HAL
open science

Programmation haute performance pour architectures hybrides

Rachid Habel

To cite this version:

Rachid Habel. Programmation haute performance pour architectures hybrides. Calcul parallèle, distribué et partagé [cs.DC]. Ecole Nationale Supérieure des Mines de Paris, 2014. Français. NNT : 2014ENMP0025 . tel-01101782v2

HAL Id: tel-01101782

https://pastel.hal.science/tel-01101782v2

Submitted on 18 Jan 2016

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.

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°432 :
Sciences des Métiers de l'Ingénieur

Doctorat ParisTech

THÈSE

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des Mines de Paris

Spécialité « Informatique temps-réel, robotique et automatique »

présentée et soutenue publiquement par

Rachid HABEL

le 19 novembre 2014

Programmation Haute Performance pour Architectures Hybrides

Directeur de thèse : **François IRIGOIN**
Encadrante de thèse : **Frédérique SILBER-CHAUSSUMIER**

Jury

Cédric BASTOUL, Professeur, Université de Strasbourg	Rapporteur
Frédéric DESPREZ, Directeur de recherche, LIP, ENS de Lyon	Président
Stef GRAILLAT, Professeur, LIP6, Université Paris 6	Rapporteur
François IRIGOIN, Directeur de recherche, CRI, MINES ParisTech	Examinateur
Jean-Luc LAMOTTE, Professeur, LIP6, Université Paris 6	Examinateur
Frédérique SILBER-CHAUSSUMIER, Maître de conférences, TELECOM SudParis	Examinatrice

T H È S E

MINES ParisTech
Centre de Recherche en Informatique
35, rue Saint-Honoré, 77305 Fontainebleau Cedex, France

Remerciements

Je tiens tout d'abord à remercier chaleureusement mes encadrants. Je remercie Frédérique Silber-Chaussumier pour avoir monté le projet scientifique de cette thèse et de m'avoir fait confiance en tant que candidat et je remercie François Irigoin d'avoir accepté d'assurer sa direction. Tout en m'accordant une certaine liberté, vous avez orienté mes choix de façon pertinente et vous m'avez repris et corrigé lorsque les directions envisagées n'étaient pas probantes. Votre aide, vos conseils et vos suggestions ont transformé ma façon de réfléchir et de construire des solutions. Vous m'avez honoré par votre encadrement et je veillerai à vous rendre hommage par ce que j'accomplirai dans toute la suite de ma carrière.

Je remercie vivement les membres de mon jury de thèse. Je remercie tout d'abord mes rapporteurs, Cédric Bastoul et Stef Graillat pour avoir accepté d'évaluer mon travail. Leurs retours ont été une aide très précieuse pour améliorer mon manuscrit et présenter les contributions de ma thèse dans les communications scientifiques. Je remercie Frédéric Desprez qui m'a fait l'honneur de présider le jury. Je remercie Jean-Luc Lamotte qui m'a appris la programmation parallèle en cours de Master à Paris 6 et qui a accepté mon invitation à participer au jury.

Je remercie les membres du département informatique de TELECOM SudParis qui m'ont réservé un accueil chaleureux dès mon arrivée. Un grand merci à Alain Muller pour avoir partagé ses connaissances sur STEP et pour m'avoir aidé à prendre en main le compilateur PIPS. Merci à Elisabeth Brunet qui m'a beaucoup aidé, notamment durant le projet européen MANY. Merci à Christian Parrot, Denis Conan, Christian Shüller, François Trahay, Gaël Thomas, Daniel Millot, Dominique Bouillet, Chantal Taconet, Alda Gançarski, Olivier Berger, Christian Bac, Claire Lecocq, Sophie Chabridon, Bruno Defude, Samir Tata, Djamel Bealïd, Walid Gaaloul, Amel Bouzeghoub, Amel Mammar et tous les autres que je n'ai pas cités ici, qu'ils me le pardonnent. Je remercie les membres du Centre de Recherche en Informatique de Fontainebleau pour leur accueil et leurs conseils et soutien tout au long de cette thèse. Merci à Corinne Ancourt, Claire Medrala, Pierre Jouvelot, Fabien Coelho, Claude Tadonki et tous les autres. Je remercie particulièrement Brigitte Houassine et Jacqueline Altimira qui m'ont aidé dans toutes mes démarches administratives. Un grand merci également à Fabienne Jézéquel et Pierre Fortin qui ont encadré mon stage de Master et avec qui j'ai continué à travailler pendant ma thèse.

Mes remerciements vont également à tous les doctorants et post-doctorants de TELECOM SudParis, du CRI et du LIP6. Nous avons partagé des discussions passionnantes et passionnées sur plein de sujets ainsi que des parties de babyfoot et de ping pong, des moments de convivialité et de décompression pour mieux repartir.

Je tiens également à remercier la direction de la recherche de MINES ParisTech et tout particulièrement Régine Molins, Daria Le Bozec et Pamela Vaulot.

Enfin, je remercie les membres de ma famille pour leur aide et soutien. Merci à mes sœurs Nadia et Kahina. Merci à ma Ferroudja chérie, sans qui rien ne serait arrivé.

À mes parents, à Ferroudja

Table des matières

1 Introduction **1**
1.1 Les architectures parallèles 1
1.2 La programmation parallèle 7
1.2.1 Grain de parallélisme 7
1.2.2 Parallélisme de tâche 8
1.2.3 Parallélisme pipeliné 8
1.2.4 Parallélisme de données 8
1.2.5 Loi d'Amdhal 9
1.3 Le calcul scientifique : dense et creux 9
1.4 Contexte et objectifs de la thèse 10
1.5 Défis posés par la programmation parallèle sur architectures hybrides 10

2 Programmation pour machines hybrides **13**
2.1 Bibliothèques 13
2.1.1 Communications coopératives, unilatérales 13
2.1.2 MPI 14
2.1.3 GASNet 15
2.1.4 Global Arrays 15
2.1.5 StarPU 15
2.1.6 Bilan des bibliothèques 16
2.2 Directives de compilation 16
2.2.1 OpenMP 16
2.2.2 STEP : *Système de Transformation pour l'Exécution Parallèle* 18
2.2.3 HMPP 19
2.2.4 OpenACC 20
2.2.5 hiCUDA 20
2.2.6 Cetus 20
2.2.7 XcalableMP 20
2.2.8 OmpSs 21
2.2.9 Bilan des solutions à base de directives 22
2.3 Langages de programmation 22
2.3.1 High Performance Fortran 22
2.3.2 Le langage dHPF 25
2.3.3 Forces et faiblesses de HPF et de dHPF 26
2.3.4 PGAS : *Partitioned Global Address Space* 27

2.3.5 SPOC 28
2.3.6 Bilan des langages de programmation 28
2.4 Approches de haut niveau 29
2.4.1 Distribution automatique 29
2.4.2 Modèle polyédrique 29
2.4.3 Mémoire distribuée partagée 30
2.4.4 Bilan des approches de haut niveau 30
2.5 Conclusion 31

3 dSTEP : directives pour la distribution des calculs et des données 32
3.1 Contexte 32
3.2 Modèle de programmation 33
3.2.1 Définitions : processus, processeur, mémoire locale 34
3.2.2 Distribution des calculs 34
3.2.3 La directive *dstep gridify* 35
3.2.4 Exploitation des cœurs d'un même nœud ou d'un accélérateur 37
3.2.5 Applicabilité de la directive *dstep gridify* 37
3.2.6 Exemples d'utilisation de la directive *dstep gridify* 37
3.2.7 Distribution des tableaux 39
3.2.8 Le halo 40
3.2.9 Le halo total 41
3.3 Modèle d'exécution 42
3.3.1 Vue globale du modèle de programmation de *dSTEP* 43
3.4 Conclusion 48

4 Modèle de distribution 49
4.1 HPFC 49
4.2 Modèle de distribution 49
4.2.1 Ensemble de processus 50
4.2.2 Grille virtuelle simple 50
4.2.3 Grille virtuelle de processus étendue 52
4.2.4 Domaine séquentiel et domaine distribué 58
4.2.5 Modèle de distribution des calculs sans calculs redondants . . 58
4.2.6 Modèle de distribution de tableau avec halo 64
4.3 Modèle combiné de distribution des calculs et des données 75
4.3.1 Définitions 75
4.3.2 Fonctions de changement de domaine 76
4.3.3 Unicité locale d'un élément de tableau dans le domaine distribué 76
4.3.4 Bonne distribution d'un tableau par rapport à un nid de boucles 78
4.3.5 Conservation du parallélisme dans le domaine distribué 79
4.3.6 Préservation de l'ordre séquentiel dans le domaine distribué. . 81
4.3.7 Correction de la valeur d'un élément dans le domaine distribué 81
4.3.8 Correction des valeurs mémoire dans le domaine distribué . . 82
4.3.9 Conclusion 83
4.4 Conclusion 83

5 Compilation de *dSTEP* **84**
5.1 PIPS . . . 84
5.2 Allocation mémoire . . . 85
5.2.1 Taille mémoire allouée pour un tableau distribué . . . 86
5.3 Compilation d'un nid de boucles . . . 87
5.3.1 Représentation syntaxique d'un nid de boucles . . . 87
5.3.2 Schéma général de compilation d'un nid de boucles . . . 88
5.3.3 La fonction *belongs_to* . . . 89
5.3.4 Compilation d'un nid de boucles parallèle . . . 89
5.3.5 Compilation d'un nid de boucles ordonné . . . 89
5.3.6 L'ensemble *Before* pour une tranche d'itérations . . . 92
5.3.7 Compilation d'une tranche d'itérations . . . 92
5.3.8 Exploitation des différents cœurs d'un même nœud . . . 92
5.4 Génération des communications . . . 93
5.4.1 Les voisins . . . 94
5.4.2 Génération des *send* pour une tranche d'itérations . . . 95
5.4.3 Génération des *recvs* . . . 95
5.4.4 Complétion des communications . . . 96
5.4.5 Correction de la mémoire distribuée . . . 96
5.5 Optimisations . . . 99
5.5.1 Utilisation des régions de tableaux . . . 99
5.5.2 Adaptation de la compilation d'un nid de boucle ordonné . . . 99
5.5.3 Adaptation de la compilation d'une tranche d'itérations . . . 100
5.5.4 Adaptation de la génération des communications *send* . . . 100
5.5.5 Adaptation de la génération des *recvs* . . . 102
5.5.6 Recouvrement des communications par les calculs . . . 104
5.6 Exemple de génération de code . . . 105
5.7 Conclusion . . . 108

6 Extension du schéma de compilation pour les machines multi-GPU **111**
6.1 Introduction . . . 111
6.2 Programmation des GPGPUs . . . 112
6.2.1 CUDA . . . 112
6.2.2 OpenCL . . . 112
6.3 Solutions pour l'exportation automatiques de calculs sur GPUs . . . 113
6.3.1 par4all . . . 113
6.3.2 Approches utilisant le modèle polyédrique . . . 113
6.4 Pourquoi ne pas réutiliser un *offload-compiler* . . . 114
6.5 Contraintes . . . 114
6.6 Modèle d'exécution . . . 114
6.7 Allocation des données sur GPU . . . 115
6.8 Compilation d'une tranche d'itérations pour GPU . . . 115
6.9 Communications entre hôte et accélérateur . . . 116
6.10 Les réductions . . . 118
6.11 Conclusion . . . 118

7 Modèle de coût **119**
7.1 Introduction . . . 119
7.2 Quelques modèles de coût . . . 119
7.3 Modèle de coût de *dSTEP* . . . 120
7.3.1 Hypothèses . . . 120
7.3.2 Temps d'exécution d'un nid de boucles distribué . . . 121
7.4 Utilisation du modèle de coût . . . 124
7.5 Comparaison entre les distributions multi-partitionnée et standard . . . 125
7.5.1 Nid de boucles parallèle . . . 125
7.5.2 Nid de boucles ordonné . . . 126
7.5.3 Simplification de $\Delta_{ordered}$ . . . 127
7.6 Conclusion . . . 129

8 Résultats expérimentaux **130**
8.1 Introduction . . . 130
8.2 Environnement de tests . . . 130
8.3 Multiplication de matrices . . . 131
8.3.1 Insertion des directives . . . 131
8.3.2 Code généré . . . 135
8.3.3 Évaluation des performances . . . 135
8.4 Jacobi . . . 136
8.4.1 Insertion des directives . . . 137
8.4.2 Évaluation des performances . . . 137
8.4.3 Jacobi Multi-GPU . . . 137
8.5 Le kernel Adi . . . 138
8.5.1 Choix des directives . . . 138
8.5.2 Évaluation des performances . . . 139
8.6 NAS BT . . . 141
8.6.1 Mise en œuvre de la distribution . . . 142
8.6.2 Optimisations . . . 143
8.6.3 Résultats pour la classe C . . . 145
8.6.4 Résultats pour la classe D . . . 147
8.7 LDPC . . . 148
8.8 Conclusion . . . 148

9 Conclusion **153**
9.1 Rappel du contexte . . . 153
9.2 Contributions . . . 153
9.2.1 Modèle général pour la distribution des calculs et des données 154
9.2.2 Schéma générique de compilation . . . 154
9.2.3 Génération de communications efficaces . . . 154
9.2.4 Implémentation d'un prototype . . . 155
9.2.5 Résultats expérimentaux . . . 155
9.3 Originalités . . . 155
9.3.1 Séparation des distributions des données et des calculs . . . 155

9.3.2 Plusieurs types de distributions et d'ordonnancement 156
9.3.3 Flot de contrôle simplifié 156
9.3.4 Modèle de coût 156
9.4 Limitations 156
9.5 Perspectives 156
9.5.1 Implémentation du backend GPU 156
9.5.2 Optimisation au niveau du nœud de calcul 157
9.5.3 Extension du caractère hybride 157
9.5.4 Plus d'expériences 158

Bibliographie **159**

Liste des tableaux

1.1 Exemples de clusters 5

2.1 Bilan des bibliothèques 16
2.2 Bilan des solutions à base de directives 22
2.3 Bilan des langages de programmation 29
2.4 Bilan des approches de haut niveau 30

3.1 Sémantique des types de distribution pour une boucle 36
3.2 Sémantique des types d'ordonnancement 36
3.3 Sémantique des types de distribution d'une dimension de tableau . . 40

4.1 Exemples de grilles virtuelles simples 50
4.2 Exemples de grilles virtuelles simples pour plusieurs valeurs de *nb_procs* 51

8.1 Valeurs des paramètres des classes C et D pour le programme BT . . 143

Table des figures

1.1 Taxonomie de Flynn 2
1.2 Mémoire partagée et mémoire distribuée 3
1.3 Fin de l'augmentation de la fréquence d'horloge d'un seul processeur . 4
1.4 Un nœud NUMA du cluster Adonis sur Grid5000 5
1.5 Accélérateurs utilisés dans les architectures parallèles 6
1.6 Performances et débits mémoire des GPUs comparés aux CPUs . . . 6
1.7 Architectures parallèles 7
1.8 Architectures hybrides 7
1.9 Environnement de programmation hybride 11

2.1 Classification des approches pour machines parallèles hybrides 14
2.2 Distribution des données avec HPF 23
2.3 Le problème des calculs en front-d'onde en HPF 25
2.4 dHPF : multi-partitioning et exécution en front d'onde 26

3.1 Exemples de distribution d'un tableau 2D sur 4 processus 41
3.2 Exemples de distribution avec halo d'un tableau 2D sur 4 processus . 42
3.3 Exemple de halo total (b), distribution sur 4 processus 43
3.4 Accès en écriture générés par le premier nid de boucles 45
3.5 Communications des éléments du tableau A 45
3.6 Accès en lecture générés par le deuxième nid de boucles 46
3.7 Accès en écriture générés par le troisième nid de boucles 47
3.8 Écritures générées par le quatrième nid de boucles, premier temps . . 47
3.9 Écritures générées par le quatrième nid de boucles, deuxième temps . 48

4.1 Construction d'une grille virtuelle simple de processus 52
4.2 Illustration de la distribution multi-partitionnée 54
4.3 Construction d'une grille virtuelle étendue 56
4.4 Illustration de la fonction *extend_id* 57
4.5 Illustration de la fonction *loop_nest_iteration* 61
4.6 Illustration de la violation de la contrainte de validité des halos . . . 67
4.7 Distribution bidimensionnelle par blocs avec halo 69
4.8 Distribution bi-dimensionnelle cyclique avec halo 72
4.9 Distribution bi-dimensionnelle multi-partitionnée 74
4.10 Changement de domaine 77
4.11 Exemple de code vérifiant la propriété de bonne distribution. 79

4.12 Exemple de code ne vérifiant pas la propriété de bonne distribution. . 80

5.1 Vue globale du compilateur dSTEP. 85

6.1 Modèle de communication multi-GPU 115

7.1 Équilibrage de la charge de calcul avec la distribution cyclique. 122
7.2 Tailles des messages échangés 123

8.1 Caractéristiques du cluster Edel. 131
8.2 Distribution des matrices A, B et C 133
8.3 Comparaison des distributions répliquée et avec halo total 134
8.4 Temps d'exécution de Matmul, N = 2048 137
8.5 Temps d'exécution de Matmul, N = 8192 138
8.6 Temps d'exécution de Jacobi, 1026 x 1026 139
8.7 Temps d'exécution de Jacobi, 8194 x 8194 140
8.8 Jacobi multi-GPU 141
8.9 Temps d'exécution du programme Adi 142
8.10 Graphe d'appel de la fonction adi dans le programme BT. 144
8.11 Optimisations pour BT 146
8.12 dSTEP BT classe C, comparaison avec la version MPI Fortran et UPC 147
8.13 dSTEP BT classe C, comparaison avec MPI Fortran, réseau TCP . . 148
8.14 dSTEP BT classe C, différentes grilles virtuelles de processus 149
8.15 dSTEP BT classe D, comparaison avec la version MPI Fortran 150
8.16 LDPC 152

Chapitre 1

Introduction

Un programme informatique, sous sa forme exécutable par un ordinateur, est un ensemble d'instructions. Une *exécution séquentielle* d'un programme consiste à exécuter ses instructions en séquence, une seule instruction à la fois. En revanche, une exécution *parallèle* d'un programme est l'exécution simultanée de plusieurs de ses instructions.

Pour obtenir une exécution parallèle, il faut : 1) avoir des architectures matérielles permettant une exécution parallèle et 2) écrire des programmes parallèles, à savoir des programmes exploitant les capacités de traitement parallèle offertes par les machines.

1.1 Les architectures parallèles

Les machines parallèles, permettant d'exécuter plus rapidement des programmes de plus en plus complexes, existent depuis les années soixante. Dans un article paru en 1972, Flynn [41] dresse une classification des différentes architectures des ordinateurs selon deux dimensions : la possibilité d'exécuter plusieurs instructions simultanément et la capacité d'appliquer les instructions sur des données multiples. La classe SISD *(Single Instruction Single Data)* représente les ordinateurs séquentiels. Les trois autres catégories d'architectures, SIMD *(Single Instruction Multiple Data)*, MISD *(Multiple Instruction Single Data)* et MIMD *(Multiple Instruction Multiple Data)* sont des architectures parallèles. Cette classification, représentée en Figure 1.1, est désignée depuis par *taxonomie de Flynn*.

Dans [37], Duncan dresse une classification des machines parallèles connues jusqu'au début des années quatre-vingt-dix, plus générale que la taxonomie de Flynn. On y trouve notamment les machines *vectorielles*, destinées à accélérer les calculs sur les vecteurs et les matrices. Les machines vectorielles sont constituées d'unités fonctionnelles formées de plusieurs étages reliés en pipeline et de registres vectoriels. Le parallélisme des architectures vectorielles est observé lorsque le pipeline est rempli ; un résultat de calcul est alors produit à chaque pas de temps correspondant à la durée d'exécution d'un seul étage. Les architectures *systoliques* visaient à réduire la latence des accès mémoire en faisant traverser les données par des unités de traitement spécialisées successives avant de les réécrire en mémoire. On trouve également

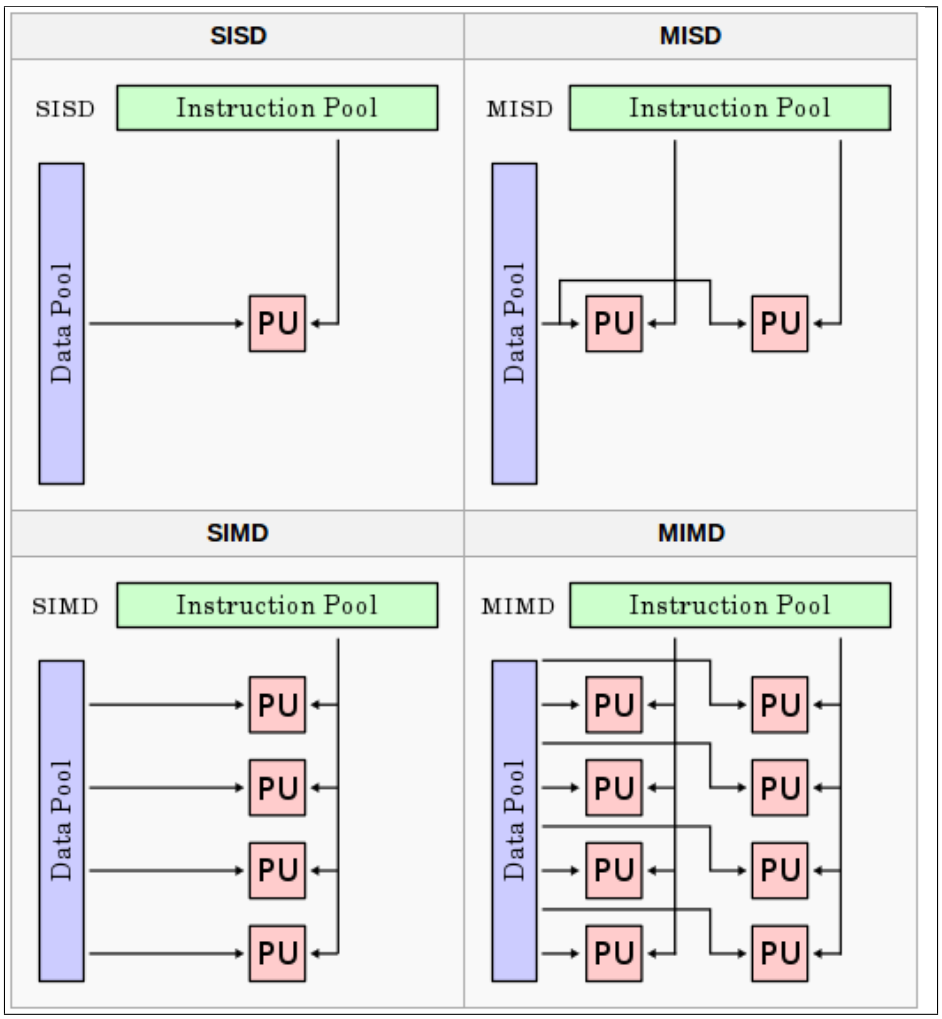


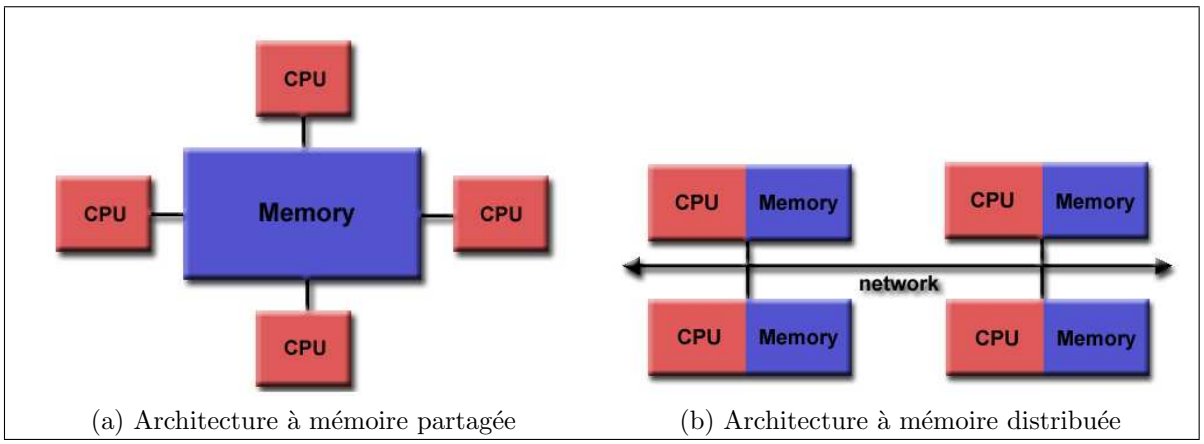
FIGURE 1.1 – Taxonomie de Flynn (source : Wikipedia)

des machines à *mémoire partagée* et des machines à *mémoire distribuée* qui sont classées dans la catégorie MIMD.

Une machine à mémoire partagée est constituée de plusieurs processeurs partageant l'accès à une seule mémoire centrale (fig. 1.2a). Les architectures à mémoire distribuée sont constituées de processeurs indépendants ayant chacun une mémoire physique propre (fig. 1.2b) et reliés par un réseau.

Plusieurs topologies d'interconnexion réseau pour les machines à mémoire distribuée ont été utilisées : les architectures en anneaux, en grilles, en cubes ou en hyper-cubes. Les organisations en tores apparues par la suite sont organisés en grilles multi-dimensionnelles où les connexions relient les voisins immédiats et les nœuds aux extrémités de la grille.

La caractéristique commune aux architectures à mémoire distribuée et à mémoire partagée est la mise en commun de plusieurs processeurs afin d'augmenter les performances des machines parallèles ainsi construites. En même temps, la fréquence



(a) Architecture à mémoire partagée (b) Architecture à mémoire distribuée

FIGURE 1.2 – Mémoire partagée et mémoire distribuée (source : Lawrence Livermore National Laboratory)

d'horloge des processeurs a augmenté exponentiellement depuis les années 1970. Cette évolution est conforme à une observation faite par Moore [70], devenue par la suite la *loi de Moore*, qui stipule que le nombre de transistors intégrés sur le circuit d'un microprocesseur doublait environ tous les deux ans. Cette évolution des performances a permis d'apporter des améliorations aux applications existantes et nouvellement écrites sans que les programmeurs n'aient eu à fournir des efforts particuliers pour les adapter aux nouveaux processeurs.

Dans un article paru en 2004, intitulé *The Free Lunch is Over* [87], Sutter annonce et explique les raisons de la fin de l'augmentation de la fréquence d'horloge d'un seul processeur, à cause notamment de la difficulté à dissiper la chaleur libérée. On voit sur la figure 1.3 l'aplatissement de la courbe représentant la fréquence d'horloge des processeurs depuis 2004. Ces limitations ont amené les constructeurs à intégrer sur le même circuit plusieurs microprocesseurs, appelés *cœurs*, pour avoir encore plus de puissance : c'est l'avènement des architectures *multi-cœurs*.

Si les architectures multi-cœurs continuent d'offrir de plus en plus de puissance de calcul, il n'est pas possible d'en bénéficier sans un effort explicite de programmation. Elles rendent ainsi la programmation parallèle nécessaire pour améliorer les performances des programmes et ce dès le niveau le plus élémentaire des architectures parallèles : *le nœud*.

Contrairement aux anciennes architectures à mémoire partagée (fig. 1.2) dans lesquelles les différents processeurs avaient des temps d'accès uniformes à la mémoire partagée (*UMA : Uniform Memory Access*), les nouvelles architectures à mémoire partagée ont des temps d'accès non uniformes aux données (*NUMA : Non Uniform Memory Access*). Cette non-uniformité est liée aux différents niveaux de caches que traversent les données, comme le montre la figure 1.4. Les architectures *(ccNUMA : cache-coherent NUMA)* implémentent, au niveau physique, un protocole de cohérence de cache pour maintenir une vision cohérente des données au niveau de chaque cœur de calcul.

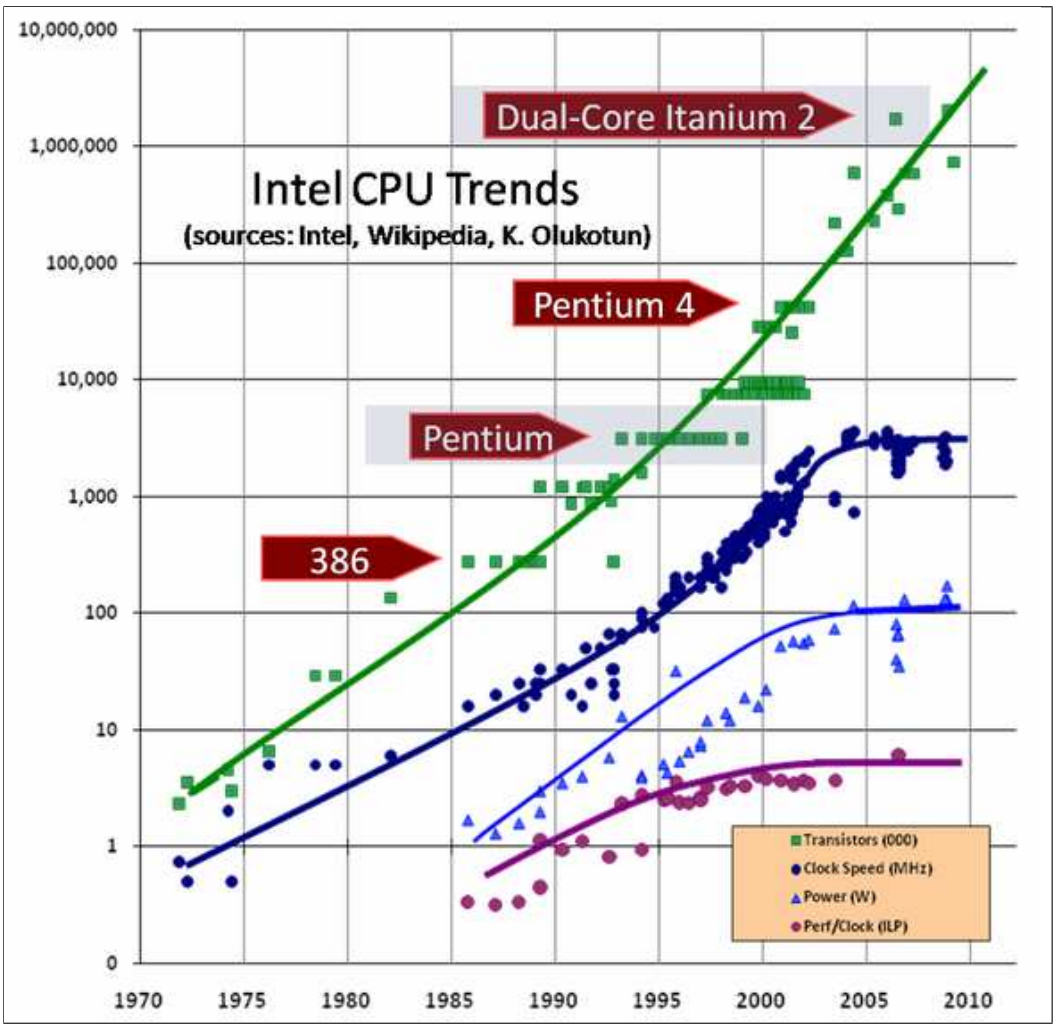


FIGURE 1.3 – Fin de l'augmentation de la fréquence d'horloge d'un seul processeur

La figure 1.4 montre l'architecture d'un nœud NUMA d'un cluster de la plateforme de calcul Grid'5000 [17]. Grâce à l'outil `hwloc` [21], on peut visualiser la topologie de ce nœud, avec deux processeurs, ayant chacun quatre cœurs. Chaque cœur a deux niveaux de cache propres et un troisième niveau partagé avec les autres cœurs d'un même processeur. Tous les cœurs partagent la même mémoire centrale.

Au niveau du nœud, on ajoute souvent un accélérateur ou un co-processeur spécialisé pour accélérer certains types de calculs. Les GPGPUs [79] (*General-Purpose Graphical Processing Units*) sont les accélérateurs les plus utilisés dans les clusters, notamment les GPUs NVIDIA comme le montre la figure 1.5. Les GPUs offrent des performances théoriques crêtes très élevées mais surtout des débits mémoire un ordre de grandeur plus grands que les CPUs comme le montre la figure 1.6 pour les GPUs NVIDIA.

Les architectures à mémoire distribuée, lorsqu'elles sont constituées d'un ensemble de nœuds homogènes, forment un *cluster* [35]. Les clusters constituent une forme très

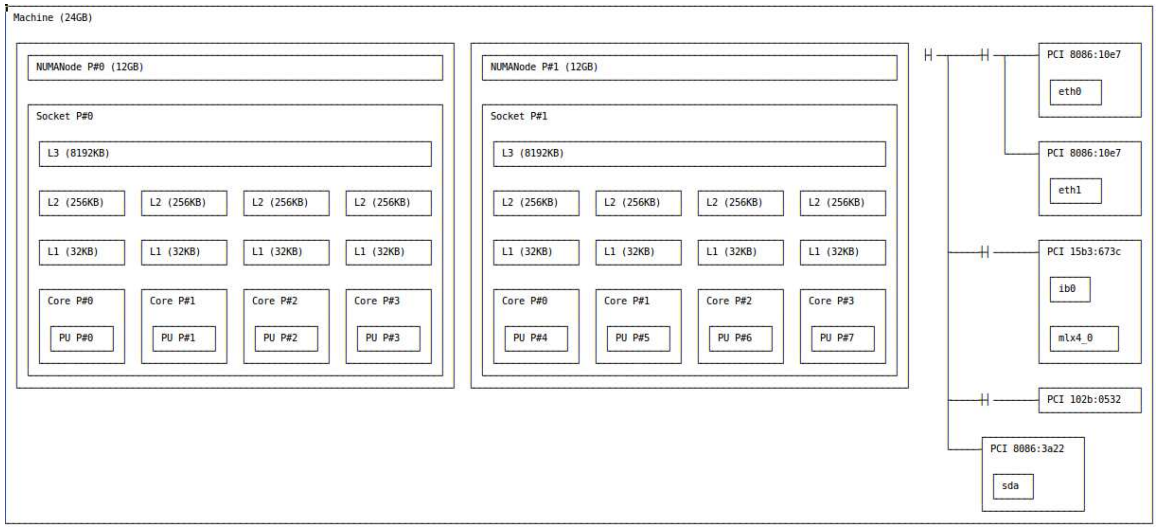


FIGURE 1.4 – Un nœud NUMA du cluster Adonis sur Grid5000

	NœUds NUMA	Accélérateurs	Réseau
B313 Salle machines à Télécom SudParis	13 nœuds x 4 cœurs = 52 cœurs	13 NVIDIA Quadro 2000	Ethernet 1 GB/s
Adonis Cluster de Grid5000	10 nœuds x 2 cpus x 4 cœurs = 80 cœurs	20 NVIDIA Tesla T10	Ethernet 1 GB/s Infiniband 40 GB/s
Titan #2 du Top500 en 2013, Cray	18688 nœuds x 1 cpu x 16 cœurs = 299008 cœurs	18688 NVIDIA Tesla K20	Cray GEMINI Interconnect 4.68-20 GB/s Topologie réseau en tore

TABLE 1.1 – Exemples de clusters

répandue d'architectures parallèles grâce au rapport élevé performances / coût de production . La figure 1.7 confirme cet état de fait pour les architectures parallèles actuelles. Une variante rare de clusters, appelée *Distributed Shared Memory* [6], émule une mémoire partagée au niveau de tous les cœurs d'un cluster.

De nos jours, les clusters sont formés de nœuds de calcul reliés par le réseau ; chaque nœud étant constitué de plusieurs processeurs multi-cœurs avec éventuellement des accélérateurs associés. On retrouve donc, dans ces machines, l'aspect mémoire distribuée entre les nœuds, mémoire partagée pour les processeurs et les cœurs d'un seul nœud et éventuellement un accélérateur graphique. On désigne ces clusters par *architectures hybrides* 1.8.

Les architectures hybrides constituent les systèmes cibles du travail de cette thèse.

Exemples de clusters. Le tableau 1.1 montre les caractéristiques de trois clusters de calcul, du cluster de laboratoire au deuxième super-calculateur du Top500 [1].

1. http ://www.top500.org/lists/2014/06

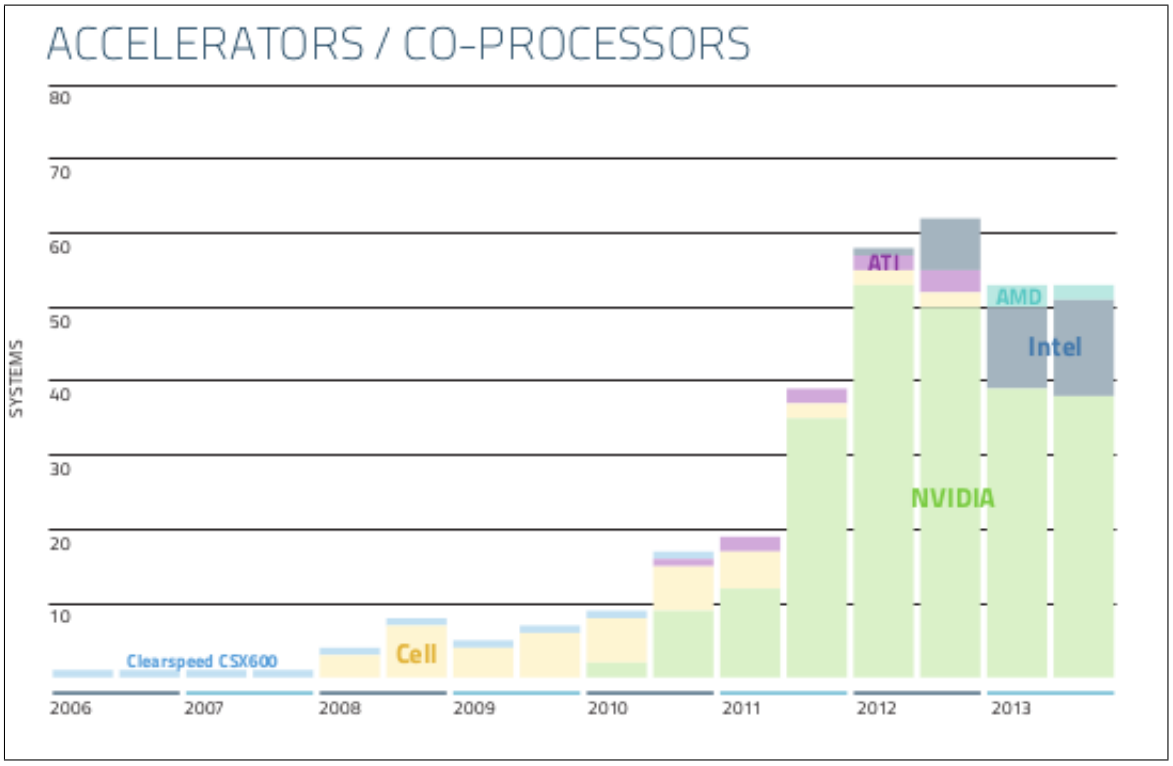
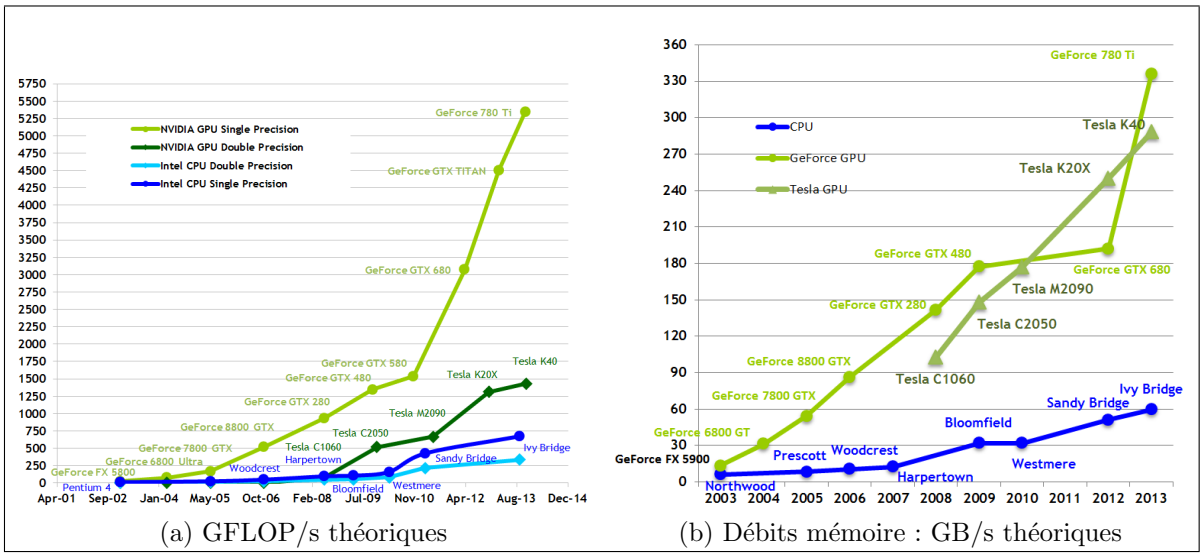


FIGURE 1.5 – Accélérateurs utilisés dans les architectures parallèles (source : http ://www.top500.org/)



(a) GFLOP/s théoriques (b) Débits mémoire : GB/s théoriques

FIGURE 1.6 – Performances et débits mémoire des GPUs NVIDIA comparés aux processeurs Intel (source : CUDA C Programming Guide)

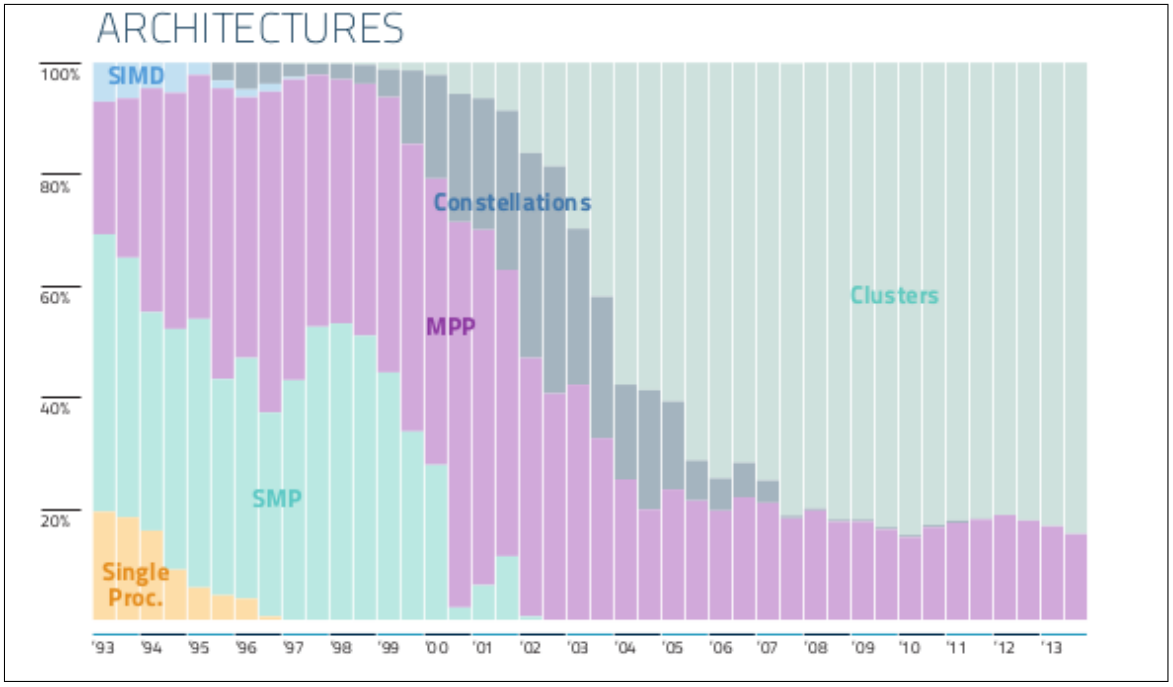
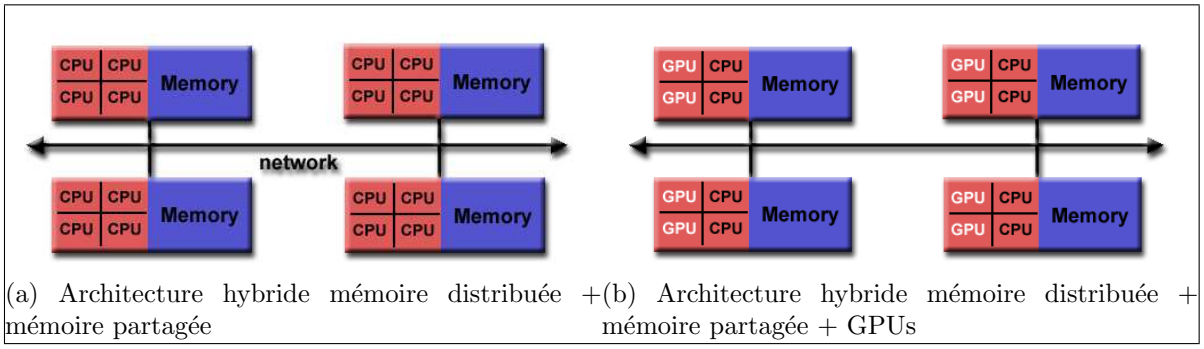


FIGURE 1.7 – Architectures parallèles (source : http ://www.top500.org/)



(a) Architecture hybride mémoire distribuée + mémoire partagée

(b) Architecture hybride mémoire distribuée + mémoire partagée + GPUs

FIGURE 1.8 – Architectures hybrides (source : Lawrence Livermore National Laboratory)

1.2 La programmation parallèle

1.2.1 Grain de parallélisme

Un programme parallèle est constitué de plusieurs parties qui s'exécutent en parallèle en interagissant à certains points du programme : les points de synchronisation qui incluent les communications. Le nombre d'interactions définit le grain de parallélisme du programme. Ainsi, un programme parallèle à grain fin est caractérisé par des synchronisations fréquentes pour des périodes de calcul courtes. A l'opposé, le parallélisme à gros grain est caractérisé par de longues périodes de calculs parallèles,

avant l'exécution de points de synchronisation. En absence de synchronisations, un programme parallèle est dit complètement parallèle (*embarrassingly parallel*).

1.2.2 Parallélisme de tâche

Le parallélisme de tâche consiste à isoler des parties d'un programme pouvant s'exécuter indépendamment les unes des autres, formant ainsi des tâches. Une tâche consomme des données en entrée et en produit en sortie et peut être exécutée dès que ses données en entrée sont prêtes. Le parallélisme de tâche est adapté aux applications irrégulières.

1.2.3 Parallélisme pipeliné

Le parallélisme pipeliné consiste à organiser les calculs en plusieurs tâches successives. Les tâches n'ayant pas de dépendance entre elles peuvent être exécutées en parallèle selon le modèle de parallélisme de tâches. Les autres tâches sont contraintes par les dépendances producteur/consommateur et sont donc non parallélisables à priori. Cependant, pour des applications à flux continu, comme les applications de traitement du signal, l'organisation des différentes tâches en pipeline permet à chaque étage d'effectuer en parallèle une partie du programme sur les données. Ainsi, en régime stationnaire, le traitement est entièrement parallélisé.

1.2.4 Parallélisme de données

Le parallélisme de données [52] consiste à appliquer le même calcul à des sous-ensembles différents de données. Ce calcul peut être une même instruction, par exemple une addition, appliqués à tous les éléments de deux vecteurs. Le modèle d'exécution SPMD [32] *Single Program Multiple Data* découle de ce paradigme et est largement utilisé pour écrire des programmes scientifiques pour architectures à mémoire distribuée. Il s'agit d'appliquer le même programme, sur plusieurs processus, à des parties différentes des données en fonction de l'identité de chaque processus. Des points de synchronisations globaux permettent de rétablir le contrôle au même niveau sur tous les processus. Ainsi, en dehors de ces points de synchronisation globale, le contrôle peut avancer différemment sur chaque processus, apportant un degré supplémentaire de parallélisme. Le parallélisme de données convient à des calculs denses sur des données de très grande taille, caractéristiques d'une famille de programmes de calcul scientifique. Les architectures de calcul telles que les GPUs de NVIDIA [76] sont, par construction, adaptées à ce type d'applications : elles sont constituées d'un très grand nombre de cœurs capables de calculer mais avec une logique de contrôle moins complexe que les CPUs. Les cœurs sont regroupés dans des unités appelées *Streaming Multiprocessors* qui réalisent le contrôle sur les cœurs en ordonnançant les threads à l'exécution, typiquement par groupes de trente-deux.

1.2.5 Loi d'Amdhal

La parallélisation ne peut réduire le temps d'exécution que si le programme présente suffisamment de parallélisme. Cette notion a été formalisée par Amdhal [2]. La loi d'Amdhal (eq 1.1) donne une borne supérieure théorique de l'accélération, S, d'un programme en fonction des temps de calcul des parties parallélisables, α, et des parties séquentielles (non parallélisables), par rapport au temps d'exécution séquentiel. Cette borne est atteinte lorsque le nombre de processus (ou threads) exécutant le programme, P, devient très grand.

$$S(P) = \frac{1}{1 - \alpha + \frac{\alpha}{P}} \tag{1.1}$$

La loi d'Amdhal permet de savoir si un effort de parallélisation est justifié pour toute une application. Dans une approche de parallélisation incrémentale, elle permet de cibler en priorité les régions exposant le plus de parallélisme. Cependant, dans la réalité, les performances obtenues oscillent autour de la prédiction théorique d'accélération soit positivement grâce notamment à des effets de cache plus favorables après découpage des données et parfois négativement si le support d'exécution du programme parallèle ajoute un surcoût important, principalement lié à la synchronisation, qui inclut les communications en mémoire distribuée.

1.3 Le calcul scientifique : dense et creux

Le calcul scientifique est un domaine à l'intersection des mathématiques, des sciences de l'ingénieur et de l'informatique. Des modèles mathématiques sont construits pour le traitement du signal ou pour décrire l'évolution de phénomènes naturels tels que l'écoulement des fluides ou les déformations d'un support matériel. Des algorithmes numériques implémentent la résolution de ces modèles. L'exécution des programmes qui en résultent permet de transformer, d'observer ou de reproduire les phénomènes naturels modélisés : prévisions météo, simulation de tests de collisions automobiles, effets spéciaux au cinéma, etc.

La parallélisation de ces programmes permet à la fois d'avoir des simulations plus rapides et de traiter des données de plus en plus grandes (zones météorologiques plus larges, maillages plus fins, ..).

L'information traitée en calcul scientifique est constituée de très grands ensembles de points représentant les objets modélisés comme les mailles ou les points des images. Ainsi, les données sont souvent stockées dans des tableaux multi-dimensionnels parcourus par des nids de boucles à plusieurs niveaux pour leur appliquer différents traitements.

La nature des problèmes modélisés se reflète dans les valeurs stockées dans les tableaux utilisés. Ainsi, les maillages non structurés, par exemple, génèrent des tableaux avec des éléments ne portant pas d'information (valeurs nulles). Cette classe de problèmes est désignée comme *calcul creux*. En opposition, le *calcul dense* est caractérisé par des tableaux sans valeurs nulles (ou presque). Le calcul scientifique

creux utilise des techniques de compression de matrices par lignes ou par colonnes (*CRS : Compressed Row Storage, CCS : Compressed Column Storage*) pour limiter le gaspillage de l'espace mémoire alloué [89]. Des tableaux d'indirection contenant les coordonnées des éléments compressés sont alors utilisés pour accéder à ces tableaux. Il en résulte des programmes plus difficiles à paralléliser sur des machines à mémoire distribuée. Des techniques de découverte d'informations sur les valeurs modifiées à l'exécution comme l'inspection/exécution sont utilisées [84] pour générer les communications.

1.4 Contexte et objectifs de la thèse

Le travail de cette thèse se place dans le contexte de la programmation des applications scientifiques denses, à parallélisme de données, sur des architectures hybrides.

L'objectif de la thèse est de proposer un modèle de programmation efficace dans le contexte défini. L'efficacité recherchée est définie selon trois dimensions :

1. la facilité de programmation : le programmeur doit apporter des informations sur le parallélisme et les données, sans avoir à entrer dans les détails relatifs à l'environnement d'exécution parallèle ;
2. la réduction du temps d'exécution globale du programme,
3. la réduction de l'empreinte mémoire sur chaque nœud de la machine parallèle pour pouvoir traiter des problèmes de très grande taille.

1.5 Défis posés par la programmation parallèle sur architectures hybrides

Les machines parallèles hybrides sont difficiles à programmer efficacement car elles nécessitent d'utiliser plusieurs modèles de programmation. De plus, la multiplication des niveaux de mémoire rend difficile la garantie de la correction du programme parallèle. Toutes ces considérations éloignent le programmeur des aspects purement algorithmiques du problème à résoudre. Enfin, le débogage et l'extensibilité sont d'autant plus difficiles à effectuer et à garantir.

La figure 1.9 montre un exemple d'environnement de programmation parallèle hybride. Pour chaque niveau, on indique certains des langages ou bibliothèques utilisés, qui seront détaillés au chapitre suivant. Pour tirer parti des performances de la machine parallèle, le programmeur doit gérer les spécificités de chaque niveau, avec le modèle de programmation adéquat.

Les points suivant sont à considérer lors de la programmation efficace des architectures hybrides.

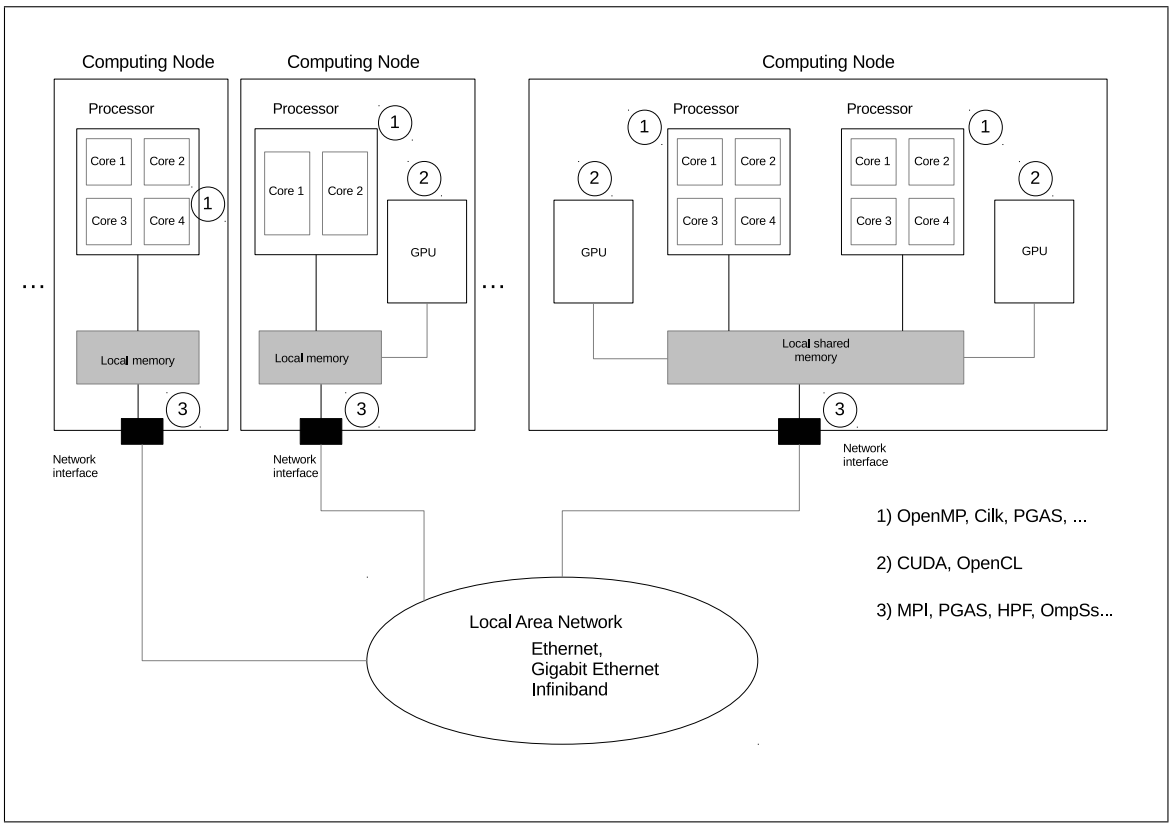


FIGURE 1.9 – Environnement de programmation hybride

La distribution des calculs. Les calculs doivent être répartis de la façon la plus équilibrée possible entre les différentes ressources de calcul afin de tendre vers la minimisation du temps d'exécution global du programme.

La distribution des données. Les données doivent être placées de façon à offrir la localité des calculs tout en minimisant l'empreinte mémoire sur chaque nœud.

Les communications. Pour que les communications soient efficaces, elles doivent être agrégées et non redondantes. Elles doivent également permettre de réaliser un recouvrement par les calculs afin d'absorber au maximum la latence du réseau.

Une solution au problème de la programmation efficace des architectures hybrides doit apporter des réponses satisfaisantes aux besoins et difficultés soulevées précédemment. La solution doit être également efficace en terme de programmabilité. Il faut offrir au programmeur un modèle de programmation à la fois simple et expressif mais également un modèle de performance prévisible.

L'organisation de ce manuscrit est la suivante : le chapitre 2 étudie les solutions existantes pour la programmation des architectures parallèles hybrides ; dans le chapitre 3, nous proposons *dSTEP*, un modèle de programmation pour la programmation

des machines parallèles hybrides ; dans le chapitre 4, nous formalisons la distribution et décrivons les propriétés nécessaires à une génération de code correcte ; le chapitre 5 présente la compilation efficace de *dSTEP*, notamment au niveau des communications générées ; le chapitre 6 démontre l'extensibilité de *dSTEP* aux architectures hybrides équipées de GPUs ; le chapitre 7 propose un modèle de coût pour aider le programmeur à choisir la meilleure distribution en fonction des caractéristiques de la machine parallèle disponible ; des résultats expérimentaux sont présentés au chapitre 8 ; enfin, nous concluons en résumant les contributions de cette thèse et en présentant les perspectives d'extension du travail réalisé au chapitre 9.

Chapitre 2

Programmation pour machines hybrides

Dans ce chapitre, nous présentons les solutions existantes pour la programmation des machines parallèles hybrides dans le contexte du calcul scientifique. Selon le niveau d'abstraction par rapport à l'architecture cible, nous classifions les différentes approches en quatre couches, comme représenté sur la figure 2.1 :

1. les bibliothèques,
2. les directives de compilation,
3. les langages de programmation,
4. les approches de haut niveau.

Nous relevons les limitations des approches existantes et nous montrons la nécessité d'une nouvelle solution que nous positionnons par rapport aux travaux existants. Nous nous intéressons particulièrement aux directives de compilation.

2.1 Bibliothèques

2.1.1 Communications coopératives, unilatérales

Les bibliothèques utilisées pour programmer les machines à mémoire distribuée utilisent le paradigme de passage de messages. Il existe deux façons d'implémenter un échange de message entre deux processus.

1. Les deux processus coopèrent pour échanger le message : le processus envoyant les données appelle explicitement une routine d'envoi (*send*) et le processus recevant les données appelle explicitement une routine de réception (*receive*). Il faut donc que les processus impliqués dans un échange de données connaissent exactement le type, la taille, et l'adresse des données échangées.
2. Un processus peut accéder aux données d'un autre processus sans que ce dernier ne coopère explicitement à l'échange du message (*one-sided communication*).

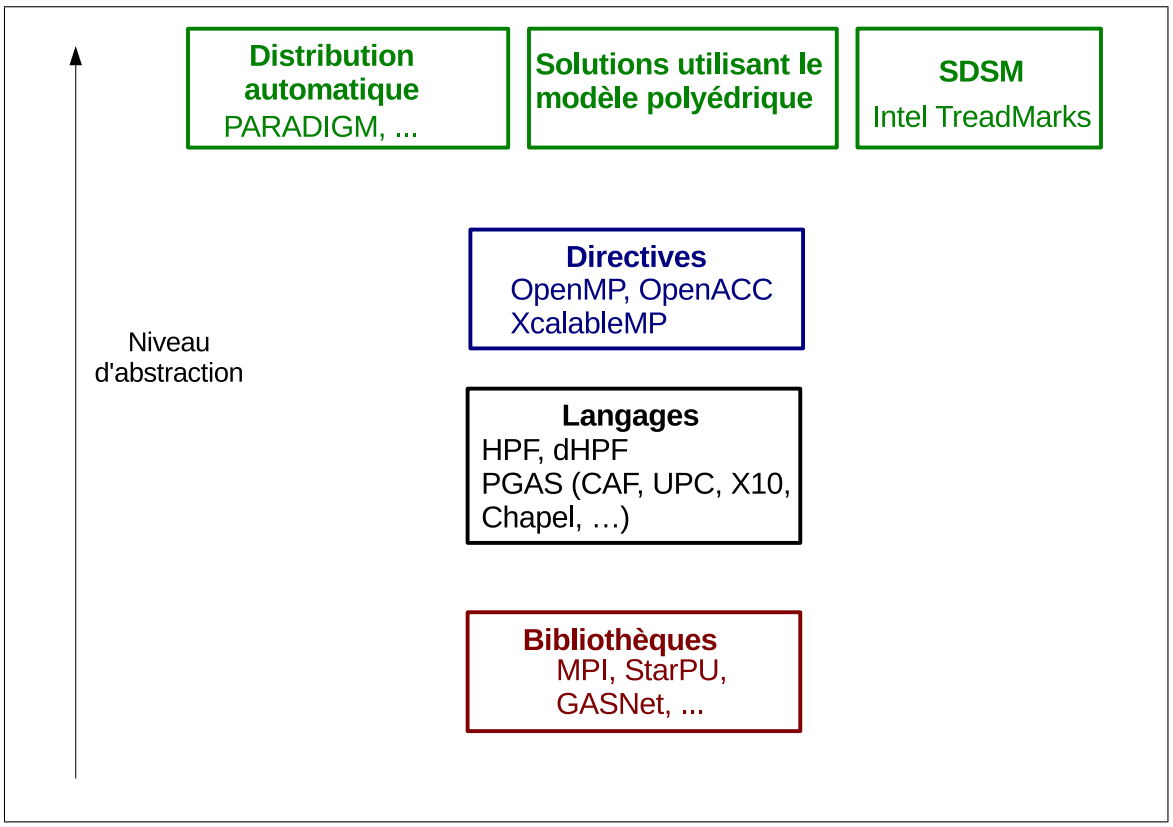


FIGURE 2.1 – Classification des différentes approches de la programmation pour machines parallèles hybrides

Les routines de base implémentées dans ce type de communication sont l'envoi de données (*put*), la récupération de données (*get*) et l'accumulation du résultat d'un calcul (*accumulate*).

Les communications unilatérales nécessitent un support matériel spécifique et peuvent introduire des surcoûts importants à l'exécution dûs au mécanisme de complétion et de notification de complétion des communications. Pour les applications à parallélisme de données, le paradigme des communications coopératives reste la solution la plus largement utilisée. Les communications unilatérales sont plus adaptées aux applications irrégulières et constituent le support de communication des langages PGAS (voir la section 2.3).

2.1.2 MPI

MPI [67] est un standard pour la programmation à échange de messages en C/C++ et Fortran. Il offre une riche collection de primitives de communication point-à-point et collectives. Depuis la version 2.1 du standard, les communications unilatérales sont supportées. MPI définit des communicateurs, ensembles de processus participant à l'exécution répartie d'un programme. Les communicateurs peuvent être des sous-ensembles des processus disponibles, permettant ainsi un découpage

hiérarchique des traitements selon les sous-domaines de l'application. Les processus peuvent également être structurés virtuellement en des topologies cartésiennes pour simplifier la conception des communications pour des problèmes à voisinage. Enfin, MPI offre un support pour les entrées-sorties parallèles.

MPI est une spécification d'interface pour la programmation à échange de message dont il existe plusieurs implémentations comme MPICH-2 [47] et OpenMPI [45]. MPI est largement adopté par la communauté du calcul scientifique pour la programmation des architectures à mémoire distribuée. Cependant, il requiert un effort de programmation élevé à un niveau de détail très fin. MPI est, de ce fait, qualifié de langage assembleur de la mémoire distribuée.

Un programme MPI peut être exécuté aussi bien en mémoire distribuée qu'en mémoire partagée. Cependant MPI n'est pas utilisé comme solution de choix en mémoire partagée car il demande un effort de programmation beaucoup plus important par rapport à une solution plus adaptée comme OpenMP par exemple (voir section 2.2.1). De plus, d'un point de vue système, les processus MPI sont des entités plus lourdes que les threads utilisés dans les modèles de programmation pour mémoire partagée. Enfin, MPI n'offre pas de support pour la programmation des accélérateurs.

2.1.3 GASNet

La bibliothèque de communication GASNet [18] offre une implémentation efficace des communications unilatérales. Elle constitue un support d'exécution pour les langages PGAS [97] (*Partitioned Global Address Space*) plutôt qu'une interface directement utilisée par le programmeur applicatif. GASNet utilise les *messages actifs* [94](*active messages*). Un message actif contient, en plus des données, un entête avec une adresse d'un code, voire directement un code, à exécuter par le processus récepteur pour extraire les données du message.

2.1.4 Global Arrays

Global Arrays [74] est une des premières tentatives d'adaptation du modèle de programmation à mémoire partagée pour les mémoires distribuées. L'interface *Global Arrays* donne au programmeur un contrôle explicite sur les données : distinction entre des données locales, à accès rapide, et des données partagées, à accès plus lent. Le programmeur peut créer des tableaux distribués, et effectuer des accès asynchrones sur ces tableaux sans coopération explicite entre les processus. Il doit ainsi insérer les primitives de synchronisation nécessaires afin de garantir la correction des accès concurrents.

2.1.5 StarPU

StarPU [9] est un modèle de programmation à parallélisme de tâches pour la programmation hybride. Le programmeur définit des *codelets* : des fonctions qui peuvent être implémentées sur plusieurs types d'architectures (CPU, GPU, CELL, ...). L'application d'un codelet sur un ensemble de données constitue une tâche, pour laquelle

les accès aux données (lecture, écriture ou lecture et écriture) sont explicitées. Le support d'exécution de StarPU ordonnance automatiquement et efficacement ces tâches sur les ressources de calcul disponibles en gérant les mouvements de données entre les différentes mémoires d'une architecture hétérogène.

En mémoire distribuée, StarPU offre un ensemble de fonctions de communication au dessus de MPI pour effectuer des transferts explicites de données [8]. StarPU permet également de générer des communications de façon transparente en mémoire distribuée. Pour cela, le programmeur doit définir la distribution des données, et soumettre au support d'exécution des `MPI tasks`. La distribution est effectuée via des *handles*, qui sont des structures de données permettant de gérer la cohérence des variables distribués. De plus, la distribution est spécifiée par une fonction définie par le programmeur et appelée pour chaque élément des données à distribuer. Cette stratégie peut induire un surcoût important lors de la distribution de données de très grande taille.

Le listing 2.1 tiré de la documentation en ligne de StarPU montre un exemple de distribution de données. On remarque que la matrice `matrix` est répliquée sur tous les nœuds (ligne 1). On déclare ensuite pour la structure de données `data_handles`, élément par élément, un propriétaire (ligne 8). L'utilisateur définit également la fonction de distribution comme dans le code du listing 2.2 pour une distribution par blocs.

2.1.6 Bilan des bibliothèques

Le tableau 2.1 résume le bilan des bibliothèques de communication utilisées pour la programmation des architectures hybrides.

	MD	MP	Acc.	Facilité	Dist. données	Dist. calculs	Comm.	Calcul	Parallélsime	Portabilité
MPI	✓	✓			programmeur	programmeur	explicites	dense et creux	données	✓
GASNet	✓	✓			programmeur	programmeur	explicites	dense et creux	données	
GA	✓	✓			programmeur	programmeur	explicites	dense et creux	données	
StarPU	✓	✓	✓		programmeur	programmeur	explicites	dense et creux	tâches	✓

Table 2.1 – Bilan des bibliothèques (MD : Mémoire Distribuée, MP : Mémoire Partagée, Acc. : Accélérateurs, Dist. : Distribution, Comm. : Communications)

2.2 Directives de compilation

2.2.1 OpenMP

OpenMP [31] est un standard de fait pour la programmation des architectures à mémoire partagée en C/C++ et Fortran. Il utilise un modèle de programmation basé sur les processus légers, ou *threads*, appelé modèle *fork-join* : lorsqu'un thread rencontre le début d'une région parallèle, il crée une équipe de threads, dont il devient le thread principal. À la fin de l'exécution d'une région parallèle, tous les threads d'une même équipe s'arrêtent, sauf le thread principal qui continue son exécution. Des

```
unsigned matrix[X][Y];
starpu_data_handle_t data_handles[X][Y];
for(x = 0; x < X; x++) {
  for (y = 0; y < Y; y++) {
    int mpi_rank = my_distrib(x, y, size);
    if (mpi_rank == my_rank)
      /* Owning data */
      starpu_variable_data_register(&data_handles[x][y], 0,
                                    (uintptr_t)&(matrix[x][y]),
                                    sizeof(unsigned));
    else if (my_rank == my_distrib(x+1, y, size) ||
             my_rank == my_distrib(x-1, y, size) ||
             my_rank == my_distrib(x, y+1, size) ||
             my_rank == my_distrib(x, y-1, size))
      /* I don't own that index, but will need it for my
    computations */
      starpu_variable_data_register(&data_handles[x][y], -1,
      (uintptr_t)NULL, sizeof(unsigned));
    else
      /* I know it's useless to allocate anything for this */
      data_handles[x][y] = NULL;
      if (data_handles[x][y]) {
        starpu_data_set_rank(data_handles[x][y], mpi_rank);
        starpu_data_set_tag(data_handles[x][y], x*X+y);
      }
  }
}
```

Listing 2.1 – Distribution de données avec StarPU

```
/* Returns the MPI node number where data is */
int my_distrib(int x, int y, int nb_nodes) {
  /* Block distrib */
  return ((int)(x / sqrt(nb_nodes) +
                (y / sqrt(nb_nodes)) *
                sqrt(nb_nodes))) % nb_nodes;
}
```

Listing 2.2 – Définition d'une fonction de distribution par le programmeur dans StarPU

constructions de partage de travail permettent de spécifier que certaines instructions peuvent être exécutées en parallèle par les différents threads d'une équipe. OpenMP offre un modèle mémoire à *consistance relâchée*, où les threads exécutant une région parallèle ont une vue locale de l'état des variables accédées au niveau des cœurs de calcul, ce qui permet d'avoir des lectures rapides et des écritures non propagées aux autres cœurs lors de l'exécution d'une région parallèle. La cohérence globale, c'est à dire la vision unifiée des valeurs des variables partagées par tous les threads, est rétablie aux points de synchronisation. Il existe des synchronisations implicites en

début et fin des régions parallèles et explicites en utilisant les barrières et la directive *flush*.

OpenMP est particulièrement adapté à la parallélisation d'applications à parallélisme de données. La directive `omp for` permet de réaliser un partage de travail au niveau des itérations d'une boucle parallèle avec la possibilité d'indiquer plusieurs types de partitionnement et d'affectation des itérations aux différents threads. De plus, OpenMP permet de suivre une approche de parallélisation incrémentale en parallélisant séparément les boucles d'un programme.

La version 3.0 [77] du standard OpenMP introduit un support pour le parallélisme de tâches et un contrôle à l'exécution de l'affectation des threads aux cœurs. La version 4.0 [78] élargit les architectures cibles d'OpenMP avec l'introduction d'un ensemble de directives pour l'exécution sur accélérateurs. La directive `omp target` permet de déclarer un environnement d'exécution sur un accélérateur. La directive `omp declare target` permet de déclarer des fonctions ou des variables pour lesquelles le compilateur OpenMP génère des versions pour GPUs. La clause `map` permet de spécifier le mouvement de données entre un hôte et un accélérateur. La version 4.9.0 de GCC [1], sortie en avril 2014, annonce l'implémentation des fonctionnalité d'OpenMP 4.0.

Bien qu'OpenMP offre une interface de programmation intuitive et incrémentale pour la parallélisation de programmes, il reste limité à la mémoire partagée et plus récemment aux accélérateurs et n'offre pas de support pour la programmation des architectures à mémoire distribuée. Des travaux se sont intéressés à l'extension d'OpenMP pour les clusters comme le projet *Distributed OpenMP* de PGI [66]. Intel a proposé *Cluster OpenMP* [53], une extension d'OpenMP pour les clusters qui ajoute la directive *sharable* à OpenMP pour déclarer les variables allouées en mémoire distribuée et dont les mises à jour sont gérées par un SDSM (*Software Distributed Shared Memory*). Basumalik *et al.* [14] proposent une transformation de programmes OpenMP vers MPI avec réplication totale des données.

2.2.2 STEP : *Système de Transformation pour l'Exécution Parallèle*

L'outil STEP [68, 69], développé par l'équipe HP2 (*Haute Performance et Parallélisme*) de TELECOM SudParis constitue le point de départ du travail de cette thèse. STEP permet la transformation d'un programme OpenMP C ou Fortran en un programme OpenMP et MPI. Il a été étendu à la génération de code pour clusters de GPUs en implémentant une transformation de programmes HMPP vers HMPP et MPI [85]. STEP permet de générer des communications efficaces grâce à l'utilisation des analyses interprocédurales de régions de tableaux [30] offertes par le compilateur PIPS [55, 3]. L'efficacité de STEP a été démontrée sur plusieurs programmes de calcul scientifique tels que la multiplication de matrices, le solveur Jacobi et le programme de dynamique moléculaire MD. STEP a été également appliqué avec succès à une application de propagation d'ondes où des réductions significatives des communica-

1. http ://gcc.gnu.org/gcc-4.9/changes.html

tions ont été réalisées grâce aux régions décrivant les communications et maintenues dynamiquement : *send*, *recv* et *up-to-date*.

Au début de ce travail de thèse, la limitation principale de STEP était la réplication intégrale des tableaux dans le code généré, ce qui empêche l'exécution des programmes générés sur des données de très grande taille. Cette limitation est d'autant plus importante dans le contexte des clusters de GPUs que la mémoire d'un seul GPU ne peut être étendue contrairement à la mémoire centrale d'un CPU.

2.2.3 HMPP

HMPP(*Hybrid Multicore Parallel Programming*) [34] est un modèle de programmation pour accélérateurs développé par CAPS Entreprise[2]. C'est un ensemble de directives pour la spécification des parties de code à exécuter sur accélérateur. Deux directives principales sont utilisées en HMPP : la directive *codelet* permet d'indiquer qu'une fonction est à exécuter sur un accélérateur, et la directive *callsite* permet d'indiquer les sites d'appels dans le code où la version accélérateur du code d'une fonction doit être appelée. D'autres directives permettent de spécifier la politique de transfert de données entre la mémoire d'un hôte et celle d'un accélérateur.

Pour optimiser la génération de code pour les codelets, HMPP offre un autre ensemble de directives, désigné comme *HMPP Codelet Generator* [24]. La directive *hmppcg gridify* notamment permet de spécifier plusieurs dimensions parallèles d'un nid de boucles. D'autres directives permettent d'indiquer au compilateur des transformations à appliquer aux nids de boucles annotés telles que la permutation ou la distribution de boucles.

Le listing 2.3 montre un exemple d'utilisation de la directive *hmppcg gridify* pour exprimer qu'une boucle i est parallèle, avec indication des variables privées et globales de la boucle dans des clauses.

```
1 #pragma hmppcg gridify, private (mylocalvar), global (v1, v2, alpha)
2
3  for( i = 0 ; i < n ; i++ ) {
4    v1[i] = alpha * v2[i] + v1[i];
5    mylocalvar= v1[i];
6
7  }
```

Listing 2.3 – Exemple d'utilisation de la directive hmppcg gridify

Comme OpenMP, HMPP offre une solution de parallélisation incrémentale, qui permet de porter progressivement plusieurs parties d'un programme sur accélérateurs. La spécification HMPP a donné naissance à la proposition d'un standard ouvert : OpenHMPP [38]. HMPP ne traite cependant pas la programmation des machines à mémoire distribuée et à mémoire partagée.

2. http ://www.caps-entreprise.com

2.2.4 OpenACC

OpenACC [29, 95] est un standard récent pour la programmation d'accélérateurs en C/C++ et Fortran élaboré notamment par CAPS Entreprise, PGI et Cray. L'objectif d'OpenACC est de devenir pour les accélérateurs ce qu'OpenMP est pour les machines à mémoire partagée en termes de portabilité, performances et d'adoption par la communauté du calcul haute performance. OpenACC offre un ensemble de directives pour la spécification des parties d'un programme ou des fonctions à exécuter sur un accélérateur ainsi que les mouvements de données entre les mémoires de l'hôte et de l'accélérateur. OpenACC reste cependant spécifique aux accélérateurs et ne cible ni la mémoire distribuée ni la mémoire partagée.

2.2.5 hiCUDA

hiCUDA [50] est un modèle de programmation à base de directives pour la simplification de la programmation en CUDA. Le code à exporter sur GPU est délimité par la directive *kernel* et le découpage des itérations par la clause *loop_partition*. Le modèle mémoire de hiCUDA reflète l'organisation mémoire de l'architecture CUDA. Il propose les directives *global*, *constant*, *texture* et *shared* pour déclarer des variables allouées dans la mémoire du même nom dans le modèle CUDA. Les transferts mémoire sont à la charge du programmeur en utilisant les clause *copyin* et *copyout* pour les variables référencées dans les kernels.

2.2.6 Cetus

Basumalik *et al.* [15] proposent une transformation OpenMP vers MPI implémentée dans le compilateur Cetus [60]. Dans cette solution, les données sont totalement répliquées. Lee *et al.* [61] étendent ce travail à une transformation OpenMP vers CUDA où les transferts mémoire sont générés automatiquement et sont optimisés grâce à une analyse des définitions/utilisations des données entre l'hôte et l'accélérateur.

2.2.7 XcalableMP

XcalableMP [73] est un langage de programmation parallèle, influencé par OpenMP et HPF. Il offre aux programmeurs deux vues sur les objets distribués : une vue locale, compatible avec le langage CAF, et une vue globale consistant en un ensemble de directives. Les directives XcalableMP peuvent être insérées dans des programmes écrits en langage C et Fortran et permettent au programmeur d'indiquer au compilateur la distribution des données et des calculs, mais ce dernier doit également expliciter les endroits auxquels les communications doivent avoir lieu et sur quelles données. La distribution des tableaux est effectuée avec des directives similaires à HPF, exprimant des distributions par blocs et cycliques ainsi que par blocs de différentes tailles. Un voisinage peut être indiqué pour les données distribuées avec une directive *shadow*. Les boucles sont partitionnées avec la directive *loop* et

la clause *threads* est utilisée pour indiquer la génération de code pour mémoire partagée. La directive *gmove* est utilisée pour des instructions d'affectation utilisant une syntaxe de sections de tableaux. Dans ce cas, si les éléments lus se trouvent sur des processus distants, alors des communications sont générées. La directive *reflect* indique au compilateur qu'il faut mettre à jour les éléments des zones *shadow*.

XcalableMP offre au programmeur la possibilité de contrôler à la fois la distribution des données et des calculs mais lui impose de gérer explicitement les communications par des directives. Aucun moyen de vérification des communications ainsi décrites n'est offert, et le programme peut produire un résultat incorrect si les directives de communications ne sont pas bien utilisées. De plus, XcalableMP n'offre pas de support pour la distribution multi-partitionnée et ne sépare pas les notions de boucle parallèle et distribuée.

2.2.8 OmpSs

OmpSs [36] est une solution de programmation parallèle aynchrone à base de directives. La directive *task* permet de déclarer une fonction ou une partie de code comme étant une tâche pouvant s'exécuter en parallèle. À une tâche sont associées des informations sur les accès aux données via les clauses *in*, *out* et *inout*. Le compilateur d'OmpSs génère ensuite un code parallèle dans lequel les tâches sont éligibles à l'exécution dès que leurs dépendances sont satisfaites. OmpSs peut générer du code pour plusieurs architectures spécifiées par la directive *target*. Le support d'exécution d'OmpSs, *Nanos++*, est responsable de l'ordonnancement des tâches ainsi que de la cohérence mémoire du programme. Pour une exécution distribuée [22], Nanos++ implémente une gestion centralisée des tâches ainsi que des communications. Une instance du support d'exécution est lancée sur chaque nœud et une instance principale lance et termine les tâches et toutes les communications entre les différents nœuds passent par cette instance, constituant ainsi un goulot d'étranglement. De plus, toute la mémoire globale du programme est allouée sur le nœud central, ce qui limite la capacité mémoire des programmes ainsi parallélisés. OmpSs a ajouté par la suite le concept de régions [23] pour spécifier de façon plus fine des parties d'éléments de tableaux au niveau des clauses de dépendance et d'utilisation des données. Les régions ainsi décrites peuvent notamment se chevaucher. Les clauses d'utilisation de données, `copy_in`, `copy_out` et `copy_inout` permettent de spécifier le mouvement des données et de générer des communications dans un contexte distribué. Une limitation importante d'OmpSs en mémoire distribuée est la réplication de toutes les données sur tous les nœuds. Le programmeur est également responsable de la description précise des accès aux données afin que le compilateur génère des communications correctes, ce qui constitue un effort de programmation élevé.

2.2.9 Bilan des solutions à base de directives

Le tableau 2.2 résume le bilan des approches par directives pour la programmation des architectures hybrides. Les solutions existantes ne traitent pas tous les niveaux de l'architecture parallèle et des solutions telles que STEP, Cetus ou OmpSs étendent leurs modèles de programmation aux mémoires distribuées. Cependant, ces solutions répliquent totalement les données du programme, constituant ainsi une limitation aux traitement de données de très grande taille.

	MD	MP	Acc.	Facilité	Dist. données	Dist. calculs	Comm.	Calcul	Parallélsime	Portabilité
OpenMP		✓	✓(4.0)	✓		programmeur		dense creux	données tâches (3.0)	✓
STEP	✓	✓	✓	✓		programmeur	implicites	dense	données	✓
HMPP			✓			programmeur	implcites explicites	dense	données	
OpenACC			✓	✓		programmeur	implicites explicites	dense	données	
hiCUDA			✓	✓		programmeur	explicites	dense		
Cetus	✓	✓	✓	✓		programmeur compilateur	implicites	dense	données	✓
XcalableMP	✓	✓			programmeur	programmeur	explicites	dense	données	
OmpSs	✓	✓	✓			programmeur	explicites	dense creux	tâches	

Table 2.2 – Bilan des solutions à base de directives (MD : Mémoire Distribuée, MP : Mémoire Partagée, Acc. : Accélérateurs, Dist. : Distribution, Comm. : Communications)

2.3 Langages de programmation

2.3.1 High Performance Fortran

Le langage HPF [51] (*High Performance Fortran*) a été conçu dans les années 90 avec l'ambition de devenir un standard pour la programmation des machines parallèles à mémoire distribuée. Construit au dessus du langage Fortran 95, HPF a ainsi pour objectif d'offrir une solution de choix à la fois pour la parallélisation incrémentale des codes de calcul scientifique existants et pour l'écriture de nouvelles applications parallèles. Bien que HPF soit un langage à part entière, on s'intéresse à sa partie directives, leur expressivité, leur compilation et leur impact sur les performances du code généré.

Les directives HPF sont des indications au compilateur sur la façon d'implémenter le programme sur une machine à mémoire distribuée et n'affectent pas sa sémantique. Deux types d'indications sont notamment données au compilateur : la façon de distribuer les données et le parallélisme potentiel des boucles référençant les données distribuées.

Le modèle global de distribution des données avec HPF est décrit dans la figure 2.2. Les éléments de tableaux peuvent être alignés les uns aux autres avec une directive

ALIGN et plusieurs tableaux peuvent être alignés au même tableau virtuel, appelé *template*. Les tableaux, parfois via des templates, sont ensuite distribués avec la directive DISTRIBUTE sur un ensemble de processeurs déclarés avec la directive PROCESSORS.

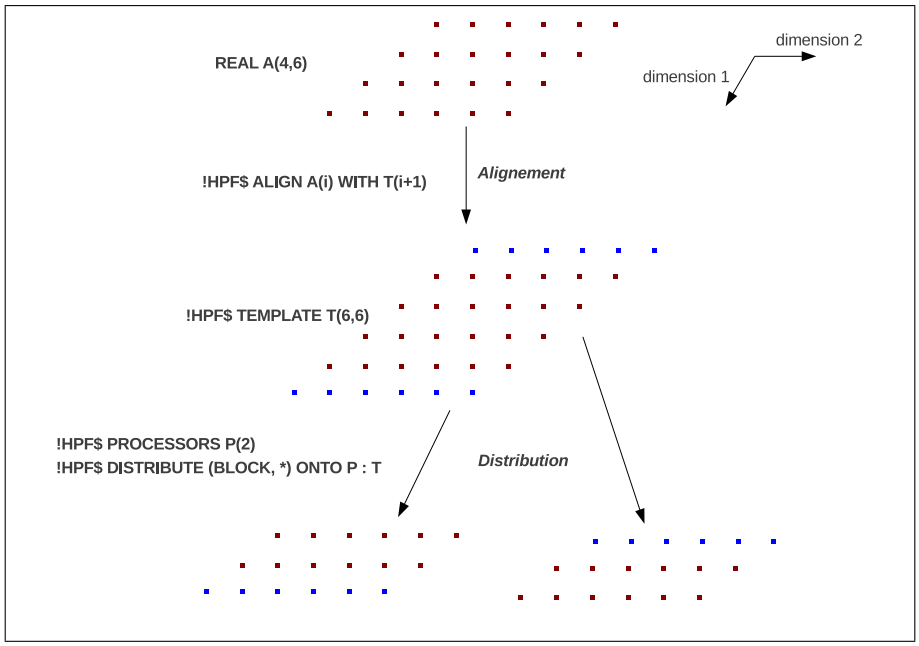


FIGURE 2.2 – Distribution des données avec HPF

HPF offre au programmeur le moyen d'indiquer au compilateur quelles itérations de boucles peuvent être exécutées en parallèle. La directive INDEPENDENT indique que les instructions d'une boucle ne comportent pas de dépendances entre les itérations. Une clause de réduction, REDUCTION, peut être ajoutée à une boucle parallèle pour déclarer une variable pour laquelle le compilateur doit implémenter une réduction parallèle. La clause NEW permet d'indiquer des variables pouvant être privatisées, une opération utile notamment pour la gestion des tableaux temporaires aux nids de boucles. La construction de boucles FORALL, passée dans le langage Fortran 95, permet de regrouper plusieurs indices de boucles. Couplée avec la directive INDEPENDENT, plusieurs dimensions d'un nid de boucles peuvent ainsi être déclarées parallèles. La construction FORALL évalue tous les membres droits des expressions d'affectation avant d'écrire les membres gauches.

Les compilateurs HPF interprètent les directives de distribution pour générer un programme parallèle dans lequel chaque processeur est *propriétaire* d'une partie des données distribuées, c'est-à-dire que les données sont alloués uniquement sur ce processeur. Cependant, aucune indication n'est donnée au compilateur sur la façon de répartir les itérations sur les différents processeurs. La plupart des compilateurs HPF utilisent une heuristique connue sous le nom de *owner-computes*. Cette règle stipule que le calcul d'une expression est effectué par le propriétaire de la donnée où le

résultat est stocké. Des communications doivent être générées afin de récupérer tous les éléments nécessaires au calcul et dont le processeur actif n'est pas propriétaire. Cependant, cette règle n'est pas toujours suffisante pour répartir les itérations. Un contre-exemple évident est la présence pour une même itération de plusieurs instances d'instructions écrivant des éléments de différents tableaux. Par exemple, dans le code suivant, il n'est pas évident de déterminer quel processeur doit exécuter une itération `i` si les propriétaires des éléments `A(i)` et `B(i-1)` ne sont pas le même processeur.

```
DO i=1, N
  A(i) = f(i)
  B(i-1) = g(i)
END DO
```

L'extension `ON HOME` de HPF permet de lever la contrainte *owner-computes* et de généraliser la définition des processeurs devant exécuter une instruction au propriétaire d'une référence quelconque. Dans l'exemple qui suit, le propriétaire de l'élément `B(i)` exécute l'instruction. Si ce processeur n'est pas propriétaire de l'élément écrit `A(i-1)`, alors une communication doit être générée pour envoyer la nouvelle valeur de cet élément à son propriétaire.

```
!HPF$ ON HOME(B(i))
  A(i-1) = B(i) + C(i)
```

Les distributions régulières par *blocs* et *bloc-cycliques* proposées par HPF permettent de paralléliser des programmes pour lesquels les dimensions parallèles des différents nids de boucles du programme coïncident avec les dimensions distribuées des tableaux accédés. Des schémas d'accès tels que les balayages par lignes, rencontrés dans les applications implémentant des intégrations ADI (Alternate Direct Implicit) présentent un défi pour les distributions HPF de base. En effet, pour ces types d'accès, quelle que soit la dimension distribuée, elle sera a un moment accédée de façon séquentielle par une boucle du programme, alors que l'accès aux autres dimensions est parallèle. Pour pouvoir bénéficier du parallélisme sur toutes les dimensions, HPF propose la directive `REDISTRIBUTE` qui permet de redistribuer dynamiquement les tableaux afin d'avoir, pour chaque nid de boucles, une correspondance entre les dimensions parallèles et les dimensions distribuées des tableaux. La figure 2.3 montre un exemple illustrant ce problème. Si le tableau est distribué par lignes (a), alors le calcul sur la première dimension est séquentialisé et prend quatre temps de calcul en plus des communications induites par les dépendances de données entre les processeurs. Si le tableau est distribué par colonnes, alors le problème symétrique se poserait pour les dépendances sur la deuxième dimension. Si le tableau est distribué sur les deux dimensions, alors il faudrait deux temps de calcul, avec la moitié des processeurs inactifs à chaque temps (b). Enfin, si le tableau est redistribué dynamiquement afin d'exploiter le parallélisme selon la deuxième dimension, alors le calcul se fait en un seul temps (c). Si la dernière solution apporte un meilleur équilibrage de charge, le coût des communications nécessaires à la redistribution peut être prohibitif pour les performances globales du programme. Les mesures de performances des benchmarks

NAS parallélisés avec HPF [43] montrent une perte de performance importante pour les programmes SP et BT à cause du coût important de la redistribution.

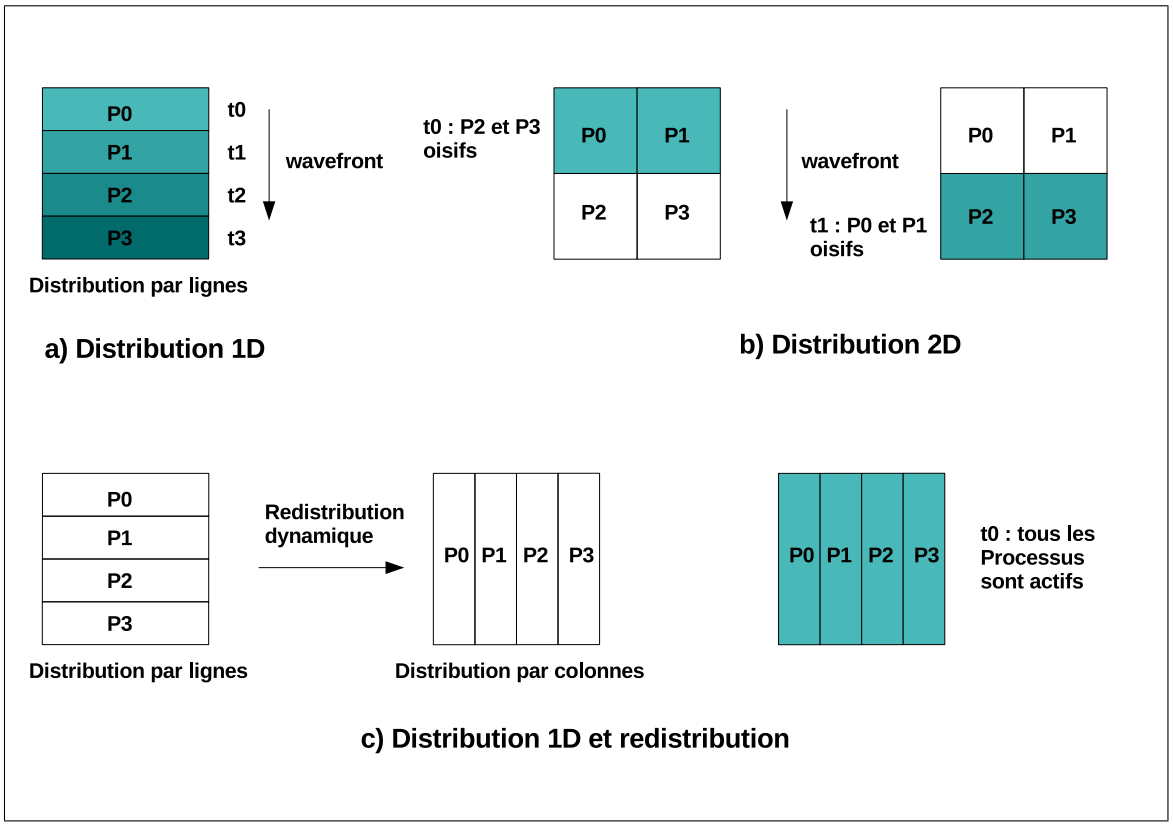


FIGURE 2.3 – Le problème des calculs en front-d'onde en HPF avec différentes stratégies de distribution

2.3.2 Le langage dHPF

Le compilateur dHPF [65] de l'université Rice apporte des améliorations à l'expressivité et à la compilation des directives HPF. Le partitionnement des itérations des nids de boucles sur les processeurs distribués, appelé CP (*Computation Partitioning*) est plus général que la règle *owner-computes*. Il s'agit d'unions de propriétaires de références en lecture ou en écriture, pour chaque instruction. De plus, plusieurs CP indépendants peuvent être définis pour différentes instructions d'un nid de boucles. De façon générale, pour les tableaux A_j référencés par les expressions f_j à l'itération i, l'ensemble CP est définit par :

$$CP : \bigcup_{j=1}^{j=n} \{\text{ON HOME } A_j(f_j(i))\}$$

Une optimisation importante implémentée par dHPF consiste à réduire le nombre des communications en répliquant les calculs définissant des valeurs de tableaux temporaires. Cette optimisation se base sur le contrôle des calculs par le CP. La réplication

partielle des itérations n'est cependant intéressante que si le coût des communications sur les temporaires est plus important que le coût des communications induites par les accès en lecture aux tableaux nécessaires au calcul des éléments partiellement répliqués.

Enfin, dHPF implémente le *multi-partitioning*, une généralisation de la distribution par blocs de HPF introduite dans les *NAS Parallel Benchmarks* [11]. Dans ce type de distribution, quelle que soit la dimension considérée, les données sont distribuées de façon équilibrée sur tous les processeurs. La figure 2.4 montre un exemple d'une telle distribution d'un tableau à deux dimensions sur quatre processeurs (a). Lors du balayage de n'importe quelle dimension (b), dans n'importe quelle direction, tous les processeurs sont actifs en même temps, sans nécessiter de redistribution. La charge est ainsi tout le temps équilibrée entre les différents processeurs.

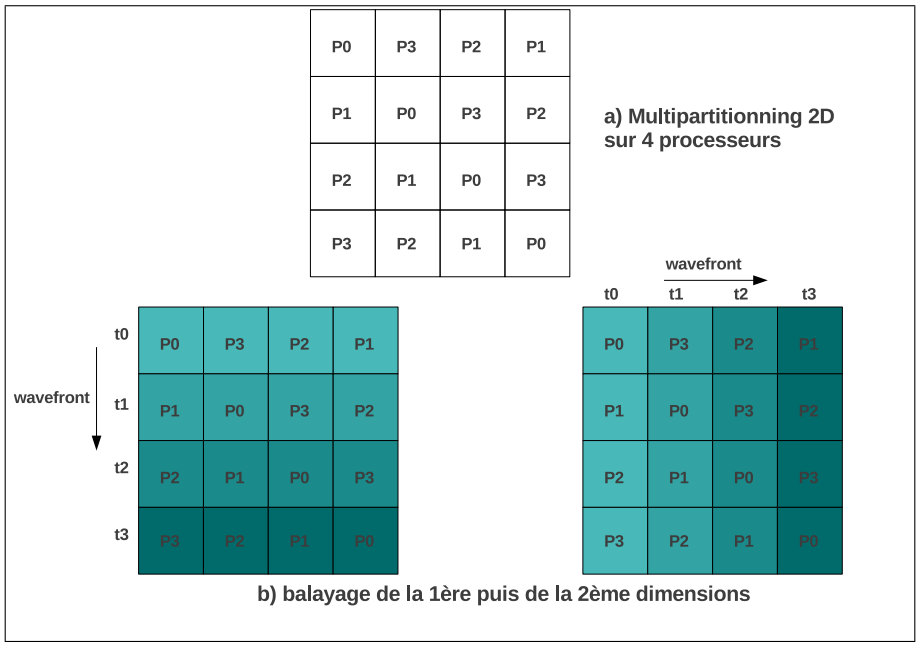


FIGURE 2.4 – dHPF : multi-partitioning et exécution en front d'onde

2.3.3 Forces et faiblesses de HPF et de dHPF

HPF permet de paralléliser, avec un coût de développement raisonnable, des programmes scientifiques réguliers avec un parallélisme uniforme (les mêmes dimensions sont accédées en parallèle d'un nid de boucles à un autre). Les améliorations apportées par dHPF élargissent ce panel d'applications en implémentant le *multi-partitioning* ainsi que d'importantes optimisations telles que la réplication partielle des calculs pour éliminer certaines communications. Cependant, la répartition des itérations avec la règle *owner-computes* ou la généralisation des *CP* dans dHPF conduit à la génération d'un code à contrôle complexe, qu'il peut être difficile à porter sur des architectures de

type GPU par exemple. De plus, le programmeur n'a aucun contrôle sur la répartition des itérations et les communications que cela implique.

2.3.4 PGAS : *Partitioned Global Address Space*

Les PGAS sont une famille de langages de programmation parallèle offrant un modèle de programmation à mémoire globale visible à toutes les unités d'exécution d'un programme.

Co-Array Fortran

Le langage CAF [75](*Co-Array Fortran*) est une extension du Fortran 95 qui cible la communauté de programmeurs Fortran avec une approche différente de celle de HPF. La notion fondamentale de CAF est la co-dimension : le programmeur peut déclarer sur un processeur des éléments supplémentaires comme une colonne ou une ligne résidant sur un processeur distant. Ces éléments sont ensuite accédés de façon transparente. Cependant, des constructions de synchronisation doivent être soigneusement utilisées afin de maintenir ces éléments dans un état mémoire globalement cohérent. Le langage CAF 2.0 [64] de l'université Rice est une nouvelle version de CAF qui introduit des notions de bas niveau telles que les sous-ensembles de processeurs et la topologie.

Unified Parallel C

UPC [90] (Unified Parallel C) est une extension du langage C offrant un modèle de programmation à mémoire distribuée partagée. Une variable en UPC est déclarée visible par tous les `threads` dans l'espace global par l'attribut `shared`. Le partitionnement des itérations d'un nid de boucles est spécifié par la construction `upc_for` avec un paramètre d'affinité assignant les itérations aux threads. Les tableaux partagés sont distribués sur les threads avec les distributions classiques par blocs et bloc-cycliques. Le modèle mémoire offre deux niveaux de cohérence : stricte et relâchée. Des routines de synchronisation, comme `upc_barrier` sont utilisées pour garantir la cohérence des variables partagées. Des extensions ont été apportées à UPC pour programmer les clusters de GPU [27].

Chapel

Le langage Chapel [25], développé par Cray, utilise le concept de *locale* pour abstraire les unités de la machine parallèle capables d'exécuter des tâches et de stocker des variables. Des domaines d'indices sont utilisés pour déclarer des tableaux. Les domaines sont ensuite distribués sur les *locales* selon différents types de distribution. Les boucles parallèles sont exprimées par la construction `forall` et le partitionnement des itérations suit la distribution des indices des domaines mais peut être contrôlé par la clause `on`. Le concept de *halo* [33] est introduit pour étendre l'espace de stockage d'un tableau afin de répliquer les éléments du voisinage et les rendre visible et modifiables explicitement sur chaque locale.

X10

X10 [26] est un langage PGAS orienté-objet, basé sur le langage Java. Des ensembles de points entiers, appelés régions, sont définis et auxquels peuvent être associés des éléments de tableaux. Les régions sont ensuite distribuées sur les différentes unités de mémoire, appelées *places* avec des distributions classiques ou via des distributions définies par l'utilisateur. Les différentes tâches du programme sont appelées des *activités*. Le modèle mémoire de X10 est *Localement Synchrone et Globalement Asynchrone* : une activité accède de façon synchrone aux données de sa propre *place*, mais pour accéder aux données stockées sur d'autres *places*, des activités asynchrones sont créées sur les places distantes. Enfin, les itérateurs `foreach` et `ateach` permettent de parcourir les domaines d'une ou de plusieurs *places*.

2.3.5 SPOC

Bourgoin *et al.* [19, 20] proposent SPOC, une solution de programmation de haut niveau pour GPGPUs utilisant le langage OCaml. SPOC définit un type *vector* générique pour stocker les données de grande taille. Les transferts mémoire entre le CPU hôte et le GPGPU sont gérés automatiquement par le support d'exécution de SPOC. SPOC réutilise le *Garbage Collector* d'OCaml afin de libérer la mémoire occupée par les données qui ne sont plus utilisées, évitant ainsi de saturer la mémoire du GPU. Les auteurs démontrent l'applicabilité de leur solution à la fois sur des petits programmes de calcul scientifique mais également sur une grande application de physique atomique sur laquelle ils atteignent plus de 80% des performances de la version Fortran CUDA écrite à la main [42], tout en réduisant considérablement l'effort de programmation requis.

2.3.6 Bilan des langages de programmation

HPF et dHPF permettent de simplifier la programmation des architectures à mémoire distribuée mais ne traitent pas des aspects hybrides en présence de processeurs multi-cœurs et des accélérateurs. Bien que les langages PGAS soient très expressifs pour la programmation des machines parallèles à mémoire distribuée, ils présentent beaucoup de détails de bas niveau et des mécanismes de synchronisation complexes qui requièrent un effort important de la part du programmeur. Le tableau 2.3 résume le bilan des langages de programmation utilisés pour la programmation des architectures hybrides.

	MD	MP	Acc.	Facilité	Dist. données	Dist. calculs	Comm.	Calcul	Parallélsime	Portabilité
HPF, dHPF	✓			✓	programmeur	compilateur	implicites	dense creux	données	
CAF	✓	✓			programmeur	programmeur	implicites	dense creux	données	
UPC	✓	✓	✓ (extensions)		programmeur	programmeur	implicites	dense creux	données tâches	
Chapel	✓	✓			programmeur	programmeur	implicites	dense creux	données tâches	
X10	✓	✓			programmeur	programmeur	implicites	dense creux	données tâches	✓
SPOC			✓	✓		programmeur	implicites	dense	données	✓

TABLE 2.3 – Bilan des langages de programmation (MD : Mémoire Distribuée, MP : Mémoire Partagée, Acc. : Accélérateurs, Dist. : Distribution, Comm. : Communications)

2.4 Approches de haut niveau

Les approches de haut niveau cachent les détails de l'architecture au programmeur. Nous nous intéressons aux approches de haut niveau pour la dimension mémoire distribuée de la programmation des architectures hybrides car la distribution des données constitue l'aspect le plus difficile dans ce contexte. À ce niveau d'abstraction, la distribution des données et les communications sont complètement transparentes pour le programmeur, même si elles impactent les performances.

2.4.1 Distribution automatique

Les solutions automatiques décomposent le problème de la distribution en deux sous-problèmes : l'alignement des tableaux les uns par rapport aux autres et la distribution des tableaux alignés sur les processeurs disponibles. Li et Chen [63] ont montré que trouver un alignement optimal est un problème NP-complet. Au delà de la difficulté de décider automatiquement d'un alignement et d'une distribution, le problème de la génération d'un code distribué efficace reste posé. Les défis sont notamment le découpage efficace des espaces d'itération afin d'avoir un bon équilibrage de charge ainsi que la génération de communications efficaces. Le compilateur PARADIGM [12] est un exemple de solution de génération automatique de code pour mémoire distribuée.

2.4.2 Génération de code pour machines à mémoire distribuée en utilisant le modèle polyédrique

Les techniques de génération de code dans le modèle polyédrique ont été étendues par Yuki *et al.* [98] pour la génération de code pour mémoire distribuée. Les auteurs utilisent le tuilage paramétrique des nids de boucles [56] étendu à la mémoire distribuée. La mémoire référencée par chaque tuile est allouée, avec une extension, nommée *halo*, qui couvre les déplacements dans les expressions d'accès aux données

sur chaque dimension. Cette solution utilise la puissance du modèle polyédrique pour les transformations et la génération de code mais reste limitée aux codes à contrôle statique [40], avec une limitation aux frontières des appels de fonctions. De plus, si la méthode est détaillée pour un nid de boucles, la gestion de la mémoire et du parallélisme à travers plusieurs nids de boucles successifs n'est pas explicitée.

2.4.3 Mémoire distribuée partagée

La mémoire distribuée partagée émule une mémoire partagée au dessus d'un système à mémoire distribuée. Lorsque l'implémentation est logicielle, on parle de SDSM (*Software Distributed Shared Memory*).

Un exemple de SDSM est la technologie TreadMarks [6] qui implémente un modèle mémoire à cohérence relâchée où plusieurs écrivains peuvent simultanément accéder à la même page mémoire et où les mises à jours des pages répliquées n'ont lieu qu'aux points de synchronisation du programme. En dépit de plusieurs améliorations techniques, les SDSM souffrent d'un surcoût important à l'exécution en raison de la granularité des échanges mémoire fixés à la taille d'une page. Le *false sharing* implique la communication de pages allouées pour plusieurs variables dès la modification de l'une d'entre elles.

2.4.4 Bilan des approches de haut niveau

Les approches de haut niveau, si elles ont pour ambition de cacher complètement le placement des données ainsi que les communications au programmeur, ne permettent pas d'indiquer au compilateur des informations pourtant connues du programmeur. Il est également à constater que les approches complètement automatiques ne sont pas largement adoptées en parallélisation pour mémoire distribuée. En effet, elles ne réussissent pas toujours à obtenir des informations suffisamment précises pour être exploitées pour la distribution de données pour un programme dans sa globalité. Le tableau 2.4 résume le bilan des approches de haut niveau pour la programmation des architectures hybrides.

	MD	MP	Acc.	Facilité	Dist. données	Dist. calculs	Comm.	Calcul	Parallélsime	Portabilité
SDSM	✓			✓	système	programmeur	implicites	dense creux	données tâches	
Distribution automatique	✓			✓	compilateur	compilateur	implicites	dense	données	
Modèle polyédrique	✓	✓	✓	✓		compilateur	implicites	dense	données	

Table 2.4 – Bilan des approches de haut niveau (MD : Mémoire Distribuée, MP : Mémoire Partagée, Acc. : Accélérateurs, Dist. : Distribution, Comm. : Communications)

2.5 Conclusion

Dans ce chapitre, nous avons classifié les solutions existantes pour la programmation des architectures parallèles hybrides selon le niveau d'abstraction par rapport à l'architecture cible. La liste des solutions étudiées n'est bien sûre pas exhaustive, mais est représentative des problèmes posés à chaque niveau d'abstraction et des solutions envisagées pour les résoudre.

Aucune des approches et solutions existantes n'apporte de solution globale au problème de la programmation des architectures hybrides selon les critères énumérés dans les bilans de comparaison. Les bibliothèques de bas niveau telles que MPI et GASNet permettent de cibler la mémoire distribuée à un niveau de détail très fin mais demandent en conséquence un effort de programmation et de déboggage élevés, tout en restant restreintes aux mémoire distribuées. Les PGAS incorporent les fonctionnalités nécessaires à la programmation des mémoires distribuées directement au niveau du langage, et offrent pour certains un support pour la mémoire partagée et les GPUs. Cependant, ces langages, qui ne sont pas des standards, demandent un effort de programmation important à l'utilisateur qui doit notamment gérer les synchronisations des accès aux données distribuées afin de garantir leur cohérence. Les approches de haut niveau se heurtent à des problèmes de décidabilité pour la distribution automatique et à des problèmes de prédictibilité des performances pour les SDSM. Les approches de haut niveau basées sur le modèle polyédrique implémentent des analyses mathématiques sophistiquées pour l'extraction du parallélisme et la transformation de code pour les architectures hybrides. Cependant, elles restent limitées à des codes à contrôle statique et la gestion de la distribution des données pour la globalité d'un programme ne semble pas encore effective dans ce modèle.

Nous pensons qu'une approche à base de directives pour la programmation des architectures hybrides est à la fois pratique et viable. En effet, en se basant sur un langage de base standardisé et largement utilisé par la communauté du calcul scientifique, les codes existants peuvent être portés sans réécriture complète. Des applications nouvelles peuvent également être écrites dans cette solution sans avoir à apprendre un nouveau langage ou les fonctionalités d'une nouvelle bibliothèque. Enfin, l'insertion de directives permet de maintenir un seul programme source, et de pouvoir le compiler vers plusieurs types de cible dès que les informations essentielles telles que le parallélisme et le placement des données sont uniformément exprimés. Il est à noter qu'une solution à base de directives n'exclut pas l'interaction avec d'autres approches, les directives pouvant en effet provenir d'une autre solution telle que le modèle polyédrique pour l'extraction du parallélisme.

Nous avons montré que les approches existantes à base de directives ne proposaient pas une solution unifiée pour la programmation des machines parallèles hybrides. Dans le chapitre suivant nous proposons *dSTEP*, une solution à base de directives pour la programmation des architectures hybrides qui a pour objectif d'offrir un modèle de programmation unifié pour les architectures hybride, l'efficacité et la facilité de programmation.

Chapitre 3

dSTEP : directives pour la distribution des calculs et des données

Dans ce chapitre, nous proposons *dSTEP*, une solution à base de directives pour la programmation des architectures parallèles hybrides. Nous avons trois objectifs :

1. l'unification de la programmation pour architectures parallèles hybrides dans un seul modèle de programmation, exploitant tous les niveaux de parallélisme offerts par l'architecture de façon transparente pour le programmeur. Le modèle de programmation doit être à la fois expressif et simple à utiliser pour permettre au programmeur, ou à d'autres outils d'analyse en amont, de passer au compilateur les informations de haut niveau sur l'application ;
2. l'efficacité de la programmation : il s'agit à la fois d'atteindre des performances élevées, de passer à l'échelle en nombre de ressources de calcul et de pouvoir traiter des problèmes de très grande taille ;
3. la facilité de programmation : requérir un effort de programmation raisonnable de la part du programmeur.

3.1 Contexte

Nous nous plaçons dans le contexte de la programmation d'applications scientifiques denses à parallélisme de données sur architectures hybrides. Ces applications, telles que les problèmes de différences finies, sont largement présentes en calcul scientifique et en simulation numérique. Des exemples représentatifs de telles applications sont les benchmarks NAS [11] BT, SP et LU mais surtout des applications industrielles de traitement du signal telle que l'application industrielle LDPC de communication radio, implémentant les algorithmes d'encodage d'information du même nom [82].

dSTEP vise la parallélisation d'un programme dans sa globalité. La distribution des tableaux est spécifiée au niveau de leurs déclarations et reste fixe tout au long du programme. La spécification du parallélisme et de la distribution des nids de boucles est ensuite indiquée pour tous les nids de boucles accédant à des tableaux distribués.

La séparation de la spécification de la distribution des calculs et des données constitue une différence majeure par rapport à HPF [51]. Le programmeur a ainsi le contrôle sur ces deux aspects de la parallélisation d'un programme, ce qui permet au compilateur de *dSTEP* de générer un code avec un flot de contrôle et des communications plus simples que HPF car il n'existe pas de notion de propriétaire de données. Contrairement à XcalableMP [73], *dSTEP* ne place pas la responsabilité de la spécification des communications sur le programmeur.

Exemple de code généré par un compilateur HPF. Les exemples de codes montrés ici sont tirés de la documentation en ligne du compilateur dHPF [1]. Le listing 3.1 montre un programme HPF simple avec une boucle parallèle qui implique une communication sur le tableau `b` (lignes 15-17). Le listing 3.2 montre une partie du code généré pour ce programme correspondant à la boucle parallèle. On constate que, même pour ce programme simple, les accès locaux sont distingués des accès non locaux, qui sont résolus séparément (ligne 8). Le flot de contrôle d'un programme généré par un compilateur HPF est d'autant plus complexe que plusieurs références en écriture sont présentes dans un nid de boucles.

```
1        program simpletest
2           integer i
3           double precision a(100), b(100)
4
5  CHPF$    processors p(4)
6  CHPF$    template t(100)
7  CHPF$    align a(i) with t(i)
8  CHPF$    align b(i) with t(i)
9  CHPF$    distribute t(block) onto p
10
11 C        -- Initializations --
12
13          a(1) = 0
14
15          do i = 2, 100 - 1
16              a(i) = 0.25 * b(i-1)
17          enddo
18
19         end
```

Listing 3.1 – Exemple d'un code simple en HPF

3.2 Modèle de programmation

dSTEP est un ensemble de directives de compilation permettant d'exprimer à la fois la distribution des données et celle des calculs sur des programmes écrits dans le langage C. Le compilateur *dSTEP* analyse ces directives et génère un programme pour machines parallèles hybrides : le programme généré est hybride en cela qu'il exploite les différents nœuds d'une machine à mémoire distribuée, et au niveau de

1. http ://www.cs.rice.edu/ dsystem/dhpf/

```
C
C       --<< Iterations that might access non-local values >>--
C
C       Loop section ---[ i = ((25 * myid1) + 1) ]---
C
        if (1 .le. 25 * myid1 .and. 25 * myid1 .le. 98) then
          i = 25 * myid1 + 1
          lnltmp1 = hpf_nonlocal_lookupd(hash$nonlocals, b$data, i -
          1)
          a(i) = 0.25 * lnltmp1
        endif
C
C       --<< Iterations that access only local values >>--
C
C       Loop section ---[ max(((25 * myid1) + 2), 2) <= i <= min(((25
  * myid1) +
C 25), 99) ]---
C
        do i = max(25 * myid1 + 2, 2), min(25 * myid1 + 25, 99)
          a(i) = 0.25 * b(i - 1)
        enddo
```

Listing 3.2 – Code généré par le compilateur dHPF

chaque nœud, le programme utilise les différents cœurs de calcul disponibles ou les accélérateurs attachés.

3.2.1 Définitions : processus, processeur, mémoire locale

On utilise dans la suite le terme *processus* dans son acception système, à savoir une instance d'exécution d'un programme avec son environnement mémoire propre, que l'on désigne comme *mémoire locale*. Les processus sont des entités logiques, indépendantes des ressources de calcul sous-jacentes que sont les *processeurs* constitués par les CPUs et les GPUs ; on parle de distribution des calculs et des données sur un ensemble de processus indépendamment de leur affectation aux processeurs physiques qui est contrôlée par l'utilisateur au lancement du programme généré. Ces définitions permettent d'établir les modèles de programmation et de distribution, qui sont valables même si l'utilisateur décide d'exécuter plusieurs processus sur un même processeur. Dans la pratique, cependant, de meilleures performances sont généralement obtenues en lançant un seul processus sur chaque nœud physique de la machine parallèle hybride et autant de *threads* que de cœurs sur chaque processeur CPU.

3.2.2 Distribution des calculs

En *OpenMP*, on peut facilement indiquer le partitionnement d'un ensemble d'itérations parallèles avec la directive de partage de travail *omp parallel for*. Les itérations sont alors exécutées indépendamment par les différents threads du programme. Le listing 3.3 montre la parallélisation avec OpenMP d'un code implémentant la version naïve de la multiplication de matrices de la forme $C = C + A \times B$. La directive *omp parallel for* indique que la boucle i est pa-

rallèle et que ses itérations peuvent être partitionnées entre les différents threads disponibles à l'exécution. En OpenMP, la clause `schedule(static, taille)` permet de définir une taille de bloc d'itérations à affecter de façon cyclique aux threads créés.

```
1 #pragma omp parallel for private(i, j, k)
2   for (i = 0; i < M; i++)
3     for (j = 0; j < M; j++)
4       for (k = 0; k < M; k++)
5         C[i][j] =  C[i][j] + A[i][k] * B[k][j];
```

Listing 3.3 – Parallélisation OpenMP de la dimension i du nid de boucles i, j, k

Dans cet exemple, l'ensemble des itérations de la première dimension du nid de boucles est partitionné par blocs. On remarque que la dimension j est également parallèle et on aurait pu la partitionner à la place de i. Mais si on veut exprimer le partitionnement des deux dimensions i et j à la fois, on ne peut pas le faire avec OpenMP. En effet, OpenMP permet d'exprimer que les deux premières dimensions du nid de boucles sont parallèles et peuvent être fusionnées avec la clause *collapse* comme un seul espace d'itérations à partitionner sur les threads disponibles, mais ne permet pas d'exprimer que chacune de ces dimensions est parallèle séparément avec une seule directive. On peut cependant utiliser deux directives OpenMP afin d'exprimer un parallélisme imbriqué pour les dimensions i et j.

3.2.3 La directive *dstep gridify*

Nous pourrions réutiliser la directive *omp parallel for* dans *dSTEP* pour exprimer la distribution d'une dimension d'un nid de boucles. Cette directive indique que les itérations d'une telle dimension sont à distribuer entre les différents processus disponibles et que leurs exécutions peuvent être effectuées en parallèle. Cependant, la sémantique d'une telle directive est restreinte et n'est pas assez expressive. Tout d'abord, cette directive ne permet de travailler que sur une seule dimension d'un nid de boucles. Ensuite, elle mélange deux informations : la distribution et l'ordonnancement des itérations.

Nous proposons la directive *dstep gridify*, inspirée de la directive du même nom dans HMPP, et qui est plus générale que la directive *omp parallel for*. La directive *dstep gridify* est original par rapport aux directives *omp parallel for* et *hmppcg gridify* car elle exprime de façon plus générale, pour chaque dimension d'un nid de boucles, à la fois une information de distribution et une information d'ordonnancement des itérations.

Le type de distribution est indiqué par un attribut *dist* et le type d'exécution est indiqué par un attribut *sched*. Le listing 3.4 montre la syntaxe générale et informelle de la directive *dstep gridify* pour un nid de boucles avec les dimensions `i1, i2, ...`

Le tableau 3.1 définit les valeurs possibles de l'attribut de distribution ainsi que leur sémantique.

```
1 #pragma dstep gridify(i1(dist=dist_type[, diag]; sched=sched_type),
    i2(...), ...)
```

Listing 3.4 – Syntaxe de la directive *dstep gridify*

Type de distribution (attribut *dist*)	Sémantique
block	Les itérations de la boucle sont distribuées par blocs de la même taille et affectées aux processus disponibles. Chaque processus se voit affecté un seul bloc.
cyclic	Les itérations de la boucle sont distribuées par blocs de la même taille. La taille de bloc est définie par le programmeur. Les blocs sont attribués aux processus disponibles de façon cyclique.
* (distribution répliquée)	Toutes les itérations de la boucle sont attribuées à chaque processus en un seul bloc.
all	Les itérations de la boucle sont découpées par blocs de la même taille. La taille de bloc est définie par le programmeur. Tous les blocs sont affectés à chaque processus.

TABLE 3.1 – Sémantique des types de distribution de l'ensemble des itérations d'une boucle

La distribution diagonalisée. Pour un nid de boucles à deux dimensions ou plus, on peut indiquer qu'une dimension est diagonalisée avec le mot clé *diag*, donnant lieu à une distribution multi-partitionnée du nid de boucles (voir la distribution multi-partitionnée dans dHPF, section 2.3.2). Dans une telle distribution, chaque processus se voit affecté un ensemble de blocs d'itérations parallèles suivant une diagonale de la grille formée par l'ensemble des blocs. Le qualificatif *diag* peut être ajouté à une distribution *block* ou répliquée.

Le tableau 3.2 définit les valeurs possibles de l'attribut d'ordonnancement de la directive *dstep gridify* ainsi que leur sémantique.

Type d'ordonnancement (attribut *sched*)	Sémantique
parallel	Les itérations peuvent être exécutées indépendamment par les différents processus
ordered	L'exécution des itérations sur les différents processus et entre les processus doit respecter l'ordre initial des itérations de cette dimension
owner	Seuls les processus possédant les données accédées exécutent les itérations de cette dimension.

TABLE 3.2 – Sémantique des types d'ordonnancement de l'ensemble des itérations d'une boucle

3.2.4 Exploitation des cœurs d'un même nœud ou d'un accélérateur

Le programmeur indique pour chaque dimension d'un nid de boucles distribué un type d'ordonnancement. Pour un nid de boucles portant une directive *dstep gridify*, les itérations de la dimension parallèle la plus externe, après découpage entre les différents nœuds d'une architecture distribuée, sont exécutées au niveau de chaque nœud par les différents threads disponibles. Le découpage des itérations sur les threads disponibles s'effectue par blocs. Ce sont en revanche les dimensions parallèles les plus internes d'un nid de boucles qui peuvent être exportées sur un accélérateur si le programmeur choisit de compiler pour cette cible (voir le chapitre 6).

3.2.5 Applicabilité de la directive *dstep gridify*

La directive `dstep gridify` peut être utilisée pour distribuer plusieurs dimensions d'un nid de boucles avec les caractéristiques suivantes :

- les bornes de boucles peuvent être des constantes ou des expressions symboliques ne dépendant pas des dimensions de boucles englobantes distribuées ;
- les expressions d'accès aux tableaux dans le nid de boucles sont des fonctions affines des indices de boucles distribuées ;
- les incréments de boucles peuvent être positifs ou négatifs, d'une valeur absolue supérieure ou égale à un ;
- les nids de boucles peuvent être imparfaitement imbriqués si :
  - les instructions s'intercalant entre les différentes boucles ne contiennent pas de références à des tableaux distribués,
  - les boucles non parfaitement imbriqués correspondent aux boucles les plus internes ayant une distribution de type *all*.

3.2.6 Exemples d'utilisation de la directive *dstep gridify*

Exemple de distribution par blocs parallèles. On reprend le code de multiplication de matrices, exposé dans le Listing 3.3, en utilisant la directive *gridify* pour exprimer les mêmes informations que la directive *omp parallel for* : on distribue la dimension i par blocs et on indique que l'exécution des itérations est parallèle (listing 3.5).

```
1 #pragma dstep gridify(i(dist=block, sched=parallel))
2   for (i = 0; i < M; i++) {
3     for (j = 0; j < M; j++) {
4       double c = C[i][j];
5       for (k = 0; k < M; k++) {
6         c =  c + A[i][k] * B[k][j];
7       }
8       C[i][j] = c; }}
```

Listing 3.5 – Distribution avec dstep gridify de la boucle i

Lorsque la distribution des itérations d'une boucle est par blocs, et que leur exécution est parallèle, on peut omettre les attributs *dist* et *sched* qui deviennent dans ce cas des valeurs par défaut.

```
1 #pragma dstep gridify(i)
```

–

Le listing 3.6 montre la distribution des trois dimensions i, j et k du code de la multiplication de matrices. Les dimensions i et j sont distribuées par blocs avec un ordonnancement parallèle. La dimension k est distribuée avec une distribution *all* indiquant une taille de bloc de 64. Pour cette dimension, toutes les itérations seront exécutées par chaque processus par blocs de taille 64, avec un ordonnancement parallèle.

```
1 #pragma dstep gridify(i, j, k(dist=all(64); sched=ordered)))
2   for (i = 0; i < M; i++) {
3     for (j = 0; j < M; j++) {
4       double c = C[i][j];
5       for (k = 0; k < M; k++) {
6         c =  c + A[i][k] * B[k][j];
7       }
8       C[i][j] = c; }}
```

Listing 3.6 – Distribution des boucles i, j et k

Exemple de distribution répliquée. Une distribution répliquée s'utilise typiquement lorsque l'étendue d'une dimension d'un nid de boucles est réduite à un seul élément, par exemple $i = 1$ dans le Listing 3.7.

```
1   i = 1;
2   for (j = 1; j < grid_points[1]-1; j++) {
3     for (k = 1; k < grid_points[2]-1; k++) {
4       for (m = 0; m < 5; m++) {
5   rhs[i][j][k][m] = rhs[i][j][k][m]- dssp *
6     ( 5.0*u[i][j][k][m] - 4.0*u[i+1][j][k][m] +
7       u[i+2][j][k][m]);
8       }}}
```

Listing 3.7 – Itération isolée i (code extrait de NAS BT)

Cet exemple, Listing 3.7, peut être réécrit, comme le montre le Listing 3.8, avec la directive `dstep gridify` en :

- introduisant une boucle sur i allant de 1 à 1,
- répliquant la dimension i, pour que les itérations i (une seule ici) soient affectées à tous les processus de cette dimension.

L'ordonnancement *owner* indique que seuls les propriétaires des données accédées selon cette dimension effectuent le calcul.

```
1 #pragma dstep gridify(i(dist=*; sched=owner), j, k)
2   for (i = 1; i <= 1; i++) {
3     for (j = 1; j < grid_points[1]-1; j++) {
4       for (k = 1; k < grid_points[2]-1; k++) {
5         for (m = 0; m < 5; m++) {
6           rhs[i][j][k][m] = rhs[i][j][k][m]- dssp *
7           ( 5.0*u[i][j][k][m] - 4.0*u[i+1][j][k][m] +
8           u[i+2][j][k][m]);
9   }}}}
```

Listing 3.8 – Distribution répliquée de la dimension i (code extrait de NAS BT)

Exemple de distribution ordonnée. Une dimension d'un nid de boucles n'est pas parallèle lorsqu'elle introduit des dépendances entre les différentes itérations de cette dimension. Avec la directive *dstep gridify*, une telle dimension peut être distribuée mais son exécution doit être indiquée comme ordonnée. Dans le code du listing 3.9, la dimension i n'est pas parallèle car la fonction `matmul_sub`, qui écrit des éléments du tableaux *lhs* en lisant des élements du même tableau, porte des dépendances sur la dimension i.

```
1   for (i = 1; i < isize; i++) {
2     for (j = 1; j < grid_points[1]-1; j++) {
3       for (k = 1; k < grid_points[2]-1; k++) {
4           matvec_sub(lhs[i][j][k][AA],
5                      rhs[i-1][j][k], rhs[i][j][k]);
6   matmul_sub(lhs[i][j][k][AA],
7        lhs[i-1][j][k][CC],
8        lhs[i][j][k][BB]);
9
10  binvcrhs( lhs[i][j][k][BB],
11      lhs[i][j][k][CC],
12      rhs[i][j][k] );}}}
```

Listing 3.9 – Dépendances sur la dimension i (code extrait de NAS BT)

Avec la directive `dstep gridify`, les trois dimensions de ce nid de boucles peuvent être distribuées, avec indication d'une exécution ordonnée pour la première dimension, comme le montre l'exemple du listing 3.10.

3.2.7 Distribution des tableaux

La distribution d'un tableau est exprimée par la directive `dstep distribute`, inspirée de HPF. Cette directive indique un type de distribution pour chaque dimension d'un tableau à distribuer. On définit quatre types de distributions : bloc, bloc-cyclique, répliquée et diagonalisée. Le tableau 3.3 définit les types de distribution possibles pour une dimension d'un tableau.

La figure 3.1 montre quelques exemples de distribution d'un tableau à deux dimensions sur quatre processus selon différents types de distribution pour chaque dimen-

```
1  #pragma dstep gridify(i(dist=block; sched=ordered), j, k)
2    for (i = 1; i < isize; i++) {
3      for (j = 1; j < grid_points[1]-1; j++) {
4        for (k = 1; k < grid_points[2]-1; k++) {
5          matvec_sub(lhs[i][j][k][AA],
6                     rhs[i-1][j][k], rhs[i][j][k]);
7         matmul_sub(lhs[i][j][k][AA],
8                    lhs[i-1][j][k][CC],
9                    lhs[i][j][k][BB]);
10   binvcrhs(lhs[i][j][k][BB], lhs[i][j][k][CC], rhs[i][j][k]);}}}
```

Listing 3.10 – Distribution ordonnée de la dimension i (code extrait de NAS BT)

Type de distribution	Sémantique
block	Les éléments de tableau selon cette dimension sont distribués par blocs de la même taille et affectés aux processus disponibles. Chaque processus se voit affecté un seul bloc.
cyclic	Les éléments de tableau selon cette dimension sont distribués par blocs de la même taille. La taille de bloc est définie par le programmeur. Les blocs sont attribués aux processus disponibles de façon cyclique.
* (distribution répliquée)	Tous les éléments de tableau selon cette dimension sont répliqués sur chaque processus.
diag	Les éléments de tableau selon une diagonale sont affectés par blocs à chaque processus.

TABLE 3.3 – Sémantique des types de distribution des éléments d'une dimension de tableau

sion. La sous-figure (e) montre une distribution diagonalisée de la deuxième dimension du tableau. On remarque que les blocs affectés au processus 0 sont disposés selon la diagonale et les blocs affectés aux autres processus forment des lignes parallèles à cette diagonale. Ce type de distribution présente les propriétés d'une distribution multi-partitionnée : 1) une coupe par un hyperplan (une droite dans l'exemple) selon n'importe quelle dimension touchera des blocs de chaque processus, 2) pour chaque bloc, le bloc voisin selon une direction est toujours affecté au même processus.

3.2.8 Le halo

La distribution des itérations d'un nid de boucles affecte des ensembles d'itérations, ou *tranches d'itérations*, à chaque processus. Pour un tableau donné, les différentes tranches d'itérations exécutées sur un même processus n'accèdent pas toujours aux mêmes ensembles d'éléments du tableau. En effet, les fonctions d'accès aux tableaux varient ainsi que les bornes de boucles. Afin de préparer un espace mémoire local contenant une copie de tous les éléments référencés par tous les nids de boucles du programme, on définit un *halo*, inspiré des *shadow area* de XcalableMP [72] et des

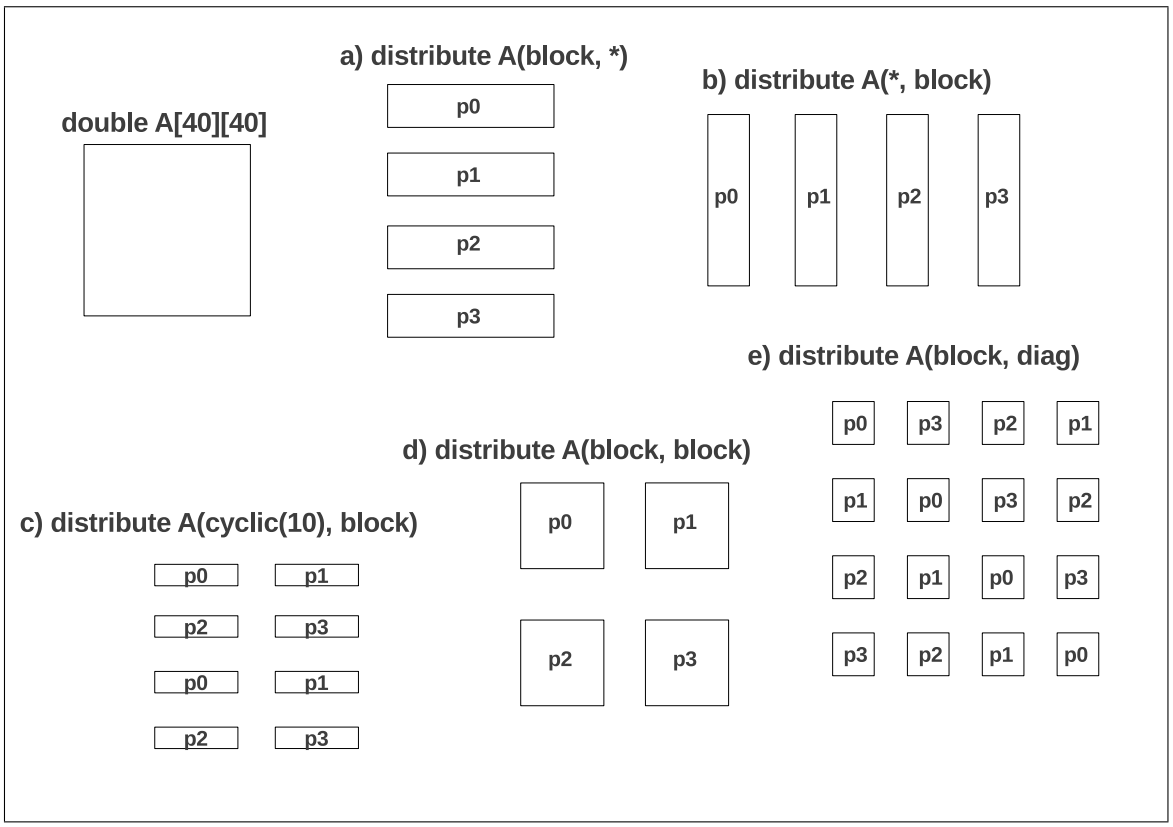


FIGURE 3.1 – Exemples de distribution d'un tableau 2D sur 4 processus

halos de Chapel [33], qui exprime une réplication locale de certains éléments distants en mémoire distribuée. Un halo est composé d'éléments supplémentaires contigus ajoutés de chaque côté d'une dimension distribuée d'un tableau. La figure 3.2 reprend les exemples de distribution du tableau `A` de la figure 3.1 en ajoutant des halos. La distribution (d) `distribute A(1:block:2, block)` ajoute un halo à chaque bloc de la première dimension. Ce halo a une portée égale à un du côté inférieur de cette dimension et égale à deux du côté supérieur.

3.2.9 Le halo total

Le halo peut couvrir tous les éléments supérieurs ou inférieurs d'une dimension, il s'agit alors d'un halo total, désigné par la constante `_H_`. Le halo total permet d'exprimer la réplication des éléments d'une dimension tout en maintenant une distribution multidimensionnelle. La figure 3.3 montre, pour un tableau à deux dimensions, une distribution monodimensionnelle, où la seconde dimension est répliquée (a) et une distribution bidimensionnelle, avec halo total sur la seconde dimension (b).

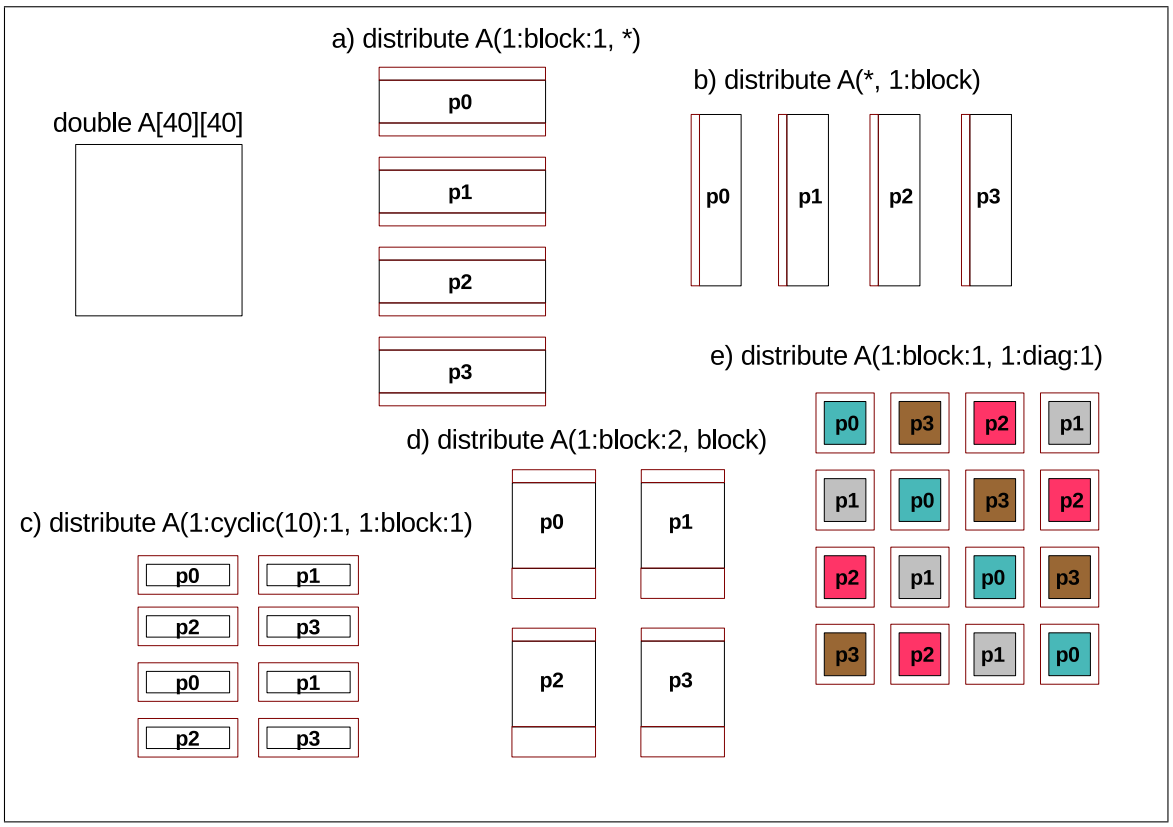


FIGURE 3.2 – Exemples de distribution avec halo d'un tableau 2D sur 4 processus

3.3 Modèle d'exécution

À partir d'un programme écrit en `C` et annoté de directives *dSTEP*, on génère un programme parallèle conforme au modèle d'exécution *SPMD*. Le modèle d'exécution de dSTEP se définit par les propriétés suivantes :

1. les parties du programme hors `gridify` sont exécutées de façon redondante par tous les processus ;
2. chaque processus alloue une partie des données du programmes et exécute une partie des itérations des nid de boucles, comme indiqué par les directives *distribute* et *gridify* ;
3. pour un `gridify` parallèle, chaque processus exécute les itérations qui lui sont affectées indépendamment des autres processus ;
4. une exécution `owner` d'une dimension n'active le calcul que pour les processus possédant toutes les données accédées selon cette dimension ;
5. la présence d'une dimension `ordered` dans une directive *gridify* résulte en une exécution séquentielle de cette dimension. Cette exécution respecte l'ordre initial, au niveau de chaque processus et entre les processus ;
6. des communications asynchrones sont générées pour mettre à jour les données répliquées lorsqu'elles sont modifiées ;

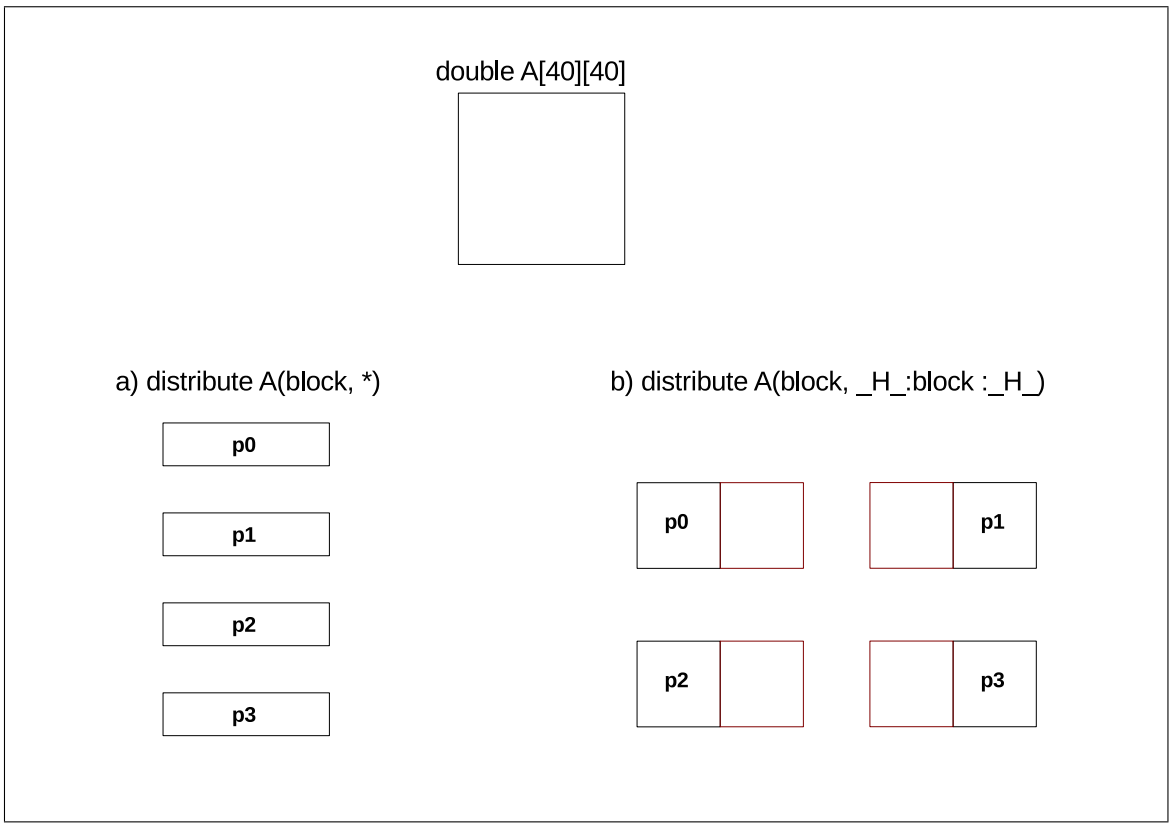


FIGURE 3.3 – Exemple de halo total (b), distribution sur 4 processus

7. le modèle d'exécution de *dSTEP* garantit que les communications sur un ensemble d'éléments de tableaux seront terminées au point du programme juste avant leur utilisation. La complétion des communications constituent donc un mécanisme de synchronisation dans *dSTEP*.

3.3.1 Vue globale du modèle de programmation de *dSTEP*

Le listing 3.11 illustre la combinaison de la distribution des données et celle des calculs ainsi que l'utilisation du halo. On montre l'exécution du code, incluant les communications, généré par le compilateur *dSTEP* sur quatre processus. Les tableaux bidimensionnels A et B sont distribués sur leurs deux dimensions par blocs avec la directive *distribute*. Pour le tableau A, on ajoute un halo de 1 de chaque côté des deux dimensions du tableau pour satisfaire la localité des accès de tous les nids de boucles du programme.

Le premier nid de boucles initialise le tableau A. Les accès sont parallèles sur les deux dimensions. On distribue les dimensions i et j implicitement par blocs, avec un ordonnancement parallèle. La figure 3.4 montre l'exécution de ce premier nid de boucles. On y montre les éléments du tableau A alloués sur chaque processus avec leurs indices dans l'espace séquentiel (voir la section suivante). Les éléments de tableau représentés par un rond vide représentent les éléments du halo. On voit

ainsi comment certains éléments du tableau A sont répliqués sur plusieurs processus, comme l'élément $A[0][4]$, qui est alloué sur les processus $p0$ et $p1$. On représente les itérations affectées à chaque processus par des croix. Les croix rouges sur les éléments du tableau A représentent les éléments alloués sur un processus et qui sont accédés par la tranche d'itérations exécutée sur ce processus. L'existence d'éléments répliqués déclenche des communications lorsque ces derniers sont modifiés. Sur la figure 3.5, on représente par des flèches les communications déclenchées par le processus $p0$: mise à jour des éléments modifiés par $p0$ et dont des copies sont allouées sur $p1$, $p2$ et $p3$.

```
1
2  #dstep distribute A(1:block:1, 1:block:1)
3  double A[8][8];
4  #dstep distribute B(block, block)
5  double B[8][8];
6
7  #pragma dstep gridify(i, j)
8    for (i = 0; i < 8; i++)
9      for (j = 0; j < 8; j++)
10       A[i][j] = (double)((i+j) / 1000);
11
12 #pragma dstep gridify(i, j)
13   for (i = 1; i < 7; i++)
14     for (j = 1; j < 7; j++)
15       B[i][j] = A[i][j] +
16                 A[i-1][j] + A[i+1][j] +
17                 A[i][j-1] + A[i][j+1];
18
19 #pragma dstep gridify(i, j(dist=*; sched=owner))
20   for (i = 0; i < 8; i++)
21     for (j = 0; j <= 0; j++)
22       A[i][j] = 0.001;
23
24 #pragma dstep gridify(i(dist=block; sched=ordered), j)
25   for (i = 1; i < 7; i++)
26     for (j = 0; j < 8; j++)
27       A[i][j] = A[i-1][j] + B[i][j];
```

Listing 3.11 – Succession de plusieurs nids de boucles avec différents gridify

Le deuxième nid de boucles montre un schéma d'accès aux éléments du tableau A avec des translations de $+1$ et -1 des indices i et j. Grâce au halo, ces accès sont locaux à tous les processus. Ce sont les communications déclenchées après l'exécution du premier nid de boucles qui ont permis de mettre à jour ces éléments. Les valeurs ainsi lues sont donc locales et correctes (figure 3.6).

Le troisième nid de boucles accède à la première colonne du tableau A. Comme cette colonne n'est pas allouée sur tous les processus, il faut utiliser une distribution répliquée de la dimension j avec un ordonnancement *owner*. Sur la figure 3.7, on voit comment les itérations de la dimension j (dans ce cas une seule, l'itération $j == 0$) sont affectées à tous les processus. Cependant, seuls les processus possédant les données accédées, c'est-à-dire $p0$ et $p2$ dans ce cas, exécutent les tranches d'itérations

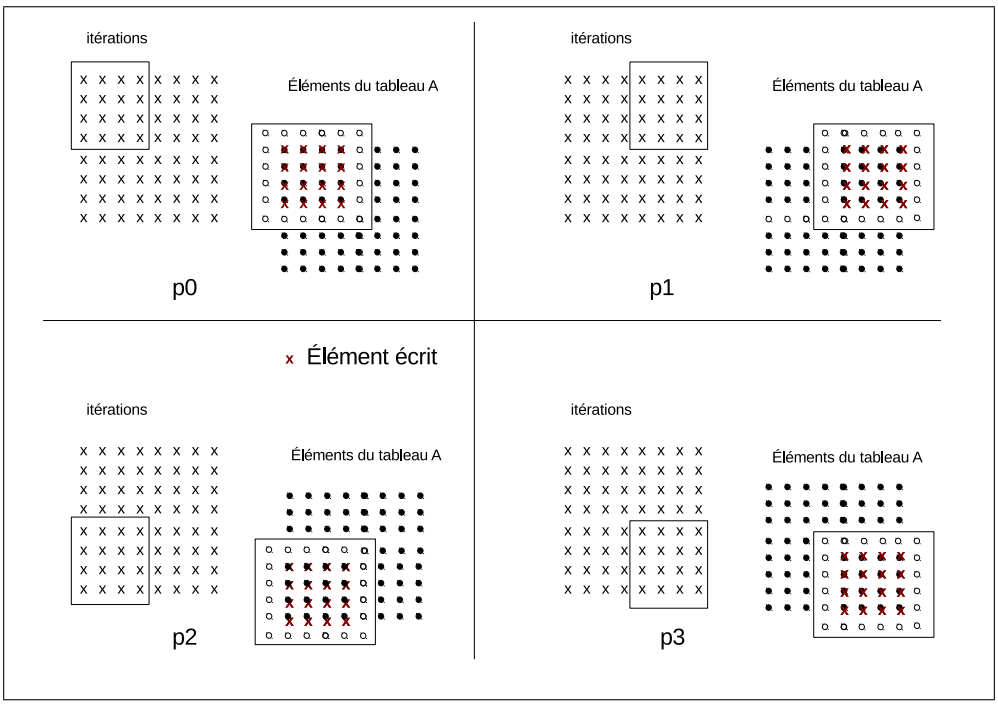


FIGURE 3.4 – Accès en écriture générés par le premier nid de boucles sur 4 processus pour le tableau A

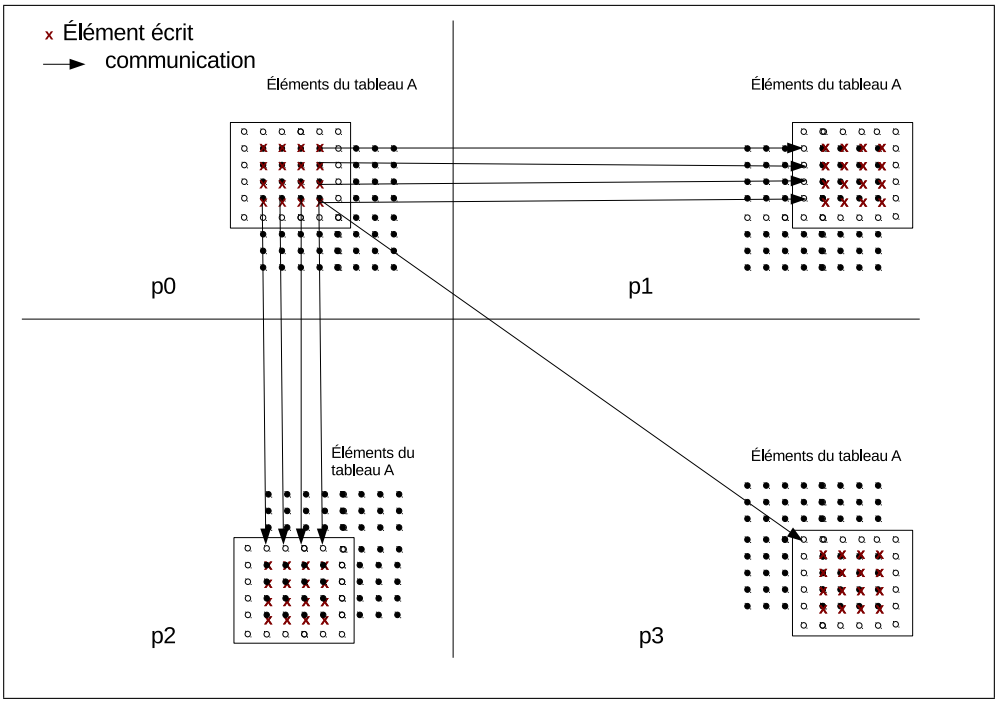


FIGURE 3.5 – Communications déclenchées par le processus p0 pour les éléments du tableau A

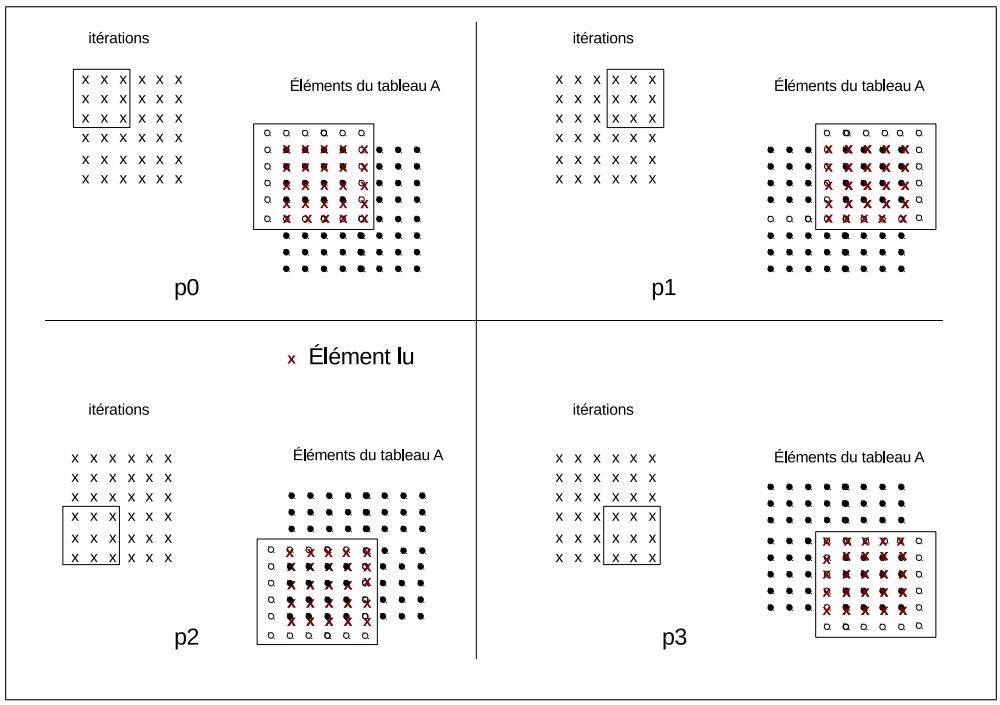


FIGURE 3.6 – Accès en lecture générés par le deuxième nid de boucles sur 4 processus pour le tableau A. La localité et la correction des valeurs lues sont assurées par le halo.

qui leurs sont affectées. Les processus $p1$ et $p3$ n'effectuent pas le calcul. On notera que des communications sont générées le long de la première dimension (qui est distribuée par bloc) pour mettre à jour les éléments répliqués.

Enfin, le quatrième nid de boucles présente une contrainte d'ordonnancement sur sa première dimension exprimée par l'utilisation du type d'ordonnancement *ordered*, alors que la deuxième dimension est parallèle. Les figures 3.8 et 3.9 montrent l'exécution de ce nid de boucles en deux temps. Dans un premier temps (figures 3.8), seuls les processus $p0$ et $p1$ sont actifs. Après l'exécution des tranches d'itérations affectées à ces processus, des communications sont déclenchées afin de mettre à jour les éléments répliquées sur $p2$ et $p3$ dont dépend la suite du calcul. À la réception de ces données, les processus $p2$ et $p3$ exécutent à leur tour leur tranches d'itérations dans un deuxième temps (figure 3.9).

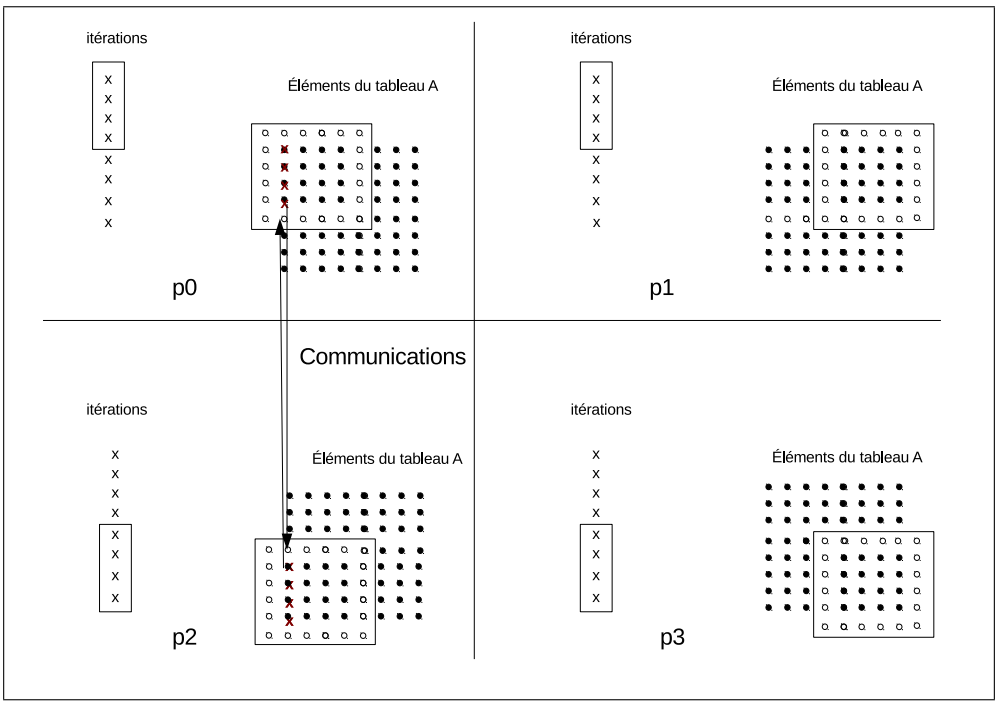


FIGURE 3.7 – Accès en écriture générés par le troisième nid de boucles sur 4 processus pour le tableau A.

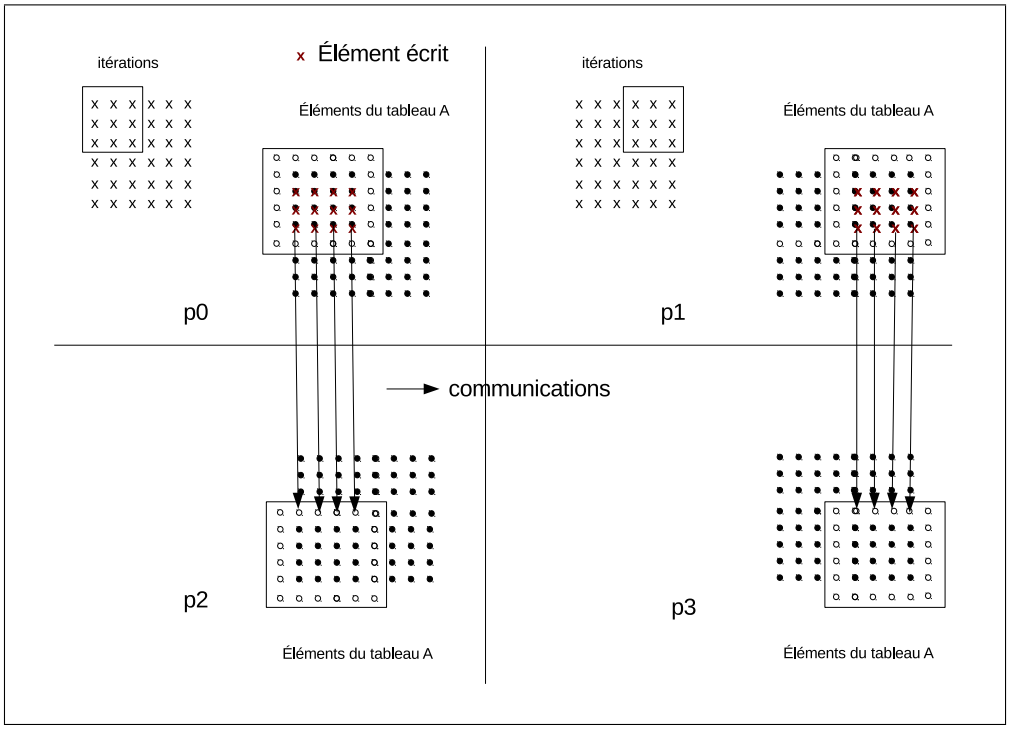


FIGURE 3.8 – Accès en écriture générés par le quatrième nid de boucles sur 4 processus pour le tableau A, premier temps

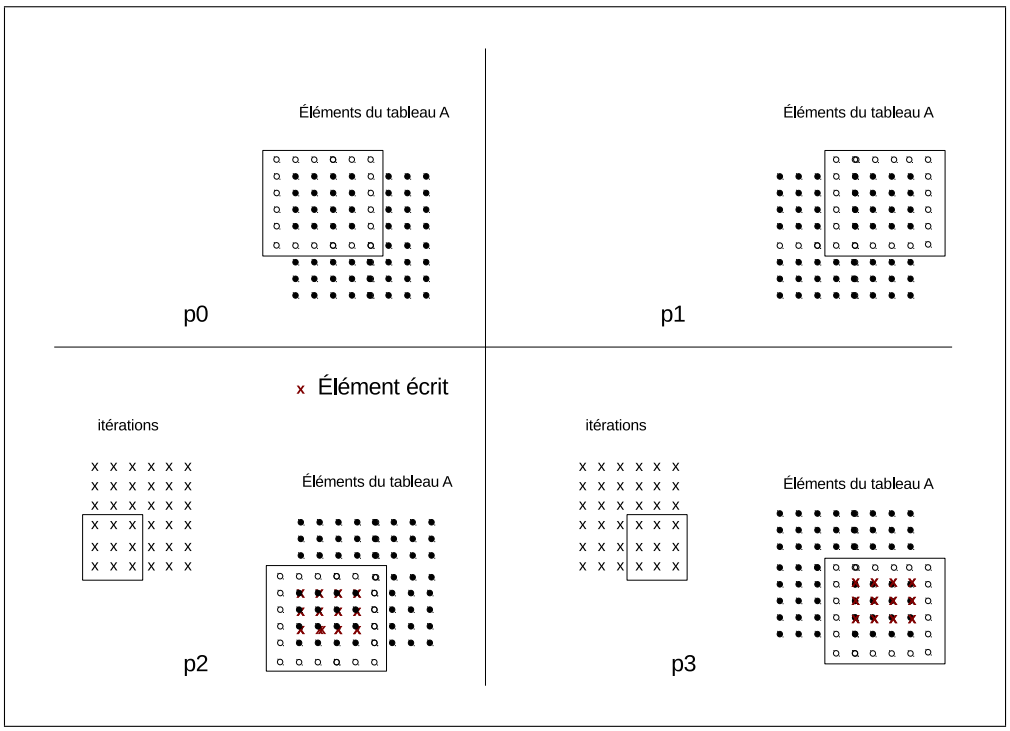


FIGURE 3.9 – Accès en écriture générés par le quatrième nid de boucles sur 4 processus pour le tableau A, deuxième temps

3.4 Conclusion

Nous avons présenté un modèle de programmation unifié pour les architectures hybrides à base de deux directives. La directive *dstep distribute* permet d'exprimer la distribution des tableaux avec plusieurs types de distributions possibles inspirées des langages HPF et dHPF. Dans la définition de la distribution de chaque dimension, nous intégrons le concept de halo qui permet de déclarer des éléments répliqués. La directive *dstep gridify* permet d'exprimer à la fois la distribution des dimensions d'un nid de boucles et leurs ordonnancements. Cette directive porte l'information utile pour la génération de code sur une architecture parallèle hybride à mémoire distribuée et à mémoire partagée ou avec des accélérateurs attachés à chaque nœud.

Chapitre 4

Modèle de distribution

Dans ce chapitre, nous présentons un modèle de distribution des données et un modèle de distribution des calculs. Nous établissons ensuite une relation entre ces deux modèles basée sur les fonctions d'accès aux éléments de tableaux. Le formalisme que nous utilisons est inspiré de la modélisation du compilateur *HPFC* présenté à la section suivante, mais nos modèles de programmation et de distribution sont originaux.

4.1 HPFC

HPFC est un compilateur HPF développé à l'École des Mines de Paris par Coelho *et al* [7]. Les auteurs ont modélisé la compilation de HPF par un formalisme d'algèbre linéaire décrivant les *templates*, les processeurs et les tailles des blocs d'éléments de tableaux distribuées avec des matrices. La distribution est alors définie par la mise en équation de toutes ces matrices.

Nous nous inspirons de ce travail pour modéliser la distribution, avec les différences suivantes :

- nous n'utilisons pas de notion de *template*,
- nous modélisons à la fois la distribution des données et la distribution des calculs,
- nous introduisons la notion de halo pour la distribution des données,
- nous intégrons dans le même modèle unifié les distributions bloc, cyclique et multi-partitionnée.

4.2 Modèle de distribution

Les éléments de tableaux et les itérations des nids de boucles sont distribués sur des *grilles virtuelles de processus*. À chaque tableau distribué ou nid de boucles distribué est associée sa propre grille virtuelle de processus, dont la construction dépend des informations de distribution.

Nous allons tout d'abord expliquer la construction d'une grille virtuelle *simple*, en absence de dimension diagonalisée, puis la construction d'une grille virtuelle

étendue correspondant à une distribution multi-partitionnée. Le second cas est une généralisation du premier, et on parlera par la suite de grille virtuelle de processus dans le cas général.

4.2.1 Ensemble de processus

Le nombre de processus est fixé par le programmeur au lancement du programme et ne change pas tout au long de son exécution. Ce nombre est représenté par la variable *nb_procs*.

Définition. On définit $\mathcal{P}$ l'ensemble des processus, avec p l'identifiant d'un processus :

$$\mathcal{P} = \{p | 0 \leq p < nb_procs\} \tag{4.1}$$

4.2.2 Grille virtuelle simple

Une grille virtuelle simple de processus est construite à partir des informations de distribution et de l'ensemble de processus $\mathcal{P}$. Le nombre de dimensions de la grille est égal au nombre de dimensions distribuées. La construction d'une telle grille est définie par les fonctions de topologie MPI.

Exemples. Le tableau 4.1 montre la construction d'une grille virtuelle simple correspondant au code du listing 4.1 pour un tableau 2-dimensionnel `A`. Les éléments du tableau `A` seront distribués sur une grille 2-dimensionnelle car la directive *dstep distribute* indique une distribution 2-dimensionnelle. La grille virtuelle simple construite sera caractérisée par la matrice diagonale $P' = \begin{bmatrix} P'_{0,0} & 0 \\ 0 & P'_{1,1} \end{bmatrix}$ où $P'_{0,0}$ (resp. $P'_{1,1}$) est le nombre de processus alloués à la première (resp. à la deuxième) dimension de la grille.

```
1 #pragma dstep distribute A(1:block:1, 2:cyclic(6):1)
2 double A[m][n];
```

Listing 4.1 – Exemple d'un tableau distribué sur ses deux dimensions

nb_procs	Forme de la grille virtuelle
4	2×2
8	4×2
11	11×1

TABLE 4.1 – Exemples de grilles virtuelles simples correspondant au listing 4.1 pour plusieurs valeurs de *nb_procs*

Remarques.

- Les fonctions de topologie MPI déterminent la forme de la grille, c'est-à-dire ici les valeurs $P'_{0,0}$ et $P'_{1,1}$. Ainsi, avec huit processus, on aura toujours une grille 4×2 pour une distribution 2-dimensionnelle où aucune dimension n'est répliquée ni diagonalisée comme le cas du listing 4.1. On n'obtiendra pas une grille 2×4 ou 8×1 par exemple dans ce cas.
- Avec onze processus, la grille virtuelle construite, bien qu'elle soit 2-dimensionnelle, aura un effet identique à une grille mono-dimensionnelle car le nombre de processus ici n'est factorisable qu'en 11×1. L'alternative 1×11 ne se réalisera pas dans ce cas car les fonctions de topologie MPI le définissent ainsi.

Lorsqu'une dimension est distribuée avec une distribution répliquée, comme indiquée dans le listing 4.2 pour la première dimension du tableau `A`, le nombre de processus affectés à cette dimension dans la grille virtuelle simple est égale à 1, comme le montre le tableau 4.2.

```
1 #pragma dstep distribute A(*, 2:cyclic(6):1)
2 double A[m][n];
```

Listing 4.2 – Exemple d'un tableau distribué sur ses deux dimensions, avec une distribution répliquée sur la première dimension

nb_procs	Forme de la grille virtuelle
4	1×4
8	1×8
11	1×11

TABLE 4.2 – Exemples de grilles virtuelles simples pour plusieurs valeurs de *nb_procs* pour le code du listing 4.2

Remarque. Dans toute la suite, $\overrightarrow{cte}$ est le vecteur dont toutes les composantes sont égales à la constante entière *cte*, de la même dimension que les matrices qui apparaissent dans la même expression. Par exemple, si P' est une matrice de dimension 3, alors dans l'expression $P'\vec{1}$, le vecteur $\vec{1}$ vaut $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$.

Définition. Une grille virtuelle simple de processus est définie par l'ensemble $\mathcal{P}'_{grid}$ (équation 4.2). Le vecteur $\vec{p}_{grid}$ est l'identifiant d'un processus dans cette grille et P' est une matrice diagonale contenant le nombre de processus de la grille selon chaque dimension.

$$\mathcal{P}'_{grid} = \{\vec{p}_{grid} | \vec{0} \leq \vec{p}_{grid} < P'\vec{1}\} \tag{4.2}$$

Définition. On définit la fonction *id_in_grid* qui associe un identifiant de processus dans l'ensemble de processus $\mathcal{P}$ à un identifiant dans la grille virtuelle simple $\mathcal{P}'_{grid}$.

$$id_in_grid : \begin{array}{ll} \mathcal{P} & \rightarrow \mathcal{P}'_{grid} \\ p & \mapsto \vec{p}_{grid} \end{array} \tag{4.3}$$

La figure 4.1 montre la construction de la grille virtuelle simple de processus correspondant à la première entrée de la table 4.1 en utilisant la fonction *id_in_grid*.

Remarque importante. La fonction *id_in_grid* est construite en utilisant les fonctions de topologie MPI qui garantissent sa bijectivité.

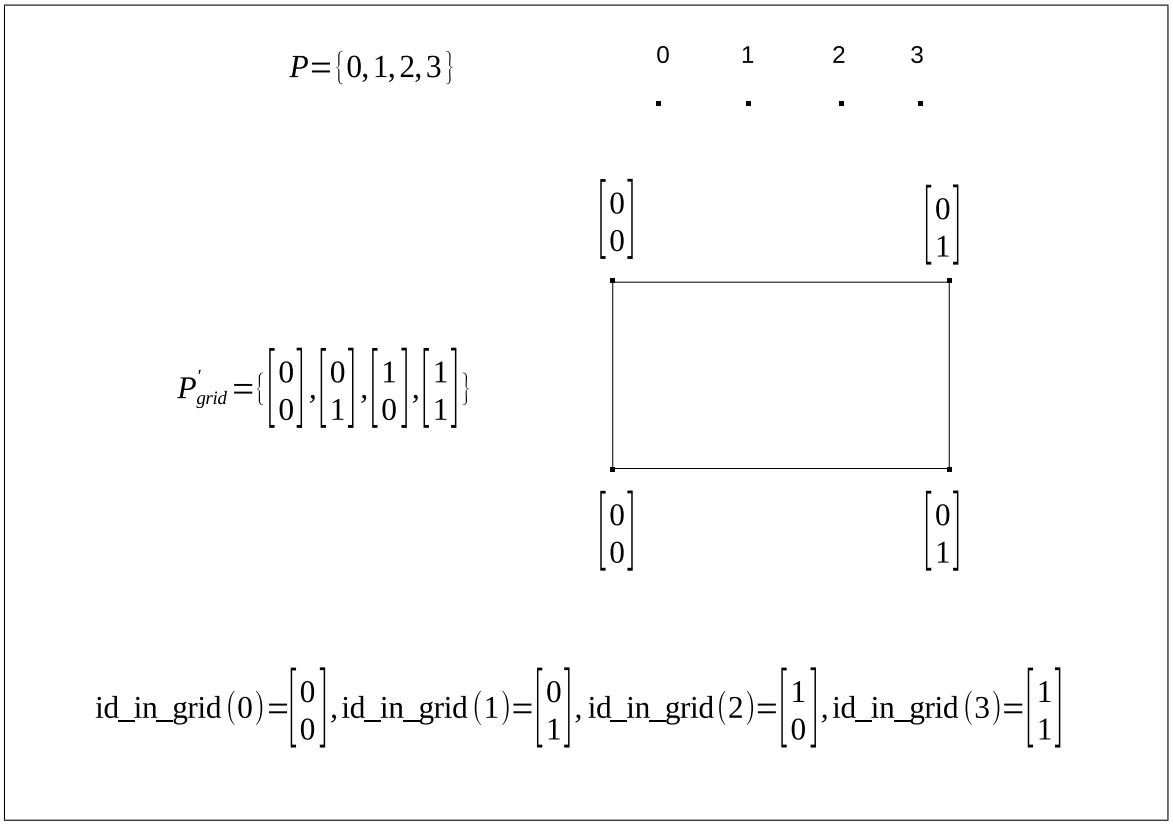


FIGURE 4.1 – Construction de la grille virtuelle simple de processus correspondant à la première entrée de la table 4.1

4.2.3 Grille virtuelle de processus étendue

Un tableau ou un nid de boucles avec un nombre de dimensions supérieur ou égal à deux peut avoir une dimension distribuée avec une distribution diagonalisée en utilisant le mot clé *diag*. La distribution ainsi obtenue est dite *multi-partitionnée*, pour reprendre le vocabulaire utilisé dans l'implémentation de référence des *benchmarks NAS BT* et *SP* [11] pour ce type de distribution. Nous rappelons sur un exemple ce qu'est une distribution multi-partitionnée ainsi que son intérêt. Nous explicitons

ensuite la construction d'une grille virtuelle, que l'on qualifiera *d'étendue*, associée à un tableau ou un nid de boucle avec une distribution multi-partitionnée.

Contrainte du modèle. Une seule dimension d'un nid de boucles ou d'un tableau peut être diagonalisée.

Intérêt. Une distribution multi-partitionnée permet d'obtenir un meilleur équilibrage de charge pour des schémas d'accès *à balayage*. Il s'agit de tableaux multi-dimensionnels qui sont accédés par des nids de boucles avec au moins une dimension comportant une dépendance, imposant ainsi un ordre séquentiel sur cette dimension. Dans ce type de programmes, la dimension ordonnée change entre les différents nids de boucles successifs du programme.

Exemple. La figure 4.2 montre un exemple d'exécution d'un code effectuant un balayage par lignes d'un tableau 2-dimensionnel sur quatre processus. Si le tableau parcouru est distribué par lignes, alors il faudrait quatre *étapes* successives pour effectuer le calcul. Les différents niveaux de gris indiqués sur la figure montre ces différentes étapes, rappelant une *vague* avançant par blocs de lignes. Dans ce cas, à chaque étape, un seul processus est actif et les trois autres sont inactifs, en attente des résultats des étapes précédentes. Il est ainsi clair que cette configuration résulte en une dégradation des performance de l'exécution parallèle.

Si en revanche on distribuait le tableau par blocs de colonnes alors le problème de la dépendance sur la première dimension ne se poserait plus et les calculs seraient effectués en une seule étape où tous les processus sont actifs. Mais si un nid de boucle suivant accède au même tableau avec une dépendance sur la deuxième dimension, alors le problème symétrique se poserait.

Une distribution 2-dimensionnelle ne résoudrait pas non plus le problème de l'équilibrage de charge car il y aurait deux étapes de calcul et pour chaque étape, la moitié des processus seraient inactifs.

Maintenant que nous avons rappelé le problème posé par ce type de schéma d'accès, nous verrons comment une distribution multi-partitionnée permet de régler le problème de l'équilibrage de charge. Dans notre exemple, si la deuxième dimension du tableau est distribuée avec un type *diag*, on obtient une distribution multi-partitionnée comme indiqué sur la figure 4.2. Les blocs du tableau `A` sont affectés aux différents processus suivant une diagonale (nous donnerons les détails de la construction d'une telle distribution dans les sections suivantes). Dans ce cas, on a toujours quatre étapes de calcul, mais la différence majeure avec les distributions précédentes est qu'à chaque étape de calcul, tous les processus sont actifs.

Remarque importante. La distribution multi-partitionnée, illustrée par l'exemple précédent, permet d'obtenir un meilleur équilibrage de charge, mais introduit des communications supplémentaires entre les différents blocs de données, qui sont plus nombreux que dans les autres exemples de distribution. Le gain effectif d'une telle distribution découle directement de la différence entre le gain en équilibrage de charge

et les surcoûts des communications induites, décrits dans le chapitre 7 sur le modèle de coût de dSTEP.

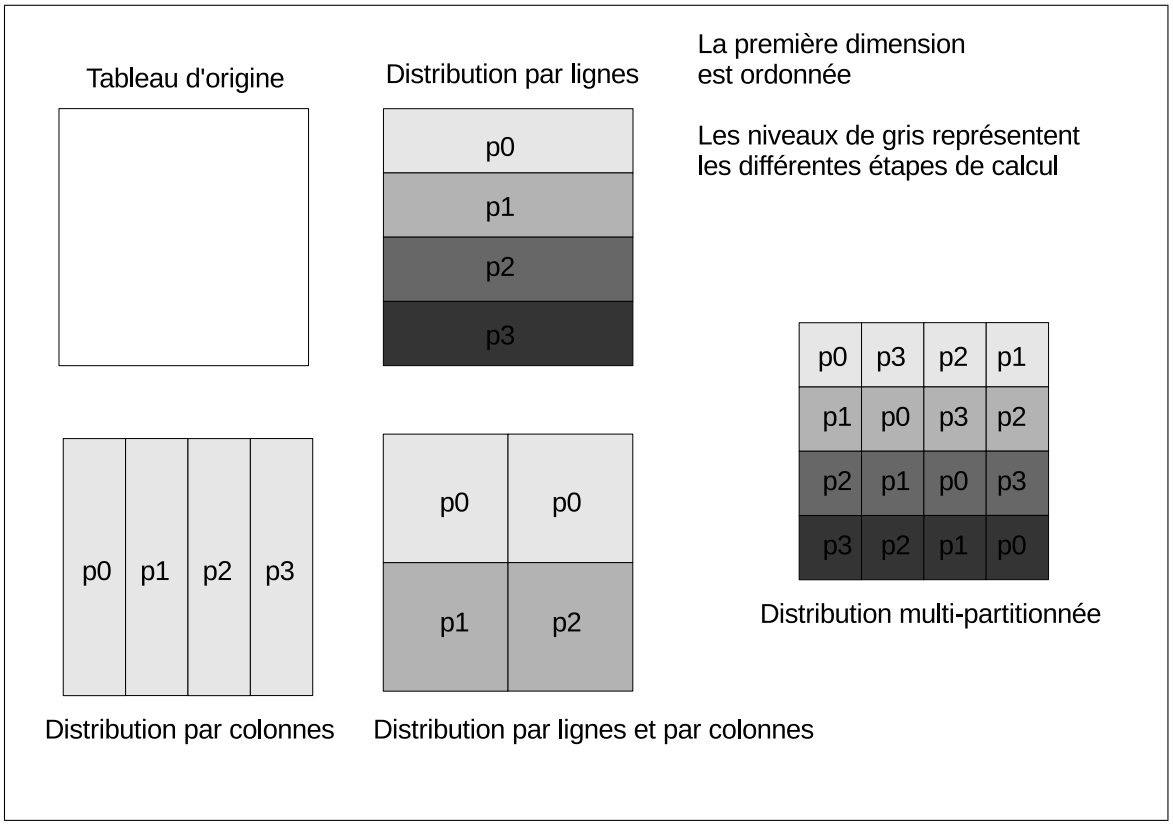


FIGURE 4.2 – Illustration de la distribution multi-partitionnée sur un tableau 2-dimensionnel

Une distribution multi-partitionnée est effectuée sur une grille de processus étendue, dont nous donnons la définition dans la section suivante.

Définition. Une grille virtuelle étendue de processus est définie par l'ensemble $\mathcal{P}_{grid}$ (équation 4.5). Le vecteur $\vec{\tilde{p}}_{grid}$ est un identifiant de processus virtuel dans cette grille. La matrice P, de dimension d, est une matrice diagonale contenant le nombre de processus selon chaque dimension. La matrice P est obtenue à partir de la matrice P' (équation 4.4) en l'étendant d'une dimension correspondant à la dimension diagonalisée et en donnant comme valeur à cette dimension un nombre de processus virtuels correspondant au minimum du nombre de processus affectés aux autres dimensions, autres les dimensions répliquées.

$$P_{k,k} = \begin{cases} P'_{k,k} & \textit{si k est non diagonalisée} \\ MIN\{P'_{j,j}, 0 \leq j < d, j \neq k, j \text{ n'est pas répliquée}\} & \textit{sinon} \end{cases} \tag{4.4}$$

$$\mathcal{P}_{grid} = \{\vec{\tilde{p}}_{grid} | \vec{0} \leq \vec{\tilde{p}}_{grid} < P\vec{1}\} \tag{4.5}$$

Exemple. Le listing 4.3 montre l'utilisation de la directive *dstep distribute* pour indiquer que la deuxième dimension du tableau `A` est diagonalisée. Les différentes étapes permettant de construire la grille virtuelle de processus étendue pour cette distribution sont illustrées par la figure 4.3. Sur la grille virtuelle étendue, il y a seize identifiants de processus alors que l'on ne dispose réellement que de quatre processus. En effet, les processus affectés à la deuxième dimension de la grille, correspondant à la dimension diagonalisée, n'existent pas. Ils sont obtenus en associant *vprocs* processus virtuels à chaque processus *réel*. Dans l'exemple précédent, $vprocs = 4$.

Définition. Pour une grille virtuelle de processus, on définit :

$$vprocs = \begin{cases} P_{k,k} & si \text{ une dimension } k \text{ est diagonalisée,} \\ 1 & sinon \end{cases} \tag{4.6}$$

La matrice P a donc la propriété suivante :

$$det(P) = nb_procs \times vprocs \neq 0 \tag{4.7}$$

```
1 #pragma dstep distribute A(1:block:1, 1:diag:1)
2 double A[m][n];
```

Listing 4.3 – Exemple d'un tableau distribué sur ses deux dimensions, avec une distribution diagonalisée sur la deuxième dimension

Définition. La fonction *extend_id* (équation 4.8) associe *vprocs* identifiants de processus dans une grille virtuelle étendue à chaque identifiant de processus dans une grille virtuelle simple. L'ensemble $\mathcal{V}$ est défini par $\mathcal{V} = [0, vprocs[$.

$$\begin{aligned} extend_id : \quad & \mathcal{P}'_{grid} \times \mathcal{V} \rightarrow \mathcal{P}_{grid} \\ & (\vec{p'}_{grid}, v) \rightarrow (\vec{p'}_{grid} + v\vec{1}) mod_v P\vec{1} \end{aligned} \tag{4.8}$$

Exemple. La figure 4.4 montre les quatre processus virtuels associés au processus $p = 1$.

Théorème. 1. *La fonction extend_id est bijective.*

La bijectivité de la fonction *extend_id* permet de garantir que tous les processus virtuels peuvent être énumérés : chaque processus obtient son identifiant en interrogeant l'environnement d'exécution, et énumère les *vprocs* qui lui sont affectés grâce à la fonction *extend_id*. Les grilles virtuelles étant les supports sur lesquels les nids de boucles et les tableaux sont distribués, cette propriété garantit que toutes les itérations et les éléments de tableaux distribués peuvent être visités (voir le chapitre compilation 5).

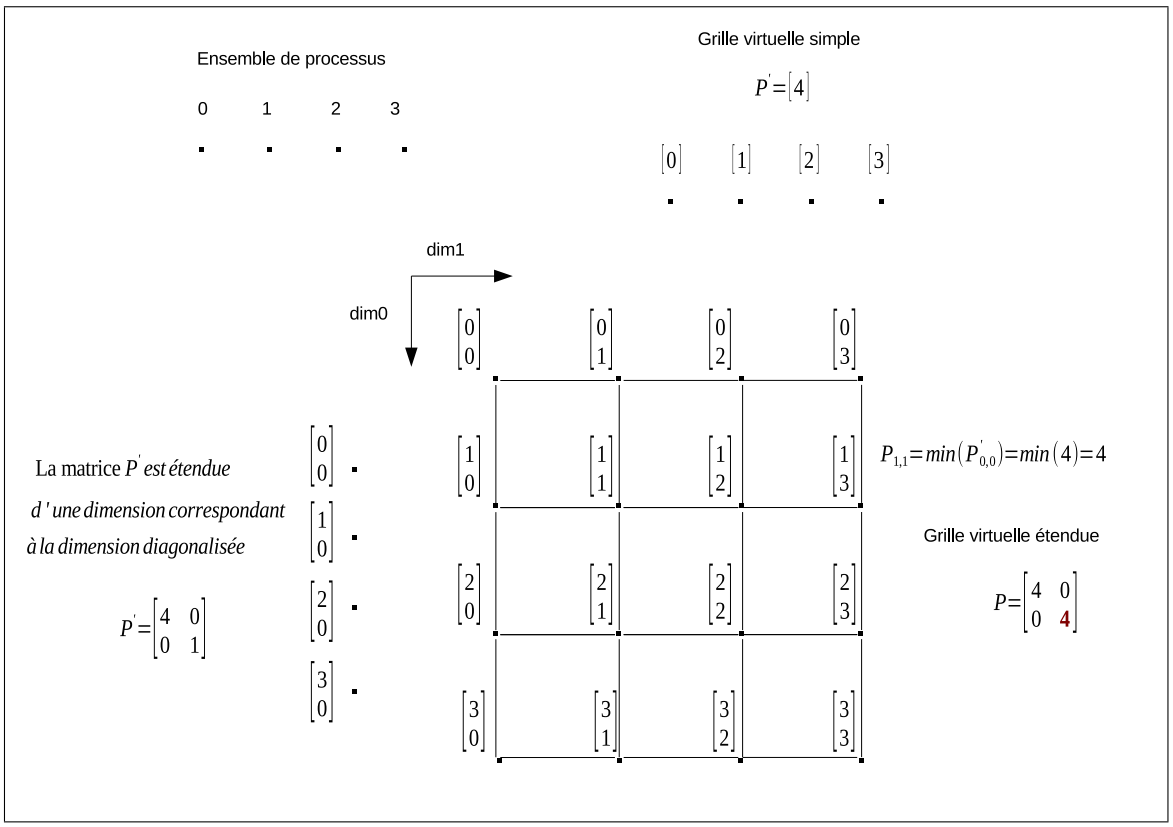


FIGURE 4.3 – Construction d'une grille virtuelle étendue

Démonstration. Si aucune dimension n'est diagonalisée, la fonction *extend_id* est la fonction identité, qui est trivialement bijective. Sinon, soit k la dimension diagonalisée de la grille :

a) Soit $(\vec{p'}_{grid}, v), (\vec{q'}_{grid}, w) \in \mathcal{P}'_{grid} \times \mathcal{V}$ avec $(\vec{p'}_{grid}, v) \neq (\vec{q'}_{grid}, w)$

On suppose que $extend_id(\vec{p'}_{grid}, v) = extend_id(\vec{q'}_{grid}, w)$

$\implies (\vec{p'}_{grid} + v\vec{1}) mod_v P\vec{1} = (\vec{q'}_{grid} + w\vec{1}) mod_v P\vec{1}$

L'équation précédente exprime que les vecteurs $(\vec{p'}_{grid} + v\vec{1})$ et $(\vec{q'}_{grid} + w\vec{1})$ ont le même modulo, composante par composante, par le vecteur $(P\vec{1})$. Leur difference est donc forcément un multiple du vecteur $(P\vec{1})$, ce qui s'ecrit :

$((\vec{p'}_{grid} + v\vec{1}) - (\vec{q'}_{grid} + w\vec{1})) mod_v P\vec{1} = \vec{0}$

$\implies ((\vec{p'}_{grid} - \vec{q'}_{grid}) + (v - w)\vec{1}) mod_v P\vec{1} = \vec{0}$

$\implies ((\vec{p'}_{grid}(k) - \vec{q'}_{grid}(k)) + (v - w)) \mod P_{k,k} = 0$

Or, on a par définition $\vec{p'}_{grid}(k) = \vec{q'}_{grid}(k) = 0$,

ce qui implique que $(v - w) \mod P_{k,k} = 0$, c'est-à-dire que $(v - w) = nP_{k,k}, n \in \mathbb{Z}$

On a par ailleurs $0 \leq v < P_{k,k}$ et $0 \leq w < P_{k,k}$, ce qui implique que $-P_{k,k} < v - w < P_{k,k}$.

On en déduit que n ne peut être que nul, donc que $v = w$.

On peut maintenant réécrire $(\vec{p'}_{grid} - \vec{q'}_{grid} + (v - w)\vec{1}) mod_v P\vec{1} = \vec{0}$ en

$(\vec{p'}_{grid} - \vec{q'}_{grid}) mod_v P\vec{1} = \vec{0}$

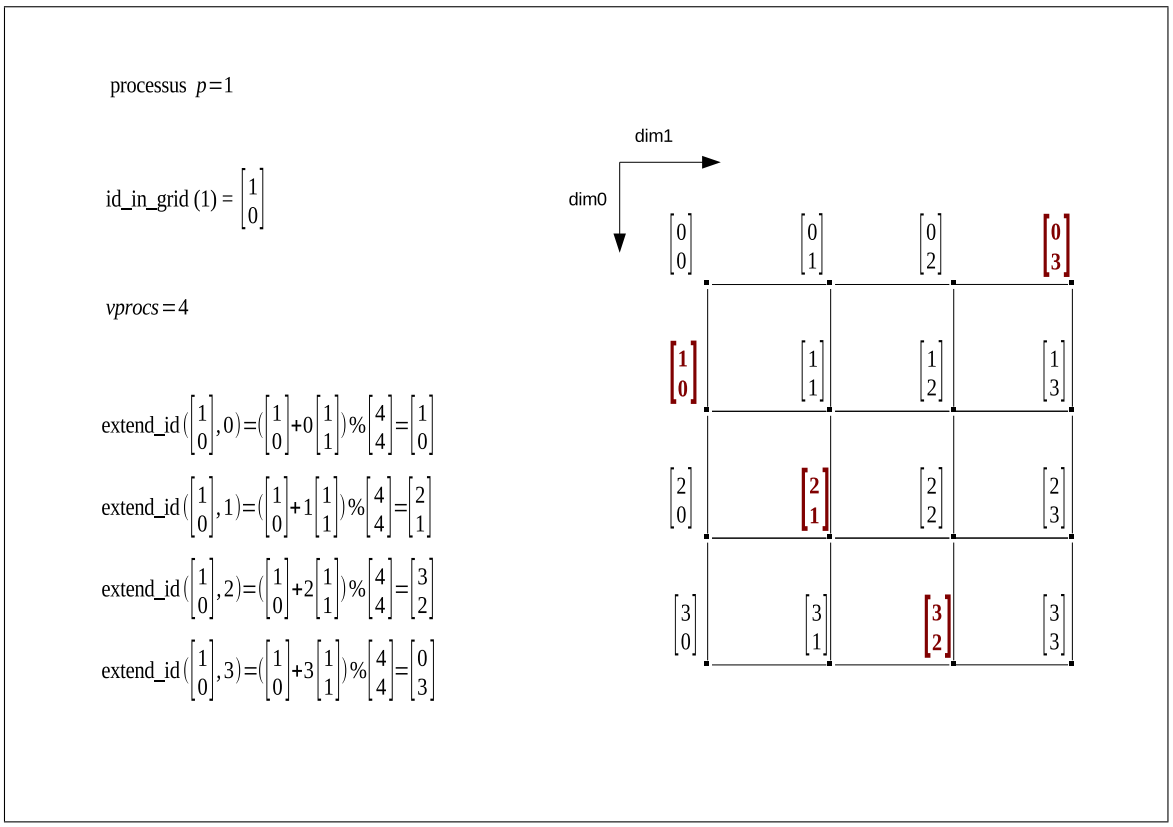


FIGURE 4.4 – Illustration de la fonction *extend_id*

Soit une dimension $k', k' \neq k$,
$(\vec{p'}_{grid} - \vec{q'}_{grid}) mod_v P\vec{1} = \vec{0} \implies (\vec{p'}_{grid}(k') - \vec{q'}_{grid}(k')) \mod P_{k',k'} = 0$
$\implies \vec{p'}_{grid}(k') - \vec{q'}_{grid}(k') = nP_{k',k'}, n \in \mathbb{Z}$

On a par ailleurs $0 \leq \vec{p'}_{grid}(k') < P_{k',k'}$ et $0 \leq \vec{q'}_{grid}(k') < P_{k',k'}$, ce qui implique que $-P_{k',k'} < \vec{p'}_{grid}(k') - \vec{q'}_{grid}(k') < P_{k',k'}$,

On en déduit que n ne peut être que nul, donc que $\vec{p'}_{grid}(k') = \vec{q'}_{grid}(k')$ pour tout k', c'est à dire que $\vec{p'}_{grid} = \vec{q'}_{grid}$.

Le résultat $v = w$ et $\vec{p'}_{grid} = \vec{q'}_{grid}$ est en contradiction avec notre hypothèse, *extend_id* est donc une *injection*.

b) On montre que *extend_id* est une surjection, c'est-à-dire qu'il existe un antécédent $(\vec{p'}_{grid}, v) \in \mathcal{P}'_{grid} \times \mathcal{V}$ pour tout élément $\vec{\tilde{p}} \in \mathcal{P}_{grid}$.

On vérifie facilement que l'élément $(\vec{p'}_{grid} = (\vec{\tilde{p}}_{grid} + P\vec{1} - \tilde{p}(k)\vec{1}) mod_v P\vec{1}, v = \vec{\tilde{p}}(k))$ est un antécédent de l'élément $\vec{\tilde{p}} \in \mathcal{P}_{grid}$.

La fonction *extend_id* est donc bien une *bijection*.

□

Remarque. Une grille virtuelle simple n'est qu'un cas particulier d'une grille virtuelle étendue dans laquelle $vprocs = 1$ et la fonction *extend_id* est la fonction iden-

tité. Dans toute la suite, nous parlerons de grilles virtuelles de processus au sens large.

4.2.4 Domaine séquentiel et domaine distribué

Le but du modèle de distribution est de définir des fonctions permettant de construire un schéma de compilation permettant de transformer un programme en entrée annoté avec les directives *dSTEP* en un programme parallèle pour machines parallèles hybrides tout en montrant la validité de la transformation. On distingue donc deux domaines : le *domaine séquentiel* qui est celui du code en entrée et le *domaine distribué* qui est celui du code parallèle généré. On définit les objets manipulés dans ces deux domaines ainsi que les fonctions les reliant.

4.2.5 Modèle de distribution des calculs sans calculs redondants

1) Nid de boucles dans le domaine séquentiel

Dans le domaine séquentiel, le domaine d'itération d'un nid de boucles (*loop nest*) est caractérisé par :

- d un entier représentant le nombre de dimensions gridifiées successives du nid de boucles,
- $\vec{i}$ un vecteur d'entiers de dimension d, représentant une itération du domaine d'itération,
- L_{ln} et U_{ln}, matrices diagonales de rang d contenant les bornes inférieures et supérieures du domaine d'itération du nid de boucles.

Un domaine d'itérations d'un nid de boucles dans le domaine séquentiel est défini par un ensemble ln :

$$ln = \{\vec{i} | L_{ln}\vec{1} \leq \vec{i} < U_{ln}\vec{1}\} \tag{4.9}$$

Exemple. L'ensemble des itérations du nid de boucles i, j du listing 4.4 est défini par tous les vecteurs d'entiers à deux dimensions bornés par les matrices L_{ln} et U_{ln} suivantes :

$$L_{ln} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, U_{ln} = \begin{bmatrix} 100 & 0 \\ 0 & 99 \end{bmatrix}$$

```
1 #pragma step gridify(i, j)
2 for(i = 0; i < 100; i++)
3   for(j = 1; j < 99; j++)
4     ...
```

Listing 4.4 – Exemple des matrices L_{ln} et U_{ln}

2) Grille virtuelle de processus pour un ensemble d'itérations

À un nid de boucles ln avec d dimensions gridifiées, on associe une grille virtuelle de processus P_{ln} de rang d. L'ensemble $\mathcal{P}_{ln}$ des processus de la grille virtuelle ainsi construite est défini par :

$$\mathcal{P}_{ln} = \{\tilde{p}_{ln} | 0 \leq \tilde{p}_{ln} < P_{ln} v_1\} \tag{4.10}$$

Remarque. La grille virtuelle de processus d'un nid de boucles ln ne dépend pas des bornes d'itérations U_{ln} et L_{ln} de ce nid de boucles.

3) Bloc d'itérations

On découpe le nombre d'itérations de chaque niveau d'un nid de boucles ln en des blocs d'itérations d'une même taille. Ces tailles de blocs sont représentées par une matrice diagonale B_{ln}. Les tailles de blocs sont définies en fonction du type de distribution.

4) Type de distribution d'une dimension

Distribution par blocs. La taille de bloc d'une dimension k distribuée par blocs (équation 4.11) est calculée en divisant le nombre d'itérations de cette dimension par le nombre de processus disponibles dans la grille virtuelle de processus pour cette dimension. Dans ce cas, le nombre de cycles sur cette dimension est égal à 1.

$$B_{ln_{k,k}} = \lceil (U_{ln_{k,k}} - L_{ln_{k,k}}) / P_{ln_{k,k}} \rceil \tag{4.11}$$

Distribution bloc-cyclique. La taille de bloc est donnée par le programmeur. Le nombre de cycles est calculé à partir de la taille du bloc, du nombre d'itérations sur cette dimension et du nombre de processus affectés à cette dimension dans la grille virtuelle de processus associée au nid de boucles.

Distribution répliquée. Une dimension répliquée affecte un seul processus à la dimension correspondante de la grille virtuelle. La taille de bloc est égale au nombre d'itérations selon cette dimension et le nombre de cycles est égal à 1.

Distribution diagonalisée. On pose la contrainte pratique qu'une dimension diagonalisée ne peut être cyclique même si ce cas est capturé dans le modèle de distribution. La taille de bloc est égale au nombre d'itérations selon cette dimension divisé par le minimum du nombre de processus affectés aux autres dimensions dans la grille virtuelle de processus. Le nombre de cycles est égal à 1.

Distribution all. Une dimension distribuée avec le type *all* ne génère pas de dimension correspondante dans la grille virtuelle : une telle dimension ne consomme pas de processus. La taille de bloc est définie par le programmeur.

Remarque. Dans tous les cas, la taille de bloc pour chaque dimension est non nulle.

$$det(B_{ln}) \neq 0 \tag{4.12}$$

5) Nombre de cycles

Pour chaque dimension d'un nid de boucles, le nombre maximal de cycles est égal au nombre d'itérations selon cette dimension divisé par le produit du nombre de processus affectés à cette dimension et de la taille de bloc selon cette dimension.

On définit C_{ln}, une matrice diagonale de rang d contenant le nombre maximal de cycles pour chaque dimension du nid de boucles.

$$C_{ln} = \lceil (U_{ln} - L_{ln}) B_{ln}^{-1} P_{ln}^{-1} \rceil \tag{4.13}$$

Remarque. L'équation précédente calcule le nombre maximal de cycles. Chaque cycle définit un ensemble d'itérations qu'il faut ensuite contraindre dans le code généré par les bornes d'itérations de ln afin de supprimer les éventuelles itérations superflues introduites (voir la définition de l'ensemble, équation ln 4.9).

L'ensemble des cycles $\mathcal{C}_{ln}$ d'un nid de boucles distribué est alors défini par

$$\mathcal{C}_{ln} = \{\vec{c}_{ln} | 0 \leq \vec{c}_{ln} < C_{ln}\vec{1}\} \tag{4.14}$$

où $\vec{c}_{ln}$ est vecteur de dimension d contenant l'identifiant du cycle pour chaque dimension du nid de boucles.

6) Itération dans un bloc

Pour chaque bloc de taille B_{ln}, on définit un ensemble d'itérations $\mathcal{L}_{ln}$ propres au bloc, définies par :

$$\mathcal{L}_{ln} = \{\vec{l}_{ln} | 0 \leq \vec{l}_{ln} < B_{ln}\vec{1}\} \tag{4.15}$$

7) Ensemble d'itérations dans le domaine distribué Dans le domaine distribué, un domaine d'itérations d'un nid de boucles est défini par un ensemble ln_{dist} des éléments $(\vec{p}_{ln}, \vec{c}_{ln}, \vec{l}_{ln})$ comme suit :

$$ln_{dist} = \mathcal{P}_{ln} \times \mathcal{C}_{ln} \times \mathcal{L}_{ln} \tag{4.16}$$

8) Changement de domaine pour les itérations d'un nid de boucles

On relie une itération $(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}, \vec{l}_{ln}) \in \mathcal{P}_{ln} \times \mathcal{C}_{ln} \times \mathcal{L}_{ln}$ dans le domaine distribué à une itération $\vec{i} \in ln$ dans le domaine séquentiel par l'équation 4.17 qui définit la fonction *loop_nest_iteration*.

$$\vec{i} = L_{ln}\vec{1} + B_{ln}P_{ln}\vec{c}_{ln} + B_{ln}\vec{\tilde{p}}_{ln} + \vec{l}_{ln} \tag{4.17}$$

La figure 4.5 illustre cette équation pour le nid de boucles 2-dimensionnel du listing 4.5.

```
1 #pragma step gridify(i(dist=cyclic(2); sched=...), j)
2 for(i = 0; i < 7; i++)
3   for(j = 0; j < 5; j++)
4     ...
```

Listing 4.5 – Exemple d'un nid de boucles 2-dimensionnel

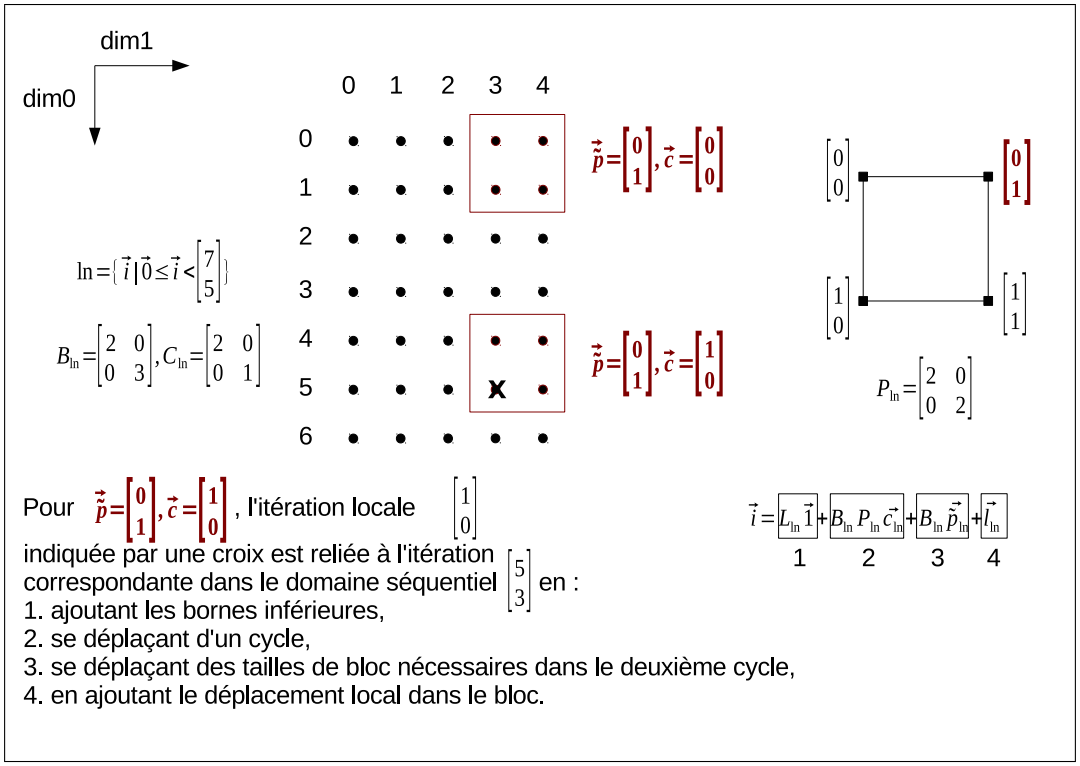


FIGURE 4.5 – Illustration de la fonction *loop_nest_iteration*

Remarque. L'équation 4.17 calcule l'itération $\vec{i}$ pour tout triplet $(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}, \vec{l}_{ln}) \in \mathcal{P}_{ln} \times \mathcal{C}_{ln} \times \mathcal{L}_{ln}$. Il peut ainsi y avoir des itérations $\vec{i}$ superflues qui n'appartiennent pas à ln. Ces itérations sont éliminées par la suite dans le code généré en vérifiant la contrainte $L_{ln}\vec{1} \le \vec{i} < U_{ln}v_1$.

8) Calcul du cycle correspondant à une itération $\vec{i}$

Théorème. 2. $\vec{i} - L_{ln}\vec{1} = B_{ln}P_{ln}\vec{c}_{ln} + B_{ln}\vec{\tilde{p}}_{ln} + \vec{l}_{ln}$ *est une expression de division par* $B_{ln}P_{ln}$.

Preuve. Pour que l'équation $\vec{i} - L_{ln}\vec{1} = B_{ln}P_{ln}\vec{c}_{ln} + B_{ln}\vec{\tilde{p}}_{ln} + \vec{l}_{ln}$ dont tous les termes sont positifs (notamment $\vec{i} \ge L_{ln}\vec{1}$) soit une expression de division par $B_{ln}P_{ln}$, il faut et il suffit que :

1. les matrices diagonales B_{ln} et P_{ln} soient inversibles (déterminants non nuls donc tous les coefficients diagonaux non nuls),

2. $B_{ln}\vec{\tilde{p}}_{ln} + \vec{l}_{ln} < B_{ln}P_{ln}\vec{1}$.

1) On a $det(P_{ln}) \neq 0$ et $det(B_{ln}) \neq 0$ par définition de ces matrices,
2) On a d'une part
$\vec{\tilde{p}}_{ln} \leq P_{ln}\vec{1} - \vec{1}$
donc
$B_{ln}\vec{\tilde{p}}_{ln} \leq B_{ln}(P_{ln}\vec{1} - \vec{1})$,
d'autre part
$\vec{l}_{ln} < B_{ln}\vec{1}$
On peut donc écrire
$B_{ln}\vec{\tilde{p}}_{ln} + \vec{l}_{ln} < B_{ln}(P_{ln}\vec{1} - \vec{1}) + B_{ln}\vec{1}$,
ce qui équivaut à
$B_{ln}\tilde{p}_{ln} + \vec{l}_{ln} < B_{ln}P_{ln}\vec{1}$
De 1 et 2 on déduit que $\vec{i} - L_{ln}\vec{1} = B_{ln}P_{ln}\vec{c}_{ln} + B_{ln}\vec{\tilde{p}}_{ln} + \vec{l}_{ln}$ est une expression de division par $B_{ln}P_{ln}$. □

$\vec{c}_{ln}$ est le résultat d'une division par $B_{ln}P_{ln}$. On écrit :

$$\vec{c}_{ln} = \lfloor P_{ln}^{-1}B_{ln}^{-1}(\vec{i} - L_{ln}\vec{1})\rfloor \tag{4.18}$$

Remarque. Cette équation exprime le fait qu'une itération du nid de boucles appartient à un cycle unique.

On définit la fonction *iteration_cycle* qui calcule ce cycle :

$$\begin{array}{rll} iteration_cycle: & ln \rightarrow & \mathcal{C}_{ln} \\ & \vec{i} \mapsto & \vec{c}_{ln} = \lfloor P_{ln}^{-1}B_{ln}^{-1}(\vec{i} - L_{ln}\vec{1})\rfloor \end{array} \tag{4.19}$$

9) Calcul du processus exécutant une itération $\vec{i}$

On construit à partir de l'équation 4.17 l'équation suivante :

$$\vec{i} - L_{ln}\vec{1} = B_{ln}(\vec{\tilde{p}}_{ln} + P_{ln}\vec{c}_{ln}) + \vec{l}_{ln} \tag{4.20}$$

Théorème. 3. *L'équation $\vec{i} - L_{ln}\vec{1} = B_{ln}(\vec{p}_{ln} + P_{ln}\vec{c}_{ln}) + \vec{l}_{ln}$ est une expression de division par B_{ln}.*

Preuve. Tous les termes de l'équation sont positifs. Il faut et il suffit donc de montrer que $det(B_{ln}) \neq 0$ et que $\vec{l}_{ln} < B_{ln}\vec{1}$.

1) Par définition, la matrice diagonale B_{ln} ne possède que des coefficients non nuls sur sa diagonale, donc $det(B_{ln}) \neq 0$,

2) $\vec{l}_{ln} < B_{ln}\vec{1}$ par la définition de l'ensemble $\mathcal{L}_{ln}$ (4.15)

On en déduit que $\vec{i} - L_{ln}\vec{1} = B_{ln}(\vec{\tilde{p}}_{ln} + P_{ln}\vec{c}_{ln}) + \vec{l}_{ln}$ est une expression de division par B_{ln}. □

On peut donc écrire :

$$\vec{\tilde{p}}_{ln} + P_{ln}\vec{c}_{ln} = \lfloor B_{ln}^{-1}(\vec{i} - L_{ln}\vec{1})\rfloor \tag{4.21}$$

Donc

$$\vec{\tilde{p}}_{ln} = \lfloor B_{ln}^{-1}(\vec{i} - L_{ln}\vec{1})\rfloor - P_{ln}\vec{c}_{ln} \tag{4.22}$$

Remarque. Cette équation exprime le fait qu'une itération est exécutée par un processus unique.

On définit la fonction *iteration_executor* qui associe à une itération $\vec{i}$ le processus $\tilde{p}_{ln}$ qui l'exécute.

$$\begin{array}{lll} iteration_executor : & ln \rightarrow & \mathcal{P}_{ln} \\ & \vec{i} \mapsto & \vec{\tilde{p}}_{ln} = \lfloor B_{ln}^{-1}(\vec{i} - L_{ln}\vec{1})\rfloor - P_{ln} iteration_cycle(\vec{i}) \end{array} \tag{4.23}$$

10) Calcul du déplacement local dans un bloc correspondant à une itération $\vec{i}$

On construit à partir de l'équation 4.17 l'équation suivante qui calcule l'itération locale dans un bloc :

$$\vec{l}_{ln} = \vec{i} - L_{ln}\vec{1} - B_{ln}P_{ln}\vec{c}_{ln} - B_{ln}\vec{\tilde{p}}_{ln} \tag{4.24}$$

11) Exemple de distribution d'un nid de boucles

```
1 #pragma step gridify(i, j)
2   for (i=1; i <  1024; i++)
3     for (j=1; j < 1024; j++) {
4       double neighbour = cos(table[i-1][j]) + sin(table[i][j-1]) +
5                          sin(table[i][j+1]) + cos(table[i+1][j]);
6       tableOut[i][j]   = neighbour/3;
7     }
```

Listing 4.6 – Distribution d'un nid de boucles sans calculs redondants

Le code du listing 4.6 montre un nid de boucles ln distribué sur ses dimensions i et j. Si l'on exécute la distribution sur quatre processus, on aura :

$L_{ln} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, U_{ln} = \begin{bmatrix} 1024 & 0 \\ 0 & 1024 \end{bmatrix}, P_{ln} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, B_{ln} = \begin{bmatrix} 512 & 0 \\ 0 & 512 \end{bmatrix}$

Soit l'itération $\begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} 600 \\ 200 \end{bmatrix}$ dans le domaine séquentiel. En appliquant l'équation 4.18 pour le calcul du cycle, on trouve :

$\vec{c}_{ln} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

En appliquant l'équation 4.22 pour le calcul de l'identifiant du processus, on trouve :

$\vec{\tilde{p}}_{ln} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Et en appliquant l'équation 4.24 pour le calcul de l'itération locale, on trouve :

$\vec{l}_{ln} = \begin{bmatrix} 87 \\ 199 \end{bmatrix}$

4.2.6 Modèle de distribution de tableau avec halo

Cette section utilise des notions similaires à celles présentées à la section précédente. On remplace une itération par un élément de tableau, un ensemble d'itérations par un ensemble d'éléments de tableau et un bloc d'itérations par un bloc d'éléments de tableaux. Les matrices P_{ln}, B_{ln} et C_{ln} deviennent P_x, B_x et C_x pour un tableau x. On ajoute le *halo* au modèle de distribution.

1) Tableau dans le domaine séquentiel

Dans le domaine séquentiel est caractérisé par :

- d un entier représentant le nombre de dimensions du tableau,
- $\vec{a}$ un vecteur d'entiers de dimension d, contenant les indices situant un élément dans le tableau,
- D_x une matrice diagonale de rang d contenant les tailles du tableau selon chaque dimension distribuée.

L' ensemble x des éléments d'un tableau de dimensions D_x est défini par :

$$x = \{\vec{a} | 0 \leq \vec{a} < D_x \vec{1}\} \tag{4.25}$$

Exemple.

```
double A[M][N];
```

Le tableau `A` est défini par la matrice $D_A = \begin{bmatrix} M & 0 \\ 0 & N \end{bmatrix}$

2) Grille virtuelle de processus d'un tableau

À un tableau x distribué sur d dimensions, on associe une grille virtuelle de processus P_x de rang d. L'ensemble des processus de la grille virtuelle ainsi construite est défini par :

$$\mathcal{P}_x = \{\vec{\tilde{p}}_x | \vec{0} \leq \vec{\tilde{p}}_x < P_x \vec{1}\} \tag{4.26}$$

Remarque. La grille virtuelle P_x ne dépend pas des tailles des dimensions du tableau x. Elle dépend du nombre de processus disponibles, du nombre de dimensions du tableau x et du type de distribution pour chaque dimension.

3) Bloc d'éléments de tableau

On découpe le nombre d'éléments de chaque dimension d'un tableau x en des blocs d'éléments d'une même taille. Ces tailles de blocs sont représentées par une matrice diagonale B_x. Comme pour les itérations, il existe quatre façons de définir les tailles de blocs : *bloc*, *bloc-cyclique*, *répliquée* et *diagonalisée*.

4) Nombre de cycles

De façon similaire aux cycles d'un nid de boucles distribué, on définit le nombre maximal de cycles pour un tableau x de taille D_x distribué sur une grille virtuelle P_x en des blocs B_x par la matrice C_x :

$$C_x = \lceil D_x B_x^{-1} P_x^{-1} \rceil \tag{4.27}$$

Remarque. L'équation précédente peut introduire un cycle superflu sur certains processus, ce qui conduit à allouer dans le domaine distribué de la mémoire non utilisée. On peut optimiser le code généré pour ne pas allouer ces éléments de tableaux superflus mais cette optimisation n'est pas nécessaire à la correction du code généré (voir les propriétés du modèle).

L'ensemble $\mathcal{C}_x$ des cycles d'un tableau distribué est défini par :

$$\mathcal{C}_x = \{c_x | \vec{0} \leq \vec{c}_x < C_x \vec{1}\} \tag{4.28}$$

avec $\vec{c}_x$ un vecteur de taille d contenant l'identifiant du cycle pour chaque dimension du tableau.

5) Élément de tableau dans un bloc avec halo

Pour chaque bloc B_x, on définit un ensemble d'éléments propres au bloc définis par l'ensemble $\mathcal{L}'_x$:

$$\mathcal{L}'_x = \{\vec{l}_x | 0 \leq \vec{l}_x < B_x \vec{1}\} \tag{4.29}$$

En présence de halo, on étend cet ensemble d'éléments avec les quantités H_{low_x} et H_{up_x} pour définir l'ensemble $\mathcal{L}_x$

$$\mathcal{L}_x = \{\vec{l}_x | \vec{0} \leq \vec{l}_x < (B_x + H_{low_x} + H_{up_x})\vec{1}\} \tag{4.30}$$

6) Changement de domaine pour les éléments d'un tableau

On relie un élément de tableau $\vec{a} \in x$ dans le domaine séquentiel et un élément de tableau $(\vec{\tilde{p}}_x, \vec{c}_x, \vec{l}_x) \in \mathcal{P}_x \times \mathcal{B}_x \times \mathcal{L}_x$ dans le domaine distribué par l'équation 4.31 qui définit la fonction *array_element*.

$$\vec{a} = B_x P_x \vec{c}_x + B_x \vec{\tilde{p}}_x - H_{low_x} \vec{1} + \vec{l}_x \tag{4.31}$$

7) Calcul du cycle correspondant à un élément de tableau $\vec{a}$ et à un processus $\vec{\tilde{p}}_x$

A partir de l'équation 4.31, on écrit :

$$\vec{a} - B_x\vec{\tilde{p}}_x + H_{low_x}\vec{1} = B_xP_x\vec{c}_x + \vec{l}_x \tag{4.32}$$

Contrainte de validité des halos. Pour pouvoir associer un élément de tableau unique $\vec{a} \in x$ à un triplet $(\vec{\tilde{p}}_x, \vec{c}_x, \vec{l}_x) \in \mathcal{P}_x \times \mathcal{B}_x \times \mathcal{L}_x$ pour un identifiant de processus $\vec{\tilde{p}}_x$ fixé, il faut que les éléments $\vec{l}_x$ de cycles différents ne se recouvrent pas sur $\vec{\tilde{p}}_x$. Cette contrainte est exprimée par l'inéquation 4.33.

$$\vec{0} \leq (Hlow_x + Hup_x)\vec{1} \leq (B_xP_x - B_x)\vec{1} \tag{4.33}$$

Illustration. La figure 4.6 montre un contre-exemple où le halo est invalide. Le tableau mono-dimensionnel `A` de taille seize est distribué cycliquement avec une taille de bloc valant quatre et en utilisant un halo inférieur et supérieur égal à trois. Si cette distribution est exécutée sur deux processus et que l'on considère l'élément `A[5]` dans l'espace séquentiel, on constate que cet élément se trouve alloué dans l'espace distribué sur le processus $\vec{\tilde{p}} = \begin{bmatrix}0\end{bmatrix}$ à la fois sur le cycle $\vec{c} = \begin{bmatrix}0\end{bmatrix}$ et $\vec{c} = \begin{bmatrix}1\end{bmatrix}$. On ne sait pas dans ce cas associer un cycle unique à certains éléments de l'espace séquentiel sur un processus donné. Le problème vient de la violation de la contrainte de validité des halos : $\begin{bmatrix}0\end{bmatrix} \leq \begin{bmatrix}3\end{bmatrix} + \begin{bmatrix}3\end{bmatrix} \leq \begin{bmatrix}4\end{bmatrix} \times \begin{bmatrix}2\end{bmatrix} - \begin{bmatrix}3\end{bmatrix} \iff \begin{bmatrix}0\end{bmatrix} \leq \begin{bmatrix}6\end{bmatrix} \leq \begin{bmatrix}5\end{bmatrix}$.

Théorème. 4. *Pour $(\vec{a} - B_x\vec{\tilde{p}}_x + Hlow_x\vec{1}) \geq \vec{0}$, si $\vec{0} \leq Hlow_x + Hup_x \leq B_xP_x - B_x$ alors l'équation $\vec{a} - B_x\vec{\tilde{p}}_x + Hlow_x\vec{1} = B_xP_x\vec{c}_x + \vec{l}_x$ est une expression de division par B_xP_x.*

Preuve du théorème 4. Pour calculer un cycle unique, il faut que l'équation 4.32 soit une expression de division par B_xP_x. Il faut donc que

$$\vec{l}_x < B_xP_x\vec{1} \tag{4.34}$$

On a par définition $\vec{0} \leq \vec{l}_x \leq (Hlow_x + B_x + Hup_x)\vec{1} - \vec{1}$. La contrainte 4.34 est donc équivalente à la contrainte sur les halos :

$$(Hlow_x + B_x + Hup_x)\vec{1} - \vec{1} < B_xP_x\vec{1} \tag{4.35}$$

1) On a, par définition des matrices P_x et B_x, $det(P_x) \neq 0$ et $det(B_x) \neq 0$
2) On a, en respectant la contrainte sur la validité des halos 4.35

$$\vec{0} \leq \vec{l}_x < P_xB_x\vec{1} \tag{4.36}$$

De 1 et 2 on déduit que pour $(\vec{a} - B_x\vec{\tilde{p}}_x + Hlow_x\vec{1})$ positif et un halo valide, c'est-à-dire qui vérifie la contrainte 4.35, l'équation $\vec{a} - B_x\vec{\tilde{p}}_x + Hlow_x\vec{1} = B_xP_x\vec{c}_x + \vec{l}_x$

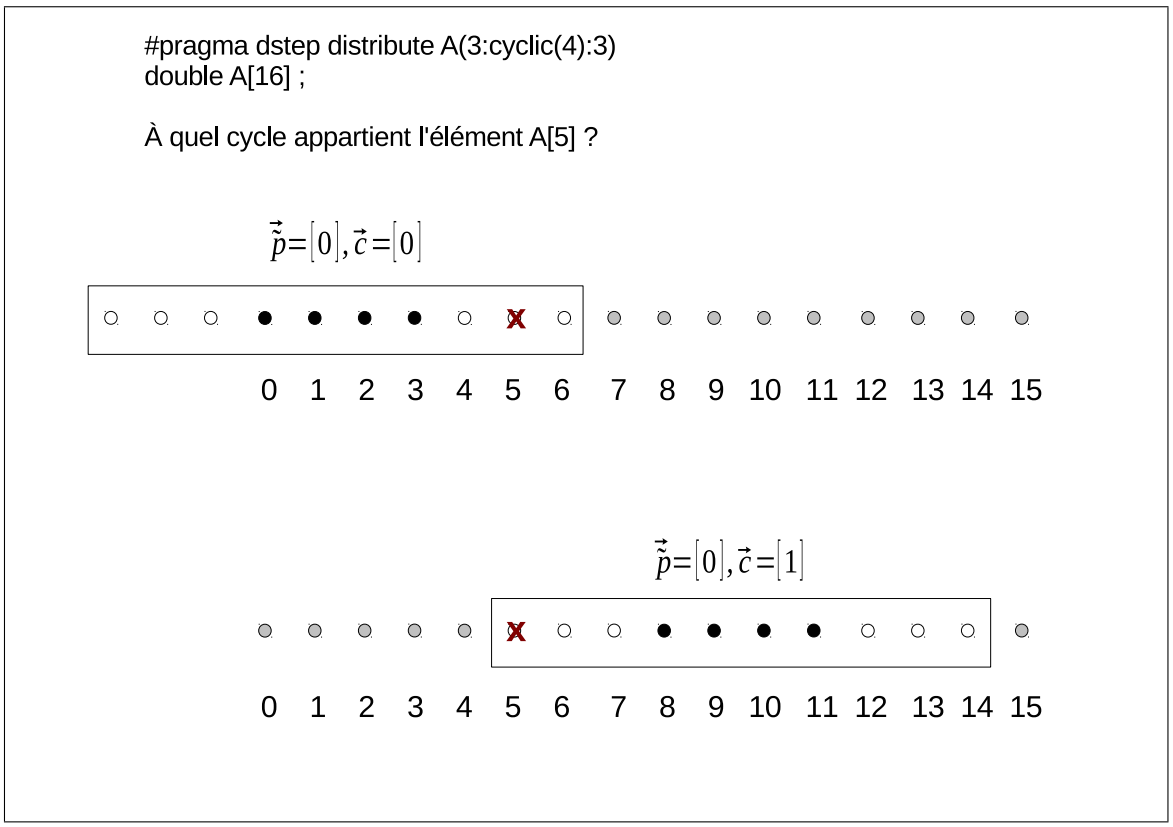


FIGURE 4.6 – Illustration de la violation de la contrainte de validité des halos

est une expression de division par $B_x P_x$. □

On peut donc calculer l'unique cycle correspondant :

$$\vec{c}_x = \lfloor P_x^{-1} B_x^{-1} (\vec{a} - B_x \vec{\tilde{p}}_x + Hlow_x \vec{1}) \rfloor \tag{4.37}$$

Remarques.

1. Lorsque la contrainte sur les halos n'est pas respectée, la distribution est incorrecte. Dans le code généré, la validité de cette contrainte est vérifiée et un message d'erreur est affiché au programmeur lorsqu'elle n'est pas vérifiée.
2. Pour une dimension distribuée sans cycle (nombre de cycles égal à 1), la contrainte sur les halos est toujours vérifiée. Le halo pour une telle dimension peut alors recouvrir toute l'étendue de la dimension et la distribution reste correcte (c'est le cas du halo total).
3. Si $\vec{a} - B_x \vec{\tilde{p}}_x + H_{low_x} \vec{1} < \vec{0}$ ou $(\lfloor P_x^{-1} B_x^{-1} (\vec{a} - B_x \vec{\tilde{p}}_x + Hlow_x \vec{1}) \rfloor) \notin \mathcal{C}$ alors le cycle n'est pas défini pour le point $(\vec{a}, \vec{\tilde{p}}_x)$. On définit la fonction $cycle_defined$ qui vérifie ces conditions.

On définit la fonction partielle $array_cycle$ qui calcule un cycle unique $\vec{c}_x$ lorsque ce cycle est défini pour un point $(\vec{a}, \vec{\tilde{p}}_x)$:

$$\begin{aligned} array_cycle: \quad & x \times \mathcal{P}_x \to \mathcal{C}_x \\ & (\vec{a}, \vec{\tilde{p}}_x) \to \lfloor P_x^{-1} B_x^{-1} (\vec{a} - B_x \vec{\tilde{p}}_x + Hlow_x \vec{1}) \rfloor \end{aligned} \tag{4.38}$$

8) Calcul du déplacement local correspondant à un élément de tableau $\vec{a}$ et à un processus $\vec{\tilde{p}}_x$.

On construit à partir de l'équation 4.31 l'équation suivante qui calcule le déplacement local :

$$\vec{l}_x = \vec{a} - P_x B_x \vec{c}_x - B_x \vec{\tilde{p}}_x + Hlow_x \vec{1} \tag{4.39}$$

Remarque. Si $\vec{l}_x \notin \mathcal{L}_x$, alors le déplacement local n'est pas défini pour le point $(\vec{a}, \vec{\tilde{p}}_x, \vec{c}_x)$. On définit la fonction *local_offset_defined* qui vérifie cette condition.

On définit la fonction partielle *local_offset* qui calcule le déplacement local pour les points $(\vec{a}, \vec{\tilde{p}}_x, \vec{c}_x)$.

$$\begin{aligned} local_offset: \quad & x \times \mathcal{P}_x \times \mathcal{C}_x \to \mathcal{L}_x \\ & (\vec{a}, \vec{\tilde{p}}_x, \vec{c}_x) \to \vec{a} - P_x B_x \vec{c}_x - B_x \vec{\tilde{p}}_x + Hlow_x \vec{1} \end{aligned} \tag{4.40}$$

9) Exemples de distribution de tableau avec halo.

On se donne un tableau `A` à deux dimensions de taille 8×8. Soit l'élément `A[5][3]` dans le domaine séquentiel. On veut calculer le représentant de cet élément dans le domaine distribué sur chaque processus pour plusieurs types de distribution.

```
1 #step distribute table(block, 1:block:1)
2 double A[16][16];
```

Listing 4.7 – Distribution par blocs

a) Distribution 2-dimensionnelle, sans cycle. La figure 4.7 montre l'exécution de la distribution du listing 4.7 sur quatre processus. Voici les matrices qui définissent cette distribution :

$$D_A = \begin{bmatrix} 8 & 0 \\ 0 & 8 \end{bmatrix}, P_A = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, B_A = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}, Hlow_A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, Hup_A = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

Nombre maximal de cycles. Le nombre de cycles est calculé par l'équation 4.27 :

$$C_x = \left\lceil \begin{bmatrix} 8 & 0 \\ 0 & 8 \end{bmatrix} \begin{bmatrix} 1/4 & 0 \\ 0 & 1/4 \end{bmatrix} \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} \right\rceil = \left\lceil \begin{bmatrix} 8/8 & 0 \\ 0 & 8/8 \end{bmatrix} \right\rceil = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

On a donc un seul cycle sur chaque dimension.

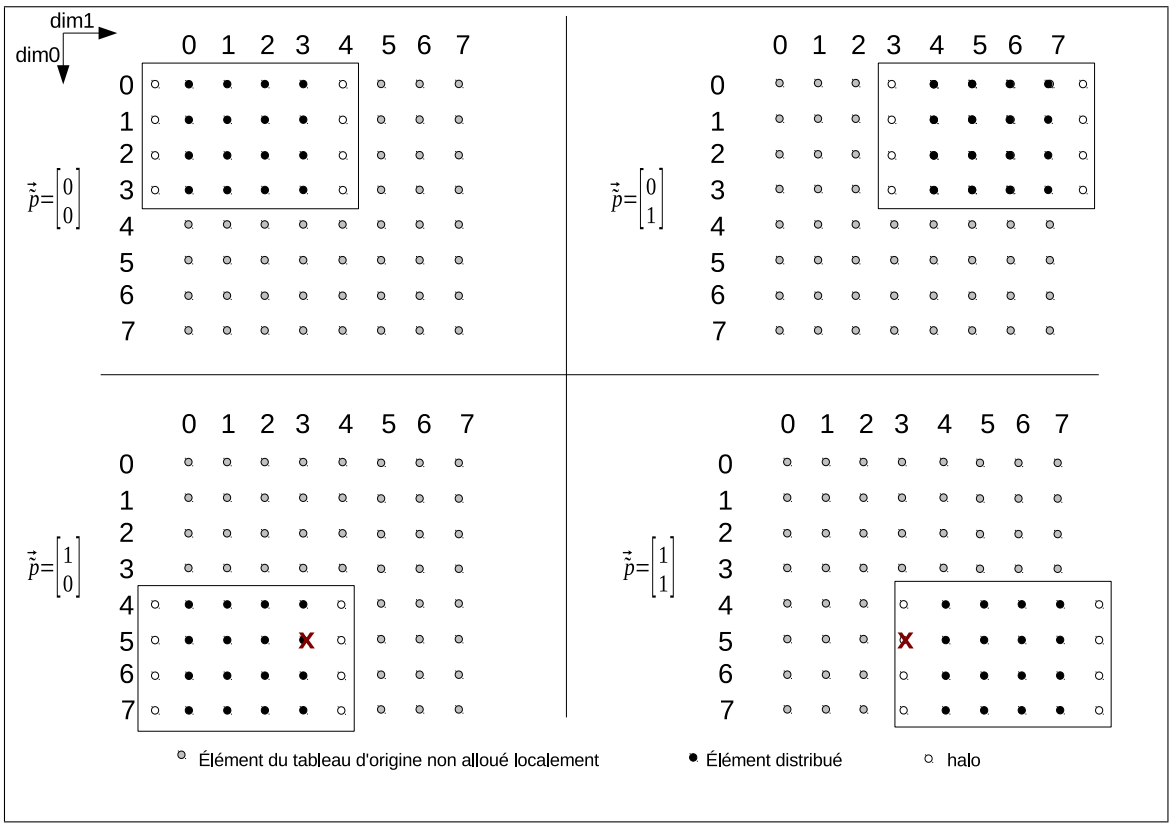


FIGURE 4.7 – Distribution bidimensionnelle par blocs avec halo

Calcul du cycle sur $\vec{\tilde{p}} = \begin{bmatrix}0\\0\end{bmatrix}$

On applique la fonction *array_cycle* 4.38.

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/2 & 0\\0 & 1/2\end{bmatrix}\begin{bmatrix}1/4 & 0\\0 & 1/4\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8 & 0\\0 & 1/8\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8 & 0\\0 & 1/8\end{bmatrix}\begin{bmatrix}5\\4\end{bmatrix}\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}5/8\\3/8\end{bmatrix}\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right) = \begin{bmatrix}1\\1\end{bmatrix}$$

Le cycle calculé est non défini car ne vérifie pas la contrainte $\begin{bmatrix}0\\0\end{bmatrix} \leq \begin{bmatrix}1\\1\end{bmatrix} < \begin{bmatrix}1\\1\end{bmatrix}$.

On en déduit qu'il n'y a pas de représentant local de l'élément $\begin{bmatrix}5\\3\end{bmatrix}$ sur le processus $\begin{bmatrix}0\\0\end{bmatrix}$

Calcul du cycle sur $\vec{\tilde{p}} = \begin{bmatrix}0\\1\end{bmatrix}$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/2 & 0\\0 & 1/2\end{bmatrix}\begin{bmatrix}1/4 & 0\\0 & 1/4\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8 & 0\\0 & 1/8\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}0\\4\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8 & 0\\0 & 1/8\end{bmatrix}\begin{bmatrix}5\\0\end{bmatrix}\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}5/8\\0/8\end{bmatrix}\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right) = \begin{bmatrix}1\\0\end{bmatrix}$$

Le cycle calculé est non défini car ne vérifie pas la contrainte $\begin{bmatrix}0\\0\end{bmatrix} \leq \begin{bmatrix}1\\0\end{bmatrix} < \begin{bmatrix}1\\1\end{bmatrix}$.

On en déduit qu'il n'y a pas de représentant local de l'élément $\begin{bmatrix}5\\3\end{bmatrix}$ sur le processus $\begin{bmatrix}0\\1\end{bmatrix}$

Calcul du cycle sur $\vec{\tilde{p}} = \begin{bmatrix}1\\0\end{bmatrix}$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/2 & 0\\0 & 1/2\end{bmatrix}\begin{bmatrix}1/4 & 0\\0 & 1/4\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8 & 0\\0 & 1/8\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}4\\0\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8 & 0\\0 & 1/8\end{bmatrix}\begin{bmatrix}1\\4\end{bmatrix}\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8\\4/8\end{bmatrix}\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}\right) = \begin{bmatrix}0\\0\end{bmatrix}$$

Le cycle calculé est parfaitement valide. On peut donc calculer le représentant local de l'élément considéré.

Calcul du déplacement local. On calcule maintenant le déplacement local avec l'équation 4.39 :

$$l_A = local_offset(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}) = \begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}\begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$$

$$l_A = \begin{bmatrix}1\\4\end{bmatrix}$$

Ce déplacement local est correct car vérifie la contrainte $\begin{bmatrix}0\\0\end{bmatrix} \leq \begin{bmatrix}1\\4\end{bmatrix} < \begin{bmatrix}4\\6\end{bmatrix}$.

Calcul du cycle sur $\vec{\tilde{p}} = \begin{bmatrix}1\\1\end{bmatrix}$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/2 & 0\\0 & 1/2\end{bmatrix}\begin{bmatrix}1/4 & 0\\0 & 1/4\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8 & 0\\0 & 1/8\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}4\\4\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8 & 0\\0 & 1/8\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/8\\0/8\end{bmatrix}\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right) = \begin{bmatrix}0\\0\end{bmatrix}$$

Le cycle calculé est parfaitement valide. On peut donc calculer le représentant local de l'élément considéré.

Calcul du déplacement local. On calcule maintenant le déplacement local avec l'équation 4.39 :

$$l_A = local_offset(\begin{bmatrix}5\\3\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}0\\0\end{bmatrix}) = \begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}\begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$$

$$l_A = \begin{bmatrix}1\\0\end{bmatrix}$$

Ce déplacement local est correct car vérifie la contrainte $\begin{bmatrix}0\\0\end{bmatrix} \leq \begin{bmatrix}1\\0\end{bmatrix} < \begin{bmatrix}4\\6\end{bmatrix}$.

```
1 #step distribute table(block, 1:cyclic(2):1)
2 double A[16][16];
```

Listing 4.8 – Distribution cyclique

b) Distribution bi-dimensionnelle, cyclique sur la deuxième dimension. La figure 4.8 montre l'exécution de la distribution du listing 4.8 sur quatre processus. Voici les matrices qui définissent cette distribution :

$$D_A = \begin{bmatrix}8 & 0\\0 & 8\end{bmatrix}, P_A = \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}, B_A = \begin{bmatrix}4 & 0\\0 & 2\end{bmatrix}, Hlow_A = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}, Hup_A = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}$$

Nombre maximal de cycles. Le nombre de cycles est calculé par l'équation 4.27 :

$$C_x = \left\lceil \begin{bmatrix}8 & 0\\0 & 8\end{bmatrix}\begin{bmatrix}1/4 & 0\\0 & 1/2\end{bmatrix}\begin{bmatrix}1/2 & 0\\0 & 1/2\end{bmatrix}\right\rceil = \left\lceil \begin{bmatrix}8/8 & 0\\0 & 8/4\end{bmatrix}\right\rceil = \begin{bmatrix}1 & 0\\0 & 2\end{bmatrix}$$

On a donc un seul cycle selon la première dimension et deux cycles selon la deuxième.

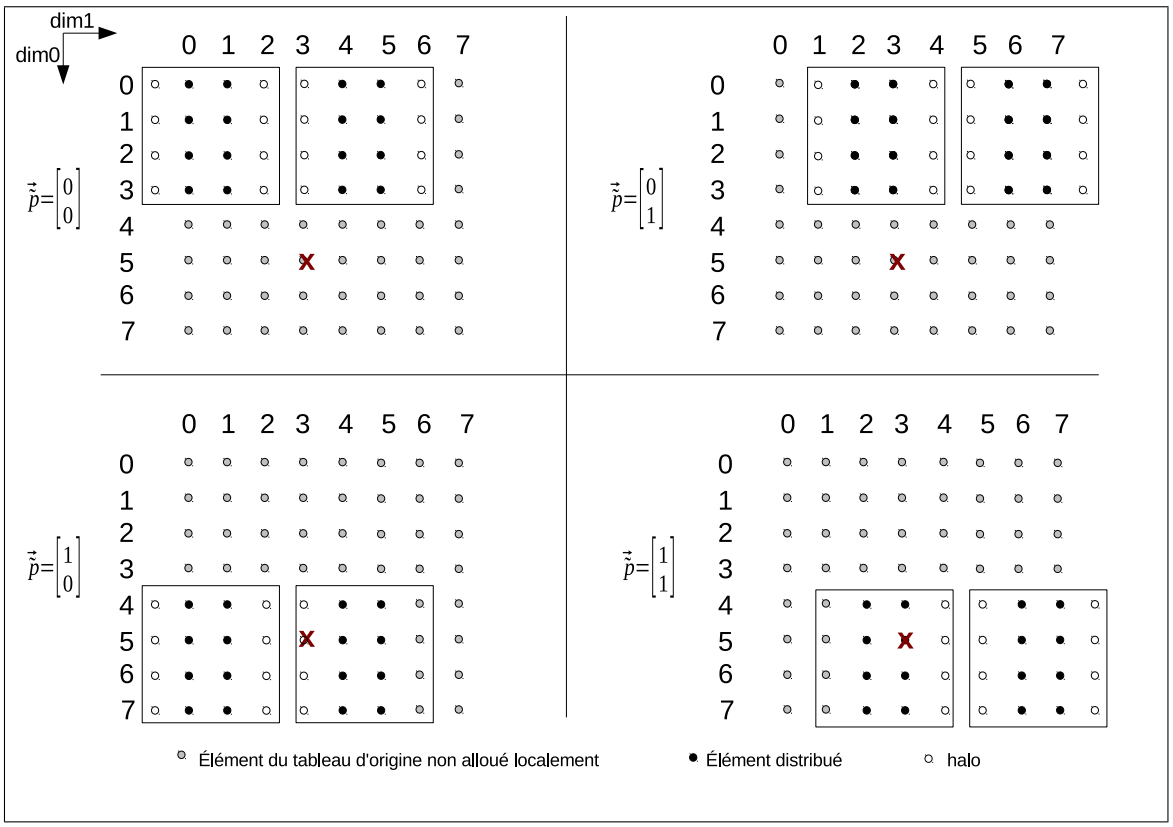


FIGURE 4.8 – Distribution bi-dimensionnelle cyclique avec halo

Calcul du cycle sur $\vec{\tilde{p}} = \begin{bmatrix}0\\0\end{bmatrix}$

On applique la fonction *array_cycle* 4.38.

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right) = \left\lfloor \begin{bmatrix}1/2 & 0\\0 & 1/2\end{bmatrix}\begin{bmatrix}1/4 & 0\\0 & 1/2\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 2\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right) = \begin{bmatrix}0\\1\end{bmatrix}$$

Le cycle calculé est défini.

Calcul du déplacement local.

$$\vec{l}_A = local_offset(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}) = \begin{bmatrix}5\\3\end{bmatrix} - \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}\begin{bmatrix}4 & 0\\0 & 2\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$$

$$\vec{l}_A = \begin{bmatrix}5\\0\end{bmatrix}$$

Ce déplacement local n'est pas défini car ne vérifie pas $\vec{0} \leq \begin{bmatrix}5\\0\end{bmatrix} < \begin{bmatrix}4\\4\end{bmatrix}$. On en conclut que l'élément $\vec{a} = \begin{bmatrix}5\\3\end{bmatrix}$ n'a pas de représentant local sur $\vec{\tilde{p}} = \begin{bmatrix}0\\0\end{bmatrix}$.

Calcul du cycle sur $\vec{\tilde{p}} = \begin{bmatrix}0\\1\end{bmatrix}$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix}\right)=\left\lfloor\begin{bmatrix}1/2&0\\0&1/2\end{bmatrix}\begin{bmatrix}1/4&0\\0&1/2\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix}-\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}+\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right)=\begin{bmatrix}0\\0\end{bmatrix}$$

Le cycle calculé est défini.

Calcul du déplacement local.

$$\vec{l}_A=local_offset(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix})=\begin{bmatrix}5\\3\end{bmatrix}-\begin{bmatrix}2&0\\0&2\end{bmatrix}\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}-\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}+\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$$

$$\vec{l}_A=\begin{bmatrix}5\\0\end{bmatrix}$$

Ce déplacement local n'est pas défini car ne vérifie pas $\vec{0}\leq\begin{bmatrix}5\\0\end{bmatrix}<\begin{bmatrix}4\\4\end{bmatrix}$. On en conclut que l'élément $\vec{a}=\begin{bmatrix}5\\3\end{bmatrix}$ n'a pas de représentant local sur $\vec{\tilde{p}}=\begin{bmatrix}0\\1\end{bmatrix}$.

Calcul du cycle sur $\vec{\tilde{p}}=\begin{bmatrix}1\\0\end{bmatrix}$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix}\right)=\left\lfloor\begin{bmatrix}1/2&0\\0&1/2\end{bmatrix}\begin{bmatrix}1/4&0\\0&1/2\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix}-\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}+\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix}\right)=\begin{bmatrix}0\\1\end{bmatrix}$$

Le cycle calculé est défini.

Calcul du déplacement local.

$$\vec{l}_A=local_offset(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\0\end{bmatrix},\begin{bmatrix}0\\1\end{bmatrix})=\begin{bmatrix}5\\3\end{bmatrix}-\begin{bmatrix}2&0\\0&2\end{bmatrix}\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}-\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}+\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$$

$$\vec{l}_A=\begin{bmatrix}1\\0\end{bmatrix}$$

Ce déplacement local est défini.

Calcul du cycle sur $\vec{\tilde{p}}=\begin{bmatrix}1\\1\end{bmatrix}$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}\right)=\left\lfloor\begin{bmatrix}1/2&0\\0&1/2\end{bmatrix}\begin{bmatrix}1/4&0\\0&1/2\end{bmatrix}\left(\begin{bmatrix}5\\3\end{bmatrix}-\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}+\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}\right)\right\rfloor$$

$$array_cycle\left(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix}\right)=\begin{bmatrix}0\\0\end{bmatrix}$$

Le cycle calculé est défini.

Calcul du déplacement local.

$$\vec{l}_A=local_offset(\begin{bmatrix}5\\3\end{bmatrix},\begin{bmatrix}1\\1\end{bmatrix},\begin{bmatrix}0\\0\end{bmatrix})=\begin{bmatrix}5\\3\end{bmatrix}-\begin{bmatrix}2&0\\0&2\end{bmatrix}\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix}-\begin{bmatrix}4&0\\0&2\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}+\begin{bmatrix}0&0\\0&1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix}$$

$$\vec{l}_A=\begin{bmatrix}1\\2\end{bmatrix}$$

Ce déplacement local est défini.

```
#step distribute table(block, 1:diag:1)
double A[16][16];
```

Listing 4.9 – Distribution multi-partitionnée

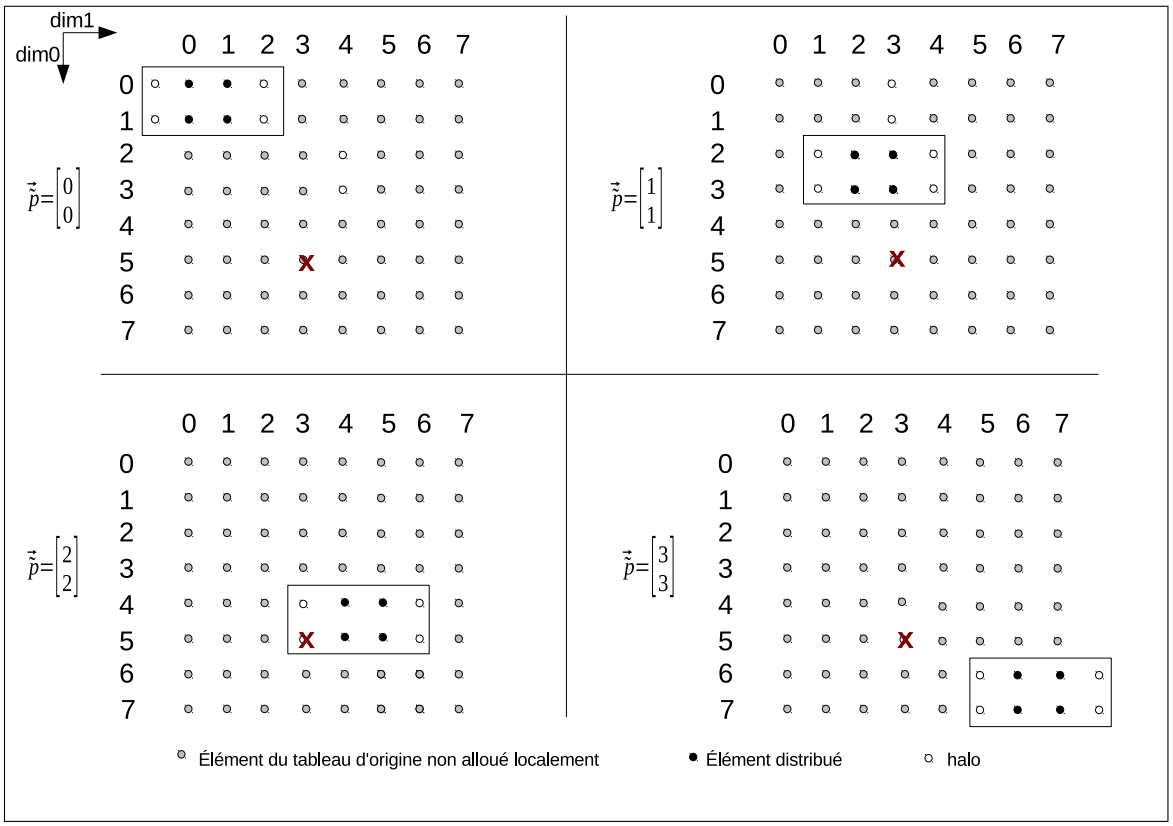


FIGURE 4.9 – Distribution bi-dimensionnelle multi-partitionnée

c) Distribution bi-dimensionnelle, diagonalisée sur la deuxième dimension. La figure 4.9 montre l'exécution de la distribution du listing 4.9 sur quatre processus. La grille virtuelle correspondante est constituée de seize processus virtuels. Pour simplifier, nous ne montrons la distribution que sur les processus virtuels correspondant au processus $\begin{bmatrix}0\\0\end{bmatrix}$. Voici les matrices qui définissent cette distribution :

$$D_A = \begin{bmatrix}8 & 0\\0 & 8\end{bmatrix}, P_A = \begin{bmatrix}4 & 0\\0 & 4\end{bmatrix}, B_A = \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}, Hlow_A = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}, Hup_A = \begin{bmatrix}0 & 0\\0 & 1\end{bmatrix}$$

Nombre maximal de cycles. Le nombre de cycles est calculé par l'équation 4.27 :
$C_x = \left\lceil \begin{bmatrix}8 & 0\\0 & 8\end{bmatrix} \begin{bmatrix}1/2 & 0\\0 & 1/2\end{bmatrix} \begin{bmatrix}1/4 & 0\\0 & 1/4\end{bmatrix} \right\rceil = \left\lceil \begin{bmatrix}8/8 & 0\\0 & 8/8\end{bmatrix} \right\rceil = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$

Pour simplifier, nous ne montrons le calcul du cycle et du déplacement local que sur le processus étendu $\vec{\tilde{p}} = \begin{bmatrix}2\\2\end{bmatrix}$ qui possède un représentant de l'élément $\vec{a} = \begin{bmatrix}5\\3\end{bmatrix}$.

Calcul du cycle sur $\vec{\tilde{p}} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$

$$array_cycle\left(\begin{bmatrix} 5 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \end{bmatrix}\right) = \left\lfloor \begin{bmatrix} 1/4 & 0 \\ 0 & 1/4 \end{bmatrix} \begin{bmatrix} 1/2 & 0 \\ 0 & 1/2 \end{bmatrix} \left(\begin{bmatrix} 5 \\ 3 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \right\rfloor$$

$$array_cycle\left(\begin{bmatrix} 5 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Le cycle calculé est défini.

Calcul du déplacement local.
$\vec{l}_A = local_offset(\begin{bmatrix} 5 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}) = \begin{bmatrix} 5 \\ 3 \end{bmatrix} - \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$
. Ce déplacement local est défini.

4.3 Modèle combiné de distribution des calculs et des données

Les références aux éléments de tableau construisent une correspondance entre les itérations et les éléments de tableau. Une référence à un élément de tableau dans un nid de boucles est une fonction du vecteur itération du nid de boucles. Dans le domaine distribué, chaque processus exécute un ensemble de tranches d'itérations. Il faut s'assurer que pour chaque référence à un élément de tableau accédé pour une itération donnée, il existe un représentant local de cet élément, que ce représentant est unique et il faut pouvoir calculer la nouvelle référence dans le domaine distribué. Il faut également pouvoir retrouver tous les représentants d'un tel élément sur tous les autres processus afin de mettre à jour les données répliquées si nécessaire.

4.3.1 Définitions

1) Référence à un élément de tableau dans le domaine séquentiel : La $m^{ième}$ référence à un élément de tableau x dans le domaine séquentiel est une fonction ref_x^m :

$$\begin{aligned} ref_x^m : \quad & ln \rightarrow x \\ & \vec{i} \mapsto \vec{a} \end{aligned} \tag{4.41}$$

2) Ensembles des références d'un nid de boucles. On définit REF_{ln}, l'ensemble des références à des tableaux dans un nid de boucles ln et $REF_{ln}(X)$ l'ensemble des références à un tableau particulier X.

3) Référence en lecture, référence en écriture. Les fonctions is_read_ref et is_write_ref, appliquées à une référence, indiquent s'il s'agit d'une référence en lecture ou en écriture, respectivement.

4.3.2 Fonctions de changement de domaine

La fonction *loop_nest_iteration* 4.42 associe une itération dans l'espace distribué à l'itération correspondante dans l'espace séquentiel. Elle est définie en utilisant l'équation 4.17 décrivant le modèle de distribution des itérations d'un nid de boucles. On notera que cette définition garantit que les itérations d'une tranche d'itérations sont incluses dans les bornes de l'espace d'itération dans le domaine séquentiel.

$$\begin{aligned} \mathit{loop_nest_iteration}: \quad & \mathcal{P}_{ln} \times \mathcal{C}_{ln} \times \mathcal{L}_{ln} \rightarrow ln \\ & (\vec{\tilde{p}}_{ln}, \vec{c}_{ln}, \vec{l}_{ln}) \mapsto \vec{i} = L_{ln}\vec{1} + B_{ln}P_{ln}\vec{c}_{ln} + B_{ln}\vec{\tilde{p}}_{ln} + \vec{l}_{ln} \end{aligned} \tag{4.42}$$

La fonction *iteration_slice* 4.43 calcule une tranche d'itérations, qui est un ensemble d'itérations correspondant à un cycle donné sur un processus donné. Le symbole $\wp$ représente l'ensemble des parties d'un ensemble.

$$\begin{aligned} \mathit{iteration_slice}: \quad & \mathcal{P}_{ln} \times \mathcal{C}_{ln} \rightarrow \wp(ln) \\ & (\vec{\tilde{p}}_{ln}, \vec{c}_{ln}) \mapsto \{\vec{i} | \mathit{loop_nest_iteration}(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}, \vec{0}) \leq \vec{i} \wedge \\ & \vec{i} < \mathit{loop_nest_iteration}(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}, B_{ln}\vec{1}) \wedge \\ & L_{ln}\vec{1} \leq \vec{i} \wedge \vec{i} < U_{ln}\vec{1}\} \end{aligned} \tag{4.43}$$

Remarque. Un des intérêts de la distribution est de permettre l'exécution de programmes parallèles sur des données de très grande taille. Il se peut, dans certains cas, que la mémoire séquentielle d'un seul nœud ne suffise pas pour allouer toutes les données du programme séquentiel. Le raisonnement domaine séquentiel/domaine distribué reste cependant valide indépendamment de la capacité mémoire d'un seul nœud. Le but de cette modélisation est de montrer la correction du code parallèle généré, c'est-à-dire qu'il calcule le même résultat que le programme séquentiel.

4.3.3 Unicité locale d'un élément de tableau dans le domaine distribué

Chaque élément de tableau x dans le domaine séquentiel a au plus un représentant $(\vec{\tilde{p}}_x, \vec{c}_x, \vec{l}_x)$ sur un processus $\vec{\tilde{p}}_x$ dans le domaine distribué. Cette propriété est assurée par la contrainte imposée sur les halos dans la distribution des données (voir l'équation 4.35 page 66). On définit la fonction *element_copy* qui calcule un tel élément lorsqu'il est défini.

$$\begin{aligned} \mathit{element_defined}: \quad & \mathcal{C}_x \times \mathcal{L}_x \rightarrow \{true, false\} \\ & (\vec{c}_x, \vec{l}_x) \rightarrow \begin{cases} true \text{ si } \mathit{cycle_defined}(\vec{c}_x) \wedge \mathit{local_offset_defined}(\vec{l}_x) \\ false \text{ sinon} \end{cases} \end{aligned} \tag{4.44}$$

$$
\begin{aligned}
element_copy : \quad & x \times \mathcal{P}_x \rightarrow \mathcal{C}_x \times \mathcal{L}_x \cup \{element_undefined\} \\
& (\vec{a}, \vec{\tilde{p}}_x) \rightarrow \begin{cases} (\vec{c}_x = array_cycle(\vec{a}, \vec{\tilde{p}}_x), \vec{l}_x = local_offset(\vec{a}, \vec{\tilde{p}}_x, \vec{c}_x)) \text{ si } element_defined(\vec{c}_x, \vec{l}_x) \\ element_undefined \text{ sinon} \end{cases}
\end{aligned}
\tag{4.45}
$$

La fonction *element_copy* permet de retrouver le représentant dans le domaine distribué d'un élément dans le domaine séquentiel en fixant un identifiant de processus étendu. Comme tous les éléments ne sont pas répliqués sur tous les processus, il se peut qu'un élément donné du domaine séquentiel n'ait pas de représentant dans le domaine distribué pour un processus donné (point indéfini).

La figure 4.10 résume les fonctions permettant de passer entre les domaines séquentiel et distribué. La fonction *iteration_slice* calcule une tranche d'itération correspondant à un cycle donné pour un processus donné. Les itérations de cet ensemble, via les références ref_x^m aux éléments d'un tableaux x, permettent de retrouver les éléments accédés pour le tableau x dans l'espace séquentiel. La fonction *element_copy* permet enfin de retrouver les représentants de ces éléments dans l'espace distribué.

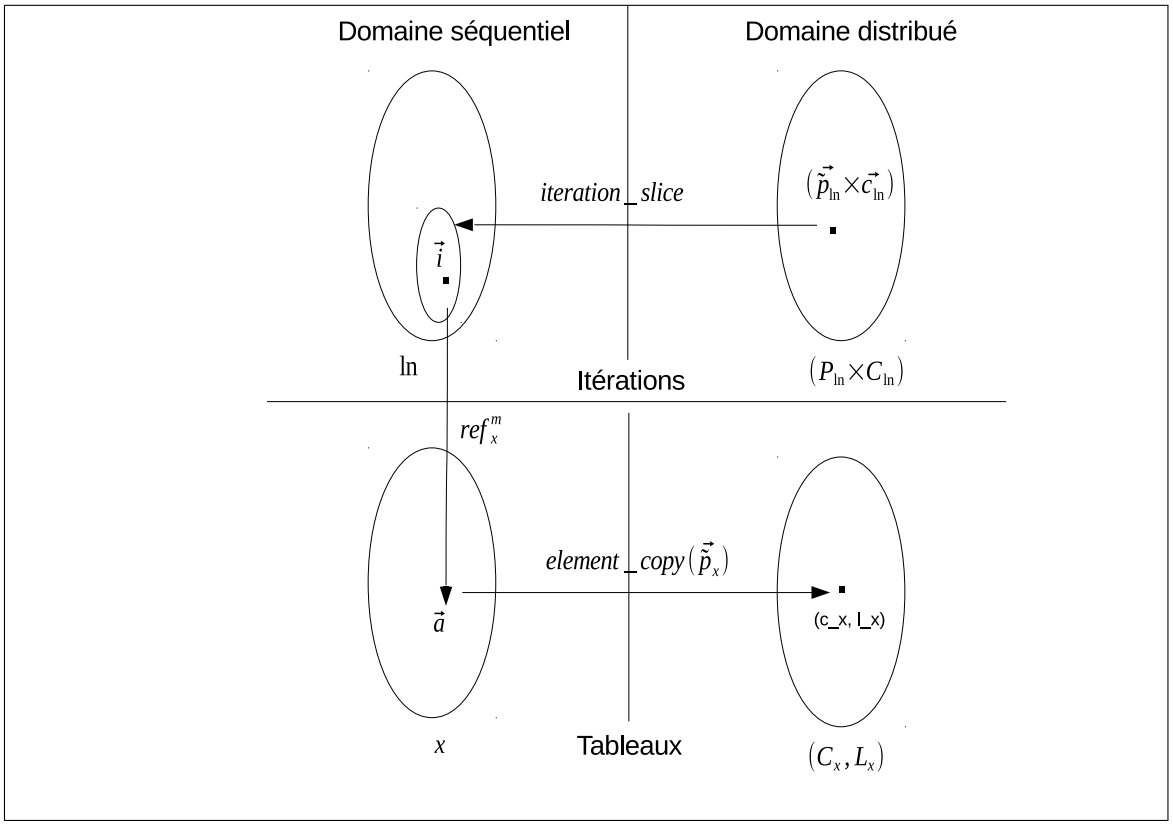


FIGURE 4.10 – Changement de domaine

4.3.4 Bonne distribution d'un tableau par rapport à un nid de boucles

Un tableau x est bien distribué par rapport à un nid de boucles ln si quelle que soit la référence ref_x^m à ce tableau, accédé pour toute itération $\vec{i} \in ln$ par un processus $\vec{\tilde{p}}_{ln}$, alors soit l'ensemble $Procs_x = \{\vec{\tilde{p}}_x | \vec{\tilde{p}}_x = extend_id(p, v), p = id_in_grid^-1(first(extend_id^{-1}(\vec{\tilde{p}}_{ln}))), v \in \mathcal{V}_x\}$:

1. Tous les processus $\vec{\tilde{p}}_x \in Proc_x$ vérifient la propriété d'unicité locale pour l'élément $ref_x(\vec{i})$,
2. (a) Si aucune dimension du nid de boucles ln n'a un type d'exécution *owner*, alors il existe au moins un processus $\vec{\tilde{p}}_x \in Procs_x$ tel que l'élément $ref_x^m(\vec{i})$ possède un représentant local sur ce processus,

   (b) Sinon, soit la tranche d'itérations $is = iteration_slice(\vec{\tilde{p}}_{ln}, iteration_cycle(\vec{i}))$, quelles que soient les itérations $\vec{i}_1, \vec{i}_2 \in is, \vec{i}_1 \neq \vec{i}_2$, alors soit il existe au moins un processus $\vec{\tilde{p}}_x \in Procs_x$ tel que les éléments $ref_x(\vec{i}_1)$ et $ref_x(\vec{i}_2)$ possèdent chacun un représentant local unique sur ce processus, soit $ref_x(\vec{i}_1)$ et $ref_x(\vec{i}_2)$ n'ont aucun représentant sur aucun de ces $\vec{\tilde{p}}_x$

Exemples. Les figures 4.11 et 4.12 montrent des exemples de codes avec un tableau mono-dimensionnel distribué par blocs sur deux processus.

Dans le premier cas, figure 4.11, la propriété de bonne distribution est vérifiée sur chaque processus pour les deux nids de boucles. Pour la première boucle, une tranche de huit itérations est affectée à chaque processus. On remarquera que la dernière itération (i = 15) n'est pas exécutée sur le processus $\vec{\tilde{p}} = [1]$ car elle dépasse la borne supérieure de la boucle. Pour chaque itération, l'élément accédé possède un représentant local unique sur le processus exécutant l'itération. La deuxième boucle a un type d'exécution *owner* : les itérations 0 et 1 sont affectées à chaque processus, mais seul le processus $\vec{\tilde{p}} = [0]$ les exécute car il possède un représentant local de chacun des éléments accédés.

Dans le second cas, figure 4.12, la propriété de bonne distribution n'est pas vérifiée. Pour la première boucle, la dernière itération affectée à $\vec{\tilde{p}} = [0]$ accède à un élément qui ne possède pas de représentant sur ce processus. Pour la deuxième boucle, les deux itérations 7 et 8 sont affectées à chaque processus. Sur $\vec{\tilde{p}} = [1]$, les représentants locaux des éléments accédés existent, mais sur $\vec{\tilde{p}} = [0]$, l'élément accédé par l'itération 7 possède un représentant local, ce qui n'est pas le cas de l'élément accédé par l'itération 8 appartenant à la même tranche d'itérations.

Remarque. La propriété de bonne distribution, qui implique la propriété d'unicité locale, est vérifiée dynamiquement par le code généré (voir la section Compilation).

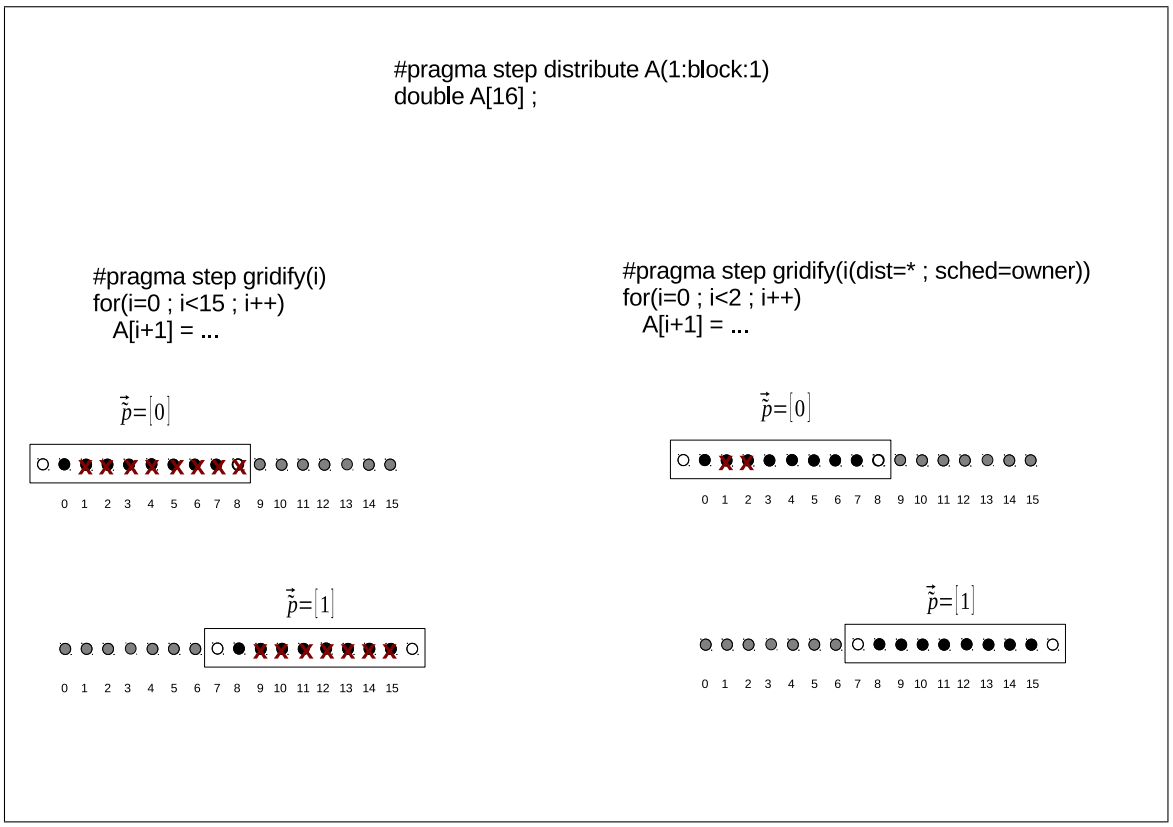


FIGURE 4.11 – Exemple de code vérifiant la propriété de bonne distribution.

4.3.5 Conservation du parallélisme dans le domaine distribué

Propriété d'un nid de boucles parallèle ln dans l'espace séquentiel. Quelles que soient les références ref_x^m, ref_x^n à un tableau x dans ln, si au moins une de ces références est une écriture, alors : $ref_x^m(\vec{i}) \neq ref_x^n(\vec{i'})$ pour tous $\vec{i}, \vec{i'} \in ln$.

Théorème. 5. *Pour un nid de boucles parallèle ln vérifiant la propriété de bonne distribution, quelles que soient les itérations $\vec{i}, \vec{i'} \in ln, \vec{i} \neq \vec{i'}$, quelles que soient les références ref_x^m, ref_x^n à un tableau x, si au moins une de ces références est une écriture, alors si $\vec{\tilde{p}}_x$ (resp. $\vec{\tilde{p}'}_x$) est un processus possédant un représentant local de $ref_x^m(\vec{i})$ (resp. $ref_x^n(\vec{i'})$) alors on a :*

$$(\vec{\tilde{p}}_x, \vec{c}_x, \vec{l}_x) \neq (\vec{\tilde{p}'}_x, \vec{c'}_x, \vec{l'}_x)$$

avec :

$$\vec{c}_x = array_cycle(ref_x^m(\vec{i}), \vec{p}_x),$$
$$\vec{l}_x = local_offset(ref_x^m(\vec{i}), \vec{p}_x, \vec{c}_x)$$
$$\vec{c'}_x = array_cycle(ref_x^m(\vec{i}), \vec{p}_x),$$
$$\vec{l'}_x = local_offset(ref_x^n(\vec{i'}), \vec{p'}_x, \vec{c'}_x)$$

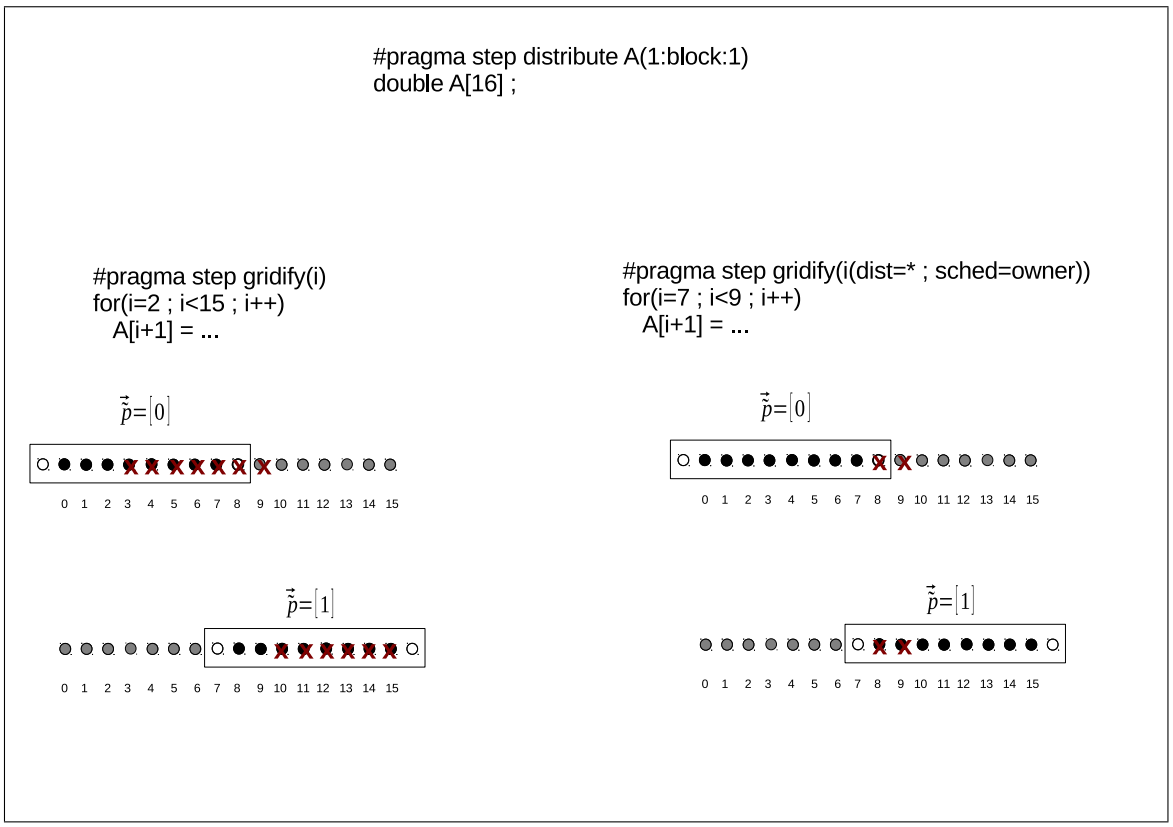


FIGURE 4.12 – Exemple de code ne vérifiant pas la propriété de bonne distribution.

Ce théorème exprime le fait que les accès indépendants dans le domaine séquentiel sont conservés dans le domaine distribué. Ceci implique la possibilité de générer un code parallèle dans le domaine distribué pour un nid de boucles parallèle dans le code en entrée.

Démonstration. On suppose que

$$(\vec{\tilde{p}}_x, \vec{c}_x, \vec{l}_x) = (\vec{\tilde{p}'}_x, \vec{c'}_x, \vec{l'}_x) \tag{4.46}$$

L'hypothèse 4.46 implique que $\vec{l}_x = \vec{l'}_x$ et $\vec{\tilde{p}}_x = \vec{\tilde{p}'}_x$ et $\vec{l}_x = \vec{l'}_x$

$$\iff (\vec{l}_x = local_offset(ref_x^m(\vec{i}), \vec{p}_x, \vec{c}_x)) = (\vec{l'}_x = local_offset(ref_x^n(\vec{i'}), \vec{p}_x, \vec{c}_x))$$

$$\iff ref_x^m(\vec{i}) - P_x B_x \vec{c}_x - B_x \vec{\tilde{p}}_x + Hlow_x \vec{1} = ref_x^n(\vec{i'}) - P_x B_x \vec{c}_x - B_x \vec{\tilde{p}}_x + Hlow_x \vec{1}$$

$$\iff ref_x^m(\vec{i}) = ref_x^n(\vec{i'})$$

Ce résultat est en contradiction avec la propriété d'un nid de boucles parallèle pour lequel $ref_x^m(\vec{i}) \neq ref_x^n(\vec{i'})$. On en conclut que l'hypothèse 4.46 est fausse et que donc le parallélisme est conservé dans le domaine distribué. □

4.3.6 Préservation de l'ordre séquentiel dans le domaine distribué.

Lorsqu'au moins une dimension d'un nid de boucles est ordonnée, il faut exécuter les itérations dans le même ordre selon les dimensions ordonnées dans le domaine distribué que celui du domaine séquentiel.

Théorème. 6. *Pour un nid de boucles ordonné ln vérifiant la propriété de bonne distribution et dont l'ensemble des itérations est muni d'un ordre lexicographique* $<_{ln}$*, quelles que soient les itérations* $\vec{i}, \vec{i'} \in ln$*, si* $\vec{i} <_{ln} \vec{i'}$*, alors l'itération* $\vec{i}$ *peut être exécutée avant l'itération* $\vec{i'}$ *dans le domaine distribué.*

Ce théorème exprime le fait qu'il est possible de générer pour un nid de boucles ordonné une énumération des itérations dans le domaine distribué respectant l'ordre des itérations dans le domaine séquentiel.

Démonstration.
a) Si les itérations $\vec{i}$ et $\vec{i'}$ appartiennent à la même tranche d'itérations $iteration_slice(\vec{\tilde{p}}_{ln} = iteration_executor(\vec{i}), \vec{c} = iteration_cycle(\vec{i}))$, alors il suffit d'exécuter ces itérations en respectant leur ordre séquentiel sur le processus $\vec{\tilde{p}}_{ln}$.

b) Sinon, soit $is = iteration_slice(\vec{\tilde{p}}_{ln} = iteration_executor(\vec{i}), \vec{c} = iteration_cycle(\vec{i}))$ et $is' = iteration_slice(\vec{\tilde{p}'}_{ln} = iteration_executor(\vec{i'}), \vec{c'} = iteration_cycle(\vec{i'}))$. Soit $\vec{last}_i = max_{<_{ln}}(is)$ et $\vec{first}_{i'} = min_{<_{ln}}(is')$. Le modèle de distribution des itérations sans calculs redondants garantit que $is \cap is' = \emptyset$. On a donc forcément $\vec{last}_i <_{ln} \vec{first}_{i'}$.

Pour garantir une exécution ordonnée des itérations $\vec{i}$ et $\vec{i'}$ dans le domaine distribué, il suffit de synchroniser les itérations $\vec{last}_i$ et $\vec{first}_{i'}$: on bloque l'exécution de l'itération $\vec{first}_{i'}$ jusqu'à la fin de l'exécution de l'itération $\vec{last}_i$. On a ainsi un moyen de garantir le respect de l'ordre initial pour les itérations de deux tranches d'itérations différentes en synchronisant seulement leurs itérations extrémales. Ceci est réalisé dans le code généré via les communications entre les processus $\vec{\tilde{p}}_{ln}$ et $\vec{\tilde{p}'}_{ln}$ (voir la section *génération des communications* 5.4 du chapitre *compilation* 5).

□

4.3.7 Correction de la valeur d'un élément dans le domaine distribué

Pour raisonner sur l'espace distribué avant de voir les détails de la génération de code, nous donnons la propriété suivante du code généré.

Propriété. Le code généré pour un nid de boucles ln avec n références à un tableau x comporte n références aux éléments correspondant dans l'espace distribué apparaissant dans le même ordre syntaxique que dans le domaine séquentiel. Les expressions du programme ne comportant pas de référence à des tableaux distribués sont inchangées.

Point du programme. Pour un nid de boucle ln et pour un tableau x référencé dans ln, un point du programme est parfaitement défini par le couple $(m, \vec{i})$ où m désigne la $m^{ième}$ référence à x et $\vec{i}$ est l'itération qui accède à cette référence.

Mémoire séquentielle. La mémoire séquentielle est un environnement MS qui associe une valeur à chaque variable dans le domaine séquentiel à un point donné du programme.

Mémoire distribué. La mémoire distribuée est un environnement MD qui associe une valeur à chaque variable dans le domaine distribué à un point donné du programme.

Définition. Un élément $(\vec{p}_x, \vec{c}_x, \vec{l}_x)$ du domaine distribué accédé à la $m^{ième}$ référence par l'itération $\vec{i}$ a une valeur mémoire correcte si et seulement si :

$$MD((\vec{p}_x, \vec{c}_x, \vec{l}_x), m, \vec{i}) = MS(array_element(\vec{p}_x, \vec{c}_x, \vec{l}_x), m, \vec{i})$$

4.3.8 Correction des valeurs mémoire des éléments de tableau dans le domaine distribué pour une tranche d'itérations

Théorème. 7. *Si la propriété de bonne distribution est vérifiée pour un nid de boucles ln, alors quelle que soit l'itération $\vec{i} \in ln$ exécutée dans le domaine distribuée, quelle que soit la référence $ref_x^m \in REF(ln)$ dans l'espace séquentiel, alors pour un $\vec{\tilde{p}}_x$ possédant un représentant local de l'élément $ref_x^m(min(is))$ et de l'élément $ref_x^m(max(is))$ avec $is = iteration_slice(iteration_executor(\vec{i}), iteration_cycle(\vec{i}))$, on a :*

$$MD((\vec{p}_x, \vec{c}_x, \vec{l}_x), m, \vec{i}) = MS(ref_x^m(\vec{i}), m, \vec{i})$$

Démonstration.
Hypothèse H1 : on admet que les valeurs mémoire dans le domaine distribué sont correctes à l'entrée d'une tranche d'itérations.

Soit $ref_x^0, ..., ref_x^k, ..., ref_x^{n-1}$ les n références au tableau x dans le nid de boucles ln dans leurs ordre d'apparition syntaxique dans le code en entrée. Il s'agit d'un ordre total sur lequel on peut appliquer une preuve par induction.

Cas de base, k = 0.

a) La référence ref_x^0 est une lecture, alors on a $MD((\vec{\tilde{p}}_x, \vec{c}_x, \vec{l}_x), 0, \vec{i}) = MS(ref_x^0(\vec{i}), 0, \vec{i})$ par l'hypothèse $H1$.

b) La référence ref_x^0 est une écriture : si l'expression qui calcule la valeur de l'élément $(\vec{\tilde{p}}_x, \vec{c}_x, \vec{l}_x)$ contient des références à des tableaux distribués, alors leurs valeurs dans le domaine distribué sont correctes comme montré en (a). La valeur calculée est donc correcte.

Cas inductif. Pour $m < n$, on suppose que la propriété est vraie pour tout $m' < m$ et on montre qu'elle aussi vraie pour m.

a) La référence ref_x^m est une lecture : si la valeur lue a été écrite par un référence précédente, alors sa valeur est correcte par hypothèse d'induction. Sinon, cette valeur est inchangée depuis l'entrée dans la tranche d'itérations et sa valeur est donc correcte par l'hypothèse $H1$.

b) La référence ref_x^m est une écriture : si l'expression qui calcule la valeur de l'élément $(\vec{p}_x, \vec{c}_x, \vec{l}_x)$ contient des références à des tableaux distribués, alors ce sont des références en lecture et leurs valeurs sont correctes comme montré en (a). La valeur calculée est donc correcte.

On en conclut que les éléments accédés dans le domaine distribué ont des valeurs correctes. □

Remarques sur l'hypothèse $H1$.

1. L'hypothèse $H1$ n'est pas nécessaire pour les nids de boucles définissant les valeurs des éléments de tableau distribués sans faire intervenir d'autres tableaux distribués. Ce cas peut notamment se présenter lors de la phase d'initialisation des tableaux.
2. Nous garantirons l'hypothèse $H1$ dans la partie compilation en faisant intervenir les communications qui rétablissent la correction de la mémoire distribuée (voir la section 5.4.5).

4.3.9 Conclusion

Le modèle de distribution garantit la correction du code généré en assurant que :

1. tout élément accédé possède au plus un représentant unique sur un processus,
2. les contraintes d'ordonnancement du code en entrée sont respectées dans le domaine distribué et le modèle de distribution conserve le parallélisme contenu dans le code en entrée,
3. les valeurs lues et écrites dans le domaine distribué sont correctes par rapport au domaine séquentiel grâce à l'ordonnancement correct des itérations assuré par les communications.

4.4 Conclusion

Dans ce chapitre, nous avons modélisé la distribution des nids de boucles sans calcul redondant et la distribution des tableaux avec *halos* en utilisant un formalisme d'algèbre linéaire. Les objets ainsi modélisés appartiennent à deux domaines distincts : le domaine séquentiel, qui représente les objets du code original, et le domaine distribué, qui représente les objets du code parallèle généré. Le schéma de compilation, présenté au chapitre suivant, consiste à passer du domaine séquentiel au domaine distribué, en s'appuyant sur les propriétés du modèle de distribution présentées dans ce chapitre.

Chapitre 5

Compilation de *dSTEP*

Dans ce chapitre, nous présentons la compilation des programmes *dSTEP* basé sur les propriétés du modèle combiné de distribution définies au chapitre précédent (chapitre 3). La figure 5.1 montre une vue générale de notre compilateur, qui prend en entrée des programmes annotés avec des directives *dstep distribute* et *dstep gridify* et qui génère un programme parallèle hybride. Le programme parallèle généré fait appel aux fonctions implémentées dans le support d'exécution de *dSTEP* pour réaliser notamment les communications. Au moment de la rédaction de cette thèse, nous traitons des codes en entrée écrits dans le langage C, mais le schéma de compilation présenté peut aussi bien être implémenté pour prendre en entrée des programmes écrits en Fortran.

5.1 PIPS

Notre compilateur se base sur la plateforme de compilation PIPS [3, 55] qui offre les fonctionnalités d'analyse syntaxique, de construction et de manipulation de l'arbre de syntaxe abstraite, des analyses sémantiques puissantes basées sur les données du programme ainsi que des *pretty-printers* qui permettent de générer les programmes en sortie.

Dans notre schéma de compilation, nous utilisons les régions de tableaux définies par Creusillet *et al.* [30] et implémentées dans PIPS. Une région de tableau est un ensemble d'éléments du tableau défini par un polyèdre convexe. Les régions de tableaux, en plus de leur description géométrique, comportent une information de type et de précision. Le type d'une région reflète l'action des instructions du programme sur ses éléments. Ainsi, une région de type *READ* (resp. *WRITE*) représente les éléments lus (resp. écrits) par un ensemble d'instructions. De plus, l'information de flot de données aux différents points du programme est également capturée. Pour un ensemble d'instructions du programme, une région *IN* est une région de type *READ* dont les éléments ne sont pas redéfinis par un ensemble d'instructions. Une région *OUT* pour un ensemble d'instructions est une région de type *WRITE* dont les éléments seront utilisés à un point succédant à cet ensemble d'instruction dans le graphe de contrôle du programme. Enfin, l'information de précision indique si la des-

cription de la région est exacte ($EXACT$) ou s'il s'agit d'une approximation (MAY). Les analyses de régions de tableaux dans PIPS sont *interprocédurales* : elles sont appliquées pour calculer les régions de tableaux pour tout un programme contenant plusieurs appels de fonctions avec des tableaux en arguments.

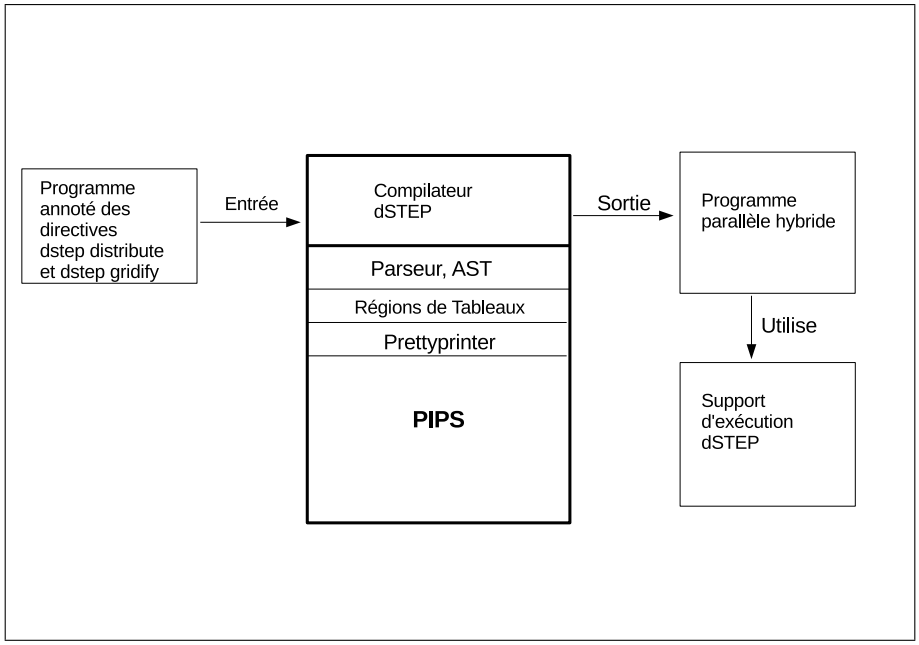


FIGURE 5.1 – Vue globale du compilateur dSTEP.

Nous présentons d'abord l'allocation des tableaux distribués sur chaque processus. Nous montrons ensuite la compilation des nids de boucles distribués qui implique à la fois la génération de références aux éléments de tableaux dans le domaine distribué correspondant aux références dans le domaine séquentiel ainsi que la génération des communications nécessaires pour la mise à jour des données répliquées. Nous présentons enfin une série d'optimisations pour améliorer les performances du code généré.

5.2 Allocation mémoire

L'algorithme 1 montre la génération de l'allocation mémoire pour un tableau x du code en entrée annoté d'une directive *dstep distribute*. On génère une variable de même nom augmentées de deux dimensions : une dimension pour les cycles et une dimension pour les processus virtuels.

Vérification de la contrainte sur les halos d'un tableau. Lors de l'allocation mémoire pour un tableau distribué, la validité des halos est vérifiée (cf. contrainte sur les halos 4.35). Si un halo est trop grand, ce qui peut arriver dans le cas d'une distribution cyclique, alors un message est affiché pour inviter le programmeur à choisir une autre valeur dans l'intervalle des halos valides.

Algorithm 1 *generate_memory_allocation*

Entrée: $D, B, Hlow, Hup, \mathcal{C}, \mathcal{V}$
Sortie: remplacement de la déclaration de x par une déclaration distribuée

1: $generate_memory_allocation(X[D_{0,0}]...[D_{d-1,d-1}]) \equiv$
2: **for** each dimension k of array x **do**
3: **if** dimension k has a total halo **then** $L_{k,k} = D_{k,k}$
4: **else**
5: $L_{k,k} = Hlow_{k,k} + B_{k,k} + Hup_{k,k}$
6: **end if**
7: **end for**
8: X[$|\mathcal{C}|$][$|\mathcal{V}|$][$L_{0,0}$]...[$L_{d-1,d-1}$]

5.2.1 Taille mémoire allouée pour un tableau distribué

L'expression 5.1 calcule la taille mémoire allouée sur chaque processus pour un tableau distribué de taille d en fonction de la taille du tableau d'origine. On pose $H = Hlow + Hup$. La véritable empreinte mémoire est obtenue en multipliant cette valeur par la taille de l'élément de base du tableau. Le premier terme de l'équation est la valeur exacte pour les distributions sans cycle et une sous-approximation à quelques éléments près pour les distributions cycliques où des tailles de blocs sont imposées par l'utilisateur. On admet tout de même qu'il représente la mémoire allouée sur chaque processus avant de considérer les halos.

$$\begin{aligned} &det\lceil(DP^{-1})\rceil + |\mathcal{C}|\,|\mathcal{V}|\,(H_{0,0}(H_{1,1}+B_{1,1})(H_{2,2}+B_{2,2})...(H_{d-1,d-1}+B_{d-1,d-1})+ \\ &H_{1,1}B_{0,0}(H_{2,2}+B_{2,2}) + ...(H_{d-1,d-1}+B_{d-1,d-1}) \\ &...+ \\ &H_{d-1,d-1}B_{0,0}...B_{d-2,d-2} \end{aligned} \tag{5.1}$$

Remarques.

- En absence de halo sur toutes les dimensions du tableau, la taille allouée sur chaque processus est approximativement la taille du tableau d'origine divisée par le nombre de processus.
- Avec un halo, les cycles ou une dimension diagonalisée augmentent la taille allouée sur chaque processus par rapport à une distribution sans cycle et sans multi-partitioning.

Exemple. Soit `A` un tableau bi-dimensionnel d'entiers de taille $28 \times 38 = 1064$ distribué sur ses deux dimensions. Si la distribution est exécutée sur neuf processus, les éléments de `A` seront distribués sur une grille virtuelle de la forme 3×3.

Dans le Listing 5.1, le tableau `A` est distribué sans cycle. En appliquant l'équation 5.1, on obtient une taille mémoire pour chaque processus égale à :

$(\lceil 28/3\rceil \times \lceil 38/3\rceil) + (1 \times 1)((1+1)(2+13+2) + (2+2)(10)) = 204.$

Dans le Listing 5.2, le tableau A est distribué avec une distribution cyclique de taille cinq sur sa première dimension et par blocs sur la seconde. Le nombre de cycles

```
1 #pragma dstep distribute A(1:block:1, 2:block:2)
2 int A[28][38];
```

Listing 5.1 – Distribution par blocs

dans ce cas est égal deux. En appliquant l'équation 5.1, on obtient une taille mémoire pour chaque processus égale à :

$$(\lceil 28/3 \rceil \times \lceil 38/3 \rceil) + (2 \times 1)((1+1)(2+13+2) + (2+2)(5)) = 238.$$

```
1 #pragma dstep distribute A(1:cyclic(5):1, 2:block:2)
2 int A[28][38];
```

Listing 5.2 – Distribution cyclique

On multiplie alors ces valeurs par `sizeof(int)` pour obtenir la véritable taille mémoire occupée. On constate que la distribution cyclique, dans ce cas, occupe plus de mémoire que la distribution sans cycles ($238 > 204$).

5.3 Compilation d'un nid de boucles

5.3.1 Représentation syntaxique d'un nid de boucles

Un nid de boucles portant une directive *dstep gridify* est modélisé par le code de l'algorithme 2. Le corps du nid de boucles est abstrait par la représentation syntaxique *body* qui montre les objets d'intérêt : les références à des éléments de tableaux distribués. Ces références peuvent apparaître dans des expressions quelconques ou comme arguments d'appels de fonctions. Si une fonction est appelée dans *body* avec des tableaux distribués en arguments, alors le schéma de compilation est appliqué à la fonction appelée. On désigne la $m^{\text{ème}}$ référence à un tableau x pour une itération $\vec{i}$ par $ref_x^m(\vec{i})$. Ainsi, x et m peuvent être considérées comme des variables pour parcourir l'ensemble des tableaux et des références à ces tableaux dans un nid de boucles.

Les itérations du nid de boucles ln sont bornées par les matrices diagonales L_{ln} et U_{ln}. L'incrément, pour chaque niveau du nid de boucles peut être positif ou négatif (parcours croissant ou décroissant). Les incréments de tous les niveaux du nid de boucles sont représentés par le vecteur $\vec{inc}$. Les points de l'ensemble des itérations sont énumérés par un vecteur itération $\vec{i}$, avec éventuellement des sauts si la valeur absolue d'au moins une composante du vecteur incrément n'est pas égale à 1.

Algorithm 2 Représentation générique d'un nid de boucles annoté d'une directive *dstep gridify*

#pragma dstep gridify(...) [private(var_list)] [firstprivate(var_list)] [reduction(op :var_list)]
for $\vec{i} \in \{\vec{i} | \vec{i} = L_{ln}\vec{1} + n|\vec{inc}|, n \in \mathbb{N} \wedge L_{ln}\vec{1} \leq \vec{i} < U_{ln}\vec{1}\}$ **do**
  $body(..., X[ref_x^m(\vec{i})], ...)$
end for

Exemple Dans le code du listing 5.3, l'incrément de la dimension j est négatif et celui de la dimension k vaut deux. Ce code est représenté par la forme simplifiée de l'algorithme 3, avec les références aux tableaux mises en évidence.

```
1 #pragma step gridify(i, j, k)
2   for (i = 0; i < M/2; i++)
3     for (j = N-1; j >= 0; j--)
4       for (k = 1; k < K; k = k+2)
5       {
6         A[i][j][k] = A[2*i][j][k-1] + B[i-1][j][k];
7         f(..., A[i][j][k-1], ..., B, ...);
8       }
```

Listing 5.3 – Exemple d'un nid de boucle distribué

Algorithm 3 Représentation du nid de boucles distribué

#pragma step gridify(...)

for $\vec{i} \in \{\vec{i} | \vec{i} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + n \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}, n \in \mathbb{N}, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \leq \vec{i} < \begin{bmatrix} M/2 \\ N-1 \\ K \end{bmatrix}\}$ **do**

$body(A[ref_A^0(\begin{bmatrix} i \\ j \\ k \end{bmatrix})], A[ref_A^1(\begin{bmatrix} i \\ j \\ k \end{bmatrix})], B[ref_B^0(\begin{bmatrix} i \\ j \\ k \end{bmatrix})], A[ref_A^2(\begin{bmatrix} i \\ j \\ k \end{bmatrix})], B)$

end for

Remarque. Dans les schémas de compilation présentés par la suite, on mélangera le code du compilateur et le code généré par ce dernier. Le code du compilateur sera distingué par une indication *compiler* en commentaire.

5.3.2 Schéma général de compilation d'un nid de boucles

Au cœur du schéma de compilation se trouve la notion de *tranche d'itérations*. Une tranche d'itérations est un ensemble d'itérations pour un processus et un cycle donnés. Il faut énumérer correctement toutes les tranches d'itérations, et pour chaque tranche d'itérations :

1. traduire les références du domaine séquentiel en des références aux tableaux distribués (changement de domaine),
2. générer les communications pour mettre à jour les éléments répliqués si nécessaire.

On distingue deux cas : les nids de boucles parallèles, dont aucune dimension n'est ordonnée, et les nids de boucles ordonnés, qui contiennent au moins une dimension ordonnée.

5.3.3 La fonction *belongs_to*

La fonction *belongs_to* permet de retrouver, sur un processus $\vec{p}_x$, un couple $(\vec{c}_x, v_x)$ pour lequel deux éléments $\vec{a}_1, \vec{a}_2 \in x$ possèdent des représentant locaux. Le but de cette fonction est double :

1. vérifier la propriété de bonne distribution pour le tableau x,
2. le couple $(\vec{c}_x, v_x)$ calculé constitue l'information permettant d'effectuer le changement de domaine pour les références au tableau x.

Algorithm 4 *belongs_to*

Entrée: $\vec{p}_x, \vec{a}_1, \vec{a}_2, \mathcal{V}_x$
Sortie: un couple $(\vec{c}_x, v_x)$ où les éléments $\vec{a}_1, \vec{a}_2$ possèdent des représentants local ou un élément indéfini

```
1: for v ∈ V_x do
2:     p̃_x = extend_id(p_x, v)
3:     (c_1, l_1) = element_copy(a_1, p̃_x)
4:     if element_defined((c_1, l_1)) then
5:         (c_2, l_2) = element_copy(a_2, p̃_x)
6:         if element_defined((c_2, l_2)) ∧ (c_2 == c_1) then
7:             return (c_1, v)
8:         end if
9:     end if
10: end for
11: return (cycle_undefined, v_undefined)
```

5.3.4 Compilation d'un nid de boucles parallèle

Il n'existe pas de contrainte sur l'ordre d'exécution des itérations dans l'espace séquentiel pour un nid de boucles parallèles. Nous avons montré au chapitre 4 présentant le modèle de distribution que ce parallélisme est conservé dans l'espace distribué. L'algorithme 5 montre la compilation d'un nid de boucles parallèle. Chaque processus énumère les tranches d'itérations qu'il doit exécuter (lignes 1-5). Pour chaque tranche d'itérations, on génère le code pour le changement de domaine (ligne 6) et pour les communications *send* asynchrones (ligne 7). Ce schéma de compilation permet de générer un code dans lequel les communications *send* de plusieurs tranches d'itération sont recouvertes par le calcul des autres tranches d'itérations du même nid de boucle. Aux lignes 10-12, on montre la génération de code pour d'éventuelles réductions parallèles du nid de boucles. On génère ensuite le code pour les communications *recv* asynchrones (ligne 13). Enfin, on génère la complétion des communications *send* et *recv* asynchrones générées précédemment.

5.3.5 Compilation d'un nid de boucles ordonné

Dans le code généré pour un nid de boucles ordonné, l'ordre séquentiel des itérations selon les dimensions *ordered* doit être respecté. Cette contrainte impacte

Algorithm 5 *compile_parallel*

Entrée: $ln_stmt, body, \mathcal{V}_{ln}, \mathcal{C}_{ln}$
Sortie: le code ln_stmt compilé dans le domaine distribué

1: $p = rank()$
2: $\vec{p}_{ln} = id_in_grid_{ln}(p)$
3: **for** $v_{ln} \in \mathcal{V}_{ln}$ **do**
4: $\quad \vec{\tilde{p}}_{ln} = extend_id(\vec{p}_{ln}, v_{ln})$
5: $\quad$ **for** $\vec{c}_{ln} \in \mathcal{C}_{ln}$ **do**
6: $\quad\quad compile_iteration_slice(ln_stmt, body, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln})$ ▷ compiler
7: $\quad\quad generate_sends(ln_stmt, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln})$ ▷ compiler
8: $\quad$ **end for**
9: **end for**
10: **if** reduction **then**
11: $\quad generate_parallel_reductions(op, var_list)$ ▷ compiler
12: **end if**
13: $generate_recvs(ln_stmt, \emptyset)$ ▷ compiler
14: $generate_completes(ln_stmt)$ ▷ compiler

l'énumération des tranches d'itérations et les points auxquels sont insérées les communications. Les énumérations $scan_c$ et $scan_v$ définies dans les sections suivantes sont utilisées pour parcourir respectivement les cycles et les blocs d'un nid de boucles ordonné. Avant l'exécution d'une tranche d'itérations, on s'assure que les données qu'elle utilise et qui sont produites par d'autres tranches d'itérations la *précédant* sont reçues. L'opérateur de comparaison $<_{ln}$ est propre à chaque nid de boucles et est construit selon les dimensions ordonnées et du sens de leur incrément. Concrètement, cet opérateur est codé par un vecteur d'entiers de la même dimension que ln. Toutes les composantes correspondant à une dimension non ordonnée ont une valeur nulle. Une dimension *ordered* a la valeur 1 pour un incrément positif et -1 pour un incrément négatif.

Énumération des cycles d'un nid de boucles ordonné ln**.** Pour un nid de boucles ln de dimension d, de vecteur incrément $\vec{inc}$ et de nombre de cycles C_{ln}, on définit les vecteurs $\vec{first_c}$ et $\vec{inc_c}$ de dimension d :

$$\vec{first_c}_i = \begin{cases} 0 & \text{si } \vec{inc}_i > 0 \\ C_{lnd-1,d-1} & \text{sinon} \end{cases} \tag{5.2}$$

$$\vec{inc_c}_i = \begin{cases} 1 & \text{si } \vec{inc}_i > 0 \\ -1 & \text{sinon} \end{cases} \tag{5.3}$$

Une énumération $scan_c$ compatible avec l'ordre du nid de boucles ordonné ln est définie par la suite $\vec{c}_n$ (équation 5.4) à $det(C_{ln})$ termes.

$$\begin{cases} \vec{c}_{n+1} = \vec{c}_n + \vec{inc_c} \\ \vec{c}_0 = \vec{first}_c \end{cases} \tag{5.4}$$

Énumération des processus virtuels d'un nid de boucles ordonné ln**.** Si aucune dimension d'un nid de boucles n'est diagonalisée, alors l'énumération des processus virtuels $vprocs = 1$ est triviale. Pour un nid de boucles ln de dimension d, de vecteur incrément $\vec{inc}$ avec une dimension diagonalisée k, soit j la première dimension ordonné du nid de boucles ln. On définit inc_v par :

$$inc_v = \begin{cases} 1 & \text{si } \vec{inc}_j > 0 \\ -1 & \text{sinon} \end{cases} \tag{5.5}$$

On définit le processus $\vec{first_p}_{ln}$, qui est le premier processus rencontré lors du parcours de la première dimension ordonnée de ln par :

$$\vec{first_p}_{lni} = \begin{cases} 0 & \text{si } \vec{inc}_i > 0 \\ P_{ln}\vec{1} - \vec{1} & \text{sinon} \end{cases} \tag{5.6}$$

L'énumération $scan_v$ définie par la suite v_n à $vprocs$ termes (équation 5.7) est compatible avec l'ordre du nid de boucles ordonné ln.

$$\begin{cases} v_{n+1} = (v_n + P_{k,k} + inc_v)\%P_{k,k} \\ v_0 = second(extend_id^{-1}(\vec{first_p}_{ln})) \end{cases} \tag{5.7}$$

Exemple. Voici un exemple de nid de boucles avec des dimensions *ordered* (listing 5.4). Le vecteur codant l'opérateur $<_{ln}$ correspondant est $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$. Dans cet ordre, l'itération $\begin{bmatrix} 10 \\ 20 \\ 30 \end{bmatrix}$ est inférieure aux itérations $\begin{bmatrix} 5 \\ 21 \\ 30 \end{bmatrix}$ et $\begin{bmatrix} 10 \\ 19 \\ 31 \end{bmatrix}$.

```
1 #pragma step gridify(i(dist=block; sched=ordered), j, k(dist=block;
    sched=ordered))
2   for (i = 99; i >= 0; i -= 2)
3     for (j = 0; j < 100; j++)
4       for (k = 0; k < 100;  k++)
5       {
6         ...
7       }
```

Listing 5.4 – Exemple d'un nid de boucles ordonné

5.3.6 L'ensemble *Before* pour une tranche d'itérations

On définit l'ensemble *Before* pour une tranche d'itérations d'un nid de boucle muni de l'ordre lexicographique $<_{ln}$ par l'équation 5.8.

$$\begin{aligned} Before(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}) = \; & \{(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln}) \in \mathcal{P}_{ln} \times \mathcal{C}_{ln} | \\ & (\vec{last} = max_{<_{ln}}(iteration_slice(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln}))) <_{ln} (\vec{first} = min_{<_{ln}}(iteration_slice(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}))) \\ & \wedge |\vec{last} - \vec{first}| = |\vec{inc}|\} \end{aligned} \tag{5.8}$$

L'algorithme 6 montre la compilation d'un nid de boucles ordonné. Pour chaque tranche d'itérations, les *contributions* à la définition des éléments d'un bloc de tableau par les tranches d'itérations *précédentes*, *i.e.* appartenant à l'ensemble *Before* (voir équation 5.8), sont d'abord considérées (ligne 10). On insère des appels de réception de données bloquants afin de garantir que le calcul n'avance pas tant que les dépendances ne sont pas satisfaites depuis les processus distants exécutant les tranches d'itérations précédentes. L'ensemble *Received* mémorise les tranches d'itérations dont les contributions ont déjà été considérées. Ces tranches d'itérations ne seront pas prises en compte dans la génération des *recv* restants, asynchrones, à la fin de l'exécution du nid de boucles (ligne 29). La compilation d'une tranche d'itération (ligne 22), la génération des *send* asynchrones (ligne 23) ainsi que la génération de code pour d'éventuelles réductions parallèles (lignes 26-28) sont identiques à un nid de boucle parallèle.

5.3.7 Compilation d'une tranche d'itérations

La compilation d'une tranche d'itérations suit le même schéma dans tous les cas (algorithme 7). L'ensemble des itérations de la tranche d'itérations est calculé et étendu selon chaque dimension *owner* à toutes les itérations de cette dimension (lignes 1-4). Les références aux tableaux distribués sont traduites du domaine séquentiel au domaine distribué (lignes 29-34). La fonction *belongs_to* calcule le cycle et le processus virtuel correspondant à deux éléments de tableau du domaine séquentiel. Si une telle paire n'est pas définie et qu'il n'existe pas de dimension *owner* dans le nid de boucles, alors la distribution n'est pas correcte et le programme est arrêté. En présence d'au moins une dimension *owner* du nid de boucles, le code généré teste si tous les éléments accédés sont alloués sur le processus courant (paires $(\vec{c}_x, v_x)$ définies) ou si aucun de ces éléments n'est alloué. Dans ce dernier cas, le processus courant n'exécute pas la tranche d'itérations. Si aucune de ces deux conditions n'est vérifiée, alors la distribution n'est pas correcte. Il s'agit de la vérification dynamique de la propriété de *bonne distribution* (lignes 12-22) définie au chapitre précédent (voir la section 4.3.4). Le changement de domaine est effectué pour chaque référence à un tableau distribué dans le nid de boucles.

5.3.8 Exploitation des différents cœurs d'un même nœud

Le schéma de compilation traduisant les références d'un nid de boucles (algorithme 7) génère un code visitant les points d'une tranche d'itérations (ligne 29).

Algorithm 6 *compile_ordered*

Entrée: $ln_stmt, body, \mathcal{V}_{ln}, \mathcal{C}_{ln}$
Sortie: le code ln_stmt compilé dans le domaine distribué

1: $p = rank()$
2: $\vec{p}_{ln} = id_in_grid_{ln}(p)$
3: $Received = \emptyset$
4: **for** v_{ln} $scan_v$ $\mathcal{V}_{ln}$ **do**
5: $\quad \vec{\tilde{p}}_{ln} = extend_id(\vec{p}_{ln}, v_{ln})$
6: $\quad$ **for** c_{ln} $scan_c$ $\mathcal{C}_{ln}$ **do**
7: $\quad\quad is = iteration_slice(\vec{\tilde{p}}_{ln}, \vec{c}_{ln})$
8: $\quad\quad$ **for** each array x written in ln_stmt **do** ▷ compiler
9: $\quad\quad\quad \vec{p}_x = id_in_grid_x(p)$
10: $\quad\quad\quad$ **for** $(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln}) \in Before(\vec{p}_{ln}, \vec{c}_{ln})$ **do**
11: $\quad\quad\quad\quad$ **for** each WRITE reference ref_x^m to array x in ln_stmt **do** ▷ compiler
12: $\quad\quad\quad\quad\quad (\vec{c_x^m}, v_x^m) = belongs_to(\vec{p}_x, ref_x^m(min(is)), ref_x^m(max(is)))$
13: $\quad\quad\quad\quad\quad$ **if** $element_undefined(\vec{c_x^m}, v_x^m)$ **then**
14: $\quad\quad\quad\quad\quad\quad$ *abort("bad distribution")*
15: $\quad\quad\quad\quad\quad$ **end if**
16: $\quad\quad\quad\quad\quad \vec{\tilde{p}}_x^m = extend_id(\vec{p}_x, v_x^m)$
17: $\quad\quad\quad\quad\quad generate_sync_recv(stmt_ln, ref_x^m, \vec{\tilde{p}}_x^m, \vec{c_x^m}, \vec{\tilde{q}}_{ln}, \vec{c'}_{ln})$ ▷ compiler
18: $\quad\quad\quad\quad$ **end for**
19: $\quad\quad\quad\quad Received = Received \cup \{(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln})\}$
20: $\quad\quad\quad$ **end for**
21: $\quad\quad$ **end for**
22: $\quad\quad compile_iteration_slice(body, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln})$ ▷ compiler
23: $\quad\quad generate_sends(ln_stmt, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln})$ ▷ compiler
24: $\quad$ **end for**
25: **end for**
26: **if** reduction **then** ▷ compiler
27: $\quad generate_parallel_reductions(op, var_list)$ ▷ compiler
28: **end if**
29: $generate_recvs(ln_stmt, Received)$ ▷ compiler
30: $generate_completes(ln_stmt)$ ▷ compiler

Dans le code en sortie, nous générons une directive *omp parallel for* au dessus de la dimension parallèle la plus externe du vecteur $\vec{i}$. Si une telle dimension est entourée de dimensions ordonnées, alors ces dernières sont passées à la dimension parallèle dans une clause *firstprivate*. Nous n'avons pas montré l'insertion des directives OpenMP pour garder une représentation compacte du vecteur $\vec{i}$. La même remarque vaut pour le schéma de compilation des références en utilisant les régions de tableaux (voir l'algorithme 13).

5.4 Génération des communications

Des communications doivent être générées car certains éléments de tableaux dans l'espace séquentiel possèdent plusieurs copies dans l'espace distribué pour les distributions avec halo : la valeur d'un élément modifié par une tranche d'itérations doit être propagée à toutes les copies de cet élément. Un processus dont la tranche d'itérations modifie un élément propage sa valeur aux copies : ce sont les envois de données ou les

Algorithm 7 *compile_iteration_slice*

Entrée: $body, p, \vec{\vec{p}}_{ln}, \vec{c}_{ln}$
Sortie: le code d'une tranche d'itération compilé dans le domaine distribué

1: $is = iteration_slice(\vec{\vec{p}}_{ln}, \vec{c}_{ln})$
2: **for** each owner dimension k in loop nest ln **do**
3: $\quad extend(is, ln, k)$
4: **end for**
5: $\vec{i}_{low} = min(is)$
6: $\vec{i}_{up} = max(is)$
7: $computes = true$
8: $skips = true$
9: **for** each array x referenced in ln_stmt **do** ▷ compiler
10: $\quad \vec{p}_x = id_in_grid_x(p)$
11: $\quad$ **for** each reference m to array x **do** ▷ compiler
12: $\quad\quad (\vec{c_x^m}, v_x^m) = belongs_to(\vec{p}_x, ref_x^m(\vec{i}_{low}), ref_x^m(\vec{i}_{up}))$
13: $\quad\quad$ **if** $element_undefined((\vec{c_x^m}, v_x^m))$ **then**
14: $\quad\quad\quad$ **if** $!owner$ **then**
15: $\quad\quad\quad\quad$ *abort("bad distribution")*
16: $\quad\quad\quad$ **else**
17: $\quad\quad\quad\quad computes = false$
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **else**
20: $\quad\quad\quad \vec{\vec{p}}_x^m = extend_id(\vec{p}_x, v_x^m)$
21: $\quad\quad\quad skips = false$
22: $\quad\quad$ **end if**
23: $\quad$ **end for**
24: **end for**
25: **if** $!(computes \oplus skips)$ **then**
26: $\quad$ *abort("bad distribution")*
27: **end if**
28: **if** $computes$ **then**
29: $\quad$ **for** $\vec{i} = min_{<_{ln}}(is); \vec{i} <_{ln} max_{<_{ln}}(is) + \vec{inc}; \vec{i} = \vec{i} + \vec{inc}$ **do**
30: $\quad\quad$ **for** each reference ref_x^m in ln_stmt **do** ▷ compiler
31: $\quad\quad\quad (\vec{c_x^m}, \vec{l_x^m}) = element_copy(ref_x^m(\vec{i}), \vec{\vec{p}}_x^m)$
32: $\quad\quad$ **end for**
33: $\quad\quad body(..., X[\vec{c}_x^m][v_x^m][\vec{l_x^m}], ...)$
34: $\quad$ **end for**
35: **end if**

communications *send*. Un processus possédant des éléments dont les valeurs ont été modifiées par d'autres processus reçoit les nouvelles valeurs : ce sont les communications *recvs*. Pour les sends et les recvs, on s'intéresse exclusivement aux références en écriture car seules ces références sont susceptibles de modifier la mémoire distribuée.

5.4.1 Les voisins

Lors de la génération des communications, il n'est pas nécessaire de parcourir tous les processus pour calculer des intersections d'éléments de tableaux à échanger. En effet, pour chaque tableau x, la connaissance des tailles de blocs et du halo permet de calculer, pour chaque processus $\vec{p}_x$, les processus avec lesquels il est susceptible

de communiquer pour ce tableau. Cet ensemble constitue les *voisins* du processus $\vec{p}_x$ pour le tableau x (voir l'équation 5.9). On pose $L_x = Hlow_x + B_x + Hup_x$.

$$\begin{aligned} neighbours(\vec{\tilde{p}}_x) = \quad & \{\vec{\tilde{q}}_x | \vec{\tilde{q}}_x \in \mathcal{P}_x, \vec{\tilde{q}}_x \neq \vec{\tilde{p}}_x, \exists \vec{c}_x, \vec{c'_x} \in \mathcal{C}_x, \\ & \{\vec{a} | array_element(\vec{\tilde{p}}_x, \vec{c}_x, \vec{0}) \leq \vec{a} < array_element(\vec{\tilde{p}}_x, \vec{c}_x, L_x\vec{1})\} \cap \\ & \{\vec{a'} | array_element(\vec{\tilde{q}}_x, \vec{c'_x}, \vec{0}) \leq \vec{a'} < array_element(\vec{\tilde{q}}_x, \vec{c'_x}, L_x\vec{1})\} \neq \emptyset\} \end{aligned} \tag{5.9}$$

5.4.2 Génération des *send* pour une tranche d'itérations

L'algorithme 8 présente la génération des *send* pour une tranche d'itérations. Une tranche d'itérations est définie pour un processus étendu $\vec{\tilde{p}}_{ln}$ et un cycle du nid de boucles $\vec{c}_{ln}$ et est étendue en présence de dimensions *owner* aux bornes initiales des boucles selon ces dimensions (lignes 2-5). On calcule le cycle et le procesus virtuel du tableau auxquels appartiennent les éléments écrits. Le test sur la validité de ce couple d'éléments permet de ne pas générer de *send* pour un processus n'ayant pas effectué de calcul (en présence de dimension *owner*). On parcourt ensuite les voisins de ce processus à la recherche de copies des éléments modifiés afin de leur envoyer les nouvelles valeurs. Pour éviter qu'un processus ne communique avec lui même les données qu'il a modifiées, on distingue le processus parcouru du processus courant : $\vec{\tilde{q}}_x \neq \vec{\tilde{p}}_x$. Cette condition est déjà satisfaite car $\vec{\tilde{q}}_x$ appartient aux voisins de $\vec{\tilde{p}}_x$. Cependant, si le nid de boucles comporte des dimensions *owner*, les mises-à-jour selon les dimensions correspondantes du tableau x auront été effectuées, dans la phase de calcul, en redondance sur tous les processus sur lesquels les données accédées sont allouées. Il faut donc projeter (éliminer) ses dimensions des vecteurs $\vec{\tilde{p}}_x$ et $\vec{\tilde{q}}_x$ avant comparaison. Enfin, si les tableau x est distribué avec une distribution multi-partitionnée, la dimension diagonalisée des processus $\vec{\tilde{p}}_x$ et $\vec{\tilde{q}}_x$ n'est pas projetée pour permettre d'effectuer la communication selon cette dimension même si elle était accédé par une dimension *owner* du nid de boucles (ligne 14-16). Une fois le voisin $\vec{\tilde{q}}_x$ identifié, une communication *send* est générée pour lui envoyer une copie de chaque élément modifié par $\vec{\tilde{p}}_x$ dont le processus $\vec{\tilde{q}}_x$ possède une copie locale (lignes 18-23). On notera que pour pouvoir réaliser la communication *send*, on a besoin de connaître le vrai processus q auquel correspond le processus $\vec{\tilde{q}}_x$ (ligne 17).

5.4.3 Génération des *recvs*

Pour un nid de boucles ordonné, on génère une version synchrone des *recvs* (algorithme 9). Ce schéma impose, pour une tranche d'itérations, d'attendre la fin de l'exécution de toutes les tranches d'itérations la précédant (voir *compile_ordered* 6). On retrouve ainsi un ordonnancement distribué conforme à l'ordonnancement dans l'espace séquentiel.

Les autres *recvs* sont générés, tel que présenté à l'algorithm 10 de façon symétrique aux *sends* : les tranches d'itérations des autres processus sont énumérées et les éléments modifiés sont reçus sur le processus local s'il en possède des copies. On

Algorithm 8 *generate_sends*

Entrée: $ln_stmt, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln}$
Sortie: le code de ln_stmt avec les communications *send* insérées

1: $\vec{p}_{ln} = id_in_grid_{ln}(p)$
2: $is = iteration_set(\vec{\tilde{p}}_{ln}, \vec{c}_{ln})$
3: **for** each owner dimension k in loop nest ln **do**
4: $\quad is = extend(is, ln, k)$
5: **end for**
6: **for** each array x written in ln_stmt **do** ▷ compiler
7: $\quad \vec{p}_x = id_in_grid_x(p)$
8: $\quad$ **for** each WRITE reference ref_x^m to array x in ln_stmt **do** ▷ compiler
9: $\quad\quad owner_x^m = owner_dimensions(ref_x^m, ln)$ ▷ compiler
10: $\quad\quad (\vec{c}_x^m, v_x^m) = belongs_to(\vec{p}_x, ref_x^m(min(is)), ref_x^m(max(is)))$
11: $\quad\quad$ **if** $element_defined((\vec{c}_x^m, v_x^m))$ **then**
12: $\quad\quad\quad \vec{\tilde{p}}_x^m = extend_id_x(\vec{p}_x^m, v_x^m)$
13: $\quad\quad\quad$ **for** $\vec{\tilde{q}}_x \in neighbours(\vec{\tilde{p}}_x)$ **do**
14: $\quad\quad\quad\quad \vec{me} = project_owner(\vec{\tilde{p}}_x^m, (owner_x - diag_x)$ ▷ compiler
15: $\quad\quad\quad\quad \vec{other} = project_owner(\vec{\tilde{q}}_x, (owner_x - diag_x))$ ▷ compiler
16: $\quad\quad\quad\quad$ **if** $\vec{me} \neq \vec{other}$ **then**
17: $\quad\quad\quad\quad\quad q = id_in_grid_x^{-1}(first(extend_id^{-1}(\vec{\tilde{q}}_x)))$
18: $\quad\quad\quad\quad\quad$ **for** $\vec{i} = min_{<_{ln}}(is); \vec{i} <_{ln} max_{<_{ln}}(is) + \vec{inc}; \vec{i} = \vec{i} + \vec{inc}$ **do**
19: $\quad\quad\quad\quad\quad\quad$ **if** $element_defined(element_copy(ref_x^m(\vec{i}), \vec{\tilde{q}}_x))$ **then**
20: $\quad\quad\quad\quad\quad\quad\quad (\vec{c}_x^m, \vec{l}_x^m) = element_copy(ref_x^m(\vec{i}), \vec{\tilde{p}}_x^m)$
21: $\quad\quad\quad\quad\quad\quad\quad async_send(X[\vec{c}_x^m][\vec{v}_x^m][\vec{l}_x^m], q)$
22: $\quad\quad\quad\quad\quad\quad$ **end if**
23: $\quad\quad\quad\quad\quad$ **end for**
24: $\quad\quad\quad\quad$ **end if**
25: $\quad\quad\quad$ **end for**
26: $\quad\quad$ **end if**
27: $\quad$ **end for**
28: **end for**

exclut toutefois de cette énumération toutes les tranches d'itérations dont les contributions auraient déjà été considérées dans la phase de réception synchrone (ligne 15).

5.4.4 Complétion des communications

À chaque communication sur un tableau, un descripteur de requête est associé. La complétion des communications sur un tableau consiste alors simplement à terminer toutes les communications en progression sur ce tableau identifiées par les requêtes associées (algorithme 11).

5.4.5 Correction de la mémoire distribuée après exécution des communications

Maintenant que la génération des communications *send* et *recv* est définie pour un nid de boucles à ordonnancement quelconque, nous montrons comment les communi-

Algorithm 9 *generate_sync_recvs*

Entrée: $ln_stmt, ref_x^m, \vec{\tilde{p}}_x, \vec{c}_x, \vec{\tilde{p}'}_{ln}, \vec{c'}_{ln}$
Sortie: le code de ln_stmt avec les communications $recv$ synchrones insérées pour la référence ref_x^m

1: $p' = id_in_grid^{-1}(first(extend_id^{-1}(\vec{\tilde{p}'}_{ln})))$
2: $\vec{p'_x} = id_in_grid_x(p')$
3: $owner_x^m = owner_dimensions(ref_x^m, ln)$
4: $is = iteration_slice(\vec{\tilde{p}'}_{ln}, \vec{c'}_{ln})$
5: **for** each owner dimension k in loop nest ln **do**
6: $\quad is = extend(is, ln, k)$
7: **end for**
8: $(\vec{c'_x}, v'_x) = belongs_to(\vec{p'_x}, ref_x^m(min(is)), ref_x^m(max(is)))$
9: **if** $element_defined(\vec{c'_x}, v'_x)$ **then**
10: $\quad \vec{\tilde{p}'}_x = extend_id(\vec{p'}_x, v'_x)$
11: $\quad \vec{me} = project_owner(\vec{\tilde{p}}_x, (owner_x^m - diag_x))$
12: $\quad \vec{other} = project_owner(\vec{\tilde{p}'}_x, (owner_x^m - diag_x))$
13: $\quad$ **if** $(\vec{me} \neq \vec{other})$ **then**
14: $\quad\quad$ **for** $\vec{i} = min_{<_{ln}}(is); \vec{i} <_{ln} max_{<_{ln}}(is) + \vec{inc}; \vec{i} = \vec{i} + \vec{inc}$ **do**
15: $\quad\quad\quad (\vec{c}_x, \vec{l}_x) = element_copy(ref_x^m(\vec{i'}), \vec{\tilde{p}}_x)$
16: $\quad\quad\quad$ **if** $element_defined((\vec{c}_x, \vec{l}_x))$ **then**
17: $\quad\quad\quad\quad sync_recv(p', X[\vec{c}_x][v_x][\vec{l}_x])$
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end for**
20: $\quad$ **end if**
21: **end if**

cations garantissent la correction de la mémoire distribuée comme défini au chapitre *Modèle de distribution 4*.

Génération des communications pour un nid de boucles parallèle

Après exécution d'une tranche d'itérations, on insère les *send* correspondant aux éléments modifiés. Ceci permet de recouvrir les communications *send* avec le calcul de plusieurs tranches d'itérations. Le nid de boucles étant parallèle, il n'existe pas de dépendances entre itérations et donc pas de dépendances entre tranches d'itérations. Les *recvs* sont ainsi insérés après exécution de toutes les tranches d'itérations du nid de boucles. Toutes les communications (*send* et *recv*) peuvent ensuite être terminées. La complétion des communications à la fin de l'exécution d'un nid de boucles garantit que la mémoire distribuée est correcte avant l'exécution de la première tranche d'itération d'un prochain nid de boucles voir l'(algorithme 5).

Génération des communications pour un nid de boucles ordonné

Dans ce cas, l'ordre séquentiel des dimensions *ordered* doit être respecté lors de l'exécution des tranches d'itérations dans

l'espace distribué (algorithme 6). Cet ordre est garantit par la génération des communications synchrones (algorithme 6) pour toutes les tranches d'itérations possédant des *précédences* dans l'ordre initial du nid de boucles (voir la définition de l'ensemble

Algorithm 10 *generate_recvs*

Entrée: $ln_stmt, Received$

Sortie: le code de ln_stmt avec les communications $recv$ asynchrones insérées

1: $p = rank()$
2: **for** each array x written in ln_stmt **do** ▷ compiler
3: $\vec{p}_x = id_in_grid_x(p)$
4: **for** each WRITE reference ref_x^m to array x in ln_stmt **do** ▷ compiler
5: $owner_x^m = owner_dimensions(ref_x^m)$
6: **for** $v_x \in \mathcal{V}_x$ **do**
7: $\vec{\tilde{p}}_x = extend_id(\vec{p}_x, v_x)$
8: **for** $\vec{\tilde{q}}_x \in neighbours(\vec{\tilde{p}}_x)$ **do**
9: $\vec{q}_x = first(extend_id^-1(\vec{\tilde{q}}_x))$
10: $q = id_in_grid_x^-1(\vec{q}_x))$
11: $\vec{q}_{ln} = id_in_grid_{ln}(q)$
12: **for** $w_{ln} \in \mathcal{V}_{ln}$ **do**
13: $\vec{\tilde{q}}_{ln} = extend_id(\vec{q_{ln}}, w_{ln})$
14: **for** $\vec{c'}_{ln} \in \mathcal{C}_{ln}$ **do**
15: **if** $(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln}) \notin Received$ **then**
16: $is' = iteration_slice(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln})$
17: **for** each owner dimension k in ln **do**
18: $is' = extend(is', ln, k)$
19: **end for**
20: $(\vec{c_x}', w_x) = belongs_to(\vec{q_x}, ref_x^m(min(is')), ref_x^m(max(is')))$
21: **if** $element_defined((\vec{c_x}', w_x))$ **then**
22: $\vec{me} = project_owner(\vec{\tilde{p}}_x, owner_x^m - diag_x)$
23: $\vec{other} = project_owner(\vec{\tilde{q}}_x, owner_x^m - diag_x)$
24: **if** $\vec{me} \neq \vec{other}$ **then**
25: **for** $\vec{i} = min_{<_{ln}}(is'); \vec{i} <_{ln} max_{<_{ln}}(is') + \vec{inc}; \vec{i} = \vec{i} + \vec{inc}$ **do**
26: $(\vec{c_x}, \vec{l_x}) = element_copy(ref_x^m(\vec{i}'), \vec{\tilde{p}}_x)$
27: **if** $element_defined((\vec{c_x}, \vec{l_x}))$ **then**
28: $async_recv(q, X[\vec{c}_x][v_x][\vec{l_x}])$
29: **end if**
30: **end for**
31: **end if**
32: **end if**
33: **end if**
34: **end for**
35: **end for**
36: **end for**
37: **end for**
38: **end for**
39: **end for**

Before 5.8). Chaque processus exécutant une tranche d'itérations doit recevoir les données des processus exécutant les tranches d'itérations précédentes si ces dernières modifient des éléments dont des copies sont allouées sur ce processus. Cette communication garantit la correction de la mémoire distribuée avant l'exécution de la tranche d'itérations et assure ainsi que les éléments écrits auront des valeurs correctes.

Algorithm 11 *generate_completes*

Entrée: *ln_stmt*
Sortie: *ln_stmt* avec les complétions des communications insérées

```
1: for each array x written in ln_stmt do                    ▷ compiler
2:     for request ∈ Pending(x) do
3:         complete_comm(request)
4:         Pending(x) = Pending(x) − {(request)}
5:     end for
6: end for
```

5.5 Optimisations

Le schéma de compilation présenté à la section précédente montre le changement de domaine et la génération des communications élément par élément. Pour obtenir des performances, il n'est pas réaliste de déployer directement ce type de code car le surcoût à l'exécution serait exorbitant. Il faut donc apporter des améliorations au code généré afin d'obtenir des performances acceptables. Une première source d'amélioration est l'utilisation des régions de tableaux afin de réduire les tests de changement de domaine à un seul test par tranche d'itérations. L'utilisation des régions de tableaux permet également de regrouper toutes les communications d'une tranche d'itérations en une seule communication. Par la suite, nous montrerons comment exploiter la nature asynchrone des communications pour générer automatiquement des recouvrement calculs/communications sous certaines conditions.

5.5.1 Utilisation des régions de tableaux

On utilise les régions de tableaux *READ*, *WRITE* et *OUT* pour adapter les schémas de compilation présentés précédemment afin d'utiliser des résumés pour les communications et le changement de domaine.

5.5.2 Adaptation de la compilation d'un nid de boucle ordonné

Nous adaptons l'algorithme 6 en utilisant les régions de tableau $READ$ et $WRITE$ (algorithme 12) pour calculer l'ensemble des éléments accédés par une tranche d'itérations en lecture et en écriture pour chaque tableau x (ligne 11). La région $accessed_x$ est obtenue en calculant l'union des régions $READ_x$ et $WRITE_x$ pour tout le code du nid de boucles ln puis en projetant la région obtenue aux bornes de boucles de la tranche d'itérations considérée. La région $accessed_x$ regroupe toutes les références au tableau x. Pour une tranche d'itérations, les contributions de toutes les autres tranches d'itérations précédentes sont considérées pour la région $accessed_x$ (lignes 8-20).

Algorithm 12 *compile_ordered*

Entrée: $ln_stmt, body, \mathcal{V}_{ln}, \mathcal{C}_{ln}$
Sortie: le code ln_stmt compilé dans le domaine distribué

1: $p = rank()$
2: $\vec{p}_{ln} = id_in_grid_{ln}(p)$
3: $Received = \emptyset$
4: **for** v_{ln} $scan_v$ $\mathcal{V}_{ln}$ **do**
5: $\quad \vec{\tilde{p}}_{ln} = extend_id(\vec{p}_{ln}, v_{ln})$
6: $\quad$ **for** c_{ln} $scan_c$ $\mathcal{C}_{ln}$ **do**
7: $\quad\quad is = iteration_slice(\vec{\tilde{p}}_{ln}, \vec{c}_{ln})$
8: $\quad\quad$ **for** each array x written in ln_stmt **do** ▷ compiler
9: $\quad\quad\quad \vec{p}_x = id_in_grid_x(p)$
10: $\quad\quad\quad$ **for** $(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln}) \in Before(\vec{p}_{ln}, \vec{c}_{ln})$ **do**
11: $\quad\quad\quad\quad accessed_x = read_region(x, body, is) \cup write_region(x, body, is)$
12: $\quad\quad\quad\quad (\vec{c_x}, v_x) = belongs_to(\vec{p}_x, min(accessed_x), max(accessed_x))$
13: $\quad\quad\quad\quad$ **if** $element_undefined(\vec{c_x}, v_x)$ **then**
14: $\quad\quad\quad\quad\quad abort(\text{"bad distribution"})$
15: $\quad\quad\quad\quad$ **end if**
16: $\quad\quad\quad\quad \vec{\tilde{p}}_x = extend_id(\vec{p}_x, v_x)$
17: $\quad\quad\quad\quad generate_sync_recv(stmt_ln, \vec{\tilde{p}}_x, \vec{c}_x, \vec{\tilde{q}}_{ln}, \vec{c'}_{ln})$ ▷ compiler
18: $\quad\quad\quad$ **end for**
19: $\quad\quad\quad Received = Received \cup \{(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln})\}$
20: $\quad\quad$ **end for**
21: $\quad\quad compile_iteration_slice(body, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln})$ ▷ compiler
22: $\quad\quad generate_sends(ln_stmt, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln})$ ▷ compiler
23: $\quad$ **end for**
24: **end for**
25: **if** reduction **then** ▷ compiler
26: $\quad generate_parallel_reductions(op, var_list)$ ▷ compiler
27: **end if**
28: $generate_recvs(ln_stmt, Received)$ ▷ compiler
29: $generate_completes(ln_stmt)$ ▷ compiler

5.5.3 Adaptation de la compilation d'une tranche d'itérations

Dans l'algorithme 13, on résume tous les accès en lecture et en écriture à un tableau x pour une tranche d'itérations par une région $accessed_x$. Le test de localité est alors effectué sur cette région, sans avoir à considérer une à une toutes les références au tableau x. Ensuite, un vecteur de translation $\vec{shift}_x$ est calculé pour la région accédée (ligne 21) afin de traduire les accès pour toutes les références à ce tableau dans le domaine distribué (lignes 29-31) sans avoir à calculer explicitement la valeur $\vec{l}_x^m$ pour chaque référence ref_x^m au tableau x pour toute itération $\vec{i}$ (voir l'algorithme 7).

5.5.4 Adaptation de la génération des communications *send*

Nous adaptons l'algorithme 8 en utilisant les régions de tableau $WRITE$ et OUT (algorithme 14). La région $WRITE$ d'un tableau définit tous les éléments écrits par la tranche d'itérations à sa sortie, résumant ainsi toutes les références en écriture à ce tableau. Un premier avantage de l'utilisation des régions $WRITE$ est donc de

Algorithm 13 *compile_iteration_slice*

Entrée: $body, p, \vec{\vec{p}}_{ln}, \vec{c}_{ln}$
Sortie: le code d'une tranche d'itération compilé dans le domaine distribué

1: $is = iteration_slice(\vec{\vec{p}}_{ln}, \vec{c}_{ln})$
2: **for** each owner dimension k in loop nest ln **do**
3: $\quad extend(is, ln, k)$
4: **end for**
5: $\vec{i}_{low} = min(is)$
6: $\vec{i}_{up} = max(is)$
7: $computes = true$
8: $skips = true$
9: **for** each array x referenced in ln_stmt **do** ▷ compiler
10: $\quad \vec{p}_x = id_in_grid_x(p)$
11: $\quad accessed_x = read_region(x, body, is) \cup write_region(x, body, is)$
12: $\quad (\vec{c_x}, v_x) = belongs_to(\vec{p}_x, min(accessed_x), max(accessed_x))$
13: $\quad$ **if** $element_undefined((\vec{c_x}, v_x))$ **then**
14: $\quad\quad$ **if** $!owner$ **then**
15: $\quad\quad\quad$ *abort("bad distribution")*
16: $\quad\quad$ **else**
17: $\quad\quad\quad computes = false$
18: $\quad\quad$ **end if**
19: $\quad$ **else**
20: $\quad\quad \vec{\vec{p}}_x = extend_id(\vec{p}_x, v_x)$
21: $\quad\quad \vec{shift}_x = array_element(\vec{\vec{p}}_x, \vec{c}_x, \vec{0})$
22: $\quad\quad skips = false$
23: $\quad$ **end if**
24: **end for**
25: **if** $!(computes \oplus skips)$ **then**
26: $\quad$ *abort("bad distribution")*
27: **end if**
28: **if** $computes$ **then**
29: $\quad$ **for** $\vec{i} = min_{<_{ln}}(is); \vec{i} <_{ln} max_{<_{ln}}(is) + \vec{inc}; \vec{i} = \vec{i} + \vec{inc}$ **do**
30: $\quad\quad body(..., X[\vec{c}_x][v_x][ref_x^m(\vec{i}) - shift_x], ...)$
31: $\quad$ **end for**
32: **end if**

considérer la dernière valeur écrite dans la tranche d'itérations pour chaque élément et ne pas générer de communications pour des éléments écrits par une instance d'instruction et tués par d'autres instances d'instruction de la tranche d'itérations considérée. Un second avantage est de regrouper tous les éléments d'un tableau à envoyer pour une tranche d'itérations en un seul message.

Une région $WRITE_x$ pour une tranche d'itérations décrit les éléments écrits pour l'ensemble de ses itérations, indépendamment de leur utilisation future. L'intersection de cette région avec la région OUT_x définit une région $live_x$ (ligne 11), qui permet de ne retenir pour la communication que les éléments effectivement utilisés dans la suite des calculs. Lorsque l'on considère un programme dans l'espace séquentiel, une région OUT_x pour un morceau de code est toujours un sous ensemble de la région $WRITE_x$. Dans l'espace distribué, cependant, on projette la région $WRITE_x$ de l'espace séquentiel aux bornes de la tranche d'itérations considérée alors que la région OUT_x garde la même signification que dans l'espace séquentiel.

Pour chaque processus parmi les voisins (lignes 16-19), si la région $live_x$ a une intersection non vide avec la mémoire allouée localement sur ce processus alors une communication *send* est générée (lignes 20-26).

Algorithm 14 *generate_sends*

Entrée: $ln_stmt, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln}$
Sortie: le code de ln_stmt avec les communications *send* insérées

1: $\vec{p}_{ln} = id_in_grid_{ln}(p)$
2: $is = iteration_set(\vec{\tilde{p}}_{ln}, \vec{c}_{ln})$
3: **for** each owner dimension k in loop nest ln **do**
4: $\quad is = extend(is, ln, k)$
5: **end for**
6: **for** each array x written in ln_stmt **do** ▷ compiler
7: $\quad \vec{p}_x = id_in_grid_x(p)$
8: $\quad owner_x = owner_dimensions(x, ln_stmt)$ ▷ compiler
9: $\quad write_x = write_region(x, body, is)$
10: $\quad out_x = out_region(x, body)$
11: $\quad live_x = write_x \cap out_x$
12: $\quad$ **if** $live_x \neq empty_region$ **then**
13: $\quad\quad (\vec{c}_x, v_x) = belongs_to(\vec{p}_x, min(live_x), max(live_x))$
14: $\quad\quad$ **if** $element_defined((\vec{c}_x, v_x))$ **then**
15: $\quad\quad\quad \vec{\tilde{p}}_x = extend_id(\vec{p}_x, v_x)$
16: $\quad\quad\quad$ **for** $\vec{\tilde{q}}_x \in neighbours(\vec{\tilde{p}}_x)$ **do**
17: $\quad\quad\quad\quad \vec{me} = project_owner(\vec{\tilde{p}}_x, (owner_x - diag_x))$ ▷ compiler
18: $\quad\quad\quad\quad \vec{other} = project_owner(\vec{\tilde{q}}_x, (owner_x - diag_x))$ ▷ compiler
19: $\quad\quad\quad\quad$ **if** $\vec{me} \neq \vec{other}$ **then**
20: $\quad\quad\quad\quad\quad$ **for** $\vec{c'_x} \in \mathcal{C}_x$ **do**
21: $\quad\quad\quad\quad\quad\quad to_send = live_x \cap \{\vec{a} | array_element(\vec{\tilde{q}}_x, \vec{c'_x}, \vec{0}) \leq \vec{a} < array_element(\vec{\tilde{q}}_x, \vec{c'_x}, L_x\vec{1})\}$
22: $\quad\quad\quad\quad\quad\quad$ **if** $to_send \neq \emptyset$ **then**
23: $\quad\quad\quad\quad\quad\quad\quad \vec{shift}_x = array_element(\vec{\tilde{p}}_x, \vec{c}_x, \vec{0})$
24: $\quad\quad\quad\quad\quad\quad\quad async_send(X[\vec{c}_x][v_x], q, to_send, \vec{shift}_x)$
25: $\quad\quad\quad\quad\quad\quad$ **end if**
26: $\quad\quad\quad\quad\quad$ **end for**
27: $\quad\quad\quad\quad$ **end if**
28: $\quad\quad\quad$ **end for**
29: $\quad\quad$ **end if**
30: $\quad$ **end if**
31: **end for**

5.5.5 Adaptation de la génération des *recvs*

Nous adaptons l'algorithme 10 pour la génération des communications *recv* en utilisant les régions de tableau (algorithme 15) de façon symétrique à la génération des communications *send*. Les régions *WRITE* et *OUT* sont utilisées pour construire des résumés des éléments à recevoir. On parcourt les tranches d'itérations exécutées par les autres processus pour construire des régions $live_x$. Si l'intersection d'une région $live_x$ avec la mémoire allouée localement pour un tableau x est non nulle, alors une

communication *async_recv* est générée. La version synchrone pour un nid de boucles ordonné en utilisant les régions de tableau est décrite par l'algorithme 16.

Algorithm 15 *generate_recvs*

Entrée: $ln_stmt, Received$
Sortie: le code de ln_stmt avec les communications *recv* asynchrones insérées

1: $p = rank()$
2: **for** each array x written in ln_stmt **do** ▷ compiler
3: $\vec{p}_x = id_in_grid_x(p)$
4: $owner_x = owner_dimensions(x, ln_stmt)$
5: **for** $v_x \in \mathcal{V}_x$ **do**
6: $\vec{\tilde{p}}_x = extend_id(\vec{p}_x, v_x)$
7: **for** $\vec{\tilde{q}}_x \in neighbours(\vec{\tilde{p}}_x)$ **do**
8: $\vec{q}_x = first(extend_id^{-1}(\vec{\tilde{q}}_x))$
9: $q = id_in_grid_x^{-1}(\vec{q}_x))$
10: $\vec{q}_{ln} = id_in_grid_{ln}(q)$
11: **for** $w_{ln} \in \mathcal{V}_{ln}$ **do**
12: $\vec{\tilde{q}}_{ln} = extend_id(\vec{q}_{ln}, w_{ln})$
13: **for** $c'_{ln} \in \mathcal{C}_{ln}$ **do**
14: $is' = iteration_slice(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln})$
15: **for** each owner dimension k in ln **do**
16: $is' = extend(is', ln, k)$
17: **end for**
18: $write'_x = write_region(x, body, is')$
19: $out'_x = out_region(x, body)$
20: $live'_x = write'_x \cap out'_x$
21: **if** $live'_x \neq \emptyset$ **then**
22: $(\vec{c'}_x, w_x) = belongs_to(\vec{q}_x, min(live'_x), max(live'_x))$
23: **if** $element_defined(\vec{c'}_x, w_x)$ **then**
24: $\vec{me} = project_owner(\vec{\tilde{p}}_x, (owner_x - diag_x))$
25: $\vec{other} = project_owner(\vec{\tilde{q}}_x, (owner_x - diag_x))$
26: **if** $\vec{me} \neq \vec{other}$ **then**
27: **for** $c_x \in \mathcal{C}_x$ **do**
28: $to_recv = live'_x \cap \{\vec{a} | \vec{a} \geq array_element(\vec{\tilde{p}}_x, \vec{c}_x, \vec{0}) \wedge \vec{a} < array_element(\vec{\tilde{p}}_x, \vec{c}_x, L_x\vec{1})\}$
29: **if** $to_recv \neq empty_region$ **then**
30: $\vec{shift}_x = array_element(\vec{\tilde{p}}_x, \vec{c}_x, \vec{0})$
31: $async_recv(q, X[\vec{c}_x][v_x], to_recv, \vec{shift}_x)$
32: **end if**
33: **end for**
34: **end if**
35: **end if**
36: **end if**
37: **end for**
38: **end for**
39: **end for**
40: **end for**
41: **end for**

Algorithm 16 *generate_sync_recvs*

Entrée: $ln_stmt, \vec{\tilde{p}}_x, \vec{c}_x, \vec{\tilde{q}}_{ln}, \vec{c'}_{ln}$
Sortie: le code de ln_stmt avec les communications $recv$ synchrones insérées

1: $q = id_in_grid_x^{-1}(first(extend_id^{-1}(\vec{\tilde{q}}_{ln})))$
2: $\vec{q_x} = id_in_grid_x(q)$
3: $owner_x = owner_dimensions(x, ln_stmt)$
4: $is' = iteration_slice(\vec{\tilde{q}}_{ln}, \vec{c'}_{ln})$
5: **for** each owner dimension k in loop nest ln **do**
6: $\quad is' = extend(is', ln, k)$
7: **end for**
8: $write'_x = write_region(x, body, is')$
9: $out'_x = out_region(x, body)$
10: $live'_x = write'_x \cap out'_x$
11: **if** $live'_x \neq \emptyset$ **then**
12: $\quad (\vec{c'_x}, w_x) = belongs_to(\vec{q}_x, min(live'_x), max(live'_x))$
13: $\quad$ **if** $element_defined(\vec{c'_x}, w_x)$ **then**
14: $\quad\quad \vec{\tilde{q}}_x = extend_id(\vec{q}_x, w_x)$
15: $\quad\quad \vec{me} = project_owner(\vec{\tilde{p}}_x, (owner_x - diag_x))$
16: $\quad\quad \vec{other} = project_owner(\vec{\tilde{q}}_x, (owner_x - diag_x))$
17: $\quad\quad$ **if** $(\vec{me} \neq \vec{other})$ **then**
18: $\quad\quad\quad$ **for** $c_x \in \mathcal{C}_x$ **do**
19: $\quad\quad\quad\quad to_recv = live'_x \cap \{\vec{a} | \vec{a} \geq array_element(\vec{\tilde{p}}_x, \vec{c}_x, \vec{0}) \wedge \vec{a} < array_element(\vec{\tilde{p}}_x, \vec{c}_x, L_x\vec{1})\}$
20: $\quad\quad\quad\quad$ **if** $to_recv \neq empty_region$ **then**
21: $\quad\quad\quad\quad\quad \vec{shift}_x = array_element(\vec{\tilde{p}}_x, \vec{c}_x, \vec{0})$
22: $\quad\quad\quad\quad\quad sync_recv(q, X[\vec{c}_x][v_x], to_recv, \vec{shift}_x)$
23: $\quad\quad\quad\quad$ **end if**
24: $\quad\quad\quad$ **end for**
25: $\quad\quad$ **end if**
26: $\quad$ **end if**
27: **end if**

5.5.6 Recouvrement des communications par les calculs

Les communications générées, à l'exception d'une partie des communications pour un nid de boucles ordonné sont asynchrones. Il faut donc assurer leur terminaison avant toute utilisation future des données communiquées. Dans le schéma de compilation d'un nid de boucles ordonné, les communications dont dépend l'exécution d'une tranche d'itérations sont terminées avant son exécution. Les autres communications sont terminées après exécution du nid de boucles. Pour un nid de boucles à ordonnancement parallèle, les communications sont également terminées après du nid de boucles. Ce schéma offre par construction un moyen simple de recouvrir les communications par les calculs des tranches d'itérations d'un même nid de boucles. Ce recouvrement est cependant insuffisant car :

1. Les complétions interviennent trop tôt : tous les calculs qui s'intercalent entre la fin de l'exécution d'un nid de boucles et la prochaine utilisation des tableaux impliqués dans des communications pourraient se faire en parallèle de la progression de ces communications.

2. La complétion des communications pour un tableau donné concerne toutes les communications en progression sur ce tableau, or des parties différentes du tableau peuvent être utilisées à différents moments.

Nous améliorons les deux points précédents en utilisant les régions *accessed*, *to_send* et *to_recv*. On associe à chaque tableau impliqué dans une communication la région *to_send* ou *to_recv* concernée. Ensuite, pour chaque région *accessed*, on calcule les intersections avec les régions *to_send* et *to_recv* associées au tableau. Pour chaque intersection non vide, il faut terminer la communication correspondante avant de poursuivre le calcul.

Dans les algorithmes présentés précédemment, on remplace la complétion explicite des communications pour un nid de boucles ln, $generate_completes(ln_stmt)$, par l'algorithme 17 qui ne termine une communication que si les données concernées sont effectivement accédées par une tranche d'itérations. L'ensemble $Pending(x)$ contient des couples décrivant les communications en progression sur le tableau x. Il s'agit de la région et de l'identifiant de la requête associés à la communication. Lorsqu'une communication est complétée, le couple correspondant est retiré de l'ensemble $Pending(x)$. On insère dans le code généré pour *compile_iteration_slice*, pour chaque région $accessed_x$, un appel $complete_on_access(x, accessed_x)$ juste avant l'exécution de la tranche d'itérations.

Algorithm 17 *complete_on_access*

Entrée: ln_stmt, $accessed_x$

Sortie: ln_stmt avec les complétions des communications insérées si la région $accessed_x$ est impliquée dans une communication

1: **for** $(request, region) \in Pending(x)$ **do**
2: **if** $(accessed_x \cap region) \neq \emptyset$ **then**
3: $complete_comm(request)$
4: $Pending(x) = Pending(x) - \{(request, region)\}$
5: **end if**
6: **end for**

5.6 Exemple de génération de code

Dans cette section nous montrons la génération de code pour un nid de boucles représentant un calcul de substitution arrière lors de la résolution d'un système d'équations, extrait du programme *NAS BT*. La dimension i est ordonnée et la dimension k est diagonalisée. Toutes les dimensions du nid de boucles sont distribuées par blocs (listing 5.5).

Dans le code généré par *dSTEP*, le nid de boucles est *outliné* dans une fonction `x_backsubstitute_GRIDIFY_HYBRID` et remplacé par un appel à cette fonction (listing 5.6).

Les listing suivants montre le code généré pour la fonction `x_backsubstitute_GRIDIFY_HYBRID`. Le listing 5.7 montre les déclarations des variables correspondant notamment aux matrices de distribution, codées comme des vecteurs, à la gestion des cycles, des

```
1  void x_backsubstitute(double rhs[162][162][162][5], double lhs
       [162][162][162][3][5][5]) {
2
3    int i, j, k, m, n;
4
5  #pragma step gridify(i(dist=block; sched=ordered), j, k(dist=block, diag; sched=
       parallel)) private(m, n)
6    for (i = grid_points[0]-2; i >= 0; i--) {
7      for (j = 1; j < grid_points[1]-1; j++) {
8        for (k = 1; k < grid_points[2]-1; k++) {
9          for (m = 0; m < BLOCK_SIZE; m++) {
10           for (n = 0; n < BLOCK_SIZE; n++) {
11             rhs[i][j][k][m] = rhs[i][j][k][m]
12               - lhs[i][j][k][CC][m][n]*rhs[i+1][j][k][n];
13           }
14         }
15       }
16     }
17   }
18
19 }
```

Listing 5.5 – Distribution du nid de boucles de la fonction *x_backsubsitute* de BT

```
1 void x_backsubstitute(void *rhs, void *lhs)
2 {
3    int i, j, k, m, n;
4    x_backsubstitute_GRIDIFY_HYBRID(grid_points, &i, &j, &k, &m, &n, lhs, rhs);
5 }
```

Listing 5.6 – Remplacement du nid de boucles de la fonction *x_backsubsitute* par un appel de fonction

processus virtuels ainsi que des incréments et de l'ordre du nid de boucles. À la ligne 7, la fonction `DSTEP_DISTRIBUTE_LOOP` calcule les valeurs des matrices de distribution du nid de boucles. Un identifiant unique (60 dans ce cas) est généré pour ce nid de boucles afin de ne calculer les informations de distribution qu'une seule fois. Ces informations ensuite directement réutilisées pour toutes les prochaines invocations du nid de boucles.

Le listing 5.8 montre les communications *recv* pour les tranches d'itérations *précédentes*, imposées par la contrainte d'ordre sur la dimension i du nid de boucles. Ici, les *recv* bloquants sont implémentés comme des *recv* non bloquants, avec un appel explicite à la complétion des communications correspondantes juste avant le calcul (voir ligne 6 du listing 5.9).

Le listing 5.9 montre le calcul du nid de boucles pour chaque tranche d'itérations. Le calcul est effectué pour les nouvelles bornes de boucles calculées pour la tranche d'itérations courante et les références sont traduites dans le domaine distribué. Les lignes 17-18 montre la génération des communications pour la tranche d'itérations. La région $live_x$ est calculée pour le tableau *rhs* (appelée ici `__send_rhs`) et passée au support d'exécution pour calculer les intersections avec la mémoire des voisins et effectuer les communications correspondantes.

Enfin, le listing 5.10 montre la génération des communications *recv* restantes pour le nid de boucles (une première vague de *recv* ayant été réalisée avant le calcul). La

```
1  void x_backsubstitute_GRIDIFY_HYBRID(int grid_points[3], int i_0[1], int j_0[1], int
       k_0[1], int m_0[1], int n_0[1], void *lhs, void *rhs)
2  {
3     //PIPS generated variable
4     int _i, _c_rhs, _v_rhs, _c_VECT_rhs[4], _LOW_rhs[4], _DIMS_rhs[4], _c_lhs, _v_lhs,
        _c_VECT_lhs[6], _LOW_lhs[6], _DIMS_lhs[6], _OWNERS_rhs[4] = {0, 0, 0, 0},
       _computes = 1, _C[3], _p[3], _order[3] = {-1, 0, 0}, _L[3] = {0, 1, 1}, _U[3] = {
       grid_points[0]-2, grid_points[1]-1-1, grid_points[2]-1-1}, _LOW[3], _INCR[3] =
       {-1, 1, 1}, _UP[3], _P[3] = {0, 0, 1}, _t_p[3], _rank, __rank, __p[3], __t_p[3],
       _v, ___v, _vprocs, _NB_NODES, _NB_c, _NB_N_rhs;
5     int *_N_rhs;
6     int _n_rhs, i, j, k, m, n;
7     DSTEP_DISTRIBUTE_LOOP(60, _C, 2, 3, DSTEP_INT_DIV(grid_points[0]-2+1-0, _P[0], 1),
        DSTEP_INT_DIV(grid_points[1]-1-1+1-1, _P[1], 1), DSTEP_INT_DIV(grid_points
       [2]-1-1+1-1, _P[2], 1));
8     n = *n_0;
9     m = *m_0;
10    k = *k_0;
11    j = *j_0;
12    i = *i_0;
```

Listing 5.7 – Code généré: déclarations, distribution du nid de boucles

ligne 31 montre la complétion de toutes les communications non encore terminées de ce nid de boucles.

```
1   //The first vproc of the diagonalized dimension
2   _v = DSTEP_FIRST_VPROC(_P, _p, _vprocs, 0, -1);
3   for (___v = 0;___v<_vprocs;___v++) {
4      int _c[3] = {0, 0, 0};
5      DSTEP_EXTEND_ID(3, _P, _p, _v, _t_p);
6      DSTEP_LOOP_BOUNDS(3, _LOW, _UP, _B, _P, _c, _t_p, _L, _U, _INCR);
7      int DSTEP_i_LOW = _LOW[0];
8      int DSTEP_i_UP = _UP[0];
9      int DSTEP_j_LOW = _LOW[1];
10     int DSTEP_j_UP = _UP[1];
11     int DSTEP_k_LOW = _LOW[2];
12     int DSTEP_k_UP = _UP[2];
13     //The rectangular hull of the accessed region of array rhs in the loop
       statement
14     int _ACCESSED_rhs[4][2] = {{DSTEP_GENERIC_MAX(2, 0, DSTEP_i_LOW), DSTEP_i_UP
       +1}, {DSTEP_GENERIC_MAX(2, DSTEP_j_LOW, 1), DSTEP_j_UP}, {DSTEP_GENERIC_MAX(2,
       DSTEP_k_LOW, 1), DSTEP_k_UP}, {0, 4}};
15     DSTEP_BELONGS_TO(rhs, _DIMS_rhs, &_c_rhs, &_v_rhs, _c_VECT_rhs, _LOW_rhs,
       _ACCESSED_rhs, 0, &_computes, 1);
16     double (*rhs__)[_vprocs][_DIMS_rhs[0]][_DIMS_rhs[1]][_DIMS_rhs[2]][_DIMS_rhs
       [3]] = (double (*)[_vprocs][_DIMS_rhs[0]][_DIMS_rhs[1]][_DIMS_rhs[2]][_DIMS_rhs
       [3]]) rhs;
17     double (*rhs)[_vprocs][_DIMS_rhs[0]][_DIMS_rhs[1]][_DIMS_rhs[2]][_DIMS_rhs[3]]
       = rhs__;
18     //Pre recvs
19     DSTEP_NEIGHBOURS(rhs, &_NB_N_rhs, &_N_rhs);
20     for (_n_rhs = 0;_n_rhs<_NB_N_rhs;_n_rhs += 1) {
21        __rank = _N_rhs[_n_rhs];
22        DSTEP_PROC_ID(60, __rank, 3, __p);
23        for (__v = 0;__v<_vprocs;__v++) {
24           int __c[3] = {0, 0, 0};
25           DSTEP_EXTEND_ID(3, _P, __p, __v, __t_p);
26           if (_t_p[0]<_P[0]-1&&__t_p[0]==_t_p[0]+1&&(__t_p[1]-_t_p[1]>=-1&&__t_p
             [1]-_t_p[1]<=1)&&(__t_p[2]-_t_p[2]>=-1&&__t_p[2]-_t_p[2]<=1)) {
27              DSTEP_LOOP_BOUNDS(3, _LOW, _UP, _B, _P, __c, __t_p, _L, _U, _INCR);
28              int DSTEP_i_LOW = _LOW[0];
29              int DSTEP_i_UP = _UP[0];
30              int DSTEP_j_LOW = _LOW[1];
31              int DSTEP_j_UP = _UP[1];
32              int DSTEP_k_LOW = _LOW[2];
33              int DSTEP_k_UP = _UP[2];
34              int __recv_rhs[4][2] = {{DSTEP_GENERIC_MAX(2, DSTEP_i_LOW, 0),
                DSTEP_i_UP}, {DSTEP_GENERIC_MAX(2, DSTEP_j_LOW, 1), DSTEP_j_UP}, {
                DSTEP_GENERIC_MAX(2, DSTEP_k_LOW, 1), DSTEP_k_UP}, {0, 4}};
35              DSTEP_RECV(rhs, __rank, __recv_rhs, sizeof(double), _OWNERS_rhs,
                _v_rhs, 0);
36           }
37        }
38     }
```

Listing 5.8 – Code généré: réception des données produites par les tranches d'itérations précédentes

5.7 Conclusion

Nous avons présenté dans ce chapitre un schéma de compilation générique permettant de traduire un code annoté de directives *dSTEP* du domaine séquentiel au domaine distribué. Nous avons d'abord montré la compilation *élément par élément* du code en entrée puis utilisé les régions de tableaux pour améliorer les performances du code généré notamment au niveau des communications. Le schéma de compilation présenté génère un code parallèle pour mémoire distribuée et partagée. Nous

```
1       //The rectangular hull of the accessed region of array lhs in the loop
    statement
2       int _ACCESSED_lhs[6][2] = {{DSTEP_GENERIC_MAX(2, DSTEP_i_LOW, 0), DSTEP_i_UP},
    {DSTEP_GENERIC_MAX(2, DSTEP_j_LOW, 1), DSTEP_j_UP}, {DSTEP_GENERIC_MAX(2,
    DSTEP_k_LOW, 1), DSTEP_k_UP}, {2, 2}, {0, 4}, {0, 4}};
3       DSTEP_BELONGS_TO(lhs, _DIMS_lhs, &_c_lhs, &_v_lhs, _c_VECT_lhs, _LOW_lhs,
    _ACCESSED_lhs, 0, &_computes, 1);
4       double (*lhs__)[_vprocs][_DIMS_lhs[0]][_DIMS_lhs[1]][_DIMS_lhs[2]][_DIMS_lhs
    [3]][_DIMS_lhs[4]][_DIMS_lhs[5]] = (double (*)[_vprocs][_DIMS_lhs[0]][_DIMS_lhs
    [1]][_DIMS_lhs[2]][_DIMS_lhs[3]][_DIMS_lhs[4]][_DIMS_lhs[5]]) lhs;
5       double (*lhs)[_vprocs][_DIMS_lhs[0]][_DIMS_lhs[1]][_DIMS_lhs[2]][_DIMS_lhs[3]][
    _DIMS_lhs[4]][_DIMS_lhs[5]] = lhs__;
6       DSTEP_COMPLETE_RECV(rhs, _v_rhs);
7
8       if (_computes) {
9          for(i = DSTEP_i_UP; i >= DSTEP_i_LOW; i += -1)
10 #pragma omp parallel for private(n, m, j, k) firstprivate(i)
11             for(j = DSTEP_j_LOW; j <= DSTEP_j_UP; j += 1)
12                for(k = DSTEP_k_LOW; k <= DSTEP_k_UP; k += 1)
13                   for(m = 0; m <= 4; m += 1)
14                      for(n = 0; n <= 4; n += 1)
15                         rhs[0][_v_rhs][i-_LOW_rhs[0]][j-_LOW_rhs[1]][k-_LOW_rhs[2]][m
    ] = rhs[0][_v_rhs][i-_LOW_rhs[0]][j-_LOW_rhs[1]][k-_LOW_rhs[2]][m]-lhs[0][_v_lhs
    ][i-_LOW_lhs[0]][j-_LOW_lhs[1]][k-_LOW_lhs[2]][2][m][n]*rhs[0][_v_rhs][i+1-
    _LOW_rhs[0]][j-_LOW_rhs[1]][k-_LOW_rhs[2]][n];
16
17         int __send_rhs[4][2] = {{DSTEP_GENERIC_MAX(2, DSTEP_i_LOW, 0), DSTEP_i_UP},
    {DSTEP_GENERIC_MAX(2, DSTEP_j_LOW, 1), DSTEP_j_UP}, {DSTEP_GENERIC_MAX(2,
    DSTEP_k_LOW, 1), DSTEP_k_UP}, {0, 4}};
18         DSTEP_SEND(rhs, _rank, _NB_NODES, _c_rhs, _v_rhs, _c_VECT_rhs, __send_rhs,
    sizeof(double), _OWNERS_rhs);
19      }
20      //Next vproc in the order imposed by the order of the diagonalized dimension
21      _v = (_v-1+_vprocs)%_vprocs;
22   }
```

Listing 5.9 – Code généré: changement de domaine, génération des *send*

présentons dans le chapitre suivant son adaptation à la génération de code pour des machines multi-GPU.

```
1     for (_v = 0;_v<_vprocs;_v++) {
2        int _c[3] = {0, 0, 0};
3        DSTEP_EXTEND_ID(3, _P, _p, _v, _t_p);
4        //Post recvs
5        DSTEP_NEIGHBOURS(rhs, &_NB_N_rhs, &_N_rhs);
6        for (_n_rhs = 0;_n_rhs<_NB_N_rhs;_n_rhs += 1) {
7           __rank = _N_rhs[_n_rhs];
8           DSTEP_PROC_ID(60, __rank, 3, __p);
9           for (__v = 0;__v<_vprocs;__v++) {
10             int __c[3] = {0, 0, 0};
11             DSTEP_EXTEND_ID(3, _P, __p, __v, __t_p);
12             if (!(_t_p[0]<_P[0]-1&&__t_p[0]==_t_p[0]+1&&(__t_p[1]-_t_p[1]>=-1&&__t_p
     [1]-_t_p[1]<=1)&&(__t_p[2]-_t_p[2]>=-1&&__t_p[2]-_t_p[2]<=1))) {
13                DSTEP_LOOP_BOUNDS(3, _LOW, _UP, _B, _P, __c, __t_p, _L, _U, _INCR);
14                int DSTEP_i_LOW = _LOW[0];
15                int DSTEP_i_UP = _UP[0];
16                int DSTEP_j_LOW = _LOW[1];
17                int DSTEP_j_UP = _UP[1];
18                int DSTEP_k_LOW = _LOW[2];
19                int DSTEP_k_UP = _UP[2];
20                int __recv_rhs[4][2] = {{DSTEP_GENERIC_MAX(2, DSTEP_i_LOW, 0),
     DSTEP_i_UP}, {DSTEP_GENERIC_MAX(2, DSTEP_j_LOW, 1), DSTEP_j_UP}, {
     DSTEP_GENERIC_MAX(2, DSTEP_k_LOW, 1), DSTEP_k_UP}, {0, 4}};
21                DSTEP_RECV(rhs, __rank, __recv_rhs, sizeof(double), _OWNERS_rhs, _v,
     0);
22             }
23          }
24       }
25    }
26    *i_0 = i;
27    *j_0 = j;
28    *k_0 = k;
29    *m_0 = m;
30    *n_0 = n;
31    DSTEP_COMPLETE_COMM(rhs);
32 }
```

Listing 5.10 – Code généré: génération des *recv* restant, complétion des communications

Chapitre 6

Extension du schéma de compilation pour les machines multi-GPU

6.1 Introduction

Comme présenté au chapitre 1, les accélérateurs matériels sont largement utilisés dans les architectures parallèles hybrides. On peut citer parmi les accélérateurs les plus répandues les co-processeurs Xeon Phi d'Intel, les GPU CUDA de NVIDIA et les GPU fabriqués par AMD. L'architecture des GPGPUs est conçue pour permettre d'effectuer des calculs réguliers sur de grandes quantités de données de façon plus efficace que sur les processeurs CPU. En effet, contrairement aux architectures CPU, les GPUs possèdent un grand nombre de transistors dédiés aux unités de calcul et moins de transistors dédiés aux unités de contrôle et aux mémoires caches.

La nature des applications cibles des GPGPUs coïncide avec celle du modèle de programmation de *dSTEP*. Il est donc naturel d'étendre la génération de code aux architectures hybrides équipées de GPUs.

Dans ce chapitre, nous présentons les travaux autour de la programmation des GPGPUs. Nous présentons ensuite les raisons d'une telle extension dans *dSTEP* plutôt que la réutilisation d'une solution spécialisante pour accélérateur du code généré pour CPU ainsi que les conditions de sa réalisation. Enfin nous présentons les modifications apportées au schéma de compilation pour pouvoir cibler, à la demande de l'utilisateur, la génération de code pour GPUs.

Nous nous limitons à la famille des GPGPUs comme accélérateurs et plus particulièrement à l'architecture CUDA de NVIDIA car cette dernière, comme montré en introduction (ch. 1), constitue le type d'accélérateurs les plus utilisés dans les clusters de calcul haute performance. L'extension que nous proposons présente cependant des aspects génériques sur l'ordonnancement des calculs ainsi que sur les transferts de données entre un accélérateur et son hôte et entre plusieurs accélérateurs dans un contexte distribué.

6.2 Programmation des GPGPUs

6.2.1 CUDA

CUDA *(Compute Unified Device Architecture)* [57, 76] est un modèle d'architecture et de programmation développé par NVIDIA pour la programmation de ses GPUs.

Modèle de programmation CUDA

Le modèle de programmation de CUDA repose sur la notion de *kernel*, qui est une fonction exécutable sur GPU. Un kernel est exécuté par un grand nombre de *threads*. Chaque thread possède un identifiant unique, permettant ainsi de différencier les données auxquelles le kernel est appliqué. Les threads exécutant un kernel sont regroupés par blocs de threads et les blocs sont organisés dans des grilles de blocs. À l'exécution, les threads sont ordonnancés par groupes, généralement de taille trente deux, appelés warps. Chaque *Streaming Multiprocessor* possède un ou plusieurs ordonnanceurs de warps et la caractéristique principale des GPUs pour cacher la latence des accès à la mémoire est l'ordonnancement de plusieurs warps simultanément. Ainsi, les temps d'accès mémoires se trouvent cachés par les exécutions concurrentes des threads de plusieurs warps.

Un GPU CUDA possède une mémoire propre, séparée de la mémoire de l'hôte. Le programmeur doit allouer les données utilisées par un kernel sur la mémoire du GPU. Il doit ensuite assurer la cohérence des données entre la mémoire de l'hôte et celle de l'accélérateur en insérant les transferts de données adéquats.

6.2.2 OpenCL

OpenCL [86] est un langage et une interface de programmation parallèle supportant à la fois le parallélisme de données et le parallélisme de tâches. OpenCL vise l'unification de la programmation des architectures parallèles hétérogènes. Un objectif important d'OpenCL est donc d'assurer la portabilité d'un code parallèle sur plusieurs architectures : les CPUs multi-cœurs, les GPGPUs de plusieurs constructeurs ainsi que d'autres types d'accélérateurs tels que le processeur Cell [96]. La contrepartie de la portabilité est une pénalité en performance avec OpenCL comparé aux modèles de programmation spécifiques tels que CUDA pour les GPUs de NVIDIA. Dans [39], Fang *et al.* avancent cependant que les différences de performance entre OpenCL et CUDA tendent à s'estomper. D'un point de vue programmabilité, OpenCL est très verbeux ; le programmeur doit spécifier une grande quantité d'information sur le programme, sa compilation et son contexte d'exécution.

6.3 Solutions pour l'exportation automatiques de calculs sur GPUs

Les modèles de programmation pour accélérateurs présentent des aspects répétitifs et fastidieux tels que la copie des données entre le hôte et l'accélérateur, le dimensionnement et le lancement des kernels, avec toutes les erreurs potentielles que le programmeur est susceptible d'introduire à chaque niveau. Des solutions ont été proposées, sous forme de *offload-compilers*, pour effectuer toutes ces tâches automatiquement.

6.3.1 par4all

Par4all [5] est un compilateur source-à-source développé par la société SILKAN [1]. Il extrait automatiquement le parallélisme contenu dans un programme séquentiel et permet, dans le contexte de la programmation pour accélérateurs, d'isoler les régions parallèles dans des kernels exécutables sur ces derniers. Le code initial est alors remplacé par des appels à ces kernels. L'allocation des données sur la mémoire de l'accélérateur ainsi que les transferts de données entre la mémoire de l'hôte et celle de l'accélérateur sont également générés automatiquement. Les transferts de données sont cependant effectués systématiquement avant et après toute invocation d'un kernel, indépendamment de l'utilisation future des données. Amini *et al.* [4] ont développé des optimisations interprocédurales de l'utilisation des données entre le hôte et l'accélérateur qui permettent de réduire significativement le nombre de communications. Guelton *et al.* [48] ont étendu l'utilisation des régions de tableaux pour les transferts de données à des nids de boucles avec des flots de contrôle plus complexes.

6.3.2 Approches utilisant le modèle polyédrique

Le compilateur PPCG (*Polyhedral Parallel Code Generator*) [10, 93] permet de générer du code CUDA à partir d'un code séquentiel contenant des parties de code à contrôle statique. PPCG permet d'extraire une représentation polyédrique à partir d'un code en entrée contenant des parties à contrôle statique pour lesquelles les noyaux CUDA sont générés. Les transferts mémoire entre le hôte et l'accélérateur sont également générés.

Baskaran et al.[13] proposent C-to-CUDA, un compilateur source-à source qui permet de transformer automatiquement un code en C contenant des boucles à controle affine en code CUDA.

Le compilateur R-Stream [62] propose une approche similaire, avec l'application de plusieurs transformations de boucles pour améliorer l'utilisation des mémoires caches et optimiser les accès à la mémoire globale du GPU. Les auteurs indiquent la possibilité d'étendre leur travail aux multi-GPUs comme travaux futurs.

1. http ://www.silkan.com

6.4 Pourquoi ne pas réutiliser un *offload-compiler*

Les *offload-compilers* présentés précédemment permettent d'exporter automatiquement les parties de codes à exécuter sur GPU tout en générant les transferts mémoire nécessaires. Cependant, ces solutions ne traitent pas encore de la génération de code distribué pour clusters de multi-GPUs.

Nous proposons de spécialiser notre schéma de compilation pour cibler la génération de code CUDA plutôt que de réutiliser la sortie du schéma de compilation présenté dans le chapitre 5 à un *offload-compiler* pour générer le code CUDA sur chaque nœud pour les raisons suivantes :

1. Nous disposons déjà de toutes les informations permettant d'ordonnancer correctement les calculs sur un GPU. Le compilateur *dSTEP* a obtenu ces informations par la directive *dstep gridify*. Un *offload-compiler* devrait redécouvrir toutes ces informations.
2. Le contrôle autour des nids de boucles est déjà présent dans le schéma de compilation. Il suffit pour exporter le calcul sur GPU de :
   - outliner le calcul d'une tranche d'itérations dans un noyau CUDA,
   - insérer un appel au noyau à la place du code exécutant la tranche d'itérations dans le domaine distribué.
3. Les communications ne sont plus à redécouvrir, elles découlent directement des communications du schéma de compilation pour CPU.

6.5 Contraintes

Le modèle de programmation *data parallel* des accélérateurs CUDA impose des contraintes sur un nid de boucles pour qu'il soit éligible à une génération de code sur GPU. En effet, les k niveaux les plus internes du nid de boucles doivent être parallèles, avec $k \geq 1$. Cette contrainte est due à la synchronisation implémentée sur les GPU. En effet, il est très coûteux de synchroniser les *threads* d'un bloc ainsi que les blocs entre eux. Il faut que les itérations des k niveaux les plus internes d'un kernel soient toutes indépendantes pour pouvoir bénéficier des capacités de calcul à parallélisme de données des GPUs. Pour les dimensions englobantes qui sont ordonnées, l'ordre d'exécution est assuré par les appels successifs aux kernels par le hôte.

6.6 Modèle d'exécution

Le calcul de chaque tranche d'itérations est exporté sur GPU. L'ordre d'exécution des itérations ordonnées englobant les k dimensions parallèles est assuré par l'ordre d'appels aux *kernels*. Lorsqu'un GPU produit des données à envoyer à un autre GPU, il passe par la mémoire de l'hôte, comme indiqué dans la figure 6.1 (1). L'hôte envoie ensuite ces données au processus hôte de l'autre GPU (2). Enfin, un transfert de l'hôte vers l'accélérateur se fait sur le nœud destinataire (3). Les communications entre les hôtes sont déjà générées par le compilateur *dSTEP* (voir le chapitre précédent). Les

communications hôte/accélérateur et accélérateur/hôte sont gérées de façon transparente pour le programmeur avec la même garantie sur la synchronisation que dans le cas sans GPU : toute donnée dans une communication en progression est certainement disponible au moment de son utilisation future sur un GPU.

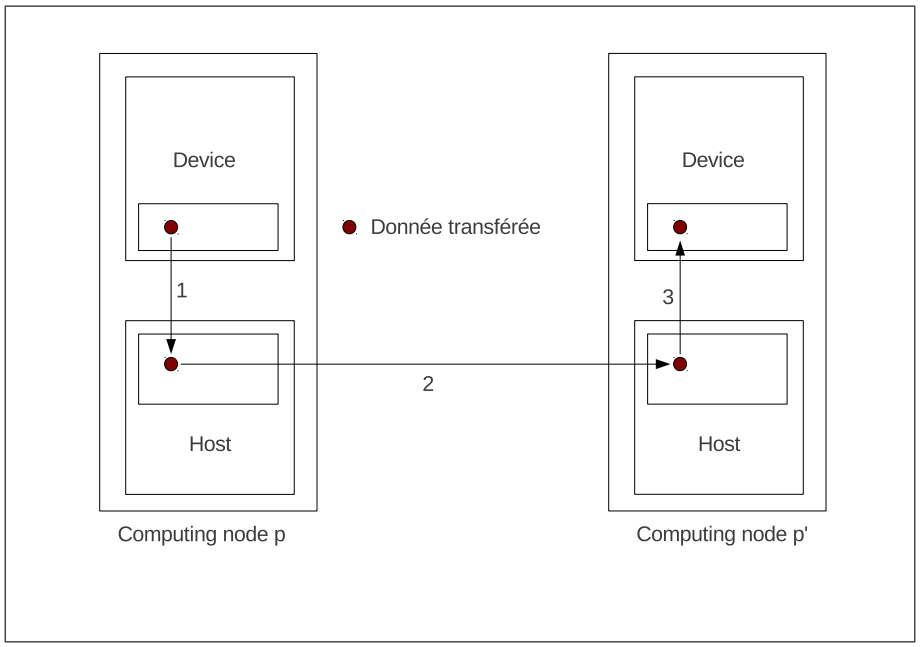


FIGURE 6.1 – Modèle de communication multi-GPU

6.7 Allocation des données sur GPU

L'allocation mémoire consiste à allouer sur GPU exactement les mêmes données que sur CPU pour chaque tableau distribué. L'algorithme 18 montre l'adaptation de l'algorithme d'allocation pour CPU (voir 1) pour générer l'allocation mémoire de chaque tableau distribué x sur GPU (ligne 9). La mémoire ainsi allouée est alors accessible par le pointeur X_gpu

6.8 Compilation d'une tranche d'itérations pour GPU

Le code vérifiant la bonne distribution est exactement le même que dans le cas CPU. La partie propre aux GPUs consiste à générer le kernel effectuant le calcul sur l'accélérateur et insérer les appels correspondants à ce kernel (algorithme 19). La fonction *project_last_dimensions* élimine les k dernières dimensions parallèles d'une tranche d'itérations. On récupère ainsi un ensemble d'itérations parcourus dans l'ordre

Algorithm 18 *generate_memory_allocation_gpu*

Entrée: $D, B, Hlow, Hup, \mathcal{C}, \mathcal{V}, type$ de base du tableau x
Sortie: remplacement de la déclaration de x par une déclaration distribuée

1: $generate_memory_allocation(X[D_{0,0}]...[D_{d-1,d-1}]) \equiv$
2: **for** each dimension k of array x **do**
3: **if** dimension k has a total halo **then** $L_{k,k} = D_{k,k}$
4: **else**
5: $L_{k,k} = Hlow_{k,k} + B_{k,k} + Hup_{k,k}$
6: **end if**
7: **end for**
8: $X[|\mathcal{C}|][|\mathcal{V}|][L_{0,0}]...[L_{d-1,d-1}]$
9: $X_{gpu} = allocate_gpu(|\mathcal{C}| \times |\mathcal{V}| \times L_{0,0}... \times L_{d-1,d-1} \times sizeof(type))$

imposé dans l'espace séquentiel sur le hôte. La fonction $project_first_dimensions$ permet de récupérer les itérations des dimensions parallèles qui seront utilisées pour créer la grille nécessaire au lancement du calcul sur GPU. La construction d'une grille sur GPU dépend ainsi de la tranche d'itérations *gpu_iterations* et des constantes $BLOCKS$ et $BLOCK$ qui donne respectivement la forme et le nombre de blocs pour chaque dimension de la grille et la forme et le nombre de threads pour chaque bloc. La tranche d'itérations est ensuite remplacée par une boucle sur les itérations non parallèles, appelant le kernel généré. On passe en arguments de cet appel les vecteurs $\vec{shift}_x$ permettant d'effectuer le changement de domaine.

Dans le code généré (présenté par l'algorithme 20) pour le kernel correspondant, on récupère d'abord l'itération locale, qui est l'identifiant du thread courant dans la grille. Ensuite, il faut calculer l'itération d'origine avec la fonction *loop_nest_iteration*. Comme le vecteur $\vec{i}_{kernel}$ peut avoir moins de composantes que le vecteur itération initial (à cause des éventuelles dimensions k' ordered consommées sur le hôte), on remplace ces composantes par des zéros. Puis, dans le vecteur résultat calculé $\vec{i''}$, on élimine les k' premières composantes. On combine ensuite les vecteurs $\vec{i'}$ et $\vec{i''}$ pour reconstruire le vecteur itération initial. Enfin, une garde est ajoutée pour s'assurer que l'itération recalculée ne sorte des bornes d'itérations initiales.

6.9 Communications entre hôte et accélérateur

Une donnée modifiée sur un GPU doit être propagée aux différentes copies allouées sur les autres GPU. Cette communication est réalisée en ajoutant deux appels de fonctions du support d'exécution (algorithme 21) : on insère, avant chaque *send* d'une donnée allouée sur GPU, un appel à la fonction *device_to_host* qui réalise le transfert de l'accélérateur à l'hôte au niveau du nœud produisant les données (ligne 1). On insère, après chaque complétion d'un *recv*, un appel à la fonction *host_to_device* qui transfère les données de l'hôte à l'accélérateur sur le nœud destinataire (lignes 8-11).

Mémoire unifiée À partir de la version 6.0, CUDA propose un mécanisme de mémoire unifiée entre le hôte et l'accélérateur. Une donnée allouée sur la mémoire de l'accélérateur peut être directement modifiée de façon transparente à la fois depuis le

Algorithm 19 *compile_iteration_slice*

Entrée: $body, p, \vec{\tilde{p}}_{ln}, \vec{c}_{ln}$
Sortie: le code d'une tranche d'itération compilé dans le domaine distribué pour GPU

1: $is = iteration_slice(\vec{\tilde{p}}_{ln}, \vec{c}_{ln})$
2: **for** each owner dimension k in loop nest ln **do**
3: $\quad extend(is, ln, k)$
4: **end for**
5: $\vec{i}_{low} = min(is)$
6: $\vec{i}_{up} = max(is)$
7: $computes = true$
8: $skips = true$
9: **for** each array x referenced in ln_stmt **do** ▷ compiler
10: $\quad \vec{p}_x = id_in_grid_x(p)$
11: $\quad accessed_x = read_region(x, body, is) \cup write_region(x, body, is)$
12: $\quad (\vec{c_x}, v_x) = belongs_to(\vec{p}_x, min(accessed_x), max(accessed_x))$
13: $\quad$ **if** $element_undefined((\vec{c_x}, v_x))$ **then**
14: $\quad\quad$ **if** $!owner$ **then**
15: $\quad\quad\quad$ *abort("bad distribution")*
16: $\quad\quad$ **else**
17: $\quad\quad\quad computes = false$
18: $\quad\quad$ **end if**
19: $\quad$ **else**
20: $\quad\quad \vec{\tilde{p}}_x = extend_id(\vec{p}_x, v_x)$
21: $\quad\quad \vec{shift}_x = array_element(\vec{\tilde{p}}_x, \vec{c}_x, \vec{0})$
22: $\quad\quad skips = false$
23: $\quad$ **end if**
24: **end for**
25: **if** $!(computes \oplus skips)$ **then**
26: $\quad$ *abort("bad distribution")*
27: **end if**
28: **if** $computes$ **then**
29: $\quad gpu_iterations = project_first_dimensions(is, k)$ ▷ compiler
30: $\quad gpu_grid = configure_gpu_grid(gpu_iterations, BLOCKS, BLOCK)$ ▷ compiler
31: $\quad cpu_iterations = project_last_dimensions(is, k)$ ▷ compiler
32: $\quad$ **for** $\vec{i'} = min_{<_{ln}}(cpu_iterations); \vec{i'} <_{ln} max_{<_{ln}}(cpu_iterations) + \vec{inc'}; \vec{i'} = \vec{i'} + \vec{inc'}$ **do**
33: $\quad\quad gpu_kernel_{ln}(gpu_grid, \vec{\tilde{p}}_{ln}, \vec{c}_{ln}, \vec{i'}, \vec{inc}, v_x, \vec{shift}_x, ...)$
34: $\quad$ **end for**
35: **end if**

Algorithm 20 gpu_kernel_{ln}

Entrée: $gpu_grid, \vec{\tilde{p}}_{ln}, \vec{c}_{ln}, \vec{i'}, \vec{inc}, v_x, \vec{shift}_x, ...$
Sortie: Ce kernel est le code généré sur GPU pour le nid de boucles ln

1: $\vec{i_{kernel}} = id_in_block_in_grid()$
2: $\vec{i''} = loop_nest_iteration(\vec{\tilde{p}}_{ln}, \vec{c}_{ln}, \begin{bmatrix} \vec{0} \\ \vec{i_{kernel}} \end{bmatrix})$
3: $\vec{i} = \begin{bmatrix} \vec{i'} \\ \vec{i''} \end{bmatrix}$
4: **if** $L_{ln} \leq \vec{i} < U_{ln} \wedge (\vec{i} - L_{ln}\vec{1}) mod_v \vec{inc} = \vec{0}$ **then**
5: $\quad body(..., X_{gpu}[\vec{c}_x][v_x][ref_x^m(\vec{i}) - \vec{shift}_x], ...)$
6: **end if**

Algorithm 21 Communications entre accélérateurs passant par les hôtes

1: $device_to_host(X_{gpu}[\vec{c}_x][v_x], X[\vec{c}_x][v_x], to_send, \vec{shift}_x)$
2: $async_send(X[\vec{c}_x][v_x], p', to_send, \vec{shift}_x)$
3: ...
4: **for** $(request, region) \in Pending(X)$ **do**
5: **if** $(accessed_x \cap region) \neq \emptyset$ **then**
6: $complete_comm(request)$
7: $Pending(X) = Pending(X) - \{(request, region)\}$
8: **if** $is_recv(request)$ **then**
9: $(\vec{c}_x, v_x, \vec{shift}_x) = get_local_information(request)$
10: $host_to_device(X[\vec{c}_x][v_x], X_{gpu}[\vec{c}_x][v_x], region, \vec{shift}_x)$
11: **end if**
12: **end if**
13: **end for**

code du hôte et depuis celui de l'accélérateur. Il n'est cependant pas clair dans quelle mesure un tel mécanisme affecterait les performances du code.

6.10 Les réductions

Toutes les réductions présentent la même caractéristique : appliquer un opérateur associatif à un ensemble de données et accumuler le résultat dans un scalaire. Il est donc opportun de fournir cette fonctionnalité comme fonctions de bibliothèque plutôt que de la générer de façon *ad hoc* pour chaque cas.

6.11 Conclusion

Nous avons montré dans ce chapitre l'adaptation du schéma de compilation de *dSTEP* pour cibler les multi-GPUs. Nous avons choisi une telle solution plutôt que la réutilisation d'un offload-compiler en raison de la présence de toutes les informations nécessaires à cette extension dans notre modèle de programmation. Les calculs sont exportés sur GPU pour les dimensions parallèles les plus internes d'un nid de boucles. Le code CPU réalise le contrôle de l'invocation de ces kernels pour les différentes tranches d'itérations ainsi que pour les dimensions ordonnées englobantes. Enfin, nous avons montré que les communications host/device découlaient directement des communications générées dans le schéma multi-CPU.

Chapitre 7

Modèle de coût

7.1 Introduction

Face à plusieurs distributions possibles, le programmeur doit être aidé dans le choix de la meilleure distribution en fonction des caractéristiques des programmes à paralléliser et de l'environnement d'exécution. On présente quelques modèles de coût existants pour répondre à ce besoin tout en montrant leurs limites. On propose ensuite un modèle de coût qui permet, non pas de trouver une solution optimale dans tous les cas, mais d'aider le programmeur à s'en approcher et, par ailleurs, de comprendre et d'anticiper les performances dans plusieurs configurations.

7.2 Quelques modèles de coût

Les premiers modèles de coût pour la programmation des machines à mémoire distribuée avaient pour but de choisir automatiquement une distribution de données.

Ainsi, Gupta [49] propose dans sa thèse un modèle de coût basé sur des contraintes pour décider la meilleure distribution qui minimise le temps d'exécution global du programme. Cette technique consiste à construire un graphe où les sommets sont les différentes dimensions des tableaux, reliés entre eux en cas d'alignement. Les arêtes sont valuées par des mesures de *qualité d'alignement* indiquant le coût des communications induites si les deux dimensions ne sont pas alignées. L'ensemble des sommets est ensuite partitionné de sorte à minimiser la somme des valuations des arêtes entre les sous-ensembles. Trouver une solution optimale à ce problème est prouvé NP-complet par Li et Chen [63]. De plus, en pratique, les compilateurs implémentant cette technique, comme PARADIGM [12], ne fournissent pas d'analyses inter-procédurales et ne peuvent donc pas détecter toutes les contraintes d'alignement.

Gracia *et al.* [46] proposent une solution de distribution automatique où le modèle de coût des calculs et des communications est capturé dans un graphe nommé *Communication Computation Graph*. Un solution optimale est recherchée en combinant deux objectifs : minimisation des communications et maximisation du parallélisme. Les auteurs appliquent cette solution pour décider la distribution d'une seule dimension de chaque tableau tout au long du programme. Ainsi, cette solution, même en apportant

une réponse optimale dans son cadre d'application, ne résout pas le problème dans le cas multi-dimensionnel avec des dimensions distribuées accédées alternativement par des boucles parallèles et non parallèles, cas par exemple des kernels ADI.

Lee *et al.* [59] présentent une approche basée sur les tableaux *dominants* et le tuilage. Un tableau dominant pour une partie de code est celui qui impose la plus grande pénalité de coût de communication. Le tuilage sert à absorber une partie du coût des communications en les exécutant en parallèle avec le traitement des différentes tuiles. Le modèle de coût sous-jacent décide une distribution pour un programme donné indépendamment des paramètres de l'environnement d'exécution (nombre de processus, caractéristiques du réseau, ...). Plus récemment, Hong *et al.* [54] ont proposé un modèle de coût pour les GPU qui prend en compte les temps des accès mémoire et les temps de calcul pour un seul GPU mais ce modèle ne traite pas le cas de plusieurs GPU en mémoire distribuée.

BSP est un modèle de calcul pour la programmation parallèle qui intègre directement dans sa définition un modèle de coût [91]. L'auteur affirme dans ce modèle que, pour avoir une exécution optimale de certains algorithmes parallèles, il faut tenir compte du ratio puissance de calcul / débit des communications de l'architecture sous-jacente.

7.3 Modèle de coût de *dSTEP*

On modélise le coût d'un nid de boucles dans le cas général : multi-dimensionnel et à ordonnancement quelconque. Le coût global d'une application est la somme des coûts de toutes les instances de ces nids de boucles ainsi modélisés auquel s'ajoute le coût des parties séquentielles exécutées en redondance par tous les processus. Le coût des communications est toujours pris en compte dans le modèle de coût même si une partie de ce coût peut être absorbée par le recouvrement calcul/communications implémenté dans *dSTEP* dans le cas où la technologie réseau utilisée l'implémente efficacement.

Le but de cette modélisation est d'aider le programmeur à choisir une distribution qui minimise le temps d'exécution d'une application dans une configuration particulière. Les paramètres à prendre en compte pour un nid de boucles sont :

- le nombre de processus,
- le ratio temps des communications / temps de calcul,
- le nombre de dimensions du nid de boucles,
- le volume de l'espace d'itérations : la taille des différentes dimensions du nid de boucles,
- les caractéristiques du réseau : la latence et la bande passante.

7.3.1 Hypothèses

1. Un nid de boucles possède au plus une seule dimension ordonnée et une seule dimension diagonalisée. Ces deux dimensions peuvent être confondues.

2. En présence de halos, on ne considère que les voisins selon la direction de ces halos. On néglige les autres voisins car les communications impliquées sont de volume moins important.

7.3.2 Temps d'exécution d'un nid de boucles distribué

Avec les hypothèses mentionnées ci-dessus, le temps d'exécution d'un nid de boucles de dimension d distribué s'écrit sous la forme :

$$t = t_{\text{calcul}} + t_{\text{communications}} + t_{\text{support d'exécution}} \tag{7.1}$$

Temps de calcul

Le temps de calcul est modélisé par la formule suivante :

$$t_{\text{calcul}} = steps \times \frac{u(\alpha, cycles)T}{\beta \prod_{i=0}^{d-1} P_{i,i}} \tag{7.2}$$

- T est le temps d'exécution séquentiel du nid de boucles. Ce temps de référence est défini pour une exécution sur un seul cœur CPU. Le temps de calcul est divisé entre les différents processus disponibles mais avec une pénalité en cas de déséquilibre de charge ;
- β est une constante matérielle qui correspond au ratio des performances théoriques entre un CPU multi-cœurs ou un GPU et un seul cœur CPU. Par exemple, pour un cluster de CPU quadri-cœurs homogènes, β vaut quatre. Pour un GPU de performance théorique crête de 1000 GFlops/s, le facteur β vaut 40 par rapport un cœur CPU avec une performance théorique maximale de 25 GFlops/s ;
- La matrice P représente la grille virtuelle de processus sur laquelle le nid de boucles est distribué ;
- *cycles* est le nombre de cycles du nid de boucles tel que défini au chapitre 4 intitulé *Modèle de distribution* ;
- La fonction u modélise le déséquilibre de charge. On se place dans le cas d'applications pour lesquelles la distribution cyclique tend à corriger ce déséquilibre. Des exemples de ces applications sont les algorithmes de factorisation tels que *LU* et *QR*. Choi *et al.* justifient l'utilisation d'une distribution bloc-cyclique pour corriger le déséquilibre de charge dans l'implémentation Scalapack de ces algorithmes [28]. Dans [71], les auteurs présentent une application de calcul de la qualité de l'air utilisant une distribution cyclique pour mieux équilibrer la charge. La figure 7.1 montre un exemple de calcul déséquilibré (représenté en bleu). Une distribution sans cycle (c'est-à-dire avec un seul cycle par dimension) du calcul sur quatre processus charge les processus $p0$, $p1$ et $p3$ de façon déséquilibrée alors que le processus $p2$ ne participe pas au calcul. Avec une distribution à deux cycles par dimension, on constate que la charge est mieux

répartie entre les différents processus. Cependant, le processus $p0$ reste moins chargé que les autres processus. L'ajout d'un cycle supplémentaire à chaque dimension permet d'équilibrer encore plus la charge. On admet donc que pour ce type d'applications, l'augmentation du nombre de cycles tend à corriger totalement le déséquilibre de charge ;

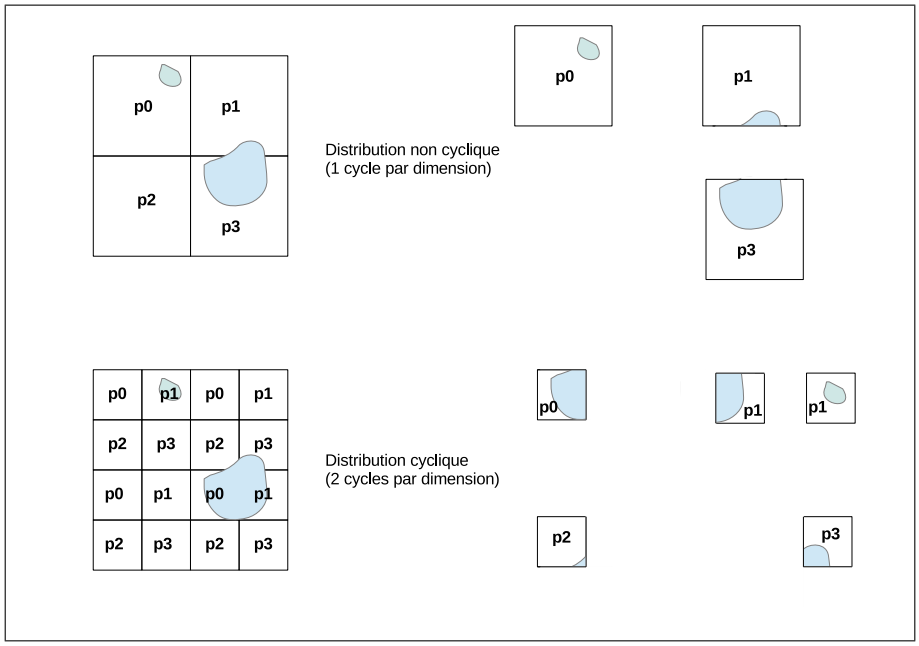


FIGURE 7.1 – Équilibrage de la charge de calcul avec la distribution cyclique.

- α est un réel qui représente la pénalité de déséquilibre de charge au niveau du processus qui a le plus de travail et qui conditionnera le temps d'exécution de tous les autres processus. La valeur de α varie entre 1 (aucun déséquilibre) et $\beta \prod_{i=0}^{d-1} P_{i,i}$ (déséquilibre total, où toute la charge de calcul est exécutée par un seul processus) ;
- La fonction u doit avoir les 2 caractéristiques suivantes :
  - $u(\alpha, 1) = \alpha$: si le nombre de cycles est égal à 1 (distribution sans cycles) alors le déséquilibre vaut α,
  - pour un nombre de cycles assez grand, le déséquilibre tend à être totalement corrigé :
$$\lim_{cycles \to +\infty} u(\alpha, cycles) = 1$$
- La fonction u peut être définie comme :
$$u(\alpha, cycles) = \frac{\alpha - 1}{cycles} + 1$$

  Dans la suite, on utilisera pour simplifier u pour représenter la valeur $u(\alpha, cycles)$;

– *steps* représente les différents pas de calculs exécutés. Ceci correspond à l'énumération des processus virtuels *vprocs*, défini au chapitre 4, pour une distribution multi-partitionnée ou bien à l'énumération des différents processus selon une dimension ordonnée.

Coût des communications

La taille des messages échangés est fonction de la taille des halos $Hlow_x$ et Hup_x et des tailles des blocs B_x pour chaque tableau x impliqué dans une communication pour le nid de boucles. La figure 7.2 représente les tailles des messages échangés pour une distribution bi-dimensionnelle d'un tableau à deux dimensions. Dans le modèle de coût, on ne prend en compte que les communications avec les voisins selon chaque dimension. On néglige les autres communications (représentées par des blocs transparents figurant aux quatre coins sur la figure).

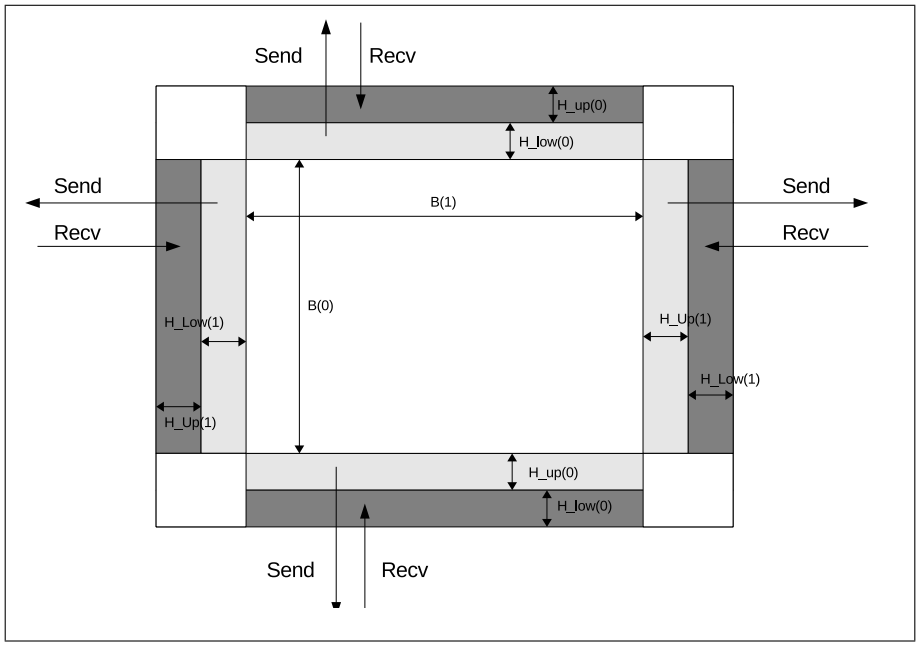


FIGURE 7.2 – Tailles des messages échangés pour un tableau écrit dans un nid de boucles

On suppose que la taille des halos inférieur et supérieur est identique pour chaque dimension, c'est-à-dire que $Hlow_x = Hup_x = H_x$. Le nombre d'éléments à communiquer d'un côté d'une dimension i est donné par la sur-approximation $N_{x_{i,i}}$:

$$N_{x_{i,i}} = H_{i,i} \times (\prod_{j=0, j \neq i}^{nb_dims_x} (2H_{j,j} + B_{j,j})) \tag{7.3}$$

Pour un tableau x, le nombre d'éléments communiqués est alors défini par l'équation 7.4. Le facteur 4 définit la contribution des communications des deux côtés de chaque dimension pour un *send* et un *recv*.

$$N_x = 4 \times \sum_{i=0}^{nb_dims_x} N_{x_{i,i}} \tag{7.4}$$

On suppose par ailleurs que la latence du réseau est constante et vaut L pour chaque message échangé. La latence engendrée par les communications de chaque tableau x impliqué dans des communications d'une tranche d'itérations est alors définie par l'équation 7.5.

$$L_x = 4 \times \sum_{i=0}^{nb_dims_x} L = 4 \times nb_dims_x \times L \tag{7.5}$$

Remarque. Le symbole L utilisé ici pour la latence n'a pas le même que le symbole du même nom utilisé dans le modèle de distribution.

Enfin, les communications sont effectuées pour chaque tranche d'itérations du nid de boucles, *i.e.* pour chaque processus virtuel et chaque cycle. Th est la bande passante du réseau. Le temps des communications est défini à l'équation 7.6.

$$t_{\text{communications}} = vprocs \times cycles \times \sum_x (\frac{N_x}{Th} + L_x) \tag{7.6}$$

Surcoût du support d'exécution

On suppose que le surcoût du support d'exécution est constant, représenté par la constante R.

Temps global d'exécution d'un nid de boucles distribué

On définit maintenant le temps d'exécution d'un nid de boucles de dimension d distribué sur P processus par l'équation 7.7.

$$t = steps \times \frac{u(\alpha, cycles)T}{\beta \prod_{i=0}^{d-1} P_{i,i}} + vprocs \times cycles \times \sum_x (\frac{N_x}{Th} + L_x) + R \tag{7.7}$$

7.4 Utilisation du modèle de coût

Le modèle de coût est utilisé pour choisir une distribution parmi plusieurs distributions possibles en fonction des caractéristiques de l'environnement d'exécution. Un exemple d'une telle situation peut être un programme avec des nids de boucles parallèles accédant à des tableaux à deux dimensions. Dans ce cas, on pourrait distribuer la première dimension des tableaux et des nids de boucles, la deuxième ou bien les deux à la fois. La quantification du coût des messages échangés avec le modèle de coût

permet de choisir la meilleure distribution. Un autre exemple est un programme qui requiert une distribution multi-partitionnée afin de faire travailler tous les processus en présence d'une dimension ordonnée du nid de boucles. Dans une telle situation, une distribution non multi-partitionnée, que l'on appellera *distribution standard* est toujours possible. Dans ce cas, à chaque fois qu'une dimension est ordonnée, tous les processus ne travaillent pas simultanément, mais on se retrouve avec moins de communications en comparaison avec une distribution standard. On se pose donc la question de savoir quelle distribution choisir en fonction de plusieurs paramètres : la taille des données, le nombre de processus, la vitesse du réseau et la puissance de calcul. Dans les sections suivantes, on quantifie cette comparaison en utilisant le modèle de coût. Le terme R s'annule toujours dans les différences. Il ne figure pas dans la suite.

7.5 Comparaison entre les distributions multi-partitionnée et standard

Hypothèse. On suppose dans ce cas que les tableaux référencés en écriture ont la même distribution que le nid de boucles.

7.5.1 Nid de boucles parallèle

Pour la distribution multi-partitionnée, *steps* est le nombre de processus virtuels *vprocs*. Pour la distribution standard, il y a un seul *vproc*, et une seule étape de calcul ($vprocs = steps = 1$).

On calcule $\Delta_{parallel}$, la différence de temps d'exécution entre la distribution multi-partitionnée et la distribution standard en utilisant la formule 7.7. La distribution standard est meilleure lorsque $\Delta_{parallel}$ est positif.

$$\Delta_{parallel} = t_{multi-partitioned} - t_{standard} \tag{7.8}$$

$$\begin{aligned}\Delta_{parallel} = vprocs \times \frac{u(\alpha, cycles)T}{\beta \prod_{i=0}^{d-1} P_{i,i}} + vprocs \times cycles \times \sum_x (\frac{N_x}{Th} + L_x) - \\ \left(1 \times \frac{u(\alpha, cycles)T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + 1 \times cycles \times \sum_x (\frac{N'_x}{Th} + L_x)\right)\end{aligned} \tag{7.9}$$

On a, par définition des grilles virtuelles de processus la relation suivante entre P et P' :

$$\prod_{i=0}^{d-1} P_{i,i} = vprocs \times \prod_{i=0}^{d-1} P'_{i,i} \tag{7.10}$$

Conformément au modèle de distribution, pour tout tableau x impliqué dans une communication, on a la relation 7.11 entre les tailles de blocs B_x de la distribution multi-partitionnée et B'_x de la distribution standard.

$$det(B_x) = f \times det(B'_x), 1 \leq f \leq vprocs \tag{7.11}$$

Les tailles des messages échangés étant proportionnelles aux tailles de blocs pour les mêmes valeurs de halos (voir l'équation 7.3), on en déduit la relation 7.12 entre les tailles des messages de la distribution multi-partitionnée et de la distribution standard.

$$N_x = f \times N'_x, 1 \leq f \leq vprocs \tag{7.12}$$

On peut réécrire l'équation 7.9 en :

$$\begin{aligned}\Delta_{parallel} = vprocs \times \frac{u(\alpha,cycles)T}{\beta \times vprocs \prod_{i=0}^{d-1} P'_{i,i}} + vprocs \times cycles \times \sum_x (\frac{N_x}{Th} + L_x) - \\ \left(\frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + cycles \times \sum_x (\frac{f \times N_x}{Th} + L_x) \right)\end{aligned} \tag{7.13}$$

Ce qui se simplifie en :

$$\begin{aligned}\Delta_{parallel} = \frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + vprocs \times cycles \times \sum_x (\frac{N_x}{Th} + L_x) - \\ \left(\frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + cycles \times f \times \sum_x (\frac{N_x}{Th} + L_x) \right)\end{aligned} \tag{7.14}$$

Ce qui se simplifie en :

$$\begin{aligned}\Delta_{parallel} = vprocs \times cycles \times \sum_x (\frac{N_x}{Th} + L_x) - \\ cycles \times f \times \sum_x (\frac{N_x}{Th} + L_x)\end{aligned} \tag{7.15}$$

Ce qui se simplifie enfin en :

$$\Delta_{parallel} = (vprocs - f) \times cycles \times \sum_x (\frac{N_x}{Th} + L_x) \tag{7.16}$$

Conclusion. D'après l'inéquation 7.12, $(vprocs - f) \geq 0$. Il en résulte que $\Delta_{parallel}$ est toujours positif ou nul. On en conclut donc que la distribution multi-partitionnée n'apporte aucun bénéfice pour un nid de boucles parallèle. Elle peut au contraire être moins performante que la distribution standard ($f < vprocs$).

7.5.2 Nid de boucles ordonné

On calcule $\Delta_{ordered}$, la différence de temps d'exécution entre la distribution multi-partitionnée et la distribution standard en utilisant la formule 7.7 pour un nid de boucles avec une seule dimension ordonnée k parmi d dimensions. La distribution standard est meilleure lorsque $\Delta_{ordered}$ est positif.

$$\Delta_{ordered} = t_{multi-partitioned} - t_{standard} \tag{7.17}$$

$$\Delta_{ordered} = vprocs \times \frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P_{i,i}} + vprocs \times cycles \times \sum_x (\frac{N_x}{Th} + L_x) - \left(P'_{k,k} \times \frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + 1 \times cycles \times \sum_x (\frac{N'_x}{Th} + L_x) \right) \tag{7.18}$$

$$\Delta_{ordered} = vprocs \times \frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P_{i,i}} + vprocs \times cycles \times (\sum_x \frac{N_x}{Th} + \sum_x L_x) - \left(P'_{k,k} \times \frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + cycles \times (\sum_x \frac{N'_x}{Th} + \sum_x L_x) \right) \tag{7.19}$$

$$\Delta_{ordered} = \frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + vprocs \times cycles \times (\sum_x \frac{N_x}{Th} + \sum_x L_x) - \left(P'_{k,k} \times \frac{u(\alpha,cycles)T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + cycles \times (\sum_x \frac{N'_x}{Th} + \sum_x L_x) \right) \tag{7.20}$$

7.5.3 Simplification de $\Delta_{ordered}$

On suppose que la charge de calcul initiale est uniforme et que donc la distribution n'est pas cyclique. On a $cycles = 1$ et $u = 1$. Sous cette hypothèse, on a $f = vprocs$ (voir l'équation 7.12).

$$\Delta_{ordered} = \frac{T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + vprocs \times (\sum_x \frac{N_x}{Th} + \sum_x L_x) - \left(P'_{k,k} \times \frac{T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + (vprocs \sum_x \frac{N_x}{Th} + \sum_x L_x) \right) \tag{7.21}$$

$$\Delta_{ordered} = \frac{T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + vprocs \times \sum_x L_x - \left(P'_{k,k} \times \frac{T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + \sum_x L_x \right) \tag{7.22}$$

$$\Delta_{ordered} = (1 - P'_{k,k}) \frac{T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + (vprocs - 1) \times \sum_x L_x \tag{7.23}$$

La distribution standard devient plus performante lorsque $\Delta_{ordered}$ est positif.

$$\Delta_{ordered} \geq 0 \implies (1 - P'_{k,k}) \frac{T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} + (vprocs - 1) \sum_x L_x \geq 0 \tag{7.24}$$

$$\implies (P'_{k,k} - 1) \frac{T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} - (vprocs - 1) \sum_x L_x \leq 0 \tag{7.25}$$

$$\implies (P'_{k,k} - 1) \frac{T}{\beta \prod_{i=0}^{d-1} P'_{i,i}} \leq (vprocs - 1) \sum_x L_x \tag{7.26}$$

Pour toute distribution sur un nombre de processus supérieur ou égal à deux, on a $vprocs > 1$. On a donc :

$$\Delta_{ordered} \geq 0 \implies \frac{(P'_{k,k} - 1)}{(vprocs - 1)} \frac{T}{\prod_{i=0}^{d-1} P'_{i,i}} \leq \beta \sum_x L_x \tag{7.27}$$

$$\implies \frac{(P'_{k,k} - 1)}{(vprocs - 1)} \frac{T}{\prod_{i=0}^{d-1} P'_{i,i}} \leq 4\beta L \sum_x nb_dims_x \tag{7.28}$$

Conclusions.

1. Le membre gauche de l'inéquation est une constante matérielle pour un environnement donné multiplié par la somme du nombre de dimensions de chaque tableau impliqué dans une communication.
2. L'augmentation du nombre de processus, représenté par le produit $\prod_{i=0}^{d-1} P'_{i,i}$ fait diminuer le membre droit de l'inéquation et laisse supposer l'existence d'un nombre de processus au delà duquel l'inéquation est vérifiée, et la distribution standard devenant par conséquent meilleure que la distribution multi-partitionnée.
3. Ce point de basculement arrive d'autant plus rapidement que l'accélération brute de chaque nœud de calcul, β, et la la latence du réseau, L, sont élevées. Une telle configuration peut être matérialisée par un cluster de GPUs connectés par un réseau *Ethernet*.

7.6 Conclusion

Face à plusieurs distributions possibles des données et des calculs d'un même programme, le programmeur peut orienter son choix en fonction de la nature du problème traité (parallélisme, taille des données) et de l'environnement d'exécution parallèle. Nous avons proposé un modèle de coût propre à *dSTEP* afin de quantifier les coûts de plusieurs types de distributions pour ainsi guider le programmeur à choisir une distribution de façon plus précise que la simple intuition. De plus, le modèle de coût proposé montre que le choix d'une distribution dans une configuration matérielle donnée n'est pas absolu et que des points de basculement entre des distributions différentes peuvent se produire au delà d'un certain nombre de processus. Enfin, le modèle de coût peut également être utilisé pour prédire ou expliquer les performances obtenues dans différentes configurations. Les conclusions de l'application du modèle de coût sont des prédictions théoriques. Pour valider ces résultats, il faut les confronter aux véritables temps d'exécution de programmes présentant plusieurs possibilités de distribution. Nous envisageons d'effectuer des expérimentations pour valider le modèle de coût de *dSTEP* dans un avenir proche.

Chapitre 8

Résultats expérimentaux

8.1 Introduction

Nous présentons dans ce chapitre la mise en œuvre de la distribution des données et des calculs sur des programmes représentatifs des applications cibles du modèle de programmation de *dSTEP*, définies au chapitre 3. Nous mesurons les performances du code généré pour les programmes suivants :

- Matmul : multiplication de matrices ;
- le solveur Jacobi, qui présente un schéma d'accès avec un *stencil* à quatre points ;
- Polybench Adi, un programme qui contient des nids de boucles avec des dimensions alternativement parallèles et ordonnées ;
- NAS BT, une application des *NAS Parallel Benchmarks* qui résout l'équation de *Navier-Stokes* en mécanique des fluides ;
- LDPC, une application industrielle de communication radio.

Pour l'application NAS BT, notre solution est comparée au code de référence écrit en Fortran MPI et au code écrit dans le langage UPC.

8.2 Environnement de tests

Nous avons utilisé la plateforme de calcul Grid'5000 [17], notamment le cluster Edel à Grenoble pour réaliser la plus grande partie des tests présentés dans cette section. Les caractéristiques techniques du cluster Edel sont indiquées dans la figure 8.1. Sauf mention spéciale, nous utilisons le réseau rapide Infiniband.

Remerciements. *Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations*

Edel

Model:	Bullx Blade B500 **72 nodes**
CPU:	Intel Xeon E5520 2.27 GHz • 72 nodes x 2 cpus per node = **144 cpus** • 144 cpus x 4 core(s) per cpu = **576 cores**
Memory:	24 GB
Network:	• 1 Gigabit Ethernet • 1 cards **InfiniBand 40G** (Mellanox ConnectX IB 4X QDR MT26428) Driver:
Storage:	64 GB / SATA/SSD Driver:
Misc:	
Date of arrival:	2009/Q3

FIGURE 8.1 – Caractéristiques du cluster Edel.

8.3 Multiplication de matrices

Nous travaillons sur un programme appelé Matmul qui effectue une multiplication de matrices de la forme $C := C + A \times B$. Ce calcul est précédé d'une phase d'initialisation des matrices A, B et C et suivi de l'affichage de la trace de la matrice C. Cette organisation du programme Matmul reflète l'insertion de la multiplication de matrices dans les programmes de calcul scientifique avec une définition des matrices A, B et C avant le calcul et la réutilisation de la matrice C par la suite. Elle permet notamment de mettre en évidence la génération des communications.

Le listing 8.1 montre le code du programme Matmul. La fonction *matrix_multiply* implémente la version naïve de la multiplication de matrices. On montre en rouge les directives *dSTEP* ajoutées à ce programme afin de le paralléliser et de distribuer ses données.

8.3.1 Insertion des directives

Distribution des données. Nous distribuons la matrice résultat C sur ses deux dimensions. Comme les accès aux dimensions de la matrice C ne présente pas de translation en fonction des indices de boucles i et j, il n'est pas nécessaire de définir des halos pour la matrice C. Nous distribuons également les matrices A et B par blocs sur leurs deux dimensions. Cependant, pour calculer un coefficient $C_{i,j}$, il faut lire toute la ligne i de la matrice A et toute la colonne j de la matrice B. Nous avons donc besoin d'un halo sur la deuxième dimension de la matrice A qui couvre toute la

```
1
2  void matrix_init(int m, double A[m][m], double val)
3  {
4    int i, j;
5   #pragma step gridify(i, j)
6    for (i=0; i<m; i++)
7      for (j=0; j<m; j++)
8        A[i][j] = (i * val) / ((i+j) * 1000);
9  }
10
11 void matrix_multiply(int m, double C[m][m], double A[m][m], double B[m][m]){
12   int i, j, k;
13  #pragma step gridify(i, j, k(dist=all(K_BLOCK_SIZE); sched=ordered))
14   for (i=0; i<M; i++)
15     for (j=0; j<M; j++){
16       double c = C[i][j];
17       for (k=0; k<M; k++)
18         c += A[i][k] * B[k][j];
19       C[i][j] = c;
20     }
21 }
22
23 double matrix_trace(int m, double C[m][m])
24 {
25   int i, j;
26   double trace = 0;
27
28  #pragma step gridify(i, j(dist=*; sched=owner)) reduction(+:trace)
29   for (i=0; i<M; i++)
30     for (j=0; j<=0; j++)
31       trace += C[i][i];
32   return trace;
33 }
34
35 int main(void)
36 {
37   unsigned int i,j,k;
38   double trace = 0.0;
39
40  #pragma step distribute A(block, _H_:block:_H_)
41   double A[M][M];
42  #pragma step distribute B(_H_:block:_H_, block)
43   double B[M][M];
44  #pragma step distribute C(block, block)
45   double C[M][M];
46
47   matrix_init(M, A, v1);
48   matrix_init(M, B, v2);
49   matrix_init(M, C, v3);
50
51   matrix_multiply(M, C, A, B);
52
53   printf("%g\n", matrix_trace(m, C));
54
55   return 0;
56 }
```

Listing 8.1 – Le programme Matmul

dimension. Nous utilisons pour cela un halo intégral représenté par la constante $_H_$. Il en est de même pour la matrice B mais sur sa première dimension. La figure 8.2 montre les données allouées sur le processus p_5 ($p_{1,1}$ dans la grille à deux dimensions) pour les matrices A, B et C dans le cas d'une exécution sur 16 processus.

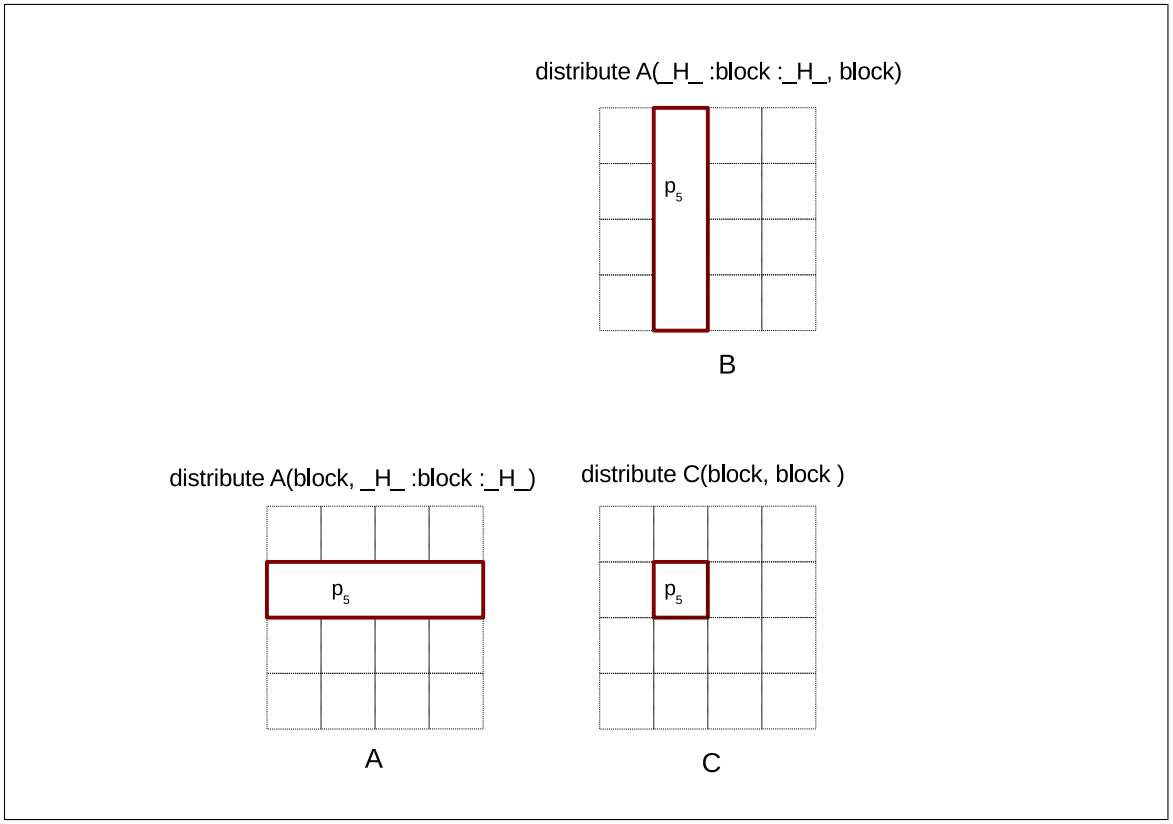


FIGURE 8.2 – Distribution des matrices A, B et C

Nous rappelons ici que la distribution des deux dimensions d'une matrice par blocs avec halo total sur l'une de ses dimensions est différente de la distribution de l'une de ses dimensions et de la réplication de l'autre comme le montre la figure 8.3 pour la matrice A sur le processus p_5.

Distribution des calculs La fonction *init_matrix* contient un nid de boucles parallèle sur ses deux dimensions. Nous utilisons alors la directive `step gridify` pour distribuer ses deux dimensions par blocs avec un ordonnancement parallèle.

La fonction *matrix_multiply* contient un nid de boucles parallèle sur ses trois dimensions i, j et k. Nous distribuons les dimensions i et j par blocs avec un ordonnancement parallèle.

Nous distribuons la dimension k avec une distribution de type *all* avec une taille de bloc K_BLOCK_SIZE dont nous pouvons contrôler la valeur. La distribution *all* ne consomme pas de processus dans la grille pour la dimension k et affecte tous les blocs de taille K_BLOCK_SIZE à chaque processus. L'ordonnancement de cette dimension est également parallèle et indiqué explicitement dans ce cas. L'idée derrière ce type de distribution est de générer plusieurs blocs pour la boucle la plus interne k afin de gagner en localité sur chaque nœud grâce à la réutilisation des données de chaque bloc.

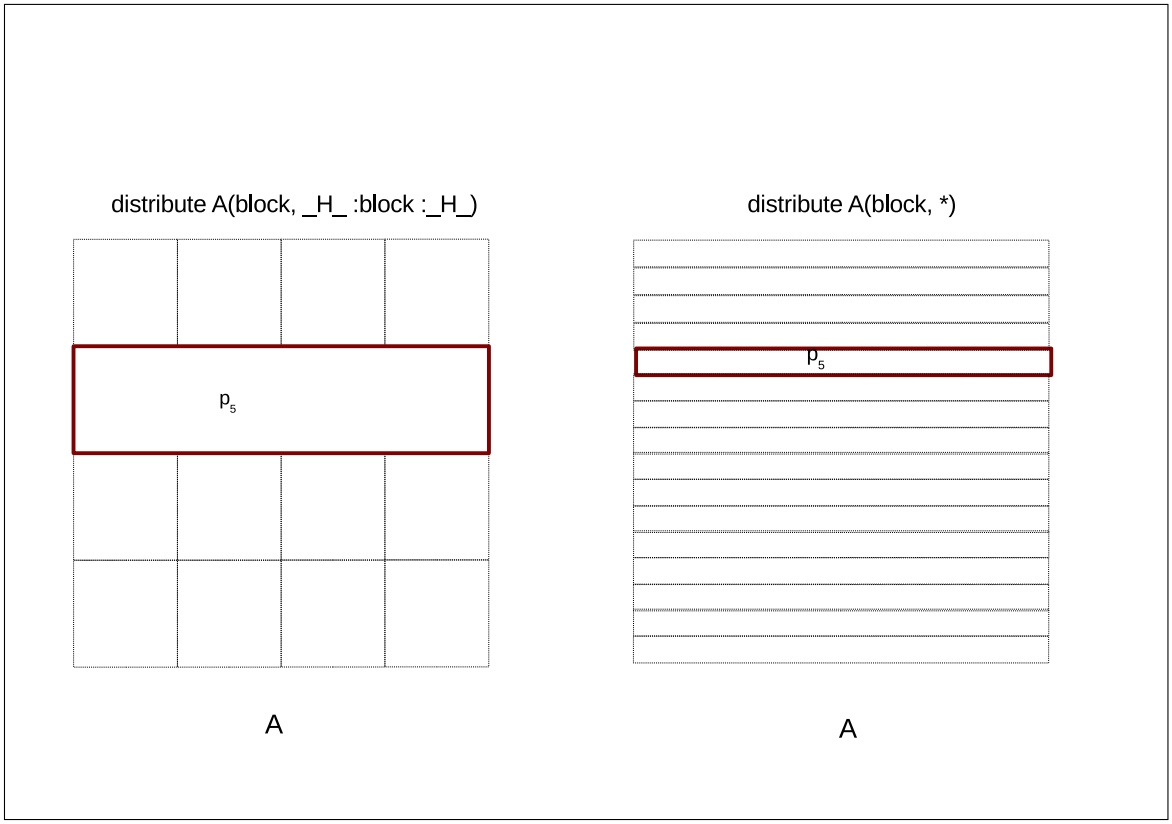


FIGURE 8.3 – Comparaison des distributions répliquée et avec halo total pour la deuxième dimension de la matrice A

La fonction *matrix_trace* calcule la trace de la matrice C. Comme il s'agit d'une réduction sur les éléments de la diagonale de cette matrice, nous pourrions écrire ce code simplement comme dans le listing 8.2, et distribuer la dimension parallèle i par blocs.

```
1 double matrix_trace(int m, double C[m][m])
2 {
3   int i;
4   double trace = 0;
5  #pragma step gridify(i) reduction(+:trace)
6   for (i=0; i<M; i++)
7       trace += C[i][i];
8   return trace;
9 }
```

Listing 8.2 – Distribution possible de la boucle de réduction

Cependant, cette distribution affecterait tous les processus disponibles à la première dimension de la boucle et générerait des accès non satisfaits pour la matrice C dont la distribution est sans halo. Pour y remédier, il suffit d'introduire une

boucle j, qui effectue une seule itération. La dimension j est alors distribuée de façon répliquée avec un ordonnancement *owner*. Il résulte de cette distribution que les deux dimensions consomment les processus disponibles à l'exécution, ce qui garantit les accès pour la matrice C sans halos. De plus, l'ordonnancement *owner* n'active le calcul que sur les processeurs possédant les éléments de la diagonale. Enfin, nous ajoutons la clause de réduction afin d'indiquer au compilateur *dSTEP* qu'il doit générer une réduction distribuée de la variable *trace* (lignes 23-33 du listing 8.1)

8.3.2 Code généré

Le listing 8.3 montre un extrait de code généré pour la fonction *matrix_multiply*. Les lignes 5 et 6 montrent le calcul des nouvelles bornes des boucles i et j. Les lignes 8 à 12 montrent le calcul du bloc et du cycle accédé pour la matrice C ainsi que le typage dynamique du pointeur sur C avec les bonnes dimensions. Ensuite le nombre de blocs de la boucle k est calculé (ligne 14) et parcouru par la boucle *_bk* (ligne 16). Pour chaque itération de la boucle *_bk*, les nouvelles bornes d'itérations de la boucle k sont calculées, ainsi que les blocs et cycles accédés pour les matrices A et B dont les pointeurs sont typés dynamiquement (lignes 18-27). Enfin, pour tout processus pour lequel le calcul est activé, le calcul est effectué avec les nouvelles bornes d'itérations des boucles i, j et k. Pour ce calcul, tous les processus sont actifs. Nous remarquerons le changement de repère sur les 2 dimensions distribuées de la matrice C. En revanche, pour la matrice A, le changement de repère est effectué uniquement sur la première dimension. La deuxième dimension de la matrice A possède un halo total et l'accès n'est donc pas traduit dans le repère local. La même remarque vaut pour la matrice B avec un changement de repère uniquement pour la deuxième dimension.

8.3.3 Évaluation des performances

Nous mesurons les temps d'exécution du code généré par le compilateur *dSTEP* du programme Matmul pour des matrices carrées de taille 2048 et 8192 en activant autant de threads OpenMP que de cœurs sur chaque nœud. Le temps de référence est le temps d'exécution du programme original avec un seul thread OpenMP sur un seul cœur de calcul. Pour le cas N=2048 (fig. 8.4), nous avons testé plusieurs tailles de bloc pour la dimension *all* et comparé avec la distribution sans clause *all* de la dimension k. Nous obtenons de meilleures performances avec la distribution *all* et pour une taille de bloc égale à 16. Nous observons cependant que les temps d'exécution ne diminuent plus significativement au delà de 128 cœurs car le ratio communications/calcul devient de plus en plus important avec le nombre de cœurs. Pour le cas N=8192 (fig. 8.5), le ratio communications/calcul devient plus favorable et nous observons un meilleur passage à l'échelle. Nous obtenons pour ce cas une accélération très élevé, 1632, pour la distribution *all* avec une taille de bloc de 16 sur de 256 cœurs grâce à une amélioration de la localité temporelle de la boucle k.

```
1
2 void matrix_multiply_GRIDIFY_HYBRID(void *A, void *B, void *C) {
3   ...
4
5   STEP_LOOP_BOUNDS(2, _LOW, _UP, _B, _P, _c, _t_p, _L, _U, _INCR);
6   int STEP_i_LOW = _LOW[0], STEP_i_UP = _UP[0], STEP_j_LOW = _LOW[1], STEP_j_UP = _UP
      [1];
7
8   int _ACCESSED_C[2][2] = {{STEP_GENERIC_MAX(2, STEP_i_LOW, 0), STEP_GENERIC_MIN(2,
      STEP_i_UP, m-1)},
9                            {STEP_GENERIC_MAX(2, STEP_j_LOW, 0), STEP_GENERIC_MIN(2,
      STEP_j_UP, m-1)}};
10  STEP_BELONGS_TO(C, _DIMS_C, &_c_C, &_b_C, _c_VECT_C, _LOW_C, _ACCESSED_C, 0, &
      _computes);
11  double (*C__)[_NB_B][_DIMS_C[0]][_DIMS_C[1]] = (double (*)[_NB_B][_DIMS_C[0]][
      _DIMS_C[1]]) C;
12  double (*C)[_NB_B][_DIMS_C[0]][_DIMS_C[1]] = C__;
13
14  int _NB_B_k = STEP_INT_DIV(m, K_BLOCK_SIZE, 1);
15
16  for(_bk = 0; _bk < _NB_B_k; _bk++) {
17    int STEP_k_LOW = _bk * ksize, STEP_k_UP = STEP_k_LOW + ksize -1;
18    int _ACCESSED_A[2][2] = {{STEP_GENERIC_MAX(2, 0, STEP_i_LOW), STEP_GENERIC_MIN(2,
       m-1, STEP_i_UP)},
19      {STEP_GENERIC_MAX(2, 0, STEP_k_LOW), STEP_GENERIC_MIN(2, m-1, STEP_k_UP)}};
20    STEP_BELONGS_TO(A, _DIMS_A, &_c_A, &_b_A, _c_VECT_A, _LOW_A, _ACCESSED_A, 0, &
      _computes, &_skips);
21    int _ACCESSED_B[2][2] = {{STEP_GENERIC_MAX(2, 0, STEP_k_LOW), STEP_GENERIC_MIN(2,
       m-1, STEP_k_UP)},
22      {STEP_GENERIC_MAX(2, 0, STEP_j_LOW), STEP_GENERIC_MIN(2, m-1, STEP_j_UP)}};
23    STEP_BELONGS_TO(B, _DIMS_B, &_c_B, &_b_B, _c_VECT_B, _LOW_B, _ACCESSED_B, 0, &
      _computes, &_skips);
24    double (*A__)[_NB_B][_DIMS_A[0]][_DIMS_A[1]] = (double (*)[_NB_B][_DIMS_A[0]][
      _DIMS_A[1]]) A;
25    double (*A)[_NB_B][_DIMS_A[0]][_DIMS_A[1]] = A__;
26    double (*B__)[_NB_B][_DIMS_B[0]][_DIMS_B[1]] = (double (*)[_NB_B][_DIMS_B[0]][
      _DIMS_B[1]]) B;
27    double (*B)[_NB_B][_DIMS_B[0]][_DIMS_B[1]] = B__;
28    if (_computes) {
29 #pragma omp parallel for private(i, j, k)
30      for(i = STEP_i_LOW; i <= STEP_i_UP; i += 1) {
31        for(j = STEP_j_LOW; j <= STEP_j_UP; j += 1) {
32          double c = C[_c_C][_b_C][i-_LOW_C[0]][j-_LOW_C[1]];
33          for(k = STEP_k_LOW; k <= STEP_k_UP; k += 1)
34            c += A[_c_A][_b_A][i-_LOW_A[0]][k] * B[_c_B][_b_B][k][j-_LOW_B[1]];
35          C[_c_C][_b_C][i-_LOW_C[0]][j-_LOW_C[1]] = c;
36        }
37      }
38    }
39  }
40 }
```

Listing 8.3 – Extrait du code généré pour la fonction matrix_multiply

8.4 Jacobi

Le code du solveur Jacobi est constitué, outre l'initialisation et l'affichage final, d'un noyau de calcul avec un motif d'accès qui présente un stencil à quatre points et un calcul de la norme. Ce code utilise deux matrices dont le contenu est échangé à chaque pas de calcul. Le code généré présente donc des communications sur les bords des matrices ainsi qu'une réduction parallèle pour calculer la norme.

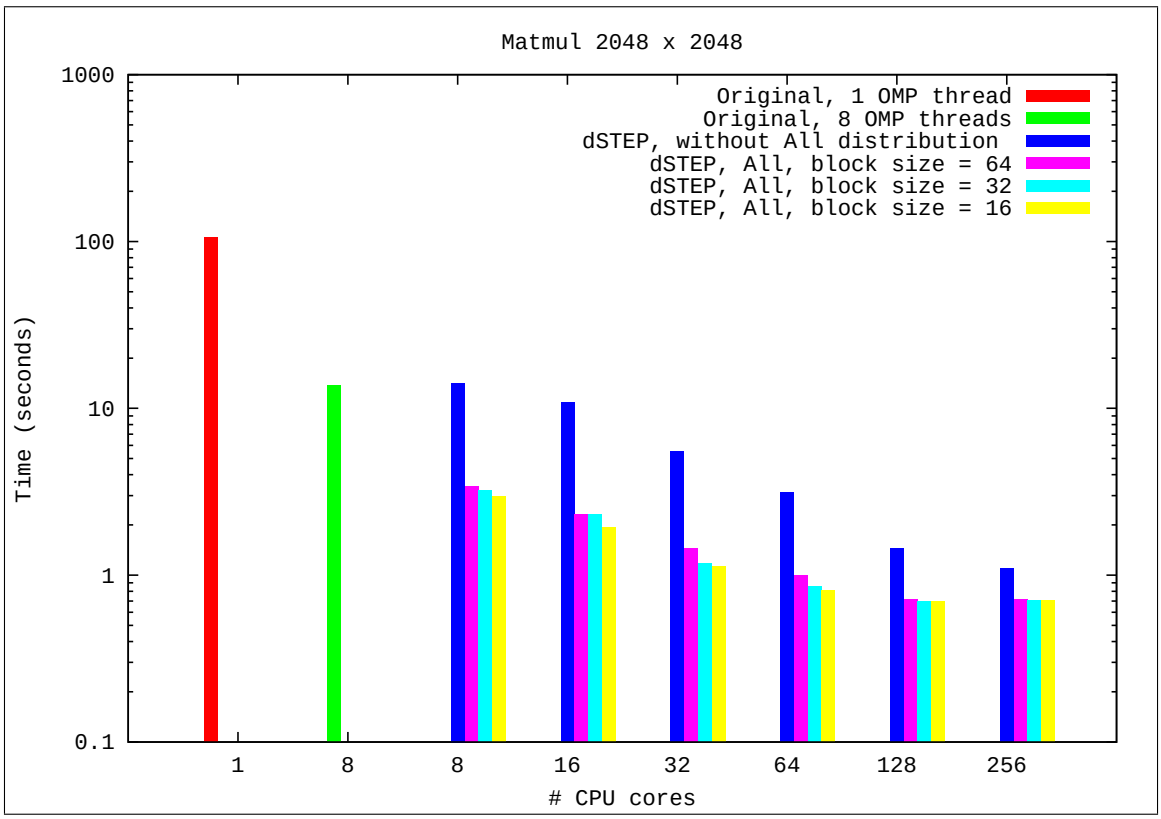


FIGURE 8.4 – Temps d'exécution de Matmul, N = 2048

8.4.1 Insertion des directives

Nous distribuons avec la directive *step distribute* les deux matrices du programme par blocs sur leurs deux dimensions avec des halos pour chaque dimension afin de satisfaire les accès en stencil du code généré. Les nids de boucles sont distribués avec la directive *step gridify* sur leurs deux dimensions, avec l'insertion d'une clause de réduction pour le calcul de la norme.

8.4.2 Évaluation des performances

Nous mesurons les temps d'exécution du code généré pour deux tailles de matrices de taille N×N. Pour le cas N=1026 (fig. 8.6), on observe une stagnation du temps d'exécution à partir de 128 cœurs. Pour le cas N=8196 (fig. 8.7), nous obtenons un meilleur passage à l'échelle du code généré, avec une accélération de 192 sur 255 cœurs.

8.4.3 Jacobi Multi-GPU

Ce test constitue une preuve de concept de la partie GPU du compilateur *dSTEP*. Contrairement aux autres programmes, le code ici n'est pas entièrement généré et les tests sont effectués sur une grappe de PCs d'une salle de cours équipés de GPUs

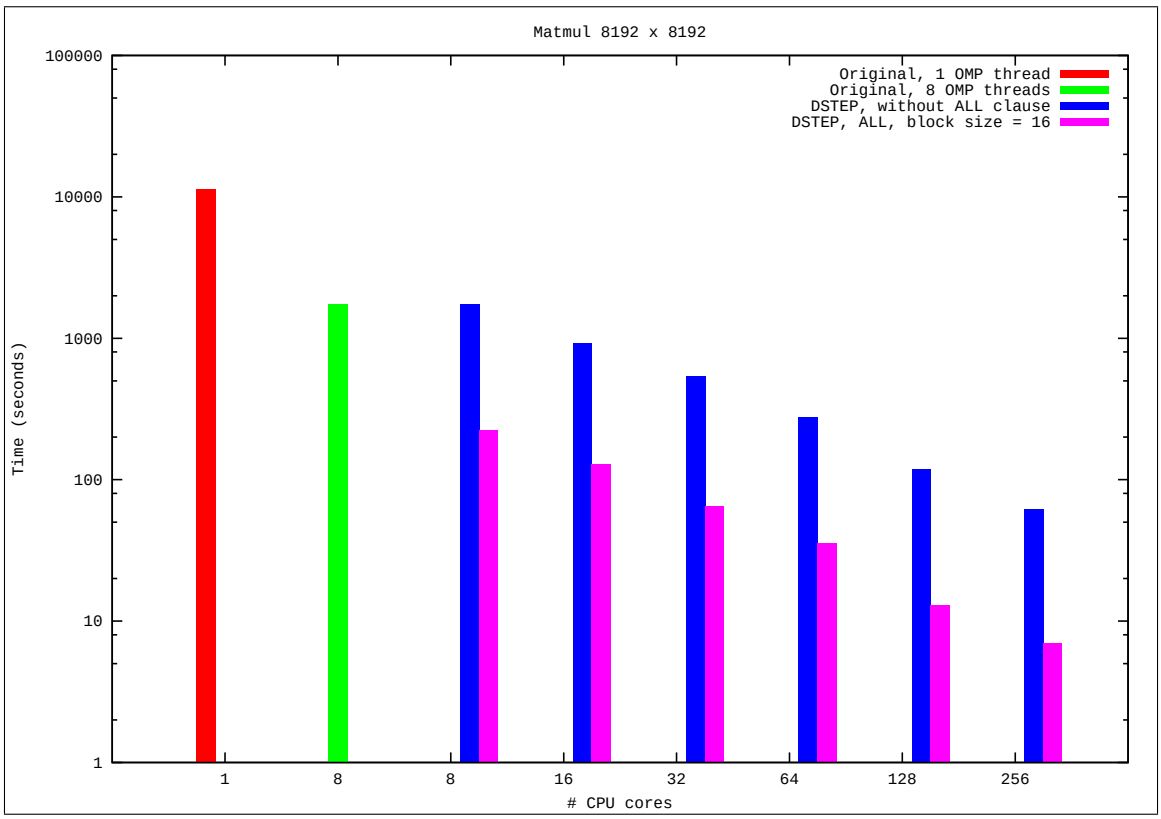


FIGURE 8.5 – Temps d'exécution de Matmul, N = 8192

et connectés par un réseau Ethernet. Les noyaux GPU sont écrits à la main, tandis que les appels à ces derniers sont générés ainsi que le reste du code. Nous utilisons deux jeux de données : *Small*, qui représente un temps de calcul significatif mais dont l'empreinte mémoire est inférieure à la capacité d'un seul GPU ; et *Large*, qui dépasse la capacité mémoire d'un seul GPU. Les accélérations obtenues par rapport à une exécution sur un seul cœur CPU sont représentées par la figure 8.8. Nous atteignons, sur 8 GPUs, une accélération de 108 pour la version *Large* et de 77 pour la version *Small*.

8.5 Le kernel Adi

Nous montrons ici la parallélisation du programme Adi de la suite Polybench [81]. Ce programme présente une succession de nids de boucles à deux dimensions avec la première et la deuxième dimension alternativement parallèle et ordonnée.

8.5.1 Choix des directives

On étudie deux types de distribution : la distribution multi-partitionnée et la distribution standard (non multi-partitionnée).

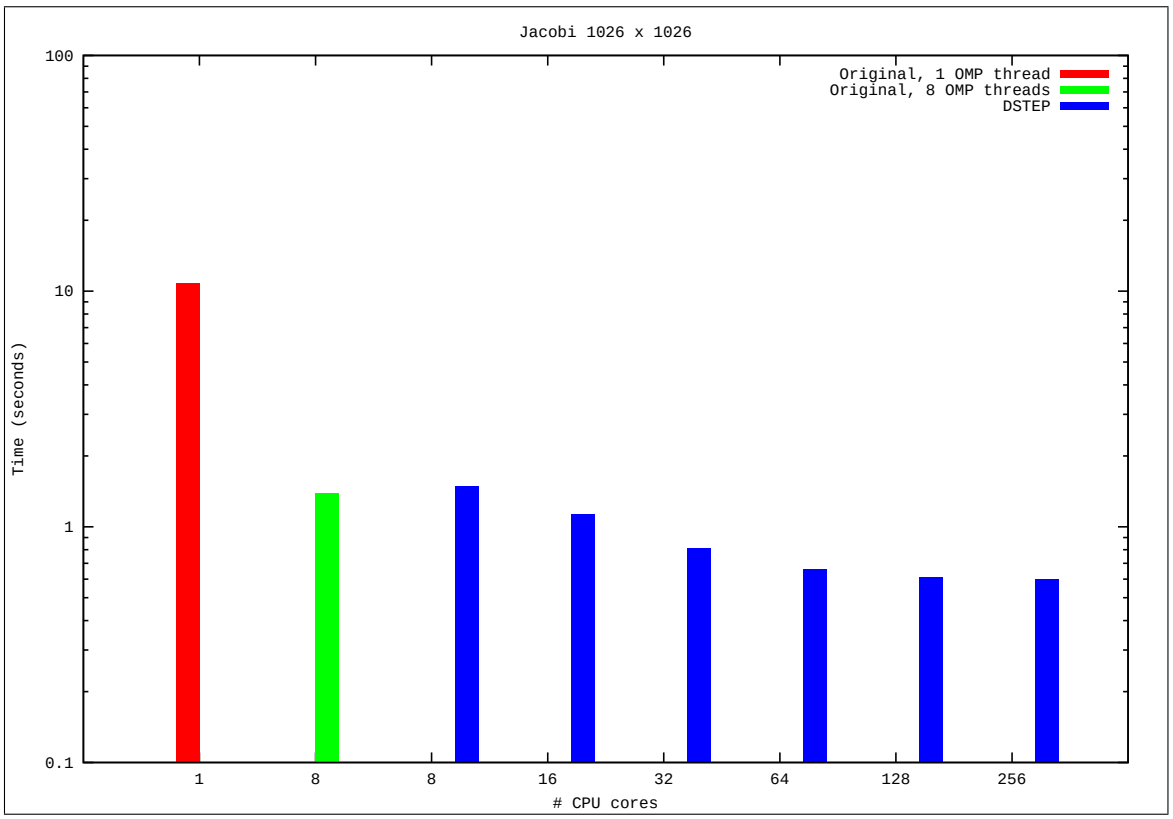


FIGURE 8.6 – Temps d'exécution de Jacobi, 1026 x 1026

Le listing 8.4 montre le code du programme Adi annoté de directives *dSTEP* pour la version standard. On montre en commentaire les directives utilisées pour la version multi-partitionnée avec une distribution diagonalisée de la deuxième dimension des matrices A, B et X et des nids de boucles. Les matrices A, B et X sont de taille 16000×16000 et le programme effectue 10 appels au kernel Adi.

8.5.2 Évaluation des performances

Nous comparons les temps d'exécution des versions standard et multi-partitionnée (fig. 8.9). Pour la version multi-partitionnée, nous étudions deux stratégies de complétion des communications. La complétion explicite consiste à compléter toutes les communications d'un nid de boucles juste après sont exécution. La complétion implicite quant à elle consiste à ne compléter les communications que lors de l'accès au tableau impliqué dans la communication. Théoriquement, la seconde stratégie est meilleure car elle permet de recouvrir tous les calculs intermédiaires, du déclenchement de la communication jusqu'au prochain accès à un tableau, avec la progression de la communication. En pratique, nous n'avons pas observé de gain de performance par rapport à la complétion explicite des communications. Ce phénomène s'explique par l'implémentation MPI utilisée qui implémente l'envoi des messages de grande taille par un mécanisme de rendez-vous. En effet, le processeur initiateur d'une communication envoie une demande de rendez-vous au récepteur du

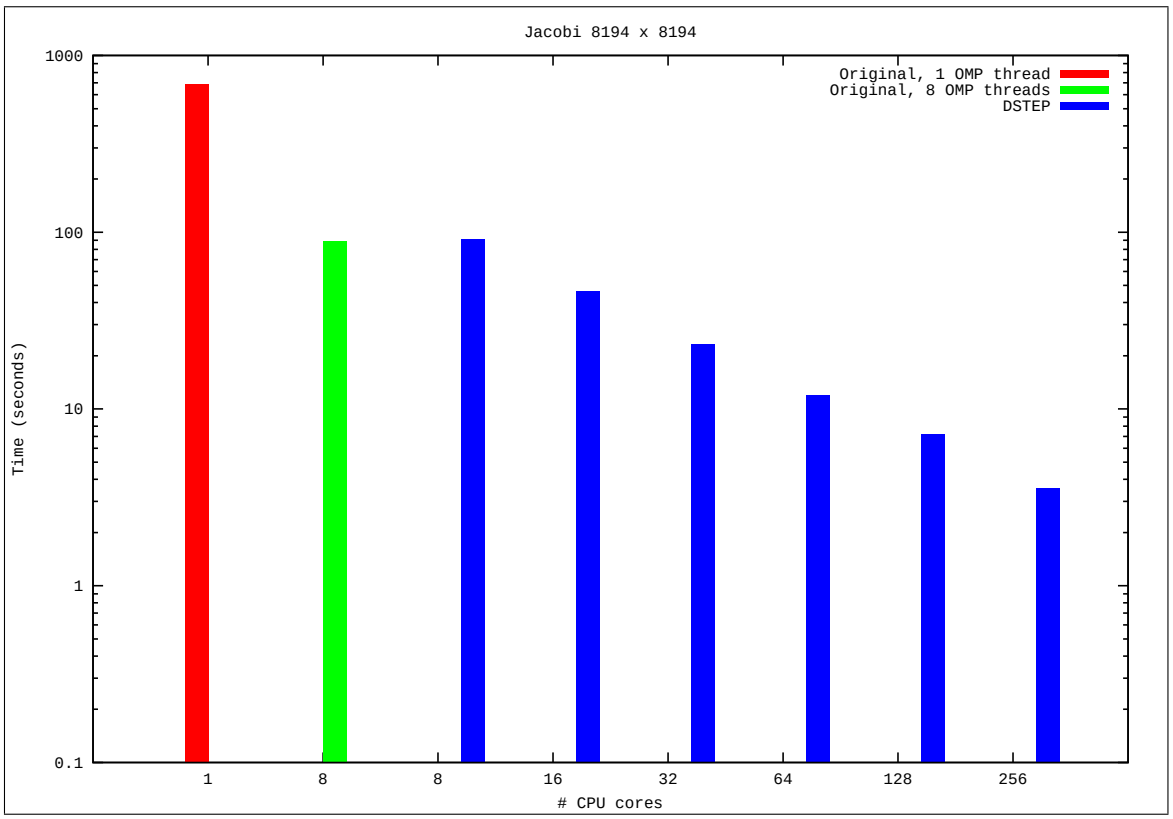


FIGURE 8.7 – Temps d'exécution de Jacobi, 8194 x 8194

message. Ce dernier renvoie un acquittement pour signaler qu'il peut commencer à recevoir les données. C'est à la réception de cet acquittement que l'envoi effectif des données commence. Dans la pratique, l'acquittement n'est envoyé que lors du test de la complétion de la communication du côté du récepteur, perdant ainsi tout l'avantage espéré du recouvrement. Une technique proposée par Saif et al. [83] permet, grâce à des tests de complétion insérés juste après l'appel asynchrone à MPI_Irecv d'envoyer un acquittement très tôt et de déclencher aussitôt l'envoi des données. Nous avons testé cette technique dans la stratégie de complétions implicites mais nous n'avons pas observé d'amélioration par rapport à la version explicite des complétions. Les travaux de Brunet et Trahay [88] sur la progression des communications apporte des solutions à ce problème. Ces améliorations interviennent au niveau du moteur des communications et dépassent le contexte de ce travail.

Nous observons que la distribution multi-partitionnée est plus performante que la version standard jusqu'à 128 cœurs. Sur 256 cœurs en revanche, la distribution standard devient plus performante. Cette observation est conforme aux prévisions du modèle de coût défini au chapitre précédent 7. En effet, en présence de dimensions ordonnées de nids de boucles, il existe une limite en nombre de processus à partir de laquelle le temps gagné en calcul est dépassé par le temps perdu en communications supplémentaire pour la distribution multi-partitionnée, la distribution standard devenant ainsi plus performante.

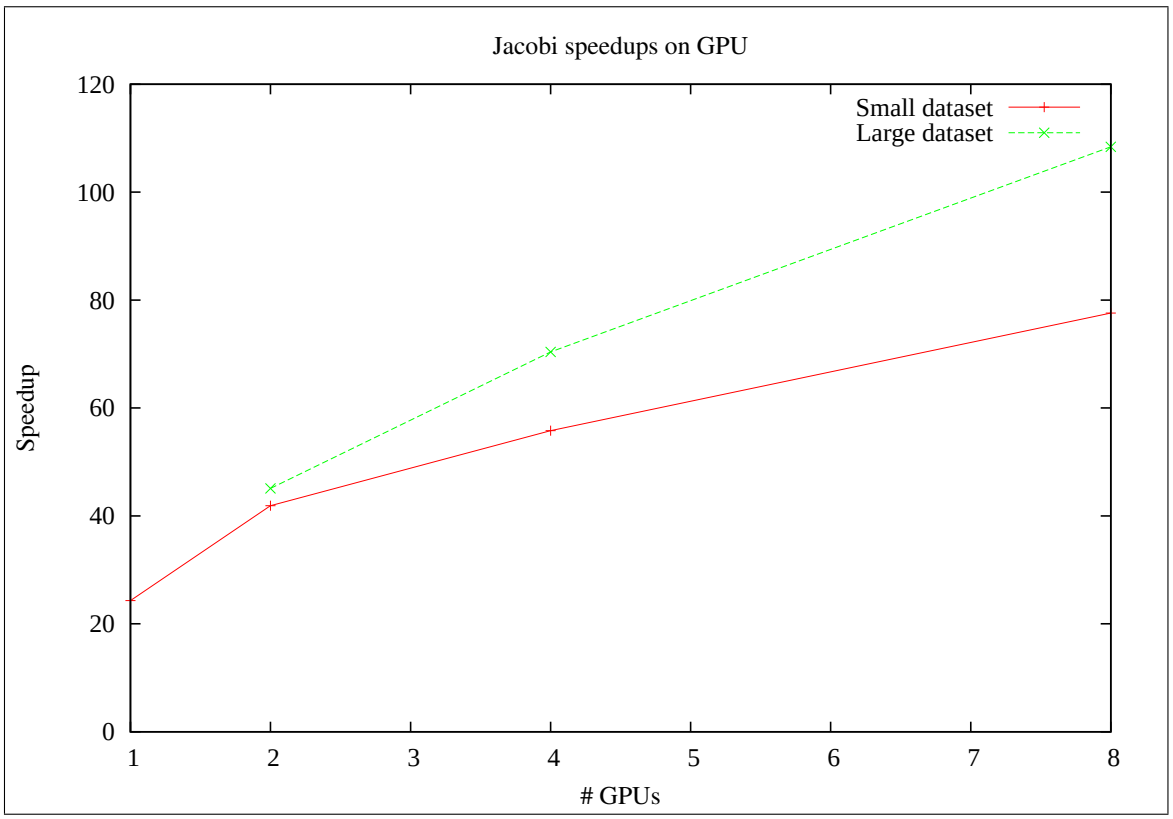


FIGURE 8.8 – Jacobi multi-GPU

Nous obtenons sur 256 cœurs une accélération de 41 pour la version multi-partitionné et de 51 pour la version standard.

8.6 NAS BT

Les *NAS Parallel Benchmarks* [11] sont un ensemble de programmes de calcul scientifique développés par la NASA. Ces programmes sont dérivés d'applications en aéro-physique mais sont plus largement représentatifs d'applications réelles de calcul scientifique. La suite des NAS Benchmarks comprend 5 petits programmes et 3 applications : BT, SP et LU. À partir de la version 3.1 de la spécification, de nouveaux programmes ont été ajoutés, notamment des versions *multi-zones* des applications BT, SP et LU pour la parallélisation hybride. La version 2.3 des NAS constitue une implémentation de référence en Fortran contenant à la fois une version séquentielle et une version MPI avec les classes de taille de données W, S, A, B et C, en ordre croissant. La version 2.4 ajoute notamment la classe D.

Nous montrons dans cette section la parallélisation avec *dSTEP* du programme BT pour les classes C et D et nous comparons les résultats obtenus à l'implémentation MPI Fortran de référence ainsi qu'à la version UPC de la classe C.

Le programme BT résout l'équation de *Navier-Stokes* de la forme $Ku = r$ en utilisant une intégration implicite alternée pour factoriser la matrice K [44]. La figure 8.10

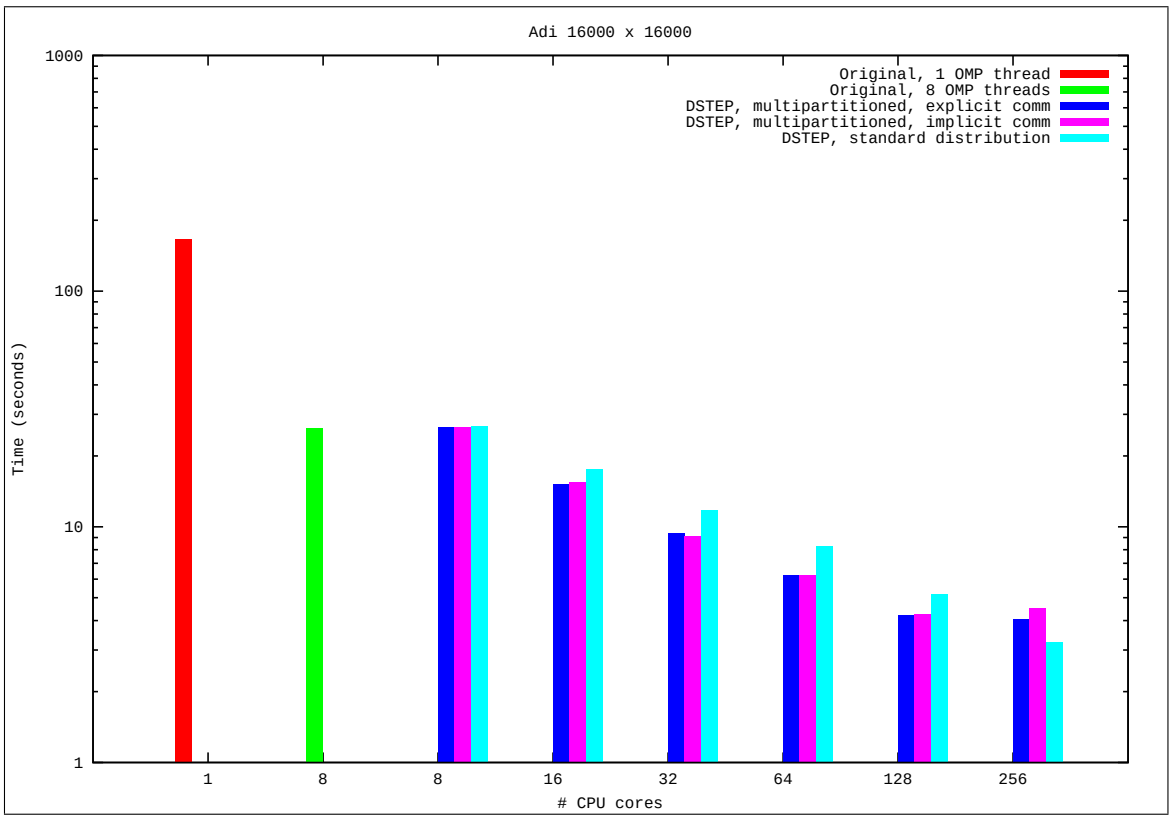


FIGURE 8.9 – Temps d'exécution du programme Adi

montre le graphe d'appel de la fonction `adi`, qui constitue le cœur de calcul de l'application BT. Les fonctions $[xyz]$*_solve_cell* et $[xyz]$*_backsubstitute* contiennent plusieurs nids de boucles avec les dimensions x, y et z qui sont ordonnées, respectivement.

8.6.1 Mise en œuvre de la distribution

Nous sommes partis d'une version de BT écrite en OpenMP C par les auteurs du projet de compilateur Omni [58]. Cette version a été traduite par les auteurs du code de référence des NAS, version 2.3, écrit en Fortran MPI. Ils ont ensuite inséré des directives OpenMP pour indiquer les boucles parallèles. A partir de ce code, nous avons mis tous les tableaux globaux dans la fonction *main*, à l'exception des tableaux cuf, q, ue et buf qui sont locaux à la fonction *exact_rhs*. Ces derniers tableaux sont étendus de deux dimensions afin de pouvoir distribuer les nids de boucles où ils sont référencés en même temps que d'autres tableaux distribués sur trois dimensions. Les tableaux $fjac$ et $njac$, bien qu'ils soient locaux aux fonctions $lhs[xyz]$, sont déclarés dans la fonction main et passés en paramètres à ces fonctions afin d'éviter leur allocation et destruction à chaque appel de ces fonctions dans les différents appels à la fonction *adi* dans le code généré.

Pour paralléliser BT, nous avons utilisé une distribution non multi-partitionnée des tableaux et des nids de boucles sur leurs trois premières dimensions. Pour les nids de boucles, nous avons indiqué les dimensions qui sont ordonnées ainsi que les

dimensions *owner*. Voici par exemple le code de la fonction *y_backsubstitute* annoté de la directive *step gridify* 8.5.

```
1 #pragma step gridify(i, j(dist=block; sched=ordered), k) private(m,
    n)
2   for (i = 1; i < grid_points[0]-1; i++) {
3     for (j = grid_points[1]-2; j >= 0; j--) {
4       for (k = 1; k < grid_points[2]-1; k++) {
5         for (m = 0; m < BLOCK_SIZE; m++) {
6           for (n = 0; n < BLOCK_SIZE; n++) {
7             rhs[i][j][k][m] = rhs[i][j][k][m]
8               - lhs[i][j][k][CC][m][n]*rhs[i][j+1][k][n]; }}}}}
```

Listing 8.5 – Parallélisation de la fonction y_backsubstitute par la directive step gridify.

Au total, nous avons inséré 16 directives *step distribute* et 72 directives *step gridify* correspondant à autant de nids de boucles parallélisés.

Nous avons utilisé deux classes pour BT : C et D. La classe C correspond aux paramètres du code d'origine, sans modifications. Pour obtenir la classe D à partir de cette même implémentation, nous avons modifié certains paramètres de BT, conformément à la spécification des NAS classe D [92] et avons inséré les valeurs de vérification correspondant à cette classe. Le tableau 8.1 présente les paramètres des classes C et D ainsi que les valeurs de référence pour la vérification des résultats.

	Classe C	Classe D
Taille de la grille de discrétisation	$162 \times 162 \times 162$	$408 \times 408 \times 408$
Pas d'intégration	1×10^-4	2×10^-5
Nombre de pas	200	250
Norme résiduelle	0.62398116551764615e+04	0.2533188551738e+05
	0.50793239190423964e+03	0.2346393716980e+04
	0.15423530093013596e+04	0.6294554366904e+04
	0.13302387929291190e+04	0.5352565376030e+04
	0.11604087428436455e+05	0.3905864038618e+05
Norme de l'erreur	0.16462008369091265e+03	0.3100009377557e+03
	0.11497107903824313e+02	0.2424086324913e+02
	0.41207446207461508e+02	0.7782212022645e+02
	0.37087651059694167e+02	0.6835623860116e+02
	0.36211053051841265e+03	0.6065737200368e+03

TABLE 8.1 – Valeurs des paramètres des classes C et D pour le programme BT

8.6.2 Optimisations

Nous avons appliqué deux types d'optimisations sur le code généré par *dSTEP* afin d'améliorer ces performances : l'échange de boucles, et l'adaptation du halo.

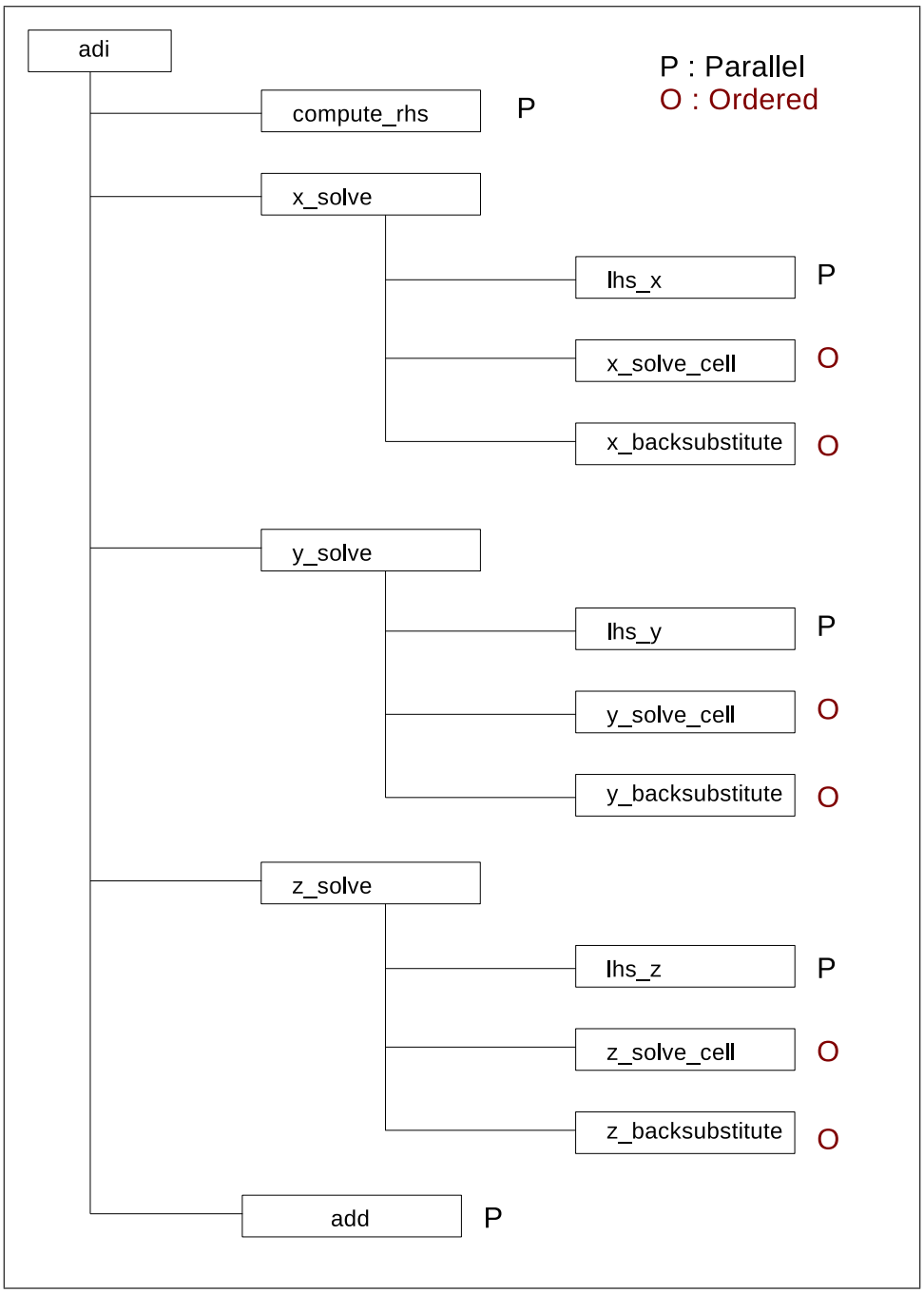


FIGURE 8.10 – Graphe d'appel de la fonction adi dans le programme BT.

Échange de boucles. L'échange de boucles a été appliqué aux nids de boucles i, j, k où la boucle i est ordonnée. Dans le code non optimisé, la directive OpenMP *parallel for* est insérée pour la boucle j, avec passage de la valeur de l'itération i englobante dans une clause OpenMP $firstprivate$. Dans ce cas, les threads partageant les itérations de la boucles j sont créés et détruits pour chaque itération de la boucle i. Échanger les boucles i et j permet de créer les threads de la boucle j une seule fois.

Adaptation des halos. L'adaptation des halos consiste à mettre à zéro la valeur des halos sur certaines dimensions des tableaux *lhs* et *rhs*. En effet, dans la fonction *x_solve* de *adi*, seuls les halos de la première dimension sont définies et utilisés. Il en va de même pour les autres dimensions dans les fonctions *y_solve* et *z_solve*. Cette optimisation permet de réduire le nombre et le volume des communications.

8.6.3 Résultats pour la classe C

Nous utilisons jusqu'à 32 nœuds du cluster Edel, correspondant à 256 cœurs. Sur chaque nœud, nous exécutons autant de processus MPI que de cœurs disponibles, à savoir huit, pour la version Fortran MPI du code. Pour le code généré par *dSTEP*, nous exécutons un processus MPI et huit threads OpenMP par nœud. Pour obtenir le temps d'exécution des codes de référence cependant, nous exécutons les codes d'origine sur un seul cœur (un processus MPI pour le code Fortran, un thread OpenMP pour le code C d'origine). Nous mesurons également le temps d'exécution du code généré par *dSTEP* sur un seul cœur avec un processus MPI et un thread OpenMP.

La figure 8.11 montre les gains apportés par les deux optimisations présentés à la section précédente par rapport à la version non optimisée du code généré sur 8 nœuds. Nous observons une amélioration cumulée des deux optimisations de l'ordre de 17% par rapport au code non optimisé. Dans la suite des résultats présentés, nous avons toujours activé ces deux optimisations.

La figure 8.12 montre les temps d'exécution obtenus avec *dSTEP* pour BT classe C comparés aux résultats de l'implémentation MPI Fortran de référence ainsi que le que la version UPC. Nous constatons d'abord que le code original en OpenMP C est moins performant sur un cœur que le code Fortran séquentiel. Nous observons également que le code généré par *dSTEP* est meilleur que le code d'origine sur un cœur. Le code généré est cependant moins performant que le code de référence en MPI Fortran car ce dernier utilise une distribution multi-partitionnée des données alors que nous avons utilisé une distribution standard. Nous notons enfin que le code généré est plus performant que la version UPC et présente un bon passage à l'échelle. L'exécution des versions MPI Fortran et UPC de BT requièrent un nombre carré de processus (threads pour UPC), il n'est donc pas possible de les exécuter sur 32 et 128 cœurs.

Utilisation d'un réseau TCP

Au chapitre 7 sur le modèle de coût de la distribution, nous avons identifié le rôle du ratio entre la vitesse de calcul et celle des communications dans le bénéfice que peut apporter une distribution multi-partitionnée par rapport à une distribution standard. Dans ce test, nous changeons la valeur de ce ratio en utilisant le réseau TCP, qui est plus lent que le réseau Infiniband. Les résultats obtenus (fig. 8.13) montre une dégradation des performances à la fois du code généré et du code de référence. Nous observons surtout que la version de référence MPI Fortran est moins performante, à partir de 64 processus, que le code généré par *dSTEP* pour la distribution standard,

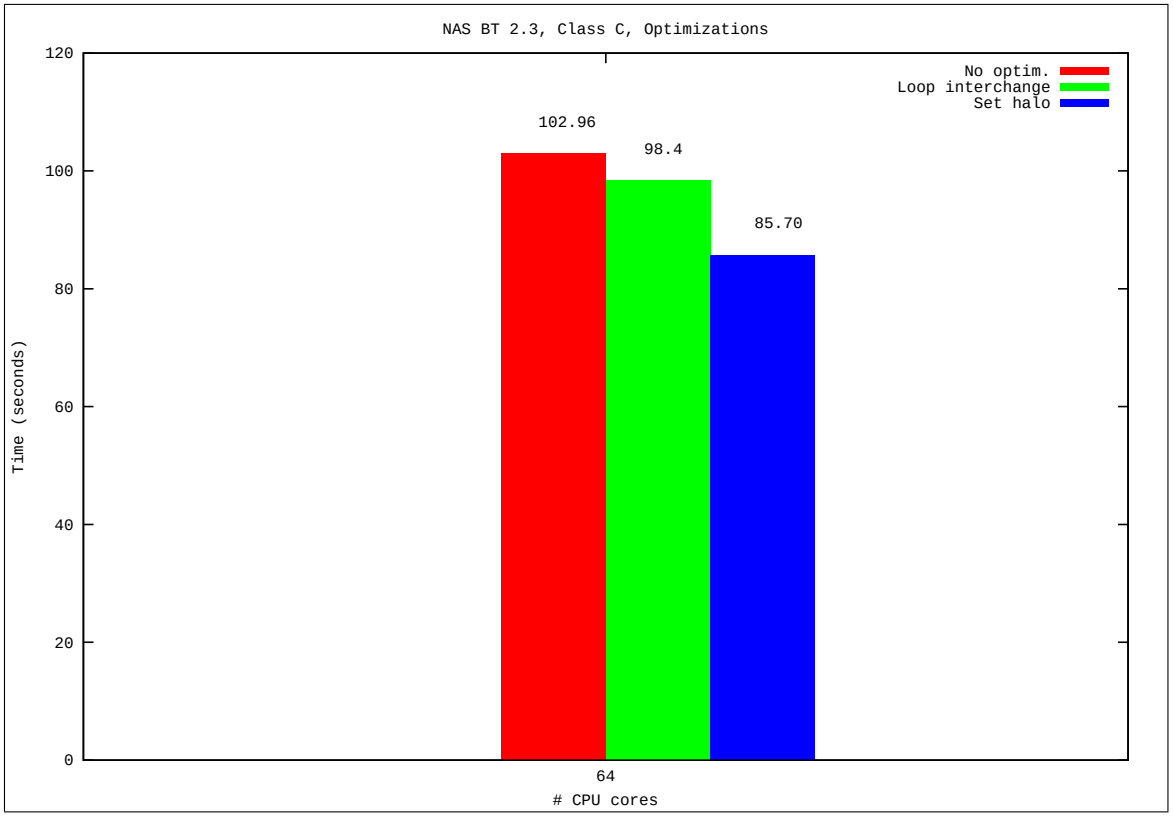


FIGURE 8.11 – Optimisations pour BT

à cause du coût prohibitif des communications supplémentaires induite par le *muti-partitioning*.

Nombre de processus à l'exécution

Les nids de boucles et les tableaux du programme BT sont distribués sur trois dimensions. À l'exécution, des grilles virtuelles (voir la section 4.2.2 sur les grilles virtuelles de processus) à trois dimensions sont construites pour les tableaux et les nids de boucles. Mais pour que la distribution soit effectivement à trois dimensions, il faut que le nombre de processus soit factorisable en trois nombres différents de l'unité. Ainsi, sur neuf processus, les grilles virtuelles construites sont de la forme $3 \times 3 \times 1$, ce qui équivaut à une distribution à deux dimensions. Avec un nombre premier tel que $11 = 11 \times 1 \times 1$, la distribution effective équivaut à une distribution mono-dimensionnelle. La figure 8.14 montre les temps d'exécution avec les trois configurations ci-dessus. Nous observons que la configuration avec huit processus, qui correspond effectivement à une distribution à trois dimensions, est plus performante que les configurations à neuf et onze processus.

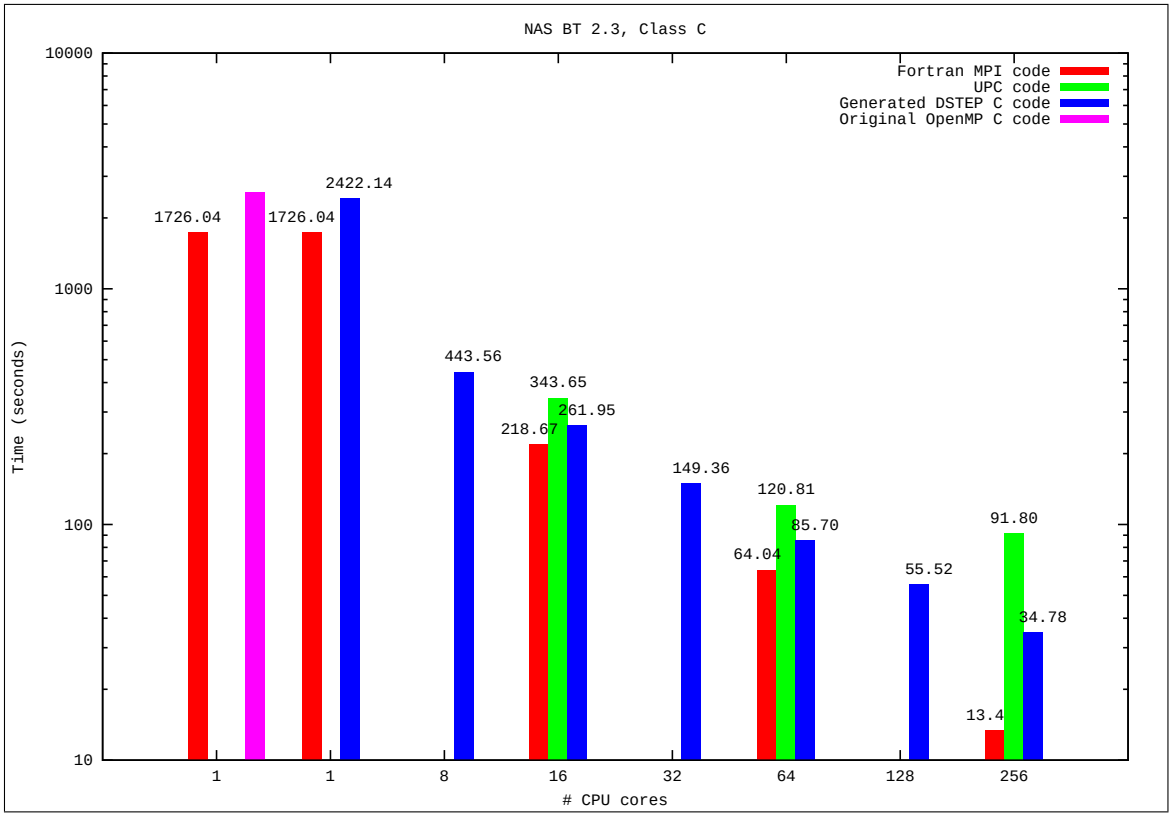


FIGURE 8.12 – Temps d'exécution du code généré pour BT classe C, comparaison avec la version MPI Fortran et UPC

8.6.4 Résultats pour la classe D

Pour la classe D, la taille des données est tellement importante que l'exécution sur un seul nœud est impossible sans l'utilisation de la mémoire *swap*. Le code généré peut être exécuté à partir de huit nœuds alors que le code de référence est exécutable à partir de quatre nœuds. Ceci est dû à l'extension des tableaux locaux à la fonction $exact_rhs$ de deux dimensions pour les besoins de la distribution. La figure 8.15 montre temps d'exécution obtenus. Comme nous ne disposons pas des temps d'exécution sur un seul cœur, nous ne pouvons pas comparer les performances des deux versions. Nous observons à la fois un bon passage à l'échelle du code généré mais des performances moins importantes que la version MPI Fortran.

Remarque. Nous avons généré et testé une version multi-partitionnée pour BT qui donne des résultats corrects pour les classes C et D. Cependant, Nous n'avons pas atteint les performances souhaitées avec ce type de distribution pour BT au moment de la présentation de ces résultats. Le travail sur l'amélioration de l'implémentation de la distribution multi-partitionnée est en cours.

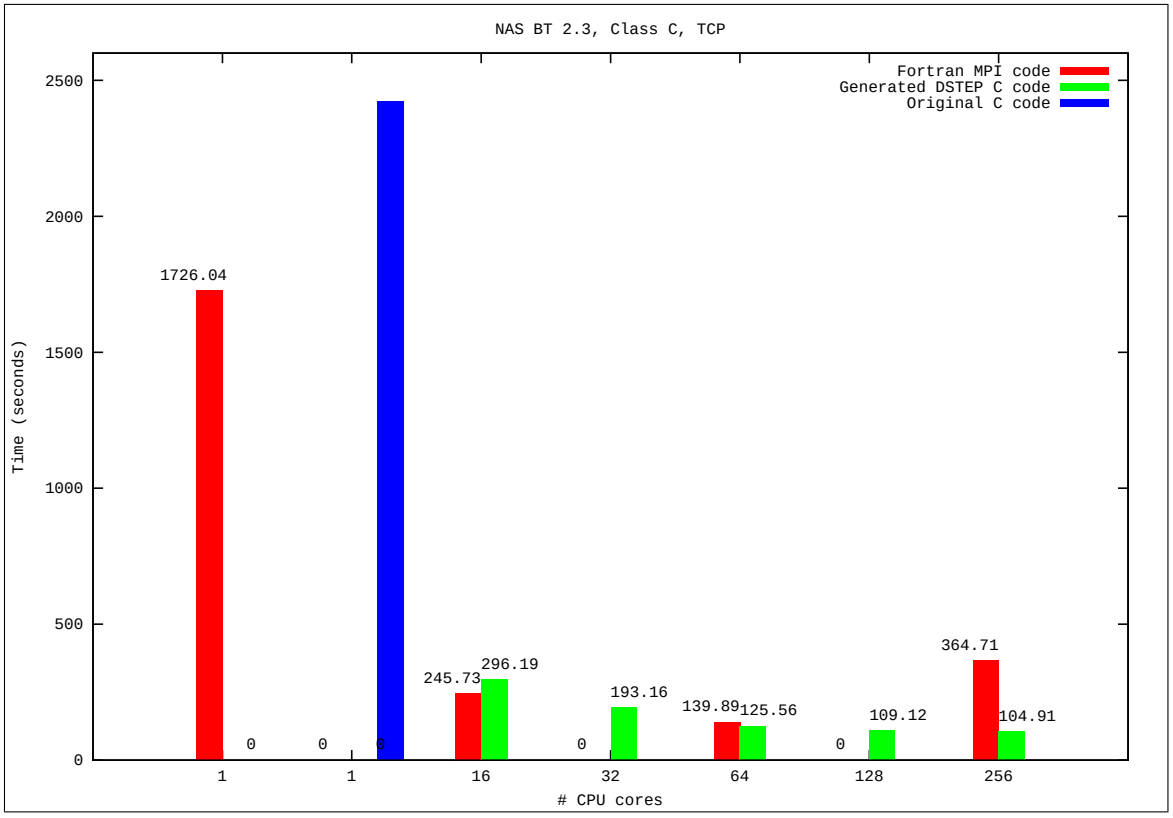


FIGURE 8.13 – Temps d'exécution du code généré pour BT classe C, comparaison avec la version MPI Fortran, utilisation d'un réseau TCP

8.7 LDPC

Le programme LDPC est un code industriel de communication radio. Dans le cadre du projet européen Many, nous avons travaillé à la génération de la version distribuée de LDPC après détection et exhibition du parallélisme par le compilateur PoCC [80]. Les tests sont effectués sur un petit cluster de processeurs *Odroids* ayant quatre cœurs par processeur. Nous obtenons, sur 16 cœurs, une accélération de 10 par rapport au code en entrée généré par PoCC et exécuté sur un cœur (voir la figure 8.16).

8.8 Conclusion

Nous avons montré dans ce chapitre l'utilisation des directives *dSTEP* pour la parallélisation de programmes représentatifs du calcul scientifique dense ainsi que sur une application industrielle. Nous avons montré qu'avec un effort raisonnable de la part du programmeur, consistant à décorer le programme en entrée avec les directives

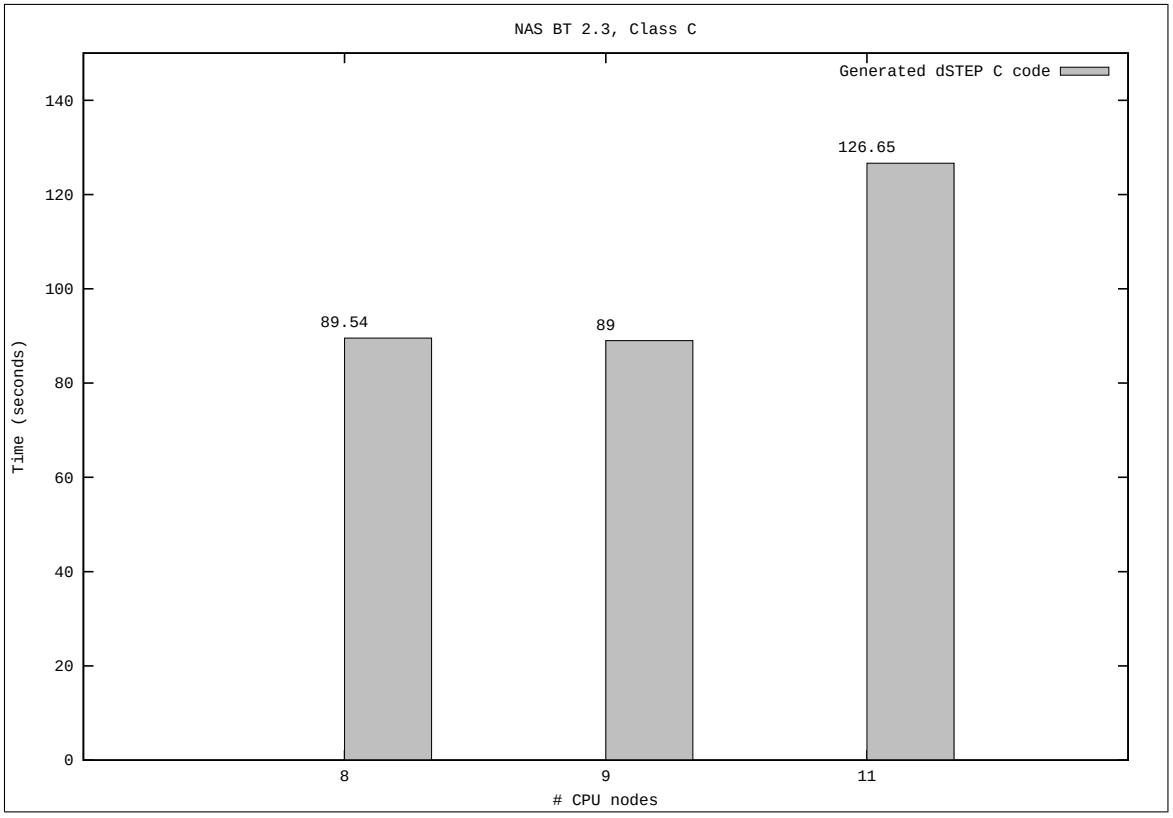


FIGURE 8.14 – Temps d'exécution du code généré pour BT classe C avec plusieurs formes de grilles virtuelles de processus

dstep distribute et *dstep gridify*, le compilateur *dSTEP* permet de générer des codes parallèles performants. Nous avons obtenu une accélération de 1632 pour la multiplication de matrices sur 256 cœurs pour des données de grande taille grâce à l'utilisation d'une distribution multi-dimensionnelle combinée avec l'utilisation du type de distribution *all*. Pour l'application NAS BT, en considérant les accélérations des différentes versions par rapport à leur exécution de référence, nous obtenons 110% (resp. 57%)des performances de la version Fortran MPI sur 64 cœurs (resp. 256 cœurs).

Nous avons également montré l'intérêt de *dSTEP* pour le traitement de problèmes de très grande taille, impossible à exécuter si les données étaient répliquées, comme c'est le cas avec le programme NAS BT pour la classe D. Nous avons enfin montré l'applicabilité de la solution proposée sur une application industrielle et son interaction avec un outil générant les directives en amont.

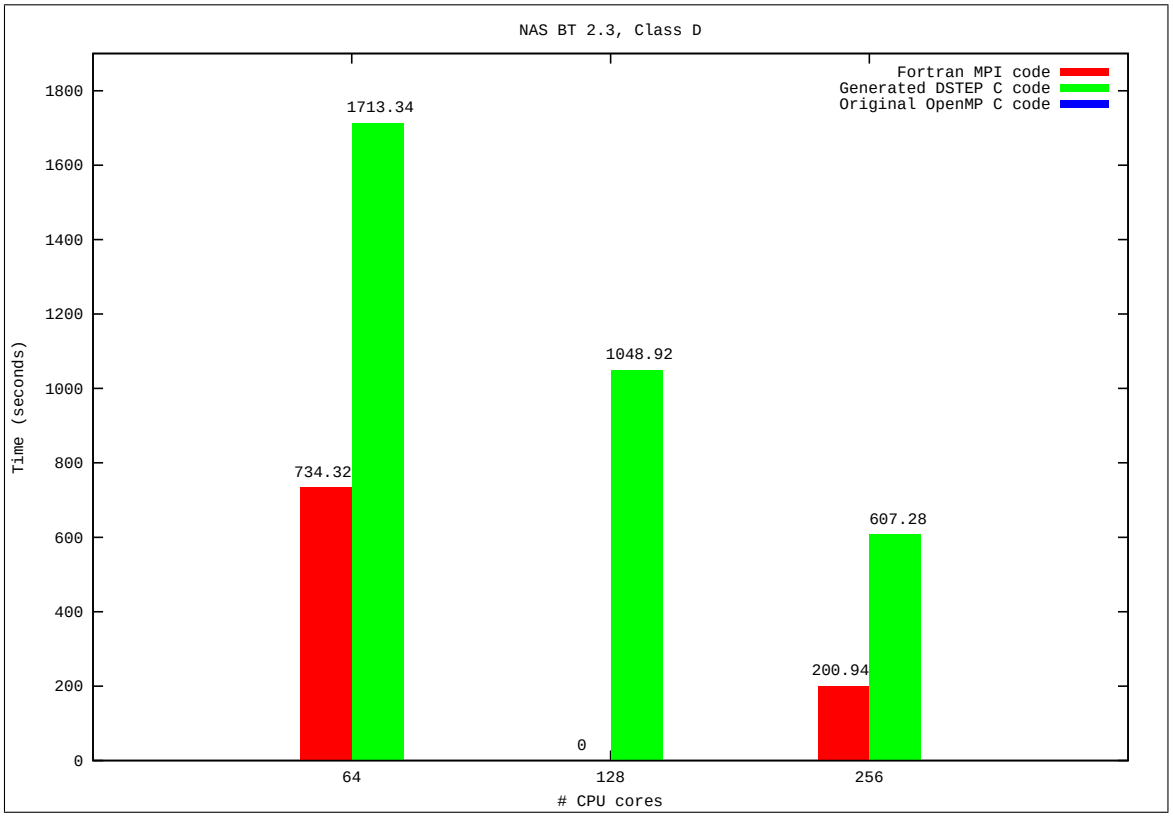


FIGURE 8.15 – Temps d'exécution du code généré pour BT classe D, comparaison avec la version MPI Fortran

```
1  void kernel_adi(int tsteps, int n, double X[n][n], double A[n][n], double B[n][n])
2  {
3    int t, i1, i2;
4    for (t = 0; t < tsteps; t++) {
5  //#pragma step gridify(i1, i2(dist=block, diag; sched=ordered))
6   #pragma step gridify(i1, i2(dist=block; sched=ordered))
7        for (i1 = 0; i1 < n; i1++)
8          for (i2 = 1; i2 < n; i2++) {
9              X[i1][i2] = X[i1][i2] - X[i1][i2-1] * A[i1][i2] /
10             B[i1][i2-1];
11             B[i1][i2] = B[i1][i2] - A[i1][i2] * A[i1][i2] /
12             B[i1][i2-1];
13           }
14 //#pragma step gridify(i1, i2(dist=*, diag; sched=owner))
15  #pragma step gridify(i1, i2(dist=*; sched=owner))
16       for (i1 = 0; i1 < n; i1++)
17         for (i2 = n-1; i2 <= n-1; i2++)
18           X[i1][i2] = X[i1][i2] / B[i1][i2];
19 //#pragma step gridify(i1, i2(dist=block, diag; sched=ordered))
20  #pragma step gridify(i1, i2(dist=block; sched=ordered))
21       for (i1 = 0; i1 < n; i1++)
22         for (i2 = n-2; i2 >= 1; i2--)
23           X[i1][i2] = (X[i1][i2] - X[i1][i2-1] * A[i1][i2-1]) /
24           B[i1][i2-1];
25 //#pragma step gridify(i1(dist=block; sched=ordered), i2(dist=block, diag; sched=
       parallel))
26  #pragma step gridify(i1(dist=block; sched=ordered), i2)
27       for (i1 = 1; i1 < n; i1++)
28         for (i2 = 0; i2 < n; i2++)
29           {
30             X[i1][i2] = X[i1][i2] - X[i1-1][i2] * A[i1][i2] /
31             B[i1-1][i2];
32             B[i1][i2] = B[i1][i2] - A[i1][i2] * A[i1][i2] /
33             B[i1-1][i2];
34           }
35 //#pragma step gridify(i1(dist=*; sched=owner), i2(dist=block, diag; sched=parallel))
36  #pragma step gridify(i1(dist=*; sched=owner), i2)
37       for (i1 = n-1; i1 <= n-1; i1++)
38         for (i2 = 0; i2 < n; i2++)
39           X[i1][i2] = X[i1][i2] / B[i1][i2];
40 //#pragma step gridify(i1(dist=block; sched=ordered), i2(dist=block, diag; sched=
       parallel))
41  #pragma step gridify(i1(dist=block; sched=ordered), i2)
42       for (i1 = n-2; i1 >= 1; i1--)
43         for (i2 = 0; i2 < n; i2++)
44           X[i1][i2] = (X[i1][i2] - X[i1-1][i2] * A[i1-1][i2]) /
45           B[i1][i2];
46     }
47 }
48 int main(void) {
49   int n = N, tsteps = TSTEPS;
50 //pragma step distribute X(4:block:4, 4:diag:4)
51  #pragma step distribute X(4:block:4, 4:block:4)
52   double X[n][n];
53 //pragma step distribute A(4:block:4, 4:diag:4)
54  #pragma step distribute A(4:block:4, 4:block:4)
55   double A[n][n];
56 //pragma step distribute X(4:block:4, 4:diag:4)
57  #pragma step distribute B(4:block:4, 4:block:4)
58   double B[n][n];
59
60   init_array (n, X, A, B);
61   kernel_adi (tsteps, n, X, A, B);
62   print_array(n, X);
63   return 0;
64 }
```

Listing 8.4 – Le code annoté du programme adi

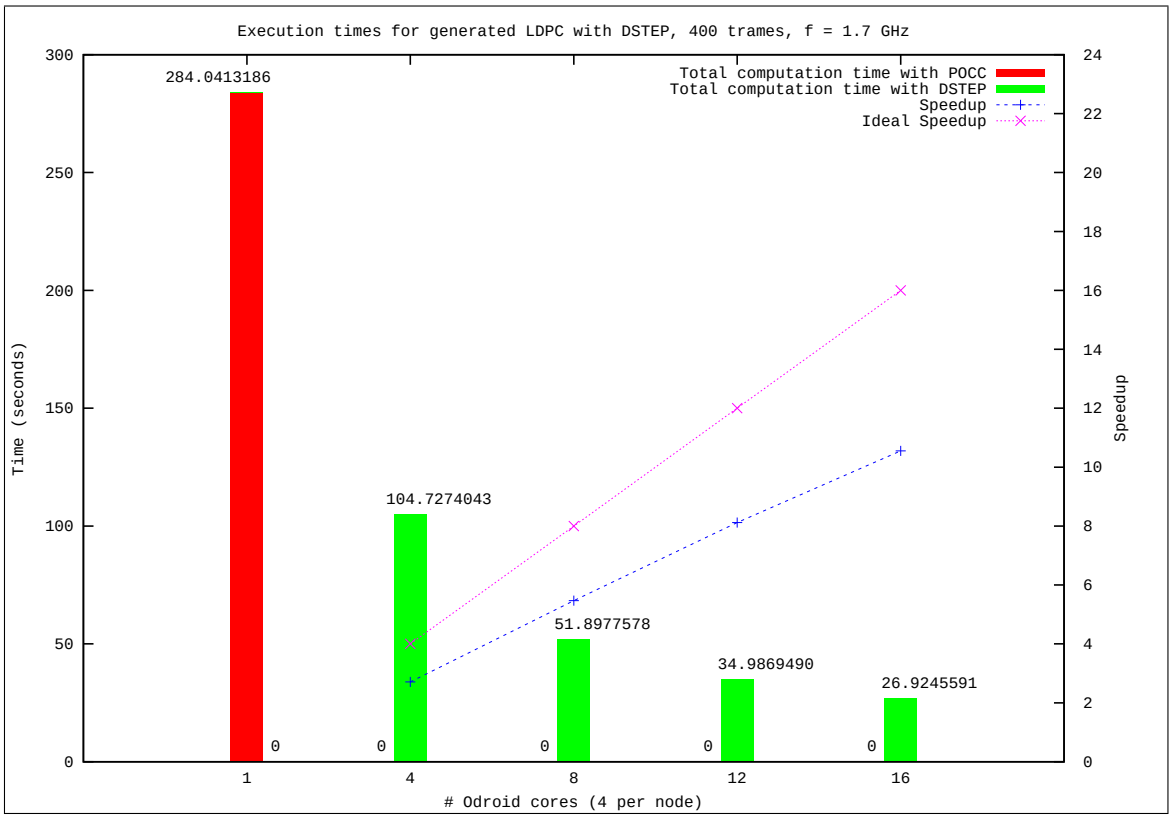


FIGURE 8.16 – LDPC

Chapitre 9

Conclusion

9.1 Rappel du contexte

Dans cette thèse, nous avons proposé *dSTEP* (*distributed STEP*), un modèle de programmation à base de directives pour la parallélisation d'applications scientifiques denses sur architectures parallèles hybrides. L'outil existant *STEP* [68] (*Système de Transformation pour l'Exécution Parallèle*), qui est le point de départ de ce travail, permet la transformation de programmes écrits en OpenMP ou HMPP en des programmes en MPI et OpenMP ou en MPI et HMPP respectivement, avec des optimisations dynamiques des communications [85]. Cependant dans cette transformation, les données sont répliquées dans le code généré, ce qui constitue une limitation au traitement de problèmes de très grande taille. De plus, une seule dimension est traitée pour chaque nid de boucles : la dimension parallèle. *dSTEP* permet de traiter à la fois la distribution des calculs et des données ainsi que de généraliser la distribution à un nombre quelconque de dimensions, pour tout type d'ordonnancement.

9.2 Contributions

Les contributions de cette thèse sont :

1. un modèle de programmation à base de directives pour la distribution des données et des calculs des programmes scientifiques denses,
2. un modèle de distribution et un schéma de compilation générique pour une génération de code correct et efficace,
3. l'implémentation d'un prototype du compilateur *dSTEP* et d'un support d'exécution,
4. des résultats expérimentaux.

9.2.1 Modèle général pour la distribution des calculs et des données

Le modèle de programmation de *dSTEP* sépare la distribution des données et celle des calculs. Ainsi, le programmeur a un contrôle plus fin sur la parallélisation de son programme. Il n'y a pas de notion de propriétaire des données. Ce modèle prend en compte un nombre quelconque de dimensions pour les tableaux et les nids de boucles. En plus de la distribution des itérations pour un nid de boucles, l'ordonnancent des itérations est pris en compte. Un halo est utilisé pour agrandir l'espace mémoire alloué sur chaque processus pour un tableau distribué afin de garantir la localité des accès des différents nids de boucles à ce tableau. Nous proposons également un halo *total*, qui recouvre la totalité des éléments d'une dimension d'un tableau, ce qui permet d'exprimer dans *dSTEP* des programmes où tous les éléments d'une dimension d'un tableau sont balayés sur tous les processus au niveau de certains nids de boucles, tout en évitant la réplication de la dimension.

9.2.2 Schéma générique de compilation

Nous avons proposé un modèle de distribution qui définit formellement la distribution et en ressort les propriétés. Nous avons ensuite dérivé de ce modèle de distribution un schéma de compilation générique dans lequel nous prouvons la correction du code généré. Le changement de domaine, c'est-à-dire la génération des références aux tableaux dans l'espace distribué est effectué de façon générique, pour tout type de distribution de tableau. Au niveau de l'ordonnancement des itérations, deux cas sont distingués : parallèle et ordonné. Dans le dernier cas, l'ordonnancement initial est assuré dans une exécution distribuée en insérant des communications bloquantes : pour un nid de boucles ordonné, un processus ne démarre le calcul qu'une fois tous les processus exécutant des itérations *précédentes* dans l'ordre initial ont terminé le calcul.

9.2.3 Génération de communications efficaces

La génération des communications est basée sur les halos. Naturellement, plus les halos sont grands, plus les messages échangés sont de tailles importantes. Les communications générées sont des *sends*, des *recvs* ainsi que des réductions parallèles. Les *sends* sont toujours asynchrones, ce qui permet au calcul d'avancer lors de l'exécution de plusieurs tranches d'itérations au niveau d'un seul processus. Les *recvs* sont asynchrones pour un nid de boucles parallèle et effectués en deux temps pour un nid de boucles ordonné : synchrones avant le calcul et le restant des messages en asynchrone par la suite. La complétion des communications se fait lors du changement de domaine au niveau d'un nid de boucles, ce qui garantit qu'une communication sur des éléments de tableau est terminée au moment de leur utilisation mais pas avant. Ainsi, les calculs qui s'intercalent entre le début de la communication et la prochaine utilisation du tableau peuvent potentiellement être effectués en parallèle de la communication.

Cette propriété du code généré n'est cependant effective que si le moteur des communications utilisé implémente efficacement le recouvrement calculs/communications.

9.2.4 Implémentation d'un prototype

On a implémenté le schéma de compilation proposé en deux parties : un compilateur et un support d'exécution. Le compilateur *dSTEP* utilise la plate-forme de compilation *PIPS* [55]. Le support d'exécution implémente les fonctions de gridification, la vérification des accès, le calcul des régions à communiquer ainsi que les communications synchrones et asynchrones et les réductions.

9.2.5 Résultats expérimentaux

Nous avons montré l'applicabilité de la solution proposée sur des programmes de calcul scientifique tels que la multiplication de matrices, le solveur Jacobi et le programme Adi de la suite Polybench ainsi que sur l'application BT des *NAS Parallel Benchmarks*. Enfin, nous avons appliqué avec succès *dSTEP* à la parallélisation d'une application industrielle, en interaction avec le compilateur-optimiseur PoCC.

9.3 Originalités

Notre solution pour la programmation des architectures parallèles hybrides présente les originalités suivantes :

1. contrôle séparé par le programmeur de la distribution des données et de la distribution des calculs ;
2. proposition d'un large choix de distributions et d'ordonnancement ;
3. génération d'un flot de contrôle simple, implémentable à la fois sur CPU et GPU ;
4. un modèle de coût pour aider le programmeur à choisir la meilleure distribution pour un problème donné dans un environnement donné.

9.3.1 Séparation des distributions des données et des calculs

Les approches traditionnelles de la distribution, comme *HPF*, s'intéressent au placement des données et en dérivent la répartition des calculs et les communications. La notion de propriétaire de données y est très forte. En permettant au programmeur de contrôler à la fois la distribution des données et des calculs, on s'affranchit de la contrainte de propriétaire de données. L'utilisation du halo permet de garantir la localité des accès et de générer des communications efficaces, aux endroits opportuns du programme.

9.3.2 Plusieurs types de distributions et d'ordonnancement

On a intégré dans le même modèle de programmation, les distributions utilisés en calcul scientifique dense : bloc, cyclique, mutlti-partitionnée et répliquée. Le halo total ajoute de l'expressivité à la distribution en répliquant tous les éléments d'une dimension tout en la distribuant. Pour un nid de boucles, on traite à la fois les dimensions ordonnées et parallèles. Les accès isolés à des éléments de tableaux distribués sont traités par un ordonnancement particulier : *owner*.

9.3.3 Flot de contrôle simplifié

Une autre différence avec *HPF* est la génération d'un flot de contrôle simple pour le code parallèle. En effet, au changement de domaine près, les accès sont exactement les mêmes que dans le code en entrée. Cette propriété permet une implémentation simplifiée du schéma de compilation à la fois sur CPU et sur GPU.

9.3.4 Modèle de coût

On a proposé un modèle de coût propre au modèle de programmation de *dSTEP* pour aider le programmeur à choisir la meilleure distribution en fonction de plusieurs paramètres. Ce modèle permet notamment de quantifier le coût des communications, et permet de comprendre le comportement de plusieurs types de distributions en fonction des vitesses relatives du réseau et des ressources de calcul.

9.4 Limitations

Le compilateur *dSTEP* ne permet pas de générer des communications optimisées lorsque les références en écriture aux éléments de tableaux ne sont pas des fonctions affines des vecteurs d'itération. *dSTEP* ne traite donc pas les programmes à accès irréguliers comme le calcul scientifique creux. En effet, ces programme utilisent des vecteurs d'indirection dans les accès aux données, pour lesquels on ne sait pas générer de communications optimisées. Des techniques d'inspection/exécution ont été proposées pour traiter ce cas de figure, notamment dans les compilateurs HPF mais nous pensons que les PGAS, avec les accès asynchrones implémentés par des communications unilatérales au niveau de la bibliothèque de communication, sont mieux adaptés pour ce type d'accès.

9.5 Perspectives

9.5.1 Implémentation du backend GPU

Nous avons étendu le schéma de compilation de *dSTEP* pour la génération de code pour GPUs. Nous avons montré que seule la partie calcul sur GPU devait être adaptée.

Le flot de contrôle dans le domaine distribué étant simple, la version GPU des calculs est facilement implémentable. Les communications entre les différents processus sont inchangées. Des communications supplémentaires entre l'hôte et l'accélérateur sont insérées, dans le support d'exécution, autour de chaque communication inter-processus. Ainsi, ces transferts sont entièrement transparents pour le programmeur et ne concernent que les éléments nécessaires : utilisation des halos et des régions OUT de PIPS [30].

9.5.2 Optimisation au niveau du nœud de calcul

Le compilateur *dSTEP* génère un code parallèle hybride efficace en consommation mémoire, en communications et en temps de calcul. Au niveau de chaque nœud de calcul, des pragmas OpenMP sont générés pour les dimensions parallèles dans la version CPU. De plus, l'utilisation de la distribution *all* permet de découper les itérations d'une boucle en blocs sur tous les processus, augmentant ainsi la localité temporelle des accès à un même bloc de données. Ceci permet de tirer profit de la puissance des différents cœurs de calcul de chaque nœud de calcul mais ne garantit pas d'en faire une utilisation optimale. Cependant, comme le flot de contrôle généré pour chaque processus est simple, nous pourrions intégrer les transformations de boucles polyédriques développées par Bastoul *et al.* [16] pour l'optimisation en mémoire partagée du code généré pour les nids de boucles afin de tirer le maximum de la puissance de chaque nœud de calcul.

Une autre possibilité d'intégration avec d'autres outils est l'utilisation d'implémentations spécifiques pour les routines d'algèbre linéaire. En effet, si le code à paralléliser contient des calculs connus d'algèbre linéaire, il peut être intéressant, dans le code généré, de remplacer le calcul généré par un appel à la routine d'algèbre linéaire correspondante qui est souvent optimisée pour tout type d'architectures (exemples : PLASMA pour les CPUs et MAGMA pour les GPUs [1])

9.5.3 Extension du caractère hybride

Le caractère hybride du code généré devrait être étendu à une utilisation simultanée des CPUs et des GPUs en mémoire distribuée. Plusieurs configurations sont possibles :

1. sur une machine à mémoire distribuée, exécuter le code CPU sur certains nœuds de calcul et le code GPU sur d'autres,
2. exécuter sur chaque nœud de calcul à la fois le code CPU et le code GPU.

La seconde configuration est plus complexe à cause des problèmes liés à la cohérence des données entre la mémoire de l'hôte et celle de l'accélérateur sur le même nœud de calcul pour une séquence de nids de boucles.

9.5.4 Plus d'expériences

Nous envisageons de prouver l'efficacité de *dSTEP* sur plus de benchmarks, comme l'application LU des NAS qui présente un déséquilibre de charge. Enfin, nous voulons surtout appliquer notre solution sur davantage d'applications industrielles traitant des volumes de données de très grande taille.

Publications

[1] Pierre Fortin, Rachid Habel, Fabienne Jezequel, Jean Luc Lamotte, and N Stan Scott. Deployment on GPUs of an Application in Computational Atomic Physics. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1359–1366. IEEE, 2011.

[2] Rachid Habel, Pierre Fortin, Fabienne Jezequel, Jean Luc Lamotte, and N Stan Scott. Numerical Validation and GPU Performance in Atomic Physics. In *Designing Scientific Applications on GPUs*, 2013.

[3] Frédérique Silber-Chaussumier, Alain Muller, and Rachid Habel. Generating Data Transfers for Distributed GPU Parallel Programs. *Journal of Parallel and Distributed Computing*, 73(12) :1649–1660, 2013.

Bibliographie

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical Linear Algebra on Emerging Architectures : The PLASMA and MAGMA Projects. *Journal of Physics : Conference Series*, 180(1), 2009.

[2] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] Mehdi Amini, Corinne Ancourt, Fabien Coelho, François Irigoin, Pierre Jouvelot, Ronan Keryell, Pierre Villalon, Béatrice Creusillet, and Serge Guelton. PIPS is Not (Just) Polyhedral Software. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Chamonix, France*, 2011.

[4] Mehdi Amini, Fabien Coelho, François Irigoin, and Ronan Keryell. Static Compilation Analysis for Host-accelerator Communication Optimization. In *Languages and Compilers for Parallel Computing*, pages 237–251. Springer, 2013.

[5] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, Pierre Villalon, et al. Par4all : From Convex Array Regions to Heterogeneous Computing. In *IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, 2012.

[6] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks : Shared memory computing on networks of workstations. *Computer*, 29(2) :18–28, 1996.

[7] Corinne Ancourt, Fabien Coelho, Francois Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static High Performance Fortran Code Distribution. *Scientific Programming*, 6(1) :3–27, 1997.

[8] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 291–298. IEEE, 2010.

[9] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : a Unified Platform for Task Scheduling on Heterogeneous Mul-

ticore Architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[10] Soufiane Baghdadi, Armin Größlinger, Albert Cohen, et al. Putting Automatic Polyhedral Compilation for GPGPU to Work. In *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, 2010.

[11] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3) :63–73, 1991.

[12] Prithviraj Banerjee, John A Chandy, Manish Gupta, Eugene W Hodges IV, John G Holm, Antonio Lain, Daniel J Palermo, Shankar Ramaswamy, and Ernesto Su. The PARADIGM Compiler for Distributed-Memory Multicomputers. *Computer*, 28(10) :37–47, 1995.

[13] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code Generation for Affine Programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[14] Ayon Basumallik and Rudolf Eigenmann. Towards Automatic Translation of OpenMP to MPI. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 189–198, New York, NY, USA, 2005. ACM.

[15] Basumallik, A. and Seung-Jai Min and Eigenmann, R. Programming Distributed Memory Sytems Using OpenMP. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8, 2007.

[16] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The Polyhedral Model is More Widely Applicable Than You Think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[17] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. Grid'5000 : a Large Scale and Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4) :481–494, 2006.

[18] Dan Bonachea. GASNet Specification, V1.1. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2002.

[19] Mathias Bourgoin, Emmanuel Chailloux, and Jean-Luc Lamotte. SPOC : GPGPU Programming Through Stream Processing with OCaml. *Parallel Processing Letters*, 22(02), 2012.

[20] Mathias Bourgoin, Emmanuel Chailloux, and Jean-Luc Lamotte. Efficient Abstractions for GPGPU Programming. *International Journal of Parallel Programming*, 42(4) :583–600, 2014.

[21] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. HWLOC : A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186. IEEE, 2010.

[22] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, RosaM. Badia, Eduard Ayguade, and Jesús Labarta. Productive Cluster Programming with OmpSs. In *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 555–566. Springer Berlin Heidelberg, 2011.

[23] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, pages 359–368, New York, NY, USA, 2013. ACM.

[24] CAPS Entreprise. HMPP Codelet Generator Directives, december 2011.

[25] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3) :291–312, 2007.

[26] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10 : an Object-oriented Approach to Non-uniform Cluster Computing. *ACM SIGPLAN Notices*, 40(10) :519–538, 2005.

[27] Li Chen, Lei Liu, Shenglin Tang, Lei Huang, Zheng Jing, Shixiong Xu, Dingfei Zhang, and Baojiang Shou. Unified parallel C for GPU Clusters : Language Extensions and Compiler Implementation. In *Languages and Compilers for Parallel Computing*, pages 151–165. Springer, 2011.

[28] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley. Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5(3) :173–184, 1996.

[29] The OpenACC Consortium. The OpenACC Programming Interface. *http ://www.openacc-standard.org*.

[30] Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. In *Languages and Compilers for Parallel Computing*, volume 1033 of *Lecture Notes in Computer Science*, pages 46–60. Springer Berlin Heidelberg, 1996.

[31] Leonardo Dagum and Ramesh Menon. OpenMP : An Industry Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE*, 5(1) :46–55, 1998.

[32] Frederica Darema. The SPMD Model : Past, Present and Future. In Yiannis Cotronis and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2131 of *Lecture Notes in Computer Science*, pages 1–1. Springer Berlin Heidelberg, 2001.

[33] Roxana E. Diaconescu and Hans P. Zima. An Approach to Data Distributions in Chapel. *International Journal of High Performance Computing Applications*, 21(3) :313–335, 2007.

[34] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP : A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[35] Jack Dongarra, Thomas Sterling, Horst Simon, and Erich Strohmaier. High-Performance Computing : Clusters, Constellations, MPPs, and Future Directions. *Computing in Science & Engineering*, 7(2) :51–59, 2005.

[36] Alejandro Duarn, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs : A Proposal For Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21(02) :173–193, 2011.

[37] Ralph Duncan. A Survey of Parallel Computer Architectures. *Computer*, 23(2) :5–16, 1990.

[38] CAPS Entreprise. OpenHMPP Directives, 2012.

[39] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225. IEEE, 2011.

[40] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20, 1991.

[41] Michael Flynn. Some Computer Organizations and their Effectiveness. *Computers, IEEE Transactions on*, 100(9) :948–960, 1972.

[42] Pierre Fortin, Rachid Habel, Fabienne Jezequel, Jean Luc Lamotte, and N Stan Scott. Deployment on GPUs of an Application in Computational Atomic Physics. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1359–1366. IEEE, 2011.

[43] Michael Frumkin, Haoqiang Jin, and Jerry Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. *NAS Techinical Report NAS-98-009*, 1998.

[44] Michael Frumkin, Haoqiang Jin, and Jerry Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. *NAS Techinical Report NAS-98-009*, 1998.

[45] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[46] Jordi Garcia, Eduard Ayguadé, and Jesús Labarta. A Novel Approach Towards Automatic Data Distribution. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 78–78, 1995.

[47] William Gropp. MPICH2 : A New Start for MPI Implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, 2002.

[48] Serge Guelton, Mehdi Amini, and Béatrice Creusillet. Beyond Do Loops : Data Transfer Generation with Convex Array Regions. In *Languages and Compilers for Parallel Computing*, pages 249–263. Springer, 2013.

[49] Manish Gupta and Prithviraj Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2) :179–193, 1992.

[50] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA : A High-level Directive-based Language for GPU Programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[51] High Performance Fortran Forum (HPFF). High Performance Fortran Language Specification, 1997.

[52] W. Daniel Hillis and Guy L. Steele Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12) :1170–1183, 1986.

[53] Jay P Hoeflinger. Extending OpenMP to Clusters. *White Paper, Intel Corporation*, 2006.

[54] Sunpyo Hong and Hyesoon Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. *SIGARCH Comput. Archit. News*, 37(3) :152–163, June 2009.

[55] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical Interprocedural Parallelization : An Overview of the PIPS Project. In *Proceedings of the 5th international conference on Supercomputing*, ICS '91, pages 244–251, New York, NY, USA, 1991. ACM.

[56] Daegon Kim. *Parameterized and Multi-level Tiled Loop Generation*. PhD thesis, Colorado State University, Fort Collins, CO, USA, 2010. AAI3419053.

[57] David Kirk. NVIDIA CUDA Software and GPU Parallel Computing Architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[58] Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato. Performance Evaluation of the Omni OpenMP Compiler. In *High Performance Computing*, pages 403–414. Springer, 2000.

[59] Peizong Lee and Zvi Meir Kedem. Automatic Data and Computation Decomposition on Distributed Memory Parallel Computers. *ACM Trans. Program. Lang. Syst.*, 24(1) :1–50, January 2002.

[60] Sang-Ik Lee, Troy A Johnson, and Rudolf Eigenmann. Cetus : An Extensible Compiler Infrastructure for Source-to-source Transformation. In *Languages and Compilers for Parallel Computing*, pages 539–553. Springer, 2004.

[61] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU : a Compiler Framework for Automatic Translation and Optimization. *ACM Sigplan Notices*, 44(4) :101–110, 2009.

[62] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A Mapping Path for Multi-GPGPU

Accelerated Computers from a Portable High-level Programming Abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 51–61. ACM, 2010.

[63] Jingke Li and Marina Chen. Index Domain Alignment : Minimizing Cost of Cross-referencing Between Distributed Arrays. In *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation, 1990.*, pages 424–433. IEEE, 1990.

[64] John Mellor-Crummey, Laksono Adhianto, William N. Scherer, III, and Guohua Jin. A New Vision for Co-Array Fortran. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 5 :1–5 :9, New York, NY, USA, 2009. ACM.

[65] Mellor-Crummey, John M. and Adve, Vikram S. and Broom, Bradley and Chavarría-Miranda, Daniel G. and Fowler, Robert J. and Jin, Guohua and Kennedy, Ken and Yi, Qing. Advanced Optimization Strategies in the Rice dHPF Compiler. *Concurrency and Computation : Practice and Experience*, 14 :741–767, 2002.

[66] John Merlin, Douglas Miles, and Vincent Schuster. Distributed OMP : Extensions to OpenMP for SMP Clusters. In *Second European Workshop on OpenMP (EWOMP)*, pages 14–15, 2000.

[67] Message Passing Interface Forum. MPI : A Message-Passing Interface Standard, Version 3.0, September 2012.

[68] Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. STEP : A Distributed OpenMP for Coarse-Grain Parallelism Tool. In *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 83–99. Springer Berlin Heidelberg, 2008.

[69] Daniel Millot, Alain Muller, Christian Parrot, and Frédérique Silber-Chaussumier. From OpenMP to MPI : First Experiments of the STEP Source-to-source Transformation Tool. In *The international Parallel Computing Conference (ParCo)*, pages 669–676, 2009.

[70] Gordon E. Moore et al. Cramming More Components onto Integrated Circuits, 1965.

[71] J Carlos Mourino, María J Martín, Patricia González, and Ramón Doallo. Dynamic Load-Balancing for the STEM-II Air Quality Model. In *Computational Science and Its Applications-ICCSA 2006*, pages 701–710. Springer, 2006.

[72] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. XcalableMP Implementation and Performance of NAS Parallel Benchmarks. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 11 :1–11 :10, New York, NY, USA, 2010. ACM.

[73] Nakao, Masahiro and Lee, Jinpil and Boku, Taisuke and Sato, Mitsuhisa. Productivity and Performance of Global-View Programming with XcalableMP PGAS Language. In *12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 402–409, 2012.

[74] Jaroslaw Nieplocha, RobertJ. Harrison, and Richard J. Littlefield. Global Arrays : A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10(2) :169–189, 1996.

[75] Robert W. Numrich and John Reid. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2) :1–31, 1998.

[76] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture - CUDA C Programming Guide*, 2014.

[77] OpenMP Architecture Review Board. OpenMP Application Program Interface - Version 3.0, May 2008.

[78] OpenMP Architecture Review Board. OpenMP Application Program Interface - Version 4.0, July 2013.

[79] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Computer graphics forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[80] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, R Ramanujam, P Sadayappan, et al. Hybrid Iterative and Model-driven Optimization in the Polyhedral Model. Technical report, Inria, 2009.

[81] Louis-Noël Pouchet. PolyBoench/C, The Polyhedral Benchmark suite. http ://www.cse.ohio-state.edu/ pouchet/software/polybench, 2014.

[82] T.J. Richardson and R.L. Urbanke. The capacity of low-density parity-check codes under message-passing decoding. *Information Theory, IEEE Transactions on*, 47(2) :599–618, Feb 2001.

[83] Taher Saif and Manish Parashar. Understanding the Behavior and Performance of Non-blocking Communications in MPI. In *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 173–182. Springer Berlin Heidelberg, 2004.

[84] Joel Saltz, Kathleen Crowley, Ravi Michandaney, and Harry Berryman. Runtime Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8(4) :303 – 312, 1990.

[85] Frédérique Silber-Chaussumier, Alain Muller, and Rachid Habel. Generating Data Transfers for Distributed GPU Parallel Programs. *Journal of Parallel and Distributed Computing*, 73(12) :1649–1660, 2013.

[86] John E Stone, David Gohara, and Guochun Shi. OpenCL : A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in science & engineering*, 12(3) :66, 2010.

[87] Herb Sutter. The Free Lunch is Over : A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3) :202–210, 2005.

[88] François Trahay, Elisabeth Brunet, Alexandre Denis, and Raymond Namyst. A Multithreaded Communication Engine for Multicore Architectures. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–7, April 2008.

[89] M. Ujaldon, E.L. Zapata, B.M. Chapman, and H.P. Zima. Vienna-Fortran/HPF Extensions for Sparse and Irregular Problems and their Compilation. *Parallel and Distributed Systems, IEEE Transactions on*, 8(10) :1068–1083, Oct 1997.

[90] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[91] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8) :103–111, August 1990.

[92] Rob F. Van der Wijngaart and Parkson Wong. NAS Parallel Benchmarks Version 2.4. Technical report, NAS technical report, NAS-02-007, 2002.

[93] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4) :54, 2013.

[94] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages : A Mechanism for Integrating Communication and Computation. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pages 430–440, New York, NY, USA, 1998. ACM.

[95] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC—first Experiences with Real-world Applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

[96] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The Potential of the Cell Processor for Scientific Computing. In *Proceedings of the 3rd conference on Computing frontiers*, pages 9–20. ACM, 2006.

[97] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 24–32. ACM, 2007.

[98] Tomofumi Yuki and Sanjay Rajopadhye. Parametrically Tiled Distributed Memory Parallelization of Polyhedral Programs. Technical report, Colorado State University Technical Report CS13-105, June 2013.

Programmation Haute Performance pour Architectures Hybrides

Résumé :
Les architectures parallèles hybrides constituées d'un grand nombre de nœuds de calcul multi-cœurs et GPUs connectés en réseau offrent des performances théoriques très élevées, de l'ordre de quelque dizaines de TeraFlops. Mais la programmation efficace de ces machines reste un défi à cause de la complexité de l'architecture et de la multiplication des modèles de programmation utilisés. L'objectif de cette thèse est d'améliorer la programmation des applications scientifiques denses sur les architectures parallèles hybrides selon trois axes : réduction des temps d'exécution, traitement de données de très grande taille et facilité de programmation. Nous avons pour cela proposé un modèle de programmation à base de directives appelé dSTEP pour exprimer à la fois la distribution des données et des calculs. Dans ce modèle, plusieurs types de distribution de données sont exprimables de façon unifiée à l'aide d'une directive "dstep distribute" et une réplication de certains éléments distribués peut être exprimée par un "halo". La directive "dstep gridify" exprime à la fois la distribution des calculs ainsi que leurs contraintes d'ordonnancement. Nous avons ensuite défini un modèle de distribution et montré la correction de la transformation de code du domaine séquentiel au domaine distribué. À partir du modèle de distribution, nous avons dérivé un schéma de compilation pour la transformation de programmes annotés de directives dSTEP en des programmes parallèles hybrides. Nous avons implémenté notre solution sous la forme d'un compilateur intégré à la plateforme de compilation PIPS ainsi qu'une bibliothèque fournissant les fonctionnalités du support d'exécution, notamment les communications. Notre solution a été validée sur des programmes de calcul scientifiques standards tirés des NAS Parallel Benchmarks et des Polybenchs ainsi que sur une application industrielle.

Mots clés : Mémoire distribuée, Mémoire partagée, Accélérateurs, Compilation, MPI, OpenMP, GPU

High-Performance Programming for Hybrid Architectures

Abstract:
Clusters of multicore/GPU nodes connected with a fast network offer very high therotical peak performances, reaching tens of TeraFlops. Unfortunately, the efficient programing of such architectures remains challenging because of their complexity and the diversity of the existing programming models. The purpose of this thesis is to improve the programmability of dense scientific applications on hybrid architectures in three ways: reducing the execution times, processing larger data sets and reducing the programming effort. We propose dSTEP, a directive-based programming model expressing both data and computation distribution. A large set of distribution types are unified in a "dstep distribute" directive and the replication of some distributed elements can be expressed using a "halo". The "dstep gridify" directive expresses both the computation distribution and the schedule constraints of loop iterations. We define a distribution model and demonstrate the correctness of the code transformation from the sequential domain to the parallel domain. From the distribution model, we derive a generic compilation scheme transforming dSTEP annotated input programs into parallel hybrid ones. We have implemented such a tool as a compiler integrated to the PIPS compilation workbench together with a library offering the runtime functionality, especially the communication. Our solution is validated on scientific programs from the NAS Parallel Benchmarks and the PolyBenchs as well as on an industrial signal procesing application.

Keywords: Distributed-memory, Shared-memory, Hardware Accelerators, Compilation, MPI, OpenMP, GPU

MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY