation de Calcium et étend son fonctionnement afin d'exploiter efficacement les architectures multi-cœurs.

QUAFF [51] est une bibliothèque C++/MPI de squelettes de parallélisme de tâches se

basant sur des techniques de métaprogrammation permettant de mettre en place automatiquement des optimisations de code lors de la compilation et d'éliminer les surcoûts générés par les appels de squelettes. L'exécution est modélisée dans un réseau de processus sur lesquels les différentes tâches composant le programme seront placées en fonction de règles de production.

SkeTo [84] mets en place des squelettes de parallélisme de données en C++ et permet la manipulation de matrices, arbres et listes distribués, de plus les listes peuvent être de taille variable. La technique de métaprogrammation des *expression templates* est utilisée pour mettre en place la fusion de fonctions, ainsi les appels imbriqués à plusieurs squelettes sont résolus à la compilation en un unique appel de fonction. Les architectures multi-cœurs sont également exploitées [68] notamment pour les opérations sur les matrices distribuées par un mécanisme d'allocation automatique de tâches.

Intel TBB [98] Bien que ne s'identifiant pas comme un langage à squelette à proprement parler, certains des algorithmes parallèles introduits par les TBB sont très similaires à des squelettes courants de parallélisme de données ou de tâches, et visent le même but de parallélisation implicite des calculs que les langages à squelettes en cachant les détails des communications ou synchronisations.

HPC++ (High Performance C++) [67] apporte des classes et patrons définissant des primitives de synchronisation et communication, ainsi qu'une implémentation parallèle de la Standard Template Library.

2.3.2.4 Fonctionnels

Le modèle de programmation par squelettes algorithmiques est par définition fonctionnel, les langages utilisant ce paradigme sont donc des candidats idéaux pour l'implémentation des bibliothèques de squelettes. Cependant les langages fonctionnels sont habituellement peu utilisés pour le calcul haute performance, hors il s'agit du domaine d'application essentiel des langages à squelettes étant donné leur utilisation implicite du parallélisme ; on retrouve donc finalement assez peu de langages à squelettes utilisant les langages fonctionnels.

Eden [26, 27] étend le langage fonctionnel Haskell en permettant la manipulation de processus parallèles distribués à un grain très fin. Les communications entre processus sont rendues implicites par l'utilisation de canaux abstraits faisant le lien entre deux processus. Les squelettes sont introduits comme un niveau supérieur d'abstraction implémenté au-dessus des processus. Ce modèle est facilement extensible, ainsi il existe une implémentation sous Eden du modèle MapReduce de Google [15].

HDC (Higher-order Divide and Conquer) [64] reprends un sous-ensemble de Haskell où les programmes sont écrits sous la forme de fonctions polymorphes d'ordre supérieur, centrées autour d'un schéma « Diviser et Conquérir ». A partir de ce

schéma général, plusieurs cas particuliers tels que des récursions de profondeur fixe ou des opérations élément par élément sont extraits par le compilateur qui peut générer des implémentations efficaces qui seront liées au programme principal qui sera transformé en C/MPI puis compilé.

OCamlP3L [42] implémente les squelettes de P3L au sein du langage fonctionnel OCaml, ce qui permet de profiter entre autres de son système de typage très puissant. La sémantique fonctionnelle est également utilisée afin de générer plusieurs interprétations concrètes dans le but de faciliter le développement et le débuggage : à partir d'un même programme fonctionnel il est possible de générer un programme séquentiel et donc déterministe, un programme parallèle calculant le même résultat en parallèle sur un ensemble de noeuds, et une interprétation graphique produisant un schéma du réseau de calcul mis en place par l'interprétation parallèle.

Skipper [100] introduit des squelettes ciblant la vision par ordinateur dans le langage Caml et bénéficie de son système de typage fort. Les squelettes de Skipper sont de deux catégories : les squelettes déclaratifs utilisés par les programmeurs pour créer les programmes à squelettes, et leurs équivalents opérationnels fournissant des implémentations spécifiques aux architectures matérielles ciblées qui seront transformés en C avant compilation.

Parmap [46] est une bibliothèque minimaliste pour OCaml. En partant du constat que le langage définit déjà sur les listes des opérateurs *map* et *fold* dont la sémantique est identique à celle des squelettes éponymes, cette bibliothèque propose de remplacer ces opérateurs par des appels de squelettes permettant ainsi la création de programmes parallèles pour architectures multi-cœurs en n'effectuant que des modifications minimales sur les programmes fonctionnels.

2.3.2.5 Squelettes pour architectures hybrides

Ces langages à squelettes permettent d'exploiter les architectures contenant des unités de calculs de natures différentes, principalement dans la forme du couple courant CPU/GPU.

SkePU [49] est une bibliothèque C++ ciblant plusieurs architectures : les processeurs séquentiels, les multi-cœurs à travers OpenMP et les GPU à travers Cuda, et l'hybridation à travers OpenCL. SkePU mets en place du parallélisme de données grâce à une structure de vecteur dérivée de son homonyme de la STL, et sur laquelle des variantes de Map et Reduce peuvent être appliquées. Les fonctions utilisateurs appliquées par les squelettes sont écrites à l'aide de macros, qui génèrent le véritable code utilisé en fonction de l'architecture ciblée. La gestion des échanges de données avec les GPU est effectuée implicitement par les vecteurs, et un mécanisme de *copie paresseuse* permet de les optimiser en effectuant les copies de données uniquement aux moments opportuns, ce qui permet d'appliquer plusieurs squelettes à la suite sur des données restant dans la mémoire

d'un GPU. La présence de multiples GPU dans un système est exploitée par une répartition automatique et équilibrée des données des vecteurs sur ceux-ci.

SkelCL [104] utilise une approche similaire à SkePU se basant uniquement sur OpenCL : il est possible d'effectuer du parallélisme de données sur GPU avec des structures de vecteurs ou de matrices en utilisant les squelettes habituels (Map, Zip, Reduce et Scan) ainsi que des calculs sur le voisinage des élements avec le squelette MapOverlap. SkelCL gère les systèmes multi-GPU en répartissant les calculs des squelettes, et propose différents modes de répartitions des données : copie totale, découpage en blocs séparés, et découpage en blocs avec recouvrement partiel qui est essentiel à l'application correcte du squelette MapOverlap.

Muesli a été récemment étendu afin de supporter des architectures hybrides [50] et apporte maintenant un support pour les GPU à travers CUDA, en plus des approches MPI et OpenMP existant déjà dans cette bibliothèque. L'utilisation des GPU se fait par le biais de nouveaux types représentant des vecteurs et matrices utilisables par ces processeurs et sur lesquels il est possible d'appliquer les squelettes de parallélisme de données avec la même syntaxe que sur les données utilisées sur les CPU habituels.

Nous avons effectué un tour d'horizon des langages à squelettes existant, ceux-ci couvrent un large champ d'applications en fournissant une grande facilité d'utilisation tout en exploitant la performances des architectures parallèles. Néanmoins, il est difficile de définir précisément à quel point la programmation dans ces langages simplifie la tâche des développeurs, et quelles sont les performances qu'ils peuvent en retirer. Nous allons donc étudier dans la section suivante les méthodes nous permettant de quantifier ces deux aspects dans le développement de programmes parallèles.

2.4 Performance et complexité de développement

2.4.1 Mesures de performance

Le *benchmarking* est la méthode habituellement utilisée pour procéder à des mesures quantitatives lors des expérimentations en informatique. L'idée de base est d'effectuer une mesure du temps d'exécution d'un programme ou d'une sous-partie représentative sur un matériel donné. Ce temps d'exécution mesuré est ensuite utilisé pour évaluer diverses métriques telles que le temps d'exécution total du programme (en secondes ou millisecondes), le débit des échanges de données ou de traitement (en bits par secondes) ou la puissance de calcul (en nombre d'opérations de calcul par seconde). Ces métriques sont ensuite interprétées en fonction du but recherché, qui est

très souvent l'évaluation du gain de performance apporté par un nouvel algorithme ou un nouveau matériel.

Le besoin de mesurer le temps d'exécution d'un algorithme donné sur un matériel particulier provient du fait qu'il est quasiment impossible de le prédire précisément. Il est possible de déterminer la complexité algorithmique, mais ensuite il nous faut savoir précisément comment le compilateur traduira cet algorithme en langage machine, et combien de temps prendra le matériel ciblé a exécuter ce code compilé. Cela implique donc de connaître très précisément comment le matériel fonctionne en interne, ce qui est généralement au-delà des connaissances du développeur moyen, de plus les fabricants de matériel informatique ne donneront pas forcément accès à toutes les informations nécessaires. Quand bien même cet environnement serait maîtrisé, plusieurs couches de complexité viennent s'ajouter sur les ordinateurs modernes : l'accès à la mémoire à travers plusieurs niveaux de cache (et leurs mécanismes de préchargement), les actions du système d'exploitation, les processus qui peuvent s'exécuter en concurrence de notre algorithme, les communication entre noeuds d'un réseau ou les processeurs au sein d'une grappe de calcul, etc. Tout cela entraîne une très grande difficulté à prédire le temps d'exécution d'un algorithme, alors que le *benchmarking* est simple à mettre en place et donne implicitement les résultats réels.

Le *benchmarking* est très souvent utilisé afin d'obtenir une estimation de la puissance de calcul d'un matériel donné. Comme expliqué précédemment, nous ne pouvons pas prédire de manière très réaliste la performance d'un algorithme sur un matériel donné quand bien même le fabricant en donnerait toutes les spécifications. D'un autre côté, ce dernier ne peut pas s'attendre à ce que les utilisateurs finaux implémentent leurs applications ou algorithmes sur ce nouveau matériel dans le seul but de mesurer ses performances, c'est pourquoi des outils de *benchmarking* firent leur apparition, afin de fournir un moyen d'évaluation de performance « prêt à l'usage ».

2.4.1.1 Premiers outils

Il existe une quantité presque infinie de programmes différents pouvant être développés sur n'importe quel ordinateur, et chacun d'entre eux est susceptible d'utiliser de manière intensive différentes parties du système. Certains peuvent demander des calculs en virgules flottantes intensifs, d'autres peuvent avoir besoin d'accéder rapidement à de grands jeux de données en mémoire, certains peuvent demander beaucoup d'entrées/sorties, ou encore du rendu graphique. Cette variété entraîne qu'il est impossible de concevoir un programme de *benchmarking* universel, à la place nous avons besoin d'outils synthétiques visant à représenter les différents besoins possibles des utilisateurs.

Les premiers outils de *benchmarking* étaient spécifiquement destinés au calcul scientifique, étant donné que les premiers supercalculateurs étaient extrêmement coûteux, et leur achat uniquement justifiable par les laboratoires de recherche. Un des outils

mathématiques les plus fréquemment utilisés est l'algèbre linéaire, les premiers benchmarks leur furent donc dédiés. Linpack [65] est la référence communément admise dans ce domaine ; son concept est relativement simple : il évalue le temps nécessaire à la résolution d'un système d'équations linéaires grâce à la méthode du pivot de Gauss. Ce *benchmark* se base sur BLAS [72], une bibliothèque très optimisée de fonctions basiques d'algèbre linéaire qui existe pour la plupart des architectures informatiques, et fournit donc un bon aperçu des performances auxquelles on peut s'attendre pour les applications faisant un usage intensif de l'algèbre linéaire.

2.4.1.2 Architectures parallèles

Les processeurs vectoriels sont apparus dans les années 70 et devinrent rapidement la norme au sein des supercalculateurs. Ceux-ci offraient un grand gain en puissance de calcul par rapport au processeurs scalaires existant en offrant la possibilité d'exécuter des calculs en parallèle sur de grands vecteurs de données, cependant les *benchmarks* existant précédemment n'étaient plus valides pour évaluer ceux-ci étant donné qu'ils furent conçus pour évaluer la performance de calcul séquentielle, ce qui n'est pas représentatif des capacités d'une architecture vectorielle. En conséquence, de nouveaux *benchmarks* parallèles furent conçus afin de déterminer la performance réelle des calculateurs vectoriels en prenant en compte leurs particularités et contraintes.

Ces machines étant toujours principalement utilisées pour le même genre de calculs lourds en opérations algébriques, un successeur à Linpack adapté à celles-ci fut créé : LAPACK [4]. Celui-ci effectue le même type de calculs algébriques, mais exploite les capacités de parallélisation des processeurs vectoriels. Les opérations sur des vecteurs ou matrices sont très efficaces sur ces architectures grâce à leur facilité de parallélisation, mais ceci ne sera pas vrai pour n'importe quel algorithme en général, le *benchmark* LAPACK n'est donc pas représentatif de la performance d'une application entière quelconque sur un processeur vectoriel.

Les Livermore Loops [88] ciblent ce problème, en fournissant 24 boucles typiques de code Fortran directement extraites de véritables codes opérationnels. Ces boucles présentent divers niveaux de parallélisme et visent à fournir un plus large aperçu des performances en calcul arithmétique des ordinateurs parallèles.

Le *benchmark* NAS [8] cible plus spécifiquement la mécanique des fluides numérique. Il consistait au départ en 5 noyaux de calcul ayant pour but d'utiliser de manière intensive les différents aspects d'un calculateur parallèle (de la puissance de calcul pure aux communications, en passant par des mélanges intermédiaires). Une particularité de ces noyaux est qu'ils ne possèdent pas d'implémentation de référence, seules des spécifications « à la main » sont fournies, ainsi les fabricants de matériels peuvent développer leurs propres implémentations optimisées. Ce *benchmark* a évolué et inclus également dorénavant des applications complètes résolvant des problèmes simplifiés de mécaniques des fluides par la résolution de système d'équation différentielles par-

tielles. Ces applications utilisent les noyaux précédents, mais mettent également en place des mécanismes d'échanges de données entre ces derniers. Étant donné que ce *benchmark* possède uniquement une spécification formelle, il est totalement indépendant des architectures matérielles et est toujours utilisé de nos jours sur les multiprocesseurs multicœurs modernes.

2.4.1.3 Évolution vers les multiprocesseurs

Les microprocesseurs scalaires introduits dans les années 80 furent rapidement utilisés afin de remplacer les processeurs vectoriels, sous la forme de multiprocesseurs à mémoire partagée ou de grappes à mémoire distribuée. Dans une grappe, les processeurs ne peuvent plus accéder directement aux données des autres processeurs et il faut donc mettre en place des mécanismes de communication dans les programmes exploitant ce type d'architecture. Des bibliothèques parmi lesquelles la désormais standard MPI (*Message Passing Interface*) sont apparues afin de mener à bien cette tâche, et les outils de *benchmarking* ont dû être adaptés à ce mode de programmation.

LAPACK fut adapté à cette architecture à travers MPI, cette version fut renommée SCALAPACK (Scalable LAPACK) [19]. Les benchmarks NAS restèrent utilisables pour cette architecture de par leur spécification abstraite.

Les suites SPLASH [101] et SPLASH-2 [119] implémentèrent des concepts similaires afin d'évaluer les performances de machines à mémoire partagée à travers un ensemble de noyaux mathématiques (transformée de Fourier rapide 1D, décomposition matricielle LU, tri par base de nombre entier, etc) et des applications entières avec un forte orientation vers le rendu graphique telles que du rendu par radiosité, lancer de rayon ou rendu volumique.

La Standard Performance Evaluation Corporation (SPEC) a également produit plusieurs *benchmarks* de référence au fil du temps, notamment un *benchmark* de référence pour CPU : SpecCPU [63], qui fût parallélisé au travers d'OpenMP pour les architectures à mémoire partagée (SpecOMP2001 [5]) et MPI pour les architectures distribuées (SpecMPI2007 [90]) afin d'évaluer leurs performances, à travers un ensemble d'applications de calcul à haute performance.

La suite ParkBench [14] étends le concept de NAS pour les multiprocesseurs parallèles. Celle-ci comprend toujours un ensemble de *kernels* simples exécutant des calculs scientifiques courants sur les matrices (multiplication, transposition, factorisation LU, décomposition QR), transformée de Fourier rapide (sur une ou trois dimensions), systèmes d'équations différentielles partielles, ainsi que certains des noyaux de NAS. Cette suite fournit également un large spectre de simulations scientifiques simplifiées telles que la mécanique des fluides, la dynamique moléculaire, la physique des plasma ou la chimie quantique. La différence essentielle avec NAS apparaît dans l'introduction d'un nouvel ensemble de benchmarks de bas niveau mesurant des propriétés du système telles que la puissance de calcul arithmétique, les goulots d'étranglements

potentiels lors des accès à la mémoire ou à ses divers niveaux de cache, la vitesse des échanges de message, le débit disponible ou encore les coûts de synchronisation. Mesurer ces caractéristiques est important sur une architecture multiprocesseur car elles sont susceptibles de varier considérablement d'un calculateur à l'autre, et les développeurs peuvent avoir besoin de ces informations pour optimiser finement leurs programmes. Cette suite est cependant dépassée aujourd'hui, sa dernière mise à jour datant de 1996.

Une suite de *benchmarking* plus récentes est PARSEC [16]. Celle-ci cible plus particulièrement les architectures de type processeurs multi-coeurs qui sont la tendance dominante actuellement. Étant donné que ce type de processeur est devenu le standard jusqu'aux machines de particuliers voire même aux machines embarquées ou téléphones portables, ceux-ci se retrouvent utilisés dans de nombreux domaines et ne sont plus limités aux calculs scientifiques, cette suite a donc un objectif beaucoup plus généraliste que les outils précédents. Elle inclut une grande diversité d'applications telles que l'analyse financière, la fouille de données ou la traitement de données multimédias ; il revient ensuite à l'utilisateur de sélectionner les applications répondant le mieux à ses besoins pour effectuer le benchmarking. Ces applications sont implémentées à travers divers modèles de parallélisme : les threads systèmes classiques, OpenMP ou encore les Thread Building Blocks (TBB) ; le modèle de programmation par tube (pipeline) est également utilisé dans 4 applications. Cette suite de benchmarking ne fournit donc pas seulement une évaluation des performances, mais donne également des indications sur quel modèle de programmation serait le mieux adapté pour un type d'application particulier sur l'architecture ciblée. PARSEC est considéré actuellement comme une référence solide dans l'évaluation de performance des architectures parallèles basées sur les processeurs multi-coeurs.

2.4.1.4 Utilisations alternatives du *benchmarking*

Le but principal des outils de *benchmarking* a toujours été de mesurer les performances d'un ordinateur ou d'une architecture donnée ; c'est un sujet critique notamment pour les super-calculateurs parallèles étant donné l'investissement très lourd qu'ils peuvent représenter. Cependant, les méthodes de *benchmarking* peuvent être utilisées à d'autres fins.

Récupérer des paramètres ou des spécifications précises du matériel utilisé peut se montrer difficile voire impossible à mettre en place de manière automatique par une application car ils peuvent être cachés par le système ou bien être d'accès restreint. X-Ray Tool [121] utilise un ensemble de micro-benchmarks pour estimer divers paramètres matériels d'un processeur tels que sa fréquence, sa latence d'exécution des instructions, son nombre de registres, ou encore la taille et la latence de ses caches. Cette idée a été étendue pour les grappes de processeurs multi-cœurs dans la suite Servet [55]. Un premier ensemble de microbenchmarks a pour but de détecter l'existence et la taille des différents niveaux de mémoire cache et leur partage entre les coeurs, ce

qui fournit une caractérisation complète de la hiérarchie mémoire. D'autres *benchmarks* étudient ensuite comment risquent de se former des goulots d'étranglement lorsque plusieurs coeurs tentent d'accéder simultanément à la mémoire centrale. Un dernier benchmark étudie les coûts de communication entre les coeurs. De par leur nature, les chemins de communications au sein de grappes de processeurs multi-cœurs varient en longueur entre les coeurs, ceux-ci se trouvent donc classés en plusieurs couches de manière similaire aux mémoires caches. Servet a comme but principal de simplifier l'utilisation de système d'*autotuning*, le résultats de ce *benchmark* permettant d'entrée de jeu de générer une application aux paramètres optimisés pour le système.

Une autre utilisation originale des techniques de *benchmarking* est introduite dans [85], où les auteurs utilisent les *benchmarks* SPEC afin d'étudier l'impact de modifications dans le système sur la variabilité des temps d'exécution. Quelques résultats de ce papiers montrent que l'environnement du terminal Unix a globalement très peu d'impact sur la variance ; la parallélisation d'une application séquentielles en une application utilisant plusieurs fils d'exécution quant à elle entraîne une grande variabilité (jusque 30% dans les tests considérés) lorsque le placement des threads est laissé au système, mais que cette variabilité dans certains cas se retrouvait supprimée en définissant une politique de placement des fils d'exécution. Un autre résultat intéressant est qu'avoir d'autres applications concurrentes qui s'exécutent sur le même système a un effet globalement très bénéfique sur la variabilité ; une explication avancée étant que dans un système plus chargé, la probabilité est plus faible d'avoir un accès concurrent aux données par plusieurs fils d'exécution d'un même programme, on obtiendrait donc un système plus régulier et plus performant dans sa globalité car le risque d'avoir des coeurs bloqués par des accès mémoires concurrents est réduit.

2.4.1.5 Évaluation statistique des résultats

Les *benchmarks* effectuent souvent les mesures une seule fois pour obtenir leurs résultats, cependant le temps d'exécution d'un programme donné sur un architecture donnée ne sera jamais constant à cause de nombreux facteurs difficiles ou impossibles à contrôler tels que les tâches systèmes tournant en arrière-plan, la politique de placement des fils d'exécution, la planification des tâches par le système, les interruptions matérielles, etc. Comme exposé dans [85], les architectures multi-cœurs tendent à exacerber cette variabilité à cause de facteurs supplémentaires tels que l'ordonnancement des *threads*, les barrières de synchronisation ou encore les accès concurrents à la mémoire.
Dans [112] les auteurs affirment qu'il est très difficile de reproduire les temps d'exécution donné dans une expérience, notamment à cause de leur nature souvent floue : s'agit-il d'une seule mesure ? de la moyenne d'un ensemble ? De la meilleure exécution ? Pour résoudre ce problème, les auteurs proposent d'analyser les temps d'exécution grâce à des outils d'analyse statistique. L'idée générale étant de mesurer deux implémentations différentes d'une même application, de calculer un facteur de gain

de temps entre ces versions et un taux de confiance sur ce facteur de gain. Chaque implémentation doit être exécutée plusieurs fois afin d'obtenir un ensemble de temps d'exécution sur lequel il est possible d'effectuer deux test : le t-test de Student sur les moyennes, ou le test Wilcoxon-Mann-Whitney sur les médianes. Chaque test donne en résultat un taux de confiance sur le rapport mesuré entre les variables concernées (moyennes ou médianes), c'est à dire la probabilité que l'ajout d'une nouvelle mesure dans l'ensemble ne modifie pas cette valeur. Le second est préféré car la médiane des temps d'exécution est stable et peu sensible aux valeurs extrêmes, de plus le test est moins exigeant sur la distribution de l'ensemble des temps mesurés et est applicable la majorité du temps. Bien qu'il faille un nombre infini de mesures pour fournir une valeur exacte du gain de performance entre deux implémentations, cette méthode nous assure d'obtenir une valeur dont le taux de confiance est élevé pour peu qu'un nombre suffisant (30 mesures est communément admis comme étant la taille d'un ensemble statistiquement significatif) d'exécutions ait été mesuré.

2.4.2 Effort de programmation

Nous nous intéressons au gains de performances obtenables par la parallélisation des algorithmes, mais nous nous intéressons également à la difficulté à mettre en place les mécanismes de parallélisation permettant d'obtenir ces performances, et les avantages que peuvent procurer les approches de haut niveau de ce point de vue. Mesurer la difficulté à mettre en place les mécanismes de parallélisation, c'est vouloir mesurer le travail supplémentaire à fournir par le développeur ; nous nous sommes donc intéressés aux métriques logicielles qui ont pour but de caractériser des propriétés de programmes par des mesures précises.

Il existe une grande variété de métriques logicielles couvrant divers besoins. Des outils de haut niveaux telles que l'Analyse des Points de Fonction [1] ou le Modèle de Coût Constructif [69] sont souvent utilisés dans le conception de logiciels afin d'estimer les coûts de développement d'une application à partir des fonctionnalités qu'elle devra proposer à l'utilisateur final.

Ce type de métrique ne nous intéresse pas car nous voulons pouvoir comparer plusieurs implémentations fournissant la même fonctionnalité mais avec des performances différentes. D'autres métriques analysent le comportement de l'application, par exemple la complexité cyclomatique [86] qui construit un graphe des différents flux d'exécution possibles et extrait des mesures caractéristiques à partir de ce graphe. Là encore ce type de métrique nous donnera peu d'informations utiles car on peut s'attendre à peu voire à aucune différence entre les graphes de plusieurs implémentations d'un même algorithme.

Ce dont nous avons besoin en fait est une métrique mesurant la quantité de travail fournie pour l'écriture du code de l'algorithme en lui-même pour chacune des implémentations. Quelques métriques telles que le nombre de lignes de code ou le nombre

d'instructions donne une mesure basique de cette propriété, mais ce n'est qu'un aperçu très approximatif du travail réel fourni par le développeur, une implémentation plus complexe ne se traduisant pas forcément par un plus grand nombre d'instructions. Halstead [59] a proposé un ensemble de métriques s'appliquant sur le code source d'un programme et estimant sa difficulté d'écriture. Nous avons choisi d'utiliser ces métriques car elles répondent parfaitement à nos besoins.

2.4.2.1 Métriques de Halstead

Les métriques de Halstead se basent sur le dénombrement des opérateurs et opérandes utilisés dans le code. De par leur nature, ces métriques peuvent être définies pour n'importe quel langage de programmation bien qu'elles soient plus adaptées aux langages impératifs. Il n'y a pas de spécification de référence sur ce qui est défini comme opérateur ou opérande pour un langage donné, nous utiliserons donc pour le langage C++ une définition raisonnable qui pourrait se résumer comme suit : les opérandes seront les types, les constantes et les identifiants définis par l'utilisateur, alors que les opérateurs seront les *storage class specifiers* (`static`, `virtual`, `inline`,...), les qualificateurs de type (`const`, `friend`, ...), les instructions spécifiques du langage (for, if, struct,namespace, typename, ...) ainsi que tous les opérateurs arithmétiques et logiques (+, ==, &&, $\rightarrow$, ...). Le symbole délimiteur `;` les paires de parenthèses ainsi que les appels de fonctions seront également considérés comme des opérateurs.

En appliquant ces définitions sur un code source, nous obtenons quatre mesures de base (table 2.1) sur le nombre et les occurrences des opérateurs et opérandes. A partir de ces mesures, nous pouvons calculer les métriques de Halstead (table 2.2). Les deux premières sont le vocabulaire et la longueur du programme, qui sont explicites. La troisième est le volume du programme, qui se base sur la longueur du programme, mais également sur le nombre total d'opérateurs utilisés. Ainsi, une implémentation utilisant autant d'instructions mais moins d'opérandes aura un volume plus faible ; cette mesure peut être vue comme une estimation de l'empreinte du code du programme en mémoire. La métrique suivante est la difficulté, celle-ci se base sur le nombre d'opérateurs et sur la fréquence d'apparition des opérandes. Cette métrique donne intuitivement un aperçu de la complexité du code : un code utilisant moins d'opérateurs différents et réutilisant au maximum les opérandes sera plus simple à comprendre. La dernière métrique est celle qui nous intéresse le plus : l'effort. Consistant simplement en le produit du volume et de la difficulté, cette métrique est censée être proportionnelle au temps de développement réel, et correspond à l'intuition naïve que l'on peut en avoir : plus un algorithme est long et complexe, plus il prendra de temps à être implémenté. Il est à noter que la complexité est un facteur important ici : un algorithme très long mais très simple ne demandera pas un effort particulièrement important.

Symbole	Mesure
N_1	nombre total d'opérateurs
N_2	nombre total d'opérandes
η_1	nombre d'opérateurs distincts
η_2	nombre d'opérandes distincts

TABLE 2.1 – *Mesures directes*

Symbole	Valeur	Métrique
η	$\eta_1 + \eta_2$	Vocabulaire
N	$N_1 + N_2$	Longueur
V	$N \times \log_2 \eta$	Volume
D	$\frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$	Difficulté
E	$D \times V$	Effort

TABLE 2.2 – *Métriques de Halstead*

2.5 Conclusion

Nous avons pu voir que le parallélisme se généralise dans le matériel pour tous les usages, qu'il devient de plus en plus complexe (hétérogénéité et multiples niveaux) et que cette tendance va s'amplifier dans le futur. De nombreux modèles de programmation parallèles permettent d'exploiter les diverses possibilités de parallélisme existantes, parmi lesquels les langages à squelettes qui nous intéressent de par leur nature très abstraite. Les modèles de programmation parallèle permettent d'exploiter le matériel avec une facilité et des résultats variables, nous avons donc décrit une méthodologie visant à mesurer de manière reproductible les performances et coûts de développement entraînés par l'utilisation un modèle de programmation donné. Nous allons appliquer dans le chapitre 3 cette méthodologie sur l'optimisation d'un calcul classique en prenant progressivement en compte les diverses caractéristiques matérielles d'une architecture représentative, ce qui nous permettra d'analyser sur ce cas particulier la productivité obtenue par différents modèles.

Notre étude des langages à squelettes montre que ceux-ci présentent des caractéristiques très variables : ils peuvent exister en tant que langages indépendants ou sous la forme de bibliothèques dans des langages existants, ils se basent sur des bibliothèques diverses pour manipuler le parallélisme au niveau matériel, les squelettes fournis permettent du parallélisme de tâches ou de données, ... Cette analyse de l'existant nous a conduit à choisir d'améliorer le fonctionnement interne et d'étendre les fonctionnalités de la bibliothèque OSL. Nous rappelons les choix de conception qui ont guidé la conception de la première version d'OSL par Noman Javed.

Le but premier de la programmation par squelettes est de permettre aux utilisateurs un développement facile d'applications parallèles, OSL doit donc utiliser un langage de programmation courant afin de demander peu d'apprentissage. Pour produire des programmes performants, ce langage doit être capable d'exploiter efficacement le matériel dans les application parallèles, et ne demander qu'un surcoût minimal lors de l'utilisation d'une bibliothèque de fonctions. Le choix s'est porté sur le C++ qui est un langage connu et en pleine évolution, parmi les plus performants et permettant grâce aux techniques de métaprogrammation la mise en place de surcouches de programmation ayant un impact négligeable sur les performances. Celui-ci est également très portable, et s'interface facilement avec les bibliothèques de parallélisme

courantes telles que MPI, qui est utilisé afin de gérer les communications nécessaires à l'application de certains squelettes.

OSL vise à cacher le plus possible le parallélisme sous-jacent, les squelettes fourniront donc du parallélisme de données afin de travailler sur des structures implicitement distribuées. Afin de pouvoir exprimer des programmes complets comme un unique assemblage de squelette, OSL doit également offrir la possibilité d'imbriquer les squelettes. Permettre un typage sûr est également une fonctionnalité importante pour faciliter la tâche des utilisateurs, en leur assurant que les programmes à squelettes compilés sont cohérents. Finalement, OSL doit être facilement extensible afin de pouvoir évoluer et répondre à de nouveaux besoins.

À la lumière de l'état de l'art que nous venons d'effectuer, ces choix de conception restent entièrement valides par rapport à notre approche. Nos travaux sur OSL, décrits dans les chapitres 4 à 7, se font donc dans la continuité de la version existant précédemment, bien qu'une grande partie de la bibliothèque ait été réimplémentée pour résoudre certaines limitations concernant les choix de conception, notamment des problèmes de performance et d'extensibilité.

3 Cas d'étude : optimisation d'un calcul de multiplication de matrices

Sommaire

3.1 Programme séquentiel . . . 42
3.1.1 Optimisation des accès mémoire . . . 43
3.1.2 Utilisation de la mémoire cache : algorithme par blocs . . . 44
3.2 Vectorisation . . . 45
3.2.1 Alignement mémoire . . . 46
3.2.2 Transposition . . . 47
3.2.3 Algorithme . . . 48
3.3 Memoire partagée . . . 50
3.4 Analyse de l'effort de programmation . . . 52
3.5 Conclusion . . . 55

Une version préliminaire des travaux présentés dans ce chapitre a été publiée dans les actes de la conférence ICA3PP 2012 [74].

Nous avons dans le chapitre précédent défini une méthodologie afin d'évaluer de manière reproductible les gains de performances et les coûts de développement de la réimplémentation d'un algorithme donné en utilisant divers modèles de programmation. Nous allons effectuer dans ce chapitre une étude pratique de l'application de cette méthodologie sur l'implémentation, l'optimisation et la parallélisation d'un algorithme classique de multiplication de matrices afin de mettre en comparaison les coûts de développement et les gains obtenus sur un cas concret fréquemment rencontré dans le calcul scientifique.

La multiplication de matrices pleines est une tâche conceptuellement simple, mais c'est un mécanisme central de l'algèbre linéaire fréquemment utilisé par exemple dans le traitement d'images, les simulations physiques ou économiques, et peut représenter

le plus gros de la charge de calcul dans ces applications. La définition du calcul est très simple : étant donné une matrice A de taille m (lignes) par k (colonnes) et une matrice B de taille k par n, la matrice C résultant de leur multiplication est obtenue en calculant le produit scalaire entre les lignes de A et les colonnes de B

$$C_{i,j} = \left(\ AB \ \right)_{i,j} = \sum_{k=1}^{n} A_{i,k} \times B_{k,j} \tag{3.1}$$

Pour calculer un seul élément de C (équation 3.1), il faut effectuer k multiplications et $k-1$ additions. C contenant $m \times n$ éléments, ceci amène donc à un total de $m \times n \times (2k-1)$ opérations, si n, m, and k sont du même ordre de grandeur, la complexité algorithmique de cet algorithme sera donc $O(n^3)$.

D'autres algorithmes ont été développés afin de diminuer ce facteur de complexité cubique. Le premier fut l'algorithme de Strassen [105] obtenant une complexité de $O(n^{2.81})$. Cependant cet algorithme applique une division récursive de la matrice en sous-matrices et n'est applicable que sur des matrices carrées, et rajoute de nombreuses opérations mémoires pour créer ces sous-matrices récursives. Comme nous le verrons par la suite, les accès mémoire tendent à être le principal goulot d'étranglement de cet algorithme, et dans la variante de Strassen les données ont une localité très faible ce qui rend difficile l'optimisation des accès mémoire et la parallélisation des calculs. Des résultats expérimentaux [41] ont montré que cet algorithme ne devenait plus efficace (sur un seul processeur) que l'algorithme de base que pour de très grandes matrices (de l'ordre du milliard d'éléments). Coppersmith-Winograd [39] ont conçu un autre algorithme qui a été amélioré jusqu'à atteindre une complexité de $O(n^{2.3727})$, mais cette complexité est handicapée par une constante multiplicative très importante, ne rendant cet algorithme utile que pour des matrices à la taille improbable en pratique.

3.1 Programme séquentiel

L'implémentation de l'algorithme en C++ est très directe : Il nous faut deux boucles imbriquées itérant sur les tailles m et n pour couvrir tous les éléments de la matrice C. Pour chacun des éléments il nous faut une boucle de taille k pour effectuer le produit scalaire entre la ligne correspondante dans A et la colonne dans B, comme décrit dans le listing 3.1.

Cette approche se montre très inefficace principalement à cause du schéma d'accès à la mémoire : pour chaque itération de la boucle interne, nous avons trois accès en lecture et un accès en écriture dans la mémoire pour seulement deux opérations arithmétiques. Sur les architectures matérielles actuelles, les opérations arithmétiques sont considérées rapides (seulement quelques cycles de l'unité de calcul) alors que les accès à la mémoire sont lents (jusqu'à plusieurs centaines de cycles). Bien que les optimisations du compilateur et les mécanismes de mémoire cache puissent réduire l'impact de ces accès à la mémoire, les unités de calcul vont passer le plus clair de

leur temps à attendre que les données soient disponibles puisque le ratio opérations arithmétiques/opérations mémoires est faible. Le nombre d'opérations arithmétiques nécessaires dans cet algorithme étant fixe, il convient de réduire autant que faire se peut les opérations sur la mémoire si l'on souhaite exploiter au mieux la puissance de calcul disponible.

Listing 3.1 – *Multiplication de matrices*

```
for (int i = 0; i < C→ _height; i++) {
    for (int j = 0; j < C→ _width; j++) {
       for (int k = 0; k < A→ _width; k++)
          C→ data(i,j) =
             C→ data(i,j) +
               A→ data(i,k) * B→ data(k,j);
    }
 }
```

3.1.1 Optimisation des accès mémoire

Si nous considérons la boucle interne, l'élément $C_{i,j}$ en train d'être calculé contient le résultat du produit scalaire en cours à chaque itération, cependant seule la valeur finale à la fin de la boucle a réellement besoin d'être écrite en mémoire. Au lieu d'y écrire chaque valeur de la somme partielle, nous pouvons stocker cette dernière dans un registre du processeur, qui est un petit élément de mémoire localisé à l'intérieur même du processeur et dont l'accès entraîne une latence très faible, voire nulle. En procédant ainsi, nous n'avons plus qu'à effectuer une seule écriture à la fin de la boucle interne et nous supprimons ainsi les accès en lecture à la matrice C. Nous réduisons donc d'un tiers le nombre d'accès en lecture, et divisons par k le nombre d'écritures.

Listing 3.2 – *Multiplication de matrices*

```
for (int i = 0; i < C→ _height; i++) {
    for (int j = 0; j < C→ _width; j++) {
       float accumulateur = 0.;
       for (int k = 0; k < A→ _width; k++)
          accumulateur =
             accumulateur +
               A→ data(i,k) * B→ data(k,j);
       }
       C→ data(i,j) = accumulateur;
    }
 }
```

3.1.2 Utilisation de la mémoire cache : algorithme par blocs

Nous avons réduit au minimum les accès à la matrice C, mais nous pouvons encore améliorer les lectures dans A et B : chaque ligne de A sera lue pour chacune des colonnes de B et inversement, mais ces données étant fixes il n'y aurait besoin idéalement que de les lire une unique fois. Il n'existe que quelques registres dans un processeur, il n'est donc pas possible d'y conserver une matrice. Les processeurs actuels possèdent également une mémoire cache de taille plus conséquente (de l'ordre de quelques mégaoctets au maximum) mais entraînant une plus grande latence d'utilisation (de l'ordre de la dizaine de cycles processeurs). Elle se place donc comme intermédiaire entre la mémoire principale et les registres. Si des données sont conservées dans le cache, après une première lecture lente en mémoire principale, les accès suivants seront rapides. L'utilisation de la mémoire cache est transparente, il s'agit donc de mettre en place des stratégies d'accès aux données favorisant son utilisation.

Une autre caractéristique intéressante de la mémoire cache est qu'elle récupère les données en mémoire centrale par lignes de cache, c'est à dire qu'un bloc ou ligne de données (habituellement 64 octets sur les architectures actuelles) est lu dans son intégralité dès lors que l'on veut accéder à une donnée présente dans ce bloc. Lorsqu'on accède à plusieurs valeurs contenues dans la même ligne, la lecture de la première valeur souffre de la latence totale de la mémoire centrale, mais les valeurs suivantes présentent dans la ligne sont ensuite accessibles avec une latence faible.

Étant donné que nous accédons aux éléments de A ligne par ligne, nous bénéficions automatiquement de ce mécanisme. Par contre nous accédons aux éléments de B par colonne, c'est à dire de manière non contigüe en mémoire. Cependant nous récupérerons quand même dans la ligne de cache les éléments contigus dans la mémoire, qui seront d'autres éléments de la matrice (tant qu'on atteint pas la fin de la zone de mémoire). On récupère donc d'autres éléments qui seront utilisés plus loin, mais ceux-ci risquent d'être effacés par le chargement d'autres lignes si la taille du cache ne permet pas de conserver toute la matrice.

Si A et B rentrent entièrement dans le cache, ces mécanismes résolvent complètement notre problème : chaque élément sera lu une seule fois dans la mémoire principale, puis restera dans le cache et tous les accès se feront avec une latence faible. Cependant la mémoire cache reste limitée en taille et ne peut contenir des grandes matrices en entier, une solution est de diviser les matrices en sous-matrices de tailles adaptées au cache, et d'adapter l'algorithme à cette structure, ce qui donne l'algorithme de multiplication de matrices par blocs [47].

Dans cet algorithme, on ne considère plus C comme un ensemble de nombres obtenus par le produit scalaire d'une ligne de A par une colonne de B, mais comme un ensemble de sous-matrices (ou *blocs*) obtenus par la somme des produits matriciels d'une ligne de blocs de A avec une colonne de blocs de B (Figure 3.1).

Si les blocs sont suffisamment petits pour être entièrement copiés dans le cache,

leur multiplication sera optimale en accès mémoire. Même si la multiplication complète des matrices n'est pas optimale, elle est composée de sous-parties qui le sont. Par exemple, si C contient 1600 lignes, l'algorithme de base lira chaque colonne de B 1600 fois, alors que si on utilise l'algorithme par blocs avec des blocs de 16 lignes, chaque bloc de B ne sera lu que 100 fois.

Un désavantage de cette approche est qu'il est nécessaire de copier les blocs en tant que matrices indépendantes, afin de regrouper les données de la sous-matrice extraite en un unique bloc continu en mémoire et bien aligné. Cependant pour des matrices suffisamment grandes les blocs sont réutilisés un grand nombre de fois, et les gains obtenus compenseront le surcoût de la copie. Effectuer une copie des données des blocs à chaque fois qu'ils sont utilisés entraîne déjà un grand gain de performance. Cependant cela demande toujours un grand nombre d'opérations mémoires inutiles, nous mettons donc en place une structure permanente conservant tous les blocs une fois créés, ainsi même si ils doivent être lus plusieurs fois, les blocs ne seront écrits qu'une seule fois en mémoire. De plus, plusieurs niveaux de cache sont disponibles sur les architectures récentes, chaque niveau étant plus grand mais plus lent que le précédent. Si nous conservons les blocs dans la mémoire, nous avons une probabilité non négligeable que les blocs soient encore dans un niveau de cache supérieur lorsqu'il faudra y accéder à nouveau.

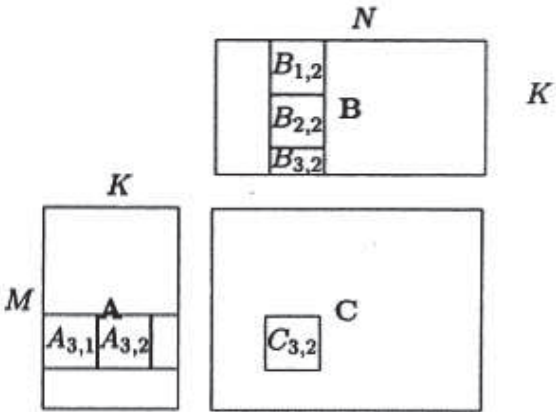


FIGURE 3.1 – *Découpage par blocs*

3.2 VECTORISATION

Dans la section précédente, nous avons optimisé les accès à la mémoire pour l'algorithme de multiplication de matrices. De par sa nature, nous ne pouvons pas réduire le nombre d'opérations arithmétiques de notre algorithme ; cependant nous pouvons utiliser des techniques de parallélisation afin d'augmenter le nombre d'opérations effectuées à chaque seconde. Une première possibilité est d'exploiter les unités vectorielles

SIMD [97] disponibles dans tous les processeurs modernes et permettant d'appliquer une même instruction simultanément à plusieurs éléments de données. Nous nous concentrons ici sur l'utilisation des unités SSE (*Streaming SIMD Extensions*) disponibles sur les processeurs de la famille x86 [106].

Les unités SSE possèdent des registres dédiés de 128 bits pouvant être utilisés par exemple pour stocker 4 nombres à virgule flottante en simple précision (32 bits), ou 2 nombres en double précision (64 bits). Ces registres sont au nombre de 8 sur les processeurs x86, et 16 dans l'architecture x86-64, et les unités SSE disposent d'instructions spécifiques à leur manipulation. Ces instructions possèdent des temps d'exécution similaires à leurs équivalentes scalaires, ce qui nous permet par exemple d'obtenir des calculs théoriquement 4 fois plus rapides si l'on travaille sur 4 nombres à la fois. Existant à la base dans le langage assembleur, ces instructions sont accessibles dans le langage C à travers des fonctions intrinsèques opérant sur des types de données de 128 bits pouvant être stockés dans les registres SSE. Les fonctions sont nommées selon le schéma suivant : `_mm_instr_suffix`, où `instr` est le nom de l'instruction et `suffix` décrit le type des données et si l'instruction doit s'appliquer sur toutes les données du registre ou sur une seule valeur. Les types de données sont nommés `__m128`, `__m128d` et `__m128i`, et représentent respectivement des registres ou des zones de taille correspondante en mémoire principale contenant 4 nombres à virgule flottante en simple précision, 2 nombres à virgule flottante en double précision, et des nombres entiers. Par exemple, la fonction `_mm_add_ps(__m128 r1, __m128 r2)` effectue l'addition simultanée des 4 valeurs (`packed`) flottantes en simple précision (`single precision`) de `r1` avec celles de `r2`.

3.2.1 Alignement mémoire

Travailler efficacement avec les instructions SSE ajoute des contraintes sur l'organisation des données en mémoire : un chargement efficace des données dans un registre SSE ne peut se faire que si le bloc de données dans la mémoire est aligné sur une adresse multiple de 16 octets. Ceci permet au bus faisant transiter les données d'effectuer cet échange en un nombre minimal d'opérations car ce dernier accède à la mémoire par des blocs d'une taille de 64 et alignés sur 64 bits.

Des instructions existent qui permettent de faire des chargements non-alignés, mais leur utilisation entraîne une dégradation des performances car elles demanderont plus d'accès à la mémoire ainsi qu'une réorganisation des données lues, alors qu'une opération de chargement aligné s'effectue de manière directe. Des fonctions telles que `posix_memalign` ou `_mm_malloc` permettent d'allouer de la mémoire suivant une contrainte d'alignement passée en paramètres.

Ceci nous assure que la première ligne d'une de nos matrices sera correctement alignée, cependant si nous stockons les données de la matrice linéairement en mémoire, les lignes suivantes risquent de ne plus être alignées si la taille d'une ligne n'est pas un

multiple de l'alignement demandé. Pour contourner ce problème, nous ajoutons artificiellement le nombre nécessaire de zéros à la fin de chaque ligne. Cette technique est appelée communément *padding* ou remplissage. Étant donné que les valeurs servent à effectuer un produit scalaire, ces zéros n'ont pas d'incidence sur le résultat final vu que les résultats de leurs multiplications seront aussi des zéros qui n'influenceront pas l'addition finale, et nous permettent d'exploiter au mieux les transferts de données.

3.2.2 Transposition

L'accès aux éléments de B reste problématique car il se fait de manière non-contigüe de par leur organisation en colonnes. Accéder à un élément isolé en mémoire reste possible mais très peu performant en SSE. Pour conserver des chargements optimaux, nous pouvons appréhender le problème plus globalement et charger 4 éléments contigus dans 4 lignes se suivant, chargeant ainsi de façon alignée une sous-matrice 4×4 dans 4 registres SSE. Cette sous-matrice est stockée par ligne dans les registres, mais nous pouvons réarranger le contenu de ces registres avec des instructions de permutation afin d'effectuer une transposition et d'obtenir ainsi une vision « par colonne », ou encore plus simplement en utilisant la macro prédéfinie `_MM_TRANSPOSE4_PS` disponible dans les fichiers d'en-tête des fonctions intrinsèques SSE. Ces opérations ne faisant intervenir que des registres ont un coût très faible. De plus leur exécution est facilement recouverte par les opérations mémoire ou arithmétiques.

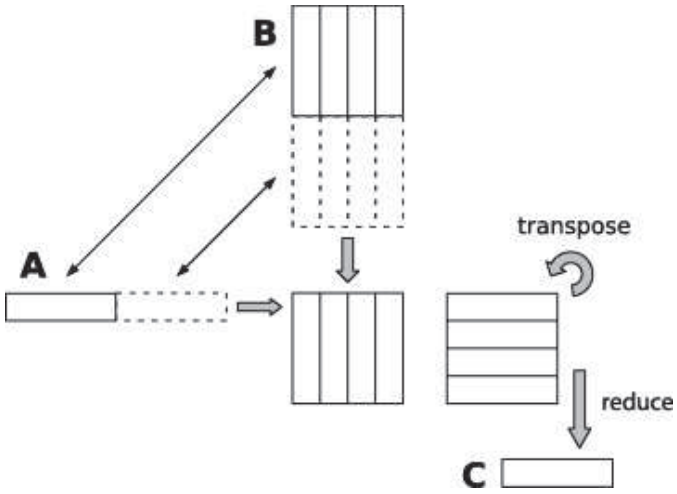


Figure 3.2 – *Mécanisme général en SSE*

3.2.3 Algorithme

Listing 3.3 – *Multiplication de matrices en SSE*

```
1  __m128 vA, vB[4], accumulator[4];
2  for(unsigned int i = 0 ; i < C→ height() ; i++) {
3    for(unsigned int j = 0 ; j < C→ width() ; j+=4) {
4      for(unsigned int k = 0 ; k < A→ width() ; k+=4) {
5        for(unsigned int l = 0 ; l < 4 ; l++)
6          vB[l] = _mm_load_ps(&B[k+l][j]);
7        _MM_TRANSPOSE4_PS(vB[0], vB[1], vB[2], vB[3]);
8        vA = _mm_load_ps(&A[i][k]));
9        for(unsigned int l = 0 ; l < 4 ; l++)
10         accumulator[l] = _mm_add_ps(accumulator,
11                                     _mm_mul_ps(vA, vB[l]));
12     }
13     for(unsigned int l = 0 ; l < 4 ; l++) {
14       accumulator[l] <- _mm_hadd_ps(_mm_hadd_ps(accumulator[l]));
15       _mm_store_ss(C[i][j+l], accumulator[l]);
16     }
17   }
18 }
```

Le cœur de notre algorithme vectorisé en SSE est donné dans le listing 3.3. La boucle externe itère sur les éléments de C ; étant donné que nous récupérons les éléments de B dans 4 colonnes à la fois, nous pouvons traiter 4 éléments de C à chaque itération. La boucle interne itère sur les éléments d'une ligne de A, là encore nous traitons les éléments 4 par 4 et à chacune de ces itérations nous chargeons depuis B (listing 3.3, lignes 5,6) puis transposons (ligne 7) un bloc 4×4 dans les registres `vB`. Nous chargeons ensuite 4 éléments de A dans le registre `vA` (ligne 8). Nous pouvons maintenant effectuer la multiplication de `vA` avec chacun des `vB` grâce à l'instruction `_mm_mul_ps`. Nous ajoutons (instruction `_mm_add_ps`) ensuite le résultat de ces multiplications à 4 registres utilisés comme accumulateurs (lignes 9,10) qui ont été initialisés à zéro en début de boucle (instruction `_mm_set_zero_ps`). À la fin de la ligne, nous possédons pour chacune des 4 colonnes de B traitées un registre accumulateur contenant 4 valeurs étant chacune une partie du produit scalaire de la ligne complète. Il nous reste à additionner ces 4 valeurs entre elles (ligne 13, grâce à l'opération d'addition horizontale `_mm_hadd_ps` appliquée deux fois) pour obtenir le produit scalaire de la ligne au complet, que nous écrivons finalement dans C (ligne 14).

Au final, pour une ligne de k éléments de A nous aurons appliqué $(k-1)$ additions, k multiplications, 3 additions horizontales, $k \times (4+1)$ lectures en mémoire et environ 1 écriture en mémoire. Nous avons donc calculé environ 4 valeurs finales de C en utilisant $2 \times k + 3$ opérations mathématiques et $5k + 1$ opérations en mémoire, ce

qui représente respectivement 25% et 62.5% du nombre d'opérations utilisées dans la version séquentielle.

De même que nous traitons 4 colonnes à la fois, il est possible de travailler à plus gros grain au niveau des lignes : si nous voulons traiter une deuxième ligne sur la même boucle interne, il nous faudra 4 registres accumulateurs supplémentaires, le registre `vA` pouvant être réutilisé directement après avoir calculé les produits pour la première ligne, et la sous-matrice 4×4 de B est réutilisée telle quelle. Ce fonctionnement peut être étendu pour un nombre quelconque de lignes à condition d'avoir suffisamment de registres, voir même au-delà car lorsque l'on utilise plus de données de type `__m128` qu'il n'y a de registres disponibles, le compilateur choisi de placer les données dans la pile, et auront donc une forte probabilité d'être dans les premiers niveaux de cache et donc d'accès rapide. On peut voir dans cette approche une version locale de la méthode par blocs, appliquée cette fois au registres SSE. Les gains de cette modification se feront sentir au niveau des accès mémoire car on divisera le nombre de lectures dans B par le nombre de lignes traitées simultanément.

Notre implémentation SSE nous a permis d'optimiser le temps de calcul et de réduire dans une certaines mesure les accès mémoire, nous pouvons maintenant la combiner de manière transparente avec l'algorithme par blocs. Les gains des deux approches sont complémentaires : l'approche combinée offre un gain de performance très supérieur à la multiplication des gains pris individuellement. Ceci laisse suggérer que l'algorithme SSE est limité par les accès mémoire, alors que l'algorithme par blocs est limité par la puissance de calcul ; la combinaison des deux repousse donc leurs limites respectives pour un résultat supérieur.

La figure 3.3 donne un aperçu des gains mesurés pour les différentes optimisations effectuées jusque ici. Ces mesures de performances ont été effectuées en suivant la procédure décrite dans le chapitre 2.4.1.5 : chaque calcul subit 30 exécutions indépendantes afin d'obtenir un échantillon représentatif des temps d'exécution, et la valeur représentative retenue sera la médiane de chaque échantillon ; le facteur de gain entre deux variantes du programme sera donc le rapport entre les médianes des échantillons concernés. Un test statistique est ensuite appliqué sur les échantillons afin de déterminer le taux de confiance, c'est à dire la probabilité que l'ajout d'une nouvelle exécution à l'échantillon n'ait pas d'influence sur le facteur de gain calculé. Les résultats présentés ici ont tous été obtenus avec un taux de confiance minimal fixé à 95%.

Ces mesures ont été effectuées sur la machine SPEED et représentent la moyenne des gains mesurés pour des matrices de tailles variant de 256 par 256 à 2048 par 2048. Dans le cas de l'algorithme par blocs, la taille de ces derniers a été fixée à 32 par 32. Cette taille a été choisie car elle permet de conserver un bloc de A, de B et de C simultanément dans la mémoire cache, et des tests individuels faisant varier la taille du cache ont confirmé l'efficacité de ce choix.

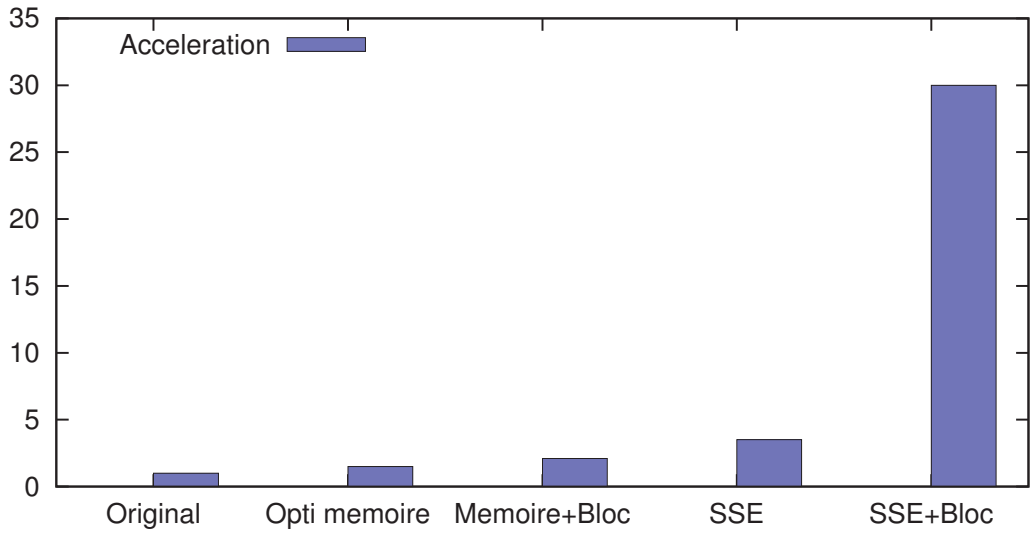


FIGURE 3.3 – *Gains des différentes optimisations*

3.3 Memoire partagée

Nous avons mis en place un mécanisme de vectorisation pour optimiser les calculs, cependant il ne s'agit que d'un parallélisme local à un simple processeur ou cœur de calcul, l'algorithme reste fondamentalement séquentiel. Il reste donc encore un pan à explorer : celui de la parallélisation à gros grain sur plusieurs processeurs ou cœurs de calcul.

Transformer un algorithme séquentiel en algorithme parallèle n'est pas une tâche simple, mais nous pouvons nous appuyer sur des bibliothèques dédiées à cette tâche telle que OpenMP, qui facilite la création et la manipulation de multiples fils d'exécution. Un des points essentiels lors de la parallélisation d'un algorithme est de conserver sa correction, ce qui implique de garder une cohérence des données entres les fils d'exécution. Les matrices A et B n'étant jamais modifiées, seules les écritures dans C sont susceptibles d'être problématiques. Étant donné le grand nombre d'éléments de C à calculer, nous décidons que le calcul d'un élément donné ne se fera qu'au sein d'un seul fil d'exécution, chaque cœur travaillera ainsi sur des calculs totalement indépendants.

Créer un fil d'exécution pour chaque élément de C n'est cependant pas un bon choix si l'on souhaite obtenir de bonnes performances, car cette création est coûteuse et nous aurons beaucoup moins de cœurs disponibles que d'éléments à calculer. Idéalement, nous aurons le même nombre de fils d'exécution que de cœurs sur la machine, voir éventuellement deux fils par cœurs sur les architectures proposant de l'*hyper-threading*. Varier le nombre de fils d'exécution se fait simplement dans OpenMP à travers une variable d'environnement. La parallélisation de code avec OpenMP se fait

par l'utilisation de directives de compilation, qui sont ensuite interprétées par le compilateur pour générer le véritable code parallèle. Par exemple, répartir de manière équilibrée les itérations d'une boucle **`for`** se fait par l'ajout de la directive **`#pragma omp parallel for`** avant la déclaration de la boucle. Comme nous avons décidé de paralléliser les calculs sur les éléments de C, chaque itération sera indépendante et ce schéma de parallélisation est suffisant. Pour des raisons de performance, il vaut mieux paralléliser la boucle la plus externe, donnant ainsi à chaque fil un ensemble de lignes à traiter. Si nous parallélisons la boucle itérant sur les colonnes de C, un ensemble de fils d'exécution sera créé pour chaque ligne. La division des calculs serait similaire dans l'ensemble, mais ce choix de parallélisation entraînerait la création d'un nombre très conséquent de fils d'exécution, ce qui est coûteux.

La parallélisation de l'algorithme par blocs se fait de manière similaire : nous répartissons l'ensemble des multiplications de blocs sur divers fils d'exécution, chaque multiplication de deux blocs étant totalement calculée dans un seul fil donné, afin de conserver les avantages de la localité des données dans le cache et de ne pas risquer de conflit d'écriture. Dans un contexte de mémoire partagée il reste souhaitable de ne créer qu'une seule fois chaque bloc. Laisser cette création de blocs dans la boucle qui sera parallélisée permettra de répartir directement leur création parmi les cœurs, mais nous devons nous assurer que plusieurs cœurs ne tentent pas de créer simultanément le même bloc. OpenMP fournit également une directive permettant d'empêcher les accès concurrents à un section de code critique : en encadrant celle-ci entre les deux directives **`#pragma omp critical`** et **`#pragma omp end critical`**, nous avons l'assurance qu'un seul fil d'exécution pourra exécuter les instructions qu'elle contient à un instant donné.

Améliorer le passage à l'échelle Un effet secondaire de l'algorithme par blocs est qu'il réduit le nombre d'itérations de la boucle principale par la taille des blocs. Nous avons au départ divisé le travail entre les tâches parallèles en répartissant les lignes de blocs, cependant l'algorithme a une complexité cubique donc l'augmentation du nombre de lignes sera beaucoup plus lent que l'augmentation de la charge de calcul totale, et nous risquons d'avoir un mauvais équilibrage des calculs, voir même d'avoir plus de processeurs que de lignes disponibles. Par exemple la machine SPEED possède 48 cœurs, si nous considérons une matrice composée de 100 lignes de blocs (par exemple une matrice 3200×3200 divisée en blocs de taille 32×32), chaque processeur effectuera deux itérations complètes traitant au total 96 lignes, il restera 4 lignes à traiter dans la dernière itération, 44 cœurs n'auront donc aucun traitement à effectuer dans cette itération ce qui représente environ 30% de temps de calcul potentiel inutilisé.

Il est assez facile de corriger ceci en fusionnant les deux boucles principales : une boucle itère sur les colonnes à l'intérieur d'une boucle itérant sur les lignes, mais tous ces calculs sont indépendants et nous pouvons les regrouper au sein d'une unique boucle itérant sur l'ensemble des blocs par un simple jeu d'écriture sur les indices. La

parallélisation de cette boucle unique reste identique, mais son nombre d'itération est beaucoup plus grand et croîtra au même rythme que la taille de la matrice. En reprenant l'exemple précédent, nous avons maintenant une boucle d'environ 10000 blocs à répartir sur 48 cœurs, dans cette répartition nous aurons 208 itérations complètes et une itération où 32 cœurs seront inactifs, ce qui ne représente plus que 0.36% de temps de calcul inutilisé.

La figure 3.4 donne un aperçu du passage à l'échelle lors de la parallélisation par OpenMP sur les 48 cœurs de la machines SPEED des différentes versions de l'algorithme présentées plus haut. Ces mesures ont été effectuées dans le même cadre que les précédentes, mais uniquement sur des matrices de taille 2048 par 2048, les matrices de tailles inférieures n'étant pas suffisamment grandes pour profiter d'une parallélisation sur plusieurs dizaines de cœurs.

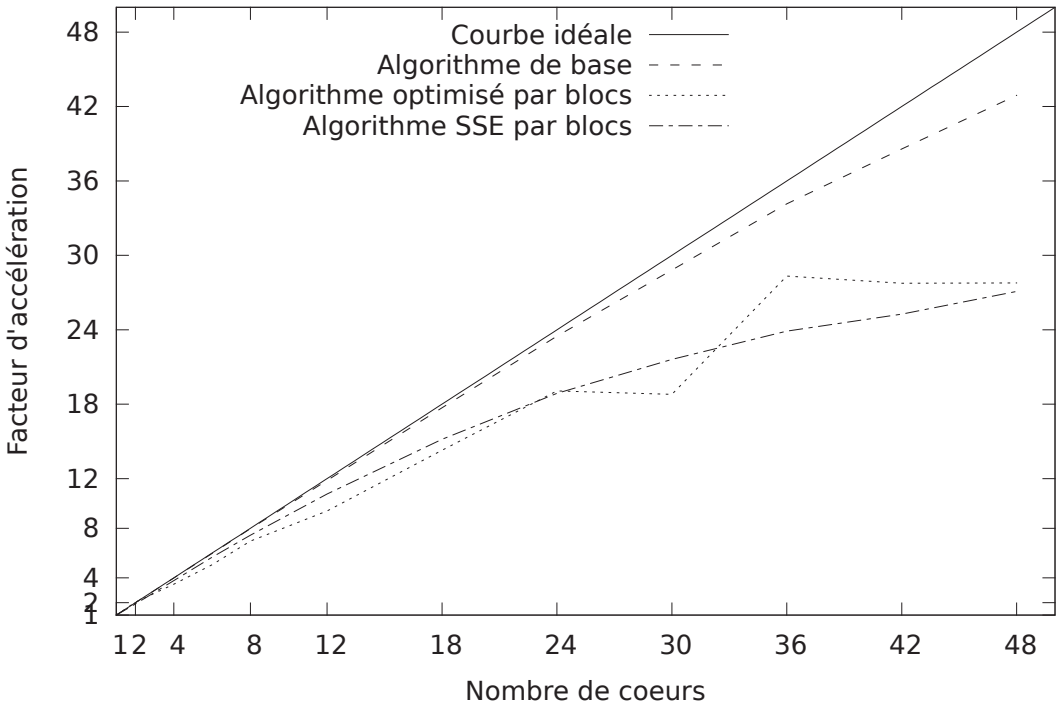


Figure 3.4 – *Gains de la parallélisation par OpenMP*

3.4 Analyse de l'effort de programmation

Nous souhaitons maintenant mettre en relief les gains obtenus par les diverses optimisations avec le travail de développement effectué pour mettre en place ces optimisations. Nous avons donc mesuré les propriétés suivantes des programmes : N_1

et N_2 les nombres totaux d'opérateurs et d'opérandes ainsi que η_1 et η_2 les nombres d'opérateurs et d'opérandes distincts. A partir de ces mesures nous avons calculé les métriques de Halstead (cf. chapitre 2.4.2.1) qui sont η le Vocabulaire, N la Longueur, V le Volume, D la Difficulté et E l'Effort. Ceci a été effectué pour chaque variante de notre algorithme, à savoir l'algorithme séquentiel naïf, la version avec réduction des accès en mémoire et la version vectorisée avec les instruction SSE, puis en les combinant avec l'approche par bloc et/ou la parallélisation OpenMP (Table 3.1).

L'algorithme de base est très simple et a un coût très faible, la première version optimisée est très peu différente et possède un coût proche. La vectorisation en SSE est quand à elle plusieurs dizaines de fois plus coûteuse, ce qui est cohérent avec le travail effectué pour son développement, voir même le sous-estime. En effet, ces métriques ont été conçues en ciblant les langages impératifs existant alors, tels que le fortran ou l'assembleur. Ces langages possèdent un vocabulaire limité que l'on peut supposer maîtrisé par le développeur. Nous pouvons de la même façon supposer les opérateurs standard du C++ connus par le développeur mais l'utilisation des instructions SSE requiert généralement un apprentissage et un travail de conception supplémentaires pour l'adaptation des algorithmes à un modèle vectorisé, ce qui augmente l'effort de développement sans que cela soit pour autant comptabilisé par les métriques de Halstead.

La parallélisation en OpenMP entraîne un coût minimal étant donné que nous avons une simple directive à ajouter avant la boucle externe. L'effort calculé devient légèrement inférieur pour l'algorithme SSE, ceci est dû à un biais dans la métrique : la valeur de Difficulté dépend du rapport $\frac{N_2}{\eta_2}$(c'est à dire le rapport entre le nombre total d'opérandes et leur nombre distinct), et ajouter trois opérandes ayant une unique occurrence aura un plus gros impact sur η_2 que sur N_2.

L'algorithme par blocs demande plus d'efforts ce qui était prévisible, cependant si nous considérons la partie divisant la matrice en blocs, celle-ci est totalement indépendante de l'algorithme de multiplication en lui-même, nous pouvons donc calculer l'effort nécessaire au développement de la partie par blocs qui s'additionnera ensuite à l'effort de l'algorithme de multiplication. Pris isolément, l'algorithme de découpage en blocs demande un effort d'environ 10 fois celui de l'algorithme de base, mais n'est qu'une fraction de l'effort demandé par l'algorithme SSE. La parallélisation OpenMP demande elle aussi plus de travail à cause des sections critiques à mettre en place ce qui représente environ 25% d'effort en plus.

Nous pouvons ensuite mettre en comparaison les gains de performance obtenus sur la machine SPEED (parallélisation OpenMP sur 48 cœurs) et l'effort de développement fourni (Figure 3.5).

La première optimisation de l'algorithme séquentiel est raisonnablement efficace car elle apporte un gain de l'ordre de 50% de performance pour un coût presque nul. L'algorithme par blocs est beaucoup moins efficace, car il demande une effort dix fois plus élevé tout en apportant seulement 50% de performance supplémentaire.

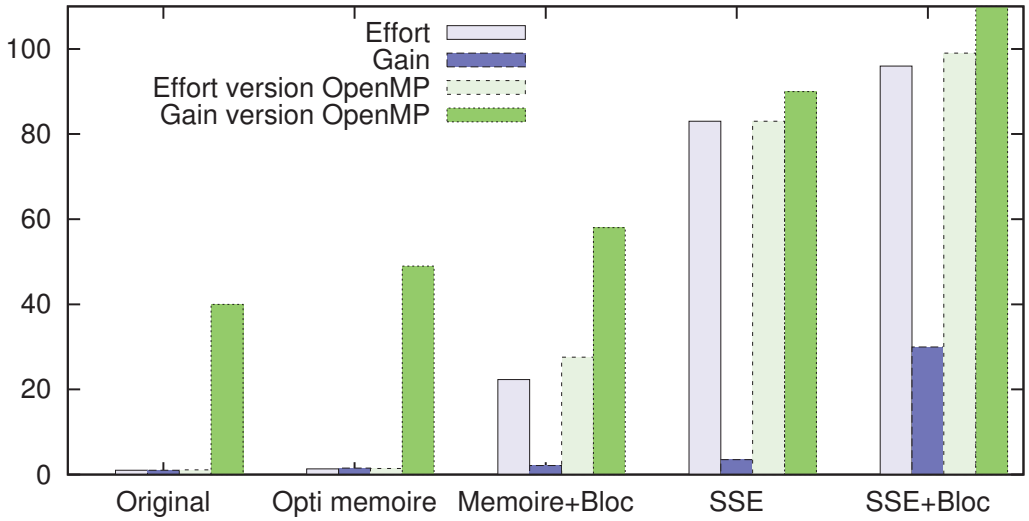


FIGURE 3.5 – *Comparaison performance - effort de développement*

Cependant il permet un meilleur passage à l'échelle lors de la parallélisation, et permet d'exploiter au maximum la vectorisation SSE, donc il devient très intéressant lorsqu'il est combiné avec les autres optimisations. Une comparaison plus intéressante est à faire entre l'algorithme SSE et la parallélisation OpenMP, qui fournissent tous deux des méthodes complémentaires pour paralléliser les calculs. L'algorithme SSE a un coût très conséquent, de l'ordre de 80 fois l'algorithme de base, mais amène un gain du même ordre, environ 30 fois plus rapide, si combiné à l'algorithme par blocs, il reste donc rentable à développer si l'on vise une efficacité maximale. Cependant, la parallélisation OpenMP a un surcoût très faible tout en fournissant un bon passage à l'échelle : selon l'algorithme parallélisé nous obtenons un facteur de gain de 28 à 42 sur les 48 cœurs de la machine SPEED. Cette optimisation est donc non seulement la moins coûteuse, mais également la plus efficace à condition de posséder un nombre conséquent de cœurs de calcul.

Au cours du développement, nous avons suivi de manière intuitive un cheminement en optimisant d'abord l'application au niveau le plus bas avec l'utilisation de registres puis les instructions SSE, avant d'aller vers des optimisations plus globales grâce à l'approche par blocs, puis la parallélisation à travers OpenMP. Avec le recul nos résultats montrent que cette approche n'était pas la plus efficace par rapport au temps de développement nécessaire. La parallélisation OpenMP aurait dû être prioritaire car elle fournit le meilleur gain de performance pour un coût de développement très faible. L'optimisation de base sur les accès mémoire aurait ensuite dû prendre place de par son gain correct pour un coût toujours faible, l'algorithme par blocs suivant et la vectorisation SSE arrivant en dernier à cause de son coût de développement très conséquent.

3.5 Conclusion

Nous avons mis en place une série d'optimisations sur un calcul classique de multiplication de matrices pleines, en prenant en compte les caractéristiques des processeurs multi-cœurs courants, à savoir l'organisation hiérarchique de la mémoire avec caches, les unités de calcul vectoriel et les multiples cœurs de calcul. Nous avons ensuite mesuré les gains de performance obtenus ainsi que l'effort de développement demandé par leur mise en place, et mis en comparaison ces mesures.

La tendance générale qui ressort de cette analyse est que les optimisations les plus proches du matériel dans un langage de bas niveau (tel que la vectorisation grâce aux fonctions intrinsèques SSE) offrent de très bons gains sur les performances brutes, et les meilleurs gains au niveau de l'efficacité (les performances atteintes par rapport au nombre d'unités de calcul utilisées), cependant celles-ci demandent une bonne maîtrise de l'architecture, demanderont à être adaptées pour être efficaces sur d'autres architectures et requièrent un temps de développement conséquent. A l'opposé, une approche de plus haut niveau telle que la parallélisation sur de multiples cœurs grâce à OpenMP est très simple à mettre en place, portable et fournit des performances du même ordre que les optimisations de bas niveau bien qu'étant nettement moins efficace, plusieurs dizaines de cœurs sont nécessaires pour obtenir les performances obtenues sur un seul cœur avec les autres optimisations. Au niveau de la productivité du développement, l'approche de haut niveau reste très largement supérieure même en ayant peu de cœurs de calcul disponibles.

Notre étude portant sur un algorithme et une architecture particulière, nous ne pouvons pas raisonnablement généraliser nos conclusions pour tous les algorithmes et toutes les architectures existantes. Nous pouvons néanmoins en extraire quelques lignes directrices : au niveau architectural, les futurs développements des processeurs s'orientent toujours sur le modèle actuel mais avec des unités de vectorisation plus larges et un nombre de cœurs toujours plus grands, et les accélérateurs visent eux aussi à intégrer toujours plus de cœurs. Du côté algorithmique, la multiplication de matrices pleines est un cas particulier se prêtant particulièrement bien à la parallélisation, au point d'être une des bases servant à évaluer les performances des ordinateurs (cf. Chapitre 2.4.1). Cependant, les gains principaux proviennent de la parallélisation du calcul, soit par vectorisation, soit par répartition du calcul sur plusieurs cœurs, or ces deux tâches présentent des difficultés similaires et il semble difficilement concevable que les gains obtenus grâce à la vectorisation d'un calcul ne puisse pas également être obtenus par une parallélisation à gros grain. On peut donc raisonnablement s'attendre à ce que les approches de haut niveau appliquées à d'autres types d'algorithmes fournissent toujours la meilleure productivité, tout en donnant des performances correctes.

Les approches de haut niveau nous semblent donc être une piste prometteuse car elles permettent aux utilisateurs d'accéder à une partie des performances atteignables

par la parallélisation du code sans requérir de grandes connaissances sur les particularités de l'architecture matérielle, tout en ayant un impact minimal sur leur productivité. Nous allons décrire dans les chapitres suivants nos travaux sur la bibliothèque OSL, qui permet la mise en place de calculs parallèles performants au sein d'un langage à squelettes qui est un modèle abstrait de très haut niveau.

Algorithme	η_1	η_2	η	N_1	N_2	N	V	D	E
Sequentiel*	16 (+2)	11 (+3)	27 (+5)	52 (+2)	33 (+3)	85(+5)	404 (450)	24,0 (23,1)	9k (10k)
Sequentiel optimisé	17 (+2)	14 (+3)	31 (+5)	60 (+2)	40 (+3)	100(+5)	495 (543)	24,3 (24,0)	12k (13k)
Vectorisé avec SSE	26 (+2)	24 (+3)	50 (+5)	408 (+2)	328 (+3)	736(+5)	4154 (4284)	177,7 (171,6)	738k (735k)
Algorithme par blocs									
Sequentiel	26 (+3)	27 (+5)	53 (+8)	224 (+9)	177 (+28)	401 (+37)	2297 (2598)	85,2 (92,9)	196k (241k)
Sequentiel optimisé	27 (+3)	30 (+5)	57 (+8)	232 (+9)	184 (+28)	416 (+37)	2426 (2728)	82,8 (90,9)	201k (248k)
Vectorisé avec SSE	36 (+3)	40 (+5)	76 (+8)	580 (+9)	572 (+28)	1152 (+37)	7198 (7600)	257,4 (260,0)	1852k (1976k)
Découpage en blocs	24 (+3)	24 (+5)	48 (+8)	172 (+7)	144 (+25)	316 (+32)	1765 (2021)	72,0 (78,7)	127k (159k)

* Entre parenthèses : coût supplémentaire des versions OpenMP

TABLE 3.1 – *Métriques de Halstead appliquées à la multiplication de matrices*

OSL

4

Sommaire

4.1 OSL pour les utilisateurs 60
4.1.1 Aperçu 60
4.1.2 Exemples 65
4.1.3 Sérialisation des données 67
4.2 OSL pour les développeurs 69
4.2.1 Principes généraux de métaprogrammation 69
4.2.2 Expression templates 76
4.2.3 Création de nouveaux squelettes 82
4.3 Conclusion 85

Orléans Skeleton Library (OSL) est une bibliothèque permettant la programmation parallèle à travers des squelettes algorithmiques. Celle-ci fût développée à l'origine dans les travaux de Noman Javed [66]. Un premier prototype avait été créé en utilisant *Bulk Synchronous Parallel ML* (*BSML*) [80], une bibliothèque pour le langage OCaml [32]. Ce prototype a permis de se concentrer sur la sémantique parallèle de haut niveau des squelettes sans se soucier des détails d'implantation, et notamment d'optimisation, du C++. Il a également été la base d'une première version d'autre prototype [25] écrit et prouvé correct avec l'assistant de preuve Coq [110].

Une première implémentation d'OSL en C++ a suivi. Le but de cette version était d'être plus accessible grâce à un langage plus répandu, et plus performante grâce aux mécanismes du langage qui sont plus efficaces. L'usage de C++ a permis la mise en place d'optimisations automatiques (fusion de boucles et élimination de variables temporaires) grâce à la technique des *expression templates* dont le fonctionnement est détaillé dans la section 4.2.2. Cette première version en C++ bien qu'utilisable souffrait de plusieurs défauts, dont notamment une structure difficilement extensible et une évaluation non optimale des squelettes nécessitant des communications. Une nouvelle version C++ remise à plat a commencé à être conçue. Nos travaux dans le cadre de cette thèse ont tout d'abord consisté en la finalisation de cette nouvelle implémentation

et la redéfinition de certains mécanismes existants. Nous avons ensuite enrichi OSL avec des mécanismes originaux par le biais de nouveaux squelettes.

4.1 OSL pour les utilisateurs

4.1.1 Aperçu

Sous sa forme actuelle, *Orléans Skeleton Library* (OSL) est donc une bibliothèque C++ permettant la programmation parallèle à travers des squelettes algorithmiques. Ces squelettes sont construits autour du modèle *Bulk Synchronous Parallelism* [113], et sont mis en place en utilisant la technique d'optimisation des *expression templates* [115] afin d'obtenir une grande efficacité des programmes générés. Les communications entre processus sont gérées grâce à la bibliothèque MPI.

Contrairement au modèle classique de programmation SPMD où un programme est vu comme un ensemble de processus différenciés par leur identifiant, OSL offre un vue globale du programme parallèle ce qui permet d'avoir un modèle de programmation où la parallélisation est implicite et très proche des modèles séquentiels classiques. Pour ce faire, les programmes OSL doivent manipuler une structure de données distribuée : les *tableaux distribués*.

Les tableaux distribués sont des tableaux unidimensionnels dont les données sont distribuées sur les différents processus du programme lors de leur création. Ceux-ci sont implémentés à travers une classe générique `DArray<T>`. Un tableau distribué est constitué de **`bsp_p`** partitions. Cette valeur **`bsp_p`** désigne le nombre d'éléments de calcul au sein de la machine parallèle BSP, et chacun d'entre eux se voit affecté un processus contenant une partition du tableau distribué. Chaque partition est en pratique un tableau classique d'éléments de type `T`. Cependant un utilisateur d'OSL a une vision globale du tableau distribué qu'il peut considérer comme un unique tableau, qu'on suppose généralement réparti de manière équilibrée entre les processus pour les programmes simples.

L'ensemble des squelettes disponibles dans OSL est présenté succinctement dans la figure 4.1 en donnant pour chacun d'entre eux sa signature et sa sémantique de manière informelle. Dans cette figure, le nombre de processus parallèles **`bsp_p`** est noté p. Comme indiqué précédemment, un tableau distribué de type `DArray<T>` peut être considéré séquentiellement comme un tableau $[t_0, \ldots, t_{t.size-1}]$ où $t.size$ est la taille (globale) du tableau (distribué) t, la même notation étant utilisée quand t est un vecteur C++ habituel.

Cependant, certains squelettes sont utilisés afin d'explicitement exposer ou modifier la distribution des tableaux distribués. Notre notation informelle doit donc pouvoir indiquer celle-ci. Nous notons une distribution sous la forme d'un indice D de son tableau distribué, D est une fonction de $\{0, \ldots, \mathbf{bsp_p}\}$ vers $\mathbb{N}$.

Les squelettes présents dans la version actuelle d'OSL recouvrent les fonctionnalités des spécifications des versions précédentes mais avec un jeu de squelette légèrement différent. Par exemple, **`gather`**, `broadcast` et **`balance`** disparaissent au profit de l'introduction de **`redistribute`** qui est leur généralisation. De plus nos travaux ont mené à la conception et l'implémentation de nouveaux squelettes pour OSL.

4.1.1.1 Squelettes conservant la distribution

Un tableau est dit équitablement distribué si :
- la plus grande partition contient au maximum un élément de plus que la plus petite partition,
- aucune partition ne possède d'identifiant de processus inférieur à celui d'une partition de taille supérieure.

Deux constructeurs créent uniquement des tableaux équitablement distribués :

1. Un premier prends une valeur `A a` et la taille globale du tableau, et réplique `a` pour chaque élément,
2. Un second prends une fonction `A f(`**`int`**`)` et la taille globale du tableau, et crée le tableau $[f(0), \ldots, f(t_{t.size-1})]$.

Un paramètre supplémentaire permet également de ne procéder à la construction de l'ensemble du tableau sur le premier processus uniquement, ce qui engendre un tableau non équitablement distribué car tous les autres processus ne contiennent alors aucun élément.

Deux autres constructeurs créent des tableaux distribués de manière quelconque. Ceux-ci prennent en paramètre un vecteur `std::vector<T> v` ou une fonction `std::vector<T> f()` et affectent à chacune des partitions le contenu du vecteur ou le résultat de l'évaluation de la fonction. Étant donnée la nature multi-processus du programme, il n'y a aucune garantie que le vecteur ou le résultat de la fonction soit le même dans tous les processus, on obtient donc une distribution imprévisible. Il est également possible pour ces constructeurs de demander la construction uniquement sur le premier processus, ce qui aboutit là encore à la création d'un tableau non équitablement distribué.

En utilisant les deux premiers constructeurs ainsi que les squelettes décrits dans la suite de cette sous-section, il est possible de ne générer que des tableaux équitablement distribués. L'utilisateur n'a pas à se préoccuper de la distribution des tableaux et peut raisonner sur son programme de la même manière qu'avec les tableaux séquentiels habituels.

Les quatre premiers squelettes de la figure 4.1 sont utilisés pour appliquer une fonction séparément à chaque élément d'un tableau distribué (**`map`**) ou d'une paire de tableaux distribués (**`zip`**). Le premier argument passé à ces deux squelettes **`map`** et **`zip`** peut être un foncteur C++ ou bien un objet fonction. Le squelette **`zip`** exige

Squelette	Signature Sémantique informelle
map	`DArray<W> map(W f(T), DArray<T> t)` $\mathbf{map}(f, [t_0,\ldots,t_{t.size-1}]_D) = [f(t_0),\ldots,f(t_{t.size-1})]_D$
zip	`DArray<W> zip(W f(T,U), DArray<T> t, DArray<U> u)` $\mathbf{zip}(f, [t_0,\ldots,t_{t.size-1}]_D, [u_0,\ldots,u_{t.size-1}]_D) = [f(t_0,u_0),\ldots,f(t_{t.size-1},u_{t.size-1})]_D$
mapIndex	`DArray<W> mapIndex(W f(int,T), DArray<T> t)` $\mathbf{mapIndex}(f, [t_0,\ldots,t_{t.size-1}]_D) = [f(0,t_0),\ldots,f(t.size-1,t_{t.size-1})]_D$
zipIndex	`DArray<W> zipIndex(W f(int,T,U), DArray<T> t, DArray<U> u)` $\mathbf{zipIndex}(f, [t_0,\ldots,t_{t.size-1}]_D, [u_0,\ldots,u_{t.size-1}]_D)$ $= [f(0,t_0,u_0),\ldots,f(t.size-1,t_{t.size-1},u_{t.size-1})]_D$
reduce	`<T> reduce(T ⊕(T,T), DArray<T> t)` $\mathbf{reduce}(\oplus, [t_0,\ldots,t_{t.size-1}]_D) = t_0 \oplus t_1 \oplus \ldots \oplus t_{t.size-1}$ où $\oplus$ est associative et $t.size > 0$
scan	`DArray<T> scan(T ⊕(T,T), DArray<T> t)` $\mathbf{scan}(\oplus, [t_0,\ldots,t_{t.size-1}]_D) = [\oplus_{i=0}^{0} t_i;\ldots;\oplus_{i=0}^{t.size-1} t_i]_D$ où $\oplus$ est associative
permute	`DArray<T> permute(int f(int), DArray<T> t)` $\mathbf{permute}(f, [t_0,\ldots,t_{t.size-1}]_D) = [t_{f^{-1}(0)},\ldots,t_{f^{-1}(t.size-1)}]_D$ où f est bijective
shift	`DArray<T> shift(int o, T f(T), DArray<T> t)` $\mathbf{shift}(o, f, [t_0,\ldots,t_{t.size-1}]_D) = [f(0),\ldots,f(o-1),t_0,\ldots,t_{t.size-1-o}]_D$
redistribute	`DArray<T> redistribute(std::vector<int> dist, DArray<T> t)` $\mathbf{redistribute}(\mathtt{dist}, [t_0,\ldots,t_{t.size-1}]_D) = [t_0,\ldots,t_{t.size-1}]_{\mathtt{dist}}$
getPartition	`DArray<std::vector<T> > getPartition(DArray<T> t)` $\mathbf{getPartition}([t_0,\ldots,t_{t.size-1}]_D)$ $= \left[[t_0,\ldots,t_{D(0)-1}],\ldots,[t_{j_i},\ldots,t_{j_i+D(i)-1}],\ldots,[t_{j_{p-1}},\ldots,t_{n-1}]\right]_{E_p}$ où $E_p(i) = 1$ et $j_i = \sum_{k=0}^{k=i-1} D(k)$
flatten	`DArray<T> flatten(DArray<std::vector<T> > t)` $\mathbf{flatten}([a_0,\ldots,a_{a.size-1}])$ $= \left[a_0[0],\ldots,a_0[a_0.size-1],a_1[0],\ldots,a_{a.size-1}[a_{a.size-1}.size-1]\right]_{D'}$ où $D'(i) = \sum_{D(i-1)\leq k<D(i)} a_k.size$ et $j_i = \sum_{k=0}^{k=i-1} D(k)$
forwardExceptions	`DArray<T> forwardExceptions(DArray<T> t)` $\mathtt{forwardException}(t) = t$ si aucune exception n'est levée lors de l'évaluation de t

FIGURE 4.1 – *OSL Skeletons*

que les deux tableaux distribués auquel il est appliqué aient la même distribution. Tant que ne sont utilisés que les deux premiers constructeurs cités précédemment et les squelettes décrits dans cette sous-section, cette égalité de distributions est présente pour peu que les tableaux distribués aient tous la même taille globale. Les variantes de **`map`** et **`zip`** préfixées par `Index` permettent d'appliquer des traitements dépendants de la position globale des éléments dans le tableau distribués. Dans ces variantes, les fonctions appliquées par les squelettes doivent prendre en argument supplémentaire un entier indiquant la position de l'élément à traiter. Aucun de ces quatre squelettes n'entraîne de communication entre les processus du programme parallèle.

`reduce` et **`scan`** appliquent un opérateur binaire *associatif* $\oplus$. **`reduce`** calcule la « somme » (au sens de l'opérateur $\oplus$) d'un tableau distribué non-vide. Le résultat de ce calcul est une valeur scalaire qui est présente dans tous les processus. **`scan`** calcule la « somme » des préfixes (au sens de $\oplus$) d'un tableau distribué dans le sens croissant de ses éléments. Si on l'applique à un tableau distribué vide, un même tableau distribué vide est retourné (il existe dans la littérature d'autres définitions de **`scan`** où ce squelette renvoie dans ce cas un tableau non-vide dont le premier élément est l'élément neutre de $\oplus$). Ces deux squelettes nécessitent aux processus de communiquer entre eux.

`permute` et **`shift`** modifient le contenu du tableau distribué passé en argument en modifiant les indices de ses éléments, mais la taille des partitions ainsi que la distribution du tableau distribué restent inchangés. **`permute`** prend en argument une fonction **`int`** `f(`**`int`**`)` qui doit être une bijection sur l'ensemble des indices du tableau distribué, et procède à l'échange des éléments selon cette bijection ; une autre sémantique est de passer en argument un tableau d'indices de taille identique au tableau distribué et dont chaque élément est l'indice de destination de l'élément correspondant dans le tableau distribué de départ. Seul l'ordre des éléments est modifié dans cette opération. **`shift`** est utilisé afin de décaler les éléments d'un tableau distribué vers la droite (respectivement la gauche) si la quantité de décalages passée en argument est positive (resp. négative). Ce décalage n'est pas circulaire : les emplacements laissés vides par le décalage sont remplis par l'application d'une fonction de remplacement passée comme second argument du squelette. Ces deux squelettes nécessitent des communications entre processus.

4.1.1.2 Squelettes manipulant la distribution

Trois squelettes permettent de modifier la distribution d'un tableau distribué : **`redistribute`**, **`getPartition`** et **`flatten`**.

redistribute procède à une redistribution des éléments, et entraîne le déplacement de données d'un processus à l'autre. Ce squelette prend en entrée une distribution sous la forme d'un vecteur d'entier positifs, et un tableau distribué. Ce tableau distribué voit

ses partitions modifiées par le squelette en fonction de la nouvelle distribution passée en paramètre. L'ordre relatif des éléments reste conservé lors de cette opération. Si l'on ne prend pas compte la distribution ce squelette est une fonction identité. Ce squelette est très général et peut être utilisé par exemple pour rassembler toutes les données sur un seul processus, ou à l'inverse répartir dans tous les processus les données contenues dans une partition donnée.

getPartition et son symétrique **flatten** modifient le point de vue sur les données, cependant aucun élément n'est déplacé entre les processus. Des communications peuvent néanmoins avoir lieu lors de l'évaluation du squelette **`flatten`**, mais uniquement afin de communiquer les informations sur la distribution entre processus.

Essentiellement, **`getPartition`** rend la partition de chaque processus visible dans le type de donnée du tableau et la rend accessible de manière unitaire. Par exemple, si nous avons un tableau distribué équitablement d dont le contenu est $[2;3;6;7;10;11]$ sur quatre processeurs, l'application de **`getPartition`** renvoie un tableau distribué de type `DA<std::vector<int> >` dont le contenu est $\big[[2;3];[6;7];[10];[11]\big]$. Après un appel à **`getPartition`**, chaque partition ne contient qu'un seul élément qui est donc un vecteur contenant tous les éléments de la partition dans sa forme précédente. La distribution, ainsi que le nombre d'éléments dans chacun des processus a changé, bien que la quantité de données contenue dans chaque processus n'ait pas changé en elle-même. **`flatten`** est l'opération symétrique : à partir d'éléments étant des vecteurs, un tableau distribué du type de base est reformé. Cette opération peut également être appliquée si plusieurs vecteurs sont présents sur un même processeur.

4.1.1.3 Autres squelettes

Une partie des travaux présentés dans cette thèse ont également consisté en l'ajout de squelettes offrant de nouvelles fonctionnalités. Le premier d'entre eux est utilisé afin de gérer de manière globale les exceptions qui peuvent survenir localement sur un processus donné. Ce squelette nommé `forwardException` prend en argument une expression de squelettes devant produire un tableau distribué, et retourne celui-ci si aucune exception ne s'est produite durant l'évaluation de l'expression. Si (au moins) un processus lève une exception, le squelette les retransmet à tous les autres processus. L'ensemble d'exceptions obtenu ainsi (encodé sous la forme d'un tableau d'exceptions) peut ensuite être rattrapé avec le bloc **`try`**/**`catch`** habituel de C++.

```
struct f {
   int operator()(int i) {
      if(i%2 == 0) throw std::exception();
      else return i;
   }
```

```
};
try
{
   forwardExceptions(map(f(), da));
}
catch ( std::vector<std::exception*> exn ) {
   /* Access to all the exceptions through exn */
}
```

Plus de détails concernant ce squelette, son implémentation et comment celui-ci apporte une plus grande expressivité à OSL sont discutés dans le chapitre 6.

Un dernier squelette permet l'application d'homomorphismes BSP (ou *BH*). Basés sur les homomorphismes de liste, ils permettent d'appliquer des algorithmes parallélisés de manière correcte sur des données distribuées à condition que ces calculs puissent s'exprimer sous une forme adaptée (composition d'une fonction et de deux homomorphismes de liste). L'intérêt principal est d'arriver plus facilement à prouver la correction d'un programme parallèle car il suffit de dériver de manière correcte la spécification originale de l'algorithme vers une forme sur laquelle *BH* peut être appliqué. Les détails sur la théorie des homomorphismes BSP et l'implémentation du squelette *BH* dans OSL sont discutés dans le chapitre 7.

4.1.2 Exemples

4.1.2.1 Produit scalaire

Un exemple basique de calcul facilement exprimable avec les squelettes d'OSL est le produit scalaire de deux vecteurs. Intuitivement, ce calcul peut être décomposé en deux étapes : premièrement multiplier chaque paire d'éléments de même position dans les deux vecteurs, puis additionner l'ensemble des résultats de ces multiplications. Ces deux actions correspondent à la sémantique du squelette **`zip`** appliquant une multiplication suivi du squelette **`reduce`** appliquant une addition.

La traduction en programme OSL découle logiquement : il nous faut définir les tableaux distribués représentant les vecteurs que l'on veut manipuler (par exemple en utilisant le constructeur utilisant un `std::vector`), puis simplement appeler les deux squelettes sur ces tableaux en utilisant des foncteurs effectuant les opérations de multiplication et d'addition. Ces foncteurs peuvent être écrits à la main, mais la bibliothèque standard fournit des objets génériques pour les opérations mathématiques de base, il est donc plus simple d'utiliser ceux-ci. Le programme OSL où `v1` et `v2` sont des `std::vector<`**`float`**`>` contenant les vecteurs à multiplier est donc :

```
DArray<float> da1(v1), da2(v2);
float resultat =
   osl::reduce(std::plus<float>(),
      osl::zip(std::multiplies<float>(), da1, da2);
```

4.1.2.2 Diffusion de la chaleur

Un exemple plus complexe est celui du calcul de la diffusion de la chaleur. Son but est de donner une approximation de la solution de l'équation différentielle partielle suivante qui décrit la diffusion de la chaleur au sein d'une barre de métal unidimensionnelle de longueur 1 :

$$\frac{\delta h}{\delta t} - \kappa \frac{\delta^2 h}{\delta^2 x} = 0 \qquad \forall t,\ h(0,t) = l \qquad \forall t,\ h(1,t) = r \tag{4.1}$$

κ est le facteur de diffusivité thermique caractéristique du métal, l et r sont les valeurs aux bornes (c'est à dire la température extérieure à chaque extrémité). Cette équation peut se discrétiser sous la forme suivante, où dt et dx sont les pas dans le temps et dans l'espace :

$$h(x,t+dt) = \frac{\kappa dt}{dx^2} \times \big(\ h(x+dx,t) + h(x-dx,t) - 2 \times h(x,t)\big) + h(x,t) \tag{4.2}$$

Nous implémentons ce calcul dans OSL en utilisant un tableau distribué pour représenter la température des différents points de discrétisation de la barre de métal (h), dx représente donc la largeur d'une case du tableau et vaudra donc 1, la finesse de la discrétisation étant déterminée par la longueur du tableau. Les pas de temps évoluent à travers les itérations d'une boucle, $h(x,t)$ représente donc l'élément à la position x du tableau h dans l'itération courante.

Nos squelettes permettent d'exprimer directement ce calcul sur l'ensemble du tableau, mais il faut le découper en éléments simples.

1. $h(x+1,t)$: cette partie de l'équation est un simple accès à la valeur de l'élément voisin dans le tableau. Ceci peut être effectué par le squelette **`Shift`**, en définissant la valeur de la condition au bord droit r comme élément à introduire en bout de tableau.

   ```
   auto right = shift(-1, r, h);
   ```

2. $h(x+1,t) + h(x-1,t)$: il faut ici effectuer l'addition de l'élément voisin obtenu ci-dessus avec l'élément voisin de l'autre côté. Cet élément s'obtient de manière similaire, et l'addition des deux valeurs correspond à l'action du squelette **`Zip`**

```
auto neighbours =
   zip(std::plus<double>(), left, right);
```

3. $-2 \times h(x,t)$: il suffit ici d'appliquer une multiplication par -2 à un unique élément, ceci s'effectue donc avec le squelette **Map** auquel nous passons le foncteur de multiplication de la bibliothèque standard C++ pour lequel nous fixons le premier argument à la valeur -2.

```
auto timesminus2 =
   map(bind(std::multiplies<double>(),-2,_1), h);
```

Le reste de la construction du calcul se fait selon les mêmes principes, et nous pouvons au final intégrer toutes les parties du calcul en une unique expression :

```
h =
   zip(std::plus<double>(),
      map(bind(
            std::multiplies<double>(),
            (delta_x * delta_x) / (diffuse * delta_t),
            _1
            ),
         zip(std::plus<double>(),
            zip(std::plus<double>(),
               shift(1, l, h),
               shift(-1, r, h)
               ),
             map(bind(std::multiplies<double>(), -2, _1), h)
            )
         ),
      h
   );
```

4.1.3 Sérialisation des données

Hormis pour **map**, **mapIndex**, **zip**, **zipIndex**, **getPartition** et **flatten**, l'évaluation de l'application des squelettes d'OSL peut nécessiter d'échanger des éléments des tableaux distribués entre les divers processus parallèles exécutés.

OSL est basée sur la bibliothèque MPI pour la mise en oeuvre des communications inter-processus, par le biais de la bibliothèque Boost.MPI du projet Boost. Boost [1] est

1. http://www.boost.org

un ensemble de bibliothèques expérimentales pour le langage C++ qui fournit des fonctionnalités très avancées allant souvent bien au-delà des utilisations habituelles du langage. À terme, les bibliothèques les plus populaires sont proposées à être intégrées dans la bibliothèque standard lorsqu'elles atteignent une maturité suffisante.

Boost.MPI permet d'abstraire les données manipulées par nos squelettes lorsqu'ils doivent communiquer entre eux, cependant il reste nécessaire d'avoir des moyens de sérialisation des données pour qu'elles puissent être utilisées dans les primitives de communications de MPI. À cet effet, nous pouvons utiliser les fonctionnalités d'une autre bibliothèque de Boost : Boost.Serialization. Cette bibliothèque fournit de nombreux moyens pour implémenter des opérateurs de sérialisation génériques.

Le concept de base consiste à définir comment un objet d'une classe donnée est sérialisé à travers une « archive » que l'on peut considérer comme étant un conteneur abstrait. La fonction de sérialisation est générique, et la classe d'archive est un paramètre *template* de cette fonction. Nous pouvons ensuite implémenter concrètement une classe d'archive ou bien utiliser celles existantes, qui fournissent typiquement des archives en mode texte pour les flux d'échange vers des fichiers ou des chaînes de caractères, des archives binaires ou encore XML. Le concept d'archive est utilisé de manière implicite dans Boost.MPI pour la mise en place des primitives de communication, ce qui nous permet de facilement échanger des objets entre processus MPI pour peu qu'ils disposent d'un opérateur de sérialisation sous cette forme.

Boost.Serialization fournit les opérateurs de sérialisation pour l'ensemble des types de bases du C++, ainsi que de nombreuses classes de la bibliothèque standard. Il est ainsi possible d'écrire d'entrée de jeu des codes tels que :

```
boost::mpi::communicator world;
std::vector<float> monvecteur(100, 3.14);
world.send(destination, tag, monvecteur);
```

Cependant, si l'on souhaite manipuler des données d'un type défini par l'utilisateur il est nécessaire de définir explicitement un opérateur de sérialisation pour ce type. Cet opérateur `serialize` peut être défini soit à l'intérieur même de la classe, soit en dehors. La première méthode est la plus simple si il est possible de modifier directement la classe pour intégrer cet opérateur, la deuxième est illustrée plus loin dans le chapitre 6.1.1. Cet opérateur décrit comment se passe la sérialisation (et implicitement la désérialisation) dans une archive (de type générique grâce à un opérateur de concaténation `&`). Il faut également déclarer la classe comme étant une classe amie de `boost::serialization::access`.

Par exemple, nous pouvons définir une classe `Paire` contenant un nombre entier et un nombre flottant et possédant un opérateur de sérialisation comme suit :

```
class Paire
{
      private :
      int _x;
      float _y;
      friend class boost::serialization::access;
      template<class Archive>
      void serialize(Archive& ar, const unsigned int version){
        ar & _x & _y;
      }
      public :
            Paire(int x = 0, float y = 0.0) : _x(x), _y(y){};
            Paire(const Paire &p) : _x(p._x), _y(p._y) {};
};
```

La taille de nos données étant constante, nous pouvons également signaler à Boost.MPI que ces objets peuvent être manipulés comme un type de base MPI, ce qui permet des appels plus efficaces aux primitives de communication. Ceci se fait avec la macro :

```
BOOST_IS_MPI_DATATYPE(Paire);
```

Les données de nos objets pouvant être décrites comme une simple concaténation d'objets de taille fixe, cet opérateur suffit à implémenter les deux mécanismes de sérialisation et désérialisation. Cependant il peut être nécessaire d'exprimer ces deux opérations de manière séparée, notamment lorsque l'on manipule des données de taille variable. Il est dans ce cas possible de définir au lieu de `serialize` deux opérateurs nommés `save_construct_data` et `load_construct_data` au sein desquels il est possible d'écrire (respectivement lire) des données dans l'archive grâce à l'opérateur de flux $\ll$ (resp. $\gg$). L'opérateur `load_construct_data` permet également d'appeler un constructeur autre que le constructeur par défaut lors de la désérialisation de l'objet.

4.2 OSL pour les développeurs

4.2.1 Principes généraux de métaprogrammation

Les *templates* ou « patrons de classes » sont un mécanisme du langage C++ permettant la mise en place de techniques de programmation générique dont le but est

de créer du code paramétrable très facilement réutilisable. Ceux-ci ont été introduits afin de pallier aux limites des macros. Ce style de programmation permet de définir des familles de classes ou de fonctions paramétrées par un ensemble de types ou de valeurs. Ce paramétrage permet d'abstraire certains points de l'implémentation qui sont spécifiés lors des utilisations spécialisées où ces types abstraits sont définis concrètement. Nous avons donc une approche assez similaire aux squelettes algorithmiques où il est possible d'exprimer séparément l'algorithme à effectuer et les détails d'implémentation. Cependant la programmation avec les *templates* est beaucoup plus générale.

4.2.1.1 Exemple de *template*

Une classe ou fonction *template* se déclare par le biais du mot-clé **`template`** suivi entre chevrons de la liste de ses paramètres. Les paramètres peuvent être de deux formes :

1. **`typename`** `identifiant` afin de définir un identifiant de type générique
2. **`int`** `N` pour définir un entier constant.

La classe ou fonction se décrit ensuite avec la syntaxe habituelle, mais il sera possible de manipuler au sein de celle-ci les identifiants de types génériques comme n'importe quel autre type défini du langage, et les entiers génériques comme n'importe quel entier constant défini. On peut par exemple mettre en place une classe générique de tableaux statiques de taille et de type quelconques :

```
template<typename T, int N>
class genarray
{
  private :
    T[N] mydata;
  public :
    genarray(){};
    ~genarray(){};
    genarray(const genarray<T, N>& src);
    ...
}
```

Une classe ou fonction *template* en tant que telle ne possède aucune existence dans le code compilé car le compilateur ne génère de code que pour des instances complètement spécialisées. Nous devons donc instancier notre classe générique afin de créer des objets manipulables dans notre programme. Dans notre exemple nous pouvons ainsi définir une instanciation de tableau contenant 5 nombres entiers, et une instanciation contenant 10 nombres flottants :

```
genarray<int,5> myarray;
genarray<double,10> myotherarray;
```

Le code correspondant à la classe `genarray` paramétrée par **`int`** et `5` ne sera créé que lors de sa première utilisation dans le code, ici à la déclaration de `myarray`. Le code généré correspondant à la même classe paramétrée par **`double`** et `10` sera donc un code différent du précédent. Bien qu'étant des spécialisation d'une même classe générique, les valeurs des paramètres *template* font partie intégrante du type de la classe spécialisée et ces deux objets possèdent donc des types totalement différents. Il est à noter que tous les éléments *template* définis à l'intérieur de la classe sont spécialisés. Ainsi le constructeur `genarray(const genarray<T, N>& src);` prend en paramètre un `genarray` similaire, on ne peut pas construire un `genarray<int,5>` à partir d'un `genarray<double,10>`.

Les codes des *templates* spécialisés sont générés puis compilés classiquement. Un *template* non-spécialisé n'aura donc aucune existence concrète dans le code final ce qui permet d'obtenir un code efficace réduit à l'essentiel, mais cela rend les codes difficiles à déboguer étant donné que seules les spécialisations sont compilées, et il peut être difficile de faire le lien entre le code généré par l'utilisation d'une spécialisation et le code générique.

4.2.1.2 Spécialisation de *templates*

Pour être réellement utilisée, une classe ou fonction *template* doit donc être instanciée en spécialisant totalement l'ensemble de ses paramètres. Cependant il est possible d'effectuer des spécialisations en restant dans le cadre générique. Les *templates* supportent la spécialisation partielle, on peut donc définir une classe *template* comme étant un cas particulier d'une autre plus générale. En reprenant notre exemple précédent, on peut définir une classe de tableaux de flottants de longueur quelconque en spécialisant `genarray` :

```
template<int N>
class genarray<float, N> {
  ...
  // operateurs surcharges specifiquement
  // pour les tableaux de floats
}
```

Il est possible de spécialiser une fonction ou classe *template* autant que l'on souhaite, éventuellement jusqu'à une spécialisation totale qui s'écrira sous la forme :

```
template<>
class genarray<float, 3> {
  ...
  // operateurs surcharges specifiquement
  // pour les tableaux de 3 floats
}
```

Lorsque le compilateur rencontre des instanciations de fonctions ou classes *template*, si plusieurs déclarations existent celui-ci utilise en priorité le code le plus spécialisé possible.

4.2.1.3 Métaprogrammation

Les *templates* ne permettent pas seulement de définir des fonctions génériques, mais permettent également la mise en place de constructions algorithmiques telles que l'arithmétique ou les structures de contrôles, qui seront donc résolues statiquement lors de la compilation [117]. Ceci permet de déléguer une partie des calculs au compilateur, et donc d'obtenir un programme plus efficace lors de son exécution car les résultats de ces calculs statiques sont directement disponibles. Cette utilisation a été mise en lumière par Erwin Unruh dans un programme effectuant, lors de la compilation, un calcul de nombres premiers comme effet de bord de l'instanciation d'objets *templates*, en affichant leur valeur sous la forme de messages d'erreurs de compilation. Des études plus poussées de ce type de mécanisme ont montré que le langage défini par les *templates* est en fait *Turing-complet* [116], et donc n'importe quel programme peut être réécrit entièrement sous la forme de fonctions ou classes *templates*.

4.2.1.4 Arithmétique

Les *templates* peuvent recevoir des paramètres de type entier contenant des valeurs explicites, ceci permet de mettre en place des calculs reproduisant le comportement des structures de boucles ou des récursions.

Un exemple classique est celui du calcul récursif de la factorielle : ce calcul est défini dans une structure, et la valeur à calculer est passée comme paramètre *template* de type entier. Le résultat du calcul est un membre de la structure, dont l'évaluation fait récursivement appel au résultat de l'évaluation de la factorielle de la valeur immédiatement inférieure :

```
template<int N>
struct fac
```

```
{
  static const int val = N*fac<N-1>::val;
};
```

De la même manière que l'on définit un cas terminal dans une récursion classique, nous devons définir une spécialisation de notre structure dont l'évaluation peut être effectuée directement :

```
template<>
struct fac<0>
{
  static const int val = 1;
};
```

Étant donné que le compilateur applique toujours le cas le plus spécialisé possible lorsqu'il a le choix entre plusieurs définitions génériques, la deuxième structure est utilisée lors de l'évaluation du cas d'arrêt `fac<0>`. Le champ `val` étant une valeur constante, il n'y a pas d'autres évaluations à effectuer pour l'obtenir, ce qui clôt les appels récursifs dès lors que l'on rencontre `fac<0>::val`.

L'évaluation d'une l'instanciation de `fac<N>` s'effectue donc à travers une série d'appels récursifs évaluant son type lors de la compilation :

```
int i = fac<4>::val;
int i = 4 * fac<3>::val;
int i = 4 * 3 * fac<2>::val;
int i = 4 * 3 * 2 * fac<1>::val;
int i = 4 * 3 * 2 * 1 * fac<0>::val;
int i = 4 * 3 * 2 * 1 * 1 = 24;
```

On peut ainsi définir de manière générale des récursions ou des boucles sur un nombre d'itérations définis, avec une structure *template* générale définissant le calcul, et une structure spécifique définissant une valeur d'arrêt. Une forte limitation est que cette arithmétique ne peut s'effectuer que sur les nombres entiers, et de par sa nature ne peut s'appliquer que sur des valeurs statiques (c'est à dire définies explicitement à la compilation).

4.2.1.5 Algorithmique générale

En dehors des entiers, il ne nous est pas possible de manipuler directement des valeurs à travers les paramètres *template* mais uniquement des types, c'est pourquoi ces derniers sont utilisés comme substituts pour représenter des valeurs en métaprogrammation. De plus, la manipulation de types se montre beaucoup plus rapide que la manipulation des entiers dans ce cadre. Afin de faire le lien entre des valeurs concrètes et les types manipulables par les *templates*, on retrouve très fréquemment des fonctions métaprogrammées nommées « adaptateur valeur/type ». Un adaptateur basique pour les booléens peut s'écrire sous la forme :

```
template <bool B>
struct bool_type
{
  static const bool value = B;
};
```

Nous pouvons ensuite déclarer deux types utilisant des spécialisations de cette structure avec les deux valeurs booléennes possibles :

```
typedef bool_type<true> true_type;
typedef bool_type<false> false_type;
```

Nous obtenons donc deux types représentant les valeurs booléennes classiques « *true* » et « *false* » que nous pouvons manipuler en tant que paramètres *templates*. Nous pouvons maintenant mettre en place diverses structures classiques du langage C sous forme métaprogrammée, tel qu'un opérateur d'égalité :

```
template<typename X, typename Y>
struct equals
{
  typedef false_type result_type;
};
template<typename T>
struct equals<T, T>
{
  typedef true_type result_type;
};
```

Cet opérateur utilise la spécialisation de *template* pour séparer le cas particulier où les deux types à comparer sont identiques et dont le résultat est le type correspondant à « *true* » défini précédemment, et le cas général où les types seront donc forcément différents et dont le résultat sera « *false* ». Cette utilisation des *templates* permet donc de mettre en place un mécanisme de filtrage par motif comme on peut trouver par exemple dans le langage OCaml.

De manière similaire, il est possible de définir facilement les opérations booléennes classiques, par exemple le « et » logique :

```
template<class A ,class B >
struct logical_and
{
  typedef false_type result_type;
};

template<>
struct logical_and < true_t , true_t >
{
  typedef true_type result_type;
};
```

Il est maintenant assez simple de mettre en place des branchements conditionnels, en définissant chacune des branches comme étant une fonction *template* spécialisée pour une valeur/type booléenne donnée :

```
template<>
struct mafonction<true_type>
{
  result = ....
};

template<>
struct mafonction<false_type>
{
  result = ....
};
```

`mafonction` peut donc être paramétrée par une valeur/type booléenne, soit directement, soit en évaluant le résultat d'une autre opération. Ainsi on pourrait écrire:

```
res = mafonction< equals<Type1, Type2>::result_type >::result
```

et le calcul de résultat se fera de manière différente selon que `Type1` et `Type2` soient identiques ou non.

Nous avons donc pu mettre en place de manière succincte les éléments essentiels au développement de programmes en utilisant la métaprogrammation par *templates* : la définition de valeurs, des opérateurs de comparaison sur celles-ci, des combinateurs logiques et des structure de boucles et de choix conditionnels.

4.2.2 Expression templates

La technique des *expression templates* est une méthode de métaprogrammation introduite par Todd Veldhuizein [115] utilisant les mécanismes d'évaluation partielle des *templates* afin d'effectuer une analyse syntaxique d'expressions définies par l'utilisateur, et leur optimisation par la génération automatique de code efficace pour leur évaluation dynamique. On retrouve ce mécanisme notamment dans la mise en place de langages enfouis dédiés.

L'idée générale est de représenter les opérateurs composant les expressions que l'on veut pouvoir manipuler par des classes *templates*, dont les opérandes sont les paramètres *template*. Dans une expression combinant plusieurs opérateurs, l'opérateur principal a comme opérande un opérateur secondaire et ainsi de suite. L'expression est construite en tant qu'un unique objet au type complexe composé des différents types des opérateurs et opérandes utilisés. La construction de cet objet au type complexe permet de mettre en place des optimisations, notamment au niveau de l'opérateur d'évaluation qui peut être unifié ce qui permet d'évaluer le résultat de l'évaluation de l'expression en un seul appel, au lieu d'effectuer des appels successifs séparément pour chacun des opérateurs. De plus, cela permet de mettre en place implicitement un mécanisme d'évaluation paresseuse : lorsque l'expression est rencontrée dans le code, seul l'objet lui correspondant est créé. Ce n'est que lorsque l'on rencontre une opération nécessitant d'accéder explicitement aux valeurs résultant de l'évaluation de cette expression (typiquement, lors d'une assignation du résultat à une nouvelle variable) que l'opérateur d'évaluation est utilisé et donc le calcul réellement effectué.

Un exemple classique est la définition d'opérations sur les tableaux. Si nous définissons une classe Array représentant des tableaux, un opérateur effectuant l'addition de deux tableaux élément par élément peut être implementé sous cette forme :

```
Array operator+(Array a, Array b){
  Array result(a.size());
  for(int i = 0; i < result.size(); i++)
```

```
    result[i] = a[i] + b[i];
  return result;
}
```

Cet opérateur nous permet d'écrire des opérations utilisant plusieurs tableaux telles que `Array res = a + b + c + d;`. Le problème de notre opérateur d'addition dans cette utilisation est son inefficacité sur l'ensemble du calcul : celui-ci est en fait décomposé sous la forme (a + (b + (c + d))), où chaque paire de parenthèse est une application de l'opérateur `+`, et entraîne donc la création d'un tableau temporaire sur lequel une boucle itère. Un calcul efficace consisterait en une unique boucle effectuant le calcul `res[i] = a[i] + b[i] + c[i] + d[i];` pour chacun des éléments concernés.

En suivant les principes des *expression templates*, nous pouvons définir un ensemble d'objets et de méthodes permettant de créer automatiquement ce type d'opérateur complexe sans alourdir la syntaxe manipulée par l'utilisateur. Nous devons d'abord définir une classe générique pour les expressions :

```
template <class A>
struct Expr {
  const A& operator () const {
    return *static_cast<const A*>(this);
  }
};
```

Cette classe de base a pour rôle de fournir une interface commune pour tous les éléments que nous voulons intégrer dans les expressions. L'élément de base que nous souhaitons manipuler ici est une structure de tableau, nous allons donc définir une classe générique `ExpArray<T>` les implémentant qui hérite de la classe de base `Expr`. Cette dernière nécessite un paramètre *template*, or nous voulons qu'un tableau soit équivalent à une expression ne contenant qu'un tableau, c'est donc la classe `ExpArray <T>` elle-même qui est le paramètre de `Expr` ici. Ce schéma où une classe dérivée hérite d'une classe de base *template* paramétrée par la classe dérivée elle-même est un *design pattern* nommé *Curiously Recursive Template Pattern* [38].

```
template<class T>
class ExpArray : public Expr<ExpArray<T> > {
  public:
  template <class A>
  void operator = (const Expr<A>& a_) {
    const A& a(a_);
```

```
    for (int i = 0; i < n; ++i)
    data[i] = a[i];
  }
  T operator[] (int i) const {
    return data[i];
  }
};
```

Cette classe se construit de façon similaire à une classe générique de tableaux habituelle, la différence principale se trouvant dans l'opérateur d'affectation. On ne veut pas seulement être capable d'affecter le contenu d'un tableau de même type, mais également le résultat de n'importe quelle expression dont l'évaluation produit un tableau. L'opérateur va donc en premier lieu appeler l'opérateur `()` de la classe `Expr`, qui renvoie l'expression convertie en son type dérivé. Ensuite, une boucle classique effectue l'affectation de chacune des valeurs grâce à l'opérateur `[]` appliqué à l'expression.

Dans le cas d'une expression ne contenant qu'un tableau, l'opérateur `[]` renvoie simplement l'élément de donnée concerné. Si nous voulons ajouter des opérateurs permettant de composer des expressions plus complexes, nous les exprimons de manière similaire à notre classe de tableaux : ce sont des classes dérivées de `Expr` dont les paramètres *templates* définissent la sémantique d'utilisation au sein des expressions, et dont l'opérateur `[]` définit l'évaluation pour un élément donné.

Ainsi, nous pouvons définir l'addition de deux expression comme suit :

```
template <class A, class B>
  class Add : public Expr<Add<A,B> > {
    const A& a_;
    const B& b_;
  public:
  Add(const A& a, const B& b) : a_(a),b_(b){}
  double operator[](int i) const {
    return a_[i] + b_[i];
  }
};
```

L'opérateur `[]` définit ici comment appliquer l'opération d'addition effectuée par cette structure sur deux sous-expressions `A` et `B`. Il est à noter que les classes `A` et `B` sont totalement génériques, on peut donc appliquer l'addition à n'importe quelle structure de données du moment que celle-ci possède un constructeur par copie et un opérateur `[]` dont les éléments renvoyés possèdent un opérateur `+`. On peut également noter que le constructeur de `Add` effectue une copie des références des sous-expressions `A`

et `B`. Ceci est nécessaire car leur évaluation dans l'opérateur `[]` n'a pas lieu au même moment que la création de l'objet et il faut donc avoir un moyen d'accéder à `A` et `B` lors de l'évaluation.

Nous pouvons donc maintenant combiner des objets `ExpArray` et `Add` pour construire nos expressions, l'instanciation des divers *templates* composant une expression aboutit à la création d'un unique objet au type complexe et possédant un unique opérateur d'évaluation `[]`. L'évaluation et l'affectation du résultat d'une expression dans un tableau s'effectue en une unique boucle, quelque soit la longueur et la complexité de l'expression, et aucun résultat temporaire n'a à être créé.

La classe `Add` reste cependant assez peu pratique à manipuler sous cette forme, pour une utilisation plus proche des opérations habituelles sur les types simples nous pouvons définir un opérateur `+` fournissant une interface plus simple :

```
template <class A, class B>
inline Add<A,B>
operator+(const Expr<A>& a, const Expr<B>& b){
  return Add<A,B>(a,b);
}
```

Il est maintenant possible de manipuler des expressions utilisant des tableaux et des additions de manière très naturelle :

```
// a, b et c sont des tableaux de nombres entiers de meme
   longueur
ExpArray<int> res = a + b + c + d;
```

Cet exemple simple illustre l'essentiel du mécanisme des *expression templates* ; leur utilisation dans un cas plus général suit le même schéma : à partir d'une grammaire définissant les expressions que l'on souhaite pouvoir évaluer, il s'agit de définir les différents éléments composant cette grammaire, c'est-à-dire la description de son arbre de syntaxe abstraite composé de noeuds assemblant plusieurs sous-arbres et de feuilles contenant les valeurs à utiliser.

Dans notre exemple précédent, nous pouvons comparer la pile d'appels utilisant l'opérateur fonctionnel simple, et l'arbre de syntaxe générant l'expression composée de *templates*. Dans la pile d'appels (figure 4.2), l'évaluation du calcul demandera 4 boucles et produira 3 tableaux temporaires. Dans l'approche par *expression templates* (figure 4.3), un type d'objet complexe est construit progressivement lors de la compilation, l'évaluation du calcul se fera donc par l'opérateur = d'un objet de type `Expr <Add<ExpArray, Add<ExpArray, Add<ExpArray, ExpArray> > > >`, opérateur qui n'effectue qu'une seule boucle sans créer de tableaux temporaires.

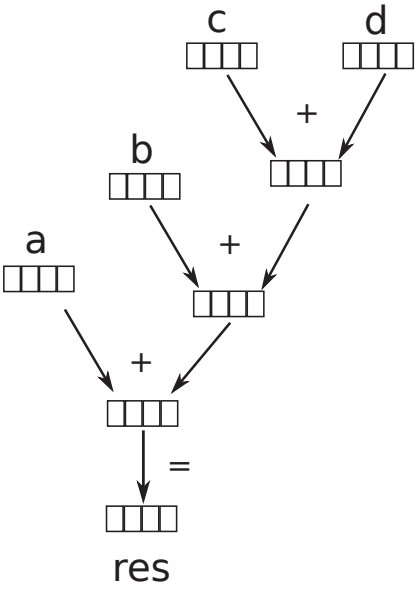


FIGURE 4.2 – *Pile d'appels*

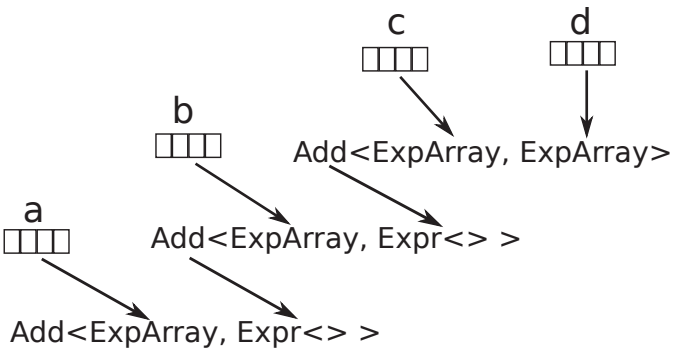


FIGURE 4.3 – *Construction d'une expression*

4.2.2.1 Expression *templates* dans OSL

La version actuelle d'OSL diffère avec la précédente notamment au niveau de l'intégration du mécanisme des *expression templates*. La structure de la bibliothèque a été remaniée afin d'être plus facilement compréhensible et extensible notamment par l'élimination de certaines structures intermédiaires cachées. Une autre différence réside dans l'évaluation des squelettes nécessitant des communications : cette évaluation est maintenant contrôlable grâce notamment à l'introduction de l'opérateur `Evaluate`.

Le modèle de programmation par squelettes mis en place dans OSL nous permet de mettre en place un mécanisme d'*expression templates* sur les tableaux distribués, et la syntaxe fonctionnelle permet une mise en place assez simple. Les tableaux distribués suivent le même schéma que la classe décrite précédemment : l'opérateur `[]` accède aux éléments locaux du tableau, l'affectation d'un autre tableau distribué se fait par déplacement ou copie des données locales, et l'affectation d'une expression déclenche l'évaluation de celle-ci à travers son opérateur `[]` pour chacun des éléments locaux au processus courant.

Les expressions en tant que telles sont utilisées sous la forme d'un type totalement générique, et décrivent généralement tout ce qui n'est pas un tableau distribué. Ce relâchement dans la structure découle en fait de la définition de nos squelettes qui possèdent une sémantique à la fois très forte et variée, et leur utilisation suivant un mode fonctionnel donne implicitement une forte structuration dans les expressions les composant.

Les squelettes sont accessibles à l'utilisateur sous la forme de fonctions *template*, par exemple l'application du squelette **`Map`** se fait par :

```
template<class F, class Exp>
inline Map<F, Exp> map(F f, const Exp& exp){
  return Map<F, Exp>(f, exp);
};
```

L'appel au squelette se fera donc en lui passant un objet `f` qui est la fonction à appliquer à l'expression constituant le deuxième paramètre. L'appel à **map**() ne déclenche pas l'évaluation de l'application de la fonction, mais crée un objet de type **Map**<F, Exp>. Ainsi une suite d'appels imbriqués telle que **map**(f1,**map**(f2,tab)) (où f1 serait de type F1, f2 de type F2 et tab un tableau distribué d'entiers) déclenche la création d'un unique objet de type complexe **Map**<F1, **Map**<F2, DArray<**int**> >.

Au sein de la classe générique **Map**<F, Exp>, le constructeur effectue une simple copie des références vers les différents paramètres, et l'opérateur [i] effectue l'évaluation de f appliqué à l'élément i de l'expression. Dans le cas de notre objet au type complexe **Map**<F1, **Map**<F2, DArray<**int**> >, l'opérateur [i] effectue donc le calcul f1(f2(tab[i])). L'affectation à un tableau distribué de cet objet effectue donc l'évaluation de cet opérateur indépendamment pour chacune des valeurs du tableau tab en une seule boucle et sans créer de structure temporaire.

Les autres squelettes de calcul sont construits de manière similaire, ce qui permet ainsi d'avoir un unique opérateur d'évaluation par élément quelle que soit la complexité de l'expression construite. Les squelettes mettant en place des communications suivent un mécanisme différent : pour des raisons de simplicité et d'efficacité, il est plus intéressant d'évaluer ces squelettes sur l'ensemble des données plutôt que de considérer leur application élément par élément. Un exemple caractéristique est la permutation de deux valeurs : il est beaucoup plus efficace d'effectuer directement la permutation de manière globale (chaque processus peut rapidement déterminer si il contient les valeurs concernées) que d'effectuer ceci localement à chaque élément. De plus cette approche globale est fortement synchronisée ce qui élimine les risques de problèmes de communication, et permet de conserver des performances prévisibles par rapport au modèle BSP.

Une autre raison nous poussant à évaluer ces squelettes de manière globale est le respect de la sémantique des expressions à squelettes : permettre une fusion de boucle à travers ces squelettes reviendrait à permettre le déplacement des communications au sein des opérations à évaluer pour résoudre l'expression. Or, même si ceci n'entraînerait pas d'erreurs dans l'évaluation du résultat, ceci pourrait néanmoins entraîner un comportement non souhaité de l'application : échanger des données puis appliquer un calcul produit le même résultat que d'appliquer le calcul en premier puis échanger les données ensuite, cependant l'utilisateur peut s'attendre à ce que ces opérations soient effectuées dans l'ordre qu'il a défini, ce qui peut aussi lui permettre d'optimiser son programme en choisissant un ordre efficace dans ses opérations.

Les squelettes s'évaluant dans leur globalité ne possédent donc pas d'opérateur d'accès par élément `[]`, mais un opérateur d'évaluation globale `()`. Ce dernier s'utilise sans difficulté particulière pour une application sur les tableaux distribués, cependant l'application à une expression peut poser problème : si le squelette manipule les éléments de données de l'expression à travers l'opérateur `[]` l'expression est traitée comme s'il s'agissait d'un tableau, mais on peut souhaiter les manipuler différemment notamment lorsque l'on modifie la structure des tableaux distribués. Dans ce type de situation, nous forçons explicitement l'évaluation de la sous-expression afin de manipuler directement le tableau distribué résultant. Ceci s'effectue par le biais d'un squelette intermédiaire nommé `Evaluate`. Ce squelette possède une implémentation spécifique à chacun des autres squelettes existant et fournit un opérateur `()` retournant le résultat de l'évaluation complète de l'expression qui lui est donnée.

4.2.3 Création de nouveaux squelettes

Une des caractéristiques souhaitées d'OSL ayant influencé le développement de cette bibliothèque est l'extensibilité, ainsi il est possible d'apporter assez facilement de nouvelles fonctionnalités à la bibliothèque en créant de nouveaux squelettes.

La première question à se poser lorsque l'on souhaite développer un nouveau squelette est celle de sa nécessité : est-ce que le nouveau squelette permettrait d'exprimer des comportements algorithmiques impossibles ou très difficiles à mettre en place avec le jeu de squelettes actuel. Les squelettes existant dans OSL couvrent une grande variété de comportement algorithmiques, il est donc probable que l'expression d'un algorithme donné puisse se faire grâce à une combinaison de squelettes avec un paramétrage particulier, auquel cas il est plus simple de définir une fonction combinant les squelettes que de créer un squelette complet.

Un tel exemple pourrait être la création d'un mécanisme de *gather* (rassemblement des données sur un seul processus). Le squelette **`redistribute`** permet de répartir les données d'un tableau distribué de façons totalement libre, le mécanisme de rassemblement est donc un cas particulier de **`redistribute`**. Nous pouvons donc implémenter assez facilement une fonction **`gather`** sur un tableau distribué :

```
template<typename T>
inline DArray<T> gather(DArray<T>& input)
{
   std::vector<int>
      newDistribution(input.getDistribution.size(), 0);
   newDistribution[0] = input.length();
   return Redistribute<DArray<T> >(input, newDistribution)();
}
```

Néanmoins cette fonction ne peut s'appliquer que sur un tableau distribué, et pas une expression à squelettes. Pour une intégration complète, il nous faut donc prévoir ce cas. Celui-ci est plus délicat à mettre en place, étant donné que le squelette **redistribute** doit déclencher explicitement l'évaluation de l'expression auquel il est appliqué afin d'éviter la fusion de boucles. Un code appliquant un traitement similaire est déjà présent dans l'appel de fonction servant d'interface pour les utilisateurs du squelette, ce code peut donc être facilement adapté pour notre cas particulier :

```
template<typename Exp>
inline DArray<typename Exp::result_type> gather(const Exp& exp)
{
   std::vector<int>
      newDistribution(exp.getDistribution.size(), 0);
   newDistribution[0] = exp.length();
   return Redistribute<DArray<typename Exp::result_type> >(
      Evaluate<Exp>()(exp), newDistribution)();
}
```

Si l'utilisation des squelettes existants est trop complexe ou limitante pour implémenter un mécanisme algorithmique, on peut donc envisager la création de nouveaux squelettes. Au niveau de l'implémentation, il est possible de classer les squelettes en deux familles au fonctionnement différent : les squelettes pouvant s'évaluer indépendamment pour chaque élément, et ceux qui s'évaluent globalement sur un tableau distribué entier. La première catégorie englobe essentiellement les squelettes purement calculatoires tels que **map**, tandis les squelettes nécessitant la mise en place de communications se retrouvent essentiellement dans la deuxième catégorie, par exemple **reduce** ou **redistribute**.

Un squelette s'appliquant indépendamment sur chaque élément peut s'appliquer indifféremment sur un tableau distribué ou une expression en produisant un. La classe le représentant possède donc en attribut une référence vers l'expression qui lui est donnée à la construction. Ceci donne pour la classe **map** :

```
template <typename F, typename Exp>
class Map
{
private:
  F f;
  const Exp& exp;
public:
  template<typename T>
  Map (F fun, DArray<T>&& e) : f(fun), exp(e) {}
....
```

L'opérateur principal à définir sera l'opérateur `[]` qui définit le calcul a effectuer pour un élément donné de l'expression, qui est également obtenu à travers un même opérateur `[]` :

```
typename F::result_type inline operator[] (int index) const
{
   return f(exp[index]);
}
```

L'ensemble des accesseurs disponibles sur les tableaux distribués (`length()`, `getLocalsize()`, `getDistribution()`, *etc.*) doivent être également mis en place.

L'utilisateur n'est pas censé manipuler directement les objets représentant les squelettes, il faut donc également définir un appel sous forme fonctionnelle dont le rôle consiste simplement à créer et renvoyer l'objet du bon type :

```
template<class F, class Exp>
inline Map<F, Exp> map(F f, const Exp& exp)
{
   return Map<F, Exp>(f, exp);
}
```

Le cas des squelettes s'appliquant sur l'ensemble d'un tableau distribué est légèrement différent. Ceux-ci ne possèdent pas d'opérateur `[]` car il n'est pas possible d'évaluer isolément la valeur d'un élément. À la place il faut définir un opérateur `()` global. L'absence d'opérateur `[]` les empêche d'être intégrés dans le mécanisme de fusion, l'opérateur `()` doit donc renvoyer un tableau distribué d'un type précis (par exemple `Exp::result_type` si le squelette ne modifie pas le type des données de l'expression en entrée).

Le squelette peut s'appliquer de manière transparente directement sur une expression, ce qui est par exemple le cas de **`reduce`**. Cependant on peut vouloir déclencher explicitement l'évaluation de l'expression passée au squelette avant d'effectuer ses calculs, ce qui est habituellement le cas dans les squelettes déplaçant des éléments entre les processus : on souhaite appliquer les squelettes contenus dans l'expression avant les échanges pour rester cohérent avec le programme que l'utilisateur a écrit. Comme indiqué précédemment avec l'exemple de la fonction **`gather`**, nous pouvons faire ceci grâce à la fonction générique `Evaluate` qui provoque explicitement l'évaluation d'une expression et renvoie le tableau distribué résultant. Cet appel se déroule de préférence dans l'appel fonctionnel du squelette, ainsi l'opérateur `()` peut être programmé en supposant qu'il s'applique uniquement à un tableau distribué.

Quelle que soit la catégorie du nouveau squelette créé, il faut donc que l'opérateur `Evaluate` soit défini sur celui-ci pour une intégration complète au sein d'OSL. Les squelettes de la seconde catégorie n'ont pas ce besoin car ceux-ci renvoie toujours

un tableau distribué, et l'opérateur `Evaluate` est déjà défini pour ces structures. Il reste néanmoins à définir cet opérateur pour les squelettes renvoyant un objet de leur type, celui-ci s'écrit assez facilement : il suffit de créer un tableau distribué résultat, de définir ses attributs, et d'effectuer explicitement l'affectation à chacun de ses éléments de l'élément correspondant dans le squelette par le biais de son opérateur `[]`.

4.3 Conclusion

Nous avons dans ce chapitre décrit OSL, une bibliothèque de squelettes algorithmiques de parallélisme de données en C++ basée sur MPI pour la gestion des communications interprocessus. Nous avons décrit les fonctionnalités principales d'OSL, et illustré comment ces fonctionnalités peuvent être exploitées pour mettre en oeuvre des programmes parallèles. Nous avons ensuite décrit les mécanismes généraux permettant la mise en place efficace de ces fonctionnalités, notamment la technique de métaprogrammation des *expression templates* qui permet de générer à la compilation des codes efficaces en réduisant le nombre de boucles de calcul et de variables temporaires créées lors de l'évaluation de compositions de squelettes algorithmiques.

Les squelettes d'OSL introduits dans ce chapitre permettent d'exprimer des algorithmes variés, cependant il reste encore de nombreuses limitations, et certains mécanismes de programmation séquentielle ne sont pas exprimables sous une forme parallèle à travers les squelettes existant. Dans le chapitre 5 nous étudions les squelettes manipulant la distribution des données dans notre environnement parallèle et les travaux que nous avons effectué sur ceux-ci pour améliorer et étendre leur fonctionnement. Nous avons ensuite mis en place de nouveaux mécanismes de programmation parallèle au sein d'OSL. Dans le chapitre 6, nous introduisons la gestion des exceptions de manière globale au sein de notre environnement d'exécution distribuée. Finalement, dans le chapitre 7 nous étudions un modèle de programmation de très haut niveau permettant le développement de programmes parallèles corrects.

Manipulations des distributions

5

Sommaire

5.1 Problématique 88
5.2 Squelettes agissant sur la distribution 88
5.2.1 Redistribute 89
5.2.2 GetPartition 90
5.2.3 Flatten 90
5.2.4 Implémentation du filtrage 91
5.3 Tri par échantillonnage régulier en parallèle 91
5.3.1 Implémentation 93
5.3.2 Coût BSP 93
5.3.3 Expérimentations 94
5.4 Conclusion 95

Une version préliminaire de ce chapitre a été publiée dans les actes de la conférence HPCS 2013 [75].

OSL rend transparent le mécanisme de distribution des données, ce qui permet à l'utilisateur de manipuler les tableaux distribués sans ce soucier de cette caractéristique avec la plupart des squelettes. Ceci permet de facilement mettre en place une large variété d'algorithmes. Cependant cela impose également d'importantes limitations à l'expressivité de la bibliothèque car la structure des tableaux ne peux pas être manipulée. Ceci entraîne entre autres l'impossibilité de modifier le nombre d'éléments de données contenus dans un tableau, un exemple classique de ce besoin étant le filtrage des données : on souhaite évaluer un prédicat sur chaque élément d'un tableau, et ne conserver que les éléments dont l'évaluation est positive (Figure 5.1).

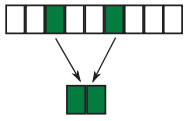


Figure 5.1 – *Filtrage de données*

5.1 Problématique

Ce comportement pourtant très simple est impossible à mettre en place si l'utilisateur ne peut modifier la structure des tableaux distribués. Permettre à l'utilisateur de directement modifier la taille des tableaux distribués n'est pas une solution adéquate car cela irait à l'encontre des principes de notre modèle de programmation visant à exprimer des comportements de haut niveau les plus abstraits possible. De plus, la nature distribuée des données fait qu'une opération modifiant le nombre d'éléments dans un tableau risque fort de ne pas modifier de manière identique chaque sous-tableau local à un processus, ce qui obligerait donc l'utilisateur à mettre en place un mécanisme afin de conserver la cohérence globale du tableau distribué. Or les mécanismes que nous voulons fournir à l'utilisateur dans OSL doivent garantir la production de résultats cohérents.

La solution envisagée pour résoudre cette problématique dans OSL est illustrée dans la figure 5.2. Une première opération permet de créer une vue partitionnée du tableau distribué. Dans cette vue, les éléments sont regroupés par localité et l'utilisateur peut manipuler ainsi les sous-tableaux contenant les éléments locaux à chaque processus. Dans le cas du filtrage, la modification peut être effectuée en appliquant par le squelette **`Map`** une fonction filtrant les éléments. Les données étant accédées sous une forme partitionnée, cette fonction de filtrage ne traitera donc pas des éléments indépendants, mais des tableaux d'éléments. Une fois le filtrage effectué, nous obtenons un nouveau tableau partitionné contenant toujours le même nombre d'éléments, mais ces éléments eux-mêmes sont de tailles différentes. Un second opérateur symétrique au premier permet d'annuler le partitionnement afin de retrouver la structure habituelle du tableau distribué. À l'issue de ces transformations, nous risquons fort d'obtenir un tableau dont la distribution des données n'est plus équilibrée. Un troisième opérateur nous permet donc de réarranger explicitement la distribution des données parmi les nœuds, tout en conservant la cohérence du tableau et donc l'ordre global de ses éléments.

Les trois opérations modifiant la distribution des données sont implémentées au sein d'OSL à travers trois squelettes : **`getPartition`**, **`Flatten`** et **`redistribute`**.

5.2 Squelettes agissant sur la distribution

La répartition des données d'un tableau distribué au sein des processus est conservée dans son attribut `distribution` qui est un vecteur d'entiers positifs contenant le nombre d'éléments présents dans chaque processus ; `distribution[i]` est donc égal à la taille du tableau local stocké dans le processus i. Bien que certains attributs d'un tableau distribué puissent varier d'un processus à l'autre (la taille locale, l'indice de début, les données, ...), sa distribution contiendra obligatoirement les mêmes valeurs pour tous les processus.

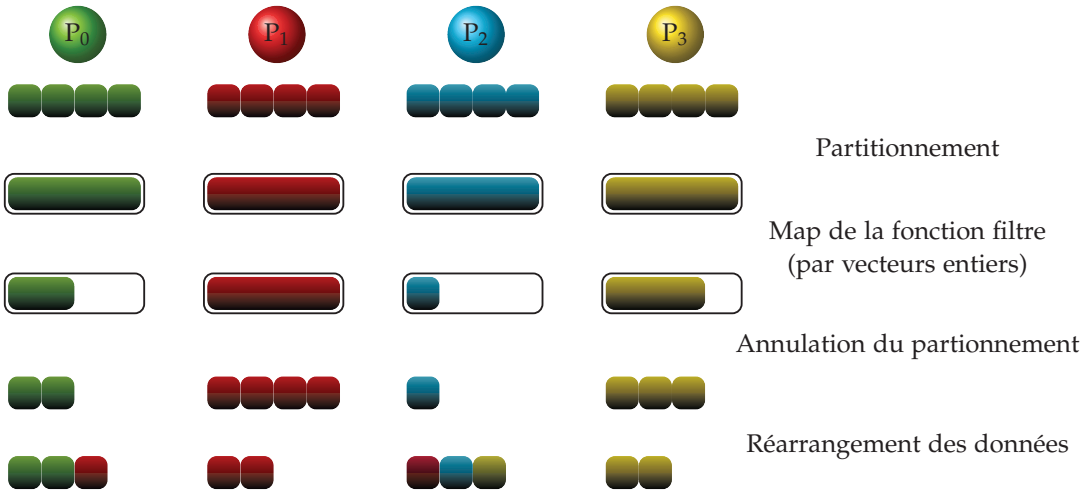


FIGURE 5.2 – *Filtrage de donnée par un mécanisme de haut niveau*

5.2.1 Redistribute

Le squelette **`redistribute`** transforme un tableau distribué en lui affectant une nouvelle distribution de ses éléments au sein de l'ensemble des processus. Bien que leur répartition soit modifiée, cette nouvelle distribution conservera l'ordre global des éléments. À ce titre, ce squelette peut être considéré comme le complémentaire de **`permute`** qui à l'opposé permet de changer l'ordre des éléments mais sans modifier leur distribution.

La distribution à appliquer doit être passée explicitement sous une forme identique à celle de l'attribut `distribution` des tableaux distribués : un `std::vector<int>` où l'entier en position i définit le nombre d'éléments dans le nœud i de la machine parallèle. Le squelette vérifie dans un premier temps que ce vecteur représente bien une distribution pouvant être appliquée au tableau distribué considéré. Le nombre d'éléments du vecteur doit être égal au nombre de processus dans la machine parallèle, chacun des éléments doit être un nombre positif et la somme de ces nombres doit être égale à la longueur globale du tableau `globalsize`.

Les processus doivent ensuite calculer les bornes pour chacun d'entre eux à la fois dans la distribution actuelle et future, afin de déterminer quels éléments doivent être communiqués. Ceci est fait en calculant la somme préfixe des vecteurs de distribution : l'indice global du premier élément du nœud i sera obtenu par `prefixSum(distribution)[i] - distribution[i]`, et l'indice global du dernier élément sera `prefixSum(distribution)[i] - 1`. Chaque nœud compare ensuite ses bornes courantes avec celles des autres nœuds dans la nouvelle distribution pour déterminer quels éléments envoyer. Ces éléments sont échangés entre tous

les nœuds à travers une unique communication globale `mpi::all_to_all`. Étant donné que la redistribution conserve l'ordre global des éléments dans le tableau distribué, chaque nœud peut directement ajouter les éléments reçus dans son vecteur `data` en suivant l'ordre des nœuds les ayant envoyés.

5.2.2 GetPartition

Le squelette **`getPartition`** ne modifie pas les données du tableau distribué, mais en donne une nouvelle vue qui met en avant leur emplacement au sein des processus : les données brutes existant au sein d'un processus sous la forme d'un `std::vector<T>` sont transformées en un `std::vector<std::vector<T> >` dont l'unique élément local est le vecteur précédent. On obtient donc au niveau global un `DArray<std::vector<T> >` à partir d'un `DArray<T>`. Le calcul de la nouvelle distribution est assez évident : chaque nœud n'aura plus qu'un unique élément dans la vue partitionnée, et le calcul des autres attributs découle naturellement de ce fait.

5.2.3 Flatten

Le squelette **`flatten`** effectue la transformation réciproque de **`getPartition`** : étant donné un `DArray<std::vector<T> >`, celui-ci extrait les données de ses vecteurs vers un simple `DArray<T>`. Bien que l'action de **`flatten`** soit symétrique à celle de **`getPartition`**, il n'est pas possible de déterminer localement quelle sera la structure finale du tableau à partir de sa distribution originale car la longueur de chaque partition peut avoir changé localement, et plusieurs partition peuvent se retrouver dans un même processus. Il est donc nécessaire de recalculer globalement les nouveaux attributs : chaque processus va d'abord calculer sa nouvelle taille locale qui sera la somme des longueurs des vecteurs locaux dans la forme partitionnée. Tous les nœuds procèdent ensuite à un échange global de cette information de taille à travers une communication de type `mpi::all_to_all`. Chaque nœud possède ainsi la taille locale de tous les autres nœuds, qu'il peut donc assembler pour construire la nouvelle distribution. La somme de toutes les longueurs de la distribution donne la valeur de la taille globale, alors que cette même somme calculée uniquement pour les longueurs des nœuds précédents donne l'indice de départ d'un nœud donné.

5.2.4 Implémentation du filtrage

Ces trois squelettes étant définis, il est maintenant possible de programmer une fonction `filter` mettant en place le mécanisme de filtrage décrit dans la figure 5.2 :

```
template<class A, class P>
struct select {
   typedef std::vector<A> result_type;
   inline std::vector<A> operator()(P p, std::vector<A> v)const
        {
      std::vector<A> a;
      for(int i=0;i<v.size();i++)
         if (p(v[i])) a.push_back(v[i]);
      return a;
   }
};

template<class A, class P>
DArray<A> filter(P p, DArray<A> d){
   return flatten(
      map(
         std::bind(select<A,P>(),p,_1),
         getPartition(d)
      )
   );
};
```

Cette fonction est générique et permet d'appliquer n'importe quel prédicat `P p`, la seule contrainte étant que ce prédicat soit une fonction prenant en entrée un élément du type de vecteur souhaité, et renvoyant en sortie une valeur booléenne donnant le résultat de l'évaluation de cet élément.

La possibilité de manipuler les distributions ouvre le champ à l'implémentation de nouveaux algorithmes parallèles. Le filtrage en est un exemple très simple mais il existe des algorithmes complexes reposant sur de telles manipulations. Nous allons étudier dans la section suivante un algorithme parallèle de tri dépendant fortement de ces nouveaux mécanismes.

5.3 Tri par échantillonnage régulier en parallèle

L'algorithme de tri par échantillonnage régulier en parallèle est un exemple complexe illustrant l'utilité des nos squelettes permettant de gérer la distribution. Cet

algorithme permet de trier un tableau sur une architecture à mémoire distribuée, tout en assurant une quantité de communication bornée [111, page 29]. Son comportement est illustré dans la figure 5.3. Étant donné un tableau distribué sur N nœuds, chaque processus commence par effectuer un tri de ses éléments locaux (a). Ensuite, il en extrait $N-1$ valeurs placées à intervalles réguliers, et ce pour chacun des sous-tableaux localement triés (b). Ces échantillons sont ensuite rassemblés sur un nœud principal, fusionnés en un unique échantillon (c) qui est à son tour échantilloné de façon régulière, et cet échantillon global est transmis à tous les nœuds (d). Les éléments de cet échantillon global sont ensuite utilisés comme bornes afin de découper les sous-tableaux locaux en N blocs plus petits sur chaque nœud (e). Ces blocs sont ensuite échangés entre les nœuds, en envoyant tous les $i$èmes blocs au nœud i (f). Finalement, chaque nœud peut fusionner ces blocs ce qui fait que chacun d'entre eux possèdera une sous-partie triée du tableau global (g). Étant donné que les mêmes valeurs du tableau d'échantillon global sont utilisées partout pour découper les blocs, nous sommes certains d'obtenir l'ensemble des valeurs finales triées dans l'ordre si nous prenons les sous-tableaux dans l'ordre des nœuds.

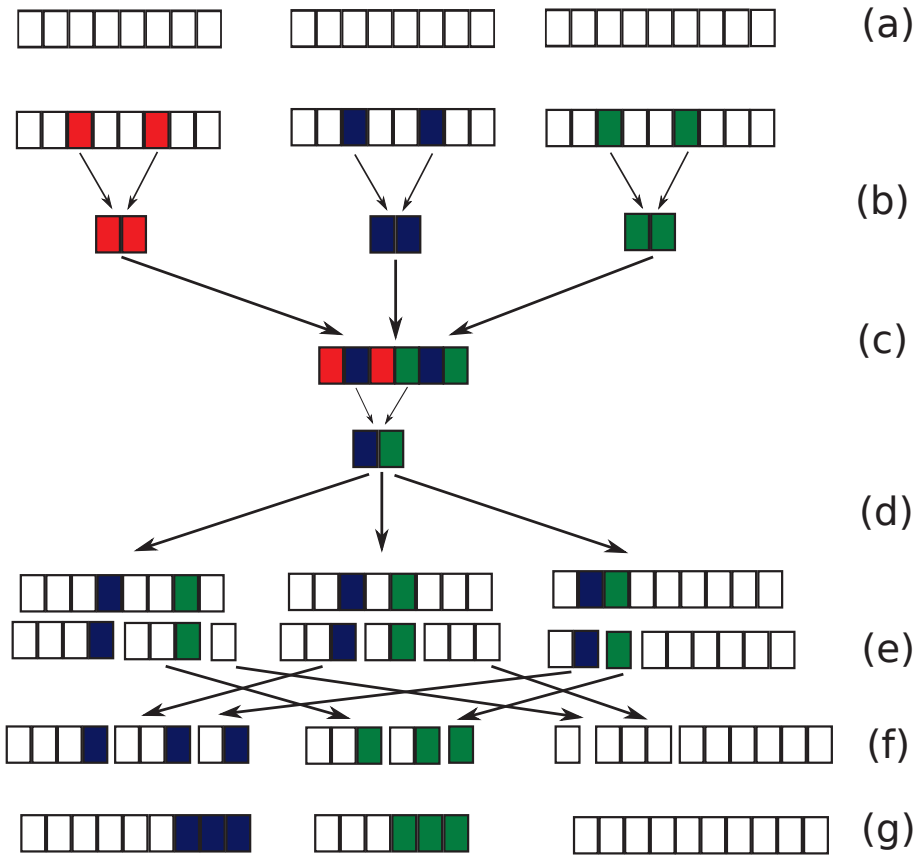


FIGURE 5.3 – *Parallel Regular Sampling Sort*

Comme nous pouvons le voir dans la figure 5.3, pour exécuter cet algorithme il

nous est nécessaire de modifier lourdement la structure du tableau original avec des opérations telles que l'exposition des sous-tableaux locaux, leur division ou fusion, l'extraction de blocs de tailles variables et leur redistribution parmi les nœuds. Ces opérations mettent en avant l'utilité des squelettes décrits précédemment 5.2.

5.3.1 Implémentation

Afin d'implémenter cet algorithme au travers de notre bibliothèque, nous devons en premier lieu implémenter les quelques fonctions séquentielles qui seront appliquées par les squelettes (en pratique, celles-ci prendront la forme de foncteurs). `vector<T>` dénotera ici précisément l'utilisation du type `std::vector<T>`

- `vector<T> localsort(vector<T> t)` cette fonction appliquée à un vecteur produit un vecteur de taille identique dont les éléments sont triés
- `localsample` un foncteur dont le constructeur prend en argument le nombre d'éléments qui seront extraits d'un vecteur entier à intervalles réguliers par son opérateur `vector<T> operator()(vector<T> fullvector)`
- `vector<T> mergevectors(vector<T> v1, vector<T> v2)` fusionne les éléments des deux vecteurs triés en un unique vecteur trié
- `vector<T> localmerge(vector<vector<T> > unmerged)` applique successivement la fonction `mergevectors` à tous les éléments d'un vecteur contenant des vecteurs triés
- `vector<vector<T> > splitvectors(vector<T> separators, vector<T> vector)` décompose un vecteur trié en un vecteur de vecteurs triés de plus petites tailles en utilisant un vecteurs d'éléments séparateurs donnant les bornes de chaque sous-vecteur

Le programme final prend la forme du listing 5.4. Les appels de squelettes sont découpés sur plusieurs lignes afin d'avoir une bonne lisibilité. Grâce à notre mécanisme d'*expression templates*, ce code produit exactement le même résultat que si l'ensemble de ces lignes étaient combinées en une seule expression : le mot-clé **`auto`** dans le standard C++11 permet l'inférence de type, et donc dans notre cas récupère le type de l'objet créé par l'expression. L'objet obtenu sera donc une expression et non le résultat de son évaluation, mais l'utilisateur n'a pas à se préoccuper de la construction du type de cet objet.

5.3.2 Coût BSP

L'analyse du coût BSP de l'algorithme de tri parallèle par échantillonage régulier effectuée dans [111] conclut que, pour un tableau de taille $n > p^3$:

$$W = \mathcal{O}(\frac{n}{p}\log n) \qquad H = \mathcal{O}(g\frac{n}{p}) \qquad L = \mathcal{O}(1)$$

```
/* (a) */ auto locallysorted =
   osl::map(localsort(),
          osl::getPartition(distributedArray) );
/* (b) */ auto samples =
   osl::map(localsample(bsp_p - 1),
          locallysorted);
/* (c+d) */ auto reduction =
   osl::reduce(mergevectors(), sample);
/* (e) */ auto splitted =
   osl::map(std::bind(splitvectors(), reduction, _1),
          locallysorted);
/* (f) */ auto permuted =
  osl::permute(Modulo(bsp_p),
             osl::flatten(splitted))
/* (g) */ auto flattened =
   osl::flatten(
      osl::map(localMerge(),
               osl::getPartition(permuted)
      )
   );
```

FIGURE 5.4 – *Tri par échantillonnage régulier en parallèle dans OSL*

5.3.3 Expérimentations

Nous avons mesuré le passage à l'échelle de notre implémentation du tri parallèle par échantillonage régulier sur la machine SPEED. Nous reprenons le protocole d'expérimentation introduit dans le chapitre 2.4.1.5, à savoir que chaque calcul est effectué 30 fois pour chaque ensemble de paramètres donnés (nombre de cœurs de calcul) et chaque exécution sera totalement indépendante des précédentes. Le gain de performance entre deux ensembles de mesures est le rapport entre leurs médianes et un test statistique est effectué afin de calculer un taux de confiance dans la reproductibilité de l'expérience, c'est à dire la probabilité que l'ajout de nouvelles mesures ne modifie pas le gain calculé. Ce taux de confiance est fixé à un seuil minimum de 95% dans nos expériences, et toutes les mesures ont été validées pour ce seuil.

Nous avons mesuré le gain de la parallélisation de l'algorithme pour deux tableaux de tailles différentes : un million, et cent millions de nombres entiers. Ces tableaux sont obtenus par une unique génération aléatoire, et les mêmes données sont ensuite réutilisées pour toutes les mesures. La figure 5.5 présente les mesures effectuées, l'exécution sur un seul cœur servant de base dans la mesure des performances pour chacune des tailles de tableaux.

Le tri du plus grand tableau devient de plus en plus performant au fur et à mesure

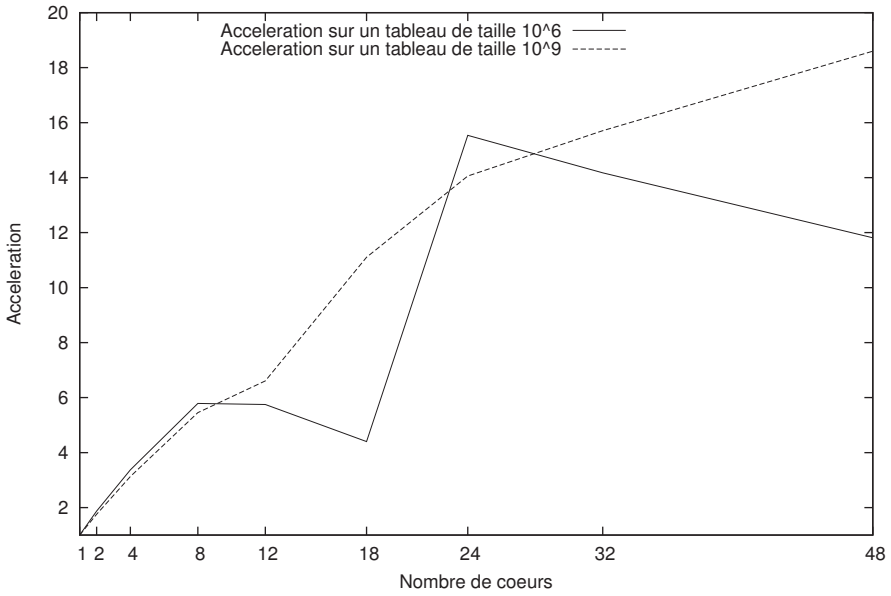


FIGURE 5.5 – *Passage à l'échelle de la parallélisation du tri par échantillonage régulier*

que le nombre de processeurs utilisés augmente, bien que de manière sous-linéaire. Les performances sont moins régulières sur le plus petit tableau, cependant la forme globale de la courbe reste similaire. Étant donné que le temps de calcul est très faible dans ce cas, seulement quelques millisecondes, des déséquilibres dans les communications ont un plus grand impact sur le temps d'exécution global, et ce phénomène a une plus grande probabilité de se produire sur un tableau de taille inférieure.

Des test effectués sur Mirev montrent une dégradation très rapide des performances dès lors que les processus se multiplient sur plusieurs nœuds, les coûts de communication dépassant rapidement les gains de la parallélisation du calcul.

Ces résultats sont cohérents avec les performances prédites par le modèle de coût BSP : dans le cas de la machine SPEED, le coût de communication g mesuré est très faible ce qui entraîne un faible coût de la phase de communication, alors que pour Mirev ce coût est relativement élevé.

5.4 Conclusion

À partir d'un cas d'utilisation simple mais problématique, nous avons mis en avant la nécessité de permettre à l'utilisateur de manipuler la structure des tableaux distribués. Nous avons en conséquence défini au sein d'OSL trois squelettes permettant de modifier la distribution des tableaux : **`getPartition`**, **`flatten`** et **`redistribute`**.

Ces trois squelettes ont ensuite été mis en application au sein d'un algorithme complexe de tri parallèle, les résultats expérimentaux montrent que cette implémentation offre des résultats corrects et passant à l'échelle sur de nombreux cœurs de calcul.

Notre contribution précise dans cette section aura premièrement été l'introduction du squelette **`redistribute`**, qui généralise le comportement de squelettes existants précédemment (broadcast, gather et balance). Nous avons ensuite retravaillé l'implémentation de **`getPartition`** et **`flatten`** au sein de la version actuelle d'OSL, et mis en place l'algorithme de tri parallèle. Celui-ci avait déjà été programmé avec succès dans la version prototype d'OSL en BSML, cependant cela n'avait pas été possible dans la première version C++ à cause de problèmes techniques concernant le partitionnement.

Un défaut important des squelettes **`getPartition`**, **`flatten`** et **`redistribute`** présentés dans ce chapitre est leur manque d'efficacité dans l'utilisation de la mémoire. En effet, l'application de l'un d'entre eux entraîne systématiquement la création d'une copie du tableau distribué en entrée. Ce comportement respecte la sémantique des squelettes, cependant il arrive fréquemment que le tableau en entrée n'ait plus besoin d'être utilisé une fois la version transformée créée. Dans ce cas, il est préférable de modifier directement le tableau distribué existant ce qui évite une coûteuse copie. Ceci est particulièrement vrai pour **`getPartition`** qui n'applique aucune modification sur les données et ne modifie que la structure du tableau distribué.

Des développements actuellement en cours sur cette partie de la bibliothèque visent donc à ajouter un mécanisme permettant de modifier directement le tableau en entrée lors de l'application des squelettes. Ce mécanisme ne doit cependant pas être appliqué de manière systématique, il faudra donc y adjoindre dans un premier temps la possibilité pour l'utilisateur de contrôler manuellement cette optimisation. À plus long terme, nous pouvons également envisager de mettre en place un système d'analyse globale des expressions de sorte à déterminer automatiquement quand cette optimisation peut être appliquée de manière sûre au sein d'une expression de squelettes.

Un autre point susceptible d'être amélioré est le mécanisme d'application du squelette **`flatten`**. Lorsque ce squelette est utilisé sur un tableau partitionné, chaque processus doit calculer le nombre d'éléments qu'il possèdera localement après avoir supprimé le partitionnement, puis un échange global de cette valeur a lieu entre tous les processus afin de pouvoir déterminer globalement la nouvelle distribution du tableau. Ces calculs et communications pourraient néanmoins être évités dans certains cas, par exemple lorsque les squelettes appliqués sur le tableau partitionné ne modifient pas sa structure (auquel cas il suffit de récupérer directement les attributs du tableau avant son partitionnement) ou lorsque l'impact des modifications peut être calculé pour l'ensemble du tableau directement dans chaque processus (par exemple, le résultat d'une permutation de partitions peut être calculé en permutant les valeurs de la distribution d'origine). La mise en place d'un tel mécanisme nécessiterait de conserver au sein des

tableaux distribués leurs attributs lors de l'application de **`getPartition`**, ce squelette pouvant être appliqué successivement un nombre quelconque de fois, nous pouvons envisager l'utilisation d'une structure de pile contenant les attributs successifs du tableau lors de chaque partitionnement. La mise en place de ces optimisations permettrait ainsi à l'utilisateur de manipuler la structure des tableaux facilement à travers des squelettes de haut niveau, tout en produisant un impact faible voire négligeable sur les performances du programme généré.

Une autre direction intéressante à explorer concernant la distribution des données serait celle de l'équilibrage. Bien que nous fournissions au développeur les moyens de distribuer librement les données, les architectures matérielles évoluent vers une hétérogénéité importante et les calculs appliqués par un programme à squelettes ne sont pas forcément équilibrés même si les données le sont, il peut donc être difficile pour le programmeur de déterminer quelle est la distribution des données entraînant les meilleures performances de son programme. Il serait donc intéressant d'étudier la mise en place d'un mécanisme d'équilibrage automatique, par exemple dans les calculs itératifs en mesurant le temps de calcul demandé par chaque noeud et en redistribuant afin de l'optimiser globalement ; ce type d'approche dans les applications MPI montre des résultats intéressants sur les grappes hétérogènes [83].

Gestion des exceptions parallèles

6

Sommaire

6.1 Support de la gestion des exceptions en parallèle 100
6.1.1 Sérialisation grâce à Boost 105
6.1.2 Relancer les exceptions 106
6.2 Implémentation dans OSL 108
6.2.1 Le squelette forwardExceptions 108
6.2.2 Retour sur trace avec exceptions en parallèle 109
6.2.3 Mesures expérimentales 110
6.3 Conclusion 112

Une version préliminaire des travaux présentés dans ce chapitre a été publiée dans les actes de la conférence HPCS 2013 [76].

De par l'augmentation de la complexité des programmes et des systèmes informatiques, il devient très probable que des évènements anormaux ou exceptionnels se produisent lors de l'exécution des programmes. Ce phénomène est d'autant plus exacerbé en présence d'architectures parallèles. Même dans le cadre d'un système certifié, les opérations d'entrées/sorties, d'allocation de mémoire ou plus généralement toutes les opérations faisant intervenir le matériel, sont susceptibles de provoquer des erreurs. Il est souhaitable que de tels évènements n'entraînent pas l'arrêt du programme ou du système, conséquemment les langages de programmation modernes proposent des mécanismes permettant de gérer de tels évènements.

Les travaux sur la gestion des exceptions commencèrent dans les années 70 [56], et aujourd'hui ces mécanismes sont devenus bien plus qu'un moyen de gérer les erreurs : ce sont dorénavant des éléments structurels des langages et de leurs modèles d'exécution. On peut distinguer trois rôles remplis par les mécanismes de gestion d'exceptions :

(1) La capacité à continuer l'exécution d'un programme en l'absence d'un résultat dont le calcul a échoué. Il peut simplement s'agir d'indiquer précisément à l'utilisateur la nature du problème avant de terminer l'exécution. Une solution possible est de renvoyer une valeur de retour spécifique, et d'avoir une valeur différente pour chaque scénario d'erreur possible. C'est la solution mise en place en

C par exemple avec la valeur de retour entière renvoyée par la fonction `main`, ou la valeur de retour de certaines fonctions MPI [103] qui permettent de signaler une erreur lors de l'exécution de ces fonctions. Cependant ces cas n'entrent pas dans la catégorie des mécanismes de gestion d'exceptions car il requièrent un traitement *explicite* des cas exceptionnels lors de tous les appels. Ceci pollue le code et le rend moins lisible, moins compréhensible et plus difficile à maintenir. Un mécanisme de gestion d'exceptions introduit un traitement *implicite* des cas exceptionnels, la quantité de code supplémentaire est faible, et traiter les cas exceptionnels au travers de structures spécifiques du langage rend le code plus clair.

(2) Lorsqu'une exception est levée, le flot d'exécution normal est interrompu, et se retrouve redirigé vers un code spécifique afin de traiter le cas exceptionnel. Une exception devrait donc déclencher un traitement particulier sans devoir parcourir pas-à-pas la pile d'appels de fonctions comme c'est le cas dans les appels de fonctions normaux.

(3) Un mécanisme de récupération et réessaie permet de revenir à un état sain du programme ou de retenter l'opération qui a échoué avec des paramètres différents.

6.1 Support de la gestion des exceptions en parallèle

Étant donné que les application OSL sont des programmes C++, l'utilisateur peut déjà intercepter les exceptions au niveau de chaque élément des tableaux distribués dans les fonctions qui seront appliquées par les squelettes. Il est également possible d'intercepter les exceptions levées lors de l'évaluation d'une expression de squelettes avec les instructions séquentielles habituelles **`try`**/**`catch`**, cependant ceci ne permet que de récupérer les exceptions levées localement à un processus donné. Ceci est problématique car cela pourrait amener certains processus à suivre des chemins d'exécution différents et ainsi compromettre la consistance ou la synchronisation implicite de notre modèle de programmation par squelettes.

Un exemple caractéristique de ce problème est illustré dans la figure 6.1. Les données d'un tableau sont distribuées sur plusieurs processus, ces données sont transformées sur chacun des nœuds (application d'un squelette de type **`Map`**), puis ces données transformées sont réduites en un seul élément (application d'un squelette de type **`Reduce`**). L'application du **`Map`** est susceptible de lever une exception pour certaines valeurs. Ces cas peuvent avoir été prévus dans la fonction appliquée sur chaque élément par **`Map`** ce qui permet de récupérer localement l'exception levée et de continuer l'exécution. Cependant si aucun traitement particulier n'est effectué lors de la réduction, le résultat final risque de ne pas être correct sans qu'il soit possible de déterminer globalement que l'exécution s'est produite de manière incorrecte sur un processus. Un exemple concret pourrait être le calcul de la somme des racines carrées d'un tableau :

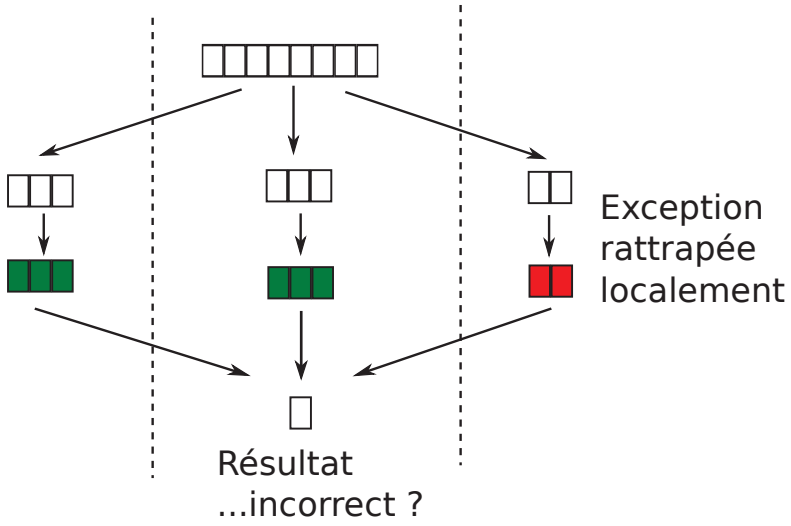


FIGURE 6.1 – *Problématique de la récupération locale*

```
struct SafeSQRT
{
      float operator()(float input) const
      {
            try {
                  return sqrt(input);
            } catch (std::domain_error & de) {
                  // Cas problematique si input est negatif
                  return 0.0;
            }
      }
} safeSQRT;
// dans le programme principal
float result = osl::reduce(std::plus<float>(), osl::map(
   safeSQRT, inputArray));
```

Il est possible de traiter ce problème en écrivant une valeur spécifique dans le résultat du **Map** lorsqu'une exception se produit, et ensuite de filtrer cette valeur dans la fonction appliquée par **Reduce** ; ce mécanisme reproduit en quelque sorte le retour de code d'erreur des fonctions système :

```
struct SaferSQRT
{
      float operator()(float input) const
      {
            try {
```

```
                return sqrt(input);
            } catch (std::domain_error & de) {
                return -1.0;
            }
        }
} saferSQRT;

struct SafeAdd
{
        float operator()(float i1, float i2) const
        {
            if ((i1 == -1.0) || (i2 == -1.0)) {
                return -1.0;
            } else {
                return i1 + i2;
            }
        }
} safeAdd;
// dans le programme principal
float result = osl::reduce(safeAdd, osl::map(saferSQRT,
   inputArray));
if (result == -1.0) throw(new std::domain_error());
```

Cependant une telle solution alourdit fortement le code et dégrade les performances étant donné que chaque squelette doit vérifier la présence de valeur représentant une erreur dans le résultat de l'application des squelettes précédents. De plus, cette solution ne peut fonctionner dans le cas général que si l'expression de squelettes permet de propager les valeurs d'erreur sur tous les processus.

La solution que nous souhaitons mettre en place pour résoudre ce problème consiste en un mécanisme permettant de récupérer les exceptions pouvant se produire lors de l'évaluation d'une expression de squelettes puis les transmettre à tous les processus du programme afin de pouvoir traiter les exceptions de manière globale et transparente. Ainsi nous conservons un avantage important des exceptions, qui est que le code décrivant leur traitement dans un programme est clairement séparé du code décrivant l'exécution normale.

Étant donné qu'un de nos buts est d'informer tous les processus lorsqu'une exception se déclenche dans l'un d'entre eux, nous avons intégré un mécanisme complet de transmission d'exceptions au sein d'OSL dans un squelette nommé **forwardExceptions**. Ce squelette permet donc de relancer globalement les exceptions s'étant produites sur un ou plusieurs processus particuliers sans influencer l'évaluation de l'expression à laquelle il est appliqué. Sémantiquement ce squelette renvoie donc exactement les données qu'il reçoit en entrée, et le cas échéant lève en tant qu'ex-

ception un vecteur contenant les exceptions s'étant produites sur les différents processus. Il nous permet donc d'ajouter la gestion des exceptions à un programme OSL sans qu'il y ait besoin de modifier les appels de squelettes ou les fonctions qu'ils appliquent :

```
struct SimpleSQRT
{
      float operator()(float input) const
      {
            return sqrt(input)
      }
} simpleSQRT;

// dans le programme principal
try {
 result = osl::forwardExceptions(osl::reduce(std::plus<float>()
    , osl::map(simpleSQRT, inputArray)));
} catch (std::vector<std::exception *> exceptions) {
// Traitement des exceptions
}
```

Pour implémenter une gestion complète des exceptions au sein d'OSL, nous devons pouvoir être capable de déterminer globalement depuis chaque processus si un (ou plusieurs) autre processus a levé une exception. Le mécanisme que nous mettons en place est illustré dans la figure 6.2 : lorsqu'une exception est levée lors de l'évaluation d'un squelette, le résultat de l'évaluation reste disponible là où il est attendu mais l'exception doit être isolée et envoyée à tous les nœuds. Chaque nœud devra ensuite relancer localement la ou les exceptions reçues, ainsi tous les nœuds verront le déclenchement de cette exception et le programmeur pourra effectuer globalement un traitement adéquat sur le résultat, de manière similaire au traitement d'une exception dans un programme séquentiel.

Nous devons donc être capable de communiquer les exceptions entre les processus, cependant la conception des mécanismes de gestion des exceptions en C++ entraîne plusieurs difficultés pour ce faire. Les exceptions sont naturellement polymorphes : toutes les exceptions fournies par la librairie standard sont dérivées de la classe de base `std::exception`, et les utilisateurs sont encouragés à créer leurs propres types d'exception similairement en dérivant de `std::exception`, bien que le langage autorise n'importe quel type d'objet à être lancé en tant qu'exception.

Les exceptions C++ possèdent les mêmes propriétés que tout autre objet, cependant le standard défini que leur lancement est effectué par valeur ce qui va donc en créer une copie à partir de leur type dynamique, alors que les objets sont habituellement copiés à partir de leur type statique lorsqu'ils sont passés comme paramètres

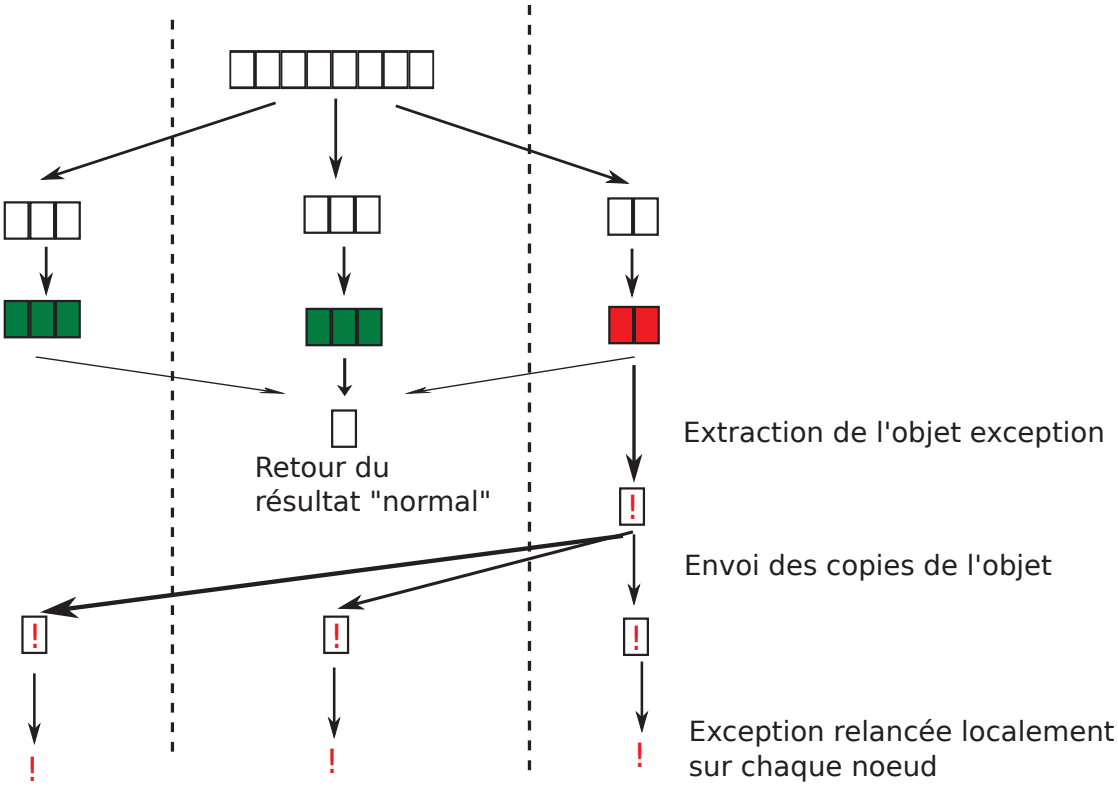


FIGURE 6.2 – *Solution envisagée*

de fonction. De plus, nous ne pouvons écrire les blocs d'interception des exceptions en les filtrant selon leur type statique. Le bloc d'interception générique **`catch(...)`** peut être utilisé pour attraper tout type d'exception, mais il ne donne pas accès à l'objet de l'exception et n'est donc pas adapté si nous souhaitons renvoyer l'exception à un autre processus. Traiter tous les types d'exceptions possibles séparément, chacun dans leur propre bloc, serait extrêmement pénible à écrire, difficilement extensible et très inefficace car il faudrait échanger entre les processus un message différent pour chacun des types d'exception existants.

Nous ne pouvons accomplir cette tâche efficacement qu'en interceptant le type d'exception le plus général possible, le sérialiser puis le transmettre en une seule communication entre les processus. La difficulté réside principalement dans le fait qu'il n'est pas directement possible de connaître le type dynamique de l'exception interceptée, mais seulement le type statique de la classe de base. De plus nous devons être capable de sérialiser l'objet dans un processus puis de le désérialiser pour créer un objet du type dynamique dérivé dans les autres processus.

Une solution naïve est de créer une hiérarchie d'objets dérivant d'une classe de base virtuelle exigeant de ses classes dérivées l'implémentation d'une fonction de sérialisation, une fonction de création d'objet et une fonction de lancement d'exception.

Les processus récepteurs peuvent ensuite utiliser une implémentation du *design pattern* classique « abstract factory » afin de procéder à la création d'un objet exception du type correct et le propager localement [9, 99].

Cette solution a cependant un grand inconvénient : l'utilisateur est obligé d'utiliser une hiérarchie d'exceptions spécifique. Nous souhaitons rester proche du standard et fournir à l'utilisateur la hiérarchie d'exceptions habituelle de la bibliothèque standard, mais malheureusement ces exceptions ne possèdent pas les fonctions de sérialisation et de construction polymorphe nécessaires à leur transmission.

Cependant, la bibliothèque Boost fournit des outils pouvant être utilisés pour résoudre ces problèmes.

6.1.1 Sérialisation grâce à Boost

Nous avons vu dans le chapitre 4.1.3 que la bibliothèque Boost.Serialization fournit les opérateurs de sérialisation pour la plupart des types d'objets courants du langage C++, mais malheureusement ce n'est pas le cas pour les exceptions. Écrire un tel opérateur pour la classe de base est très simple étant donné que celle-ci ne possède aucun attribut :

```
template<class Archive>
void serialize(Archive & ar, std::exception & e, const unsigned
    int version){}
```

Plus de modifications doivent être effectuées pour les classes telles que `std::logic_error` et ses dérivés. Celles-ci ne possèdent pas de constructeur par défaut, mais uniquement un constructeur prenant une chaîne de caractères en argument. Pour pouvoir appeler un constructeur particulier lors de la désérialisation, nous devons surcharger les opérateurs `save_construct_data` et `load_construct_data` qui permettent de découper plus finement la tâche de l'opérateur `serialize`.

```
template<class Archive>
inline void save_construct_data(Archive & ar,
            const myclass * c, const BOOST_PFTO unsigned int)
{
   std::string s(c→ what());
   ar ≪ s;
}
template<class Archive>
inline void load_construct_data(Archive & ar, myclass * c,
   const unsigned int)
{
   std::string s;
```

```
    ar ≫ s;
    ::new(c)myclass(s);
}
```

Il nous faut maintenant ajouter la possibilité de sérialiser nos exceptions à travers un pointeur de la classe de base `std::exception`, étant donné que les processus recevant ces objets sérialisés ne pourront obtenir aucune information concernant le type dynamique de ces objets. Boost.Serialization gère implicitement les pointeurs, et inclut des mécanismes permettant de reconstruire un objet d'un type dérivé à partir d'un pointeur vers sa classe de base. La première étape est d'inclure explicitement la sérialisation de la classe de base dans l'opérateur de sérialisation de la classe dérivée :

```
template<class Archive>
void serialize(Archive & ar, myclass & c,
      const unsigned int version)
{
    ar & boost::serialization::base_object<baseclass>(c);
}
```

Boost.Serialization maintient une liste de références vers les types des classes dérivées afin de permettre leur identification à partir d'un pointeur de base et invoquer le code correspondant. Ajouter une nouvelle classe dans cette liste se fait simplement en appelant la macro `BOOST_CLASS_EXPORT_KEY(classname)`. Un dernier problème vient du fait que l'opérateur de sérialisation est une fonction `template`, si nous ne travaillons qu'avec des pointeurs vers la classe de base, cet opérateur ne sera donc jamais instancié. Boost.Serialization fournit encore une fois une macro pour pallier à ce problème : `BOOST_CLASS_EXPORT_IMPLEMENT(classname)` qui instanciera explicitement le code nécessaire.

La sérialisation des exceptions standards se fait toujours selon un schéma identique, la seule différence se faisant au niveau du constructeur, selon qu'il soit sous la forme par défaut ou qu'il se fasse sur une chaîne de caractère. Nous fournissons deux macros permettant aux utilisateurs d'implémenter facilement la sérialisation de leurs propres classes d'exception, à partir du moment où celles-ci dérivent d'une classe d'exception standard et n'ajoutent pas de nouveaux attributs ou méthodes.

6.1.2 Relancer les exceptions

Nous sommes maintenant capables d'intercepter les exceptions sur des processus distants et les échanger sous la forme de pointeurs d'objets de type `std::exception`. Grâce aux mécanismes de Boost.Serialization, les objets pointés sont créés avec le type dérivé adéquat. Cependant nous ne pouvons pas directement lancer à nouveau ces

exceptions après leur réception à cause de la nature du mécanisme de lancement d'exceptions : les objets sont copiés par valeur lorsqu'ils sont lancés en tant qu'exceptions.

En C++, lorsqu'un objet est copié par valeur son type statique est utilisé, donc dans le cas d'un objet de type dérivé que l'on accède à travers un pointeur du type de base le constructeur par copie de la classe de base sera utilisé. Dans notre cas particulier, nous lancerons donc toujours un objet de type `std::exception`.

Nous avons donc besoin ici de convertir le pointeur en son type dérivé adéquat avant de lancer l'objet en tant qu'exception. Bien que nous puissions obtenir des informations sur le type dynamique des objets grâce à l'opérateur **`typeid`**, nous ne pouvons pas l'utiliser pour obtenir explicitement le type d'un objet, et donc toutes les opérations de conversion de type doivent être définies de manière statique. Une solution basique serait d'écrire une grande structure de choix (**`switch`**) comparant le **`typeid`** de l'objet pointé avec celui de tous les types d'exceptions possibles puis de procéder à la conversion de type idoine, mais cette solution est peu extensible.

Nous proposons une solution alternative basée sur le patron de conception de fabrique abstraite [53], dont le but est de pouvoir créer des objets dérivés d'une classe de base abstraite sans devoir spécifier explicitement leur type exact.

Dans notre cas, il ne s'agit pas de construire des objets mais plutôt de convertir un pointeur d'exception en son type dérivé exact, puis de lever une exception avec cet objet correctement typé. Nous avons créé une hiérarchie de foncteurs simples afin de remplir ce rôle : un `ThrowFunctor` abstrait définissant un opérateur virtuel `rethrow()` qui convertit un pointeur d'exception en son type dérivé puis le lance, et un foncteur dérivé implémentant cet opérateur pour chaque type d'exception géré.

Afin d'être capable d'appeler le foncteur correspondant précisément au type dérivé d'un pointeur d'exception, nous avons créé une classe « fabrique abstraite de lancement » (*abstract throwing factory*) qui enregistre les différentes classes d'exception avec leur foncteur associé. Une fonction générale `rethrow()` est fournie par la fabrique qui compare un pointeur en entrée avec les classes enregistrées et appelle le `ThrowFunctor` adéquat.

Il ne nous reste plus désormais qu'à enregistrer dans la fabrique abstraite de lancement toutes les paires de **`typeid`** avec leur foncteur associé que nous souhaitons pouvoir traiter. Nous attribuons ce rôle à une « fabrique de lancement » qui devra être créée pour chaque type d'exception. Le constructeur de cette fabrique procède simplement à l'enregistrement de la paire **`typeid`**/foncteur, et chaque fabrique de lancement ne sera statiquement appelée qu'une unique fois au début du programme afin d'assurer que ce constructeur ne soit appelé qu'une fois.

Étant donné que chaque foncteur et fabrique de lancement exécute exactement les mêmes instructions pour tous les types d'exceptions (seul le type de la classe considéré changera), nous avons défini une macro permettant de construire automatiquement ces objets. Ainsi, l'utilisateur peut ajouter ses propres classes d'exceptions à la fabrique

de lancement abstraite en appelant `OSL_BUILD_THROW_FACTORY(maclasse);`. Finalement, un simple appel à la méthode `AbstractThrowingFactory::rethrow();` permettra de relancer tout type de pointeur d'exception pour peu que ce type ait été intégré dans le système de la fabrique abstraite grâce à la macro précédente.

6.2 Implémentation dans OSL

6.2.1 Le squelette forwardExceptions

Nous avons résolu les difficultés de l'envoi et du relancement d'exceptions sur des processus distants, nous pouvons donc maintenant utiliser les exceptions standards dans les squelettes de communication de manière transparente comme n'importe quel autre type de données, et nous intégrons l'ensemble de ce mécanisme dans les squelette **`forwardExceptions`**.

Ce squelette ne devrait pas avoir d'impact direct sur le résultat d'une expression auquel celui-ci serait appliqué, sa valeur de retour sera donc exactement la même que la valeur de retour de l'expression à évaluer. Nous devons également rompre l'optimisation de fusion des *expression templates* lors de l'application de **`forwardExceptions`** pour deux raisons. La première est la cohérence : si nous souhaitons transmettre uniquement les exceptions levées par une sous-partie d'une expression, nous devons assurer que les boucles de calcul de cette sous-expression ne se retrouvent pas fusionnées avec celles du reste de l'expression au-delà de la portée du squelette `forwardException`. L'autre raison est l'efficacité et la simplicité : lorsque l'optimisation de fusion se produit, la boucle résultante va appliquer subséquemment les squelettes de l'expression à chaque élément du tableau distribué. Cela impliquerait que le mécanisme de transmission d'exception serait mis en place individuellement pour chaque élément du tableau distribué. Ceci entraînerait un surcoût de calcul et surtout de communications car même si aucune exception n'est levée, nous devrions quand même envoyer la sérialisation d'un pointeur nul pour informer tous les autres processus.

Ce que nous voulons ici est déterminer si une exception s'est produite dans un *processus* particulier et non sur un *élément* particulier du tableau. Ceci peut être effectué grâce à la fonction `Evaluate`. Cette fonction force l'évaluation d'une expression en affectant le résultat de cette expression à une valeur temporaire, et en retournant ensuite le contenu de cette variable temporaire.

La structure du squelette `forwardException` devient donc simple : celui-ci appelle `Evaluate` sur l'expression considérée, puis renvoie ce résultat. Cette évaluation se produit à l'intérieur d'un bloc **`try`**/**`catch`**, si une `std::exception` est interceptée (ce qui inclut donc implicitement tout type d'exception dérivant de ce type) nous sérialisons un pointeur vers celle-ci puis l'échangeons avec tous les autres processus à

travers un appel à `MPI_Alltoall`. Si aucune exception n'est levée, un pointeur nul est envoyé. Après l'appel au squelette `forwardException`, nous obtenons donc sur chaque processus un tableau de pointeurs de `std::exception` contenant l'ensemble des exceptions ayant été levées sur chaque processus. Si ce tableau contient au moins une valeur non nulle, celui-ci est relancé localement comme exception sur chacun des nœuds.

6.2.2 Retour sur trace avec exceptions en parallèle

En plus de nous donner la possibilité de reprendre une exécution correcte de nos programmes parallèles après une erreur, notre mécanisme de transmission d'exceptions améliore les possibilités d'expressivité de notre bibliothèque, par exemple en permettant l'implémentation en parallèle d'un algorithme de retour sur trace avec exceptions.

Le retour sur trace est un algorithme classique utilisé afin de trouver la solution de problèmes de satisfaction de contraintes notamment. Il effectue cette tâche en construisant incrémentalement un arbre de solutions partielles au problème, et en éliminant progressivement les branches dès qu'elles sont identifiées comme ne pouvant pas mener à une solution complète. Cet algorithme peut être appliqué à tout problème à condition qu'il soit possible de construire des solutions partielles et d'évaluer facilement leur validité.

Cet algorithme est particulièrement bien adapté aux problèmes de satisfaction de contraintes tels que, par exemple, les N reines ou le remplissage d'une grille de sudoku. Dans ce dernier problème, construire une solution partielle consiste à ajouter un chiffre quelconque dans une case vide de la grille, et le test de validité consiste à vérifier qu'aucun chiffre n'est présent plusieurs fois dans la même ligne, la même colonne ou le même bloc.

Les mécanismes d'exception sont couramment utilisés pour améliorer la simplicité de programmation et la performance de ce type d'algorithme. L'idée consiste simplement à lever une exception lorsqu'une solution complète est trouvée, et d'intercepter cette exception en dehors de la fonction de retour sur trace. Ceci court-circuite instantanément l'évaluation de toutes les solutions partielles restantes, et évitera de devoir propager le résultat des évaluations à travers la pile d'appels récursifs.

Étant capables d'intercepter des exceptions depuis des processus distants dans OSL, nous pouvons facilement écrire une version parallèle de cet algorithme avec le foncteur suivant :

```
struct BackTrack{
  inline int operator()(Node s1) {
    if (is_valid(s1)){
      std::vector<Node> children = s1.build_children();
```

```
    if (children.size() == 0)
      throw(MyException(s1.value()));
    else
      for (int i = 0; i < children.size(); i++)
            backtrack(children[i]);
   }
  }
};
```

Ce foncteur peut être utilisé de manière transparente dans notre bibliothèque. Afin de diviser l'arbre d'évaluation et avoir ainsi suffisamment d'éléments à traiter séparément pour répartir les calculs sur plusieurs processus, nous créons dans un premier temps un tableau contenant des fils (c'est à dire des solutions partielles différentes) de la racine du problème. Ensuite, nous extrayons une partie de ces nœuds dans un tableau distribué, et il ne nous reste plus qu'à appliquer localement le foncteur de retour sur trace à chacun des éléments de ce tableau distribué grâce au squelette **map**. Ce processus est répété tant qu'une exception n'est pas interceptée au niveau global.

```
std::vector initArray = rootnode.build_children();
while (!result) {
 try {
  DArray<Node> array(getSomeNodes(initArray));
  auto res = forwardExceptions(osl::map(backtrack(), array));
 } catch(std::vector<std::exception *> e){
  for (int i = 0; i < bsp_p; i++)
   if(e[i] != NULL) { result = true;
    /* nous pouvons ici extraire le resultat
     contenu dans l exception */ }
 }
}
```

6.2.3 Mesures expérimentales

Nous avons effectué deux séries de mesures en utilisant le protocole expérimental mis en place précédemment dans le chapitre 3 afin d'obtenir des mesures statistiquement fiables. Pour cela, les mesures des temps d'exécution sont répétées chacune 30 fois de façon indépendante, et les jeux de mesures sont comparés entre eux à travers un test statistique nous permettant de calculer un facteur de gain de performance médian, et un taux de confiance (défini comme devant être supérieur à 95% dans nos expériences) dans ce facteur est calculé.

Premièrement, nous nous sommes intéressés au surcoût entraîné par la transmission des exceptions sur une simulation distribuée de diffusion de chaleur en deux

dimensions implémentée avec OSL (figure 6.4). Cette simulation est exécutée durant 100 itérations sur deux tailles de grilles différentes (1000 par 1000 et 3000 par 3000), ces grilles étant remplies aléatoirement par des valeurs entre -30 et 30, et les valeurs des conditions aux bornes pour la simulation sont respectivement de -10 à gauche, 30 à droite, 20 en haut et 50 en bas. Ce surcoût a été évalué sur la grappe Mirev pour un nombre variable de processus.

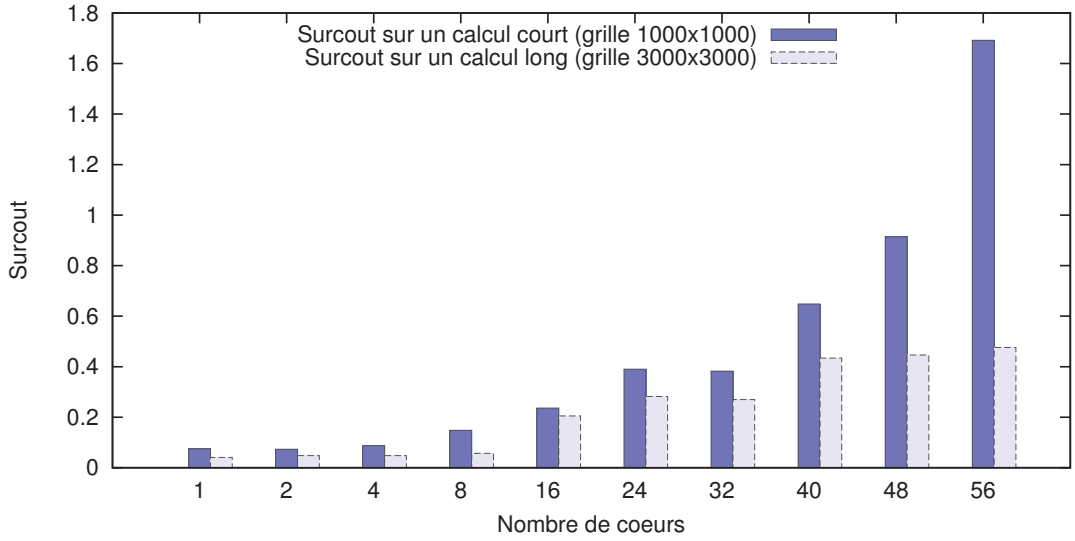


FIGURE 6.3 – *Surcoût des exceptions dans une simulation*

Le surcoût dû au traitement des exceptions est très faible tant que l'on reste sur un petit nombre de processeurs, mais cette situation se dégrade dès lors que leur nombre augmente. Ce phénomène est cohérent étant donné que le coût d'une communication de type `MPI_Alltoall` augmente quadratiquement par rapport au nombre de processus alors que la vitesse de calcul du programme n'augmente que de manière sous-linéaire. À partir d'un certain point, les gains sur le calcul sont contrebalancés par le coût de communication de la transmission d'exceptions. Sur un très petit calcul (par exemple une grille de 100 par 100), le coût des communications est d'emblée supérieur au gain obtenu par la parallélisation des calculs.

Nous avons ensuite mesuré les performances de l'algorithme parallèle de retour sur trace pour résoudre un problème de sudoku afin de comparer la version utilisant les exceptions et celle ne les utilisant pas (figure 6.4). Nous sommes partis d'un problème de sudoku contenant un nombre minimal d'éléments de départ auquel nous avons ajouté quelques éléments de la solution afin de raccourcir volontairement les temps de calcul pour effectuer les mesures en un temps raisonnable, notre but ici n'étant pas de mesurer les performances absolues de l'algorithme mais de comparer les résultats avec et sans gestion des exceptions.

Ces expérimentations sur les algorithmes de retour sur trace ne montrent pas un très bon passage à l'échelle. Ceci est dû à un mécanisme d'équilibrage des calculs très

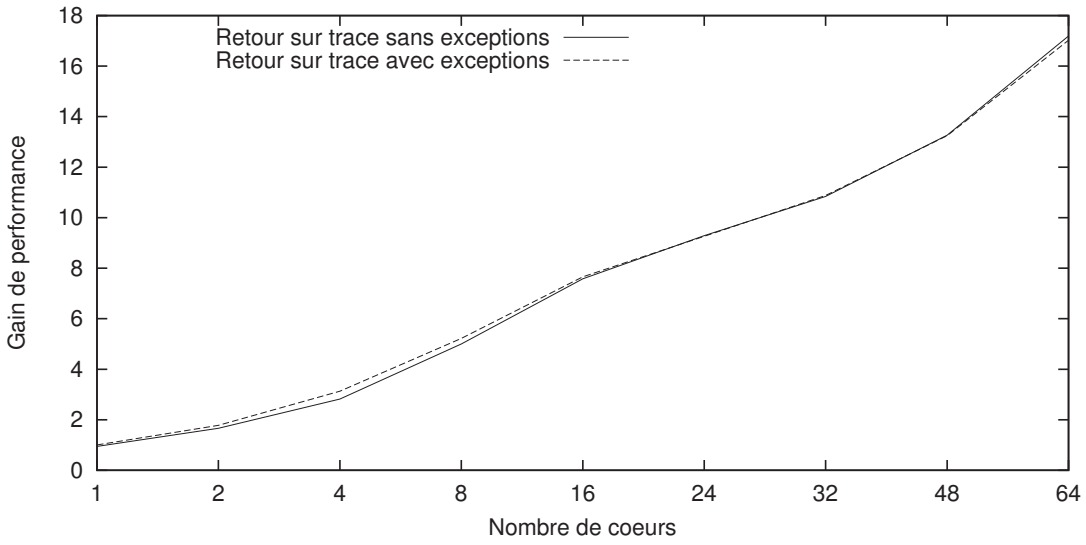


FIGURE 6.4 – *Solveur de sudoku par retour sur trace*

basique dans cet exemple. Cependant, les deux versions de l'algorithme présentent des performances quasiment identiques, avec un léger avantage pour la version avec exceptions. Ceci illustre bien l'utilité de notre mécanisme de transmission d'exceptions au niveau de l'expressivité des programmes qu'il est possible de créer : dans l'algorithme sans exceptions, nous devons explicitement collecter les résultats sur chaque processus, puis transmettre le résultat (ou son absence) à tous les processus à chaque itération de la boucle principale. Ceci nous oblige à utiliser plus d'appels de squelettes, introduire des nouveaux foncteurs et gérer la propagation de valeurs de retour dans la pile d'appel de la fonction de retour sur trace `SudokuSolve`. Tout cela entraîne au final une augmentation d'environ 50% de la taille du programme.

6.3 Conclusion

Nous avons mis en place un mécanisme permettant de gérer les exceptions dans le contexte distribué d'OSL à travers le squelette **`forwardExceptions`**. La principale difficulté était due à la nature des exceptions en C++, notamment le mécanisme de récupération qui ne permet de filtrer les exceptions que sur leur typage statique alors que les exceptions levées lors de l'exécution peuvent voir leur type dynamique varier fortement. Ces particularités rendent la communication et le relancement d'exceptions sur d'autres processus difficiles. Une fois ces obstacles levés la nature extensible d'OSL nous a permis d'intégrer facilement la récupération et le relancement des exceptions au sein d'un nouveau squelette. Les difficultés rencontrées sont particulières au langage C++. Ainsi d'autres bibliothèques de squelettes basées sur ce même langage telles que Muesli [33] ou SkeTo [84], qui offrent des fonctionnalités proches d'OSL sur des

structures de données distribuées, ne disposent pas de mécanisme global de gestion des exceptions. La gestion des exceptions reste un mécanisme délicat à manipuler dans d'autres langages du fait qu'elle perturbe le déroulement normal de l'exécution des programmes. On ne la retrouvera donc pas non plus habituellement dans les approches de parallélisme structuré se basant sur d'autres langages que le C++. Calcium [78] est une des rares bibliothèques à fournir un tel mécanisme, cependant sa conception est différente : lorsque plusieurs exceptions se produisent au sein de processus différents, une seule d'entre elles sera remontée au niveau global, ce qui engendre donc un comportement non déterministe.

Le squelette **`forwardExceptions`** que nous avons mis en place au sein d'OSL nous permet donc de récupérer les erreurs lors de l'application d'autres squelettes sur des données distribuées. Ce mécanisme entraîne un surcoût dans les temps de calcul mais celui-ci reste acceptable, les temps d'exécution restant dans le même ordre de grandeur. La possibilité de récupérer globalement les exceptions apporte de nouvelles possibilités d'expression à notre bibliothèque, permettant par exemple d'implémenter une version distribuée d'un algorithme de retour sur trace avec exceptions. Plus généralement, gérer les exceptions au sein de programmes à squelettes s'exécutant sur des données distribuées dans OSL se fait de manière similaire à la gestion des exceptions au sein d'un programme séquentiel.

Les approches de parallélisme structuré offrent un avantage certain dans la gestion et la récupération des erreurs, car il est relativement simple de déterminer où en était l'exécution du programme avant l'apparition d'une erreur. Au-delà d'une simple gestion des exceptions, nous pourrions envisager dans le futur l'introduction de mécanismes de plus haut niveau afin de gérer implicitement la reprise sur erreur de l'exécution des programmes à squelettes. Un tel mécanisme pourrait être mis en place en analysant automatiquement les expressions de squelettes et en relançant leur évaluation en cas de levée d'exception. Un mécanisme plus avancé pourrait permettre de sauvegarder les résultats intermédiaires du programme afin de relancer l'exécution du programme dans le cas où un processus se termine prématurément, cependant celà demanderait la mise en place d'un environnement d'exécution en plus de notre bibliothèque. Une approche de ce type est présentée dans [81, 82], où l'exécution de programmes parallèles se trouve encadrée par un squelette permettant d'effectuer des calculs itératifs sur un domaine donné, avec sauvegarde et reprise automatique en cas d'interruption de l'exécution. Ce type d'approche structurée montre une plus grande efficacité par rapport aux approches génériques effectuant des sauvegardes et reprises aux points de contrôle au niveau des communications MPI [61], à la fois dans la taille des sauvegardes générées et dans le surcoût entraîné par leur création.

Mettre en place ce type de mécanisme nous permettrait d'améliorer la fiabilité des programmes OSL, ce qui est important car le risque d'apparition d'erreurs d'exécution augmente fortement dès lors que les calculs se retrouvent parallélisés sur de nombreux processus.

Parallélisation correcte avec l'homomorphisme BSP

7

Sommaire

7.1 Théorie de *BH* . . . 116
7.1.1 Définitions . . . 116
7.1.2 Transformation d'une fonction *mapAround* en *BH* . . . 118
7.2 Intégration à OSL . . . 119
7.2.1 Utilisation . . . 119
7.2.2 Implémentation . . . 121
7.3 Dérivation de deux problèmes . . . 122
7.3.1 Valeurs inférieures les plus proches . . . 122
7.3.2 Multiplication matrice creuse-vecteur . . . 125
7.3.3 Mesures expérimentales . . . 127
7.4 Comparaison avec un algorithme de référence . . . 129
7.4.1 Description . . . 129
7.4.2 Performances . . . 132
7.4.3 Effort de programmation . . . 133
7.5 Conclusion . . . 134

Une partie des travaux présentés dans ce chapitre a été publiée dans les actes de la conférence Euro-Par 2013 [73].

La théorie des listes [17] est un puissant outil donnant la possibilité de développer de manière systématique des programmes parallèles corrects. À partir d'une spécification ou d'une implémentation naïve d'un programme, celle-ci nous permet d'extraire une version plus efficace en la dérivant pas-à-pas. Les squelettes algorithmiques en conjonction avec les homomorphismes de listes (auxquels nous référons par la suite en tant qu'homomorphismes pour plus de simplicité) jouent un rôle important dans le développement formel d'algorithmes parallèles [35, 57, 89].

Une notion de fonction similaire aux homomorphismes et dédiée au modèle BSP a été définie et sa théorie explorée [54, 107] dans le contexte de l'assistant de preuve

Coq [110]. De par sa nature, OSL permet d'intégrer facilement un squelette effectuant la résolution de problèmes décrits sous la forme d'un homomorphisme BSP.

7.1 Théorie de BH

7.1.1 Définitions

Nos notations se basent essentiellement sur le langage de programmation fonctionnel Haskell [94]. L'application d'une fonction se note par un espace, et les arguments peuvent être présents sans parenthèses distinctives ; $f\ a$ a donc la signification de $f(a)$. Les fonctions sont currifiées, c'est-à-dire qu'elles ne prennent qu'un seul argument et retournent une fonction appliquée partiellement ou une valeur, et l'application de fonction est associative à gauche ; $f\ a\ b$ signifie donc $(f\ a)\ b$. Les opérateurs binaires infixes apparaîtront généralement sous la forme de $\oplus$, $\otimes$ ou $\odot$. L'application de fonction est prioritaire sur tous les autres opérateurs, donc $f\ a \oplus b$ signifiera $(f\ a) \oplus b$, et non pas $f(a \oplus b)$.

Les listes sont des séquences finies de valeurs du même type. Une liste peut être soit vide, soit un singleton, soit la concaténation de deux listes. Nous notons $[]$ la liste vide, $[a]$ pour un singleton de l'élément a, et $x +\!\!+ y$ la concaténation de deux listes x et y. L'opérateur de concaténation est associatif, et les listes sont détruites par filtrage de motifs (*pattern matching*).

Définition 7.1.1 (Homomorphisme) La fonction h est un *homomorphisme*, si définie récursivement sous la forme

$$h\ [\,] = id_{\odot} \qquad h\ [a] = f\ a \qquad h\ (x +\!\!+ y) = h(x) \odot h(y)$$

où $id_{\odot}$ désigne l'élément neutre de $\odot$. h étant déterminé de manière unique par f et $\odot$, nous écrirons $h = (\![\odot, f]\!)$.

Définition 7.1.2 (Homomorphisme BSP (BH)) Étant donnés une fonction k, et deux homomorphismes $g_1 = (\![\oplus, f_1]\!)$, $g_2 = (\![\otimes, f_2]\!)$ [1], h sera un homomorphisme BSP s'il peut se définir sous la forme suivante :

$$\begin{cases} h\ []\ l\ r = [] \\ h\ [a]\ l\ r = [k\ a\ l\ r] \\ h\ (x +\!\!+ y)\ l\ r = h\ x\ l\ (g_1\ y \oplus r)\ +\!\!+\ h\ y\ (l \otimes g_2\ x)\ r \end{cases}$$

h défini ci-dessus par les fonctions k, g_1, g_2, et les opérateurs associatifs $\oplus$ et $\otimes$ sera noté sous la forme $h = BH(k, (\![\oplus, f_1]\!), (\![\otimes, f_2]\!))$

1. Voir [107] pour une discussion sur des critères plus faibles pour la définition d'homomorphismes BSP

La fonctions h est un homomorphisme *d'ordre supérieur* s'appliquant à une liste et retournant une nouvelle liste de la même longueur. En plus de la liste d'entrée, h demande deux paramètres supplémentaires l et r qui apportent les informations nécessaires pour effectuer les calculs en bout de liste. Les informations contenues dans l et r, comme défini dans la troisième équation, sont propagées depuis la gauche et la droite respectivement par les fonctions $g_2, \otimes$ et $g_1, \oplus$.

La figure 7.1 (reprise de [107]) illustre l'application d'un *BH* pour un seul élément d'une liste. L'élément résultant est obtenu par l'application de la fonction k à l'élément e de la liste d'origine et aux résumés des sous-listes à gauche et à droite de cet élément. Ces résumés sont obtenus en appliquant l'homomorphisme g_1 à la liste des éléments précédant e et l'élément additionnel l et respectivement en appliquant g_2 à la liste des éléments suivant e et l'élément r.

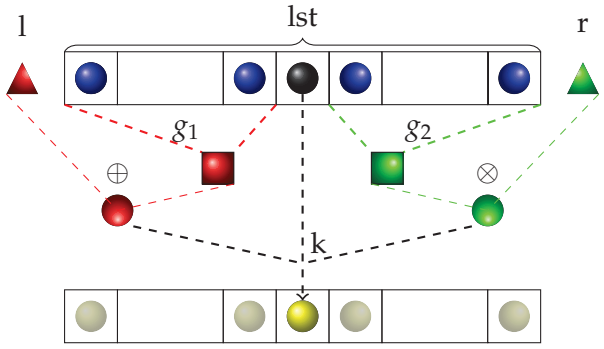


FIGURE 7.1 – *BH local*

Par définition, un *BH* peut être appliqué en parallèle étant donné qu'il est la composition de calculs locaux et d'homomorphismes, ces derniers étant facilement parallélisables [35]. La figure 7.2 (également reprise de [107]) illustre le comportement de *BH* appliqué à une liste entière distribuée sur plusieurs processus. L'application de *BH* sur un élément donné nécessitant les résumés calculés à partir de tous les éléments restants de la liste, le première étape du calcul de *BH* en parallèle consistera donc en le calcul des résumés gauche et droit sur les éléments locaux à chaque processus par l'application de g_1 et g_2. Ces résumés locaux sont ensuite échangés entre les processus, de sorte que chaque processus possède tous les résumés par g_1 des processus le précédant, et tous les résumés par g_2 des processus le suivant.

Chaque processus va ensuite réduire l'ensemble des résumés des processus précédents (respectivement suivants) et l'élément additionnel l (resp. r) en un seul élément en leur appliquant à nouveau g_1 (resp. g_2), générant une nouvelle valeur qui remplacera l (resp. r) dans les calculs locaux qui suivront. La contrainte selon laquelle g_1 et g_2 doivent être des homomorphismes vient de cette étape du calcul. En effet, dans la sémantique locale les fonctions générant les résumés pourraient potentiellement être n'importe quel type de fonction générant une unique valeur à partir d'une liste d'éléments, cependant le fait de pouvoir diviser le calcul d'un résumé en appliquant

la fonction au résultat de son application sur un ensemble de sous-listes correspond précisément à la définition d'un homomorphisme.

La dernière étape consiste finalement en l'application du calcul local décrit précédemment, en utilisant les résumés globaux obtenus après les calculs précédents comme éléments l et r. Le lien entre BH et le modèle BSP devient explicite à la vue de ce schéma : dans ce calcul nous avons une première phase de calculs locaux, suivie d'une phase de communication globale, puis d'une dernière phase de calculs locaux, ce schéma peut donc directement s'exprimer sous la forme de super-étapes BSP.

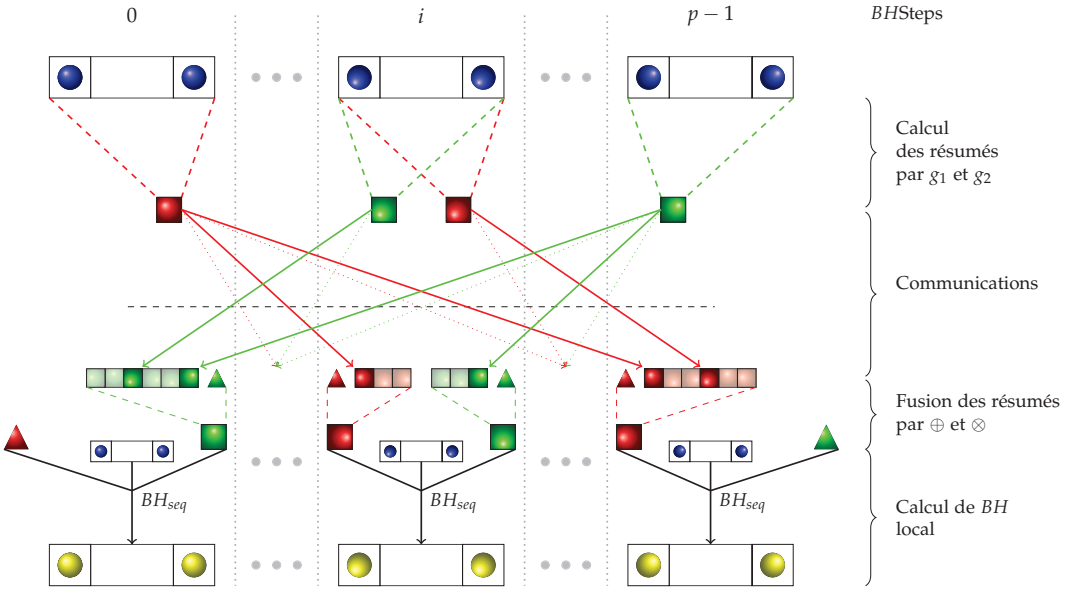


FIGURE 7.2 – *Calcul de BH en parallèle*

7.1.2 Transformation d'une fonction *mapAround* en BH

Les homomorphismes BSP sont très puissants car ils permettent de paralléliser un calcul de manière correcte, cependant il est difficile de transformer la spécification d'un programme en un BH. Au lieu de vérifier directement qu'une fonction donnée puisse être exprimée sous la forme d'un BH, nous pouvons utiliser une représentation intermédiaire beaucoup plus facile à manipuler intuitivement faisant appel au squelette *mapAround*.

Cette fonction proche de *map* exprime elle aussi des calculs indépendants sur les éléments d'une liste, mais permet d'exprimer des calculs plus complexes faisant intervenir l'ensemble des éléments de la liste. Intuitivement, *mapAround* applique une fonction séparément pour chacun des éléments d'une liste, mais cette fonction pourra extraire des informations des sous-listes à gauche et à droite de l'élément considéré

afin d'effectuer son calcul, autrement dit :

$$\begin{aligned} & \mathit{mapAround}\ f\ [x_1, x_2, \ldots, x_n] \\ = \ & [f\ ([], x_1, [x_2, \ldots, x_n]), f\ ([x_1], x_2, [x_3, \ldots, x_n]), \ldots, f\ ([x_1, \ldots, x_{n-1}], x_n, [])]. \end{aligned}$$

mapAround a un schéma d'application proche de BH, vu que pour chaque élément ce squelette applique une fonction sur ce dernier et les sous-listes à sa gauche et à sa droite. En imposant certaines restrictions sur la fonctions appliquée par *mapAround*, il est en fait possible d'obtenir une équivalence avec un BH.

Théorème 7.1.1 (Parallélisation de *mapAround* par BH) Étant donnée une fonction $h = \mathit{mapAround}\ f$, si nous pouvons décomposer f sous la forme $f\ (ls, x, rs) = k\ (g_1\ ls, x, g_2\ rs)$ où g_i est la composition d'une fonction p_i avec un homomorphisme, $g_i\ x = p_i([\![\oplus_i, k_i]\!]\ x)$, alors

$$h\ xs = BH(k', ([\![\oplus_1, k_1]\!]), ([\![\oplus_2, k_2]\!]))\ xs\ \iota_{\oplus_1}\ \iota_{\oplus_2}$$

où $k'\ (l, x, r) = k(p_1\ l, x, p_2\ r)$, $\iota_{\oplus_1}$ est le neutre (à gauche) de $\oplus_1$ et $\iota_{\oplus_2}$ est le neutre (à droite) de $\oplus_2$.

Démonstration. Ceci peut être prouvé par induction en raisonnant sur la liste d'entrée de h. La preuve détaillée en Coq est détaillée dans [54, 107]. □

La théorème 7.1.1 est général et puissant dans le sens où il permet de paralléliser non seulement *mapAround*, mais également d'autres fonctions collectives telles que *scan* au travers de BH [54, 107].

7.2 Intégration à OSL

7.2.1 Utilisation

Le calcul d'une fonction exprimée sous la forme d'un BH a été implémenté dans la bibliothèque OSL sous la forme d'un squelette.

La signature du squelette `bh` est :

```
DArray<typename K::result_type>
   bh(K k,
      Homomorphism<T, L> * hl,
      Homomorphism<T, R> * hr,
      L l,
      R r,
      const DArray<T>& temp);
```

D'après la définition 7.1.2, un *BH* est défini par une fonction `k` et deux homomorphismes `g1` et `g2`, et s'applique à une liste (ici sous la forme d'un tableau distribué `temp`) accompagné de deux éléments aux extrémités `L` et `R`.

`k` s'implémente simplement sous la forme habituelle d'un foncteur dont l'opérateur `()` prend trois arguments : le résumé gauche (qui sera le résultat de l'application de `g1` sur la partie gauche de la liste), l'élément courant et le résumé droit. Pour `g1` et `g2`, nous définissons une classe de base générique virtuelle `Homomorphism` spécifiant les nécessaires fonctions f, opérateurs $\odot$ et son élément neutre $id_{\odot}$ (Définition 7.1.1). L'utilisateur peut ensuite implémenter ses propres homomorphismes en créant une classe dérivée fournissant des implémentations concrètes pour ces trois éléments.

`k`, `g1` et `g2` sont les trois premiers paramètres de notre squelette *BH* générique qui vont déterminer les calculs qu'il va appliquer. Pour définir les données précises auxquelles ils s'appliqueront, il nous faut passer trois arguments : les éléments aux extrémités `L` et `R`, et la liste sous la forme d'un `DArray`. La valeur de retour du squelette sera un tableau distribué de la même taille dont le type des éléments sera la type résultant de l'application de `k`.

Nous pouvons ainsi facilement implémenter par exemple le calcul de la somme préfixe sur un tableau d'entiers. Premièrement, nous définissons un homomorphisme gauche qui effectue la somme des valeurs :

```
class HAdd: public Homomorphism<int, int> {
   public:
      HAdd() { neutral = 0;}
      inline int f(const int& i) {
         return i;
      }
      inline int o(const int& i1, const int& i2) {
         return i1+i2;
      }
};
```

Deuxièmement, nous n'avons aucun calcul à effectuer du côté droit, cependant nous devons tout de même fournir un homomorphisme au squelette `bh`. Nous définissons donc un homomorphisme constant renvoyant toujours la même valeur. Cet homomorphisme nommé `HConst` est défini de manière similaire à `HAdd`, mais chaque opérateur ne fera que renvoyer le nombre 0 :

```
class HConstant: public Homomorphism<int, int> {
   public:
      HConst() { neutral = 0;}
      inline int f(const int& i) {
         return 0;
```

```
      }
      inline int o(const int& i1, const int& i2) {
         return 0;
      }
};
```

Il nous reste à définir la fonction `k` qui va simplement ajouter la somme du sous-tableau de gauche à l'élément courant :

```
struct AddLeft {
   typedef int result_type;
   inline int operator()(int l, int i, int r) const {
      return l+i;
   }
};
```

Nous pouvons maintenant appliquer le squelette avec ces diverses fonctions pour calculer la somme préfixe de n'importe quel tableau distribué `d` en utilisant des zéros comme valeurs aux extrémités :

```
DArray<int> result =
   osl::bh(AddLeft(),new HAdd(), new HConst(), 0, 0, d);
```

7.2.2 Implémentation

Le squelette `bh` est implémenté en utilisant le mécanisme d'*expression templates* mis en place par la bibliothèque, de sorte qu'il s'intègre de façon transparente au sein de n'importe quelle expression OSL et puisse déclencher le mécanisme de fusion de boucles lorsqu'il y a lieu. La définition récursive des homomorphismes nous donne la possibilité de mettre en place une importante optimisation du calcul : si nous appliquons cette définition à un tableau d'éléments, la troisième règle récursive peut s'écrire comme suit :

$$h\ [x_1, \ldots, x_n] = h\ [x_1, \ldots, x_n - 1] \odot h\ [x_n] = h\ [x_1, \ldots, x_n - 1] \odot f\ (x_n).$$

Ceci nous permet donc de précalculer localement l'application de l'homomorphisme à chaque sous-tableau dans un temps linéaire car nous n'avons à appliquer f et $\odot$ qu'une seule fois par élément. Sans cette optimisation, il faudrait appliquer ces opérations i fois pour chacun des n x_i éléments, ce qui résulterait en une complexité quadratique. Les homomorphismes étant associatifs, nous pouvons de manière symétrique mettre en place la même optimisation pour l'homomorphisme droit qui s'applique en partant de la fin du tableau.

Un désavantage est que pour mettre en place cette optimisation, il nous faut considérer la part locale du tableau distribué dans son entièreté pour chacun des noeuds ; ceci nous oblige à suspendre le mécanisme de fusion qui se base sur le fait que chaque élément du tableau est traité séparément. Cependant la fusion peut toujours avoir lieu dans l'expression (si existante) produisant le tableau en entrée.

7.3 Dérivation de deux problèmes

Nous illustrons dans cette section la dérivation d'applications en utilisant la théorie *BH* à travers deux exemples non triviaux. Le premier est la recherche des valeurs inférieures les plus proches, et le second la multiplication d'une matrice creuse par un vecteur.

7.3.1 Valeurs inférieures les plus proches

Le problèmes des valeurs inférieures les plus proches (*ANSV : All Nearest Smaller Values*) se décrit comme suit :

> Étant donnée $xs = [x_1; x_2; \ldots; x_n]$ une liste d'éléments appartenant à un domaine totalement ordonné. Pour chaque x_j, on cherche l'élément inférieur à x_j le plus proche à sa gauche et celui le plus proche à sa droite. Si un tel élément n'existe pas, nous définissons la valeur $\perp$ comme solution.

Le calcul des ANSV pour un élément donné d'une liste est illustré dans la figure 7.3. Un exemple d'entrée et de sortie de la fonction *ansv* résolvant ce problème sur une liste entière pourrait être :

$$\begin{aligned} &ansv\ [3,1,4,1,5,9,2] \\ &= [(\perp,1),(\perp,\perp),(1,1),(\perp,\perp),(1,2),(5,2),(1,\perp)] \end{aligned}$$

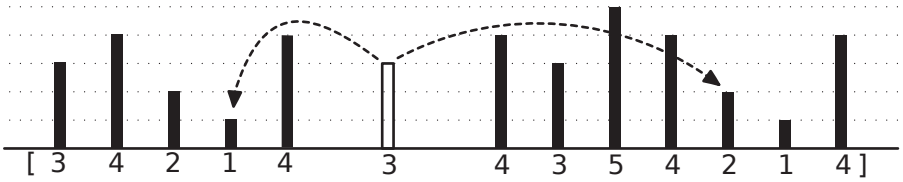


Figure 7.3 – *Calcul des ANSV pour la valeur centrale 3 (en blanc), les flèches pointent vers les deux solutions.*

Afin de pouvoir donner une spécification de *ansv* sous la forme d'une application de la fonction *mapAround*, considérons le cas d'un élément x de la liste, associé à la liste

de ses voisins de gauche l et à la liste de ses voisins de droite r. La valeur inférieure la plus proche à gauche pourrait être calculée par :

$$\begin{aligned} nsvL\ x\ [\,] &= \bot \\ nsvL\ x\ (ls +\!\!+ [l]) &= \textbf{if}\ l < x\ \textbf{then}\ l\ \textbf{else}\ nsvL\ x\ ls \end{aligned}$$

Ceci est une formalisation directe du comportement intuitif « en partant de x et en allant de la droite vers la gauche, examiner chaque valeur jusqu'à en trouver une qui soit inférieure à x ». Similairement, calculer la valeur inférieure la plus proche à droite peut se calculer par :

$$\begin{aligned} nsvR\ x\ [\,] &= \bot \\ nsvR\ x\ ([r] +\!\!+ rs) &= \textbf{if}\ r < x\ \textbf{then}\ r\ \textbf{else}\ nsvR\ x\ rs \end{aligned}$$

Une spécification directe du problème sous la forme d'une application de *mapAround* pourrait donc être :

$$\begin{aligned} &ansv\ xs = mapAround\ nsv\ xs \\ &\quad \textbf{where}\ nsv\ (ls, x, rs) = (nsvL\ x\ ls, nsvR\ x\ rs) \end{aligned}$$

Cependant, pour pouvoir appliquer le théorème 7.1.1, les calculs sur les listes gauche et droite doivent s'exprimer sous la forme d'un homomorphisme indépendant de l'élément central, ce qui n'est pas le cas car *nvsL* et *nsvR* s'appliquent à l'élément central x en plus des sous-listes ls et rs. Nous raffinons donc nos définitions, et décidons en premier lieu d'appliquer une fonction sélectionnant les *candidats* des listes gauches et droites, puis en déterminant parmi ces candidats la solution correcte pour l'élément courant. De par la symétrie du problème entre la gauche et la droite, nous nous limiterons à traiter le cas de droite.

Étant donné que nous cherchons les valeurs inférieures les plus proches, nous pouvons éliminer en toute sécurité les éléments égaux ou supérieurs à un candidat précédent, et donc ne retenir qu'un ensemble de candidats de valeurs décroissantes. Par exemple, si la sous-liste à droite d'un élément est $[4;3;5;4;2;1;4]$ nous savons que 5, le second et le troisième 4 ne peuvent être des solutions car ils sont précédés de valeurs inférieures à eux-mêmes ; la liste des candidats potentiels est donc $[4;3;2;1]$ 7.4.

À partir de la liste des candidats, nous obtenons facilement la valeur inférieure la plus proche par la fonction suivante :

$$\begin{aligned} pickupR\ x\ [\,] &= \bot \\ pickupR\ x\ ([r] +\!\!+ rs) &= \textbf{if}\ r < x\ \textbf{then}\ r \\ &\qquad \textbf{else}\ pickupR\ x\ rs \end{aligned}$$

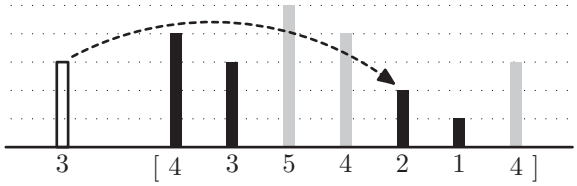


FIGURE 7.4 – *Les candidats d'une sous-liste à droite. Les valeurs se succédant en conservant un ordre décroissant sont conservées comme candidats (en noir) tandis que les autres valeurs sont éliminées (en gris).*

Il nous reste à définir le calcul des candidats sous la forme d'un homomorphisme que nous nommons $candidateR = (\![\oplus_r, f]\!)$. Cet homomorphisme produit une liste de valeurs, il faut donc que les arguments de l'opérateur $\oplus_r$ soient également des listes. Étant donné que nous souhaitons appliquer *candidateR* à une liste de valeurs, nous devons transformer chaque élément de cette liste en une liste singleton, ce qui entraîne donc que la fonction f est définie de sorte que $f\ x = [x]$. Nous nommerons cette fonction particulière *singleton*.

Nous devons maintenant définir comment l'opérateur $\oplus_r$ combine les listes de candidats. Cet opérateur devant être associatif, il doit pouvoir s'appliquer à la liste des singletons de gauche à droite et inversement. Dans le premier cas, le premier argument de $\oplus_r$ est la liste des candidats calculée jusque là, et le second argument est une liste singleton. Dans le second cas le premier argument est une liste singleton et le second argument est la liste des candidats calculés jusque là, mais en partant de la droite.

Le seul cas où le premier argument pourrait être la liste vide est donc lors du parcours de la gauche vers la droite, quand la liste de candidats est vide. Le second argument sera donc le premier candidat dans ce cas. Ceci nous donne donc la première équation définissant $\oplus_r$:

$$[]\ \oplus_r\ ys = ys \tag{7.1}$$

Considérons maintenant la seconde équation dans le cas où le premier argument est non vide. Dans le point de vue « gauche-à-droite » nous nous intéressons au dernier élément de la liste, c'est-à-dire le plus petit candidat. Dans le cas droite-à-gauche il n'y a qu'un seul élément (qui est donc le dernier) : l'élément considéré pour l'ajout à la liste de candidats. Nous appelons cet élément x dans la suite du raisonnement.

Dans le point de vue « droite-à-gauche », nous avons une liste de candidats ys construite de la droite vers la gauche. Cette liste de candidats est triée dans un ordre décroissant, nous n'avons donc à conserver que la partie contenant des éléments inférieurs à x :

$$\begin{aligned} keep\ x\ [\,] &= [\,] \\ keep\ x\ ([y] +\!\!+\ ys) &= \textbf{if}\ y < x\ \textbf{then}\ [y] +\!\!+\ ys \\ &\quad\ \ \textbf{else}\ keep\ x\ ys \end{aligned}$$

Dans le point de vue « gauche-à-droite », ys est une liste singleton $[y]$. Si x est plus grand que y, nous devons ajouter s à la liste de candidats, sinon cette dernière reste inchangée. Ceci est un cas particulier du raisonnement de droite à gauche, où nous conservons dans $[y]$ les éléments inférieurs à x. Ceci conclut le raisonnement, et la seconde équation est donc :

$$(xs \;+\!\!+\; [x]) \;\oplus_r\; ys = xs \;+\!\!+\; [x] \;+\!\!+\; (keep\ x\ ys) \tag{7.2}$$

Par exemple, dans la vue de gauche à droite, si la liste des candidats actuelle est $[8;4]$ et la liste singleton $[7]$:

$$[8;4] \;\oplus_r\; [7] = [8;4].$$

Dans la vue de droite à gauche, si la liste singleton est $[4]$ et la liste de candidats actuelle est $[5;4;1]$:

$$[4] \;\oplus_r\; [5;4;1] = [4;1].$$

De manière symétrique, nous pouvons définir la fonction *pickupL* et l'homomorphisme $candidateL = ([\oplus_l, f])$. Nous pouvons finalement réécrire *nsv* comme suit :

$$\begin{aligned} nsv\ (ls, x, rs) &= k\ ([\oplus_l, f])\ ls, x, ([\oplus_r, f])\ rs) \\ &\textbf{where}\ k\ (ls, x, rs)\ =\ (pickupL\ x\ ls,\ pickupR\ x\ rs) \end{aligned}$$

Cette version possède la forme demandée par les hypothèses du Théorème 7.1.1, qui peut donc être appliquée pour dériver *ansv* en un homomorphisme BSP.

7.3.2 Multiplication matrice creuse-vecteur

Les matrices creuses sont souvent compressées dans des représentations par tableau. Nous développons un programme parallèle correct effectuant la multiplication d'une matrice creuse avec un vecteur au travers de la théorie de BH.

Nous nous basons sur la représentation en tableau constitués de triplets (y, x, a) :
– y : l'indice parmi les lignes de l'élément non nul,
– x : l'indice parmi les colonnes de l'élément non nul, et
– a : la valeur de l'élément non nul.

Nous supposons les éléments triés par rapport aux indices y et x. Par exemple, la matrice A suivante est représentée par le tableau *as* contenant 5 triplets.

$$A = \begin{pmatrix} 1.1 & 2.2 & 0 \\ 0 & 1.3 & 1.4 \\ 0 & 0 & 3.5 \end{pmatrix} \qquad \begin{aligned} as = [&(0,0,1.1),\ (0,1,2.2),\\ &(1,1,1.3),\ (1,2,1.4),\ (2,2,3.5)] \end{aligned}$$

Dans la multiplication matrice-vecteur, il y a un élément par ligne dans le résultat. Nous placerons le résultat dans le premier élément de sa ligne, et effacerons les autres valeurs avec une valeur factice que nous indiquerons par $\Box$. Par exemple, multiplier la représentation en tableau *as* par un vecteur $[3.0, 4.0, 1.0]$ produira :

$$\mathit{mult\ as}\ [3.0, 4.0, 1.0] = [(0, 0, 12.1),\ (0, 1, \Box),\ (1, 1, 6.6),\ (1, 2, \Box),\ (2, 2, 3.5)]\ .$$

Il est à noter qu'il nous est possible d'appliquer la compaction de tableau [54] afin de réduire les résultats en un simple vecteur $[12.1, 6.6, 3.5]$.

Nous devons développer une spécification de ce problème utilisant la fonction *mapAround*. La première étape importante est de déterminer quelles sont les informations précises dont nous avons besoin depuis la gauche et la droite pour un élément donné. Nous devons déterminer si un élément est le premier de sa ligne afin de savoir si la somme des valeurs d'une nouvelle ligne commence et si le résultat sera écrit à cette position ; cela se fait facilement en comparant l'indice de ligne de cet élément avec le précédent.

Pour calculer la valeur du résultat, nous avons également besoin des éléments situés plus à droite et qui appartiennent à la même ligne de la matrice que l'élément actuel. Ces éléments sont multipliés par les éléments correspondant dans le vecteur, et la somme de ses résultats est effectuée. Les valeurs transmises depuis la droite seront donc l'indice de ligne de l'élément suivant, et la somme partielle des multiplications du reste de cette ligne.

Nous pouvons donc développer une spécification basée sur la fonction *mapAround* à partir de ces observations. Dans le programme suivant, $v\langle i\rangle$ désigne l'élément i du vecteur v.

$$\begin{aligned}
&\mathit{mult\ as}\ v = \mathit{mapAround}\ (f\ v)\ \mathit{as}\\
&\textbf{where}\ f\ v\ (ls, (y, x, a), rs) = \textbf{let}\ y_l = g_l\ ls; (y_r, s_r) = g_r\ v\ rs\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in if}\ (y_l == y)\ \textbf{then}\ (y, x, \Box)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{elseif}\ (y_r == y)\ \textbf{then}\ (y, x, v\langle x\rangle * a + s_r)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ (y, x, v\langle x\rangle * a)
\end{aligned}$$

Nous donnons ensuite les définitions des fonctions auxiliaires et vérifions qu'il s'agit bien d'homomorphismes. La fonction g_l renvoie simplement l'indice de ligne du dernier élément d'une liste, et est un homomorphisme

$$g_l = ([\gg, \lambda(x, y, a).y]) \quad \textbf{where}\ \ a \gg b = b\ ,$$

et n'importe quelle valeur (nous avons utilisé -1 ici) est un neutre à gauche de l'opérateur $\gg$.

La fonction $g_r\ v$ est plus complexe et définie comme suit :

$$
\begin{aligned}
g_r\ v\ [(y,x,a)] &= (y, a * v\langle x\rangle)\\
g_r\ v\ [as +\!\!+ (y,x,a)] &= \textbf{let}\ (y',s) = g_r\ v\ as\\
&\qquad\qquad \textbf{in}\ (y', \textbf{if}\ y' == y\ \textbf{then}\ s + a * v[x]\ \textit{else}\ s)
\end{aligned}
$$

Cette fonction est bien un homomorphisme :

$$
\begin{aligned}
&g_r\ v\ [(y,x,a)] = (y, a * v\langle x\rangle)\\
&g_r\ v\ (ls +\!\!+ rs) = g_r\ v\ ls \odot g_r\ v\ rs\\
&\qquad \textbf{where}\ (y_l,s_l) \odot (y_r,s_r) = \textbf{if}\ y_l == y_r\ \textbf{then}\ (y_l, s_l + s_r)\ \textbf{else}\ (y_l,s_l)
\end{aligned}
$$

Le neutre à droite de l'opérateur $\odot$ est $(-1,0)$.

Nous pouvons donc appliquer le théorème 7.1.1 à la spécification précédente pour obtenir le BH suivant :

$$
\begin{aligned}
&\mathit{mult}\ as\ v = BH(k\ v, ([\odot, \lambda(y,x,a).(y, a * v\langle x\rangle)]), ([\gg, \lambda(x,y,a).y]))\ as\\
&\quad \textbf{where}\ k\ v\ (y_l,(y,x,a),(y_r,s)) = \textbf{if}\ y == y_l\ \textbf{then}\ (y,x,\square)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{elseif}\ y == y_r\ \textbf{then}\ (y,x,a * v\langle x\rangle + s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ (y,x,a * v\langle x\rangle)\\
&\qquad a \gg b = b\\
&\qquad (y_l,s_l) \odot (y_r,s_r) = \textbf{if}\ y_l == y_r\ \textbf{then}\ (y_l, s_l + s_r)\ \textbf{else}\ (y_l,s_l)
\end{aligned}
$$

7.3.3 Mesures expérimentales

Nous avons mis en place les programmes décrits précédemment calculant les valeurs inférieures les plus proches et la multiplication matrice creuse-vecteur en utilisant notre implémentation du squelette BH dans la librairie OSL. Dans un premier temps, nous avons mesuré les performances obtenues par leur parallélisation sur plusieurs cœurs de calcul sur la machine à mémoire partagée SPEED (figure 7.6) ainsi que sur la grappe Mirev (figure 7.5). Nous avons à nouveau mis en place le protocole de test utilisé dans les chapitres précédents (cf. Chapitre 2.4.1.5), à savoir que le programme a été exécuté 30 fois de suite pour chacune des quantités de coeurs utilisées et la médiane de chaque ensemble de temps d'exécution est utilisée pour mesurer le gain de performance. Les résultats sont validés par un test statistique afin de déterminer si le taux de confiance dans la reproductibilité des mesures est suffisamment élevé, ce dernier a été fixé à un minimum de 95% pour tous les résultats présentés ici.

Le calcul des valeurs inférieures les plus proches a été effectué sur un tableau contenant 10^7 éléments initialisés aléatoirement. La multiplication d'une matrice creuse avec un vecteur a été effectuée sur une matrice aléatoire de 10^9 éléments parmi lesquels 10% sont non nuls, constituant donc un ensemble de 10^8 éléments de données concrètes.

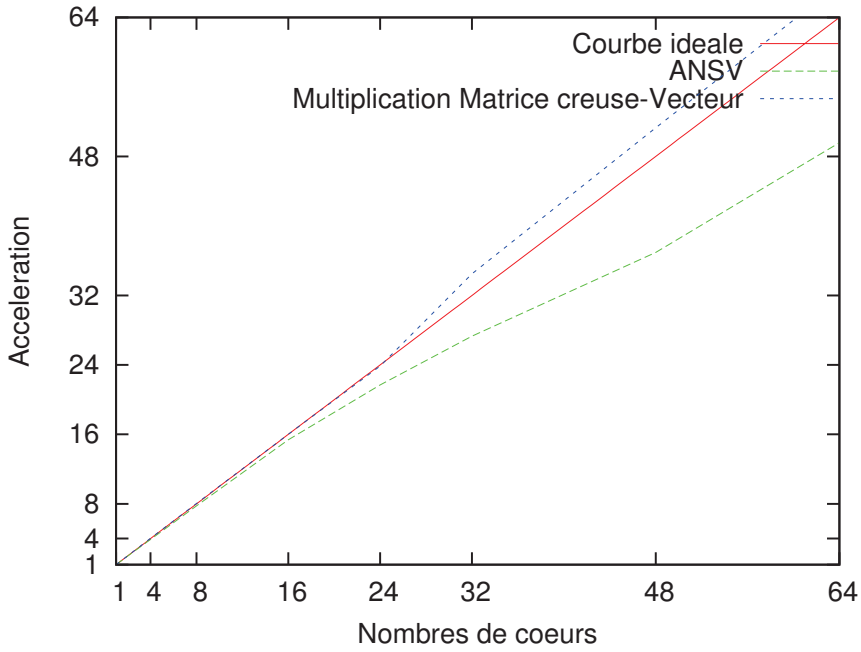


FIGURE 7.5 – *Parallélisation en mémoire distribuée*

Le problème des valeurs inférieures les plus proches passe assez bien à l'échelle bien que de manière sous-linéaire, nous pouvons nous attendre à ce que sa performance plafonne sur un grand nombre de cœurs. Ceci peut s'expliquer par le fait que chaque processus doit communiquer ses deux tableaux locaux de candidats à tous les autres processus, le nombre de tableaux transmis augmente donc de manière quadratique par rapport au nombre de processus, alors que la taille de ces tableaux de candidats n'a pas de lien direct avec le nombre de processus (on peut s'attendre à ce qu'elle varie peu sur de très grands tableaux). Le coût de la phase de communication devrait donc finir par surpasser le gain obtenu par la parallélisation du calcul sur un grand nombre de cœurs.

La multiplication matrice creuse-vecteur quant à elle se parallélise de façon parfaitement linéaire. Dans ce problème les processus n'ont qu'une très faible quantité de données à s'échanger, les résumés calculés consistant en seulement deux nombres, les coûts de communication ont donc un impact négligeable à cette échelle. Nous obtenons également un gain super-linéaire sur l'architecture à mémoire distribuée dès lors que plusieurs dizaines de coeurs sont utilisés. Ceci semblerait indiquer que ce calcul particulier soit limité par les débits d'accès à la mémoire sur la machine SPEED à mémoire partagée.

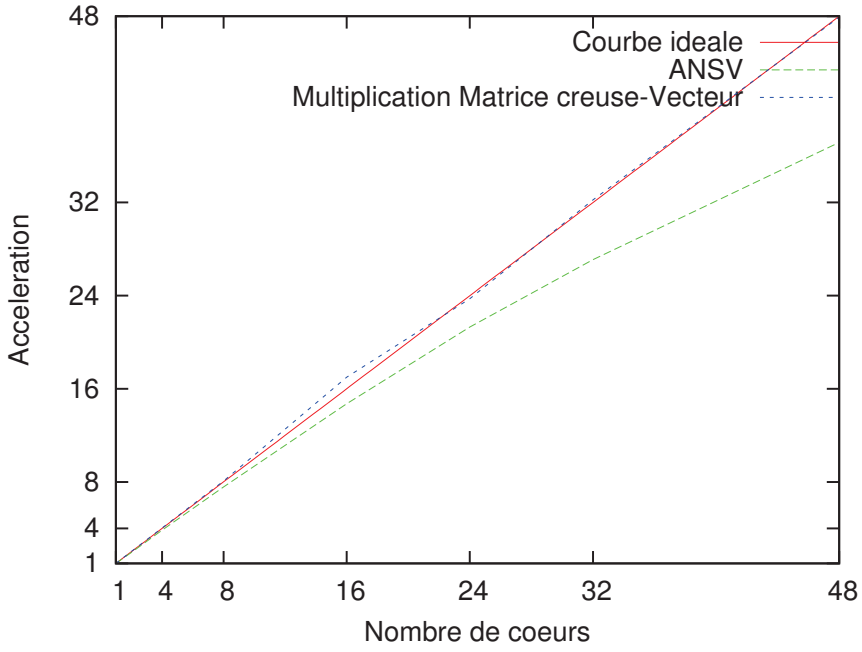


FIGURE 7.6 – *Parallélisation en mémoire partagée*

Dans l'ensemble, il y a assez peu de différences de performance entre les deux architectures. La différence majeure entre les deux architecture se situe dans le débit des communications, qui sera naturellement beaucoup plus faible sur Mirev. La proximité de performances entre les deux architectures signifie donc que la résolution de ces problème en parallèle à travers *BH* demande principalement des calculs locaux et proportionnellement peu de communications. Ceci se montre cohérent avec le modèle de calcul de *BH*, en effet dans celui-ci nous avons besoin d'appliquer des calculs sur la totalité des éléments du tableau considéré, mais nous ne communiquerons que les résumés obtenus qui seront à priori de taille très inférieure.

7.4 Comparaison avec un algorithme de référence

7.4.1 Description

Nous souhaitons dans cette section comparer l'implémentation d'un problème à travers *BH* avec une implémentation parallèle de référence. Une implémentation de l'algorithme résolvant le problème des ANSV basée sur le modèle BSP a été publiée par He et Huang en 2001 [62]. Cet algorithme effectue une résolution locale sur chaque processus, puis effectue une analyse des minima locaux pour déterminer des ensembles optimaux d'éléments devant être échangés afin de résoudre complètement

le problème sur la globalité des éléments. Le déroulement de cet algorithme sur un exemple concret est illustré dans l'annexe B

La première étape consiste donc à calculer les solutions d'ANSV sur les éléments locaux à chaque processus, pour ce faire nous pouvons par exemple utiliser l'approche classique effectuant ce calcul grâce à une pile :

```
for (int i = 0; i < input.size(); i++) {
   while ( (!candidates.empty()) &&
            (candidates.top() >= input[i]) )
      candidates.pop();
   if (candidates.empty())
      solutions[i] = INT_MIN;
   else
      solutions[i] = candidates.top();
   candidates.push(input[i]);
}
```

Intuitivement, cette approche parcourt l'ensemble des éléments un par un, et empile l'élément courant à chaque itération. La solution pour un élément donné se trouve dans la pile au moment où l'élément sera traité, on dépile donc tous les éléments plus grands que l'élément courant. Si la pile est vide après ce dépilage, il n'y avait donc aucun élément inférieur à la valeur courante qui ne possède donc pas de valeur inférieure plus proche, sinon l'élément au sommet de la pile restante sera la solution. Cet algorithme est très efficace car chacune des valeurs ne sera au total lue, empilée et dépilée qu'une seul fois, sa complexité est donc linéaire. Cet algorithme nous donne les valeurs inférieures les plus proches à gauche, pour obtenir les valeurs à droite il faudra à nouveau effectuer le même calcul mais en itérant à l'envers à partir de la fin du tableau.

L'application de cet algorithme sur les éléments locaux à chaque processus nous donne une solution partielle, nous sommes entre autres certains que dans un sous-tableau donné tous les éléments précédant sa valeur minimum posséderont une solution droite locale qui sera au plus loin le minimum en question, et symétriquement les éléments suivant le minimum posséderont une solution gauche qui sera au plus loin le minimum. L'algorithme va ensuite analyser les minima de chaque processus afin de déterminer les sous-ensembles d'éléments situés sur d'autres processus contenant les valeurs nécessaires au calcul des solutions globales.

Les processus s'échangent leurs minima et calculent les solutions de ANSV sur le tableau des minima A_{min}, ceci conclut la première super-étape BSP du programme. Si le minimum min_i du processus i possède comme solution à droite le minimum min_j du noeud j, nous pouvons définir deux sous-ensembles : $Seq1i$ au sein de i et $Seq2i$ au sein de j, et nous pourrons trouver les solutions à droite manquantes pour $Seq1i$ dans

*Seq*2*i*, et symétriquement *Seq*1*i* contiendra les solutions à gauche manquantes dans *Seq*2*i*.

*Seq*1*i* commence à l'élément min_i et se termine au dernier élément de i à condition que $j = i + 1$. Si ce n'est pas le cas, nous devons déterminer l'unique processus k tel que son minimum local min_k ait min_i comme solution (au sein de l'ensemble A_{min}) à gauche, et min_j comme solution à droite. *Seq*1*i* se finira alors sur l'élément solution gauche de min_k dans i. *Seq*2*i* se défini de façon symétrique : son élément de fin est solution droite de min_i dans j, et son premier élément est soit le premier élément de j si il est le voisin de i, soit l'élément qui sera la solution droite de min_k dans j (cf. Figure 7.7).

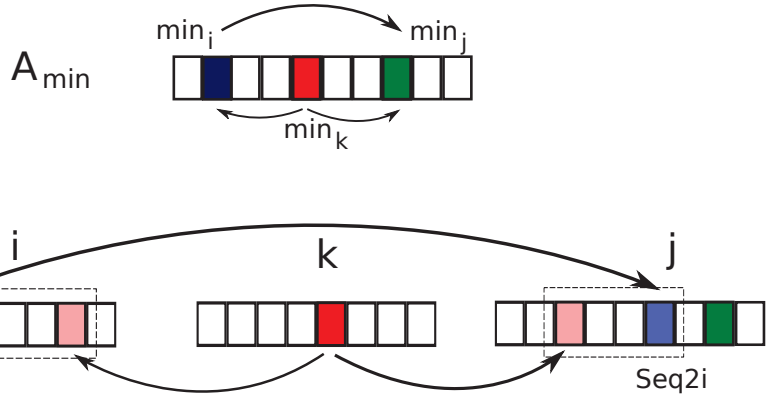


FIGURE 7.7 – *Calcul des séquences Seq1i et Seq2j. Les flèches représentent la relation « valeur inférieure la plus proche » dans la direction correspondante. Le premier tableau ne contient que les minima, le tableaux du bas représentent la totalité des données sur les processus i, j et k.*

Symétriquement, il existe deux sous-séquences *Seq*3*i* sur le processus i et *Seq*4*i* sur le processus j' si $min_{j'}$ est la solution gauche de min_i au sein de l'ensemble A_{min}. Les solutions droites manquantes dans *Seq*4*i* seront trouvées au sein de *Seq*3*i*, et les *Seq*3*i* trouveront leurs solutions gauches manquantes dans *Seq*4*i*. Ces deux séquences se construisent de manière totalement symétrique à *Seq*1*i* et *Seq*2*i*, c'est à dire en remplaçant « solution gauche » par « solution droite » et vice-versa dans le calcul des extrémités de ces ensembles décrits plus haut.

La seconde super-étape BSP du programme consistera donc à calculer sur chaque noeud i les deux ensembles *Seq*1*i* et *Seq*3*i* s'ils existent, ainsi que les éventuels *Seq*2*x* et *Seq*4*y* (qui peuvent être plusieurs sur le noeuds i, car min_i peut être une solution droite pour plusieurs min_x ou une solution gauche pour plusieurs min_y). L'ensemble des séquences *Seq*2*x* et *Seq*4*y* calculées sur chaque processus est ensuite envoyé aux noeuds x et y correspondants.

Au début de la dernière super-étape BSP, chaque processus i possède donc localement tous les sous-ensembles *Seq*1*i* à *Seq*4*i* nécessaires au calcul des solutions manquantes. Nous pouvons donc calculer ANSV localement sur la concaténation de *Seq*1*i* avec *Seq*2*i* et la concaténation de *Seq*4*i* avec *Seq*3*i*. Nous obtenons donc locale-

version / np	1	2	4	8	16	32
OSL	39.07 s	17.54 s	9.99 s	6.21 s	4.43 s	3.67 s
HH	0.47 s	0.56 s	0.31 s	0.34 s	0.20 s	0.12 s

TABLE 7.1 – *Comparaison des deux algorithmes ANSV*

ment les solutions droites manquantes pour $Seq1i$ et les solutions gauches manquantes pour $Seq3i$. $Seq2i$ contiendra également des nouvelles solutions gauches pour ses éléments, celles-ci sont donc renvoyées à son processus d'origine, de même pour $Seq4i$ qui contiendra de nouvelles solutions droites pour son processus d'origine. Après avoir résolu ANSV pour toutes les concaténations de sous-ensembles existants et avoir retransmis les solutions aux processus distants, nous aurons sur chaque noeud la totalité des solutions qui n'avaient pu être obtenues lors du premier calcul local.

7.4.2 Performances

Nous comparons les performances obtenues par cet algorithme et celles obtenues par notre implémentation utilisant le squelette BH dans OSL. Il est facilement prévisible que l'implémentation en BH soit nettement moins performante : créer et parcourir les tableaux de candidats demandera plus d'opérations que l'approche directe de la pile, et nous devrons échanger entre tous les noeuds les tableaux de candidats générés alors que l'algorithme de He et Huang optimise les communications en calculant un ensemble minimal d'éléments devant être échangés entre les processus.

Les mesures ont été effectuées dans les mêmes conditions que les mesures sur les programmes utilisant BH (cf. 7.3.3), sur la machine à mémoire partagée SPEED. Cependant, à cause de la contrainte imposée par le second algorithme de ne pas avoir de doublons dans le jeu de données nous n'avons pas utilisé de données aléatoires (car il devient difficile de générer de grands tableaux aléatoires sans doublons). A la place, nous avons créé des tableaux contenant l'ensemble des nombres de 1 à N (N étant la taille du tableau) en les distribuant dans un ordre aléatoire. Il est à noter que les résultats obtenus par l'implémentation utilisant BH sur ces ensembles particuliers sont beaucoup moins bons que sur les ensemble aléatoires, et se parallélisent également moins bien.

Comme nous nous y attendions, le second algorithme se montre très largement plus performant, cependant il présente plusieurs désavantages. Premièrement, celui-ci présuppose que l'ensemble de valeurs à traiter ne comporte aucun doublons. Cette condition est importante notamment pour le calcul des sous-ensembles $SeqNi$: l'existence de doublons au sein de l'ensemble des minima locaux peut nous amener à une situation où il n'existe pas de valeur min_k satisfaisant aux conditions demandées pour la construction d'un sous-ensemble. L'ensemble des minima pour un même ensemble

de valeurs variant selon le nombre de processus utilisés, la seule façon de garantir une exécution correcte est donc d'exiger l'unicité de chacune des valeurs. Cet algorithme ne peut donc pas s'appliquer à tous les cas, contrainte que n'a pas la version utilisant *BH*. À l'aide de l'assistant de preuve Coq, une *implantation* en BSML de *BH* a été prouvée correcte. De plus une *implantation* du problème ANSV avec *BH* a également été prouvée correcte. Ce sont donc des *programmes* qui ont été prouvés corrects avec un assistant de preuve. Même si les versions C++ de *BH* et ANSV n'ont pas été prouvées formellement correctes, elles sont proches de ces deux autres versions. Seul l'algorithme de He et Huang est montré correct dans [13, 62], et c'est une preuve rigoureuse mais non formelle. L'implantation de cet algorithme n'a pas été vérifiée : ce serait un très gros travail de le faire. À ce sujet, nous avons pu observer marginalement des résultats erronés lors des expérimentations sur cet algorithme (de l'ordre d'une dizaine de valeurs fausses sur des ensemble de plusieurs millions) sans toutefois réussir à trouver la cause de ces erreurs.

7.4.3 Effort de programmation

Nous sommes en présence de deux implémentations de programmes résolvant le même problème, l'une d'entre elles étant très performante tandis que l'autre est mise en place dans un modèle de haut niveau. Nous avons donc ici une bonne occasion de comparer selon l'axe de la difficulté de développement nos approches de haut niveau avec des codes plus optimisés, nous allons donc appliquer les métriques de Halstead sur ces codes afin de chiffrer l'effort nécessaire à leur programmation (Table 7.2).

La version utilisant *BH* est facile à analyser car l'implémentation est divisée en trois fonctions distinctes : `CandidatsL` qui est l'homomorphisme à gauche, `CandidatsR` qui est l'homomorphisme à droite et `Pickup` qui est la fonction k à appliquer sur chaque triplet (résumé gauche, élément central, résumé droit). De plus, `CandidatsL` et `CandidatsR` effectuent exactement le même calcul en inversant simplement l'ordre des itérateurs utilisés, donc ces deux fonctions auront le même coût.

Ces deux fonctions sont conceptuellement très simples, leur coût total reste donc faible, du même ordre de grandeur que l'algorithme séquentiel par pile. A l'opposé, l'algorithme de He et Huang possède une coût total plusieurs dizaines de fois supérieur ce qui est cohérent par rapport au temps que nous avons dû passer sur son développement.

En comparant ces mesures avec les résultats expérimentaux, nous pouvons constater que le rapport entre le temps de développement et les performances obtenues est assez proche pour les deux approches, aucune des deux ne semblent donc particulièrement se distinguer dans le cas général, chacune ayant ses propres avantages. Cependant l'approche utilisant *BH* est moins contraignante (applicable sur n'importe quel jeu de données) et est correcte par construction.

Une des raisons de la différence de performance se trouve dans le calcul séquen-

tiel qui n'est pas optimal dans l'implémentation *BH* car celui-ci utilise la sémantique basée sur *mapAround* où chaque élément est calculé indépendamment et nécessite le calcul des résumés gauches et droits complets. Une extension envisagée de nos travaux consisterait donc à étendre le fonctionnement du squelette *BH* afin de permettre à l'utilisateur l'application d'une fonction séquentielle plus efficace (dans notre exemple, le calcul d'ANSV par pile) sur l'ensemble du tableau lors de la dernière étape du calcul, en lieu et place de l'application de la fonction k (`Pickup` dans notre exemple) séparément sur chacun des éléments.

Algorithme / Métriques	η_1	η_2	η	N_1	N_2	N	V	D	E
BH									
CandidatsL	19	9	28	42	25	67	322	26,4	8500
Pickup	18	16	34	67	50	117	595	28,1	16660
HH									
Ansv Séquentiel	22	15	37	85	49	134	698	35,9	25058
Fonction principale	35	66	101	579	416	995	6625	110,3	730737

Table 7.2 – *Métriques de Halstead appliquées aux implémentations d'ANSV*

7.5 Conclusion

Dans ce chapitre, nous avons étudié la théorie des homomorphismes BSP et implémenté ses mécanismes à travers un squelette dans la bibliothèque OSL. Nous avons ensuite effectué la dérivation de deux problèmes algorithmiques classiques afin d'effectuer leur calcul grâce au mécanisme des homomorphismes BSP. Des expérimentations sur des jeux de données de tailles conséquentes montrent que la parallélisation de ces algorithmes fournit de bons résultats à la fois sur des architectures en mémoire partagée et en mémoire distribuée. La comparaison avec un algorithme parallèle de référence montre que les performances obtenues sont beaucoup moins efficaces, mais la mesure de la difficulté de programmation montre qu'à l'inverse l'approche par homomorphisme BSP est beaucoup plus rapide à développer.

Cette approche s'inscrit donc de manière cohérente dans notre démarche en fournissant au développeur des outils permettant de facilement développer des programmes parallèles offrant des performances raisonnables, en apportant en plus la possibilité de développer des programmes corrects.

Conclusion

8

Sommaire

8.1 Bilan 135
8.2 Perspectives 136

8.1 Bilan

Les travaux présentés dans cette thèse s'intéressent aux modèles de programmation parallèles et leurs apports pour les développeurs et terme de performance, d'expressivité et de productivité. Nous avons décrit un protocole expérimental permettant de mesurer de manière fiable les performance et le coût de développement d'un programme. Ce protocole a été appliqué sur l'optimisation d'un algorithme classique de multiplication de matrices à travers divers modèles, dont la vectorisation par instructions SSE et la parallélisation sur plusieurs fils d'exécution grâce à la bibliothèque OpenMP. Les résultats de ces expérimentation vont dans le sens de notre intuition de départ, à savoir que les modèles de bas niveau sont les plus à même d'obtenir les meilleures performances possibles mais impliquent des temps de développement très longs alors que les approches de haut niveau fournissent une bien meilleure productivité pour les programmeurs tout en apportant des performances acceptables.

La suite des travaux s'est donc orientée sur les langages à squelettes à travers la bibliothèque *Orléans Skeleton Library*. Une première version de cette bibliothèque existait préalablement à nos travaux, notre première contribution sur cette dernière a été l'implémentation d'une version remaniée afin de lever des limitations existantes notamment au niveau de son extensibilité et des performances des programmes générés.

Une fonctionnalité d'OSL peu courante au sein des langages à squelettes est la possibilité de manipuler explicitement la distribution des données au sein de l'architecture parallèle utilisée. Nous avons généralisé le mécanisme de distribution en introduisant un nouveau squelette **`redistribute`** qui remplace des squelettes spécifiques existant précédemment. Les squelettes de manipulation de distribution ont été appliqués pour

la programmation d'un algorithme complexe de tri en parallèle, ce dernier offre de bonnes performances tout en ayant été assez simple à développer.

Nous avons ensuite enrichi les capacités d'expression d'OSL en introduisant deux mécanismes : le premier est le support des exceptions dans un cadre distribué, ce qui permet ainsi l'utilisation des mécanismes d'exception dans les programmes à squelettes de manière similaire à leur utilisation habituelle au sein des programmes séquentiels. Les mesures expérimentales ont montré que l'utilisation de ce mécanisme a un impact non négligeable sur les performances, mais celui-ci reste néanmoins acceptable.

Le dernier mécanisme que nous avons apporté à OSL est le support du calcul des homomorphismes BSP qui permettent d'exprimer dans une sémantique de très haut niveau des algorithmes dont la parallélisation est implicite et prouvée correcte. Le calcul par homomorphisme BSP est un modèle extrêmement abstrait. Le calcul des valeurs inférieures les plus proches dans un tableau, que nous avons implémenté dans ce modèle, possède également une implémentation de référence utilisant un modèle de bas niveau. Nous avons donc appliqué à nouveau le protocole expérimental mis en place dans les premier travaux afin de comparer les performances et le coût de développement des deux approches. Les conclusions sur ces expérimentations sont dans la lignée des résultats précédents : l'approche de bas niveau offre les meilleures performances mais sa mise en place est très longue, alors que le programme utilisant l'homomorphisme BSP est extrêmement simple à implémenter tout en fournissant des performances acceptables et se parallélise bien.

8.2 Perspectives

Les divers travaux que nous avons effectués sur OSL ouvrent la voie à de nombreuses possibilités d'amélioration.

Un premier axe possible serait celui de l'expressivité. Dans le chapitre 5 nous avons étudié les possibilités offertes à l'utilisateur de modifier la distribution des données. Cependant les architectures matérielles évoluent vers une hétérogénéité importante et les calculs appliqués par un programme à squelettes ne sont pas forcément équilibrés même si les données le sont. Il peut donc être difficile pour le programmeur de déterminer la distribution des données entraînant les meilleures performances. Une piste envisagée serait donc l'ajout d'un mécanisme d'équilibrage automatique afin de décharger le programmeur de cette tâche. Dans le même ordre d'idées, les travaux du chapitre 6 permettent aux programmeurs d'utiliser les exceptions de manière globale dans les programmes à squelettes afin de récupérer les erreurs d'exécution dans l'environnement distribué, il serait intéressant de continuer dans cette voie afin de mettre en place des mécanismes automatiques de récupération d'erreur.

Au-delà du matériel ciblé, certains mécanismes actuels de la bibliothèque sont intrinsèquement peu efficaces dans de nombreux cas. Les opérations modifiant la structure des tableaux présentées dans le chapitre 5 effectuent des copies souvent inutiles, les homomorphismes BSP décrits au chapitre 7 pourraient exécuter un calcul plus efficace dans leur partie locale, et au niveau architectural seule la mémoire distribuée est correctement exploitée.

Une des raisons avancées expliquant le manque de performances des homomorphismes BSP vient de la sémantique séquentielle de ce calcul qui utilise le même découpage par homomorphismes que la partie parallèle. L'utilisation des homomorphismes est importante pour l'obtention des valeurs résumant les données distribuées sur les processus distant tout en minimisant les coûts de communication. Cependant leur utilisation dans la dernière étape locale qui effectue le calcul du résultat final indépendamment pour chaque élément ne semble pas idéale. Nous souhaiterions ajouter la possibilité pour l'utilisateur de fournir une autre fonction appliquant le calcul sur l'ensemble des données locales, ceci afin qu'il puisse mettre en place un calcul mieux adapté pour chaque algorithme implémenté dans ce modèle.

Les opérations sur la structure des tableaux entraînent systématiquement la création d'une copie du tableau en entrée. Dans de nombreux cas cette copie n'est pas utile car le tableau original n'est plus utilisé dans la suite du programme, et une simple modification dans la structure du tableau d'origine aurait suffit. Un tel mécanisme serait en fait utile pour tous les squelettes dont le calcul ne peut pas être fusionné par le mécanisme d'*expression templates*. Conjointement à l'ajout de ce mécanisme, il faudrait donner à l'utilisateur la possibilité de contrôler son utilisation, car il se peut que le tableau ait besoin d'être réutilisé dans son état d'origine. Une autre possibilité plus élégante mais plus délicate à mettre en place serait une analyse automatique des programmes à squelettes afin de déterminer le comportement idéal à adopter, des outils existant tels que la bibliothèque Boost.Proto, qui permet par des techniques de métaprogrammation l'analyse d'expressions, pourraient être utilisés à cette fin.

Plus généralement, l'analyse des expressions de squelettes est une voie qu'il serait intéressant d'explorer pour la mise en place d'optimisations plus globales par des règles de réécriture. La fusion de boucles grâce aux *expression templates* est déjà en soi un exemple de ce type d'optimisation mais cette dernière ne peut s'appliquer que localement sur des squelettes successifs compatibles.

Une dernière piste importante à explorer est l'exploitation du matériel par la bibliothèque. OSL exploite efficacement la mémoire distribuée mais il reste d'autres caractéristiques du matériel dont l'exploitation pourrait apporter des gains de performance conséquent, notamment la vectorisation par les unités SIMD, la mémoire partagée par les multi-processeurs multi-cœurs et la présence d'accélérateurs graphiques. La possibilité de mettre en place des règles de réécriture serait ici aussi un atout afin de transformer le code pour l'adapter à ces mécanismes particuliers.

Annexe

Les systèmes SPEED et Mirev

A

Sommaire

A.1 SPEED 141
A.2 Mirev 141

A.1 SPEED

La machine SPEED est un unique nœud de calcul conçu autour d'une architecture NUMA. Celle-ci possède 4 processeurs AMD Opteron « Magny-Cours », chaque processeur possédant 12 cœurs et étant couplé à une banque de mémoire individuelle de 16 Go, pour un total de 48 cœurs et 64 Go de mémoire sur l'ensemble du système.

Le système installé sur cette machine est Ubuntu 13.04, le compilateur utilisé est gcc 4.7.3 et MPI est manipulé grâce à OpenMPI 1.5.

Processeurs	4
Cœurs par processeur	12
RAM	64Go
Nombre total de cœurs	48
Type de processeur	AMD Opteron Processor 6174 2.2 GHz

A.2 Mirev

Mirev (*Mur d'Images pour la REalité Virtuelle*) est une grappe originellement conçue pour des applications de visualisation et de réalité virtuelle. Elle comporte 8 nœuds

reliés par un réseau Gigabit Ethernet et contenant chacun 2 processeurs quadri-cœurs et 16 Go de mémoire. La grappe contient également une machine serveur spécifique, un NAS de grande capacité et des nœuds plus anciens contenant des processeurs double-cœurs mais nous n'utilisons pas ces éléments dans nos expériences.

Le système installé sur cette grappe est Ubuntu 13.04, le compilateur utilisé est gcc 4.7.3 et MPI est manipulé grâce à OpenMPI 1.6.4.

Nombre de nœuds	8
Processeurs par nœud	2
Cœurs par processeur	4
RAM	16 Go
Nombre total de cœurs	64
Type de processeur	AMD Opteron QuadCore 2376 2.3 GHz

B

Exemple détaillé du calcul des valeurs inférieures les plus proches

Nous illustrons dans la table B.1 le déroulement des différentes étapes de l'algorithme BSP de référence pour le calcul des ANSV [62] sur un exemple concret. Dans cet exemple, le tableau pour lequel nous voulons trouver les ANSV est distribué sur trois processus A, B et C. Ces trois lettres désignent indifféremment le processus ou la partie du tableau qu'il contient. MinX désigne l'élément minimal du tableau X. -> et <- désignent respectivement les relations « a pour NSV droite » et « est la NSV gauche de ».

La première étape consiste à calculer les ANSV localement à chaque tableau en n'utilisant que les valeurs directement visibles dans le processus courant. Nous construisons ensuite le tableau global contenant les minima de chaque tableau. Dans ce tableau, nous analysons les relations -> et <- existantes. Pour chacune de ces relations, nous pouvons définir deux sous-séquences sur chacun des processus en relation. La résolution de ANSV sur la concaténation de ces sous-séquence nous fournit de nouvelles solutions, qui sont ensuite fusionnées avec les premiers résultats n'utilisant que les éléments locaux.

Processus A	Processus B	Processus C
[19, 14, 24, 10, 22, 12, 16, 25]	[23, 37, 38, 24, 15, 20, 28, 26]	[18, 21, 11, 8, 5, 13, 17]
ANSV(A)	ANSV(B)	ANSV(C)
[(⊥,14)(⊥,10)(14,10)(⊥,⊥)(10,12)(10,⊥)(12,⊥)(16,⊥)]	[(⊥,15)(23,24)(37,24)(23,15)(⊥,⊥)(15,⊥)(20,⊥)(20,⊥)]	[(⊥,11)(18,11)(⊥,8)(⊥,5)(⊥,⊥)(5,⊥)(13,⊥)]
Tableau des minimas [10, 15, 5]		
MinB(15) -> MinC(5) et C est voisin de B		
	Seq1B : de MinB à la fin de B [15, 20, 28, 26]	Seq2B : du début de C au match de MinB [18, 21, 11]
	ANSV(Seq1B ++ Seq2B) [(⊥,10)(15,18)(20,26)(20,18)] ++ [(15,11)(18,11)(⊥,⊥)]	
MinA(10) -> MinC(5) et C n'est pas voisin de A MinA(10) <- MinB(15) -> MinC(5) : B est le noeud intermédiaire		
Seq1A : de MinA au match de MinB		Seq2A : du match de MinB au match de MinA
[10, 22, 12]		[11, 8]
	ANSV(Seq1A ++ Seq2A) [(⊥,8)(10,12)(⊥,11)] ++ [(10,8)(⊥,⊥)]	
MinA(10) <- MinB(15) et B est voisin de A		
Seq4B : du match de MinB à la fin de A	Seq3B : du début de B à MinB	
[12, 16, 25]	[23, 37, 38, 24, 15]	
ANSV(Seq4B ++ Seq3B) [(⊥,⊥)(12,15)(16,23)] ++ [(16,15)(23,15)(37,24)(23,15)(12,⊥)]		
Fusion de ANSV(A) et des solutions de Seq1A et Seq4B	Fusion de ANSV(B) et des solutions de Seq1B et Seq3B	Fusion de ANSV(C) et des solutions de Seq2B et Seq2A
[(⊥,14)(⊥,10)(14,10)(⊥,8)(10,12)(10,11)(12,15)(16,23)]	[(16,15)(23,15)(37,24)(23,15)(12,10)(15,18)(20,26)(20,18)]	[(15,11)(18,11)(10,8)(⊥,5)(⊥,⊥)(5,⊥)(13,⊥)]

Table B.1 – *Exemple de calcul des ANSV*

Bibliographie

[1] A. Albrecht. Measuring Application Development Productivity. In I. B. M. Press, editor, *IBM Application Development Symp.*, pages 83–92, Oct. 1979. Cité page 37.

[2] M. Aldinucci, M. Danelutto, and P. Dazzi. Muskel: An expandable skeleton environment. *Scientific International Journal for Parallel and Distributed Computing*, 8:325–341, 2007. Cité page 28.

[3] M. Aldinucci, M. Danelutto, and P. Teti. An Advanced Environment Supporting Structured Parallel Programming in Java. *Future Generation Computer Systems*, 19:611–626, 2002. Cité page 28.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999. Cité page 33.

[5] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In R. Eigenmann and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2001. Cité page 34.

[6] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20:404–418, 2009. Cité page 15.

[7] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, 1999. Cité page 26.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks - summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM. Cité page 33.

[9] K. Bańczyk, T. Boiński, and H. Krawczyk. Object serialization and remote exception pattern for distributed C++/MPI application. In *PaCT*, LNCS, pages 188–193, Berlin, Heidelberg, 2007. Springer. Cité page 105.

[10] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Evaluating the performance of skeleton-based high level parallel programs. In *The International Conference on Computational Science (ICCS 2004), Part III, LNCS*, pages 299–306. Springer Verlag, 2004. Cité page 27.

[11] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In J. C. Cunha and P. D. Medeiros, editors, *11th International Euro-Par Conference*, LNCS 3648, pages 761–770. Springer, 2005. Cité page 27.

[12] M. Beran. Decomposable bulk synchronous parallel computers. In *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, SOFSEM '99, pages 349–359, London, UK, 1999. Springer-Verlag. Cité page 22.

[13] O. Berkman, B. Schieber, and U. Vishkin. Optimal Doubly Logarithmic Parallel Algorithms Based on Finding All Nearest Smaller Values. *Journal of Algorithms*, 14(3):344–370, 1993. Cité page 133.

[14] M. Berry. Public international benchmarks for parallel computers: Parkbench committee: Report-1. *Sci. Program.*, 3(2):100–146, June 1994. Cité page 34.

[15] J. Berthold, M. Dieterle, and R. Loogen. Implementing parallel google map-reduce in eden. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par'09, pages 990–1002, Berlin, Heidelberg, 2009. Springer-Verlag. Cité page 29.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM. Cité page 35.

[17] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987. Cité page 115.

[18] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004. Cité page 22.

[19] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. Cité page 34.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995. Cité page 15.

[21] G. H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In Bougé et al. [24]. Cité page 27.

[22] G.-H. Botorog and H. Kuchen. Skil: an imperative language with algorithmic skeletons for efficient distributed programming. In *5th Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996. Cité page 27.

[23] G.-H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196:71–107, 1998. Cité page 27.

[24] L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors. *Euro-Par'96 Parallel Processing*, number 1123–1124 in LNCS, Lyon, August 1996. LIP-ENSL, Springer. Cité pages 146 et 147.

[25] W. Bousdira, F. Loulergue, and J. Tesson. A Verified Library of Algorithmic Skeletons on Evenly Distributed Arrays. In *ICA3PP 2012*, LNCS, pages 218–232. Springer, 2012. Cité page 59.

[26] S. Breitinger. *Design and Implementation of the Parallel Functional Language Eden*. PhD thesis, Philipps-Universität Marburg, 1998. Cité page 29.

[27] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Pena-Marí. Eden– the paradise of functional concurrent programming. In Bougé et al. [24]. Cité page 29.

[28] D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2004. Cité page 22.

[29] D. Caromel and M. Leyton. Fine tuning algorithmic skeletons. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Euro-Par Parallel Processing*, volume 4641 of *LNCS 4641*, pages 72–81. Springer, 2007. Cité page 28.

[30] D. Caromel and M. Leyton. A transparent non-invasive file data model for algorithmic skeletons. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–10, 2008. Cité page 28.

[31] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 35–46, New York, NY, USA, 2011. ACM. Cité page 22.

[32] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly France, 2000. freely available in english at `http://caml.inria.fr/oreilly-book`. Cité page 59.

[33] P. Ciechanowicz and H. Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In *IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 108–113, 2010. Cité pages 28 et 112.

[34] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989. Available at `http://homepages.inf.ed.ac.uk/mic/Pubs`. Cité pages 2 et 23.

[35] M. Cole. Parallel Programming with List Homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995. Cité pages 115 et 117.

[36] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004. Cité page 27.

[37] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. Cité page 17.

[38] J. O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7:24–27, February 1995. Cité page 77.

[39] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990. Cité page 42.

[40] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998. Cité page 15.

[41] P. D'Alberto and A. Nicolau. Adaptive Strassen and ATLAS's DGEMM: A Fast Square-Matrix Multiply for Modern High-Performance Systems. In *Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, HPCASIA, pages 45–, Washington, DC, USA, 2005. IEEE Computer Society. Cité page 42.

[42] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. OcamlP3L a functional parallel programming system. Liens-98-1, ENS, 1998. Cité page 30.

[43] M. Danelutto and M. Stigliani. SKElib: Parallel Programming with Skeletons in C. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par 2000, pages 1175–1184, 2000. Cité page 27.

[44] M. Danelutto and P. Teti. Lithium: A Structured Parallel Programming Environment in Java. In P. M. A. Sloot, C. J. K. Tan, J. Dongarra, and A. G. Hoekstra, editors, *ICCS*, volume 2330 of *LNCS*, pages 844–853. Springer, 2002. Cité page 28.

[45] J. Darlington, Y.-k. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 19–28, New York, NY, USA, 1995. ACM. Cité page 26.

[46] R. Di Cosmo and M. Danelutto. A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. In *ICCS*, volume 9 of *Procedia Computer Science*, pages 1837–1846. Elsevier, 2012. Cité page 30.

[47] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Mar. 1990. Cité page 44.

[48] N. Drosinos and N. Koziris. Performance Comparison of Pure MPI vs Hybrid MPI-OpenMP Parallelization Models on SMP Clusters. In *In 18th Int. Parallel and Distributed Symposium*, page 15, 2004. Cité page 17.

[49] J. Enmyren and C. Kessler. SkePU: A Multi-Backend Skeleton Programming Library for Multi-GPU Systems. In *4th workshop on High-Level Parallel Programming and Applications (HLPP)*. ACM, 2010. Cité page 30.

[50] S. Ernsting and H. Kuchen. Algorithmic skeletons for multi-core, multi-gpu systems and clusters. *Int. J. High Perform. Comput. Netw.*, 7(2):129–138, Apr. 2012. Cité page 31.

[51] J. Falcou, J. Sérot, T. Chateau, and J.-T. Lapresté. Quaff: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32:604–615, 2006. Cité page 28.

[52] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC'78, pages 114–118, New York, NY, USA, 1978. ACM. Cité page 20.

[53] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. Cité page 107.

[54] L. Gesbert, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson. Systematic Development of Correct Bulk Synchronous Parallel Programs. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 334–340. IEEE, 2010. Cité pages 5, 115, 119 et 126.

[55] J. Gonzalez-Dominguez, G. Taboada, B. Fraguela, M. Martin, and J. Tourio. Servet: A benchmark suite for autotuning on multicore clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–9, 2010. Cité page 35.

[56] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975. Cité page 99.

[57] S. Gorlatch and H. Bischof. Formal Derivation of Divide-and-Conquer Programs: A Case Study in the Multidimensional FFT's. In D. Mery, editor, *Formal Methods for Parallel Programming: Theory and Applications*, pages 80–94, 1997. Cité page 115.

[58] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202 – 220, 2009. Cité page 16.

[59] M. H. Halstead. *Elements of Software Science*. Elsevier Science Ltd, 1977. Cité page 38.

[60] K. Hamidouche, J. Falcou, and D. Etiemble. Hybrid bulk synchronous parallelism library for clustered SMP architectures. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 55–62, New York, NY, USA, 2010. ACM. Cité page 22.

[61] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006. Cité page 113.

[62] X. He and C.-H. Huang. Communication Efficient BSP Algorithm for All Nearest Smaller Values Problem. *Journal of Parallel and Distributed Computing*, 61(10):1425–1438, 2001. Cité pages 129, 133 et 143.

[63] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006. Cité page 34.

[64] C. A. Herrmann and C. Lengauer. Hdc: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000. Cité page 29.

[65] A. P. J. Dongarra, P. Luszczek. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. Cité page 33.

[66] N. Javed. *Meta-programmed Algorithmic Skeletons: Implementations, Performances and Semantics*. PhD thesis, LIFO, University of Orléans, October 2011. Cité page 59.

[67] E. Johnson and D. Gannon. HPC++: experiments with the parallel standard template library. In *Proceedings of the 11th international conference on Supercomputing*, ICS 97, pages 124–131. ACM, 1997. Cité page 29.

[68] Y. Karasawa and H. Iwasaki. A Parallel Skeleton Library for Multi-core Clusters. In *International Conference on Parallel Processing (ICPP)*, pages 84–91. IEEE Computer Society, 2009. Cité page 29.

[69] C. F. Kemerer. An empirical validation of software cost estimation models. *Commun. ACM*, 30(5):416–429, May 1987. Cité page 37.

[70] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM. Cité page 22.

[71] H. Kuchen. A Skeleton Library. In *8th International Euro-Par Conference*, LNCS 2400, pages 620–629. Springer, 2002. Cité page 28.

[72] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5:308–323, September 1979. Cité page 33.

[73] J. Legaux, Z. Hu, F. Loulergue, K. Matsuzaki, and J. Tesson. Programming with BSP Homomorphisms. In F. Wolf, B. Mohr, and D. Ney, editors, *Euro-Par 2013 Parallel Processing*, number 8097 in LNCS, pages 446–457. Springer, 2013. Cité page 115.

[74] J. Legaux, S. Jubertie, and F. Loulergue. Experiments in Parallel Matrix Multiplication on Multi-Core Systems. In *ICA3PP 2012*, LNCS, pages 362–376. Springer, 2012. Outstanding Paper Nominee Award. Cité page 41.

[75] J. Legaux, F. Loulergue, and S. Jubertie. Managing Arbitrary Distributions of Arrays in Orléans Skeleton Library. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 437–444. IEEE, 2013. Cité page 87.

[76] J. Legaux, F. Loulergue, and S. Jubertie. OSL: an algorithmic skeleton library with exceptions. In *International Conference on Computational Science (ICCS)*, volume 18 of *Procedia Computer Science*, pages 260–269. Elsevier, 2013. Cité page 99.

[77] M. Leyton. *Advanced features for algorithmic skeleton programming*. PhD thesis, Université de Nice Sophia Anti Polis, 2008. Cité page 28.

[78] M. Leyton, L. Henrio, and J. M. Piquer. Exceptions for algorithmic skeletons. In P. D'Ambra, M. R. Guarracino, and D. Talia, editors, *16th International Euro-Par Conference*, LNCS 6272, pages 14–25. Springer, 2010. Cité page 113.

[79] M. Leyton and J. M. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *PDP*, pages 289–296. IEEE, 2010. Cité page 28.

[80] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *ICCS*, volume 3515 of *LNCS*, pages 1046–1054. Springer, 2005. Cité pages 22 et 59.

[81] C. Makassikis, V. Galtier, and S. Vialle. A skeletal-based approach for the development of fault-tolerant spmd applications. In *Proceedings of the 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '10, pages 239–248, Washington, DC, USA, 2010. IEEE Computer Society. Cité page 113.

[82] C. Makassikis, S. Vialle, and X. Warin. Ft-grelosss: A skeletal-based approach towards application parallelization and low-overhead fault tolerance. In *Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '12, pages 237–244, Washington, DC, USA, 2012. IEEE Computer Society. Cité page 113.

[83] G. Martín, M.-C. Marinescu, D. Singh, and J. Carretero. Flex-mpi: An mpi extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In F. Wolf, B. Mohr, and D. Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 138–149. Springer Berlin Heidelberg, 2013. Cité page 97.

[84] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *InfoScale'06: Proceedings of the 1st international conference on Scalable information systems*. ACM Press, 2006. Cité pages 29 et 112.

[85] A. Mazouz, S.-A.-A. Touati, and D. Barthou. Study of variations of native program execution times on multi-core architectures. In *Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems*, CISIS '10, pages 919–924, Washington, DC, USA, 2010. IEEE Computer Society. Cité page 36.

[86] T. J. Mccabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. Cité page 37.

[87] W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996. Cité page 20.

[88] F. McMahon. The Livermore Fortran Kernels: A Computer Test of Numerical Performance Range. Technical report, Technical Report UCRL-55745, Lawrence Livermore National Laboratory, 1986. Cité page 33.

[89] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic Inversion Generates Divide-and-Conquer Parallel Programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 146–155. ACM Press, 2007. Cité page 115.

[90] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder. SPEC MPI2007 an application benchmark suite for parallel systems using MPI. *Concurr. Comput. : Pract. Exper.*, 22:191–205, February 2010. Cité page 34.

[91] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011. Cité page 17.

[92] B. Nichols, D. Buttlar, and J. Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996. Cité page 14.

[93] OpenMP Architecture Review Board. OpenMP Application Program Interface version 3.0, may 2008. Cité page 15.

[94] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly, 2008. Cité page 116.

[95] S. Pelagatti. *A Methodology for the Development and the Support of Massively Parallel Programs*. PhD thesis, Univ. Pisa, 1993. Cité page 26.

[96] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998. Cité page 2.

[97] A. Peleg and U. Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):42–50, 1996. Cité page 46.

[98] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007. Cité pages 16 et 29.

[99] M. Rintala. Exceptions in remote procedure calls using C++ template metaprogramming. *Softw. Pract. Exper.*, 37(3):231–246, Mar. 2007. Cité page 105.

[100] J. Serot and D. Ginhac. Skeletons for parallel image processing: an overview of the skipper project. *Parallel Computing*, 28(12):1685–1708, 2002. Cité page 30.

[101] J. P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. Technical report, Stanford University, Stanford, CA, USA, 1992. Cité page 34.

[102] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997. Cité page 22.

[103] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998. Cité pages 16 et 100.

[104] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU Programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182, 2011. Cité page 31.

[105] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. Cité page 42.

[106] A. Strey and M. Bange. Performance Analysis of Intel's MMX and SSE: A Case Study. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 142–147, London, UK, 2001. Springer-Verlag. Cité page 46.

[107] J. Tesson. *Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels*. PhD thesis, LIFO, University of Orléans, November 2011. Cité pages 115, 116, 117 et 119.

[108] J. Tesson, H. Hashimoto, Z. Hu, F. Loulergue, and M. Takeichi. Program Calculation in Coq. In *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST2010)*, LNCS 6486, pages 163–179. Springer, 2010. Cité page 5.

[109] J. Tesson and F. Loulergue. A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation. In *International Conference on Computational Science (ICCS)*, Procedia Computer Science, pages 36–45. Elsevier, 2011. Cité page 5.

[110] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`. Cité pages 59 et 116.

[111] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998. Cité pages 92 et 93.

[112] S.-A.-A. Touati, J. Worms, and S. Briais. The speedup-test: a statistical methodology for programme speedup analysis and computation. *Concurrency and Computation: Practice and Experience*, 25(10):1410–1426, 2013. Cité page 36.

[113] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103, 1990. Cité pages 20 et 60.

[114] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28:1709–1732, 2002. Cité page 26.

[115] T. Veldhuizen. Techniques for Scientific C++. Computer science technical report 542, Indiana University, 2000. Cité pages 60 et 76.

[116] T. L. Veldhuizen. C++ templates are turing complete. Technical report, 2003. Cité page 72.

[117] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998. Cité page 72.

[118] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the javatm system. In *Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*, COOTS'96, pages 17–17, Berkeley, CA, USA, 1996. USENIX Association. Cité page 22.

[119] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. *SIGARCH Comput. Archit. News*, 23:24–36, May 1995. Cité page 34.

[120] G. Yaikhom, M. Cole, S. Gilmore, and J. Hillston. A structural approach for modelling performance of systems using skeletons. *Electr. Notes Theor. Comput. Sci.*, 190(3):167–183, 2007. Cité page 27.

[121] K. Yotov, K. Pingali, and P. Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, QEST '05, pages 168–, Washington, DC, USA, 2005. IEEE Computer Society. Cité page 35.

Ce document a ecrit a l'aide de texte GNU Emacs et du logiciel de composition typographique LaTeX 2ε.

Joeffrey LEGAUX

Squelettes algorithmiques pour la programmation et l'exécution efficaces de codes parallèles

Les architectures parallèles sont désormais présentes dans tous les matériels informatiques, mais les programmeurs ne sont généralement pas formés à leur programmation dans les modèles explicites tels que MPI ou les Pthreads. Il y a un besoin important de modèles plus abstraits tels que les squelettes algorithmiques qui sont une approche structurée. Ceux-ci peuvent être vus comme des fonctions d'ordre supérieur synthétisant le comportement d'algorithmes parallèles récurrents que le développeur peut ensuite combiner pour créer ses programmes. Les développeurs souhaitent obtenir de meilleures performances grâce aux programmes parallèles, mais le temps de développement est également un facteur très important. Les approches par squelettes algorithmiques fournissent des résultats intéressants dans ces deux aspects.
La bibliothèque *Orléans Skeleton Library* ou OSL fournit un ensemble de squelettes algorithmiques de parallélisme de données quasi-synchrones dans le langage C++ et utilise des techniques de programmation avancées pour atteindre une bonne efficacité. Nous avons amélioré OSL afin de lui apporter de meilleures performances et une plus grande expressivité. Nous avons voulu analyser le rapport entre les performances des programmes et l'effort de programmation nécessaire sur OSL et d'autres modèles de programmation parallèle. La comparaison rigoureuse entre des programmes parallèles dans OSL et leurs équivalents de bas niveau montre une bien meilleure productivité pour les modèles de haut niveau qui offrent une grande facilité d'utilisation tout en produisant des performances acceptables.

Mots clés : Modèles de haut niveau, squelettes algorithmiques, parallélisme quasi-synchrone, effort de programmation, expressivité, exceptions, distribution des données, homomorphismes

Algorithmic skeletons for efficient programming and execution of parallel codes

Parallel architectures have now reached every computing device, but software developers generally lack the skills to program them through explicit models such as MPI or the Pthreads. There is a need for more abstract models such as the algorithmic skeletons which are a structured approach. They can be viewed as higher order functions that represent the behaviour of common parallel algorithms, and those are combined by the programmer to generate parallel programs. Programmers want to obtain better performances through the usage of parallelism, but the development time implied is also an important factor. Algorithmic skeletons provide interesting results in both those fields.
The *Orléans Skeleton Library* or OSL provides a set of algorithmic skeletons for data parallelism within the bulk synchronous parallel model for the C++ language. It uses advanced metaprogramming techniques to obtain good performances. We improved OSL in order to obtain better performances from its generated programs, and extended its expressivity. We wanted to analyze the ratio between the performance of programs and the development effort needed within OSL and other parallel programming models. The comparison between parallel programs written within OSL and their equivalents in low level parallel models shows a better productivity for high level models : they are easy to use for the programmers while providing decent performances.

Keywords : High level programming models, algorithmic skeletons, bulk synchronous parallelism, development effort, expressivity, exceptions, data distribution, homomorphisms

PUCVL
Pôle Universités
Centre Val de Loire

Laboratoire d'Informatique Fondamentale d'Orléans
(LIFO)
Université d'Orléans - Faculté des Sciences
Bâtiment IIIA
Rue Léonard de Vinci
B.P. 6759
F-45067 ORLEANS Cedex 2, France

UNIVERSITE D'ORLEANS