Modèles de programmation des applications de traitement du signal et de l'image sur cluster parallèle et hétérogène

Farouk Mansouri

To cite this version:

Farouk Mansouri. Modèles de programmation des applications de traitement du signal et de l'image sur cluster parallèle et hétérogène. Traitement du signal et de l'image. Université Grenoble Alpes, 2015. Français. <NNT : 2015GREAT063>. <tel-01224338>

HAL Id: tel-01224338
https://tel.archives-ouvertes.fr/tel-01224338

Submitted on 4 Nov 2015

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

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

UNIVERSITÉ GRENOBLE ALPES

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE ALPES

Spécialité : **Signal, Image, Parole, Télécoms**

Arrêté ministériel : 7 août 2006

Présentée par
Farouk Mansouri

Thèse dirigée par **Dominique Houzet** et
co-encadré par **Sylvain Huet**

préparée au sein du laboratoire
Grenoble Image Parole Signal Automatique (Gipsa-Lab)
dans **l'école doctorale électronique électrotechnique automatique et traitement du signal (EEATS)**

Modèle de programmation des applications de traitement du signal et de l'image sur cluster parallèle et hétérogène

Thèse soutenue publiquement le **14/10/2015**,
devant le jury composé de :

Emmanuel Jeannot
INRIA Bordeaux, Rapporteur
Lionel Lacassagne
LRI Paris sud, Rapporteur
Tanguy RISSET
IRISA Lyon, Examinateur, Président du jury
Thierry Gautier
INRIA Rhône-Alpes, Examinateur
Dominique Houzet
GIPSA, Directeur de thèse
Sylvain Huet
GIPSA, Encadrant de thèse

UNIVERSITÉ DE GRENOBLE ALPES

ÉCOLE DOCTORALE EEATS

École doctorale électronique électrotechnique automatique et traitement du signal

T H È S E

pour obtenir le titre de

docteur en sciences

de l'Université de Grenoble Alpes

Mention : Signal, Image, Parole, Télécoms

Présentée et soutenue par

Farouk Mansouri

Modèle de programmation des applications de traitement du signal et de l'image sur cluster parallèle et hétérogène

Thèse dirigée par Dominique Houzet

préparée au Laboratoire Grenoble image parole signal automatique Gipsa-Lab

soutenue le 14/10/2015

Jury :

Rapporteurs :	Emmanuel Jeannot	-	INRIA Bordeaux
	Lionel Lacassagne	-	LRI Paris sud
Directeur :	Dominique Houzet	-	Gipsa Grenoble
Encadrant :	Sylvain Huet	-	Gipsa Grenoble
Président :	Tanguy RISSET	-	IRISA Rhône-Alpes
Examinateur :	Tanguy RISSET	-	IRISA Lyon
	Thierry Gautier	-	INRIA Lyon

Remerciements

Cette thèse de doctorat a été réalisée au sein de l'équipe Adéquation Algorithme Architecture (AAA) du laboratoire «Gipsa-Lab» de Grenoble.

Je remercie tout d'abord Dieu mon seigneur et mon créateur.

Je tiens à exprimer ma sincère reconnaissance à Dominique Houzet et Sylvain Huet d'avoir dirigé et encadré cette thèse. La qualité de leur encadrement m'a permis d'entreprendre des travaux de recherche enrichissants et motivants. Ce fut réellement une chance et un plaisir de travailler avec eux.

Je remercie ma chère épouse pour sa confiance, sa patience et son soutien permanent. A travers elle, j'ai pu trouver la force d'entamer puis d'achever cette expérience. Ses sacrifices et sa force m'ont montré combien une femme peut apporter à un homme.

Je remercie mes parents pour leurs encouragements réguliers. Djamel a été pour moi un exemple de persévérance et de courage. Fatima a été ma confidente et mon support.

Je remercie mes beaux parents d'avoir cru en moi et de m'avoir considéré comme leur propre fils. Dr Ameur a été une forte inspiration pour moi. Saida de part sa gentillesse et sa bienveillance est pour moi une seconde maman.

Je remercie Leila et Stéphane pour leur accueil, puis pour leur soutien sans lequel cette aventure aurait été improbable. Petite dédicace à la petite Eva qui a vu le jour durant cette période.

Je remercie Amina et Said qui m'ont épaulé et pensé à moi dans des moments difficiles. Dans le futur, j'encouragerai à mon tour les petits Adam et Anis à poursuivre un parcours équivalent.

Je remercie Nadia pour ses interventions linguistiques et sa gentillesse. Grâce à elle, je me suis rapproché d'avantage de la langue de Molière.

A mes frères Ilyes, Chakib et Raken je transmets un grand merci d'avoir été présents pour me faire rire et me changer les idées quand j'en avais besoin.

Enfin, je voudrais simplement saluer collègues, amis et familles qui m'ont entouré et soutenu durant ces trois années.

Table des matières

Table des sigles et acronymes **xiii**

Introduction **1**

I Contexte et motivations **5**

1 Les architectures parallèles et hétérogènes **9**

1.1 Introduction 9

1.2 Les architectures parallèles et hétérogènes 10

1.3 Programmation des architectures parallèles et hétérogènes 16

1.4 Conclusion 18

2 Les modèles de programmation pour les architectures parallèles et hétérogènes **19**

2.1 Introduction 19

2.2 Modèles de programmation parallèles et propriétés 19

2.3 Classification des modèles de programmation 22

2.4 Les environnements de programmation parallèles et hétérogènes 23

2.5 Discussion 26

3 Les applications DSP **29**

3.1 Introduction sur les applications DSP 29

3.2 Modélisation des applications DSP 30

3.3 Le modèle de calcul graphe de flot de donnée 32

3.4 Les environnements de développement pour les applications DSP spécifiées avec le modèle de calcul graphe flot de données 35

3.5 Implémentations d'applications de type DSP sur architectures parallèles et hétérogènes 37

II Mig-PACCO : La migration de tâches dans un modèle de programmation BSP avec Pipeline 45

4 Les modèles BSP et PACCO 49

4.1 Introduction 49

4.2 Le modèle BSP 49

4.3 Environnements d'implémentation basés sur le modèle BSP 51

4.4 PACCO 52

4.5 Conclusion 60

5 La migration de tâches dans PACCO 61

5.1 Contexte et motivations 61

5.2 Stratégies de migration 63

5.3 Implémentation 64

5.4 Validation 68

5.5 Conclusion 74

III SignalPU 77

6 SignalPU : Un modèle de programmation DFG basé sur StarPU 81

6.1 Introduction 81

6.2 L'environnement StarPU 81

6.3 SignalPU 87

6.4 Comparaison et discussion 95

6.5 Conclusion 96

7 Validation **97**

7.1 Introduction 97

7.2 Opérateurs de charge paramétrables pour la description d'applications synthetiques 97

7.3 Implémentations 99

7.4 Expériences et résultats 101

7.5 Conclusion 107

IV Expérimentations et comparaisons **109**

8 L'application de saillance **113**

8.1 Introduction 113

8.2 Description de l'application et des implémentations 114

8.3 Expérimentations et résultats 116

8.4 Conclusion 121

9 L'application de suivi **123**

9.1 Introduction 123

9.2 Application de suivi et implémentations 123

9.3 Expérimentations et résultats 126

9.4 Conclusion 128

Conclusion **131**

9.5 Conclusion 131

9.6 Perspectives 133

A Exemples de différents programmes pour l'implémentation d'une addition de vecteurs **135**

B Codes de l'application synthétique **139**

Bibliographie de l'auteur **145**

Bibliographie **151**

Table des figures

1.1 Illustration du gain de performance du à la combinaison de la loi de Moore et de la réduction de Dennard. Cette combinaison prédit qu'en multipliant le nombre de transistors par 2 et leur vitesse par 1.4 et en réduisant leur capacité et leur voltage de 0.7, il est possible de produire un gain de 2.8 sans augmenter la consommation énergétique . . . 9

1.2 Évolution des caractéristiques des machines de calcul entre 1975 et 2015 . . . 10

1.3 Exemple d'un cluster hétérogène . . . 11

1.4 Architecture Multiprocesseurs ccNUMA incluant des CPU Multi-cœurs . . . 12

1.5 Architecture multiprocesseurs à accès mémoire symétrique . . . 13

1.6 Architecture multiprocesseurs à accès mémoire non uniforme . . . 13

1.7 Schéma illustrant les caractéristiques générales des accélérateurs . . . 14

1.8 Digramme des principaux blocs du GPU Kepler GK110 . . . 15

1.9 Digramme des principaux blocs composants le Xeon phi . . . 16

1.10 Digramme des principaux blocs composant le FPGA . . . 16

2.1 Représentation du modèle de programmation sous forme de pont reliant les applications et les architectures. . . . 20

2.2 Classification des environnements de programmation parallèle et hétérogène . . . 23

2.3 Tableau comparatif des différents environnements de programmation parallèle sur cluster hétérogène . . . 28

3.1 Chaine de traitement numérique du signal . . . 29

3.2 Traitement par flux de données . . . 30

3.3 Traitement par blocs de données . . . 30

3.4 Représentation de l'équation $y_{(n)} = a * x_{(n)} + b * x_{(n-1)} + c * x_{(n-2)}$ à l'aide d'un schéma de blocs fonctionnels . . . 31

3.5 Représentation par graphe de flot de données . . . 31

3.6 Représentation de différents modèles de calcul graphe de flot de données . . . 34

3.7 Les modèles de calcul à réseau de processus 35

4.1 Illustration des étapes de calcul du modèle BSP 50

4.2 Flot de conception du modèle PACCO 53

4.3 Graphe d'architecture 54

4.4 Graphe d'application 55

4.5 "Mapping" entre graphe d'application et graphe d'architecture 55

4.6 Illustration des étapes de traitement et de la conception de l'environnement PACCO. Les parties en gris représentent les points d'entrée 56

4.7 Illustration de l'insertion des buffers mémoire dans le graphe d'implémentation 58

4.8 Illustration de l'optimisation des allocations dans le cas de l'insertion des buffers mémoire du graphe d'implémentation de la figure 4.5 59

5.1 Illustration du processus de migration de tâches (acteur) dans l'environnement PACCO 62

5.2 Illustration de la migration d'une tâche dans une application de type "streaming" 64

5.3 Illustration de l'exécution de la migration au sein de la super étape dans un mode d'exécution avec recouvrement calcul-communication. Le Thread de synchronisation met à jour le graphe d'implémentation (GI) 65

5.4 Illustration du graphe d'implémentation mis à jour dans le cas de migration d'un acteur avec chemin originel et de migration de tailles égales. Le couple sur chaque arrête représente le nombre d'itérations à partir duquel l'arc sera actif pour l'élément de gauche et le nombre d'itérations à partir duquel l'arc sera inactif pour l'element de droite, cela relativement à l'itération de déclenchement de la migration. 66

5.5 Illustration du graphe d'implémentation mis à jour dans le cas de la migration d'un acteur avec chemin originel plus long que le chemin de migration. Le couple sur chaque arrête représente le temps en nombre d'itérations de début pour l'indice de gauche et de fin pour l'indice de droite, relativement à l'itération pendant laquelle la demande de migration à eu lieu 67

5.6 Illustration du graphe d'implémentation mis à jour dans le cas de migration d'un acteur avec chemin originel moins long en nombre de buffers que le chemin de migration. Le couple sur chaque arrêtes représente le temps en nombre d'itération de début pour l'indice de gauche et de fin pour l'indice de droite, relativement à l'itération pendant laquelle la demande de migration à eu lieu 68

5.7 Illustration de la mise à jour du graphe d'implémentation de l'application de saillance dans un contexte de migration avec chemin plus long 70

5.8 Illustration de l'augmentation du débit de production des images de sortie . . . 71

5.9 Illustration de la mise à jour du graphe d'implémentation de l'application de saillance dans un contexte de migration avec chemins égaux 72

5.10 Illustration de la stabilisation du débit de production des images de sortie dans un scénario de migration pour la libération d'un élément de calcul 73

5.11 Illustration de la mise à jour du graphe d'implémentation de l'application de saillance dans un contexte de migration avec chemin moins long 74

5.12 Illustration de la préservation du débit de production des images de sortie dans un scénario de migration pour la réduction de la consommation d'énergie 75

6.1 Illustration des principales composantes de StarPU 82

6.2 Positionnement de SignalPU en terme de niveau d'abstraction et de couverture d'architecture 88

6.3 Conception et composants de SignalPU 88

6.4 DFG représentant l'algorithme 3 89

6.5 Transformation du DFG de l'algorithme 3 vers un DAG de tâches exploitant différents niveaux de parallélisme 91

6.6 Troisième niveau : exécution efficace des tâches grâce au support exécutif StarPU 93

7.1 Illustration du DFG-XML de l'application synthétique 100

7.2 Comparaison des résultats de performances globales des implémentations SignalPU Vs MPI+OpenMP+CUDA. Les temps sont donnés en minute pour le traitement de 1000 images. 103

7.3 Comparaison des résultats de performances globales des implémentations SignalPU Vs MPI+OpenMP+CUDA. Les temps sont données en secondes pour le traitement de 1000 images. 104

7.4 Chronogramme représentant l'exécution de l'implémentation "Scheduling" dynamique 105

7.5 Chronogramme représentant l'exécution de l'implémentation "Scheduling" statique 105

7.6 Chronogramme représentant l'exécution de l'implémentation "Scheduling" statique avec dépliement du graphe . . . 105

7.7 Chronogramme représentant l'exécution de l'implémentation "Scheduling" dynamique avec dépliement du graphe . . . 106

7.8 Evolution du pourcentage des surcoûts dans le traitement d'un nombre d'images allant de 1000 jusqu'à 10000. . . . 107

7.9 Apport de la persistance des initialisations et de l'auto-tuning sur la performance des tâches . . . 107

8.1 Illustration des étapes du modèle de calcul de cartes de saillance visuelle proposé par [RHP11] . . . 113

8.2 Illustration du graphe d'implémentation de l'application de calcul de cartes de saillance avec l'environnement PACCO . . . 115

8.3 Illustration du DFG-XML de l'application de calcul de cartes de saillance avec l'environnement SignalPU . . . 116

8.4 Comparaison des résultats des performances globales SignalPU Vs PACCO . . . 117

8.5 Comparaison des performances Scheduling dynamique Vs Scheduling statique d'une implémentation SignalPU . . . 119

8.6 Évolution des surcoûts dans les implémentations SignalPU et PACCO . . . 120

8.7 Évolution des performances des tâches dans une implémentation SignalPU . . . 121

9.1 Illustration des étapes de traitement d'une vidéo pour le suivi de personne . . . 123

9.2 XML-DFG modélisant l'application de tracking dans SignalPU. Seules les dépendances de données sont présentes . . . 126

9.3 Dépliement du graphe (J=4) et ajout des dépendances d'exécution additionnelles (arcs bleu) sur le graphe d'application de l'application de suivi . . . 126

9.4 Performances globales de SignalPU vs PACCO sur différentes architectures . . . 127

9.5 Évolution du FPS et du temps de non utilisation en faisant varier le degré de dépliement du graphe, "J", dans SignalPU. L'architecture utilisée est composée de 4 Cœurs CPUs . . . 128

Liste des tableaux

1.1 Taxonomie de Flynn 11

7.1 Architecture du cluster utilisé pour les expériences 102

Table des sigles et acronymes

DSP	*Digital Signal Processing*
DFG	*Data Flow Graph*
DDF	*Dynamic Data Flow*
DAG	*Directional Acyclic Graph*
SDF	*Synchronous Data Flow*
MdPP	*Modèle de Programmation Parallèle*
MdC	*Modèle de Calcul*
MdE	*Modèle d'Exécution*
FSM	*Finite-State Machine*

Introduction générale

L'évolution des machines de calcul tend vers des architectures parallèles et hétérogènes. Leur topologie composée de plusieurs nœuds connectés incluant des unités de traitement de différents types leur permet de produire de grandes performances. En effet, à travers les clusters, plusieurs niveaux de parallélisme peuvent être exploités comme le parallélisme de données, de tâches, ou d'instructions. Cependant, pour programmer ces machines, l'utilisateur doit faire face à plusieurs difficultés compliquant ainsi leur exploitation. Le programmeur doit décomposer son algorithme en parties pouvant être traitées parallèlement, l'objectif étant de faire apparaître du parallélisme de données ou de tâches. L'utilisateur doit également gérer les communications et synchronisations entre les processus parallèles. Pour cela, il doit manipuler plusieurs environnements dépendant du ou des supports de connexion : réseau, mémoire partagée, etc. Finalement, le programmeur doit s'occuper de placer les tâches ou les processus sur les différents éléments de calcul. Il fait alors face à des problèmes de répartition des charges sur une architecture hétérogène en tenant compte des mouvements des données. Outre ces contraintes, pour augmenter les performances de calcul, le programmeur peut être amené à effectuer plusieurs optimisations comme le recouvrement calcul-communication, ou la limitation des synchronisations.

Pour s'acquitter aisément de toutes ces tâches, le programmeur peut utiliser des modèles de programmation offrant des abstractions de la machine et facilitant son utilisation. Ces modèles permettent d'augmenter la productivité en cachant toutes ou une partie des étapes citées précédemment. Ces modèles de programmation doivent également permettre d'exploiter efficacement l'architecture. Par conséquent, ils doivent apporter à la fois productivité et efficacité. Ces objectifs sont en contradiction et les satisfaire les deux en même temps est une mission difficile. Un modèle à bas niveau d'abstraction permet un grand contrôle de la machine. Le programmeur peut optimiser finement son programme en utilisant les fonctionnalités du modèle. Cependant, cela réduit sa productivité car il doit gérer une partie ou toutes les étapes d'implémentation. Contrairement à ça, un modèle à haut niveau d'abstraction facilite la tâche du programmeur dans la gestion de : la décomposition en parties parallèles, les communications, les synchronisations et le placement, ce qui augmente sa productivité. Cependant, son implémentation peut perdre en efficacité comparée à celles comportant des spécifications bas niveaux. Plus simplement, la problématique peut être exprimée comme suit : comment proposer au programmeur un modèle de programmation permettant de faciliter les étapes d'implémentation et, en même temps, garantir une efficacité élevée ?

Pour répondre à cette problématique, nous proposons dans cette thèse d'exploiter l'idée qu'un modèle de programmation spécifique à un domaine applicatif offre des abstractions de haut niveaux, adaptées et efficaces en même temps. En effet, il est possible en caractérisant une certaine famille d'applications de produire un modèle de programmation leur correspondant avec des fonctionnalités de haut niveau. Dans cette démarche, nous proposons deux contributions principales avec deux modèles de programmation PACCO et SignalPU, spécifiques à l'implémentation d'applications du traitement du signal et de l'image. Dans ces modèles de

programmation, les applications de type Digital Signal Processing (DSP) sont caractérisées par le modèle de calcul à graphe de flot de données. A partir de cette description, ces modèles permettent d'implémenter facilement et efficacement des applications de type DSP sur un cluster hétérogène CPU-GPU. Le programmeur décompose implicitement son applications avec un Data-Flow Graph (DFG) et ne gère pas les communications et les synchronisations. Cependant, dans le premier modèle PACCO, comme dans d'autres environnements équivalents de la littérature discutés plus loin, le programmeur doit représenter l'architecture de calcul et associer manuellement et statiquement les tâches aux éléments de calcul. Ce placement statique convient à l'execution des applications statiques i.e. avec des temps de traitement et de communication stable, sur des architectures homogènes. Mais, il est non adapté à l'optimisation des applications dynamiques i.e. avec des charges de calcul variables sur des architectures hétérogènes. En outre, un placement statique s'il est manuel, force le programmeur à connaître les spécificités de l'architecture et implique un reparamétrage pour chaque architecture de calcul, ce qui réduit sa portabilité. Pour palier à cela, notre première contribution est :

Enrichir le modèle PACCO avec une fonctionnalité de migration de tâches en ligne. Cette fonctionnalité peut être utilisée dans différents scénarios :

1. Migration pour la répartition dynamique des charges de calcul.
2. Migration pour la réduction de la consommation d'énergie.
3. Migration pour la libération d'élément de calcul.

Cette contribution permet ainsi d'augmenter sa performance via un meilleur équilibrage des charges sur les éléments de calcul hétérogènes. Elle permet également d'augmenter son niveaux d'abstraction dans le sens ou l'utilisateur n'est plus chargé d'optimiser les placements. Cependant, l'environnement PACCO reste limité et nécessite du programmeur de connaître et de représenter l'architecture de calcul. En outre, du point de vu des performances de son modèle d'exécution, l'environnement est synchronisé à l'aide de barrière globale suivant le modèle BSP, ce qui retarde les calculs des itérations successives. Ces limitations sont résolues dans notre deuxième contribution principale : SignalPU.

Il est basé sur la combinaison du modèle de calcul graphe de flot de donnée et le modèle d'exécution dynamique StarPU que nous enrichissons avec plusieurs fonctionnalités adaptées aux applications de type DSP énumérées comme suit :

1. Construction dynamique du Directional Acyclic Graph (DAG) de tâches.
2. Dépliement (unfolding) automatique du graphe d'application DFG.
3. Allocation et réutilisation implicites des buffers mémoire.
4. Distribution implicite des calcul sur nœuds MPI.
5. Préservation des parties d'initialisation des tâches.
6. Auto-tuning dynamique des kernel GPU.

Cette conception permet d'une part d'offrir une couche d'abstraction supplémentaire par rapport à d'autres environnements basés sur les DFG comme PACCO. En effet, le placement

statique est remplacé par une distribution dynamique des tâches. En outre, le programmeur n'a pas besoin de représenter l'architecture de calcul, ce qui permet une grande portabilité du code. D'autre part, les performances de l'architecture sont mieux exploitées grâce à un support exécutif synchronisé sur les dépendances de données et comportant plusieurs fonctionnalités d'optimisation.

Le présent manuscrit comporte quatre parties.

Dans la première partie (partie I), nous présentons dans le chapitre 1 les architectures de type clusters parallèles et hétérogènes. Nous présentons certains éléments de calcul susceptibles de les composer et discutons de leurs caractéristiques matérielles. Dans le chapitre 2, nous présentons les modèles de programmation utilisés pour faciliter l'implémentation sur cluster. Nous les classons selon différentes métriques d'évaluation et terminons ce chapitre par les environnements d'implémentation sur architectures parallèles et hétérogènes et les comparons. Nous consacrons le chapitre 3 à la famille applicative à laquelle nous nous intéressons ainsi qu'à ses caractéristiques et les modèles de calcul usuellement utilisés pour la modéliser. Nous présentons à la fin de ce chapitre un état de l'art sur l'implémentation des applications de type DSP sur architectures parallèles et hétérogènes.

Dans la deuxième partie (partie II), nous présentons dans deux chapitres un premier modèle PACCO développé pour l'implémentation d'applications DSP ainsi que notre contribution à ce modèle. Dans le chapitre 4, nous décrivons en premier le modèle BSP utilisé comme support exécutif dans PACCO. Dans le chapitre 5, nous proposons une approche de migration de tâches et décrivons sa conception et son implémentation. Ensuite, nous validons cette approche sur une application du monde réel dans trois scénarios différents : migration pour la répartition dynamique des charges de calcul, migration pour la réduction de la consommation d'énergie, migration pour la libération d'un élément de calcul.

La troisième partie (partie III) est consacrée au modèle de programmation spécifique aux applications de type DSP : SignalPU qui constitue notre contribution principale. Ce modèle est basé sur une combinaison du modèle de calcul DFG synchrone à taux unique et du modèle d'exécution StarPU auquel nous rajoutons des fonctionnalités spécifiques. Cette partie est également composée de deux chapitres. Dans le chapitre 6, nous introduisons en premier StarPU et décrivons son utilisation et son module d'exécution sur un exemple d'implémentation d'une application DSP. Ensuite, nous présentons notre approche SignalPU et ses différentes fonctionnalités. Finalement, nous comparons les implémentations des deux modèles pour distinguer les apports de SignalPU face à StarPU. Dans le chapitre 7, nous présentons une validation de SignalPU sur une application synthétique. Nous introduisons en premier une bibliothèque d'opérateurs de charge permettant de modéliser des applications DSP "synthétiques" avec des charges de calcul et de communication variables. Nous implémentons avec celle-ci une application synthétique que nous comparons en terme d'expressivité et d'abstraction avec une implémentation basée sur MPI+OpenMP+CUDA. Nous comparons ensuite les exécutions et les performances des deux implémentations.

Dans la quatrième et dernière partie (partie IV), nous validons notre modèle de programmation SignalPU sur des applications de traitement d'image du monde réel. Dans le chapitre

8, nous présentons une comparaison des implémentations PACCO et SignalPU d'une application de calcul de carte de Saillance visuelle. Le chapitre 9 contient lui la validation à travers une comparaison de PACCO et SignaPU sur l'implémentation d'une application de suivi de personne dans une vidéo avec une contrainte d'ordre de production des données de sortie.

Première partie

Contexte et motivations

Cette partie introduit les principaux aspects étudiés dans cette thèse. Nous présentons en premier dans le chapitre 1 les architectures parallèles et hétérogènes. Nous décrivons leur caractéristiques matérielles et comment bien exploiter leurs unités de calcul. Ensuite, dans le chapitre 2, nous présentons les modèles de programmation proposés pour faciliter l'implémentation sur ces architectures, nous les classifions et discutons de leur intérêt à travers des métriques d'évaluation. Nous présentons ensuite quelques environnements de programmation illustrant ces modèles. Finalement, dans le chapitre 3, nous présentons les applications de traitement de l'image et du signal et leur besoins en puissance de calcul. Nous introduisons les modèles de calcul adaptés à leurs implémentation en détaillant plus particulièrement les modèles DFG adaptés aux algorithmes DSP. Nous présentons ensuite quelques environnements de programmation basés sur ce modèle. La fin de ce chapitre est consacrée à état de l'art de l'implémentation des applications DSP sur les architectures parallèles et hétérogènes est donné. Nous discutons les avantages et inconvénients des solutions existantes et présentons nos contributions.

Chapitre 1

Les architectures parallèles et hétérogènes

1.1 Introduction

Depuis une dizaine d'années, l'évolution des architectures de calcul tend vers plus de parallélisme et d'hétérogénéité. En effet, à cause de limitations physiques (matérielles), comme l'effet de Joule et la consommation d'énergie, il n'est plus possible d'augmenter les performances des machines de calcul en augmentant simplement leur fréquences d'horloge et leur nombre de transistors comme le prédisaient la loi de de Moore et la réduction de Dennard illustrées dans la figure 1.1. Elles avaient permis un gain de puissance de calcul de deux fois tous les 18 mois sur près de 40 ans jusqu'aux années 2000. Dés lors, les constructeurs se sont orientés vers le parallélisme en exploitant la seule loi qui reste vérifiée, la loi de Moore. Les architectures incluant plusieurs unités de calcul indépendantes voient le jour, ouvrant ainsi l'aire du multi-cœurs et du many-cœurs.

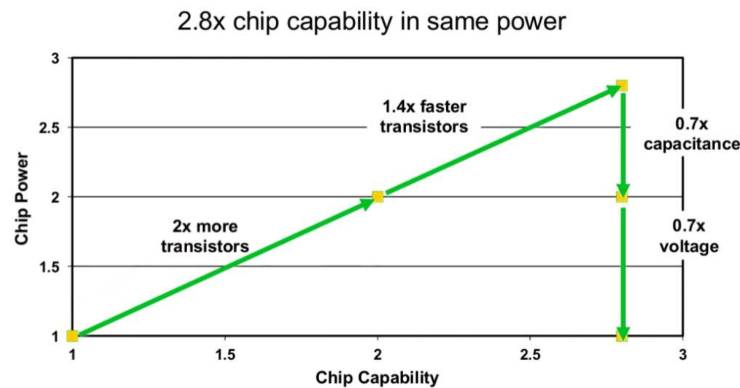

Figure 1.1 – Illustration du gain de performance du à la combinaison de la loi de Moore et de la réduction de Dennard. Cette combinaison prédit qu'en multipliant le nombre de transistors par 2 et leur vitesse par 1.4 et en réduisant leur capacité et leur voltage de 0.7, il est possible de produire un gain de 2.8 sans augmenter la consommation énergétique

Cette évolution vers le parallélisme illustrée dans la figure 1.2, touche toutes les machines de calcul, en partant du simple ordinateur de bureau comportant un processeur généraliste et un accélérateur graphique, jusqu'aux super-ordinateurs incluant des milliers d'unités de

calcul avec différentes caractéristiques. Elles peuvent traiter chacune un calcul spécifique avec de grandes performances. Ce type d'architecture hybride devient très vite une solution efficace et accessible à un large public, notamment, pour le calcul scientifique avec plusieurs domaines d'application comme la simulation numérique de phénomènes complexes, les calculs algébriques à grande taille, le traitement des images et des signaux volumineux ... En effet, il est possible aujourd'hui moyennant quelques milliers d'euro, de construire une grille de calcul composée de nœuds hétérogènes pouvant atteindre des performances importantes. Dans ce chapitre nous présentons ces architectures parallèles, leurs composants les plus utilisés et comment les programmer pour tirer avantages de leurs caractéristiques spécifiques.

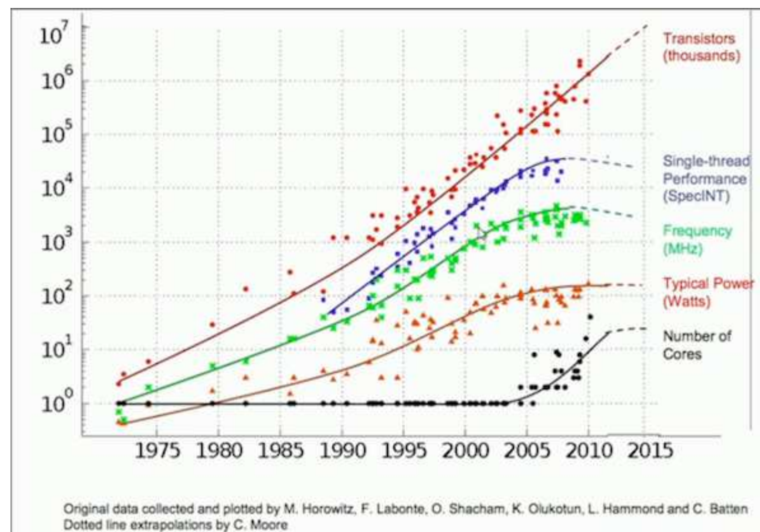

FIGURE 1.2 – Évolution des caractéristiques des machines de calcul entre 1975 et 2015

1.2 Les architectures parallèles et hétérogènes

Les architectures parallèles et hétérogènes, appelées "cluster", sont des structures de calcul composées de plusieurs nœuds connectés entre eux par un réseau à haut débit. Chaque nœud contient un ou plusieurs processeurs généralistes épaulés par un ou plusieurs processeurs dédiées (accélérateurs) que nous présentons dans les sous sections 1.2.1 et 1.2.2. Grâce à leur topologie parallèle, ils sont capables de produire de grandes performances en distribuant les calculs sur plusieurs niveaux de parallélisme :

1. Niveau des nœuds de calcul : distribuer le calcul sur plusieurs machines composants le "cluster" et interagissant à travers le réseau.
2. Niveau des unités de calcul : distribuer les calculs sur les unités de calcul au sein d'un même nœud et interagissant à travers la mémoire centrale du nœud.
3. Niveau des cœurs de calcul : distribuer les calculs sur les différents cœurs composant l'unité de calcul.

	Single Data	Multiple Data
Single instruction	SISD	SIMD
Multiple instructions	MISD	MIMD

TABLE 1.1 – Taxonomie de Flynn

Ces architectures peuvent être classifiées selon la taxonomie de Flynn [Fly72] donnée dans le tableau 1.1. Elle caractérise la relation entre le flux d'instructions à exécuter et le flux de données à traiter. Ces 4 types d'architecture sont : SISD pour "Single Instruction Single Data" représentant les processeurs séquentiels classiques qui exécutent à un moment donné une seule instruction sur une seule unité de donnée. MISD pour "Multiple Instruction Single Data" est une architecture peu répandue, les architectures en pipeline peuvent toutefois être considérées comme étant de ce type. Dans l'architecture SIMD ("Single Instruction Multiple Data"), appelée aussi architecture vectorielle ou architecture à parallélisme de donnée, plusieurs unités de données sont traitées par une seule instruction. Peu de machines sont considérées comme étant purement vectorielle comme le Crey-1 ou le Hitashi-s3600. Cependant, nous retrouvons des unité de traitement de ce type dans les dernières générations de processeurs généralistes que nous présenterons plus loin. En outre, plusieurs accélérateurs comme le GPU ou le Xeon Phi sont considérés comme appartenant à cette classe. L'architecture MIMD pour "Multiple Instruction Multiple Data", permet quand à elle de traiter plusieurs unités de données avec plusieurs différentes instructions, comme l'est le cas avec les "clusters" présentés dans cette section.

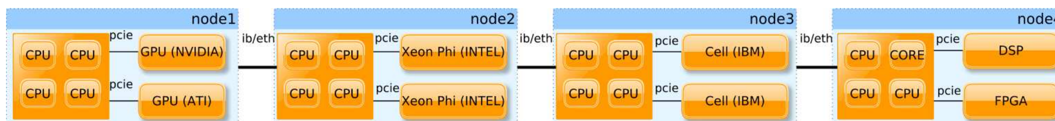

FIGURE 1.3 – Exemple d'un cluster hétérogène

En outre, l'hétérogénéité des unités de calcul dans les cluster leur permet de cibler plus efficacement différentes parties de l'application selon leur caractéristiques. En effet, chaque type de processeur présente des configurations matérielles internes différentes comme la vitesse de calcul, ou la vitesse de transfert ...etc, détaillés dans la sous-section 1.2.2. Ces caractéristiques spécifiques les rendent plus performants pour un certain type de calcul, par example, les processeurs graphiques pour le traitement des images et des vidéos. Dans la figure 1.3, nous présentons un exemple de "cluster" hétérogène comportant 4 nœuds de calcul connectés entre eux à travers des liaisons réseaux (Infiniband, Ethernet). Chaque nœud comporte un processeurs généraliste CPU, une mémoire centrale appelée mémoire hôte et deux accélérateurs connectés via le bus PCI Express. Dans ce qui suit, nous décrivons avec plus de détails les différentes unités de calcul susceptibles de composer les "cluster" et étudions leurs caractéristiques matérielles.

1.2.1 Processeurs généralistes

Communément appelés CPU pour "Central Processing Unit", ce sont des unités de calcul pouvant réaliser tout type de calcul. Grâce à leur architecture de type Von Neumann facilitant leur programmation, ils sont destinés aussi bien au marché grand public qu'à des utilisateurs professionnels. Dans la figure 1.4 nous présentons un schéma de cette architecture.

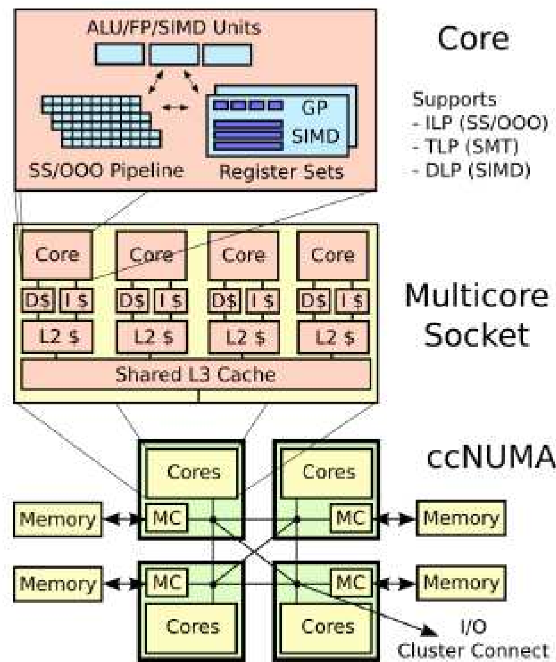

FIGURE 1.4 – Architecture Multiprocesseurs ccNUMA incluant des CPU Multi-cœurs

Un CPU est composé classiquement d'une unité arithmétique et logique (UAL), d'une unité de contrôle, de plusieurs registres à fonctions diverses (registre d'instruction, compteurs, accumulateurs...), d'une unité de gestion des entrées et de sorties et d'une horloge qui synchronise tous ces composants. Au fils du temps d'autre composants ont été rajoutés afin d'augmenter la performance du processeur en exploitant du parallélisme au niveau de l'instruction (ILP). Les plus connus d'entre eux sont : la super-scalarité (plusieurs UALs), le pipeline, l'unité de prédiction de branchement, les unités de calcul à virgules flottantes (FPU), plusieurs niveaux de caches de données et d'instructions (L1,L2,L3,L4) pour réduire la latence d'accès à la mémoire centrale et des registres vectoriels permettant de charger des tableaux de plusieurs octets et de les exécuter parallèlement dans un temps horloge réduit grâce à des extensions d'instructions (MMX, SSE, AVX). Cela permet d'exploiter du parallélisme de donnée (ILD) au sein du CPU mais nécessite généralement un alignement en mémoire centrale pour être efficace.

Dans les dernières générations de CPUs (depuis 2005), pour palier la problématique de chaleur dissipée par effet de Joule causée par une fréquence d'horloge trop élevée, le processeur intègre plusieurs circuits de calcul autonomes appelés cœurs physiques. Chaque cœur dispose de ses propres composants (compteur ordinal, ALU, registre ...) lui permettant d'exé-

cuter indépendamment une tâche ou un programme sous forme de thread. On parle alors de parallélisme au niveau des thread (TLP). Les cœurs au sein d'un processeur sont connectés à travers un bus interne et peuvent partager différents niveaux de caches.

Afin d'augmenter encore plus la performance, il est possible de regrouper plusieurs CPUs au sein d'un nœud de calcul appelé multiprocesseur. L'accès à la mémoire centrale des processeurs du nœud peut influer sur les performances et la façon de les programmer. Dans ce qui suit nous distinguons deux architectures pour les multiprocesseurs :

Architecture à accès symétrique (Symmetric MultiProcessing, SMP) Tous les processeurs sont connectés à une unique mémoire commune et se partagent les temps d'accès qu'offre la mémoire ce qui constitue généralement un goulot d'étranglement. La figure 1.5 illustre cette architecture.

Architecture à accès non uniforme (Non Uniform Memory Access, NUMA) Comme montré dans la figure 1.6, les processeurs disposent chacun de leur propre zone mémoire et sont connectés entre eux à travers un bus. Pour chaque processeur, l'accès à la mémoire dépend de l'emplacement de la donnée. De plus, chaque processeur peut disposer d'un cache : si les caches sont cohérents et permettent l'intégrité des données en mémoire centrale, on parle d'architecture ccNUMA (cache coherent NUMA) ; sinon, l'architecture est dite ncNUMA (no-cache NUMA).

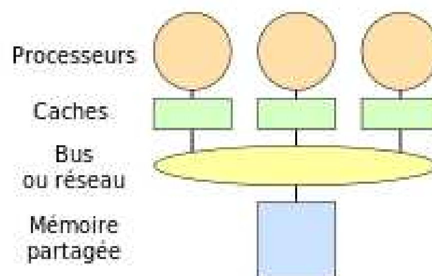

FIGURE 1.5 – Architecture multiprocesseurs à accès mémoire symétrique

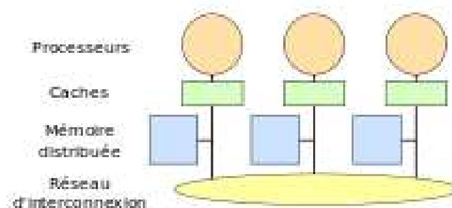

FIGURE 1.6 – Architecture multiprocesseurs à accès mémoire non uniforme

Pour programmer les CPUs composant les nœuds de calcul, ils est possible d'utiliser différents langages de programmation basés sur différents paradigmes, comme C et Fortran en impératif, HASKELL et LISP en fonctionnel. Leur architecture orientée contrôle permet une grande programmabilité (utilisabilité) au prix de circuits de contrôle des niveaux de caches

et de la mémoire centrale. Ils leur procurent notamment un accès rapide en latence aux registres de branchement, de prédictions, de décodage, de spéculation et de recherche de conflit du pipeline, permettant ainsi une forte abstraction des spécificités matérielles. Cependant, ces circuits de contrôle et de transfert interne de données consomment beaucoup d'énergie ce qui fait du CPU l'une des unités de calcul les moins efficaces énergétiquement. Par exemple, un décodeur H.264 : un CPU muti-cœurs est approximativement 500 fois moins efficace énergétiquement qu'un ASIC. En outre, l'autre limitation des performances des CPUs est la contention dûe au faible débit d'accès à la mémoire centrale. Elle engendre une sous utilisation des différents cœurs et unités de traitement. A titre d'exemple, pour le calcul d'un programme de somme de vecteurs générant 1 Flop par 8 Bytes de données, un processeur Intel Quad-coeurs I7 développe théoriquement une puissance de calcul de 48 Gigaflops à condition d'avoir 384 GB/s de débit d'accès mémoire. Or, dans le meilleur des cas, le débit réel atteint seulement 15 GB/s ce qui réduit la performance réelle à approximativement 2 Gigaflops, correspondant à 4 pour-cent de la performance théorique. Ainsi, pour produire de meilleures performances en consommant moins d'énergie, il est nécessaire de s'orienter vers l'utilisation de circuits de calcul spécialisés appelés accélérateurs que nous présentons dans la sous-section suivante.

1.2.2 Processeurs dédiées (Accélérateurs)

On appelle accélérateurs matériels tout circuit ayant pour mission de soulager le CPU. Comme illustré dans la figure 1.7, les accélérateurs disposent d'une mémoire interne, de leurs propre unités de contrôles (registre d'instruction, registre de donnée, compteur ordinal, cache ...) ainsi que d'un jeu d'instructions propre. Leur utilisation dans le calcul de haute performance est de plus en plus fréquente en raison de : leur faible consommation en énergie due à une vitesse d'horloge réduite (autour du 1 GHz), leur temps très réduit d'accès en mémoire locale (autour de 250 GO/s), le nombre importants de cœurs de calcul et leur architecture spécifique orientée vers un type de calcul précis. Cependant, pour les utiliser, il est nécessaire de les connecter à un processeur généraliste et d'utiliser des outils logiciels spécifiques à chacun imposant une programmation bas-niveaux pour compenser la faible présence de circuit de contrôle dans leur architectures. Dans ce qui suit, nous citons les plus utilisés d'entre eux pour composer des grille de calcul.

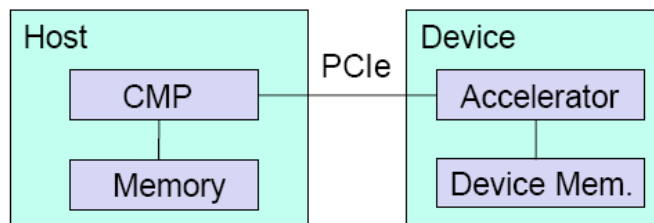

FIGURE 1.7 – Schéma illustrant les caractéristiques générales des accélérateurs

Processeur graphique GPU : pour "Graphics Processing Units", est un accélérateur dédié aux calculs graphiques mais pas seulement, il peut notamment être utilisé pour le calcul généraliste depuis qu'il est possible de les programmer via des softwares spécifiques comme CUDA, OpenCL ou OpenACC que nous présenterons avec plus de détails dans la section 2.2. Contrairement aux CPUs, les GPUs intègrent des milliers de cœurs scalaires regroupés en blocs appelés SMX pour "Streaming Multiprocessors". Les cœurs au sein du SMX partagent des registres et des mémoires locales rapides et tout les cœurs partagent une mémoire globale. Dans la figure 1.8, nous montrons une illustration d'une architecture d'un GPU NVIDIA kepler K20x comportant 13 SMX, soit 2496 CUDA cores. Dans chaque SMX, un groupe de 32 cœurs appelé Warp exécute à un moment donné la même instruction sous la forme SIMT (Single Instruction, Multiple threads) mais sur des données différentes, Ainsi, les tâches de l'utilisateur sont exécutées en exploitant le parallélisme de donnée SIMD. Il est possible aussi de lancer plusieurs instances de calcul sur le GPU qui seront exécutées par des SMX différents, ainsi le GPU peut être considéré comme étant aussi une architecture MIMD.

FIGURE 1.8 – Digramme des principaux blocs du GPU Kepler GK110

Xeon Phi : est un circuit de calcul parallèle de haute performance proposé par Intel sous l'appellation Intel Many Integrated Core (MIC). Il contient dans certaines versions jusqu'à 61 cœurs physiques de type X86 connectés entre eux à travers deux bus rapides appelés ring (1024 bits, 300 GB/s) les reliant à la mémoire locale. Chaque cœur dispose d'une unité vectorielle (512 bits), d'une unité scalaire X86 capable d'exécuter les jeux d'instruction CPUs Intel standard et de deux niveaux de cache cohérents (L1,L2) partagés entre 4 threads physiques. Cependant, dans ces cœurs, le parallélisme d'instruction reste basique avec un court pipeline et une unité de prédiction de branchement peu performante. Au sein du cluster, le Xeon Phi peut être considéré comme étant un nœud indépendant avec son propre système d'exploitation, de fichiers et de communication.

FPGA : pour "Field Programmable Gate Arrays" est une architecture composée d'un ensemble de blocs logiques configurables connectés via un réseau programmable et entourés par des blocs d'entré/sortie, comme illustré dans la figure 1.10. Grâce à ses composants

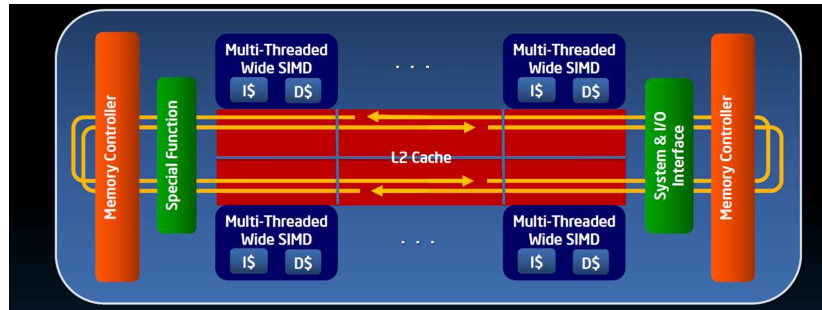

FIGURE 1.9 – Digramme des principaux blocs composants le Xeon phi

programmables offrant un fort degré de parallélisme inhérent et à sa faible consommation d'énergie, le FPGA permet aux programmeurs de produire une implémentation matérielle spécifique à leurs applications. Dans un cluster hétérogènes, le FPGA peut être intégré dans un nœud de calcul comme accélérateur pour assister un CPU, comme c'est le cas par example dans l'architecture Convey-HC1 [Mey+12], permettant ainsi grâce à des outils propriétaires à base de directives, de le décharger de parties spécifiques du calcul sous une configuration pre-établie appelée personnalité, ou dans l'architecture Maxler [HK08], ou le programmeur utilise des outils logiciels de haut niveau (Java+compilateur) pour décrire son application sous forme de graphe qui sera appliqué (mappé) sur le FPGA.

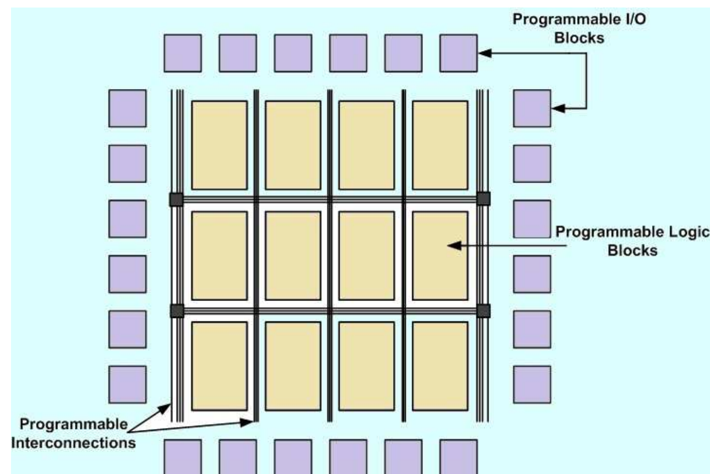

FIGURE 1.10 – Digramme des principaux blocs composant le FPGA

1.3 Programmation des architectures parallèles et hétérogènes

Les clusters hétérogènes sont capables de produire leur maximum de performances en exploitant efficacement l'ensemble des ressources matérielles. Cependant, cette tâche reste complexe et difficile à accomplir pour le programmeur lambda. En effet, il faut dans un premier niveau distribuer les calculs sur les différents nœuds. Pour cela il faut distribuer l'application en manipulant des processus. Cela entraîne des problèmes de synchronisation, de gestion des

mémoires, de transferts de donnée et de répartition des charges dans un environnement de communication par échange de messages à travers le réseaux reliant les nœuds. Ensuite, dans le deuxième niveau, il faut exploiter le parallélisme de tâche (TLP) ou de données (DLP) pour distribuer les calculs sur les unités de calcul hétérogènes au sein du nœud de façon à orienter les parties de calcul adéquates vers chacune, par exemple orienter les calculs intensifs mais hautement parallélisables avec des besoins élevés en débit mémoire vers les accélérateurs SIMD et les calculs avec forte charge d'opérations d'entrée-sortie, branchements et accès multiples à la mémoire vers les CPUs (MIMD). Pour cela, il faut exprimer l'application en utilisant des threads ou processus fils interagissants via les échanges de données. Pour cela il faut faire face à des problèmes de gestion de la mémoire centrale et des mémoires locales en effectuant des transferts de données entres elles en les recouvrant par des calculs dans le même temps. Il faut synchroniser les threads sur mémoire partagée en utilisant des barrières de synchronisation ou des outils de protection de la cohérence de données comme les Sémaphores. Il faut répartir la charge de travail sur les nœuds de mêmes types en tenant compte de leur hétérogénéité (puissance de calcul différente) et de leur capacité de transfert de données (débit PCIe). Finalement, dans le dernier niveau de l'architecture i.e. le niveau de l'unité de calcul (CPU, ou accélérateur), il faut veiller à exploiter les caractéristiques de chaque unité de calcul de manière optimale : pour les CPUs, il faut notamment privilégier les accès locaux en mémoire pour les threads des nœuds NUMA et mettre à profit les unités vectorielles et le parallélisme au niveau de l'instruction (ILP) comme la superscalarité ou le pipelining. Cela passe par l'utilisation de compilateurs performants avec des niveaux d'optimisation poussés ou par la manipulation des bibliothèques spécifiques. Pour les accélérateurs : il est conseillé aussi de suivre les bonnes pratiques recommandées pour chaque type en utilisant les langages de programmation et outils logiciels adéquats.

En résumé, afin d'optimiser les performances des architectures parallèles et hétérogènes, le programmeur doit faire face à 2 types de problèmes :

Problèmes liés au parallélisme :

1. Manipuler les processus et les threads pour exprimer le parallélisme de tâches et de données (création, synchronisation, communication, suppression).
2. Gérer les mémoires partagées et distribuées.
3. Répartir les charges sur les nœuds et sur les unités de calcul en prenant en considération les temps de calcul et les temps de communication.
4. Optimiser l'occupation des unités de calcul.

Problèmes liés à l'hétérogénéité :

1. Combiner l'utilisation de plusieurs outils logiciels pour cibler les unités de calcul hétérogènes.
2. Connaître des spécificités matérielles de chaque unité de calcul pour optimiser le code.
3. Tenir compte des différences de performance entre les unités de calcul dans la répartition de charge.

4. Identifier et orienter les calculs spécifiques vers les unités adéquates.

Ces contraintes font de l'implantation efficace des applications sur des clusters hétérogènes une mission difficile réduisant considérablement la productivité des programmeurs et la portabilité du code généré. Pour cela des outils de programmation appelés modèles de programmation parallèle ont vu le jour pour faciliter la programmation de ces "clusters" et obtenir les performances attendues. Dans le chapitre suivant nous décrivons et étudions ces modèles.

1.4 Conclusion

Dans la quête continue de performance, les architectures parallèles et hétérogènes représentent une évolution naturelle des machines de calcul. Dans ce chapitre nous avons présenté ces architectures et les unités de calculs les composants, à savoir les CPU, GPU, Xeon Phi et FPGA. Nous avons étudié les caractéristiques de ces CPUs et accélérateurs et les types de parallélisme qu'ils offrent (ILP, TLP, DLP). Nous avons discuté leurs avantages et inconvénients selon leur caractéristiques matérielles internes. Finalement, nous avons présenté les contraintes rencontrées par le programmeur pour exploiter efficacement ces architectures.

Chapitre 2

Les modèles de programmation pour les architectures parallèles et hétérogènes

2.1 Introduction

Exploiter efficacement le parallélisme sur une architecture hétérogène est un défi pour tout programmeur. En effet, il est difficile d'exprimer les propriétés naturelles des algorithmes avec les propriétés matérielles bas niveau des machines de calcul en exploitant toutes leurs performances. Pour répondre à cette problématique, il est nécessaire d'utiliser des concepts de programmation appelés modèles de programmation. Ces modèles permettent de faciliter le développement en abstrayant les spécificités matérielles avec des fonctionnalités logicielles de haut-niveau et de produire les performances attendues en implémentant efficacement ces fonctionnalités proposées. En d'autre mots, les modèles de programmations parallèles sont des concepts de liaison entre les applications des utilisateurs et les caractéristiques bas niveaux des architectures cibles comme illustré dans la figure 2.1. Dans ce chapitre, nous proposons d'étudier ces modèles. Nous présentons en premier dans la section 2.2 ces modèles et leurs propriétés permettant de les évaluer. Ensuite, nous les classons dans la section 2.3 selon leur niveau d'abstraction. Dans la section 2.4, nous présentons quelques environnements connus représentant l'implémentation des Modèle de Programmation Parallèle (MdPP)s et citons leur avantages et inconvénients pour programmer des architectures parallèles et hétérogènes.

2.2 Modèles de programmation parallèles et propriétés

Un MdPP est une abstraction d'une machine parallèle et hétérogène, offrant des fonctionnalités de programmation implémentées efficacement sur les composants de cette dernière. Comme illustré dans la figure 2.1, il représente la jonction entre la description du calcul de l'algorithme avec le modèle de calcul (MdC) décrit dans la section 3.3 du chapitre 3 et la description des spécificités matérielles de l'exécution avec le modèle d'exécution (MdE). Cette abstraction doit d'un côté permettre la portabilité du programme sur une autre architecture de la même famille. En effet, le MdPP n'a pas grand intérêt s'il propose une abstraction trop bas-niveau liée à une architecture spécifique. D'un autre côté, l'abstraction d'un MdPP n'est

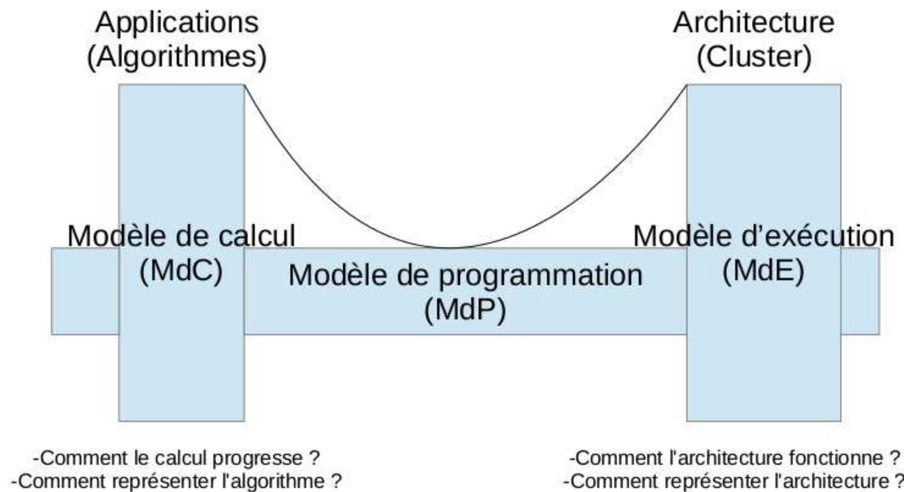

FIGURE 2.1 – Représentation du modèle de programmation sous forme de pont reliant les applications et les architectures.

pas utile si aucune méthode efficace n'est trouvée pour exécuter les programmes qu'il modélise. Un MdPP ne doit donc pas offrir une abstraction de trop haut-niveau au point de rendre impossible l'exécution de ce dernier sur un nombre étendu de machines parallèles. Pour être utile, un MdPP doit satisfaire un compromis entre abstraction et efficacité. [ST98] propose des critères d'évaluation des MdPPs. Dans ce qui suit, nous citons quelques uns d'entre eux :

Facilite la programmation (abstraction) : Un MdPP doit cacher aux programmeurs la plus part des détails de l'architecture parallèles. La structure et les caractéristiques de l'exécution du programme doivent être générés autant que possible par le mécanisme d'abstraction du MdPP plutôt qu'exprimés par le programmeur. Cela implique que le modèle doit masquer autant que possible les points suivants :

1. *La décomposition* du programme en des threads parallèles. Le programme doit être découpé en parties qui sont exécutées sur des unités de calcul indépendantes. Cela requiert de décomposer l'algorithme et les structures de données en un certain nombre d'unités.
2. *La communication* entre threads. Des communications sont nécessaires pour faire progresser le calcul dès que les données ne sont pas disponibles localement. Techniquement, elles sont fortement couplées à l'architecture. Par exemple, celles ci peuvent nécessiter des copies système de fichiers sur des architectures à mémoire partagée, ou des copies réseaux pour les architectures distribuées.
3. *La synchronisation* des threads. Dans une exécution parallèle, la synchronisation entre deux threads ou un groupe de threads est une étape inévitable pour garantir la cohérence des données. Comme c'est le cas pour les communications, les mécanismes de synchronisation dépendent aussi de l'architecture, à savoir par échange local de données (Sémaphore, Mutex, barrière), ou par échange de messages.

4. *La distribution* des threads sur les unités de calcul. Une fois le programme décomposé en parties pouvant être traité en parallèle, leur placement sur les processeurs physiques doit être fait. Ce choix est fortement dépendant des communications et synchronisation. Des parties communicantes gagnent souvent à être placées dans la même unité de calcul. Le placement peut nécessiter une vérification par type de calcul pour le faire correspondre aux unités de calculs spécifiques.

Indépendant des architectures (Performance) : Le MdPP doit pourvoir offrir à l'utilisateur la possibilité de porter des programmes écris pour une architecture parallèle vers une autre architecture parallèle sans les re-développer. En effet, d'une part, les machines de calcul peuvent différer par leur architectures mémoire (partagée, distribuée, hybride) et la nature de leurs unités de calcul (CPU, GPU, ...). D'autre part, elles ont une courte durée de vie et sont susceptibles d'être remplacées régulièrement. Redévelopper les programmes pour chaque migration peut générer des coûts élevés. Le MdPP doit offrir une abstraction suffisante pour s'adapter à différentes architectures.

Implémenté efficacement : Le MdPP doit pouvoir être implémenté efficacement sur les architectures parallèles les plus utilisées. Une implémentation efficace ne signifie pas forcément l'exploitation de toute la performance des architectures cibles. Cependant, l'implémentation parallèle doit garantir un gain minimum qui justifie son utilisation. Il existe plusieurs métriques pour évaluer l'efficacité d'une implémentation. Les plus connues d'entre elles sont le taux d'accélération S et l'efficacité parallèle E qui sont calculés comme suit :

$$S(p) = T(p)_1/T(p)_n$$
$$E(p) = S(p)/n$$

Sachant que : $T(p)_1$ et $T(p)_n$ sont les temps d'exécution sur 1 puis sur n éléments de calcul.

Ces deux métriques permettent de calculer le passage à l'échelle du programme sur une architecture parallèle de n éléments de calcul. Un MdPP doit garantir un passage à l'échelle proche de 1 ($E \to 1$).

Simple à prendre en main : Un MdPP doit être facile à comprendre et à enseigner. Il doit offrir une interface simple à utiliser et à prendre en main par des programmeurs de tous les niveaux. Il doit aussi présenter des fonctionnalités proches logiquement des fonctionnalités algorithmiques. Un MdPP simple à utiliser est plus adopté par les utilisateurs même s'il n'est pas plus efficace qu'un autre MdPP plus compliqué à utiliser.

Il est à noter que ces métriques sont parfois paradoxales et difficiles à atteindre en même temps. Il est compliqué, par exemple, pour un MdPP de garantir aux programmeurs une haute abstraction du matériel et en même temps de produire de grandes performances. Ou, par exemple, de concilier une indépendance par rapport à une large palette d'architectures matérielles et dans un même temps, offrir une facilité d'utilisation via une interface simple. Un MdPP doit trouver le bon compromis entres ces critères d'évaluation.

2.3 Classification des modèles de programmation

Les MdPP visent à abstraire les architectures parallèles et hétérogènes tout en permettant d'en tirer les meilleures performances. Dans la section précédente nous avons énuméré quelques propriétés permettant de les évaluer. Dans cette partie, nous proposons de les classer en se basant sur deux de ces métriques, à savoir, le niveau d'abstraction qu'ils offrent à l'utilisateur et le type d'architecture qu'ils couvrent.

2.3.1 Niveau d'abstraction

Le choix d'un niveau d'abstraction est à la base de la construction d'un MdPP. Il induit des propriétés importantes telles que la simplicité d'utilisation du modèle ou sa performance. En effet, pour un modèle à haut niveau d'abstraction masquant entre autre la décomposition des programmes en threads, il est important d'offrir une interface adaptée et simple d'utilisation et de garantir en même temps une performance élevée via des fonctionnalités efficaces. Dans ce qui suit nous citons les différents niveaux d'abstraction présentés dans [ST98] :

1. **Les MdPP masquant complètement le parallélisme**. Ces modèles permettent d'exécuter parallèlement un programme en décrivant seulement son but et non pas comment l'implémenter. Les programmeurs n'ont pas d'information sur la nature parallèle ou séquentielle de l'exécution de ce dernier.
2. **Les MdPP dans lesquels le parallélisme est explicitement exprimé, mais la décomposition du programme en threads est implicite**. L'utilisateur de ces modèles connaît la nature parallèle de l'exécution du programme et doit exprimer sa potentielle utilisation. Mais, il ne donne pas d'information précise sur l'exécution parallèle de chaque partie de l'algorithme.
3. **Les MdPP dans lesquels le parallélisme et la décomposition sont explicites, mais la synchronisation, la communication et la distribution sont implicites**. Ces modèles nécessitent d'exprimer la décomposition de l'algorithme avec des threads, processus ou tâches, mais gèrent automatiquement le reste de l'exécution.
4. **Les MdPP dans lesquels le parallélisme, la décomposition et la distribution sont explicites. La communication et la synchronisation sont implicites**. Dans ces modèles, en plus de devoir décomposer le programme en parties parallèles, il doit faire le choix de leur placement sur les unités de calcul. Idéalement, ce placement doit permettre d'équilibrer les charges de travail entre processeurs et tenir compte des temps de communications. Par conséquent, le programmeur doit prendre en compte les caractéristiques du matériel cible ce qui réduit considérablement la portabilité de l'application.
5. **Les MdPP dans lesquels le parallélisme, la décomposition, la distribution et les communications sont explicites. Seule la synchronisation est implicite**. Dans ces modèles, le programmeur doit explicitement prendre des décisions sur toutes les tâches d'implémentation, sauf pour la synchronisation qui elle, est gérée automatiquement.

6. **Les MdPP dans lesquels tout est explicite**. Le programmeur doit prendre des décisions sur tous les éléments de l'exécution, à savoir la décomposition en tâche, leur placement, les communications et les synchronisations. L'utilisation de ces modèles allonge les temps de développement et aboutit à des implémentations peu portables.

2.3.2 Couverture de l'architecture

Une autre propriété des MdPP est l'indépendance par rapport aux architectures. En effet, comme décrit dans le chapitre 1, les architectures parallèles peuvent avoir une hiérarchie mémoire distribuée, partagée ou hybride. Aussi, ils peuvent inclure différentes unités de calcul avec des caractéristiques différentes. Pour garantir une forte indépendance par rapport aux architectures et une bonne portabilité des programmes, les MdPP doivent être capables d'abstraire ces couches matérielles bien que l'implémentation devra prendre en compte ces différences.

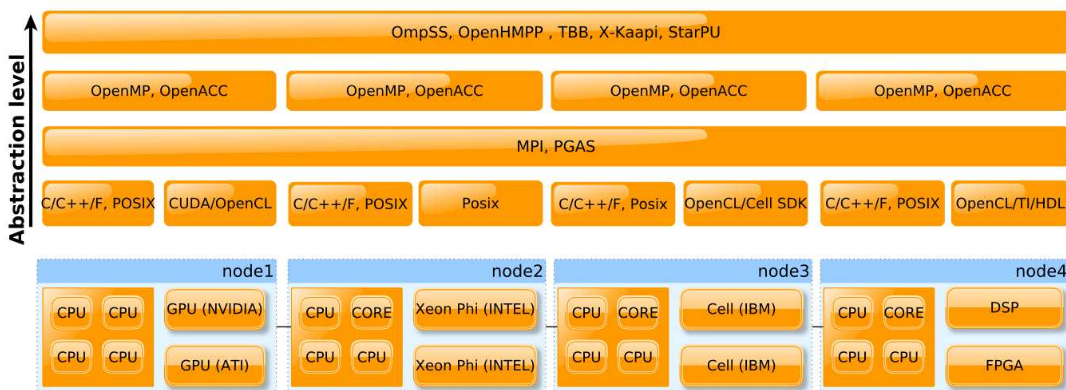

FIGURE 2.2 – Classification des environnements de programmation parallèle et hétérogène

2.4 Les environnements de programmation parallèles et hétérogènes

Dans ce qui suit, nous présentons quelques environnements de programmation utilisés pour exploiter les architectures parallèles et hétérogènes. Ils ne correspondent pas forcément aux modèles de programmations parallèle comme défini précédemment et peuvent représenter l'implémentation d'une combinaison de plusieurs d'entre eux. Ces environnements peuvent prendre plusieurs formes logicielles : il peuvent être proposés sous forme de langages ou d'extension de langages avec des syntaxes spécifiques et un compilateur dédié. Ils peuvent être sous la forme d'une librairie comportant un ensemble de fonctions et de structures de données pour aider le programmeur dans sa tâche. Finalement, il peuvent prendre la forme de directives permettant d'annoter du code séquentiel pour le paralléliser ou l'exécuter sur des accélérateurs. La figure

Fig. 2.2 positionne ces environnements sur un cluster selon leur niveaux d'abstraction (axe vertical) et l'architecture supportée (axe horizontal).

CUDA [SK10] : pour "Compute Unified Device Architecture". C'est un modèle de programmation bas niveau exprimé en langage C avec extensions. Proposé par NVIDIA, il est destiné à la programmation généralisée de ses GPU "General-Purpose Computing on Graphics Processing Units". Il comporte un compilateur générant un code binaire s'exécutant sur une architecture GPU et CPU. CUDA permet un grand contrôle sur les GPUs mais reste assez difficile à prendre en main. En effet, le programmeur doit découper son application pour être traitée finement avec un nombre important de threads. Un exemple de programme CUDA réalisant l'addition de deux vecteurs est donné dans l'annexe A. L'utilisateur doit gérer les allocations mémoire sur le GPU, les communications entre la machine hôte et le GPU, le lancement de l'exécution des calculs sur le GPU, la synchronisation des tâches. CUDA est un MdPP orienté parallélisme de données et couvre seulement les architectures à mémoire partagée au sein d'un nœud de calcul.

OpenCL [Mun+11] : pour "Open Computing Language", c'est un standard ouvert pour le calcul parallèle sur CPU, GPU, Cell, FPGA et DSP. Il est proposé par Khronos Group sous forme d'un langage de programmation bas niveau basé sur C99 avec certaines restrictions et des extensions pour le parallélisme. Il est muni d'un kit de développement "SDK" permettant de compiler les codes des fonctions pour accélérateurs de différentes architectures. OpenCL permet une forte portabilité du code et prend en compte le parallélisme de donnée et le parallélisme de tâche. Cependant, il couvre seulement les architectures à mémoire partagée dans un nœud de calcul. Dans l'annexe A, nous présentons un example d'un programme OpenCL réalisant l'addition de deux vecteurs. Le programmeur doit se charger de toutes les étapes de l'exécution : décomposition en threads, communication, synchronisation, placement et le lancement des Kernels.

MPI [Pac96] : pour "Message Passing Interface", est un modèle de programmation bas niveau par échange de messages. Il est destiné à la programmation des architectures multi-processeurs ou multi-nœud à mémoires distribuées. Différentes implémentations C/C++ ou Fortran de MPI sont proposées. Elles incluent un moteur d'exécution "runtime" pour lancer l'exécutable généré. MPI est un MdPP totalement explicite ce qui permet d'exploiter le parallélisme de tâches ou de données mais n'interagit pas avec les accélérateurs.

PGAS [Che07] : pour "Partitioned Global Address Space". C'est un modèle de programmation pour architectures à mémoire partagée et distribuée. Il a pour objectif de combiner la simplicité de référencement des données dans les modèles à espace d'adressage global avec une performance basée sur la localité des traitements. Il propose à l'utilisateur un espace d'adressage global mais partitionné logiquement. Chaque partition est locale pour un thread ou processus en correspondance à la distance matérielle (localité matérielle). L'approche PGAS

est reprise dans plusieurs langages comme Chapel [CCZ07] développé par Cray, X10 par IBM [Sar+12], Fortress [All+08] par Sun ou Unified parallel C [UPC05] par UPC consortium. En première approche, l'utilisateur de ces langages peut seulement décomposer explicitement l'algorithme, sans se préoccuper des communications, synchronisations et du placement des tâches. Cependant, s'il veut obtenir de meilleur performances, il doit explicitement optimiser les communications et le placement des tâches.

OpenMP [Cha+01] : pour "Open Multi-Processing". C'est un ensemble de directives et de variables d'environnement pour la programmation des architectures à mémoire partagée incluant les multi-cœurs et les accélérateurs GPU et Xeon Phi dans sa dernière version. OpenMP permet de générer le parallélisme de tâches et le parallélisme de données sur unité vectorielle et sur accélérateur. C'est un modèle de programmation à moyen niveau d'abstraction. L'utilisateur doit introduire des directives dans son code séquentiel pour le paralléliser ou décharger son exécution sur un accélérateur. Cependant, les synchronisations et les communications peuvent être gérées automatiquement. OpenMP ne couvre pas les architectures à mémoire distribuée et ne gère pas les placements sur architectures hétérogènes.

OpenACC [KH10] : comme OpemMP, c'est un standard ouvert de directives, il permet à l'utilisateur d'annoter du code séquentiel à l'aide de "pragma" et ainsi de paralléliser l'exécution sur multi-cœur, GPU ou sur des accélérateurs Intel (Xheon-Phi). Ce standard est le fruit de la collaboration entre Cray, CAPS, Nvidia and PGI, qui proposent chacun un compilateur pour générer du code exécutable. Comme OpenMP, OpenACC est un modèle à moyen niveau d'abstraction. Le programmeur se charge seulement de décomposer son algorithme et de placer ses parties sur les différentes unités de calcul.

OmpSS [Bue+11] : est une autre variante d'OpenMP 3.0 étendue pour supporter l'exécution asynchrone de tâche, les dépendances de données entre les tâches (data-flow) et les tâches destinées à être exécutées sur accélérateurs. Comme OpenMP, il est basé sur la "décoration" d'une version séquentielle de code existant avec des directives de compilation qui sont traduites en appels à un système d'exécution (Runtime) afin de gérer l'extraction du parallélisme et la circulation des données. Il offre un placement dynamique des tâches permettant d'équilibrer les charges entre les processeurs positionnant son niveau d'abstraction au dessus de OpenMP et OpenACC.

OpenHMPP [Hir12] : est le standard de HMPP (Hybrid Multicore Parallel Programming) conçu par CAPS. C'est un modèle de programmation haut niveau basé sur les directives. L'application est exprimée à l'aide de Codlet représentant une fonction respectant certaines contraintes (pas de retour direct des valeurs, pas d'appel a des variables globales ou de déclaration de variable statique, etc). Les Codelets sont appelées dans le programme principal par la directive "pragma hmpp" pour être exécutée (offload) par un son Runtime sur cluster de GPU et/ou de multi-cœurs.

TBB [Rei07] : pour "threading Building Blocks", est une librairie proposée par Intel pour adresser ses multi-cœurs comme les Xeon et ses many-cœurs comme le Xeon Phi. Il est orienté vers le parallélisme de tâche grâce à des fonctions permettant de dérouler une boucle "For" sur un ensemble de threads mais permet aussi leur vectorisation. Il gère les communications et les synchronisations avec un graphe de flot de données et le placement dynamique des tâches. TBB est un modèle de programmation à moyen niveau d'abstraction qui adresse des architectures à mémoire partagée seulement.

StarPU [ATN10] : est un modèle de programmation haut niveau orienté vers le parallélisme de tâches. Il couvre des architectures à mémoires partagées et distribuées et comporte un support d'exécution (runtime) capable d'extraire le parallélisme implicitement à partir du code utilisateur pour former un graphe directionnel acyclique DAG de tâches et le distribuer dynamiquement sur l'architecture ciblée. Il est proposé sous forme d'une API ou de directives et supporte les accélérateurs suivant : Multi-core, Many-Core, GPU, Cell et Xeon Phi.

X-Kaapi [GBP07] : X-Kaapi est le descendant de Kaapi, abréviation de "Kernel for Adaptative, Asynchronous Parallel and Interactive programming". C'est un modèle de programmation de haut niveau orienté vers une décomposition en tâches sur mémoire partagée et distribuée. Comme StarPU, il est doté d'un Runtime capable d'extraire le parallélisme implicitement à travers un graphe de tâches acyclique et de les distribuer dynamiquement sur l'architecture. Il supporte les multi-cœurs et les GPU, et est proposé soit sous la forme d'une API, soit sous la forme d'un compilateur KACC et d'un ensemble de directives pragma du language C.

Dans ce qui suit, nous présentons un tableau comparatif des environnements de programmation parallèles cités ci-dessus.

2.5 Discussion

Les modèles de programmation ont un objectif double, ils doivent à la fois augmenter la productivité en offrant une interface accessible à l'utilisateur et produire les performances attendues. Finalement, ils doivent couvrir une large palette d'architecture. Cependant, les architectures sont de plus en plus complexes et hétérogènes et les applications de plus en plus spécifiques avec de fortes et diverses contraintes. Certaines d'entre elles sont orientées vers le parallélisme de données SIMD avec des exigences de débit comme le traitement graphique. D'autres sont plutôt orientées vers le parallélisme de tâche MIMD avec des contraintes de latence. D'autres encore sollicitent beaucoup la mémoire (memory bound) alors que certaines requirent plutôt des calculs intensifs (compute bound). Dans ce qui suit, nous proposons de spécifier les MdPPs par domaine applicatif. En effet, en se basant sur les caractéristiques communes des applications d'un domaine spécifique, il est possible de produire un modèle d'abstraction adapté à leurs caractéristiques et ainsi permettre de concilier les objectifs de productivité, de couverture d'architecture et de performances. Dans les chapitres suivants,

nous présentons deux modèles de programmation spécifiques aux applications de traitement du signal et de l'image DSP pouvant être modélisées sous la forme d'un MdC à réseau de processus et plus précisément sous la forme d'un MdC graphe de flot de donnée synchrone (SDFG).

Environnement de développement	Décomposition en parties parallèles	Communication	Architecture mémoire	Synchronisation	Placement	Ordonnancement	Couverture architecture	Contrôle	Productivité
CUDA	Explicite (SPMD)	Explicite	Partagée	Explicite	Explicite	Dynamique (local)	* (GPU)	***	*
OpenCL	Explicite (SPMD)	Explicite	Partagée	Explicite	Explicite	Dynamique (local)	** (Accélérateurs, CPU)	***	*
MPI	Explicite (SPMD)	Explicite	Distribuée	Explicite	Explicite	Statique	*(CPU, Cluster)	***	*
PGAS	Implicite (SPMD)	Implicite	Distribuée	Explicite	Implicite	Statique	* (CPU, Cluster)	**	**
OpenMP	Implicite (SPMD)	Implicite	Partagée	Implicite	Explicite	Statique	** (Accélérateurs, CPU)	**	**
OpenACC	Implicite (SPMD)	Implicite	Partagée	Implicite	Explicite	Statique	** (GPU, CPUs)	**	**
OmpSS	Implicite (SPMD)	Implicite	Distribuée	Implicite	Implicite	Dynamique	*** (GPU, CPU, Cluster)	**	**
OpenHMPP	Implicite (MPMD)	Implicite	Partagée	Implicite	Implicite	Statique	** (GPU, CPU)	**	**
TBB	Implicite (MPMD)	Implicite	Partagée	Implicite	Implicite	Dynamique	** (Xeon Phi, CPU)	**	**
StarPU	Explicite (MPMD)	Implicite	Distribuée	Implicite	Implicite	Dynamique	*** (Accélérateurs, CPU, Cluster)	***	**
X-Kaapi	Explicite (MPMD)	Implicite	Distribuée	Implicite	Implicite	Dynamique	** (CPU, Cluster)	**	**

Figure 2.3 – Tableau comparatif des différents environnements de programmation parallèle sur cluster hétérogène

Chapitre 3

Les applications DSP

3.1 Introduction sur les applications DSP

De nos jours les applications du traitement numérique du signal DSP connaissent un grand essor et sont présentes dans de nombreux domaines comme les télécommunications, le traitement des images ou des vidéos, les traitement acoustiques etc. Elles consiste à appliquer un traitement mathématique ou algorithmique, répétitif (itératif) sur un signal d'entrée (signal sonore, images, signal radio) pour produire un autre signal ou un résultat en sortie. Une chaîne de traitement numérique du signal débute par un bloc d'échantillonnage permettant de transformer un signal analogique dépendant du temps en un signal numérique sous forme d'un ensemble de données unitaires. Les échantillons sont traités numériquement pour obtenir des résultats pouvant éventuellement être reconvertis finalement en un signal analogique.

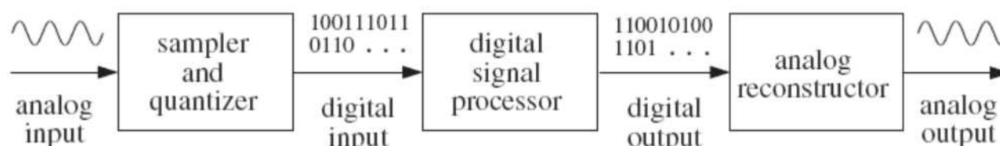

Figure 3.1 – Chaine de traitement numérique du signal

Les applications DSP peuvent être classées selon différents critères. Une première classification consiste à distinguer les applications fortement contraintes en temps ou temps réel (on-line) des applications non strictement contraintes en temps ou à temps différé (off-line).

Le traitement temps réel Dans les systèmes temps réel, un résultat est considéré correct s'il est fonctionnellement juste et s'il respecte une contrainte de temps. Cette contrainte peut être sur la latence i.e. le temps entre la lecture de la donnée d'entrée et la production de la donnée de sortie correspondante. Autrement, elle peut être sur la cadence i.e. le temps écoulé entre la production de deux données de sorties successives.

Le traitement à temps différé Dans les systèmes non contraints temporellement, les temps de calcul ne sont pas strictement bornés. L'application dispose du temps nécessaire pour traiter les données dans la limite de la patience de l'utilisateur, par exemple, dans l'application de la chair chorus [LB+13]. Les données sont récoltées, puis traitées dans des temps raisonnablement différés.

Un autre moyen de les classifier est de les distinguer selon la disponibilité des données qu'elles traitent :

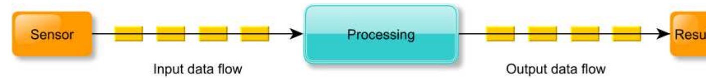

FIGURE 3.2 – Traitement par flux de données

Le traitement par flux de données Dans ce cas les données sont disponibles sous la forme d'un flux dont on ne connaît pas la fin, par exemple une vidéo ou un signal radio provenant d'un capteur donné comme illustré dans la figure 3.2. L'application n'a accès à un moment donnée qu'à une seule unité de donnée à la fois qu'elle va traiter pour produire une unité de donnée en sortie.

Le traitement par blocs de données Dans ce cas de figure, l'application traite un bloc de données entièrement disponible, par exemple, le traitement d'une image ou d'un ensemble d'images préalablement enregistrées dans un support de stockage comme illustré dans la figure 3.3. Dans ce type d'application le calcul ne peut commencer après avoir charger tout le bloc de donnée d'entrée, ce qui rend ce type d'application plus gourmand en mémoire que le précèdent.

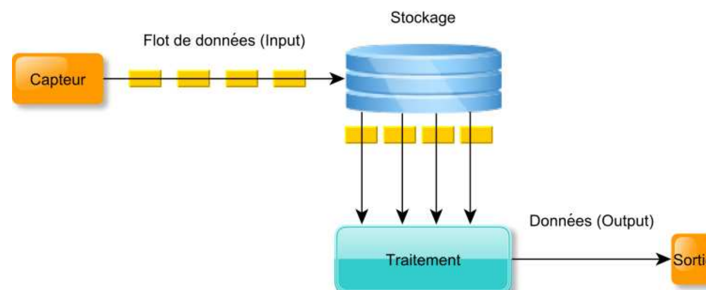

FIGURE 3.3 – Traitement par blocs de données

3.2 Modélisation des applications DSP

Comme indiqué précédemment, les algorithmes DSP consistent à appliquer de manière répétitive (itérative) des traitements sur des signaux en entrée pour produire des signaux en sortie. Ils sont souvent représentés sous forme de diagrammes composés de blocs de calcul reliés par des liens représentants les échanges de données. La figure 3.4 illustre la représentation sous la forme de digramme de blocs de l'exemple de système DSP suivant :

$$y_{(n)} = a * x_{(n)} + b * x_{(n-1)} + c * x_{(n-2)}$$

Cette équation récurrente représente un traitement itératif pour $n = 0, 1, 2, \ldots$.. Chaque itération traite une unité de donnée en entrée et produit un résultat en sortie. Le calcul de

chaque itération est représentée par des blocs et dépends des entées des 2 itérations précédentes $x_{(n-1)}$ et $x_{(n-2)}$ illustrées par des retards D dans la figure.

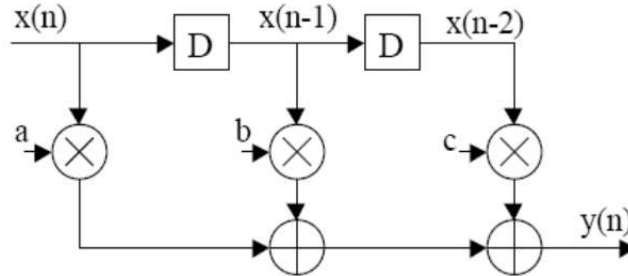

FIGURE 3.4 – Représentation de l'équation $y_{(n)} = a * x_{(n)} + b * x_{(n-1)} + c * x_{(n-2)}$ à l'aide d'un schéma de blocs fonctionnels

Cette représentation a donné naissance à une représentation plus générale pour des algorithmes plus complexes : le graphe de flot de données (Data-flow diagram) représente l'algorithme sous forme d'un graphe orienté G=(V,E) où les nœuds V représentent des calculs, des fonctions, ou des tâches et les arrêtes E représentent les flux de données échangés entre ces calculs avec éventuellement des délais. Les nœuds de calcul sont indépendants et peuvent être exécutés dès la disponibilité de leurs données d'entrées. Dans la figure 3.5, il est montré le DFG représentant l'algorithme 1.

Algorithm 1 Exemple d'algorithme d'une application DSP

Entrée : Unités de donnée d'entrée ($Data_{unit}$). Ensemble de données d'entrée ($DataSet_{in}$).
Sortie : Ensemble de donnée de sortie ($DataSet_{out}$).
1: **for each** $Data_{unit}$ in $DataSet_{in}$ **do**
2: $Var_1 \leftarrow Producer(Data_{unit})$ {Lire chaque unité de donnée d'entrée à partir de l'ensemble des entrées}
3: $Var_2 \leftarrow kernel_1(Var_1)$
4: $Var_{3_1} \leftarrow Kernel_2(Var_2)$
5: $Var_{3_2} \leftarrow Kernel_3(Var_2)$
6: $Var_4 \leftarrow Kernel_4(Var_{3_1}, Var_{3_2})$
7: $Var_5 \leftarrow Var_4$
8: $DataSet_{out} \leftarrow Consumer(Var_5)$ {Écrire chaque unité de donnée de sortie dans l'ensemble des sorties}
9: **end for**

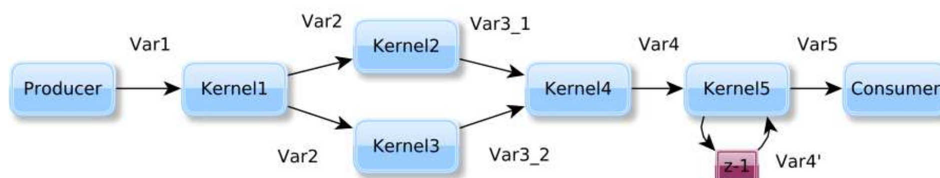

FIGURE 3.5 – Représentation par graphe de flot de données

L'algorithme 1 est un exemple d'application DSP qui traite itérativement dans la boucle principale un ensemble de données unitaires. Chaque fonction de l'algorithme est représentée par un nœud dans le graphe. Les arrêtes du graphe modélisent le passage des données intermédiaires représentant les arguments des fonctions. Le nœud *Producer* représente la fonction de production de la donnée d'entrée et le nœud *Consumer* la fonction de consommation de la donnée de sortie. Ainsi, pour traiter toutes les données, le graphe en entier est recalculé autan de fois qu'il y a de données. Cette représentation permet aussi selon le partitionnement des données d'entrée, de paramétrer **l'échelle** du programme spécifiant la taille des données à traiter par itération. En effet, une échelle élevée avec une fragmentation fine des données d'entrée et un nombre d'itérations important nécessite plusieurs éléments de calcul et risque de créer des surcoûts à l'exécution. En revanche, une échelle faible produit moins d'itérations et ce qui nécessite moins d'éléments de calcul. Cependant, les unités de données à traiter sont plus importantes ce qui risque de générer une exécution sous optimale. Un autre paramètre important dans la représentation des applications DSP sous forme de DFG est la **granularité**. Elle représente le découpage de l'application en parties de traitement permettant d'augmenter le parallélisme. Elle dépend d'une part de l'algorithme et d'autre part de l'architecture ciblée. Dans [Sin07], Sinnen la définit plus formellement comme le rapport entre le temps de calcul des nœuds et le temps de transfert des données :

$$G = \frac{\min_{T_{exec}}}{\max_{T_{transfert}}}$$

Si $G >> 1$ le graphe est dit à grosse granularité, ce qui signifie que les temps de transferts sont minimes par rapport au temps de calcul. Si $G \simeq 1$ Le graphe est dit à moyenne granularité. Finalement, Si $G << 1$, il est dit à fine granularité.

Le DFG est considéré plus formellement comme un modèle de calcul (MdC), i.e, un modèle mathématique qui décrit comment le calcul de l'algorithme progresse. Dans ce qui suit nous le présentons plus en détails ainsi que son positionnement au sein des MdCs les plus connus.

3.3 Le modèle de calcul graphe de flot de donnée

Un modèle de calcul est un modèle mathématique qui définit la sémantique exacte d'un système de calcul, i.e. quels sont ses composants, comment ils sont liés et comment ils interagissent. Le Modèle de Palcul (MdC) décrit les méthodes qui permettent de spécifier et/ou simuler et/ou exécuter un algorithme. La définition des MdC est large et couvre plusieurs modèles. Elle est proche de la notion de paradigme de programmation dans la programmation informatique définissant le modèle de construction du programme. Par exemple nous citons : le modèle impératif, le déclaratif, le fonctionnel, l'oriente objet, ou le "dataflow". Dans ce qui suit nous présentons les principaux MdCs utilisés dans le monde académique et industriel.

MdC à machine d'états finis (FSM) dans lequel des états internes du système de calcul sont définis ainsi que les règles de transition entre deux états. Les FSMs peuvent être

synchrones ou asynchrones. Ils sont généralement utilisés pour modéliser des systèmes de contrôle ou les calculs sont effectués dans les états pendant les transitions. Le paradigme de programmation impératif est équivalent aux FSMs dans le sens ou le programme comporte plusieurs états qui sont modifiés séquentiellement. Le paradigme impératif est la base des langages de programmations les plus utilisés comme C, ou d'autre paradigmes de programmation plus haut niveaux comme l'orienté objet utilisé au travers des langages comme C++, Java et FORTRAN99.

MdC à événements discrets (ED) dans lequel les modules régissent à des événements pour produire d'autres événements, ou les modules sont spécifiés avec des MdC impératifs. Ces événements sont définis dans le temps dans le sens ou leurs temps de production et de consommation sont utilisés dans la description du systèmes de calcul. Les MdC à événements discrets sont utilisés généralement pour modéliser des circuits dont le fonctionnement est cadencé par une horloge comme le FPGA et ASIC et servent de base à des langages de description de matériel comme le VHDL ou le Verilog.

MdC fonctionnels dans lequel un programme n'a pas d'état mais est représenté sous forme d'évaluations de fonctions mathématiques. Haskell, Caml, LISP ou XSLT sont des exemples de langages de programmation fonctionnelle. Les origines du MdC fonctionnel sont liées aux travaux de Church sur le Lamda calcul [Rev09] définissant tout calcul comme composition de fonctions à différents degrés avec une seule entrée chacune.

MdC à réseaux de Petri qui contient des files non ordonnées appelées transitions avec de multiples entrées et sorties et des états locaux appelés places. Les transitions et les places sont reliées par des arcs orientés formant ainsi un graphe biparti orienté. Les transitions exécutent les calculs en produisant des unités de données appelées jetons ou Tokens sur les places de destination si un jeton est disponible dans chaque arc d'entrée.

MdC à réseau de processus dans lequel des modules indépendants appelés processus échangent des jetons à travers des files ordonnées en suivant la règle premier entré premier sorti (FIFO). Les réseaux de processus sont généralement utilisés pour modéliser des algorithmes de traitement numérique du signal. La notion de temps n'est pas présente dans ce modèle, seule la notion de dépendance (qui précède qui) régit l'évolution du modèle. Les modèles de graphe de flots de données qui seront utilisés dans la suite de ce mémoire de thèse sont des sous-ensembles de ce modèle. Les modèles de calcul graphe de flots de données ont été étudiés par Lee et Parks [LP95], ainsi que leur positionnement à l'intérieur du modèle à réseau de processus comme illustré dans la figure 3.6. Chaque MdC est un compromis entre l'expressivité i.e. sa capacité d'expression d'algorithmes et sa prédictibilité i.e. la capacité de prédire le déroulement temporel de l'exécution à la compilation. Si un MdC est capable d'exprimer tous les algorithmes possibles alors il est Turing complet. Les MdCs à réseau de processus forment deux branches, une branche principale autour du modèle à réseaux de processus de Khan (KPN) présenté dans ce qui suit et une branche secondaire incluant des MdCs dérivés de la branche principale car introduisant des caractéristiques additionnelles pour des besoins spécifiques.

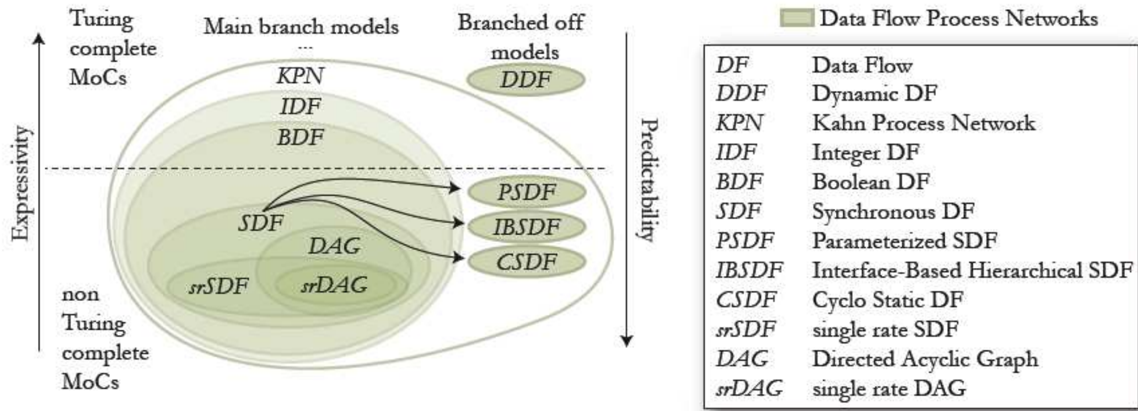

FIGURE 3.6 – Représentation de différents modèles de calcul graphe de flot de données

3.3.1 MdCs de la branche principale

La figure 3.6 présente des sous ensembles du MdC à réseau de processus de Khan (KPN). Ce dernier consiste en des processus continus communicants à travers des files FIFO infinies où l'écriture est non bloquante mais la lecture est bloquante. De ce sous ensemble le MdC KPN est le seul à ne pas être un MdC "Data-Flow Process Networks" (DFG) comme illustré avec le code couleur de la figure 3.6. Les MdCs DFG spécifient d'avantage le comportement de leurs processus quand ils reçoivent et envoient des données. Ces processus sont alors appelés des **acteurs** produisant de manière indépendante des Tokens de données dès qu'ils en reçoivent selon certaines règles. Les différences dans les règles de production distinguent la plupart des MdCs graphe de flot de données :

Graphe de flot de données dynamique (DDF) est un MdC DFG avec des règles de consommation spécifiques. Un acteur du DDF peut consulter un jeton de donnée sans le consommer, ce qui définit une lecture non bloquante sur la file FIFO. Or, dans les MdCs à réseaux de processus de Khan, les lectures sont bloquantes sur les files par définition. Par conséquent le MdC DDF est considéré comme en dehors de la branche principale comme montré dans la figure 3.6, mais reste néanmoins un MdC DFG.

Graphe de flot de données synchrone (SDF) est l'un des plus utilisé des MdC DFG en raison de sa simplicité et sa prédictibilité. Chaque acteur consomme et produit un nombre fixe de jetons à chaque exécution. Le MdC SDF ne permet pas de modéliser des algorithmes avec des branchements conditionnels et n'est donc pas Turing complet. La figue 3.7 illustre un example d'algorithme SDF. Plusieurs spécifications additionnelles dans les règles de productions du présent MdC produisent des sous ensembles. Par example le MdCs **Graphe de flot de données synchrone à taux unique (srSDF)** illustré dans la figure 3.7, est un sous ensemble du MdC SDF ou le nombre de jetons produits et consommés sur un même arc sont égaux. Le MdC **Graphe de flot de données acyclique (DAG)** (figure 3.7) est un SDF qui ne contient pas de cycle. Son dérivé le MdC **Graphe de flot de données acyclique à taux unique (srDAF)** montré

dans la figure 3.7 est un SDF qui ne contient pas de cycle et consomme et produit le même nombre de jetons sur chaque arc.

Graphe de flot de données booléen (BDF) est un SDF avec des acteurs additionnels spéciaux appelés commutateur (switch) et sélectionneur (select) comme illustré dans la figure 3.7. L'acteur commutateur produit des jetons à sa sortie vraie (ou fausse) s'il reçoit un jeton booléen de valeur vraie (ou fausse) sur son entrée de contrôle. Grâce à cette règle, le MdC BDF est capable de modéliser tous les algorithmes et est donc Turing complet. Si les jetons de contrôle sont des entiers alors le MdC est dénommé **Graphe de flot de données entier (IDF)**.

Graphe de flot de données cyclo-statique (CSDF) est une extension du SDF dans lequel des schémas de production et de consommation ont été rajoutés. Comme illustré dans la figure 3.7 un modèle (3,1) équivaut à produire successivement 3 jetons puis 1 jeton de données lors de l'exécution.

La liste de MdCs présentés ici n'est pas exhaustive. Plusieurs autres MdC graphe de flot de données (DFG) ont été étudiés dans le cadre de projets de développement d'outils de simulation et d'implémentation d'applications DSP comme PREESM[Pel+14b], Ptolemy[Pto14], ou StreamIT[GAA10]. Dans la suite nous présentons quelques un d'entre eux.

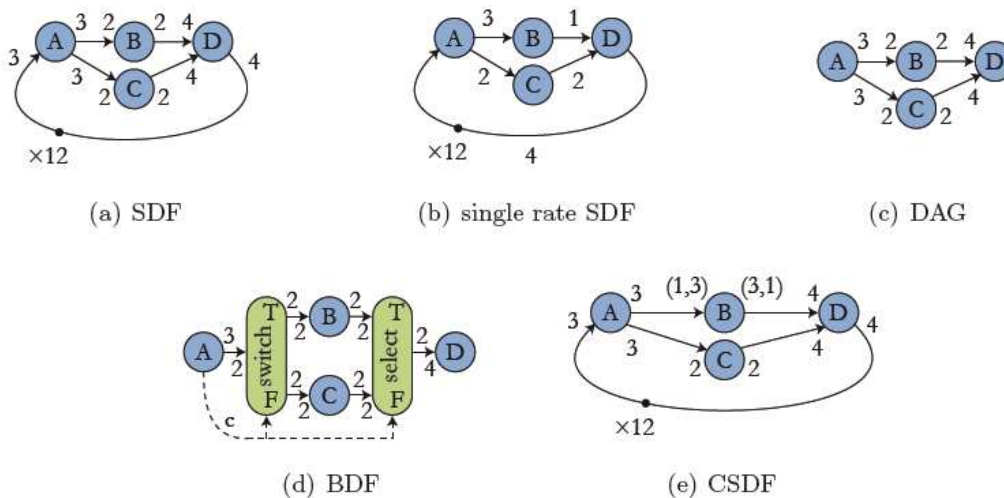

FIGURE 3.7 – Les modèles de calcul à réseau de processus

3.4 Les environnements de développement pour les applications DSP spécifiées avec le modèle de calcul graphe flot de données

Les MdCs DFG modélisent seulement comment le calcul progresse mais n'ont pas de sémantique exécutive. Par exemple, pour un acteur exécutant une opération sur des tokens d'entrée, ne dispose pas de la description des actions à entreprendre pour produire d'autres

Tokens. Un MdC DFG doit être accompagné d'un modèle d'exécution permettant de décrire l'évolution du système (acteurs, communication, structures de données, etc) sur le matériel. Se basant sur ces modèles de calcul DFG et intégrant différents modèles d'exécution, plusieurs outils d'implémentation des applications DSP ont vu le jour. Dans ce qui suit nous en citons quelques uns :

Ptolemy[Pto14] Est un environnement de conception, de simulation et d'implémentation de plusieurs types d'applications, notamment les applications de type DSP. Développé par l'université de Berkeley, il supporte la combinaison de plusieurs MdCs comme le dataflow synchrone et dynamique, le MdC à événements discrets, le MdC à machine d'états finis ou les réseaux de processus. Il permet aux utilisateurs de modéliser facilement leur applications grâce à une interface graphique sous forme de digrammes de blocs. Il comporte un modèle d'exécution complexe permettant de cibler plusieurs plates-formes matérielles comme les DSP.

StreamIT[GAA10] Est un environnement de programmation proposé par l'institut technologique du Massachusetts (MIT) basé sur le MdC dataflow synchrone, avec une syntaxe propre basée sur des structures de calcul hiérarchiques et composables appelées *stream*. Il est accompagné d'une plateforme de compilation permettant d'implanter des applications de type DSP selon un modèle d'exécution parallèle sur des machines multi-coeurs.

PREESM[Pel+14b] Est une plateforme pour la simulation et la programmation des applications de traitement du signal sur des DSP multi-coeurs. Il est basé sur une extension du MdC SDFG et propose à l'utilisateur une interface permettant de créer et de paramétrer le graphe d'entrée. Il est accompagné d'un modèle d'exécution avec une gestion de la mémoire et un ordonnancement de l'application sur l'architecture ciblée.

Simulink[Mok00] Est un outil graphique de programmation pour les applications de contrôle et de traitement du signal. Proposé par MathWork, il permet de simuler, de modéliser et d'analyser les applications grâce à son interface permettant de générer des diagrammes de blocs représentant des fonctions de calcul reliées par des arcs pour les échanges de données. Il est fortement couplé au langage MATLAB et permet de générer des programmes pouvant s'exécuter sur différentes machines.

Lustre[Hal+91] Est un langage de programmation basé sur le MdC graphe de flot de données synchrone pour le développement des systèmes réactifs dans plusieurs domaines critiques incluant les applications de type DSP. L'utilisateur exprime son application à travers une syntaxe propre au langage qui sera compilée pour produire un code séquentiel optimisé.

FAUST[FOL11] Est un langage de programmation proposé par le centre national de la création musicale, basé sur le MdC fonctionnel pour la simulation et l'implémentation des applications du traitement du son et de la musique. L'utilisateur doit utiliser la syntaxe du langage basée sur Haskell pour construire son application sous forme de combinaisons de fonctions de différents ordres ou un diagramme de blocs fonctionnels qui sera compilé pour générer du code C++.

A l'aide de différents outils de développement comme ceux cités précédemment, les applications de type DSP ont été implémentés sur différentes architectures de calcul : CPUs, DSP,

FPGA, ASIC, GPU, etc.

Pour répondre au besoin grandissant en puissance de calcul, il est inévitable aujourd'hui de s'orienter vers des architectures matérielles parallèles et hétérogènes de type cluster, introduisant des microprocesseurs multi-cœurs soutenus par des accélérateurs many-cœurs. En effet, à cause de plusieurs limitations physiques comme l'effet de Joule ou la consommation énergétique, la tendance de la technologie de calcul de haute performance s'oriente vers l'utilisation de plusieurs unités de calcul connectées entre elles à l'image du Tianhe-2, l'ordinateur le plus puissant au monde selon la liste Top500 [Top]. Regroupant 16,000 nœuds contenant chacun deux processeurs Intel Xeon E5-2692v2 à 12 cœurs et un accélérateur Intel Xeon Phi 31S1P à 57 cœurs, il atteint une performance crête de 33,86 Petaflops/s faisant de lui le meilleur candidat pour le traitement des données massives dans différents domaines de calcul, notamment le DSP. Dans ce qui suit nous présentons des implémentations d'applications de type DSP sur des architectures parallèles et hétérogènes.

3.5 Implémentations d'applications de type DSP sur architectures parallèles et hétérogènes

Les applications DSP ont été implémentées sur des architectures parallèles et hétérogènes en se basant soit sur les modèles de programmation parallèles et généralistes présentés dans le chapitre 2 ou bien sur des Modèle de Programmation (MdP) basés sur le MdC DFG. Dans ce qui suit nous présentons différentes implémentations de ce type d'application. Nous les classons par niveau d'abstraction en utilisant les critères définis dans la section 2.3 du chapitre 2.

3.5.1 Implémentation bas niveau (Fortement explicite)

Traditionnellement, les applications DSP sont implémentées sur architecture parallèles en se basant sur les MdPPs bas-niveau. Dans [Puj+12], les auteurs présentent une implémentation avec Pthread d'une application de décodage de vidéo stéréoscopique sur architecture multi-coeurs Intel Core i-7. Ils décomposent manuellement les étapes de l'algorithme en 2 blocs qui sont traités parallèlement par 2 threads indépendants communicants à travers une mémoire partagée. L'algorithme traite itérativement un ensemble d'images stéréoscopiques, une image par itération. Dans [Cam08], les auteurs utilisent MPI pour implémenter une application de lancé de rayons sur un many-coeurs Cray XD-1 avec mémoire distribuée. Ils décomposent manuellement l'algorithme en une centaine de processus communicants par envoi de messages à travers le réseau. Chaque processus traite itérativement un nombre défini de rayons. Dans [KLH14], une application de traitement "Beamforming" séquentiel d'images à ultra-sons est implémentée sur GPU. Les auteurs utilisent OpenGL et OpenCL pour décomposer l'algorithme sous forme de blocs exécutés par des threads GPU sous la forme SIMD. Chaque bloc de threads traite un sous ensemble 2D de l'image source. Dans [Kim+10], les auteurs implémentent une application d'imagerie médicale pour générer une image "B-mode". L'algorithme est manuellement décomposé en 4 "Kernels" lancés séquentiellement sur le GPU pour pro-

duire une seule image. L'algorithme est répété itérativement pour traiter un ensemble d'images source. Ces implémentations bas-niveau sont efficaces et produisent de bonnes performances en comparaison à une implémentation séquentielle. Cependant, l'effort de programmation est important. En effet, pour tous les outils utilisés (Pthread, MPI, OpenCL, CUDA, OpenGL), l'utilisateur a dû manuellement effectuer les tâches suivantes :

1. Décomposer l'algorithme sous la forme de parties parallèles en déroulant des boucles de données (échantillons) pour exploiter le parallélisme de données, ou des boucles contenants les étapes de l'algorithme pour exploiter le parallélisme de tâches.
2. Gérer les allocations et libérations de la mémoire pour chaque thread selon son utilisation de la mémoire.
3. Synchroniser les threads en utilisant des barrières pour des synchronisations générales, ou des Mutex, ou Sémaphore pour des synchronisations individuelles.
4. Gérer les communications entre threads par des copies de données sur les architectures à mémoire partagée, ou par échange de message sur les architectures à mémoire distribuée.
5. Placer (affecter) les parties parallèles sur les threads ou les processus disponibles pour distribuer le travail.

Au delà de toutes ces contraintes, il est à relever que ces outils bas-niveau sont spécifiques à certaines architectures : Pthread et MPI pour les multi-coeurs ou many-cœurs. CUDA, OpenGL pour les GPU. Finalement, ils sont aussi restreints à un type d'architecture mémoire : MPI pour les architectures à mémoire distribuée, les autres pour les architectures à mémoire partagée.

3.5.2 Implémentation à base de bibliothèques (Explicite)

Les implémentations d'applications DSP utilisent communément des librairies de fonctions prédéfinies comme Aquila, OpenCV, ou VXL. Plusieurs de ces librairies ont été adaptées pour exploiter le parallélisme des CPU ou des accélérateurs hétérogènes comme le GPU. Dans [MKM14], les auteurs implémentent une application de détection du mouvement et d'identification des silhouettes humaines à l'aide de OpenCV-GPU. Cette implémentation présente l'avantage de traiter enligne (temps réel) des images en haute définition sans nécessiter un développement coûteux. Cependant, elle est restreinte à un seul GPU et ne permet pas d'-exploiter les multi-cœurs du CPU. D'autre librairies de ce type existent comme GPUCV [HOR+10], FFTW [Fri99], NPP [SK10]. Elles permettent d'exploiter le parallélisme sans avoir à connaître les caractéristiques des architectures. Cependant, elles ne permettent pas d'exprimer du parallélisme de tâches avec dépendance de données. En outre, les performances obtenues sont limitées du fait qu'elles ne gèrent pas pour la plupart d'entre elles des difficultés telles que la répartition des charges, le recouvrement des temps de communications et de calcul, ou l'optimisation des allocations mémoires. Ces librairies ne sont également pas adaptées à des architectures hybrides incluant des unités de calculs hétérogènes avec des architectures mémoires distribuées. Finalement, l'utilisation des ces librairies restreint fortement la possibilité d'exploiter efficacement un cluster hétérogène.

3.5.3 Implémentation à base de directives (Partiellement implicite)

Des MdPP haut-niveau à base de directives sont communément utilisés pour implémenter des applications DSP sur architectures parallèles et hétérogènes. Les utilisateurs annotent explicitement leur code séquentiel en utilisant des directives spécifiques. Traduites lors de la compilation, elles permettent de décomposer automatiquement l'algorithme en déroulant des boucles de données ou de tâches, ce qui augmente la productivité et la portabilité du code. Dans l'utilisation par défaut, l'utilisateur ne s'occupe pas des communications et de la synchronisations des threads. Cependant, pour optimiser les performances, il doit intégrer des directives complémentaires ou manipuler des variables d'environnement. Dans [Pha13], les auteurs utilisent OpenMP pour implémenter une application de transformation des distances géodésiques sur architecture multi-coeurs. Ils insèrent des directives pour paralléliser les boucles sur les données 2D d'une image où chaque thread OpenMP traite une colonne de données. L'algorithme est répété itérativement jusqu'à convergence. OpenMP est un standard très utilisé sur architecture many-coeurs, multi-cœurs et accélérateurs GPU et Xeon Phi dans sa version 4.0. Il permet de dérouler des boucles ou de construire un ensemble de tâches dépendantes. Cependant, son modèle d'exécution est structuré sous la forme "Fork-Join" ce qui restreint l'expression d'un graphe de tâche avec des dépendances arrières (dépendance inter-itérations) comme est le cas dans certaines applications exprimées avec des modèles de calcul DFG cités dans la sections 3.3. Une implémentation OpenMP est discutée plus en détail dans la partie IV. Un autre modèle proche de OpenMP est OpenACC. C'est un standard pour l'implantation à base de directive sur accélérateurs comme les GPU et Xeon-Phi. Dans [Col12], les auteurs présentent l'implémentation d'une application de simulation d'ondes élastiques sur architectures hétérogènes multi-coeurs et GPUs à l'aide d'OpenACC. Ils décomposent manuellement l'algorithme en deux processus MPI contenant chacun plusieurs régions parallèles exécutées sur les GPUs. L'auteur obtient de bonnes performances en comparaison à l'effort fourni. Cependant, il a été nécessaire d'insérer manuellement des directives spécifiques pour optimiser les allocations mémoire, les communications et la distribution des régions parallèles sur les GPUs. D'autres outils à base de directives comme OmpSS ou HMPP ont été utilisés pour implémenter des applications DSP sur architectures parallèles et hétérogènes, cependant ils présentent les mêmes contraintes. Ils est en effet difficile d'exprimer des applications représentées sous forme de graphe de tâches complexe avec des dépendances inter-itérations ou des mono-dépendances (dépendances inter-itérations d'un acteur vers lui même). En outre, certains d'entre eux ne supportent pas les dépendances de données permettant de lancer la tâche dés que ses données d'entrée sont disponibles. Ils sont aussi restreints à des architectures mémoires partagées pour la plupart. Finalement, l'utilisateur doit connaître certaines des spécificités matérielles pour optimiser l'exécution par exemple la distribution des calculs sur les unités de calcul.

3.5.4 Implémentation à base de Runtime de tâches (Fortement implicite)

Il est également possible de s'appuyer sur des supports exécutifs "Runtime" pour implémenter des applications de type DSP sur architectures parallèles et hétérogènes. A l'aide de

librairie spécifique ou de directives, l'utilisateur décompose explicitement son algorithme sous la forme d'un graphe de tâches, généralement un graphe orienté acyclique (DAG). Ce DAG de tâche présente les mêmes caractéristiques que le MdC DAG présenté dans la section 3.3. Les arrêtes du graphe représentent des dépendances de données. Une tâche est exécutée dés que ses dépendances d'entrée sont toutes satisfaites. Avec ces MdPP, l'utilisateur ne gère explicitement aucune des contraintes liées aux parallélisme et à l'hétérogénéité comme les communications, les synchronisations, ou la distributions des tâche. Dans [Mah+11], les auteurs présentent une implémentation à l'aide de StarPU d'une application de détections de coins et de contours dans une base volumineuse d'images. Ils chargent les images à traiter en mémoire centrale, qui sont exécutées ensuite par l'algorithme décomposé explicitement en DAG de tâches. Chaque tâche exécute sur CPU ou sur GPU une partie de l'algorithme sur des données d'entrée pour produire un résultat intermédiaire. Les auteurs obtiennent de bonnes performances en comparaison à une implémentation séquentielle, notamment grâce à la distribution dynamique des tâches sur l'architecture permettant d'occuper équitablement toutes les unités de calcul. Cependant, il a été nécessaire de construire explicitement le DAG de tâche. Les auteurs ont dû manipulé l'API de StarPU pour créer les "Codlets", les tâches et les buffers. Ils ont dû relier les tâches dépendantes entre-elles à l'aide de fonctions et de structures de données spécifiques. Finalement, ils ont du explicitement soumettre les tâches à exécuter au Runtime. D'autres modèles à base de tâches comme Cilk [Blu+95], X-Kaapi [GBP07], TBB [Rei07] et Ptask [Ros+11] sont utilisés pour implémenter des applications DSP avec un haut niveau d'abstraction. Cependant, ils présentent tous les mêmes contraintes de manipulation de librairie.

3.5.5 Implémentation à base d'environnements à MDC DFG (DSP spécifique)

Dans la section 3.4 nous avons présenté des outils permettant de facilement simuler ou implémenter des algorithmes de type DSP sur différentes architectures. L'utilisateur exprime son application sous la forme d'un DFG. Dans [GAA10], les auteurs utilisent StreamIT pour implémenter une application de codage-décodage MPEG sur architecture parallèle (MIT Raw Architecture). Ils expriment l'algorithme à l'aide d'une syntaxe spécifique pour construire un DFG synchrone. La décomposition de l'algorithme ainsi que les communications et les synchronisation sont gérés implicitement par StreamIT. L'ordonnancement des tâches sur l'architecture est statique et est effectué lors de la compilation. Cet ordonnancement est suffisant pour des applications dites statiques (à temps de calcul et de communications statiques) sur des architectures homogènes. Cependant, il présente des inconvénients dans le cas des applications dynamiques à branchements conditionnels (temps de calcul et de communication dynamiques) représentées généralement par des modèles comme les DFGs booléen ou entier. Ces modèles sont capables d'exprimer tous les algorithmes, mais nécessitent un ordonnancement dynamique s'adaptant à l'évolution de l'application. D'autres algorithmes modélisés par des DFG statiques peuvent également présenter des temps de calcul dynamique dépendant des données traités. Ces algorithmes nécessitent aussi un ordonnancement à l'exécution pour équilibrer les charges de travail sur les unités de calcul. Finalement, l'ordonnancement

dynamique permet une indépendance par rapport à l'architecture et est mieux adapté aux architectures hétérogènes permettant de mieux exploiter les performances de chaque unité. D'autres outils comme Ptolemy, PREESM [Pel+14a], Syndex, ou LUSTRE sont des outils qui proposent des abstractions équivalentes avec des spécificités particulières, mais ils présentent les même inconvénients et sont restreints aux architectures homogènes. Des environnements plus récents essaient de répondre à certaines des ces contraintes. Dans [Huy+12], les auteurs présentent une extension de StreamIT permettant d'ordonnancer des applications sur architectures hétérogènes incluant des GPUs. Cependant, l'ordonnancement reste statique et est effectué lors de la compilation. Dans [HHR10], les auteurs proposent un environnement de compilation "source-to-source" transformant une description SystemC [Gro02] d'une application DSP vers un code exécutable sur GPUs. Cependant, ils utilisent Syndex pour ordonnancer les tâches statiquement après avoir décrit manuellement l'architecture CPU-GPU sous format XML. Finalement, dans [Sch+13], les auteurs proposent un environnement appelé DAL, basé sur OpenCL permettant d'implémenter des applications DSP spécifiées avec des MdC DFG sur des architectures parallèles et hétérogènes plus complexes. Cependant, l'ordonnancement est effectué manuellement et statiquement par l'utilisateur.

3.5.6 Récapitulatif et discussion

A partir des différentes implémentations des applications DSP sur architectures parallèles et hétérogènes présentées ci-dessus, nous les caractérisons comme suit : ce sont des applications itératives traitant un ensemble de données digitales, par exemple, un ensemble d'images 2D. Ces algorithmes sont usuellement modélisés à l'aide des MdC DFG, ou ils sont représentés sous la forme d'un graphe orienté composé de nœuds représentants les opérateurs et arrêtés représentants les échanges de données. Pour une implémentation efficace de ces applications sur architectures parallèles et hétérogènes, il est nécessaire d'exploiter ces deux caractéristiques. Premièrement, pour extraire les opérateurs capables d'être exécutés en parallèle, l'utilisateur doit exprimer son algorithme sous forme d'un ensemble de tâches à l'aide de threads ou processus. Il doit également gérer les communications et la synchronisation de ces threads sur architectures à mémoire partagée ou distribuée selon les dépendances de données de l'application. Deuxièmement, afin d'exploiter la capacité des accélérateurs à exécuter efficacement certaines fonctions. L'utilisateur doit décharger l'exécution de certaines tâches sur les accélérateurs disponibles. Pour ce faire, il doit allouer la mémoire nécessaire sur le périphérique, il doit copier les données vers ce périphérique, lancer le calcul, puis finalement, récupérer les résultats et libérer la mémoire. Troisièmement, en considérant l'aspect itératif de ces applications, le programmeur doit dérouler la boucle principale de l'application pour augmenter le nombre de tâches disponibles et ainsi augmenter l'occupation des unités de calcul. Il doit donc dupliquer les threads ou processus en charge des itérations en garantissant la cohérence des données entre itérations. Finalement, le programmeur doit aussi faire face à des difficultés liées à l'architecture comme : les temps de communication qui doivent être masqués par les exécutions, l'équilibre des charges entre les unités de calcul considérant les temps de transfert et d'exécution des tâches, la gestion efficace des buffers mémoires empêchant de créer des goulots d'étranglement dûs aux allocations et libération répétés des espaces mémoires et les

synchronisations globales pouvant retarder le calcul.

Appliquer toutes ces règles et contraintes dans l'implémentation des applications DSP est une tâche complexe. En effet, comme présenté dans la sous-section 3.5.1, en utilisant les MdPP bas-niveau comme Pthread, CUDA, ou MPI, le programmeur doit combiner la manipulation de plusieurs APIs, langages ou extensions de langages pour explicitement décomposer son algorithme, gérer la mémoire, synchroniser les threads, transférer les données et distribuer les tâches sur les unités de calcul. Ces modèles sont proches de la machine et offrent peu d'abstraction. En outre, ils sont restreints à un type d'architecture. Leur utilisation réduit fortement la productivité. Alternativement, le programmeur peut utiliser des librairies de fonctions virtuelles (GpuCV ou NPP) comme présenté dans la sous-section 3.5.2. Ces librairies permettent au programmeur d'être libéré d'une programmation bas niveau. Cependant, elles présentent une opacité par rapport à d'autres caractéristiques matérielles comme les communications, en ne permettant pas de les optimiser. Pour implémenter des applications DSP sur architectures parallèles et hétérogènes, le programmeur peut aussi utiliser les modèles à base de directives comme présenté dans la sous-section 3.5.3 avec OpenMP et OpenACC. Ils offrent, en effet, une plus grande abstraction en terme de décomposition des algorithmes, synchronisation et communications. Cependant, dans le cas des applications DSP, il est difficile d'utiliser cette approche pour exprimer un DFG complexe (des branchements inter-itérations, des mono-dépendances), ou la synchronisation par dépendance de données. En outre, afin d'augmenter les performances, le programmeur doit introduire des directives supplémentaires et initialiser des variables d'environnements pour mieux gérer les allocations, les communications et la répartition des charges. Une autre solution plus adaptée pour implémenter ces applications sur architectures parallèles et hétérogènes sont les modèles à base de tâches ou Runtime. Comme présenté dans la sous-section 3.5.4, utiliser des modèles comme StarPU ou X-Kaapi permet d'exprimer les applications DSP modélisées en DFG avec un haut niveau d'abstraction. En effet, le programmeur décompose son algorithme sous forme de DAG de tâches avec des dépendances de données. Cependant, il doit manipuler une large API de fonctions et de structures de données pour exprimer son application. Il doit construire dynamiquement le DAG de tâches en déroulant la boucle principale de l'algorithme. Finalement, il doit gérer les surcoûts dûs aux allocations mémoire, la gestions des tâches et la distribution dynamique.

Le programmeur peut également utiliser les modèles basés sur les MdC DFG adaptés à l'implémentation des applications DSP comme StreamIT, PREESM ou DAL. Le programmeur décompose son algorithme avec un DFG. Les communications et synchronisations sont gérées automatiquement par le modèle d'exécution accompagnant ces outils. Cependant, comme discuté dans la sous-section 3.3.1, une partie des ces modèles ne permettent pas l'implémentation sur des architectures hétérogènes. Une autre partie sont des outils statiques, où l'ordonnancement et le placement des tâches est déterminé lors de la compilation. D'autres outils présentent l'inconvénient d'avoir à distribuer manuellement les opérateurs sur les unités de calcul. Donc, il n'existe pas selon notre opinion des modèles basés sur le MdC DFG pour implémenter des applications DSP sur cluster hétérogène avec un modèle d'exécution (Runtime) dynamique à haut niveau d'abstraction. Dans [Lee89] ; [Bal+98], les auteurs présentent les avantages d'un runtime offrant une distribution de tâches dynamique par rapport à une distribution statique. En effet, comme présenté dans la sous-section 3.3.1, même si certains MdCs DFG sont décid-

ables et prédicables lors de la compilation, de nombreuses applications de type DSP ont des temps de calcul dépendants des données traitées. Ces applications nécessitent une distribution dynamique des tâches au fur et à mesure que l'exécution progresse afin d'occuper toutes les unités de calcul équitablement. Finalement, le placement dynamique est plus adapté à des architectures hétérogènes ou les temps de calcul dépendent aussi de l'unité de calcul. Afin de répondre à cette problématique, nous proposons dans les parties II et III deux MdPPs basés sur le MdC DFG permettant de répartir les charges dynamiquement au cours de l'exécution. Le premier modèle "PACCO" que nous présentons à été développé dans les travaux de [Bou+12]. C'est un outil DFG basé sur le modèle BSP avec un placement statique. Nous proposons de l'enrichir en rajoutant une fonctionnalité de migration de tâche permettant d'équilibrer les charges de calcul à l'exécution. Cependant, ce modèle présente des limitations que nous détaillerons, notamment la nécessite de représenter l'architecture de calcul et les barrières de synchronisation globales de son modèle d'exécution BSP. Par conséquent, nous présentons un deuxième MdPP DFG basé sur le Runtime dynamique StarPU développé par l'équipe Runtime de l'INRIA de Bordeaux, permettant d'abstraire à un haut niveau l'architecture de calcul. Dans ce modèle nous développons plusieurs fonctionnalités dynamiques pour l'adapter à l'implémentation des applications DSP.

Deuxième partie

Mig-PACCO : La migration de tâches dans un modèle de programmation BSP avec Pipeline

Pour être utile, un MdPP doit offrir une abstraction des fonctionnalités d'implémentation liées à l'architecture. Une de ces fonctionnalités est la distribution des tâches ou des Threads sur les unités de calcul. En effet, plusieurs de ces modèles basés sur le MdC DFG présentés dans chapitre 3 proposent un placement statique et éventuellement manuel des acteurs sur les éléments de calcul. Cette approche de placement est simple à implémenter du point de vue du modèle d'exécution. Elle a aussi l'avantage de ne pas entraîner de surcoûts à l'exécution. Cependant, il est à relever que le placement statique convient aux applications prédictibles et statiques sur des architectures homogènes, mais est non adapté à l'optimisation des applications dynamiques sur des architectures hétérogènes. En outre, un placement statique s'il est manuel, force le programmeur à connaître les spécificités de l'architecture : il implique un reparamétrage pour chaque architecture de calcul. Dans cette partie nous présentons une approche de migration de tâches sur un MdPP à placement statique et manuel. Le but est d'ajouter un niveau d'abstraction au modèle en question sans pour autant réduire ses performances. La migration de tâches que nous présentons peut être utilisée non seulement dans l'optimisation automatique des performances temporelles ou énergétiques en adaptant "en-ligne" le placement initial des tâches spécifié par l'utilisateur, mais elle peut aussi être utilisée dans d'autres cas comme la tolérance aux fautes dans le cas où un élément de calcul se met à dysfonctionner ou bien dans le cas d'un arrêt volontaire d'un élément de calcul de la part de l'utilisateur. Dans le chapitre suivant (chapitre 4), nous présentons d'abord l'environnement auquel la migration à été ajoutée, à savoir le modèle PACCO. Nous discutons ensuite ses limites et de l'intérêt de la migration. Puis, dans le chapitre 5, nous présentons la migration de tâches, nous détaillons son implémentation et la validons finalement sur 3 cas d'utilisation.

Chapitre 4

Les modèles BSP et PACCO

4.1 Introduction

Un modèle de programmation permet de faire le lien entre la conception haut niveau des applications et leur implémentations bas niveau des architectures. Il a pour principaux buts de faciliter l'implémentation des applications et en même temps d'optimiser son efficacité. Nous présentons dans la section 4.2 le modèle BSP. Ensuite, dans la section 4.3 nous présentons quelques environnements de programmation basés sur ce modèle. Finalement, dans la section 4.4 nous présentons l'environement PACCO qui permet d'implémenter des applications spécifiées avec un DFG sur un cluster hétérogène, en se basant sur le modèle d'execution BSP.

4.2 Le modèle BSP

C'est un modèle d'abstraction et de programmation développé par Leslie Valiant [Val90] dans les années 80 pour modéliser des algorithmes sur des machines parallèles à mémoire distribuée. La machine est abstraite par un ensemble de processeurs exécutant des threads (processus) différents. Chaque processeur est équipé d'une mémoire locale rapide. Les processeurs sont connectés entre eux à travers un réseau et peuvent se synchroniser globalement. Cette abstraction correspond à plusieurs types d'architectures matérielles, notamment aux "clusters" hétérogènes. Un algorithme exprimé avec le modèle BSP est décrit comme suit :

La figure 4.1 illustre l'exécution sous le modèle BSP. Dans ce qui suit nous présentons plus en détails les caractéristiques de ce modèle. Verticalement, le calcul est sous la forme d'étapes séquentielles appelées super-étapes. Chaque super-étape comporte les étapes suivantes :

1. Les processus effectuent des calculs locaux sur chaque processeur.
2. Les processus échangent mutuellement leur données en communicant à travers le réseau d'interconnexion. Un recouvrement des calcul locaux et des communications est possible.
3. Les processus attendent une synchronisation globale qui se produit lorsque tous les processus ont terminé.

Horizontalement, un calcul parallèle est effectué sur l'ensemble des processeurs. L'ordre d'exécution des processus et de l'établissement des communications n'a pas d'importance.

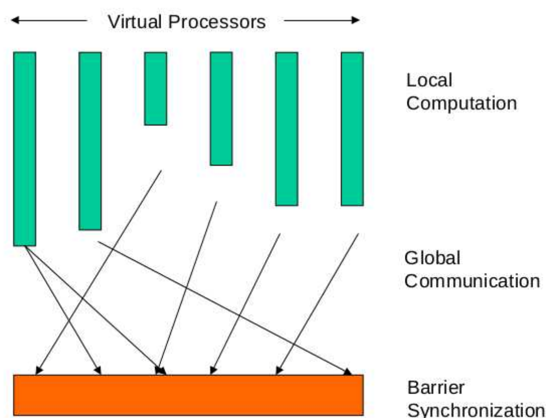

FIGURE 4.1 – Illustration des étapes de calcul du modèle BSP

4.2.1 Les communications

Dans les systèmes parallèles, il est difficile d'estimer individuellement chaque communication. En effet, il est complexe de calculer le temps de chacune d'entre elles dans le cas où elles sont effectuées en concurrence. En outre, des phénomènes comme la contention ou l'interaction avec des systèmes extérieurs peuvent perturber cette estimation. Le modèle BSP considère les communications en masse. Cela permet de calculer une borne supérieure du coût global des communications dans une super-étape. Si le nombre maximum de messages reçus ou envoyés pour l'ensemble des processus de la super-étape est h et si g représente la perméabilité ou la capacité de l'architecture à délivrer des messages de taille égale à 1, alors, le coût global des communications de taille m dans la super-étape est de mgh.

4.2.2 La synchronisation globale

La barrière de synchronisation permet de garantir la cohérence des données lors du calcul et évite les situations d'interblocage (deadlock). Cependant elle entraîne un coût l lié à :

1. La variation entre les dates de fin des calculs locaux de chaque processeur. Cela entraîne une sous utilisation des processeurs ayant une charge de travail moindre. Un équilibrage de charge est nécessaire pour éviter ce coût.
2. Le temps nécessaire à la synchronisation. Ce temps est fortement couplé aux nombres de processeurs, la taille du réseau et à la politique de synchronisation.

4.2.3 La prédictibilité du modèle

Un des grands intérêts du modèle BSP est sa prédictibilité. En effet, il est possible d'estimer le coût maximal C_{bsp} d'un algorithme BSP comme la somme des coûts C_{ss} des n super-étapes le composant :

$$C_{bsp} = \sum_{i=1}^{n} C_{ss}$$

Le coût C_{ss} d'une super-étape est déterminée comme la somme de trois parties : une partie de calcul effectif w_i sur les p processeurs, une partie de communication et une partie de synchronisation.

$$C_{ss} = \max_{i=1}^{p} (w_i) + \max_{i=1}^{p} (h_i * g) + l$$

Dans le cas d'un recouvrement calculs-communications, le coût d'une super-étape devient :

$$C_{ss} = \max(\max_{i=1}^{p}(w_i), \max_{i=1}^{p}(h_i * g)) + l$$

Le modèle BSP peut être utilisé efficacement pour implémenter des algorithmes sur des machines parallèles. Il faut néanmoins veiller à bien répartir les charges de calcul sur l'ensemble des processeurs. Cela a pour effet de réduire le temps d'attente pour la synchronisation. Il faut aussi répartir les communications entre les processus afin de réduire le terme h. Finalement, il est également intéressant de minimiser le nombre de super-étapes n dans la modélisation d'un algorithme.

4.3 Environnements d'implémentation basés sur le modèle BSP

Récemment le modèle BSP a connu un grand engouement dans l'environnement des technologies de traitement des données missives sur des clusters parallèles. En effet, depuis quelques années, la firme "Google" l'utilise dans ses technologies de recherche et d'analyse sur des graphes de grande échelle dans des outils comme Pregel [Mal+10] ou MapReduce [DG08]. En outre, il est introduit aussi dans des projets "Open-sources" de la communauté Apache comme variation de l'environnement Hadoop [Whi09] destiné à faciliter la création d'applications distribuées et échelonnables "scalables" sur plusieurs nœuds de calcul. Cela a donné naissance aux projets Apache Hama [Seo+10] et Apache Giraph [Mal+10] utilisés par plusieurs grandes industries du web comme "Facebook". D'autres travaux récents visent également à développer le modèle BSP pour mieux l'adapter à des architectures ou paradigmes de calculs spécifiques. Les plus importants d'entre eux sont le modèle BSP décomposable (D-BSP) [TK96] visant à réduire le coût global du modèle en regroupant les processeurs en sous groupes capables d'exécuter indépendamment leur super-étapes et le modèle BSP étendu (E-BSP) [JW96] visant à estimer le coût du modèle plus précisément en répartissant de façon efficace les communications dans une super-étape. Le modèle BSP a été implémenté par différentes organisations et communautés donnant naissance à différents environnements de programmation sur architectures parallèle. Dans ce qui suit nous citons quelques un d'entre eux.

BSPlib [Hil+98] est la bibliothèque standard d'implémentation des algorithmes avec le modèles BSP sur des clusters. L'utilisateur introduit des primitives simples permettant de

décomposer l'algorithme selon le mode SPMD. Les parties parallèles sont ainsi exécutées sur différentes éléments de calcul. Les communications et les synchronisations sont gérées par le support exécutif de la bibliothèque. Comparée à MPI, BSPlib permet d'écrire facilement des programmes basés sur le modèle BSP.

MulticoreBSP [Yze14] est une implémentation JAVA ou C du modèle BSP sur muti-cœurs. Son interface est similaire à celle de BSPlib avec un nombre de primitives réduit. L'utilisateur bénéficie d'abstraction sur les communications et les synchronisation. MulticoreBSP gère les Threads et les communications en se basant sur la localité des données ce qui lui permet d'atteindre de hautes performances.

BSML [Ver+03] est une bibliothèque combinant le modèle BSP et le langage fonctionnel Caml. Elle permet à l'utilisateur d'exécuter parallèlement son programme fonctionnel sur des architectures muti-cœurs en introduisant dans son code des primitives spécifiques. Les fonctions parallèles sont extraites et exécutées sur des processeurs distincts.

BSGP [HZG08] est une extension du modèle BSP sur des architectures de type GPUs. L'utilisateur exprime son programme avec une syntaxe proche du langage C en introduisant des barrières de synchronisation. Ce code est ensuite compilé pour générer les super étapes ou des streams parallèles exécutant chacun un kernel GPU. Les dépendances de données sont également gérées implicitement. Cependant, l'environnement BSGP ne gère pas la distribution des calculs entre plusieurs CPUs et/ou GPUs.

BSPonMPI [Bsp] est une implémentation du modèle BSP basée sur MPI ce qui lui permet de couvrir plus d'architectures parallèles à mémoires distribuées que les autres implémentations. L'utilisateur introduit dans son code les primitives du standard BSP qui sont traduites à la compilation vers des primitives MPI. Le programme est exécuté selon le modèle BSP en se basant sur des communications par passage de messages.

4.4 PACCO

Dans le chapitre 3, nous avons présenté des environnements de programmation basés sur le modèle de calcul DFG. Or, ce modèle permet seulement de décrire comment le calcul de l'algorithme progresse indépendamment de son implémentation sur l'architecture de calcul. Ces environnements incluent par conséquent un modèle. Il permet d'implémenter le modèle de programmation et de ce fait influe sur plusieurs de ses caractéristiques, par exemple, son efficacité ou son indépendance par rapport à l'architecture, etc. Une harmonie dans la combinaison entre les deux modèles (modèle de calcul et modèle d'exécution) composant un environnement (modèle) de programmation parallèle est nécessaire. Dans cette section nous présentons un modèle de programmation appelé PACCO pour "Parallel communication-computation overlap" basé

sur le modèle DFG comme modèle de calcul et le modèle BSP comme modèle d'exécution. Cet environnement proposé par [Bou+12] offre un haut niveau d'abstraction. L'utilisateur décrit son algorithme avec un DFG. Le parallélisme est exprimé explicitement mais la décomposition, la synchronisation et les communications sont implicites. La distribution des tâches est manuelle et statique. Il permet de couvrir des architectures à mémoires partagées comme celles à mémoire distribuée et couvre aussi des unités de calcul hétérogènes : CPU et accélérateurs GPUs. Dans ce dernier cas, le programmeur doit tout de même fournir les kernels. Dans ce qui suit nous présentons sa conception.

4.4.1 Conception

PACCO est un environnement pour l'implémentation d'application itératives de type DSP sur architectures parallèles et hétérogènes. Il est basé sur une spécification DFG de l'application et permet d'abstraire les communications qui peuvent être soit : (1) synchrone, où les calculs et les communications sont séquentialisés, ces dernières sont réalisées à la fin de la super-étape. (2) asynchrone, où les communications sont recouvertes partiellement ou entièrement par les calculs. Ainsi, des temps de transfert non négligeables peuvent être masqués durant les phases de calcul. Ces calculs peuvent être distribués (mappés) sur une seule unité de calcul et/ou sur plusieurs. L'architcture est spécifiée par un graphe orienté dans lequel les unités de calcul sont connectées par des arcs représentant les liens de communication qui peuvent être de différentes natures : PCI-express, InfiniBand, Ethernet, etc. Les acteurs reliés (mappés) à une unité de calcul sont exécutés séquentiellement selon un ordre FIFO "First in first out". Cependant, les acteurs distribués sur plusieurs unités de calcul sont exécutés en parallèle. La cohérence de données est assurée par le modèle BSP. Les processus ou threads associés à chaque élément sont synchronisés par une barrière globale. Ainsi, chaque processus exécute son sous-DFG en mode synchrone ou asynchrone et attend que les autres processus aient terminé l'exécution de leur sous-DFG. La barrière de synchronisation marque la fin de l'itération et permet de passer à l'itération suivante. Ce modèle d'exécution permet de former un pipeline logiciel : les itérations de l'algorithme peuvent être distribuées sur l'ensemble de l'architecture. Dans la figure 4.6 nous montrons une illustration d'une exécution pipeline dans le modèle PACCO.

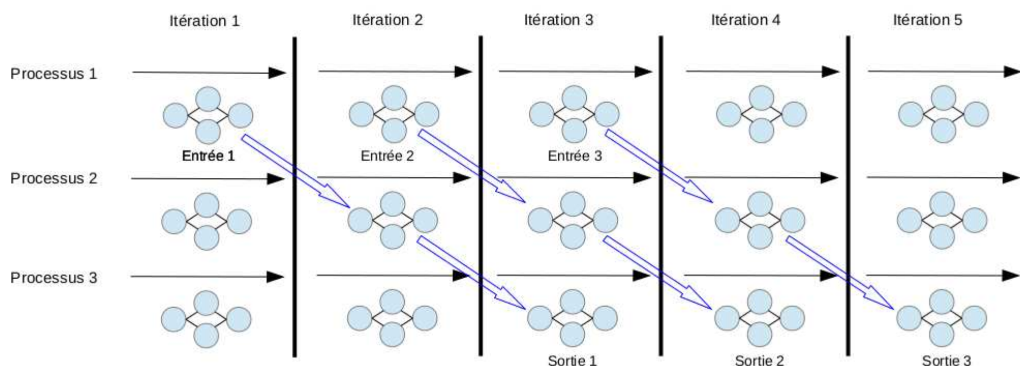

FIGURE 4.2 – Flot de conception du modèle PACCO

L'environemennt PACCO permet d'alléger l'effort de programmation des applications DSP en abstrayant les fonctionnalités bas niveau suivantes :

1. Les allocations mémoires sur unité de calcul CPU et GPU.
2. Les communications entre tâches.
3. Les synchronisations des tâches.

Dans ce qui suit nous présentons plus en détails cet environnement au travers d'un exemple d'implémentation d'une application DSP.

4.4.2 Points d'entrée du modèle

Pour implémenter une application de type DSP avec l'environnement PACCO, deux points d'entrées sont nécessaires. Le programmeur doit produire le graphe d'architecture représentant la plate-forme de calcul et le graphe d'application représentant l'algorithme sous forme de DFG.

Le graphe de l'architecture est exprimé avec le langage XML. Les nœuds représentent les unités de calcul composant l'architecture. Elles sont spécifiées par l'utilisateur en indiquant leur nom, leur type et leur interfaces. Les arcs, quant à eux, représentent les canaux de communication. Ils sont spécifiés manuellement par l'utilisateur en décrivant leur type (PCI-e, Ethernet, Infiniband). Dans la figure 4.3, nous présentons un exemple de graphe d'architecture d'un cluster de calcul composé de 2 nœuds incluant chacun un CPU et 3 GPU.

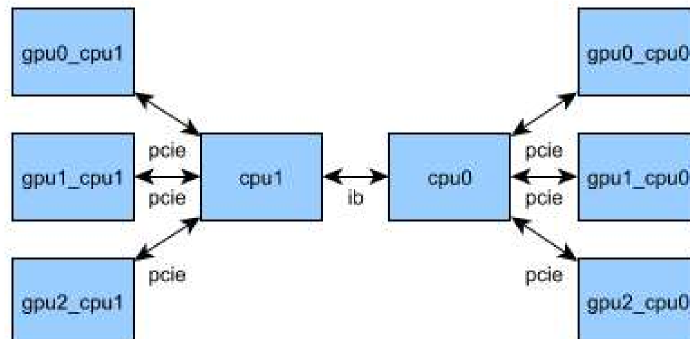

FIGURE 4.3 – Graphe d'architecture

Le graphe d'application est également exprimé avec le langage XML. Il comporte des nœuds représentant les acteurs et des arcs représentant les flots de données. La sémantique opérationnelle de ce graphe est celle du DAG à taux unique qui fait partie du sous ensemble du modèle DFG synchrone décrit dans la sous section 3.3.1 du chapitre 3. Le graphe est acyclique et les jetons produits et consommés sur un même arc sont égaux à l'unité. Un acteur est exécuté dés que ses jetons (tokens) d'entrée sont disponibles. Une fois lancé, il consomme un jetons en exécutant la fonction qui lui est associée et produit un jeton de sortie. Ce dernier est transféré à travers l'arc de communication pour être stocké dans la file qui lui est associée. Les lectures

sur les files sont bloquantes mais les écritures ne le sont pas. Le type et la taille des données que chaque arc transporte sont fixes et spécifiées par l'utilisateur.Également, la fonction associée à chaque nœud et ses arguments sont uniques et spécifiés par le programmeur.Dans la figure 4.4, nous présentons un exemple de graphe d'application synthétisé à partir d'une description XML.

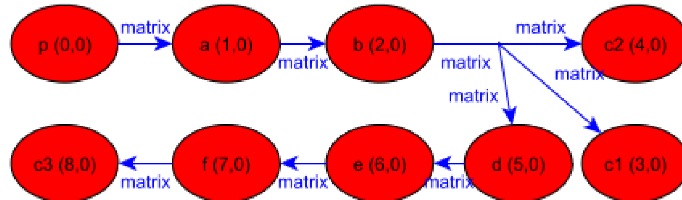

FIGURE 4.4 – Graphe d'application

Le programmeur doit également "mapper" son application en précisant pour chaque acteur l'élément de calcul de destination. Un exemple de "mapping" possible de l'application de la figure 4.4 sur l'architecture de la figure 4.3 est présenté dans la figure 4.5. Les acteurs b, a, p sont exécutés en parallèle sur les éléments $gpu1_cpu0$, $gpu2_cpu0$ et $cpu0$. Par contre les acteurs d, e et f sont exécutés séquentiellement sur le GPU $gpu1_cpu1$. Le programmeur doit finalement encapsuler les fonctions que les acteurs exécutent et les intégrer à son code sous forme de classe C++.

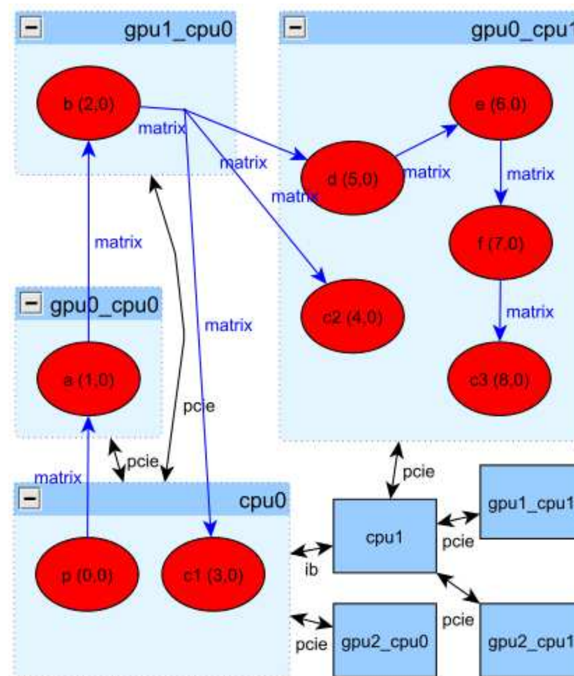

FIGURE 4.5 – "Mapping" entre graphe d'application et graphe d'architecture

Pour résumer, voici les tâches que l'utilisateur doit accomplir :

1. Exprimer en XML l'application sous forme de DFG, plus précisément "Single rate" DAG.
2. Exprimer en XML l'architecture de calcul sous forme de graphe orienté acyclique.
3. Encapsuler les fonctions exécutées par les acteurs sous forme de classe C++ en surchargeant les méthodes virtuelles Init et Exec d'une classe mère.
4. Compléter les "fonctions de résolutions" chargées de définir la taille des espaces mémoire à allouer.

La figure 4.6 illustre le flot proposé en faisant apparaitre, les étapes de traitement et les points d'entrée (en gris) de l'environnent PACCO.

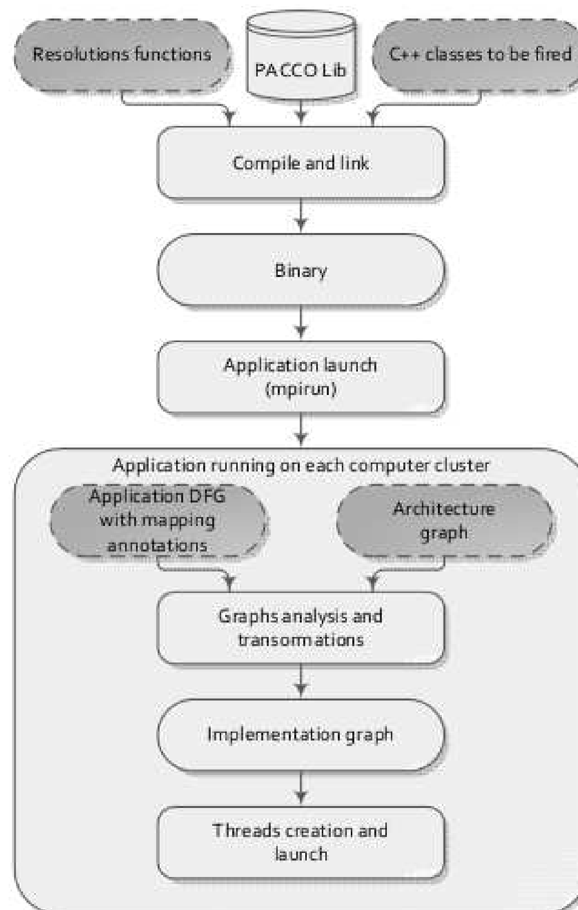

FIGURE 4.6 – Illustration des étapes de traitement et de la conception de l'environnement PACCO. Les parties en gris représentent les points d'entrée

4.4.3 L'initialisation du modèle

L'étape d'initialisation du modèle correspondant dans la figure 4.6 au cercle "graphs analysis and transformations" a pour objectif de générer le graphe d'implémentation. Ce graphe contient les informations nécessaires permettant de déterminer le chemin du flux de données,

les allocations des buffers mémoire et la distribution des acteurs sur les éléments de calcul. Ainsi, chaque processus peut identifier la taille et le type des allocations à effectuer, les fonctions à exécuter et les adresses de transfert en entrée et sortie. Cette initialisation inclut les étapes suivantes détaillées plus loin : l'ordonnancement des acteurs, l'insertion des "buffers", l'optimisation des allocations mémoires et finalement, le calcul de la latence d'exécution pour chaque acteur.

L'ordonnancement Afin de garantir la cohérence de données, l'ordonnancement dans l'environnement PACCO est régi par un algorithme récursif. Pour chaque élément de calcul, un nœud est ordonnancé si et seulement si tous ses jetons sont disponibles. Sinon on tente d'ordonnancer ses prédécesseurs. L'ordre d'ordonnancement appelé rang, ainsi calculé pour chaque acteur est inscrit sur les figures dans le premier élément du couple se trouvant après son nom. Lors de l'exécution, le processus commence par exécuter l'acteur avec le rang le plus faible.

Insertion des "buffers" mémoires Le graphe d'application contient des arcs représentant le transit des jetons de donnée entre les acteurs. Du point de vue de l'implémentation, ces arcs représentent des buffers permettant de stocker les données générées et consommées par les acteurs. L'insertion des buffers dans l'environement PACCO prend en considération : les informations spécifiées par l'utilisateur concernant le type et la taille des données transitant sur les arcs, le "mapping" des acteurs sur les éléments de calculs et le mode d'exécution (synchrone, asynchrone) spécifié par l'utilisateur. En effet, le chemin et le type d'allocation peuvent varier selon la distance physique i.e. type de connexion entre l'élément de calcul source et l'élément de calcul destination. La figure 4.7 illustre l'insertion de "buffers" de l'application présentée dans la figure 4.4, mappée sur l'architecture de la figure 4.3 selon le "mapping" de la figure 4.5. Par example, la connexion entre l'acteur b et $c2$ nécessite 4 buffers $bn2$, $bn9$, $bn10$ et $bn11$. Tandis que la connexion entre b et $c2$ nécessite 2 buffers seulement, $bn2$ et $bn9$.

Les buffers sont insérés comme suit : à la sortie de chaque acteur est inséré un buffer alloué sur le même élément de calcul. Pour chaque buffer inséré, le chemin physique le connectant avec l'acteur de destination est recherché sur le graphe d'architecture. Sur chacun chemin, d'autres buffers sont insérés pour chaque élément de calcul traversé pour atteindre l'élément de calcul de destination. La taille des buffers dépend du mode de communication : synchrone ou asynchrone et des éléments de calcul source et destination. Pour le premier mode, une allocation simple est effectuée. Pour le cas asynchrone avec connexion entre éléments de calcul différents, une allocation double est effectuée permettant un recouvrement entre le calcul et les communications.

Optimisation des allocations mémoires Elle vise à réduire le nombre de buffers alloués sur un élément de calcul. Un algorithme de coloriage de graphe est utilisé. Les buffers aptes à être fusionnés sont coloriés de la même couleur. L'objectif est de minimiser le nombre de couleurs permettant de colorer l'ensemble du graphe. Dans la figure 4.8 est illustré le résultat de l'optimisation sur le graphe de la figure 4.7. Le buffer $bn3$ dans cette figure est un résultat

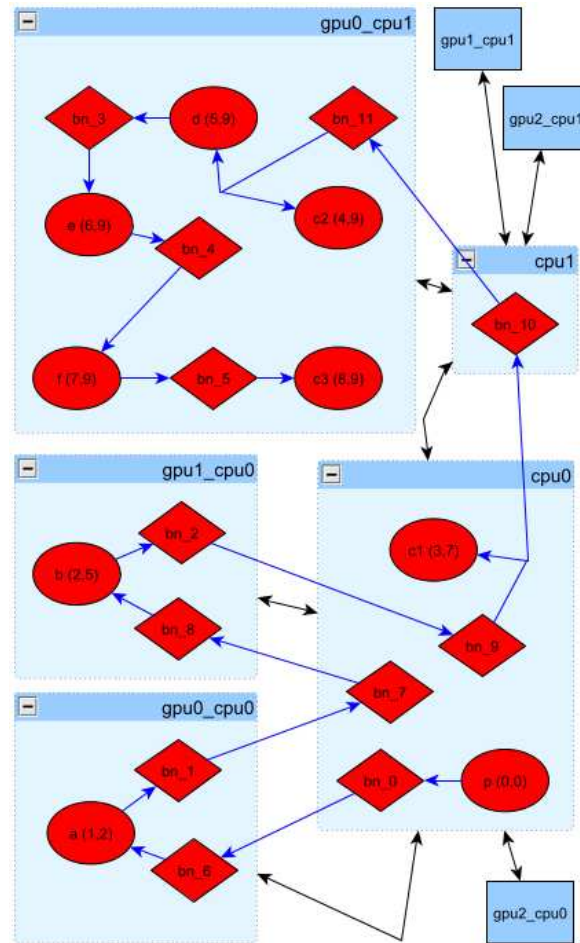

FIGURE 4.7 – Illustration de l'insertion des buffers mémoire dans le graphe d'implémentation

de fusion des buffers $bn3$ et $bn5$ de la figure 4.7. Le buffer $bn11$ est le résultat de la fusion de $bn11$ et $bn4$ de la figure 4.7. Une fusion entre deux buffers génère un buffer de taille égale au maximum des deux tailles. L'ordre d'exécution (rang) des acteurs représenté sur l'élément droit de leur l'étiquette permet de garantir la cohérence des données entre lectures et écritures des différents acteurs.

Calcul de la latence d'exécution A cause du modèle d'exécution en pipeline de l'environnement PACCO, les acteurs "mappés" sur des éléments de calculs différents traitent des données d'itérations algorithmique différentes. Tandis que ceux mappés sur le même élément de calcul traitent séquentiellement les données d'une seule itération. En effet, le calcul d'une seule unité de donnée d'entrée est distribuée sur plusieurs itérations d'exécution du modèle BSP. Le nombre d'itération dépend du "mapping" de l'application spécifié par l'utilisateur. Chaque couple de buffers connectés entraîne un cycle de retard. Par example, la latence de l'acteur $c2$ de l'application spécifiée avec le graphe d'implémentation dans la figure 4.7 est de 9 itérations. i.e pour traiter la première donnée d'entrée, l'acteur doit se déclencher à l'itéra-

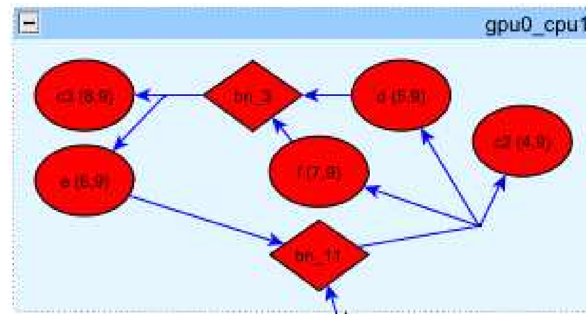

FIGURE 4.8 – Illustration de l'optimisation des allocations dans le cas de l'insertion des buffers mémoire du graphe d'implémentation de la figure 4.5

tion 9. Autrement, il traitera des données non valides. Par conséquent, chaque acteur doit se déclencher lors d'une itération donnée seulement si le numéro de l'itération est supérieur ou égal à sa latence d'exécution.

4.4.4 L'exécution du modèle

A l'exécution, un Thread POSIX est crée pour chaque élément de calcul. Tous les Threads ont accès au graphe d'implémentation. Lors de son lancement chaque Thread parcourt ce graphe afin d'identifier les communications et les acteurs qu'il devra gérer. Le lancement des communications et des fonctions associées aux acteurs suit le modèle BSP mais diffère selon le mode de communication précisé par l'utilisateur :

1. Sans recouvrement calcul-communication, une super étape consiste à :
   (a) Lancer les transferts CPU-CPU.
   (b) Attendre la fin des transferts sur tout le cluster.
   (c) Lancer les transferts CPU-GPU.
   (d) Attendre la fin des transferts CPU-GPU.
   (e) Exécuter séquentiellement les acteurs selon leur ordonnancement sur chaque élément de calcul (exécution parallèle sur l'ensemble des éléments de calcul).
   (f) Attendre la fin des exécutions des acteurs sur les éléments de calcul.
2. Avec recouvrement calcul-communication, une super étape consiste à :
   (a) Lancer les transferts asynchrones de/vers tous les éléments de calculs et exécuter séquentiellement les acteurs selon leur ordonnancement sur les éléments de calcul.
   (b) Attendre via la barrière de synchronisation la fin de toutes les tâches précédentes sur tous les éléments de calcul.

4.5 Conclusion

Dans ce chapitre nous avons étudié le modèle d'exécution BSP. Ce modèle permet l'implémentation parallèle d'algorithmes en proposant une abstraction simple des architectures cibles. Plusieurs environnements de programmation parallèles basés sur ce modèle ont vu le jour. Ils permettent de réduire l'effort du programmeur sur l'expression des communications et des synchronisations grâce à l'abstraction offerte par le modèle BSP. PACCO est l'un de ces environnements proposant une abstraction supplémentaire sur la décomposition et la modélisation des algorithmes sous forme de DFG. L'utilisateur modélise simplement son algorithme avec un graphe et n'a pas besoin de manipuler des API ou des langages pour cela. Cependant, cet environnement souffre de quelques inconvénients liés notamment à son modèle d'exécution basé sur le BSP. En effet, à cause de la synchronisation globale du modèle BSP, une bonne répartition des charges de calcul entre les processeurs est nécessaire afin d'optimiser les performances temporelles ou énergétiques. Cependant, dans l'outil PACCO le placement des tâches est manuel et statique tout au long de l'exécution ce qui réduit fortement son intérêt dans des cas de mauvais équilibre des charges. Pour répondre à cette problématique nous présentons dans le chapitre suivant, chapitre 5, une fonctionnalité de migration de tâche pouvant être utilisée, entre autres, pour migrer des acteurs lors de l'exécution afin de mieux répartir les charges de calcul.

Chapitre 5

La migration de tâches dans PACCO

5.1 Contexte et motivations

Comme présenté précédemment dans le chapitre 4, l'utilisateur PACCO spécifie manuellement le placement de chaque acteur du graphe d'application sur les unités de calcul composant l'architecture. Ce placement manuel est en outre statique et ne peut être modifié lors de l'exécution. Cette restriction génère de nombreux inconvénients du point de vue de l'efficacité du modèle de programmation PACCO :

1. L'équilibrage de charge. L'outil PACCO est basé sur un modèle d'exécution BSP. Une mauvaise répartition des charges de calcul entraîne sous utilisation de certains éléments de calcul et rallonge la durée de la super-étape.
2. Le placement manuel est fait a priori de l'exécution. Il est cependant difficile pour l'utilisateur de réussir une bonne répartition des charges dans les cas suivants : application à gros grains ne présentant pas suffisamment de parallélisme au regard de celui de l'architecture, applications dynamiques à temps de calcul et de communications variables, applications "data-dépendante" dont les traitements dépendent des données, unités de calcul hétérogènes, etc.
3. L'utilisateur est tenu de connaître les caractéristiques de l'architecture pour effectuer un placement efficace.
4. Le placement des acteurs est spécifique à l'architecture de calcul, donc non portable.

La migration de tâches développée en détail dans ce chapitre permet comme illustré dans la figure 5.1, de modifier durant l'exécution le placement des acteurs d'une application initialement placée par l'utilisateur. Elle peut être utilisée pour répondre à certains des inconvénients cités ci-dessus. Dans ce qui suit, nous présentons différents scénarios mettant en avant l'apport de la migration de tâches.

Placement dynamique pour l'optimisation du débit Dans ce scénario nous nous intéressons à augmenter le débit de calcul d'une application. Le placement initial est spécifié a priori par l'utilisateur. Cependant, ses estimations sur les temps de calcul des tâches ne sont pas exactes et son placement est non optimal. En utilisant la fonctionnalité de migration "en ligne", il est possible de modifier le placement d'un acteur afin de mieux équilibrer la

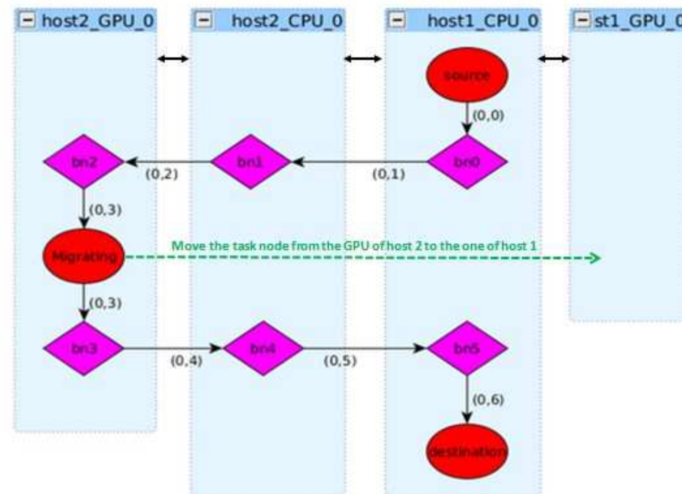

FIGURE 5.1 – Illustration du processus de migration de tâches (acteur) dans l'environnement PACCO

charge de travail. Il suffit alors de détecter les temps d'attente des éléments de calcul dans la super-étape. Une fois l'élément de calcul le moins utilisé détecté, on migre alors un des acteurs de l'élément de calcul le plus utilisé vers celui choisi auparavant. Le choix de l'acteur peut se faire en considérant deux informations : le temps de calcul de l'acteur sur l'élément original et la différence de puissance entre les éléments de calcul (origine vs destination). Le but est d'équilibrer au mieux la répartition globale.

Réduction de la consommation d'énergie Dans ce scénario, nous voulons réduire la consommation d'énergie du cluster utilisé tout en sauvegardant le débit de production, ou d'une manière plus générale en cherchant à respecter une contrainte de temps. Le but est de concentrer le calcul sur un minimum d'éléments tout en garantissant que le temps de calcul de la super-étape est inférieur ou égal à la contrainte de temps. Dans ce cas de figure, il faut migrer les acteurs de l'élément de calcul le plus faible et les répartir sur les autres éléments de calcul en minimisant les communications. Cela libère l'élément de calcul qui peut être désactivé et réduit les communications ce qui réduit la consommation d'énergie.

Libération d'un nœud de calcul Dans ce troisième scénario, on a besoin de libérer un nœud de calcul du cluster sans pour autant arrêter l'application. Les raisons peuvent être diverses, par exemple pour des raisons de maintenance du nœud. Il faut pour cela migrer tous les acteurs de cet élément de calcul vers les autres nœuds de calcul. La migration des acteurs se fait individuellement ou par groupe comme nous le détaillerons par la suite.

5.2 Stratégies de migration

La migration de tâches intervient dans le modèle d'exécution de l'environnement PACCO basé sur le modèle BSP itératif. Le but est de déplacer "en ligne" (durant le calcul) l'exécution de l'un des acteurs de l'application de son élément de calcul originel vers un autre élément de calcul. Cependant, dans le cadre du traitement d'applications "streaming", une contrainte forte se dégage. Il est en effet intéressant de minimiser l'impact de cette migration sur le flux de sortie afin de ne pas interrompre la production des données. La migration d'un acteur dans une application de type "streaming" va engendrer la modification du chemin de données le reliant aux acteurs amont et aval, comme représenté sur la figure 5.2. Cette figure illustre le cheminement des données à partir de l'acteur producteur "source" jusqu'à l'acteur de consommation "sink" en passant par des buffers "bn".

La migration de tâche a été abordée dans plusieurs travaux. Dans [EP96], Ebner et al proposent une solution pour une migration en une seule étape procèdent comme suit ainsi : ils allouent les buffers sur le nouveau chemin nommé "migration path" sur la figure 5.2. Ils transfèrent ensuite les données contenues dans les buffers de l'ancien chemin appelé "original path" vers les nouveaux buffers. Finalement, ils créent la tâche migrée contenant les mêmes caractéristiques que la tâche originelle en la mappant sur l'élément de calcul de destination. Cette stratégie est simple à mettre en œuvre, cependant, elle peut avoir un coût élevé. En effet, selon la granularité de données transférées et/ou les caractéristiques matérielles du canal de communication, cette étape peut rallonger le temps de l'itération pendant laquelle a lieu la migration, ce qui ralentit la production des données de sortie. Dans [RR+09], les auteurs proposent Mig-BSP une solution de replacement dynamique basée sur la migration de tâche dans un cluster. Cette solution se base sur l'estimation du coût de la migration pour sélectionner le bon candidat à migrer et l'élément de destination. Cependant, ici aussi, les auteurs utilisent la même approche pour effectuer la migration. Le coût de cette dernière peut se révéler important dans certains cas. Une autre stratégie de migration pouvant être adoptée est d'abandonner les données des buffers de l'ancien chemin et de les régénérer à nouveau dans le nouveau chemin. Cette approche peut être efficace à condition que le coût des calculs qui sont rejoués 2 fois ne soit pas prohibitif. Aussi, il faut régénérer le flux entrant de données, ce qui nécessite de stocker les données d'entrée.

Pour minimiser l'impact sur le flux de sortie des données durant le processus de migration, l'idée est de répartir cette dernière sur plusieurs itérations en vidant progressivement les données du chemin originel et en même temps d'alimenter le chemin de migration. Cette approche illustrée dans la figure 5.2 permet de conserver les données déjà produites et de les dissiper progressivement, sur plusieurs itérations, dans le flux en aval "downstream" et en parallèle de rediriger les nouvelles données reçues par le flot en amont "upstream" pour être traité par la nouvelle tâche migrée. Cette approche permet par conséquent de minimiser l'impact temporel de la migration. Dans ce qui suit nous présentons comment cette approche est implémentée dans l'environnement PACCO.

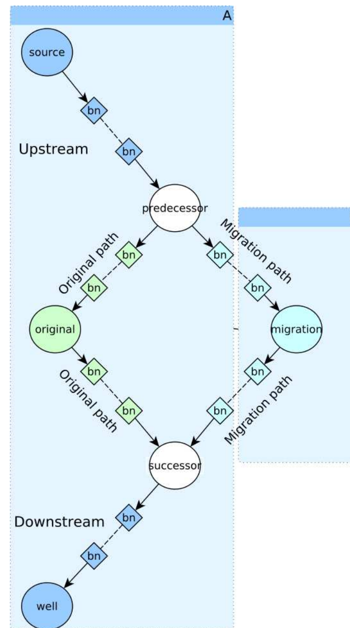

FIGURE 5.2 – Illustration de la migration d'une tâche dans une application de type "streaming"

5.3 Implémentation

Comme présenté dans le chapitre 4, PACCO permet d'implémenter une application spécifiée avec un DFG à l'aide du modèle d'exécution BSP. Ce modèle exécute itérativement des super-étapes au sein desquelles sont réalisés : (1) le calcul. (2) les communications. (3) la synchronisation globale. Comme illustré dans figure 5.3, l'action de migration quand elle est sollicitée, par example, par un ordonnanceur, intervient après l'étape 3 de synchronisation globale. C'est un Thread de synchronisation, chargé de faire attendre tous les autres Threads alloués aux éléments de calcul par un mécanisme de rendez-vous au niveau de la barrière globale qui effectue la migration. Il modifie le graphe d'implémentation en intégrant le nouvel acteur et en modifiant les chemins de données. En effet, notre stratégie de migration consiste à répartir ces actions sur plusieurs itérations du modèle d'execution en vidant progressivement les anciens chemins de données et en alimentant progressivement les nouveaux. Chaque Thread identifie les actions qui lui ont été attribuées en interrogeant le graphe d'implémentation modifié.

A la fin de l'itération pendant laquelle une demande de migration est faite, le Thread de synchronisation modifie le graphe d'implémentation comme suit : premièrement, un nouveau nœud représentant l'acteur migré comportant les mêmes caractéristiques que lui est introduit dans le graphe d'implémentation et est placé sur l'élément de calcul de destination. Ce nœud est connecté aux nœuds prédécesseur et successeur du nœud original en introduisant les buffers nécessaires. Cela génère le nouveau chemin de donnée appelé "Migration-path" dans la figure 5.2. La prochaine étape consiste à agir sur les chemins original et de migration pour respectivement les vider et remplir de données. Pour cela, nous calculons un couple associé aux arêtes composant ces chemins. Chaque couple représente la durée de vie de l'arête (les temps

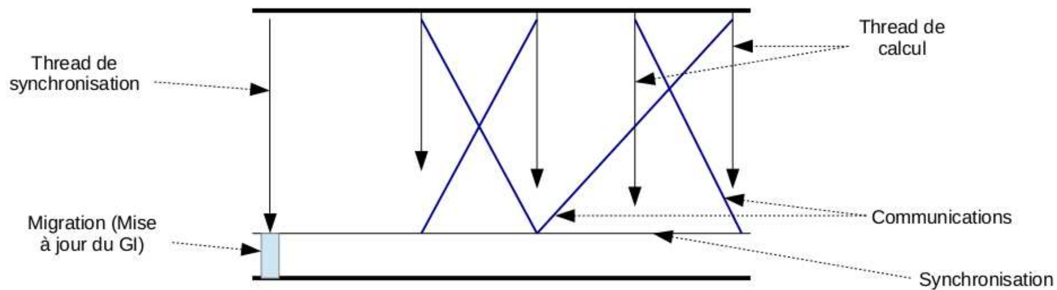

FIGURE 5.3 – Illustration de l'exécution de la migration au sein de la super étape dans un mode d'exécution avec recouvrement calcul-communication. Le Thread de synchronisation met à jour le graphe d'implémentation (GI)

de début et de fin de son arête), plus précisément, le nombre d'itérations avant de démarrer les calculs ou les communications liées à cette arrête et le nombre d'itérations avant d'arrêter les calculs et les communications liés à cette arrête. Ces couples sont mis à jour à chaque itération afin d'autoriser ou pas une communication ou un calcul de l'acteur connecté à ces arrêtes. Ainsi, au cours des prochaines itérations, le chemins originel et de migration seront vidés et remplis en parallèle de manière progressive. Cependant, ces deux chemins peuvent être de tailles différentes nécessitant un nombre d'itérations différent pour être traversés (entièrement vidés ou remplis). En effet, selon la position de l'élément de calcul de destination, le nombre de buffers d'interface présent sur le chemin de migration peut être supérieurs, ou inférieurs à ceux présent sur le chemin originel. Dans ces cas de figure, il est nécessaire de propager des contraintes sur les arcs amont ou aval du nœud originel et du nouveau nœud. Dans la suite nous présentons les 3 cas de figures pouvant se présenter :

Chemin de migration égal au chemin originel Ce cas illustré dans la figure 5.4, est le plus simple du point de vue de l'implémentation. En effet, le nombre de buffer sur le chemin de migration est égal à celui sur le chemin originel et nécessite donc le même nombre d'itérations pour être traversé. Dans l'exemple de la figure 5.4, une unité de données nécessite 3 itérations pour complètement traverser les deux chemins. Par conséquent, pour migrer l'acteur $(dummy - gpu)$ de l'élément de calcul $GPU - 1$ vers $GPU - 0$, il suffit de rediriger progressivement le flux de donnée en amont $(Producer, bn0, dummy - cpu - 1)$ vers le chemin de migration $(bn6, bn7, m - dummy - gpu, bn8, bn9)$. Pour cela on initialise : les couples de temps de début du chemin de migration permettant de le remplir progressivement comme illustré dans l'élément gauche de ses couples (0,1,2,2,3,4) et les étiquettes de temps de fin du chemin originel $(bn1, bn2, dummy - gpu, bn3, bn4)$ permettant de le vider progressivement comme illustré sur l'élément droit de ces couples (0,1,2,2,3,4). Finalement, on calcule le nombre d'itérations nécessaire pour que le chemin originel soit entièrement vidé qui dans ce cas est égal au nombre d'itérations nécessaire pour que le chemin de migration soit entièrement rempli, dans notre exemple 3 itérations. Nous renseignons cette information sur l'acteur $(dummy - cpu - 2)$ afin de remplacer à la bonne itération l'arrête de lecture des données le

connectant au chemin originel par celle le connectant au chemin de migration.

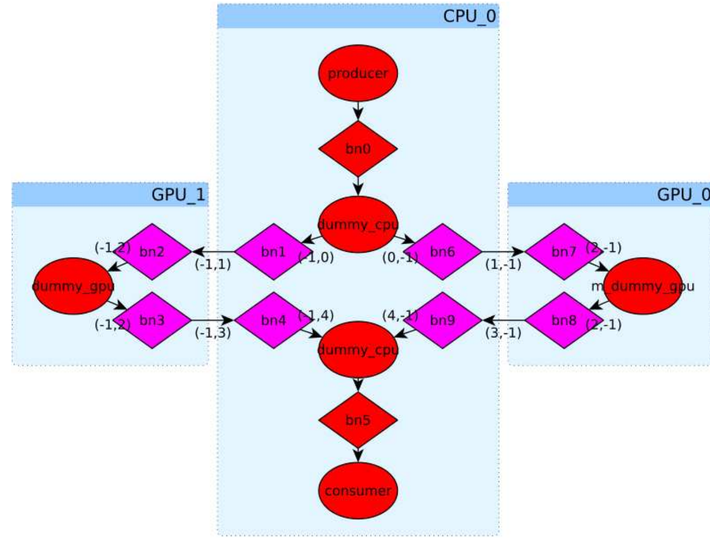

FIGURE 5.4 – Illustration du graphe d'implémentation mis à jour dans le cas de migration d'un acteur avec chemin originel et de migration de tailles égales. Le couple sur chaque arrête représente le nombre d'itérations à partir duquel l'arc sera actif pour l'élément de gauche et le nombre d'itérations à partir duquel l'arc sera inactif pour l'element de droite, cela relativement à l'itération de déclenchement de la migration.

Chemin de migration inférieur au chemin originel Dans ce cas illustré dans la figure 5.5 le chemin originel $(bn1, bn2, bn3, dummy - gpu, bn4, bn5, bn6)$ est plus long en nombre de buffers que le chemin de migration $(bn8, bn9, m - dummy - gpu, bn10, bn11)$. En effet, une unité de donnée produite par $dummy - cpu - 1$ à l'entré du chemin nécessite 5 itérations pour être stockée dans le buffer $bn6$ représentant la fin du chemin, alors qu'elle nécessite seulement 3 itérations pour traverser le chemin de migration. Par conséquent, lors de la migration, le flux de données en amont de $dummy - cpu - 1$ doit être arrêté pendant la différence de temps i.e. 2 itérations pour permettre une synchronisation entre l'arrivée des données traversant le chemin de migration avec celles qui sont vidées du chemin originel. Pour cela nous initialisons les couples des temps de fin du chemin en amont avec la valeur de la différence, 2 dans notre cas. Cette initialisation figure dans l'élément de droite des couples représentés sur les arrêtes du chemin en amont dans la figure 5.5. Il faut également, initialiser les temps de fin du chemin originel, dans notre cas (0,1,2,3,3,4,5,6) et les temps de début du chemin de migration de la même manière en respectant la latence du chemin en amont i.e. 2 itérations, ce qui produit les poids suivants (2,3,4,4,5,6) comme illustré dans la même figure. Finalement, le vidage et la production des données étant synchronisés, il faut initialiser le temps de remplacement des chemins au niveau de l'acteur $(dummy - cpu - 2)$. Cela se produit lorsque toutes les données on été consommées sur le chemin originel et que la première donnée du chemin de migration est disponible pour traitement. Dans notre exemple il est égal à 5 itérations à partir du déclenchement de la migration. Par conséquent, le flux des données en sortie n'est jamais arrêté.

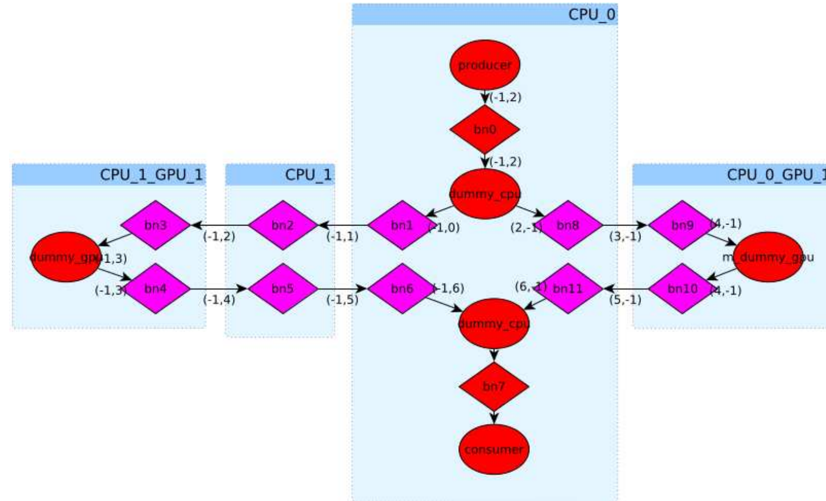

FIGURE 5.5 – Illustration du graphe d'implémentation mis à jour dans le cas de la migration d'un acteur avec chemin originel plus long que le chemin de migration. Le couple sur chaque arrête représente le temps en nombre d'itérations de début pour l'indice de gauche et de fin pour l'indice de droite, relativement à l'itération pendant laquelle la demande de migration à eu lieu

Chemin de migration supérieur au chemin originel Ce dernier cas illustré dans la figure 5.6 est rencontré quand le chemin de migration $(bn6, bn7, bn8, m - dummy - gpu, bn9, bn10, bn11)$ est plus long que le chemin originel $(bn1, bn2, dummy - gpu, bn3, bn4)$ i.e. le nombre d'itérations pour qu'une donnée traverse le chemin originel est plus grand que celui nécessaire pour qu'elle traverse le chemin de migration, à savoir, dans cet exemple respectivement 5 et 3 itérations. Par conséquent, il est nécessaire dans ce cas d'arrêter le chemin de données en aval pour un nombre d'itérations égal à la différence entre les deux chemin, à savoir 2 itérations, mais seulement lorsque le chemin originel est complètement vidé c'est à dire dans 3 itérations dans notre exemple. Concrètement, nous calculons les temps de fin du chemin originel ce qui donne pour notre exemple (0,1,2,2,3,4) et aussi les temps de début du chemin de migration de ce qui donne (0,1,2,3,3,4,5,6). Les temps de début du chemin en aval doivent aussi être initialisés incrémentalement à partir du nombre d'itérations nécessaire pour que les données traversent le chemin de migration à savoir (6,6). Aussi, les temps de fin du chemin en aval doit être calculé à partir du nombre d'itérations nécessaires pour vider tout le chemin originel ce qui donne (4,4). Cela aura pour effet dans un premier temps de consommer progressivement les données vidées du chemin originel durant les 3 premières itérations, de s'arrêter pendant le nombre d'itérations nécessaires (2 itérations) pour que les données soient disponibles au bout du chemin de migration, puis de reprendre progressivement le traitement de ces données.

Nous résumons les étapes d'impélentation de la migration dans l'algorithme 2. Il est question de modifier le graphe d'implémentation en intégrant les actions nécessaires à la migration. La première action est d'identifier dans le graphe d'implémentation le nœud représentant l'acteur à migrer. Ensuite dans l'action 2, il faut créer nœud de migration et initialiser sa structure

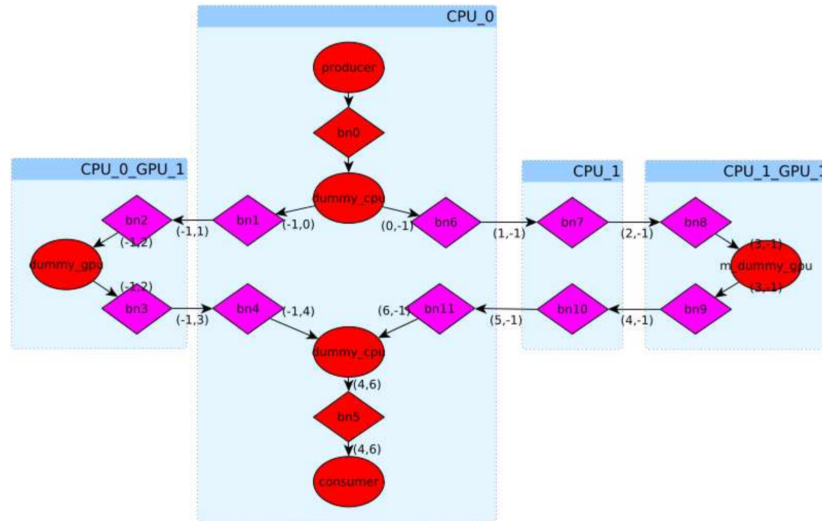

FIGURE 5.6 – Illustration du graphe d'implémentation mis à jour dans le cas de migration d'un acteur avec chemin originel moins long en nombre de buffers que le chemin de migration. Le couple sur chaque arrêtes représente le temps en nombre d'itération de début pour l'indice de gauche et de fin pour l'indice de droite, relativement à l'itération pendant laquelle la demande de migration à eu lieu

et son état à partir du nœud originel. Dans l'étape 3, on associe le nœud de migration sur l'element de calcul de destination. Dans les actions 4 et 5, on introduit les buffers dans le chemin de migration, puis dans les actions 6 et 7 on initialise le couple de temps de fin et de début afin de respectivement vider et remplir les chemins originel et de migration. L'action 9 est exécutée si le chemin de migration est inférieur au chemin original. Dans ce cas on initialise les couples du chemin en amont pour l'arrêter momentanément. Dans le cas contraire, action 12, on initialise les couples du chemin en aval. L'action 10 est reservée pou initialiser le mécanisme de changement d'arc sur le nœud successeur du nœud de migration. Finalement dans l'action 15 on diffuse le graphe d'implémentation sur tous les nœuds de calcul. Dans les itérations suivantes, chaque Thread de calcul traitera ses tâches permettant ainsi d'effectuer progressivement la migration.

5.4 Validation

Dans cette section nous présentons trois scénarios d'expérimentation de notre approche de migration de tâche. Le but est de valider notre stratégie de migration dans des contextes différents et de montrer son intérêt au travers de résultats obtenus avec une application du monde réel. L'application utilisée dans ces trois scénarios est l'application de construction d'une carte de saillance. C'est une application de traitement d'image composée d'acteurs hybrides CPU (*Capture*, *Display*) et GPU ($gpu-mask$, $gpu-FFT$, $gpu-gabor$, $gpu-IFFT$, $gpu-interaction$, $gpu-normalize$). Son implémentation dans l'environnement PACCO est décrite en détail dans le chapitre 8 de la partie IV. Dans notre contexte, l'application

Algorithm 2 Migration de tâche

Input : Implementation graph (IG). Destination computing element (CE_dest). Kernel to migrate (Ker).
Output : Updated implementation graph (IG')

1: $org \leftarrow Research(IG, Ker)$
2: $mig \leftarrow Copy(org)$
3: $IG \leftarrow Mapping(mig, CE_dest)$
4: $IG \leftarrow Buffers_introduce(Pred(org), mig)$
5: $IG \leftarrow Buffers_introduce(mig, Succ(org))$
6: $IG \leftarrow Update_weight(Original_path)$
7: $IG \leftarrow Update_weight(Migration_path)$
8: **if** $Length(Orig_path) > Length(Mig_path)$ **then**
9: $\quad IG \leftarrow Update_weight(Upstream_path)$
10: **end if**
11: **if** $Length(Orig_path) < Length(Migr_path)$ **then**
12: $\quad IG \leftarrow Update_weight(Downstream_path)$
13: **end if**
14: $IG' \leftarrow IG$
15: $Broadcast(IG')$

traite en temps continu une série d'images de taille $512x512$ pixels avec un recouvrement communications-calcul. Dans ces expériences nous redistribuons les acteurs, ou un acteur, sur différents éléments de calcul avec différentes contraintes sans pour autant arrêter le calcul. La plate-forme d'expérimentation est un cluster composé de 2 nœuds connectés par une liaison Infiniband. Chaque nœud de calcul contient un CPU Intel i-7 Core à 4 cœurs physique et deux GPUs Nvidia, une Geforce GTX 285 et une Quadro 4000. Dans ce qui suit nous présentons les 3 scénarios et décrivons leurs contextes d'exécution. Ensuite, nous analysons dans chaque scénarios l'impact des migrations sur le flux de données de sortie.

5.4.1 Scénario d'utilisation de la migration pour améliorer les performances temporelles avec chemin de migration plus long que le chemin originel

Dans ce premier scénario, nous cherchons à augmenter le débit de production des images en équilibrant les charges de calcul plus équitablement. Pour ce faire, nous migrons l'acteur $gpu - normalize$ initialement mappé sur le GPU-0 du nœud A vers le GPU-1 du nœud A comme illustré dans la figure 5.7. Ce déplacement permet de réduire le temps total de la super étape en réduisant le temps de calcul de l'acteur en question sur un GPU Quadro 4000 plus performant que le GTX 285. A noter que le chemin de migration est plus long que le chemin originel. En effet, une unité de donnée nécessite 5 itérations pour traverser le chemin de migration, alors qu'elle ne nécessite que 2 itérations pour traverser le chemin originel.

Les résultats obtenus pour cette première expérience sont présentés dans la figure 5.8. Le

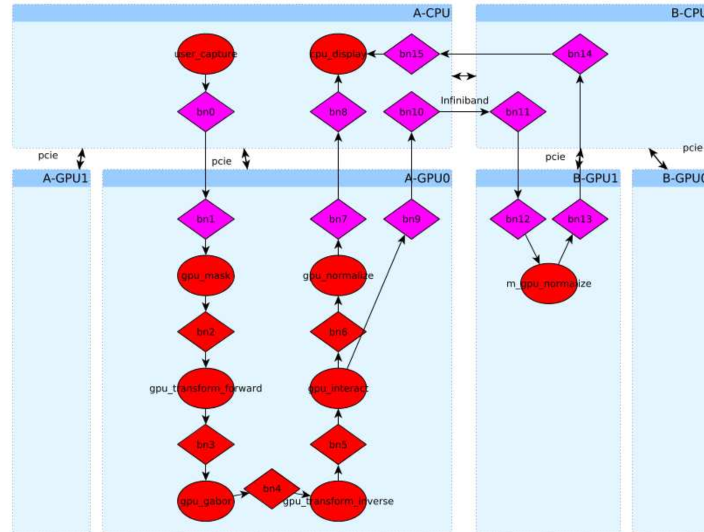

FIGURE 5.7 – Illustration de la mise à jour du graphe d'implémentation de l'application de saillance dans un contexte de migration avec chemin plus long

numéro d'itération est indiqué en abscisse. La courbe verte avec des ronds représente le temps de calcul de la super étape pour chaque itération. Les étoiles sur la courbe verte nous informent si une donnée est sortie lors de l'itération, dans notre cas si une image, a été produite ou pas. La courbe bleue représente le débit de donnée. Ces résultats nous permettent de suivre l'évolution du temps de calcul itération par itération. Ainsi, nous mesurons l'impact de la migration sur l'exécution. La migration est déclenchée dans l'itération 26 et s'exécute progressivement sur les 4 itérations suivantes. Dans ce qui suit, nous distinguons 3 phases dans le calcul :

La première, d'avant la migration ($it < 26$). Le runtime PACCO exécute tout le graphe d'implémentation originel avec des temps relativement constants, 30 ms par itération comme représenté sur la courbe verte. Exceptionnellement, dans la première itération, il construit le graphe d'implémentation , alloue des buffers et exécute les fonctions d'initialisation ce qui génère un sur-coût d'initialisation. Autrement, à cette étape du calcul, une unité de donnée de sortie, i.e. une image, est produite dans chaque itération. Par conséquent, le débit de cette étape est fonction seulement de la durée des itérations ce qui produit un débit autour de 50 FPS.

La deuxième étape concerne la période de la migration ($25 < it < 31$). Lors de celle-ci, le processus de migration met à jour, à la fin de l'itération 26, le graphe d'implémentation de l'application. Cela engendre un sur coût d'environ 15 ms comme montré dans la courbe verte. Dans l'itération 27, le runtime exécute le graphe d'implémentation mis à jour. Il commence à remplir le chemin de migration et à vider le chemin originel. Le temps de calcul de cette itération est réduit à 20 ms du fait que l'acteur $gpu - normalize$ n'est plus exécuté séquentiellement dans le GPU-1. Dans ces deux itérations (26,27) le flux de donnée n'est pas arrêté et le débit est fonction seulement du temps de calcul. Dans les itérations restantes (28,29,30) le chemin en aval est arrêté pour synchroniser l'arrivée des données sur le chemin de migration. Les données de sorties ne sont pas produites durant ces 3 itérations, le débit est donc réduit

pour atteindre environ 16 FPS même si les temps de calcul sont de 20 ms.

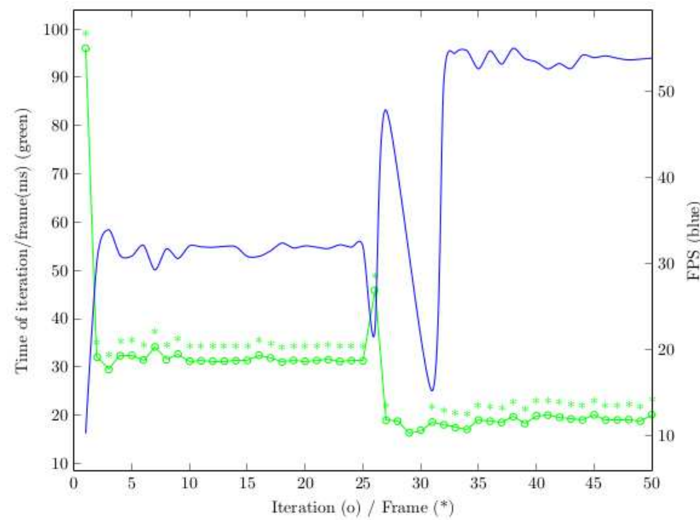

FIGURE 5.8 – Illustration de l'augmentation du débit de production des images de sortie

La troisième étape ($it > 30$) est celle d'après la migration. Le runtime exécute le graphe d'implémentation mis à jour en produisant une image de sortie par itération. Le débit dépend seulement du temps de calcul d'une itération qui est d'environ 20 ms, ce qui produit un débit de 50 FPS. Il est à noter que une fois le processus de migration terminé, le débit est amélioré grâce à une meilleure répartition des charges sur les nœuds de calcul : la durée de la super étape est réduite car le traitement de l'acteur migré est effectué en parallèle sur le GPU-1. Les communications sont masquées par les calculs ce qui réduit fortement leurs impacts sur le débit. Dans cette expérience nous avons validé notre approche de migration avec l'objectif d'améliorer la répartition des charges en ligne.

5.4.2 Scénario pour libérer un élément de calcul avec chemin de migration égal au chemin originel

Dans ce scénario, nous nous plaçons dans un contexte de libération d'un élément de calcul. Pour cela nous migrons l'acteur $gpu-normalize$ de son élément de calcul originel, le GPU-1 du nœud B vers le GPU-0 du nœud B comme illustré dans figure 5.9. Le choix de cette libération peut être motivé par une politique de tolérance aux fautes. En effet, si un élément de calcul du cluster produit des résultats erronées, il est possible via notre approche de migration de tâches de replacer ses calculs sur un autre élément sans pour autant arrêter le traitement de l'application. Les chemins de migration dans cette expérience sont de tailles égales. En effet, un jeton de données nécessite 7 itérations pour traverser les deux chemins (originel et migration). De la même façon que pour l'expérience précédente, nous mesurons l'impact de la migration sur le flux de données de sortie lors de la migration.

Les résultats de cette expérience sont représentés dans la figure 5.10. Comme dans la figure 5.8, nous présentons deux courbes, une courbe bleu représentant le débit de données de sortie

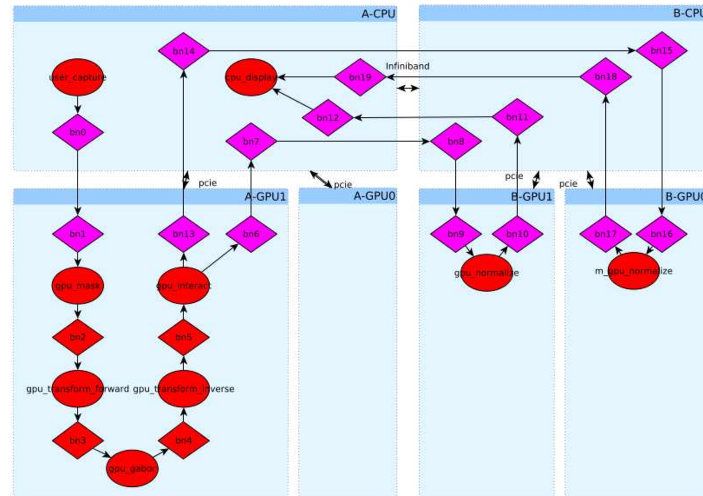

FIGURE 5.9 – Illustration de la mise à jour du graphe d'implémentation de l'application de saillance dans un contexte de migration avec chemins égaux

et une courbe verte représentant les temps d'exécution de chaque itération. Des étoiles sur cette dernière indiquent les itérations au cours desquelles une image de sortie a été produite. A l'itération 26, le mécanisme de tolérance aux fautes détecte une anomalie sur les résultats produits par l'élément de calcul GPU-1. La migration de l'acteur $gpu - normalize$ est alors déclenchée et s'étale sur les 6 itérations suivantes. Nous analysons dans ce qui suit les résultats de cette expérience en distinguant 3 étapes :

Étape d'avant la migration ($it < 26$). Le runtime exécute le graphe d'implémentation originel incluant une distribution sur l'élément de calcul présentant une anomalie (GPU-1 du nœud B). Le temps d'exécution des itérations est relativement constant d'environ 40 ms à l'exception de la première itération présentant un surcoût du à l'initialisation du runtime. Les images de sorties sont produites à hauteur de 1 image par itération ce qui aboutit à un débit d'environ 25 FPS.

Étape de migration ($25 < it < 33$). A la fin de l'itération 26 le runtime entame le processus de mise à jour du graphe d'implémentation. Il crée le nouveau nœud de destination, alloue les buffers et initialise les chemins originel et de migration pour être respectivement vidé et rempli. Cette mise à jour engendre un temps de traitement d'environ 20 ms qui augmente le temps de l'itération à 60 ms. Durant les itérations (27,28,..32) le chemin originel est vidé alors que le chemin de migration est rempli. Il est à remarquer que le flux de données de sortie n'est pas arrêté : le débit reste lié seulement au temps de traitement et est d'environ 25 FPS. Il est à noter que dans le contexte de défaillance de l'élément de calcul originel, les images produites durant ces itérations présentent aussi des anomalies.

Après la migration ($it > 32$), le runtime exécute le graphe d'implémentation mis à jour n'associant pas d'acteur à l'élément de calcul défaillant. Les données de sortie sans anomalie sont produites à chaque itération et le débit est fonction seulement du temps de traitement d'une itération. Dans cette expérience, nous avons validé notre approche de migration de tâche

dans un contexte de tolérance aux fautes. Notre approche permet de libérer un élément de calcul sans arrêter le traitement de l'application.

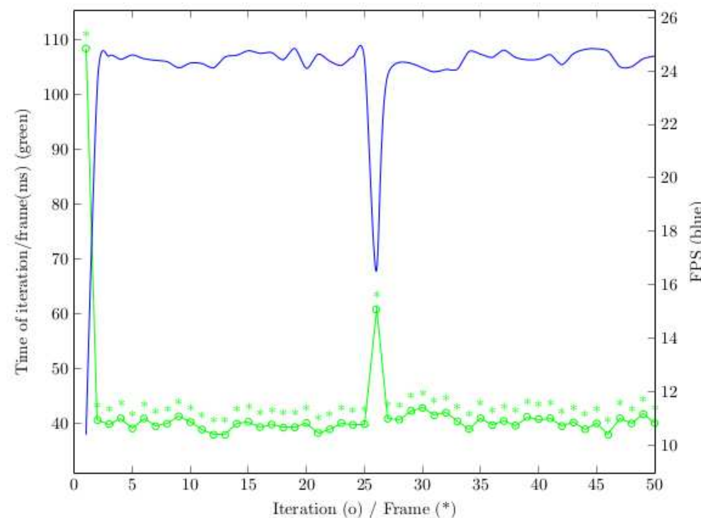

FIGURE 5.10 – Illustration de la stabilisation du débit de production des images de sortie dans un scénario de migration pour la libération d'un élément de calcul

5.4.3 Scénario pour réduire la consommation d'énergie avec chemin de migration plus court que le chemin originel

Dans ce troisième scénario, nous nous fixons un objectif de réduction de la consommation d'énergie du cluster traitant l'application de saillance, avec une distribution initiale sur deux nœuds de calcul. Le but est de minimiser le nombre de nœuds de calcul réduisant ainsi les chemins de communication et les éléments de calcul sollicités. Dans cette optique nous migrons l'acteur $gpu - normalize$ de l'élément de calcul GPU-1 du nœud B vers l'élément GPU-0 du nœud A comme présenté sur la figure 5.11. Cette migration permet de libérer le nœud B qui peut être éteint ou mis en veille. En outre, cette migration réduit les communications à travers le réseau ce qui réduit de-facto la consommation d'énergie des interfaces réseaux et des routeurs. Nous nous intéressons également, comme pour les expériences précédentes, à quantifier l'impact de cette migration sur le flux de données de sortie.

La figure 5.12 représente les résultats obtenus pour ce scénario. Nous les présentons de la même façon que sur les figures 5.8 et 5.10. La courbe bleue represente le débit. La courbe verte représente le temps des itérations. Les étoiles vertes indiquent sur quelles itérations les images de sorties ont été produites. La migration est déclenchée à l'itération 26. Dans ce qui suit nous distinguons 3 étapes dans le calcul :

Avant la migration ($it < 26$) : le runtime exécute le graphe d'implémentation originel de façon quasi régulière. Les temps de traitement des itérations sont d'environ 20 ms. Les itérations produisent toutes des données de sortie ce qui fait que le débit reste relativement constant à environ 45 FPS.

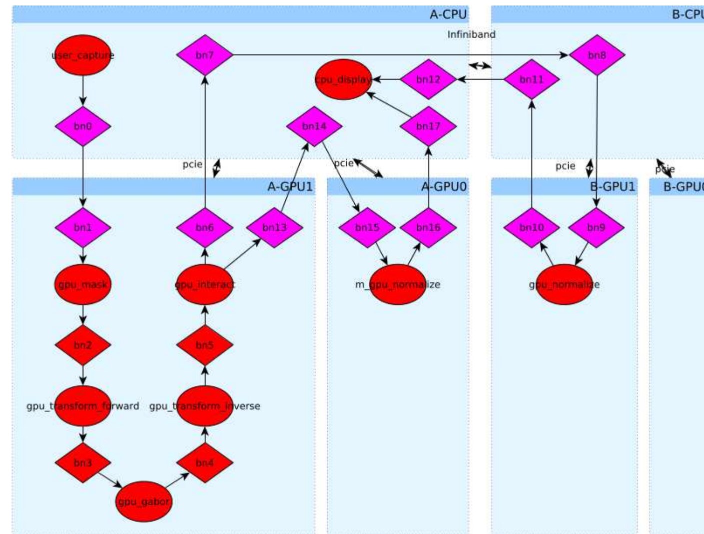

FIGURE 5.11 – Illustration de la mise à jour du graphe d'implémentation de l'application de saillance dans un contexte de migration avec chemin moins long

Lors de la migration (25 <it <33) : à l'itération 26, le processus de migration effectue d'abord la mise à jour du graphe d'implémentation (IG) et notamment du chemin en amont. Cela prolonge la durée de l'itération à 30 ms. Aux itérations 27 et 28, le runtime n'exécute pas les acteurs du chemin en amont et se contente de vider le chemin originel. Cela réduit le temps de l'itération 27 à 18 ms. Dans le reste des itérations (28,29,30,31,32), le runtime remplit le chemin de migration et vide en même temps le chemin originel, ce qui stabilise progressivement la courbe verte à 20 ms. Dans cette partie, le flux de sortie des données n'est pas arrêté au cours de ces itérations. Le débit de sortie ne dépend donc que du temps des itérations.

Après la migration ($it > 32$) : dans cette étape, le support d'exécution exécute une itération avec temps constant environs égal 20 ms (courbe verte). Aussi, toutes les itérations produisent une image comme indiqué sur la figure (étoiles vertes). Par conséquent, le débit de l'écoulement de sortie est constant à 45 FPS. Dans cette expérience nous avons montré l'utilisation de la migration de tâche pour réduire les chemins de communication et les nœuds de calcul utilisés dans le but, par exemple, de réduire la consommation d'énergie.

5.5 Conclusion

Dans ce chapitre, nous avons présenté une approche de migration de tâches en ligne pour un support exécutif BSP itératif. La stratégie de migration que nous avons adoptée porte sur une altération minimale du flux de données de sortie. En effet, cette contrainte est importante dans plusieurs contextes d'exécution d'applications de traitement d'image et de signal, domaine applicatif adressé par l'environnement PACCO. La stratégie consiste à distribuer les actions de la migration sur plusieurs itérations. Ainsi, il est possible de replacer l'exécution des tâches

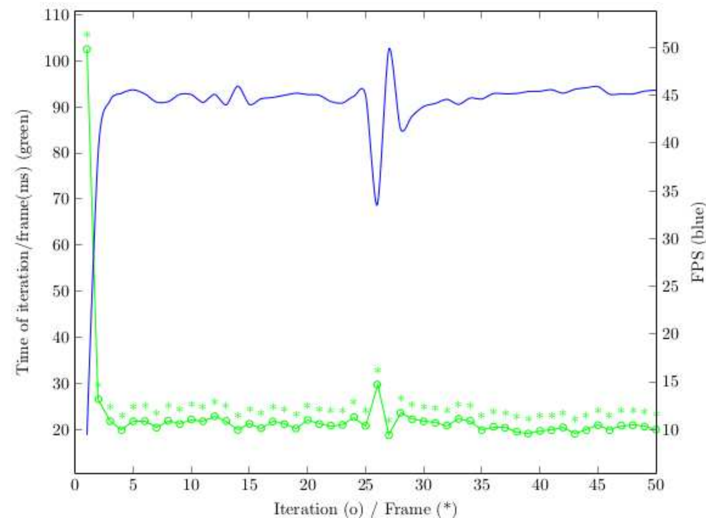

FIGURE 5.12 – Illustration de la préservation du débit de production des images de sortie dans un scénario de migration pour la réduction de la consommation d'énergie

sur différents éléments de calcul sans arrêter le calcul pour autant. Nous avons implémenté notre approche dans l'environnement d'exécution PACCO où nous avons distingué 3 cas de figures selon la différence de longueur des chemins de données originel et de migration. En effet, il est nécessaire, dans le cas ou ils sont de taille différente, de synchroniser les chemins en amont ou en aval. Nous avons effectué 3 expériences de validation sur une application du monde réel, l'application de saillance. A travers ces expériences nous avons montré l'intérêt de notre approche de migration dans 3 scénarios : un scénario de replacement pour augmenter les performances temporelles, un scénario de libération d'un élément de calcul dans un contexte de tolérance aux fautes et un scénario de réduction de la consommation d'énergie en réduisant les chemins de communication et le nombre de nœuds utilisés. En outre, les résultats présentés pour ces 3 scénarios illustrent les impacts des 3 cas de migration sur le débit de données en sortie.

Troisième partie

SignalPU

Comme discuté dans la sous-section 3.5.6 du chapitre 3, de nombreux modèles de programmation pour l'implémentation des applications DSP sont basés sur le modèle de calcul graphe de flot de données. Plusieurs environnements d'implémentation de ces applications ont été cités dans la section 3.4 du chapitre 3. Ils offrent un bon niveau d'abstraction en masquant pour certains la décomposition en parties parallèles, les communications, et les synchronisations. Cependant, ces outils sont pour la plupart des outils à placement statique et éventuellement manuel présentant les inconvénients suivants :

1. Non adéquat aux architectures hétérogènes : les charges des calculs varient selon l'élement de calcul de destination.
2. Non adapté pour des algorithmes conditionnels et dynamiques : les charges des calculs varient selon les branchements.
3. Non adapté pour les applications dépendantes des données "data-dependant" : les charges des calculs varient selon les données d'entrée.

En outre, si il est manuel :

1. L'utilisateur doit s'intéresser aux spécificités de l'architecture de calcul afin de réaliser un placement et un ordonnancement efficace.
2. Réduit de ce fait la portabilité du code.

Dans la partie précédente, nous avons proposé une solution de migration de tâches pour enrichir l'environnement PACCO, lui permettant entre autres de modifier à l'exécution le placement des acteurs. Néanmoins, plusieurs inconvénients discutés dans le chapitre 4 subsistent. En effet, d'une part, du point de vue abstraction et productivité, l'outil PACCO reste limité et nécessite de l'utilisateur de connaître l'architecture de calcul, de la modéliser et de profiler les calculs. D'autre part, du point de vue des performances de son modèle d'exécution, l'environnement est synchronisé à l'aide de barrière globale ce qui retarde les calculs des itérations successives.

Pour mieux répondre à la problématique d'implémentation d'applications de type DSP sur cluster hétérogène, nous proposons dans cette partie un modèle de programmation à haut niveau d'abstraction. Il est basé sur le modèle de calcul graphe de flot de données et le support exécutif dynamique StarPU que nous enrichissons avec plusieurs fonctionnalités adaptées aux applications de type DSP. Cette combinaison permet d'offrir une couche d'abstraction supplémentaire par rapport à d'autres environnements basés sur les DFG en abstrayant le placement par une distribution dynamique des tâches. Dans le chapitre 6, nous présentons SignalPU et décrivons sa conception et son implémentation. Ensuite, pour valider notre approche, nous présentons en premier dans le chapitre 7 une bibliothèque d'opérateur de charge permettant de simuler des applications synthétiques. Ensuite, nous présentons quelques résultats et comparaisons obtenus sur l'exécution d'une application construite avec cette bibliothèque.

Chapitre 6

SignalPU : Un modèle de programmation DFG basé sur StarPU

6.1 Introduction

Comme discuté auparavant dans le chapitre 2, pour être efficace, un modèle de programmation parallèle doit offrir plusieurs niveaux d'abstractions. Il doit réduire l'effort de programmation pour exprimer : la décomposition du problème en parties parallèles, les communications, les synchronisations, mais aussi le placement et l'ordonnancement des calculs sur l'architecture. En effet, il est intéressant pour l'utilisateur de ne pas avoir à tenir compte des caractéristiques de l'architecture dans son programme. Cela augmente fortement sa productivité et engendre une grande portabilité du code. Plusieurs environnements proposent cette abstraction mais présentent des inconvénients pour l'implémentation des applications DSP comme discuté dans la section 3.5 du chapitre 3. Dans le présent chapitre, nous proposons un modèle de programmation appelé "SignalPU", basé sur le modèle de calcul DFG et le support exécutif StarPU. Nous présentons dans la section 6.2 l'environnement StarPU. Nous décrivons ses fonctionnalités et montrons à travers un exemple comment l'utilisateur peut implémenter un algorithme DSP. Ensuite, dans la section 6.3, nous présentons notre modèle de programmation et décrivons sa conception et ses fonctionnalités. Nous terminons ce chapitre en donnant dans la section 6.4 les avantages au travers de nos contributions dans cet environnement.

6.2 L'environnement StarPU

StarPU [ATN10] est un environnement unifié pour le calcul généraliste haute performance sur architectures parallèles hétérogènes. C'est un modèle de programmation à base de tâches dépendantes offrant un haut niveau d'abstraction. Proposé par l'equipe STORM de l'Inria de Bordeaux. Pour décrire son application, l'utilisateur décompose son algorithme sous forme de DAG de tâches asynchrones. Les communications, les synchronisations et le placement et l'ordonnancement des tâches sont implicites et effectués par le moteur d'exécution de StarPU. Celui ci est basé sur un ordonnanceur offrant plusieurs politiques de placement dynamiques pour distribuer les tâches sur les éléments de calcul hétérogènes et un système de gestion de la mémoire permettant de déplacer les données nécessaires sur des architectures à mémoires partagées et distribuées. Dans la figure 6.1, nous présentons une illustration des principales

composantes de StarPU. Dans ce qui suit nous décrivons quelques fonctionnalités caractérisant ces composants :

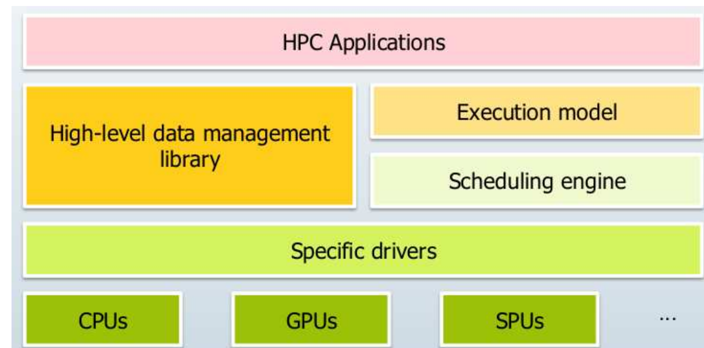

FIGURE 6.1 – Illustration des principales composantes de StarPU

La portabilité La portabilité dans StarPU est obtenue par une abstraction de la machine à base d'un modèle de tâches appelé "Codlets". Elles représentent les fonctionnalités de l'application encapsulées dans des structures accueillant plusieurs fonctions pouvant être exécutées sur des éléments de calcul hétérogènes, par exemple, une fonction CPU et une fonction GPU. Le Runtime StarPU est capable de choisir à l'exécution les fonctions les plus performantes pour les éléments de calcul disponibles.

Le transfert des données Pour abstraire les tâches de transferts de données, le modèle StarPU propose un gestionnaire des données haut niveau. Il permet de garantir la cohérence des données en rendant disponible dans chaque élément de calcul celles nécessaires pour exécuter une "codlet". Les données sont conservées sur l'élément si elles sont nécessaires pour d'autres tâches, sinon elles sont libérées. En outre, le gestionnaire de données permet de précharger les données avant que les tâches ne soient exécutées, ainsi un recouvrement communications-calculs est effectué pour augmenter les performances.

Les dépendances Les dépendances de tâches dans StarPU sont gérées de plusieurs façons : implicitement, par le moteur d'exécution en respectant les dépendances de données en lecture et en écriture entre les tâches. Par exemple, deux tâches sont dépendantes l'une de l'autre, si l'une écrit une donnée et l'autre lit cette même donnée. Autrement, elle peuvent être exprimées explicitement, en utilisant des fonctions prédéfinies pour lier deux tâches, ou en utilisant les "Tag" pour opérer un rendez-vous entre plusieurs tâches.

L'ordonnancement hétérogène StarPU permet une portabilité des performances en distribuant les calculs sur toutes les ressources de l'architecture, y compris si elle est hétérogène. En effet, en se basant sur des politiques d'ordonnancement dynamiques comme le vol de tâches "work stealing" ou la politique de HEFT, StarPU est capable d'optimiser les performances en exécutant chaque tâche sur le meilleur élément de calcul. En outre, l'ordonnancement de

tâches cherche à répartir les charges tout en préservant la localité des données afin de réduire les communications.

La distribution de calculs MPI Afin de distribuer le calcul sur plusieurs nœuds du cluster, l'environnement StarPU offre une interface supplémentaire intégrant la librairie MPI pour permettre des communications à travers le réseau. Ainsi, l'utilisateur doit seulement indiquer la distribution des données sur les nœuds MPI du cluster pour permettre au moteur d'execution de distinguer les tâches à exécuter sur chaque nœuds et pour effectuer les communications MPI nécessaires. Par conséquent, en tenant compte des disponibilités des données, le DAG de tâches est subdivisé en sous graphes, chacun exécuté sur un nœud MPI.

StarPU offre d'autres fonctionnalités, notamment une extension du langage C sous forme de directives PRAGMA permettant de paralléliser les calculs en annotant le code séquentiel. Il inclut aussi un simulateur interagissant avec SimGrid permettant de simuler des exécutions d'applications sur différentes plate-formes. En outre, il dispose d'un outil de mesure des performances "profiler" et d'un analyseur de graphe de tâches.

6.2.1 Exemple d'une implémentation DSP avec StarPU

Afin d'illustrer l'utilisabilité de StarPU pour le domaine applicatif visé, nous présentons dans cette sous-section un exemple d'implémentation d'une application avec son API. L'application est décrite dans le pseudo algorithme 3. Il s'agit d'un exemple synthétique simple d'une application de type DSP itérative. A chaque itération, une unité de donnée est produite par la fonction $Producer$, elle est ensuite traitée par plusieurs fonctions ($Kernel_1, Kernel_2, Kernel_3, Kernel_4, Kernel_5$) et finalement consommée en sortie par la fonction $Consumer$. L'application contient une dépendance inter-itération : $Kernel5$ consomme la variable $Var4$ de l'itération courante et Var'_4 contenant la valeur de $Var4$ de l'itération précédente. Le listing 6.1 présente l'implémentation de cette application avec StarPU.

Algorithm 3 Exemple d'une application de type DSP

Input : Input data set ($DataSet_{in}$).
Output : Output data set ($DataSet_{out}$).
1: **for each** $Data_{unit}$ in $DataSet_{in}$ **do**
2: $Var_1 \leftarrow Producer(Data_{unit})$ {Read each data unit from the input data set}
3: $Var_2 \leftarrow Kernel_1(Var_1)$
4: $Var_{3_1} \leftarrow Kernel_2(Var_2)$
5: $Var_{3_2} \leftarrow Kernel_3(Var_2)$
6: $Var_4 \leftarrow Kernel_4(Var_{3_1}, Var_{3_2})$
7: $Var_5 \leftarrow Kernel_5(Var_4, Var'_4)$
8: $Var'_4 \leftarrow Var_4$
9: $DataSet_{out} \leftarrow Consumer(Var_5)$ {Save each data unit in the output data set}
10: **end for**

La première étape à effectuer consiste à représenter les fonctions de l'application avec des Codlets. Pour cela il faut les déclarer dans le code en utilisant les structures adéquates de la librairie comme montré dans les lignes 1,10,24 du code 6.1. Il faut initialiser les entées de chaque Codlet : il faut définir la fonction à exécuter, la nature de l'élément de calcul visé (CPU, GPU ..), son nombre de buffers et leur type de dépendance de données (en écriture, en lecture, ...).

La deuxième étape consiste à créer les buffers d'entrées et de sorties pour chaque acteur et d'allouer l'espace mémoire nécessaire pour chacun d'eux. Pour cela, il faut utiliser les primitives de la librairie dédiées à la gestion des données. Cette étape est montrée dans les lignes 95,96,97..104. A noter qu'il faut déclarer un seul buffer s'il est utilisé par plusieurs tâches. Ainsi, les dépendances de données entre les tâches sont détectées ce qui permettra de synchroniser le traitement.

La troisième étape consiste à créer les tâches et à les configurer au sein de chaque itération. En effet, l'application considérée est itérative, elle nécessite une tâche pour traiter chaque unité de donnée avec un acteur donné. La création des tâches est illustrée dans les lignes 105,110,..149. Pour la configuration, il est nécessaire d'identifier pour chaque tâche la codlet lui correspondant, les arguments en entrée des fonctions de cette codlet, le type de tâche (Synchrone : la tâche est exécutée une fois la tâche précédente terminée, Asynchrone : l'execution de la tâche n'est pas retardée par la tâche précédente sauf s'il y a dépendance de données entre elles).

La quatrième étape consiste à soumettre les tâches de manière asynchrones et de les attendre avant d'entamer une autre itération. Pour cela il faut utiliser des fonctions prédéfinies comme montré dans les lignes 154, 155, ..160, 162.

Listing 6.1 – Exemple d'un programme StarPU

```
1  static struct starpu_codelet cl_producer=
2  {
3          .modes = { STARPU_W },
4          .cpu_funcs = {producer_cpu_func, NULL},
5          .cpu_funcs_name = {"kernel1_cpu_func", NULL},
6          .nbuffers = 1,
7          .model = &vector_scal_model
8  }
9
10 static struct starpu_codelet cl_kernel1=
11 {
12         .modes = { STARPU_R, STARPU_W , STARPU_W },
13         /* CPU implementation of the codelet */
14         .cpu_funcs = {kernel1_cpu_func, NULL},
15         .cpu_funcs_name = {"kernel1_cpu_func", NULL},
16 #ifdef STARPU_USE_CUDA
17         /* CUDA implementation of the codelet */
18         .cuda_funcs = {kernel1_cuda_func, NULL},
19         .cuda_flags = {STARPU_CUDA_ASYNC},
20 #endif
21         .nbuffers = 3,
22 }
23
24 static struct starpu_codelet cl_kernel2=
25 {
26         .modes = { STARPU_R ,STARPU_W, },
27         /* CPU implementation of the codelet */
28         .cpu_funcs = {kernel2_cpu_func, NULL},
29         .cpu_funcs_name = {"kernel1_cpu_func", NULL},
30 #ifdef STARPU_USE_CUDA
31         /* CUDA implementation of the codelet */
```

```
32          .cuda_funcs = {kernel2_cuda_func, NULL},
33          .cuda_flags = {STARPU_CUDA_ASYNC},
34  #endif
35          .nbuffers = 2,
36  }
37
38  static struct starpu_codelet cl_kernel3=
39  {
40          .modes = { STARPU_R, STARPU_W },
41          /* CPU implementation of the codelet */
42          .cpu_funcs = {kernel3_cpu_func, NULL},
43          .cpu_funcs_name = {"kernel1_cpu_func", NULL},
44  #ifdef STARPU_USE_CUDA
45          /* CUDA implementation of the codelet */
46          .cuda_funcs = {kernel3_cuda_func, NULL},
47          .cuda_flags = {STARPU_CUDA_ASYNC},
48  #endif
49          .nbuffers = 2,
50  }
51
52  static struct starpu_codelet cl_kernel4=
53  {
54          .modes = { STARPU_R, STARPU_R, STARPU_W },
55          /* CPU implementation of the codelet */
56          .cpu_funcs = {kernel4_cpu_func, NULL},
57          .cpu_funcs_name = {"kernel1_cpu_func", NULL},
58  #ifdef STARPU_USE_CUDA
59          /* CUDA implementation of the codelet */
60          .cuda_funcs = {kernel4_cuda_func, NULL},
61          .cuda_flags = {STARPU_CUDA_ASYNC},
62  #endif
63          .nbuffers = 3,
64  }
65
66  static struct starpu_codelet cl_kernel5=
67  {
68          .modes = { STARPU_R, STARPU_W, STARPU_RW },
69          /* CPU implementation of the codelet */
70          .cpu_funcs = {kernel5_cpu_func, NULL},
71          .cpu_funcs_name = {"kernel1_cpu_func", NULL},
72  #ifdef STARPU_USE_CUDA
73          /* CUDA implementation of the codelet */
74          .cuda_funcs = {kernel5_cuda_func, NULL},
75          .cuda_flags = {STARPU_CUDA_ASYNC},
76  #endif
77          .nbuffers = 3,
78  }
79
80  static struct starpu_codelet cl_consumer=
81  {
82          .modes = { STARPU_R },
83          .cpu_funcs = {consumer_cpu_func, NULL},
84          .cpu_funcs_name = {"kernel1_cpu_func", NULL},
85          .nbuffers = 1,
86  }
87
88
89  int main(int argc, char **argv)
90  {
91          int ret = starpu_init(NULL);
92          if (ret == -ENODEV) goto enodev;
93          /* We consider a vector of float that is initialized just as any of C
94           * data */
95          float Data_unit[NX];
96          starpu_data_handle_t Var1_handle,Var2_handle,Var3_1_handle,Var3_2_handle,Var4_handle,
                Var5_handle,Var5_autodep_handle;
97
98          starpu_vector_data_register(&Var1_handle, STARPU_MAIN_RAM, (uintptr_t)Data_unit, NX,
                sizeof(vector[0]));
99          starpu_vector_data_register(&Var2_handle, STARPU_MAIN_RAM, (uintptr_t)Data_unit, NX,
                sizeof(vector[0]));
100         starpu_vector_data_register(&Var3_1_handle, STARPU_MAIN_RAM, (uintptr_t)Data_unit, NX,
                sizeof(vector[0]));
101         starpu_vector_data_register(&Var3_2_handle, STARPU_MAIN_RAM, (uintptr_t)Data_unit, NX,
                sizeof(vector[0]));
102         starpu_vector_data_register(&Var4_handle, STARPU_MAIN_RAM, (uintptr_t)Data_unit, NX,
                sizeof(vector[0]));
103         starpu_vector_data_register(&Var5_handle, STARPU_MAIN_RAM, (uintptr_t)Data_unit, NX,
                sizeof(vector[0]));
104         starpu_vector_data_register(&Var5_autodep_handle, STARPU_MAIN_RAM, (uintptr_t)Data_unit,
                 NX, sizeof(vector[0]));
105
106
```

```
107         for ( ; ; )
108         {
109
110     struct starpu_task *task_producer = starpu_task_create();
111         task->synchronous = 1;
112         task->cl = &cl_producer;
113     task->handles[0] = Var1_handle;
114
115         struct starpu_task *task_kernel1 = starpu_task_create();
116         task_kernel1->synchronous = 1;
117         task_kernel1->cl = &cl_kernel1;
118     task_kernel1->cl_arg = &factor;
119         task_kernel1->cl_arg_size = sizeof(factor);
120     task_kernel1->handles[0] = task_producer->handles[0];
121     task_kernel1->handles[1] = Var2_handle;
122
123     struct starpu_task *task_kernel2 = starpu_task_create();
124         task_kernel2->synchronous = 1;
125         task_kernel2->cl = &cl_kernel2;
126     task_kernel2->handles[0] = task_kernel1->handles[1];
127     task_kernel2->handles[1] = Var3_1_handle;
128
129     struct starpu_task *task_kernel3 = starpu_task_create();
130         task_kernel3->synchronous = 1;
131         task_kernel3->cl = &cl_kernel3;
132     task_kernel3->handles[0] = task_kernel1->handles[1];
133     task_kernel3->handles[1] = Var3_2_handle;
134
135     struct starpu_task *task_kernel4 = starpu_task_create();
136         task_kernel4->synchronous = 1;
137         task_kernel4->cl = &cl_kernel4;
138     task_kernel4->handles[0] = task_kernel2->handles[1];
139     task_kernel4->handles[1] = task_kernel3->handles[1];
140     task_kernel4->handles[1] = Var4_handle;
141
142     struct starpu_task *task_kernel5 = starpu_task_create();
143         task_kernel5->synchronous = 1;
144         task_kernel5->cl = &cl_kernel5;
145     task_kernel5->handles[0] = task_kernel4->handles[2];
146     task_kernel5->handles[1] = Var4_handle;
147     task_kernel5->handles[1] = Var5_autodep_handle;
148
149     struct starpu_task *task_consumer = starpu_task_create();
150         task_kernel5->synchronous = 1;
151         task_kernel5->cl = &cl_consumer;
152     task_kernel5->handles[0] = task_kernel5->handles[1];
153
154     starpu_task_submit(task_producer);
155     starpu_task_submit(task_kernel1);
156     starpu_task_submit(task_kernel2);
157     starpu_task_submit(task_kernel3);
158     starpu_task_submit(task_kernel4);
159     starpu_task_submit(task_kernel5);
160     starpu_task_submit(task_consumer);
161
162     starpu_task_wait_for_all();
163     }
164
165   starpu_data_unregister(vector_handle);
166   starpu_shutdown();
167
168   return(1);
169
170 enodev:
171         return 77;
172 }
```

Une fois les tâches soumises, le support exécutif (runtime) les distribue dynamiquement sur le cluster suivant la politique choisie par l'utilisateur lors de l'exécution. Cette implémentation permet d'exploiter du parallélisme de tâches entre les acteurs $kernel_2$ et $kernel_3$. Elle permet aussi d'exécuter les fonctions de chaque acteur sur l'élément de calcul le plus adéquat. En outre, elle permet d'exploiter le parallélisme de données sur des accélérateurs GPU.

6.3 SignalPU

Dans la présente section, nous décrivons notre modèle de programmation appelé "SignalPU". Basé sur la combinaison du modèle de calcul DFG et l'environnement StarPU, il permet d'implémenter facilement et efficacement des applications de type DSP sur des architectures parallèles et hétérogènes. Avec SignalPU, l'utilisateur n'a pas besoin d'exprimer explicitement les spécificités de l'implémentation comme la décomposition de l'algorithme sous forme de "Codlet" et tâches, la gestion des allocations, les communications et les synchronisations, etc. Avec son interface DFG, l'utilisateur peut exploiter les différents niveaux de parallélisme dans son implémentation : parallélisme de données, parallélisme de tâches, parallélisme de graphe. En outre, il couvre des architectures à mémoires partagées et distribuées, et permet de cibler des éléments de calcul hétérogènes : CPU, GPU, Cell, et Xeon Phi.

Dans la figure 6.2, nous montrons le positionnement de notre MdPP en comparaison à plusieurs environnements cités dans le chapitre 2. En effet, SignalPU est conçu pour implémenter une catégorie d'applications : les applications DSP ou celles pouvant être modélisées avec un DFG. Cela lui permet d'offrir une abstraction supplémentaire dans la décomposition du problème. L'utilisateur exprime son algorithme plus naturellement à l'aide d'un graphe orienté. Aussi, il n'a pas besoin de s'adapter à une syntaxe particulière tant pour allouer les ressources mémoire que pour gérer les communications et les synchronisations. De ce fait, comme illustré dans la figure 6.2, il présente d'une part un niveau d'abstraction supérieur à des environnements à base de directives comme OpenMP et OpenACC et à des environnements à base de tâches comme StarPU, TBB et X-KAAPI. D'autre part, en ce qui concerne les environnements spécifiques à l'implémentation d'applications DSP comme PREESM, PACCO et DAL. SignalPU n'offre pas forcément d'abstraction supplémentaire sur la décomposition en parties parallèles, les communications ou les synchronisations, puisque ces environnements sont également basés sur le MdC DFG. Cependant, grâce à son modèle d'exécution dynamique (StarPU), il présente une meilleure abstraction au niveau de la distribution des calculs sur l'architecture. En effet, l'utilisateur n'a pas besoin de connaître les spécificités de l'architecture ni de placer ou ordonnancer manuellement les acteurs. En outre, la distribution est dynamique et peut s'adapter à différentes architectures ce qui permet une forte portabilité. Elle est également mieux adaptée à l'exécution d'applications dynamiques (ayant les caractéristiques énoncées dans les points 2 et 3 de l'introduction de la partie III) sur des architectures hétérogènes.

6.3.1 Conception générale

La figure 6.3 présente la structure en 3 niveaux de notre MdPP SignalPU. Premièrement, l'utilisateur exprime son application sous forme de DFG à l'aide de l'interface DFG-XML. Grâce à ce module, il bénéficie d'un haut niveau d'abstraction : il ne manipule pas la librairie de StarPU pour créer les dépendances de données entre tâches, ou pour spécifier les fonctions et les éléments de calcul de destination. Ce premier niveau représente le point d'entrée de notre modèle de programmation.

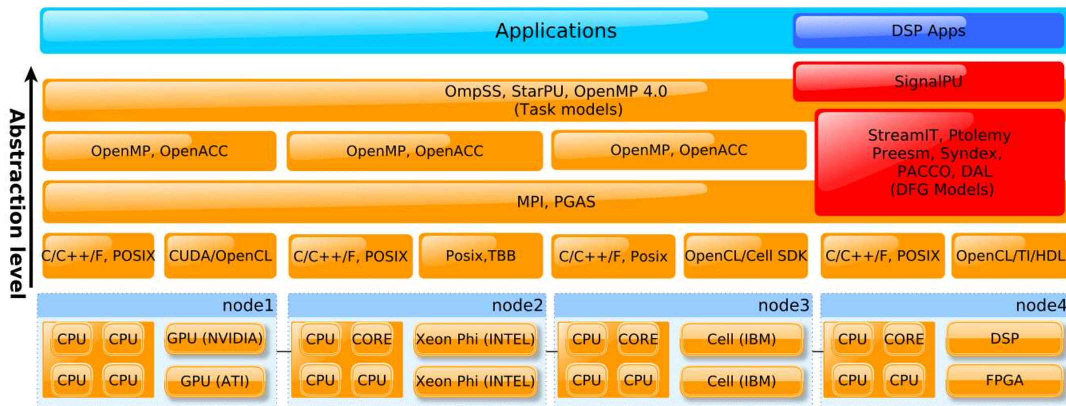

Figure 6.2 – Positionnement de SignalPU en terme de niveau d'abstraction et de couverture d'architecture

Le second niveau se charge de générer l'implémentation proprement dite. En effet, à l'exécution, ce module analyse et traite les entrées du modèle, à savoir le DFG-XML représentant l'algorithme, les fonctions (kernels) etc, et construit le DAG de tâches et l'implémentation de l'application. Pour cela, il se base sur plusieurs fonctionnalités spécifiques aux applications de type DSP : le déroulement de graphe, le pipelining de tâches, la re-utilisation des buffers et la décomposition sur nœuds MPI. Ce module permet d'une part d'exprimer plusieurs niveaux de parallélisme comme le parallélisme de tâches, de données et de graphes. D'autre part, il limite le surcoût du à la gestion des tâches et de la mémoire. Également, dans ce module, l'utilisateur ne manipule pas l'API de StarPU et bénéficie d'une construction automatique des Codlets et du DAG de tâches, d'un déroulement implicite de la boucle principale de l'application, de la décomposition implicite du calcul à travers les nœuds MPI et d'une soumission de tâches continue.

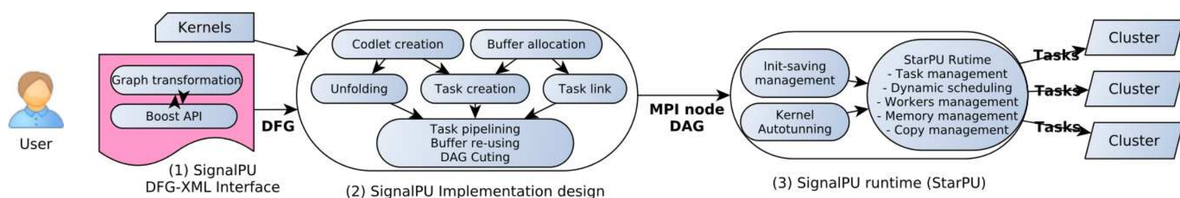

Figure 6.3 – Conception et composants de SignalPU

Finalement, dans le troisième niveau de SignalPU, le moteur d'exécution de StarPU est invoqué dans chaque nœud du cluster pour gérer les éléments de calcul et le DAG de tâches soumis et construit dynamiquement. Il est également en charge de distribuer dynamiquement les calculs sur l'architecture en répartissant les charges et en réduisant les communications, le but étant d'optimiser les performances globales du cluster. De plus, dans ce module, nous rajoutons également deux fonctionnalités spécifiques aux applications DSP afin d'optimiser les performances de chaque tâche : la fonctionnalité de préservation des initialisations et l'auto-

tuning des tâches GPUs.

6.3.2 Composants et fonctionnalités du modèle : cas d'étude sur l'exemple de l'algorithme 3

Dans ce qui suit, nous présentons les trois niveaux de l'environnement SignalPU à travers l'implémentation de l'exemple de l'algorithme 3. En outre, nous décrivons en détail les fonctionnalités de chaque module.

6.3.2.1 Niveau 1 : Modèle de calcul et interface DFG-XML

Dans ce niveau, nous proposons une interface basée sur le modèle de calcul DFG et une description XML de l'application. L'utilisateur doit décrire son algorithme sous forme de graphe de flot de données. Dans la suite nous présentons le format DFG-XML retenu et l'illustrons au travers de la description du DFG de la figure 6.4 représentant l'application décrite par l'algorithme 3. Premièrement, l'utilisateur doit modéliser les fonctions de l'algorithme sous forme d'acteurs. Chaque acteur est représenté par un nœud "Node" dans le graphe à l'aide d'une structure XML comportant plusieurs attributs comme illustré dans le listing 6.2 lignes 4-16. La structure comporte entre autre : le nom de la fonction permettant d'appeler la portion du code correspondant à son traitement, ses arguments d'entrée et de sortie et le type de d'élément de calcul ciblé (CPU, GPU, ...).

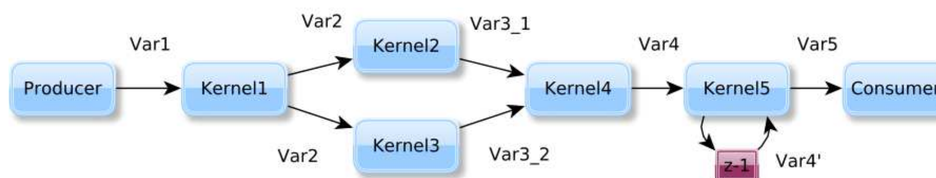

FIGURE 6.4 – DFG représentant l'algorithme 3

En second, l'utilisateur doit décrire les flots de données entre les acteurs. Pour cela, il représente chaque connexion entre deux fonctions de l'algorithme par une arrête "Edge" en utilisant une structure XML comme illustré dans le code 6.2 lignes 21-27. Cette structure comporte des attributs sur les ports d'entrée et de sortie des nœuds connectés, mais également pour définir le type et la taille des données transitant dans ce flot. Suite à cela, l'utilisateur doit introduire dans son code les fonctions référencées sous forme de 2 sous fonctions : une sous fonction d'initialisation appelée "Init" comportant le code d'initialisation des variables et des buffers internes et une sous fonction d'exécution "Exec" qui se charge de traiter les données à chaque itération. Il serait possible de réaliser une interface graphique pour générer cette description XML. Nous relevons également que cette description est compatible avec le point d'entrée de l'environnement PACCO décrit dans le chapitre 4.

Listing 6.2 – Exemple d'une description XML de l'example de l'algorithme 3

```xml
<!-- ****************************************
                NODES
****************************************-->
<node id="n0">
  <data key="k_node_id">n0</data>
  <data key="k_node_name">producer</data>
  <data key="k_kernel">producer(out0)</data>
  <data key="k_kind">CPU/GPU</data>
</node>

<node id="n1">
  <data key="k_node_id">n1</data>
  <data key="k_node_name">kernel1</data>
  <data key="k_kernel">kernel1(in0,out0)</data>
  <data key="k_kind">CPU/GPU</data>
</node>

<!-- ************************************
                EDGES
************************************ -->
<edge source="n0" target="n1">
  <data key="k_edge_name">n0_to_n1</data>
  <data key="k_port_from">out0</data>
  <data key="k_port_to">in0</data>
  <data key="k_type">mat</data>
  <data key="k_size">262144</data>
</edge>
```

6.3.2.2 Niveau 2 : Le modèle d'implémentation

Dans ce niveau le DFG-XML est analysé à l'aide de la bibliothèque Boost Graphml Reader [Pol13] pour identifier les acteurs et les flots de données. L'API C de StarPU [ATN10] est alors invoquée pour implémenter l'algorithme en exploitant plusieurs niveaux de parallélisme : le parallélisme de tâches, de données et de graphe. Comme illustré dans la figure 6.5 représentant l'implémentation de l'algorithme 3, le DFG-XML est transformé sous forme de DAG de tâches en utilisant les fonctionnalités de : dépliement de graphe, re-utilisation de buffers, pipelining de tâche, et finalement de distribution du calcul sur plusieurs nœuds MPI. Dans ce qui suit nous détaillons ces fonctionnalités.

Fonctionnalité de dépliement de graphe (Unfolding) Le dépliement de graphe appelé "Unfolding" est une transformation de duplication des blocs de calcul permettant d'augmenter le parallélisme de tâches de la description d'une applications de type DSP. Elle est classiquement utilisée dans les implémentations bas-niveau sur architecture FPGA, DSP at ASIC [PM91]. Nous proposons de l'utiliser dans notre contexte pour dérouler la boucle principale afin d'augmenter le nombre de tâches en exploitant le parallélisme de donnée et ainsi obtenir un taux d'occupation plus élevé des éléments de calcul et une meilleure répartition des charges. Dans ce qui suit nous présentons un exemple de dépliement de graphe sur une transformée en Z :

$$Z_n = a * X_n + b * Y_n$$

Si k est la k^{emme} itération d'un dépliement de degré J. Alors, l'application dépliée devient :

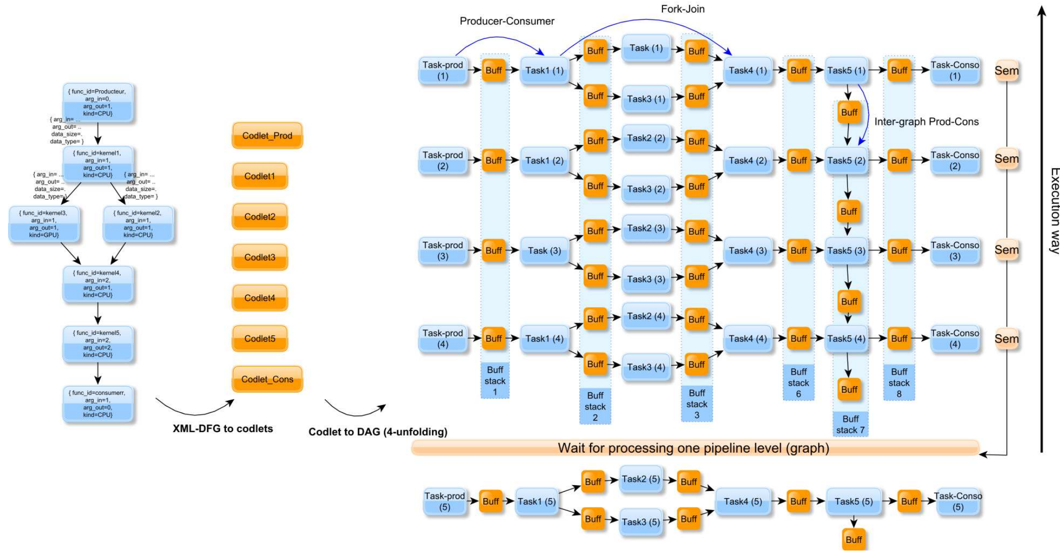

FIGURE 6.5 – Transformation du DFG de l'algorithme 3 vers un DAG de tâches exploitant différents niveaux de parallélisme

$$\begin{cases} Z_{j*k} = a * X_{j*k} + b * Y_{j*k} \\ Z_{j*k+1} = a * X_{j*k+1} + b * Y_{j*k+1} \\ .... \\ Z_{j*k+(j-1)} = a * X_{j*k+(j-1)} + b * Y_{j*k+(j-1)} \end{cases}$$

Nous illustrons dans la figure 6.5 le dépliement de dégrée 4 de l'algorithme 3. Lors de l'exécution, nous implémentons chaque acteur du DFG-XML avec la structure StarPU "Codlet". Chaque Codlet décrit les tâches et contient les informations de chaque acteur : fonctions associées, nombre d'arguments d'entrées et de sortie, type d'élément de calcul ciblé etc. Ensuite, pour chaque itération nous créons les tâches, et nous les connectons entre elles selon la description DFG-XML. Finalement, nous déployons dynamiquement plusieurs itérations du DFG pour créer le DAG de tâches. Le degré de dépliement J peut être ajusté selon le taux d'occupation, la répartition des charges et la disponibilité des ressources.

Fonctionnalité de réutilisation de buffers Les applications de type DSP ont souvent des schémas de communications statiques. En outre, les structures de données manipulées par un algorithme de type DSP restent généralement de tailles égales à travers les itérations. De ce fait, avec cette fonctionnalité, nous proposons de réutiliser les buffers afin de réduire le surcoût du à l'allocation et la libération des espaces mémoires. Lors de l'exécution, un nombre fixe de buffers est alloué en tenant compte des ressources disponibles sur le nœud de calcul, les tailles des buffers de l'application décrites dans le DFG-XML : taille de données, type de données, nombre de dépendances et finalement, le dégrée de dépliement du graphe. Dans notre exemple illustré dans la figure 6.5, nous allouons 4x8 buffers ce qui correspond à 8 buffers par niveau

de graphe et 4 niveaux correspondant à un dégrée de dépliement J=4. Ensuite, à chaque fin d'itération correspondant à la fin d'un niveau du DAG, nous affectons à la nouvelle itération les buffers libérés. Pour implémenter ce mécanisme, nous utilisons un sémaphore à compteur initialisée à J. A chaque début d'itération le sémaphore est décrémentée par l'affectation d'un niveau de buffers à un niveau du graphe. Le sémaphore est incrémenté à chaque fin d'itération par une récupération des buffers du niveau de graphe terminé. Si le sémaphore est égal à 0 alors la soumission d'une nouvelle itération est retardée. A noter que ce mécanisme permet également de contrôler le dégrée de dépliement J. Il retarde seulement la soumission d'un niveau de graphe supérieur à J mais ne retarde pas le calcul.

Fonctionnalité de "pipelining" de tâches Les applications de type DSP traitent généralement un nombre élevé d'itérations ce qui se traduit par un nombre important de tâches à gérer. Dans cette fonctionnalité nous limitons le nombre de tâches soumises au "runtime" afin de réduire les sur coûts dus à : la gestion des dépendances, l'ordonnancement, la mise à jour des statuts des tâches etc. Cette fonctionnalité est générée par le même mécanisme d'implémentation que la fonctionnalité précédente. A l'execution, seulement un nombre fixé de niveaux de pipeline est soumis ce qui limite également le nombre de tâches soumises. Ce nombre égale à J est contrôlé par le sémaphore qui contrôle également les buffers disponibles. Ainsi, chaque niveau du pipeline correspond à un niveau de graphe. Dans notre example de la figure 6.5, le pipeline à une profondeur égale à 4. Cela permet d'alimenter continuellement les éléments de calcul avec un nombre fixe de tâches sans augmenter le surcoût du "Runtime".

Fonctionnalité de distribution du calcul sur plusieurs nœuds MPI Finalement, nous proposons à l'utilisateur de distribuer le calcul de son application sur plusieurs nœuds MPI sans pour autant manipuler les primitives de la librairie StarPU-MPI [Aug+12]. En effet, l'utilisateur peut distribuer les calculs en exploitant le parallélisme de données (SIMD). Il suffit alors d'identifier pour chaque nœud MPI les données à traiter et qui lui sont accessibles. Sachant que, dans notre modèle d'exécution, une unité de donnée est traitée à chaque itération. Pour effectuer cette répartition des données, l'utilisateur doit alors spécifier à l'aide d'une fonction prédéfinie appelée $MPI_data_schedule()$ pour chaque itération le nœud de calcul qui lui est associé. Ainsi, chaque nœud MPI identifie les itérations à exécuter et les unités de données à traiter. Dans le listing 6.3, nous présentons un exemple d'implémentation de cette fonction.

Listing 6.3 – Exemple de code de la fonction $MPI_data_schedule()$

```
1  int MPI_data_distribute(int loop, int rank)
2  {
3   return( loop % rank );
4  }
```

La fonction retourne pour chaque itération *loop* de la boucle globale, le rang du nœud MPI chargé de la traiter. Les itérations sont distribuées par modulo. Ainsi, le DAG de tâches global est subdivisé dynamiquement en plusieurs sous-graphes de tâches distribuées sur plusieurs nœuds MPI. A l'exécution, chaque nœud exécute son sous-graphe de tâches en suivant le

modèle décrit précédemment. Chaque nœud comporte un pipeline de graphes déployés et exploite la réutilisation de buffers.

6.3.2.3 Niveau 3 : Le modèle d'exécution

Dans ce dernier niveau, nous nous intéressons à produire une exécution efficace des tâches soumises à l'architecture. Comme illustré dans la figure 6.6, dans chaque nœud MPI, le support exécutif StarPU est en charge de gérer le sous-DAG de tâches qui lui est attribué par l'utilisateur, comme expliqué dans la description du niveau précédent. Il ordonnance dynamiquement les tâches sur les éléments de calcul en tenant compte des dépendances de données entre tâches. Plusieurs politiques d'ordonnancement peuvent être utilisées par le programmeur lors de l'execution en manipulant des variables d'environnement. Des politiques comme le vol de tâches "Work stealing" ou HEFT ont pour objectif de répartir les charges sur les éléments de calcul tout en préservant la localité des données. En outre, l'utilisateur peut également activer de la même façon des fonctionnalités comme les copies asynchrones et le traitement en "Streaming" sur les GPUs permettant de mieux occuper ces derniers. Cela permet d'augmenter les performances en recouvrant les communications et les calculs. Cependant, pour gagner d'avantage en performance, nous proposons deux fonctionnalités spécifiques aux applications DSP que nous présentons dans le paragraphe suivant.

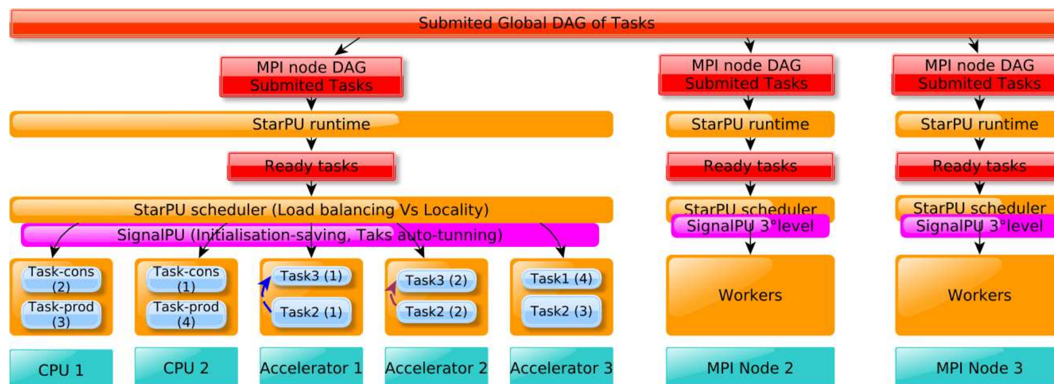

FIGURE 6.6 – Troisième niveau : exécution efficace des tâches grâce au support exécutif StarPU

Persistance des initialisations Plusieurs acteurs (fonctions) dans les applications DSP incluent une partie d'initialisation de certaines données ou de variables utilisées par la suite pour le calcul. Celle-ci peut nécessiter un temps de calcul important en comparaison à la partie de calcul propre. Par exemple, pour un filtre de Gabor utilisé dans l'application de Saillance décrite dans le chapitre 8, la partie d'initialisation est $150x$ plus longue en temps d'execution que la partie de calcul. Afin de réduire les temps d'exécution des tâches, il est donc important de n'exécuter qu'une seule fois la partie d'intilisation sur chaque unité de calcul. Or, le modèle d'exécution de StarPU n'inclut pas cette fonctionnalité, puisque les tâches correspondant à l'exécution des différentes itérations d'un acteur sont indépendantes entre elles. Nous proposons pour ces raisons d'implémenter une fonctionnalité permettant de

garantir une seule exécution par unité de calcul de la partie d'intilisation de chaque acteur comme mentionné précédemment. L'utilisateur définit les fonctions de l'acteur sous forme de deux sous-fonctions : "Init" et "Exec". A chaque première exécution d'une fonction sur un élément de calcul, les données initialisées persistent sur l'élément de calcul. Pour chaque tâche "sœur" exécutant par la suite la même fonction, lui sont affectées les données persistant sur l'élément de calcul et la partie d'initialisation n'est pas ré-exécutée.

Auto-tuning des kernel GPUs Comme déjà présenté auparavant, les clusters aujourd'hui comportent généralement des éléments de calcul hétérogènes en terme de type d'architecture, exemple : CPU, GPU, Xeon Phi, etc. Cependant, au sein d'un même type d'élément de calcul, peut également exister de l'hétérogénéité. Par exemple, un cluster peut contenir plusieurs GPU de différentes générations avec différentes configurations matérielles, et par conséquent avec différentes capacités de traitement. Les GPU peuvent différer par le nombre de cœurs qu'ils comportent, par la taille de la mémoire et des registres ou par leurs débits et latences mémoires.

Les performances d'un GPU peuvent être fortement affectées par le nombre de threads par blocs paramétrés au lancement d'un "kernel". Or, ces paramètres dépendent de plusieurs facteurs. En effet, un paramétrage optimal du nombre de thread par blocs dépend du GPU ciblé, de l'algorithme de la fonction "kernel" et de la taille des données traitées. Ces contraintes font, d'une part, du paramétrage GPU une mission difficile pour l'utilisateur qui doit non seulement connaître les configurations matérielles des périphériques, mais aussi adapter le code de chaque "kernel" en fonction du GPU ciblé et de la taille des données traitées. D'autre part, un paramétrage a priori d'un "kernel" n'est pas forcément optimal. L'outil "Occupancy calculator" proposé par NVIDIA [Ngu07] estime ces paramètres sur des calculs visant à augmenter l'occupation des cœurs GPU en tenant compte du nombre de registre et de la taille de la mémoire partagée "shared" utilisés. Or, ces paramètres peuvent dépendre de la taille des données. En outre, d'autres facteurs comme la disposition des données en mémoire globale peuvent fausser ces estimations. Afin de résoudre cette problématique, nous proposons une approche d'auto-tuning à l'exécution des paramètres de Threads par blocs dans les "kernel" GPUs. Cette fonctionnalité, spécifique à notre environnement SignalPU pour le traitement des applications de type DSP, est basée sur l'exploration itérative et exhaustive des paramètres avec un pas paramétrable dans un ensemble de possibilités à fonction 3D borné par le programmeur. Concrètement, nous proposons à l'utilisateur une fonction : $SignalPU_thread_per_block()$ permettant d'identifier les paramètres optimaux pour chaque tâche exécutant un "kernel" avec une taille de données propre sur chaque GPU. L'espace de paramètres est exploré itérativement et des mesures de temps de calcul des "kernel" sont relevées. Les paramètres présentant le temps de calcul le plus performant sont considérés comme optimaux et sont sauvegardés pour chaque couple tâche-GPU. Dans le listing 6.4, nous présentons un exemple de code utilisant la fonctionnalité d'auto-tuning des paramètres de nombre de Threads par blocs. L'intéret de cette fonctionnalité est montré dans les chapitres 7 et 8. A noter qu'il serait envisageable de proposer d'autres stratégies d'exploration plus efficaces pour réduire le nombre d'itérations de recherche des paramètres optimaux.

Listing 6.4 – Exemple de code utilisant la fonctionnalité d'auto-tuning des paramètres de nombre de Threads par blocs

```
1  void fonction_GPU_exec(void *buffers[], void* _args )
2  {
3
4  int N =  STARPU_VECTOR_GET_NX(buffers[0]);
5  float *val = (float *)STARPU_VECTOR_GET_PTR(buffers[0]);
6  float *val_out = (float *)STARPU_VECTOR_GET_PTR(buffers[1]);
7
8  dim3 ThrBloc;  dim3 Nbloc;
9
10 ThrBloc = SignalPU_thread_per_block(32,256,32,256,worker,info_codlet); /// appel de la fonction d'
       auto-tuning
11
12 Nbloc.x␣=␣sqrt(N),ThrBloc.x;␣␣␣␣Nbloc.y␣=␣sqrt(N),ThrBloc.y;
13
14 image_inverse_cuda<<<Nbloc,ThrBloc,0,starpu_cuda_get_local_stream()>>>(val,val_out,␣n);
15
16 cudaStreamSynchronize(starpu_cuda_get_local_stream());
17
18 return;
19 }
```

6.4 Comparaison et discussion

Dans cette section, nous comparons notre MdPP SignalPU avec StarPU afin d'identifier ses avantages et inconvénients. Les axes principaux de comparaison que nous choisissons sont le niveau d'abstraction et la facilité d'utilisation tant ils sont importants et influent sur plusieurs autres critères d'évaluation d'un modèle de programmation. Dans ce contexte et pour les applications de type DSP, SignalPU présente un point d'entrée avec un niveau d'abstraction supérieur à StarPU.

Cela est illustré à travers l'exemple d'implémentation de l'algorithme 3 avec les deux modèles. Pour décrire l'application et exploiter du parallélisme avec StarPU, le programmeur doit manipuler sa bibliothèque C pour créer explicitement les structures de Codlets pour chaque fonction, il doit également renseigner les attributs prédéfinis de chaque structure "Codlet" comme : le nom de la fonction, le nombre d'argument et les modes d'accès aux buffers. Il doit par la suite créer les tâches nécessaires correspondant au traitement d'une unité de donnée avec chaque fonction. Ces tâches doivent également être connectées à travers des buffers alloués manuellement avec des fonctions prédéfinies du module de gestion des données. Finalement, il doit soumettre les tâches en utilisant les fonctions StarPU adéquates. Cette manipulation est coûteuse en temps de programmation et nécessite une connaissance préalable et approfondie de la librairie C de StarPU. Avec SignalPU, l'utilisateur doit seulement exprimer son algorithme sous forme de graphe orienté. Cette représentation est plus naturelle et ne nécessite pas de pré-requis en terme de programmation. La saisie du graphe peut être simplifiée à l'aide d'interfaces graphique gérant des descriptions GraphML comme yEd [Yed].

SignalPU présente également l'avantage d'apporter plusieurs fonctionnalités spécifiques aux applications de type DSP comme le dépliement de graphe, le piplining de tâche, la réutilisation des buffers, la distribution MPI, l'auto-tuning et la persistance des initialisations. Ces fonctionnalités ne sont pas présentes nativement dans la librairie C de StarPU. Le programmeur doit les implémenter manuellement dans son code afin d'exploiter les différents niveaux

de parallélisme tout en ne générant pas de surcoûts.

6.5 Conclusion

Dans ce chapitre, nous avons présenté dans un premier temps StarPU et un exemple de son utilisation pour l'implémentation d'une application de type DSP synthétique. Ensuite, nous avons présenté SignalPU, un modèle de programmation haut niveau basé sur une extension de StarPU avec le modèle de calcul DFG. A cette extension nous avons rajouté plusieurs fonctionnalités spécifiques à l'implémentation des applications de type DSP. Cela permet de proposer un haut niveau d'abstraction et une facilité d'utilisation supplémentaire comme discuté dans la section 6.4. De cette discussion, nous avons également identifié les avantages de SignalPU, ce qui nous a permis de montrer son intérêt des points de vue de l'accessibilité et de la facilité d'utilisation en comparaison à StarPU pour le domaine spécifique des applications de type DSP. Dans le chapitre suivant, nous présentons une validation de SignalPU sous les angles de l'expressivité et des performances.

Chapitre 7

Validation

7.1 Introduction

Dans le présent chapitre, nous nous intéressons à la validation de notre modèle de programmation SignalPU au travers d'expérimentations d'implémentation d'applications synthétiques sur clusters hétérogènes. Nous présentons dans un premier temps, section 7.2, une bibliothèque d'opérateurs de charge permettant de créer des applications synthétiques. Nous montrons la construction d'une application synthétique à l'aide de ces opérateurs de simulation. Ensuite, dans la section 7.3, nous présentons l'implémentation de cette application avec SignalPU que nous comparons à une implémentation avec les environnements MPI, OpenMP et CUDA. Ces implémentations sont ensuite comparées en termes de performances dans la section 7.4.

7.2 Opérateurs de charge paramétrables pour la description d'applications synthetiques

Afin de permettre aux utilisateurs de facilement tester notre environnement de programmation SignalPU, nous avons conçu une bibliothèque d'opérateurs de charge. Ces opérateurs paramétrables comportent des implémentations sur différents éléments de calcul : CPU, GPU. Cela permet à l'utilisateur de facilement implémenter et simuler le comportement d'applications de type DSP avec différentes caractéristiques de granularités et d'échelles, présentées dans le chapitre 3.

Dans ce qui suit nous proposons de simuler l'application DSP décrite dans l'algorithme 4. Cette application simule le traitement d'un ensemble d'images de même taille avec une échelle d'une image traitée par itération. Les acteurs de charges temporelles de k microsecondes traitent les pixels de l'image en entrée. La granularité de l'application suit la décomposition fonctionnelle de l'algorithme. L'application comporte des dépendances multiples entre les acteurs exécutant des fonctions CPU/GPU. Dans le code 7.1 est représentée la description XML de deux acteurs de cette application construite en utilisant les opérateurs de charge $(producer, pixelProcessimu)$, ainsi que leurs connexions. Pour simuler un acteur de charge égale à $k1$ avec deux dépendances, il suffit alors d'initialiser le champ XML k_kernel avec le nom de la fonction $pixelProcessimu$, et le champ k_time avec la charge de temps égale à $k1$. Les acteurs sont tous construits de la même façon en utilisant des opérateurs proposés de type :

Algorithm 4 Application synthétique

Input : A image r_im of size $w \cdot l$,
Number of images (Nbr)
Output : Processed image
1: **for each** $image_i$ in Nbr **do**
2: $(Var_{1_1}, Var_{1_2}) \leftarrow Producer()$
3: $Var_{2_1} \leftarrow PixelProces(Var_{1_1}, k_1)$
4: $Var_{2_2} \leftarrow PixelProces(Var_{1_2}, k_2)$
5: $(Var_{3_1}, Var_{3_2}, Var_{3_3}) \leftarrow JoinFork(Var_{2_1}, Var_{2_2})$
6: $Var_{4_1}, \leftarrow PixelProces(Var_{3_1}, k_3)$
7: $Var_{4_2}, \leftarrow PixelProces(Var_{3_2}, k_4)$
8: $Var_{4_3}, \leftarrow PixelProces(Var_{3_3}, k_5)$
9: $Processed_image \leftarrow Consumer(Var_{4_1}, Var_{4_2}, Var_{4_3})$
10: **end for**

1. Producteur : acteur produisant une ou plusieurs sorties de données.
2. Consommateur : acteur consommant une ou plusieurs entrées de données.
3. Consommateur-Producteur : acteur consommant une entrée et produisant une sortie.
4. Join-Fork : acteur consommant une ou plusieurs entrées et produisant une ou plusieurs entrées.

Il est à noter qu'il est possible de générer des opérateurs avec charge de calcul variantes avec les itérations selon des fonctions précisées par l'utilisateur, par exemple, loi Gaussienne.

Listing 7.1 – Description XML partielle de l'algorithme 4 à l'aide d'opérateurs de charge paramétrables

```
<!-- ***************************************
                    NODES
***************************************-->
<node id="n0">
  <data key="k_node_id">n0</data>
  <data key="k_node_name">producer</data>
  <data key="k_kernel">producer(out0,out1)</data>
  <data key="k_kind">CPU</data>
</node>

<node id="n1">
  <data key="k_node_id">n1</data>
  <data key="k_node_name">PixelProces</data>
  <data key="k_kernel">PixelProcesSimu(in0,out0)</data>
  <data key="k_kind">CPU/GPU</data>
  <data key="k_time">k1</data>
</node>

<node id="n2">
  <data key="k_node_id">n2</data>
  <data key="k_node_name">PixelProces</data>
  <data key="k_kernel">PixelProcesSimu(in0,out0)</data>
  <data key="k_kind">CPU/GPU</data>
  <data key="k_time">k2</data>
</node>

<!-- ***********************************
                  EDGES
*********************************** -->
<edge source="n0" target="n1">
```

```
31      <data key="k_edge_name">n0_to_n1</data>
32      <data key="k_port_from">out0</data>
33      <data key="k_port_to">in0</data>
34      <data key="k_type">mat</data>
35      <data key="k_size">262144</data>
36    </edge>
37
38    <edge source="n0" target="n2">
39      <data key="k_edge_name">n0_to_n2</data>
40      <data key="k_port_from">out0</data>
41      <data key="k_port_to">in0</data>
42      <data key="k_type">mat</data>
43      <data key="k_size">262144</data>
44    </edge>
```

7.3 Implémentations

Dans cette section, nous présentons deux implémentations de l'application synthétique décrite précédemment. La première implémentation est construite en utilisant les opérateurs de charge paramétrables de SignalPU et la deuxième implémentation est construite en utilisant les environnements MPI, OpenMP 4.0 et CUDA. En effet, avec la version 4.0 du standard OpenMP, il est maintenant possible de décrire un graphe de tâches avec des dépendances de données, ce qui est adapté à la modélisation de notre application. Il est également possible d'exprimer des "kernels" pour accélérateurs, en l'occurrence des GPUs. Toutefois leur compilation n'est pas encore possible avec GCC. Pour cette raison, nous nous sommes tournés vers une combinaison CUDA+OpenMP dans laquelle nous utilisons OpenMP pour distribuer les calculs à travers les nœuds du Cluster. Ainsi, ces deux implémentations ciblent une exécution sur cluster hétérogène (CPU-GPU). Dans ce qui suit nous décrivons plus en détail ces deux implémentations.

7.3.1 Implémentation SignalPU

L'implémentation SignalPU de l'application synthétique consiste principalement à décrire le DFG-XML comme illustré dans le listing 7.1. Cela produit le graphe orienté présenté dans la figure 7.1. Grâce aux opérateurs de charges paramétrables, l'utilisateur n'a pas besoin de coder les fonctions et de les introduire dans l'environnement. Seule exception pour une exécution multi-nœuds MPI, il doit exprimer la distribution des images d'entrée pour chaque nœud. Pour cela il doit simplement modifier la fonction prédéfinie $MPI_data_schedule()$ selon la répartition qu'il souhaite. Le listing B.3 présente un exemple de code de cette implémentation.

7.3.2 Implémentation MPI+OpenMP+CUDA

Contrairement à une implémentation SignalPU, dans l'implémentation MPI+OpenMP+CUDA illustrée dans le listing 7.2, le programmeur doit manipuler les directives, fonctions et structures spécifiques des 3 environnements. D'abord il doit exprimer les tâches à l'aide de la directive OpenMP *task* et les connecter avec des directives *depend*

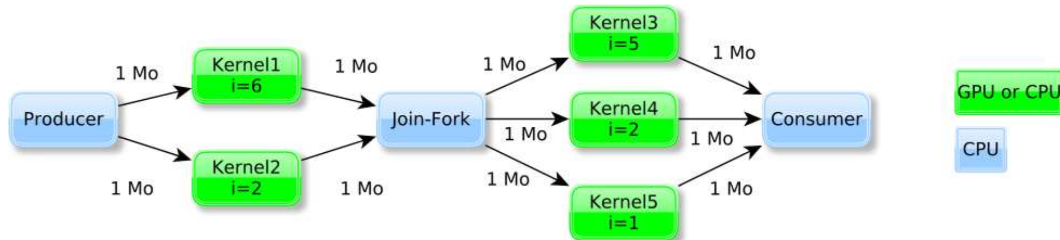

FIGURE 7.1 – Illustration du DFG-XML de l'application synthétique

afin de synchroniser les flux de données. Ensuite, il doit déployer la boucle principale sur une région parallèle de threads en utilisant la directive *parallelfor* comme montré dans la ligne 21 du listing 7.2. En outre, pour cibler les GPU sans engendrer de sur-coût, l'utilisateur doit gérer les allocations mémoire de chaque Thread en les sortant de la boucle principale comme montré dans la ligne 24. Cela permet d'éviter les libérations et allocations répétitives. Il doit également gérer les communications CPU-GPU pour les masquer et réduire leurs impacts. Pour cela, il doit les recouvrir avec des exécutions en utilisant des fonctions de transfert et d'exécution asynchrones sur différents Streams (ligne 4,6,7). Pour augmenter l'utilisation de l'architecture comportant des éléments de calcul hétérogènes, le programmeur doit explicitement répartir les charges de calcul sur les différents GPU comme montré dans la ligne 31. Aussi, afin d'optimiser les temps de calcul des tâches GPU, il doit explicitement optimiser les "kernels" en paramétrant chacun selon le GPU ciblé et les données en entrée comme illustré dans la ligne 5. Finalement, pour distribuer les calcul sur les nœuds du cluster, le programmeur doit utiliser les primitives MPI pour créer les processus parents dans chaque nœud en spécifiant les données que chaque processus doit traiter (ligne 28,29). Toutes ces optimisations sont nécessaires pour garantir une exécution efficace. Cependant, elles engendrent un effort de programmation important.

Listing 7.2 – Example de code MPI+OpenMP+CUDA pour l'implémentation de l'application synthétique (algorithme 4). Code équivalent compilé présenté en listing B.1 et B.2

```
1  void pixPrSimuCuda(float* H_in,float* D_in,
2  float*H_out,float*D_out,k,int size,int dev);
3  {
4  cudaMemcpyAsync(D_in,H_in,size,HtoD,stream_in);
5  Auto_tune(nblocks,threadsB);
6  kernelCuda<<<nblocks,threadsB,stream_exec>>>(D_in,D_out,k,n);
7  cudaMemcpyAsync(H_out,D_out,size,DtoH,stream_out);
8  }
9
10 int main(int argc, char **argv)
11 {
12  int high = 512 ; int with=512;
13  int size_img = high*with;
14  const int size_loop = (int) atoi((argv[1]));
15  const int unfolding = (int) atoi((argv[2]));
16  int rank;  int n=0; int dev;
17
18 MPI_Init(&argc,&argv);
19 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
20
21 #pragma omp  parallel num_threads(unfolding)
22 {  float * H[nb_buf];  float * D[nb_buf];
23
24  Host_alloc(H); Device_alloc(D);
25
```

```
26 #pragma omp  for private(n) private(dev)
27
28 if (rank==0) for(n=0; n<size_loop_1; ++n)
29 else for(n=size_loop_1+1; n<size_loop_2; ++n)
30 {
31 dev=load_balance(n,GPUs_number);
32
33 #pragma omp task depend(out:H[0],H[1])
34 producer( H[0], H[1] , size_img );
35
36 #pragma omp task depend(in:H[0]) depend(out:H[2])
37 PPSimuCuda(H[0],D[0],H[2],D[2],7,size,dev);
38
39 #pragma omp task depend(in:H[1]) depend(out:H[3])
40 PPSimuCuda(H[1],D[1],H[3],D[3],3,size,dev);
41
42 #pragma omp task depend(in:H[2],H[3])
43  depend(out:H[4],H[5],H[6])
44 ForkJoin(H[2],H[3],H[4],H[5],H[6],size);
45
46 #pragma omp task depend(in:H[4]) depend(out:H[7])
47 PPSimuCuda(H[4],D[4],H[7],D[7],6,size,dev);
48
49 #pragma omp task depend(in:H[5]) depend(out:H[8])
50 PPSimuCuda(H[5],D[5],H[8],D[8],2,size,dev);
51
52 #pragma omp task depend(in:H[6]) depend(out:H[9])
53 PPSimuCuda(H[6],D[6],H[9],D[9],2,size,dev);
54
55 #pragma omp task depend(in:H[7],H[8],H[9])
56 Consumer(H[7],H[8],BufH[9]);
57   }
58 }
```

Ainsi, en comparaison à l'implémentation SignalPU, le programmeur doit manipuler 3 environnements de programmation. Il doit manipuler des directives spécifiques pour décomposer son algorithme sous forme de tâches. Il doit faire face à la gestion de la mémoire et des communications CPU-GPU. Il doit également distribuer explicitement les calculs sur les Threads et les processus pour garantir une bonne répartition des charges. Finalement, il doit adapter l'execution des "Kernels" selon les caractéristiques des GPUs. Par conséquent, il est possible d'affirmer qu'il est plus facile pour le programmeur d'utiliser SignalPU pour implémenter des applications DSP sur cluster hétérogène. Dans ce qui suit, nous présentons les expérimentations et résultats obtenus avec les deux implémentations pour comparaison et évaluation des approches.

7.4 Expériences et résultats

Dans cette section, nous présentons 4 expérimentations et leurs résultats sur les deux implémentations de l'application synthétique. Le but est d'évaluer la performance de notre approche SignalPU en la comparant aux performances obtenues avec MPI+OpenMP+CUDA. L'architecture utilisée pour ces expériences est un cluster hétérogène CPU-GPU composé des deux nœuds de calcul décrits dans le tableau 7.1. Les nœuds sont connectés à travers un réseau "Infiniband".

Node	CPU	GPU
Archi1	Intel Core i7 920 (4 Cores)	2 NVIDIA Quadro 4000 + NVIDIA Quadro 2000
Archi2	Intel Core i7 3820 (4 Cores)	NVIDIA GTX TITAN + NVIDIA GTX 780
Archi3	Intel Core i7 3820 (4 Cores)	NVIDIA GTX 480 + NVIDIA GTX 480
Archi4	Intel Core i7 3820 (4 Cores)	NVIDIA GTX 380

TABLE 7.1 – Architecture du cluster utilisé pour les expériences

7.4.1 Performances globales

Dans cette première expérience nous présentons le traitement d'un nombre d'images important sur des configurations matérielles différentes. L'objectif est de mesurer l'exploitation des différents niveaux de parallélisme (tâche, données et graphe) et l'evolution des performances par rapport aux architectures. Nous traitons 1000 images avec l'application synthétique sur différentes configurations CPU-GPU en utilisant les deux implémentations : SignalPU avec un degré de dépliement du graphe J=10 et une implémentation MPI + OpenMP + CUDA présentée dans la section précédente. Les résultats et leur comparaison sont présentés dans la figure 7.2.

Pour la configuration matérielle à base de CPU seulement, nous pouvons noter que l'accélération est proportionnelle au nombre de cœurs pour les deux implémentations. En effet, grâce à l'exploitation du parallélisme de tâche et de graphe, les cœurs de calcul sont exploités efficacement ce qui permet un bon passage à l'échelle des performances. Le parallélisme de tâche est exploité grâce aux dépendances entre tâches. Le parallélisme de graphe est lui exploité grâce au dépliement du graphe. Ainsi, nous obtenons pour les deux implémentations des résultats comparables en terme d'accélération et de temps de calcul.

Pour la configuration CPU-GPU sur 1 Cœur CPU+1 Quadro 4000, les performances sont améliorées pour les deux implémentations grâce à l'exploitation additionnelle du parallélisme de données (SIMD) sur les GPU. Nous obtenons par conséquent pour l'implémentation SignalPU une accélération de 61x comparée à une exécution à 1 cœur CPU. Pour l'implémentation MPI+OpenMP+CUDA nous obtenons une accélération de seulement environ 55x. La différence de résultats est due à la différence entre les modèles d'exécution. En effet, dans l'implémentation MPI+OpenMP+CUDA le modèle d'exécution est basé sur le Fork-Join des Threads au sein de la région parallèle. Les Threads sollicitent par conséquent pour chaque tâche l'initialisation de l'élément de calcul à travers les drivers systèmes. Cependant, dans le modèle d'exécution SignalPU, les tâches sont distribuées sur des Threads associés aux éléments de calcul appelés "Workers". Par conséquent, les initialisations des éléments de calcul sont effectuées une seule fois. En outre, les Threads d'une région parallèle doivent s'attendre mutuellement "Join" ce qui n'est pas le cas dans SignalPU.

Pour l'execution sur 1 coeur CPU+2 Quadro 4000, nous obtenons une accélération d'environ 93x pour SignalPU et seulement 69x pour MPI+OpenMP+CUDA. De même, pour l'execution sur 1 coeur CPU+ 2 Quadro 4000+ 1 Quadro 2000, nous obtenons pour SignalPU une accélération d'environ 125x et seulement environ 89x pour MPI+OpenMP+CUDA. Cette différence de passage à l'echelle est due également au modèle d'exécution, mais aussi à la distribution dynamique des tâches. En effet, SignalPU dispose de plusieurs optimisations comme

la distribution désordonnée "out of order" ou le préchargement des données "Prefetching" permettant ainsi une meilleure occupation des éléments de calcul.

	1 Core (1CPU)	2 Cores (1CPU)	3 Cores (1CPU)	4 Cores (1CPU)	8 Cores (2CPUs)	12 Cores (3CPUs)	16 Cores (4CPUs)	1Core +1Quadro4000	1Core +2Quadro4000	1Core +2Quadro4000 +1Quadro 2000	2CPU+4GPUs (Archi1+Archi2)	3CPU+6GPUs (Archi1+Archi2 +Archi3)	4CPU+7GPUs (Archi1+Archi2 +Archi3+Archi4)
MPI+OpenMP+CUDA (Time)	197,00	99,67	68,17	51,17	32,59	21,52	15,34	3,58	2,27	1,93	0,95	0,77	0,72
MPI+OpenMP+CUDA (Speedup)	1,00	1,98	2,89	3,85	6,04	9,15	12,84	54,98	86,91	101,90	207,37	255,84	273,61
SignalPU (Time)	198,33	100,67	68,83	51,83	36,50	25,73	19,35	3,25	2,12	1,58	0,65	0,45	0,39
SignalPU (Speedup)	1,00	1,97	2,88	3,83	5,43	7,71	10,25	61,03	93,70	125,26	305,13	440,74	508,55

FIGURE 7.2 – Comparaison des résultats de performances globales des implémentations SignalPU Vs MPI+OpenMP+CUDA. Les temps sont donnés en minute pour le traitement de 1000 images.

Finalement, dans l'execution multi-nœuds MPI, nous obtenons pour l'exécution sur 2 nœuds une accélération d'environ 305x pour SignalPU et seulement 188x pour MPI+OpenMP+CUDA. Pour 3 nœuds, elle est d'environ 440x pour SignalPU et 255x pour l'implémentation basée sur MPI+OpenMP+CUDA. Finalement, pour 4 nœuds totalisant 4 Coeurs CPU et 7 GPUs, l'accélération sur SignalPU est d'environ 508x et de 274x sur MPI+OpenMP+CUDA. Nous expliquons cette différence de résultats par la distribution dynamique. En effet, SignalPU permet d'exploiter d'avantage les éléments de calcul de chaque nœud MPI (Archi1, Archi2, Archi3, Archi4) sachant qu'ils disposent de GPUs hétérogènes. Il distribue équitablement les charges entre les nœuds, ce qui permet d'exécuter plus de tâches dans un laps de temps donné. Or, avec une distribution statique, certains Threads (les plus performants) doivent attendre les Threads les moins performants, ce qui retarde la région parallèle. Par conséquent, pour une implémentation MPI+OpenMP+CUDA à optimisation équivalente, nous obtenons une meilleure performance globale avec SignalPU grâce à son modèle d'execution basé sur la distribution dynamique d'un DAG de tâches.

7.4.2 Dépliement de graphe et ordonnancement dynamique

Dans cette seconde expérience, nous étudions les gains en performance obtenus par la combinaison l'ordonnancement dynamique avec le dépliement du graphe "unfolding". Nous traitons 1000 images sur une configuration matérielle CPU-GPU (3 cœurs CPU+ 2 Quadro 4000 + 1 Quadro 2000) avec différentes implémentations incluant ou pas les 2 optimisations :

1. **Ordonnancement dynamique** : labellisée "Dynamique scheduling", c'est une implémentation sans dépliement de graphe (J=1) mais avec une distribution dynamique des tâches au sein d'une itération. Une répartition des charges est effectuée au sein des éléments de calcul de chaque nœud mais non pas sur l'ensemble des nœuds.
2. **Ordonnancement statique** : labellisée "Static scheduling", c'est une implémentation sans dépliement de graphe (J=1) où chaque acteur est associé statiquement à un élément de calcul. Une exploration exhaustive est effectuée préalablement pour déterminer le meilleur placement statique.

3. **Ordonnancement statique avec dépliement du graphe** : labellisée "Static scheduling with unfolding", dans cette implémentation nous déplions la boucle principale de l'application sur dix niveaux (J=10). Cependant, le placement des acteurs est statique comme dans la première implémentation.
4. **Ordonnancement dynamique avec dépliement du graphe** : labellisée "Dynamique scheduling with unfolding", dans cette implémentation nous combinons les deux optimisations. Nous déplions le graphe avec J=10 et utilisons le scheduling dynamique pour répartir les charges des acteurs sur les 10 niveaux de DAG.

Dans la figure 7.8, sont illustrés les temps de calcul obtenus pour chaque implémentation sous la forme de 6 barres. Chaque barre correspond à un élément de calcul (3 cœurs CPU + 2 Quadro 4000+ 1 Quadro 2000). Chaque barre comporte 3 temps : un temps effectif d'execution appelée "Execution", un temps d'attente appelé "Sleep" et un temps de surcoût du support exécutif appelé "Overhead". La somme des 3 temps représente le temps global de traitement de l'application.

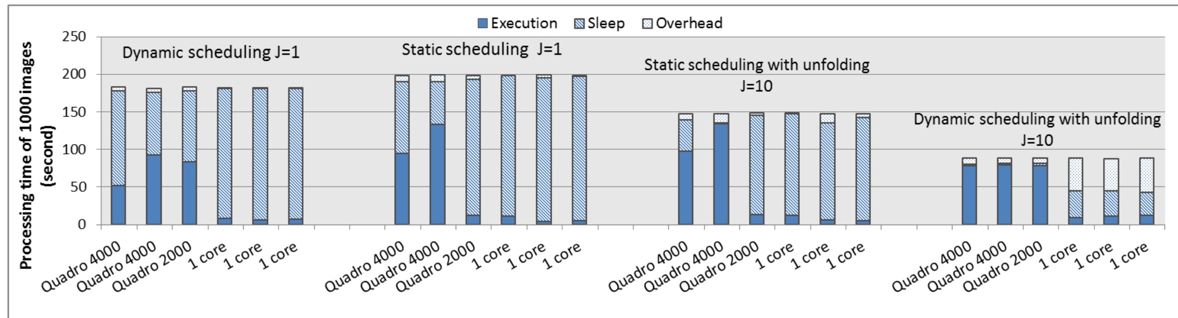

FIGURE 7.3 – Comparaison des résultats de performances globales des implémentations SignalPU Vs MPI+OpenMP+CUDA. Les temps sont données en secondes pour le traitement de 1000 images.

Le premier groupe de six barres à gauche du graphique représente les temps d'exécution de l'implémentation ("Scheduling" dynamique) sur l'architecture. Cette exécution n'est pas efficace pour les raisons suivantes : premièrement, la répartition des charges entre les 2 GPUs est effectuée mais n'est pas satisfaisante. En effet, comme montré dans les temps d'exécution effectifs labellisés "Execution", les temps sont inégaux. Le "Scheduler" est incapable de répartir les charges au sein de l'itération car les acteurs sont de granularités différentes. Deuxièmement, les temps d'attente labellisés "Sleep" sont élevés. Cela est dû à l'aspect itératif de l'application. En effet, les éléments de calcul les plus performants ou ayant moins de tâches à traiter doivent attendre les plus long avant d'entamer une nouvelle itération. L'exécution de cette implémentation est représenté dans le chronogramme de la figure 7.4.

Dans le second groupe de six barres de la figure 7.8, nous montrons les temps de calcul de l'implémentation ("Scheduling" statique). Les performances de cette implémentation sont moins bonnes que l'implémentation précédente. En effet, à cause de la distribution statique des acteurs, les temps d'exécution labellisés "Execution" sont inégaux et les temps d'attente labellisés "Sleep" sont plus important. Cela rallonge le temps de l'itération ce qui rallonge le

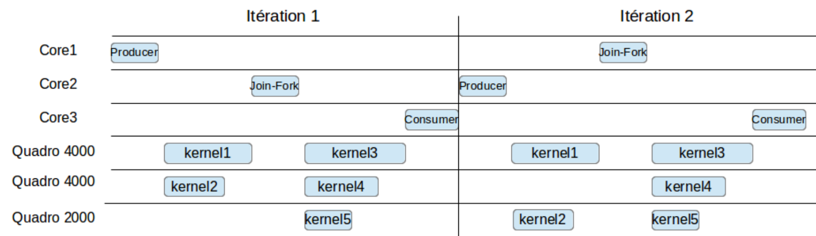

FIGURE 7.4 – Chronogramme représentant l'exécution de l'implémentation "Scheduling" dynamique

temps global. L'exécution de cette implémentation est représenté dans le chronogramme de la figure 7.5.

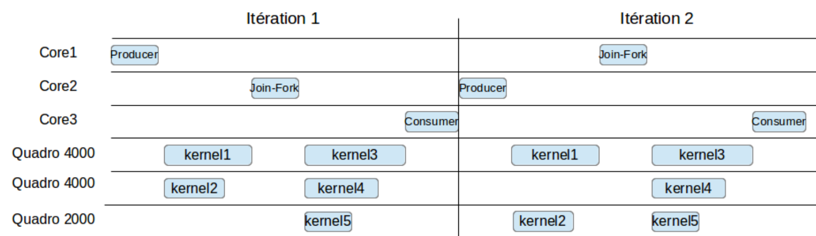

FIGURE 7.5 – Chronogramme représentant l'exécution de l'implémentation "Scheduling" statique

Dans le groupe de six barres à la troisième position, nous montrons les temps de calcul de l'implémentation ("Scheduling" statique avec dépliement du graphe). Les performances sont améliorées par rapport aux précédentes implémentations. Cela s'explique par une meilleure occupation des éléments de calcul ce qui se traduit par des temps d'attente (Sleep) moins importants. En effet, grâce aux dépliement du graphe (J=10), les éléments de calcul n'attendent pas l'exécution de la dernière tâche d'un niveau de graphe pour commencer l'exécution d'une autre itération. Cependant, les temps de traitement labellisés "Execution" restent inégaux entre les éléments de calcul à cause de la distribution statique des acteurs. L'exécution de cette implémentation est représenté dans le chronogramme de la figure 7.6.

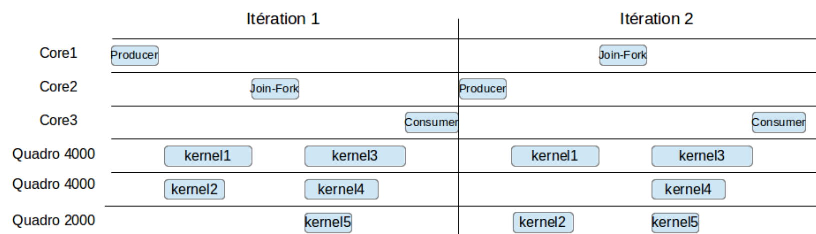

FIGURE 7.6 – Chronogramme représentant l'exécution de l'implémentation "Scheduling" statique avec dépliement du graphe

Le dernier groupe de six barres représente le temps de calcul de l'implémentation ("Scheduling" dynamique avec dépliement du graphe). Ici les performances obtenues sont meilleures

que dans les autres implémentations. Les temps d'exécution (Execution) sont améliorés et pratiquement égaux et les temps d'attente sont réduits. Cela est obtenu grâce à une combinaison de la distribution dynamique des acteurs avec un dépliement du graphe. En effet, ce dernier permet d'apporter plus de tâches au "Scheduler" ce qui lui permet de mieux répartir les charges entre les éléments de calcul. Par conséquent, comme proposé dans notre MdPP, il est intéressant et efficace de combiner l'ordonnancement dynamique avec un dépliement de graphe "unfolding" afin d'augmenter les performances d'implémentation sur architectures hétérogènes. L'exécution de cette implémentation est représenté dans le chronogramme de la figure 7.7.

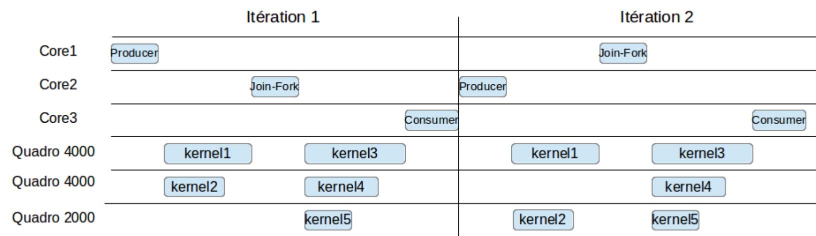

FIGURE 7.7 – Chronogramme représentant l'exécution de l'implémentation "Scheduling" dynamique avec dépliement du graphe

7.4.3 Surcoûts introduits par SignalPU

Dans cette expérience nous cherchons à mesurer l'impact des fonctionnalités spécifiques au domaine applicatif visé : dépliement du graphe, réutilisation des buffers, pipelining des tâches, etc, et de mesurer les sur coûts dus à la gestion des tâches, le "scheduling", la gestion mémoire, etc. Par cela, nous traitons un nombre d'images croissant et mesurons l'évolution des surcoûts engendrées par rapport au temps de calculs. Dans la figure nous montrons l'évolution du pourcentage des temps moyen de sur-coût dans les éléments de calcul par rapport aux temps global de calcul représentés par la courbe rouge. L'évolution du pourcentage des surcoût se stabilise vers 7%. Ce temps est du en grande partie à la gestion des allocations mémoire sur les GPUs. En effet, la distribution dynamique nécessite des allocations de déférente taille sur les mémoires de ces derniers. Cependant, il est à noter que ce temps reste inférieur aux temps d'accélération permis par la distribution dynamique (montré dans l'expérience précédente) car les éléments de calcul sont mieux exploités.

7.4.4 Apport de la persistance des initialisations et de l'auto-tuning sur la performance des tâches

La dernière expérience que nous proposons concerne le gain en performance des tâches apporté par l'utilisation des fonctionnalités de persistance des initialisations et de l'auto-tuning. Comme déjà présenté précédemment, la persistance de l'initialisation permet l'exécution de la partie d'initialisation d'un tâche une seule fois sur chaque élément de calcul. Tandis que la fonctionnalité d'auto-tuning permet de trouver les paramètres de Thread par blocs des

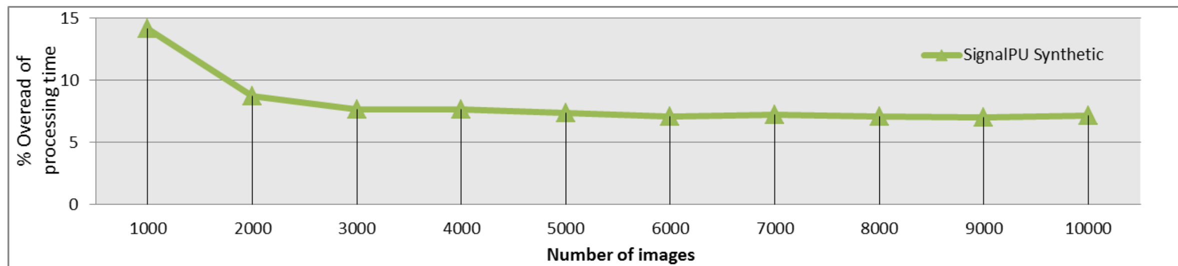

FIGURE 7.8 – Evolution du pourcentage des surcoûts dans le traitement d'un nombre d'images allant de 1000 jusqu'à 10000.

kernel GPUs offrant les meilleures performances en les explorant itérativement. Dans cette expérience nous mesurons pour l'acteur $PixelProcesingSimu()$ son temps de calcul sur deux GPUs différents. Les résultats sont montrés dans la figure 7.9. Premièrement, nous notons sur la première itération, l'exécution de la partie d'initialisation par la ligne bleue hachurée. Nous remarquons que cette partie est exécutée une seule fois pour chaque GPU. Deuxièmement, nous remarquons que les temps de calcul des Kernels peuvent varier du simple au double selon les paramètres choisis. Sur le graphe nous représentons la taille des blocs de Threads sur chaque dimension par un triplet (x,y,z). L'exploration converge vers une valeur des paramètres différente pour chaque GPU. L'exploration est effectuée itérativement et les résultats sont sauvegardés au fur et à mesure pour trouver la valeur optimale. En effet, nous obtenons respectivement les paramètres (128,1,1) et (256,1,1) pour la GTX 780 et la Quadro 4000 permettant de réduire les temps à respectivement 2 ms et 10 ms. Par conséquent, grâce aux deux fonctionnalités de persistance de l'initialisation et d'auto-tuning nous réduisons les temps de calcul et augmentons les performances des tâches.

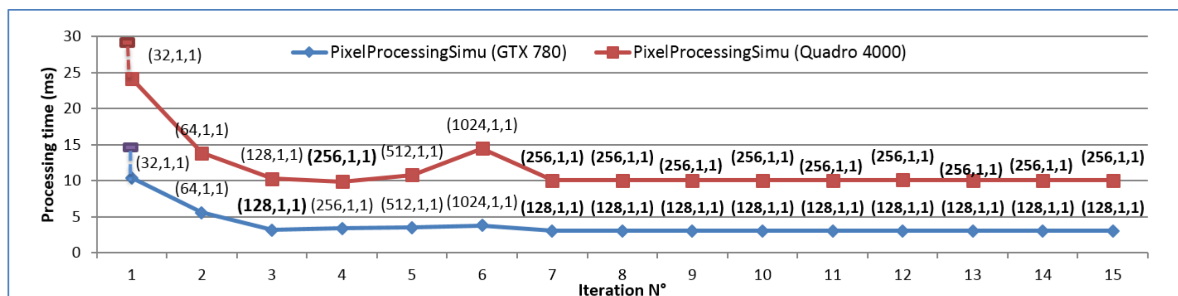

FIGURE 7.9 – Apport de la persistance des initialisations et de l'auto-tuning sur la performance des tâches

7.5 Conclusion

Nous avons présenté, dans ce chapitre, une validation de notre MdPP sur le plan de l'-expressivité et sur le plan des performances. Pour cela, nous avons dans un premier temps présenté une bibliothèque d'opérateurs de charge permettant de facilement simuler des ap-

plications de type DSP avec des granularités et des charges de communications et de calcul différentes. Ensuite, nous avons présenté une implémentation sur cluster hétérogène décrite avec MPI+OpenMP+CUDA que avons comparé avec l'implémentation SignalPU sur le plan de l'expressivité. Nous avons vu que l'implémentation SignalPU est plus facile à mettre en œuvre pour le programmeur. En outre, elle présente l'avantage d'inclure plusieurs fonctionnalités que l'utilisateur doit coder manuellement dans l'implémentation MPI+OpenMP+CUDA, comme : le dépliement du graphe, la distribution dynamique, la réutilisation des buffers, et l'auto-tuning, etc. Nous avons par la suite effectué plusieurs expérimentations de comparaison sur cluster hétérogène pour valider les performances de SignalPU. Nous avons conforté les performances globales de notre implémentation avec l'implémentation MPI+OpenMP+CUDA. Les performances obtenues avec SignalPU sont meilleures grâce notamment aux modèles d'exécution basés sur une distribution dynamique du DAG de tâches sur les éléments de calcul. En outre, à travers la deuxième expérimentation, nous avons montré l'intérêt de la fonctionnalité de dépliement du graphe pour augmenter le parallélisme et donc alimenter au mieux l'ordonnanceur dynamique. Dans la troisième expérience, nous avons mesuré les surcoût engendrés par notre flot de conception. Ce surcoût est stabilisé sur une valeur de 7% et est amorti par les gains de performances. Finalement, nous avons montré le gain de performance sur les tâches apporté par les optimisations de persistance des initialisation et l'auto-tuning. Par conséquent, nous avons validé notre approche de MdPP spécifique aux applications de type DSP en augmentant l'expressivité et en préservant les performances.

Quatrième partie

Expérimentations et comparaisons

Comme discuté dans le chapitre 2, un MdPP doit à la fois permettre d'augmenter la productivité du programmeur tout en préservant les performances. Dans ce manuscrit nous avons présenté deux MdPP spécifiques pour implémenter les applications DSP. Ces modèles sont basés sur le modèle de calcul DFG et sur des modèles d'exécutions différents. Ils présentent tous les deux un haut niveau d'abstraction en permettant au programmeur de spécifier son application avec un DFG.

Dans cette partie nous nous intéressons à l'évaluation et de la validation des deux MdPP PACCO et SignalPU présentés respectivement dans les chapitres 4 et 6. Nous utilisons ces deux modèles pour implémenter deux applications du monde réel selon des contraintes d'implémentation différentes : l'application de saillance dans un contexte de traitement massif des données, et l'application de suivi avec des contraintes de débit. Nous comparons à travers différentes expérimentations les performances des deux modèles afin de les valider. Dans le chapitre 8, sont présentés les expérimentations, les résultats et les comparaisons sur l'application de saillance. Ensuite, dans le chapitre 9, nous présentons l'application de suivi. Nous décrivons ensuite les expérimentations effectuées et montrons les résultats obtenus.

CHAPITRE 8

L'application de saillance

8.1 Introduction

Afin de valider les deux MdPP PACCO et SignalPU, nous présentons dans ce chapitre une évaluation sur une application du monde réel de traitement de l'image pour le calcul de cartes de salliance. Dans la section 8.2, nous décrivons cette application et présentons ses implémentations avec les deux MdPPs. Dans la section 8.3, nous présentons différentes expérimentations sur chaque implémentation, analysons et comparons les résultats obtenus.

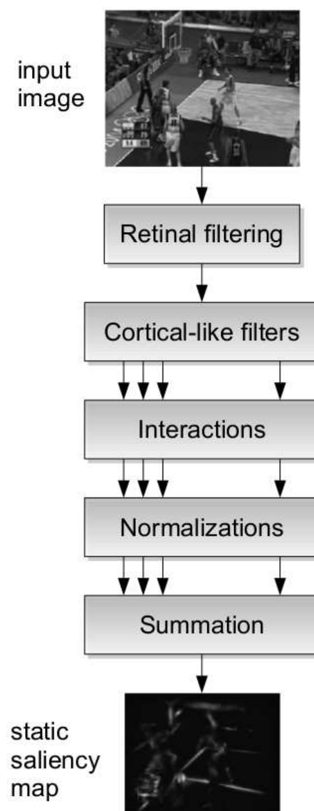

FIGURE 8.1 – Illustration des étapes du modèle de calcul de cartes de saillance visuelle proposé par [RHP11]

8.2 Description de l'application et des implémentations

L'application de calcul de cartes de saillance est une application de traitement d'images inspirée de la biologie. Elle vise à reproduire le fonctionnement de la rétine du primate pour détecter les régions d'intérêts dans une scène visuelle. En effet, son modèle s'inspire de la vision de l'être humain dans sa capacité à concentrer son attention sur des régions spécifiques.

Le modèle de calcul de cartes de Saillance que nous utilisons est celui proposé par [RHP11]. Ses différents étapes sont illustrées dans la figure 8.1. Il se base sur le traitement de l'image par plusieurs filtres afin de produire une image en niveau de gris où une zone à d'autant plus d'intérêt qu'elle est claire. L'algorithme 5 décrit en détails les étapes de traitement de ce modèle. Premièrement, l'image d'entrée (r_im) est traitée par un filtre de $Hanning$ pour simuler le premier bloc "retinal filtering" et réduire l'intensité des sommets. Ensuite, s'en suit une décomposition fréquentielle avec la transformé de Fourrier (FFT). Cette représentation fréquentielle (cf_fim) est traitée par un filtre de Gabor à 2 dimensions selon 4 bandes de fréquence et 6 orientations, réalisant ainsi le bloc "Cortical-like filters" de la figure 8.1. Les 24 cartes partielles ($cf_maps[i,j]$) sont replacées dans le domaine spatial avec la fonction de transformée de Fourrier inverse ($IFFT$) pour produire les cartes spatiales ($c_maps[i,j]$). Les pixels des ces dernières sont inhibés ou excités par la fonction $Interaction$ selon leurs fréquences et leur orientations. Les valeurs résultantes sont alors normalisées avec la fonction $Normalization$ basée sur la méthode Itti. Finalement, les 24 cartes partielles sont sommées pour produire une carte de saillance visuelle.

Algorithm 5 Application de Saillance

Input : An image r_im of size $w \cdot l$
Output : The saliency map
1: $r_fim \leftarrow Hanningfilter(r_im)$
2: $cf_fim \leftarrow FFT(r_fim)$
3: **for** $i \leftarrow 1$ **to** orientations **do**
4:   **for** $j \leftarrow 1$ **to** frequencies **do**
5:     $cf_maps[i,j] \leftarrow GaborFilter(cf_fim, i, j)$
6:     $c_maps[i,j] \leftarrow IFFT(cf_maps[i,j])$
7:     $r_maps[i,j] \leftarrow Interactions(c_maps[i,j])$
8:     $r_normaps[i,j] \leftarrow Normalizations(r_maps[i,j])$
9:   **end for**
10: **end for**
11: $saliency_map \leftarrow Summation(r_normaps[i,j])$

Dans ce qui suit, nous présentons l'implémentation de cette application en utilisant les deux modèles PACCO et SignalPU.

8.2.1 Implémentation PACCO

Dans l'implémentation PACCO, il faut tout d'abord construire le graphe d'architecture. Il représente l'architecture cible sous forme d'un graphe orienté dont les nœuds représentent les éléments de calcul et les arcs les liaisons de communication. Ensuite, il faut représenter l'application sous forme de graphe DAG à taux unique en utilisant une description XML. Pour cela, on représente les fonctions de l'algorithme ($Capture, Hanningfilter, FFT, GaborFilter, IFFT, Interactions, Normalizations, Display$) comme des acteurs dans le modèle DAG et les échanges de données comme des arcs connectant ces acteurs. Dans le graphe d'application, il est aussi nécessaire de préciser manuellement et statiquement le placement de chaque acteur sur l'architecture de calcul. Ensuite, il faut introduire les codes des fonctions en les encapsulant dans des classes C++. Finalement, il faut décrire les structures de données (taille et type) utilisées dans la description de l'application. A l'initialisation de l'environnement PACCO, des buffers sont introduits pour construire le graphe d'implémentation et le calcul est exécuté suivant le modèle BSP selon les principes présentés dans le chapitre 4 de ce mémoire. La figure 8.2 illustre l'implémentation de l'application de saillance sur un cluster à deux nœuds.

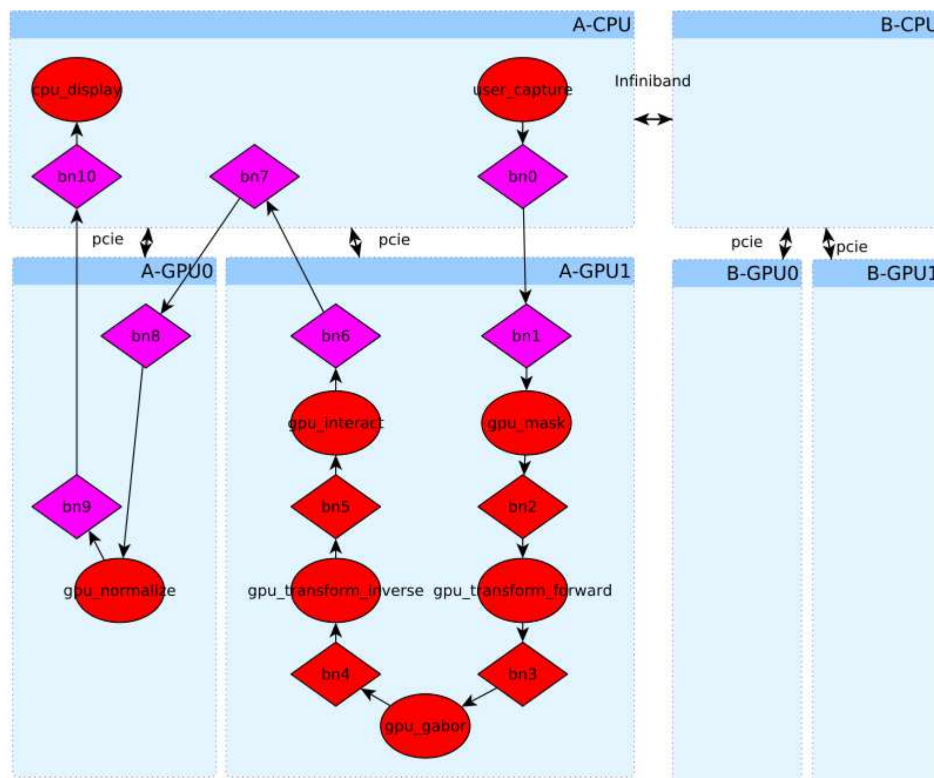

FIGURE 8.2 – Illustration du graphe d'implémentation de l'application de calcul de cartes de saillance avec l'environnement PACCO

8.2.2 Implémentation SignalPU

Pour implémenter l'application de saillance avec le modèle SignalPU, la première étape consiste à représenter l'algorithme 5 sous la forme d'un DFG synchrone à taux unique via l'interface DFG-XML. Pour cela, il faut représenter chaque fonction de l'algorithme comme un acteur accompagné de ses caractéristiques : nom et nature (CPU/GPU) des fonctions de calcul, arguments d'entrée, arguments de sortie, etc. Ensuite, nous représentons les flots de données entre les acteurs avec des arcs incluant les caractéristiques suivantes : argument d'entrée et de sortie, type et taille des données. La figure 8.3 illustre le graphe DFG-XML de l'application de calcul de cartes de saillance. La seconde étape consiste à inclure les codes des fonctions dans l'environnement en précisant pour chacune : une sous-fonction "Init" pour l'initialisation des variables, et une fonction "Exec" exécutée lors de l'évaluation de l'acteur. Enfin, il faut définir la répartition des calculs entre les nœuds MPI. Pour cela, il suffit de préciser grâce à la fonction prédéfinie $MPI_data_distribute$ la répartition des itérations du graphe entre les processus MPI. A noter que la description de l'application utilisée en entrée de SignalPU est la même que celle utilisée dans PACCO pour décrire l'application. Cependant, avec SignalPU, il n'est pas nécessaire de décrire l'architecture de calcul et il n'est donc pas nécessaire de préciser le placement des acteurs sur les éléments de calcul. Cela offre une portabilité supérieure.

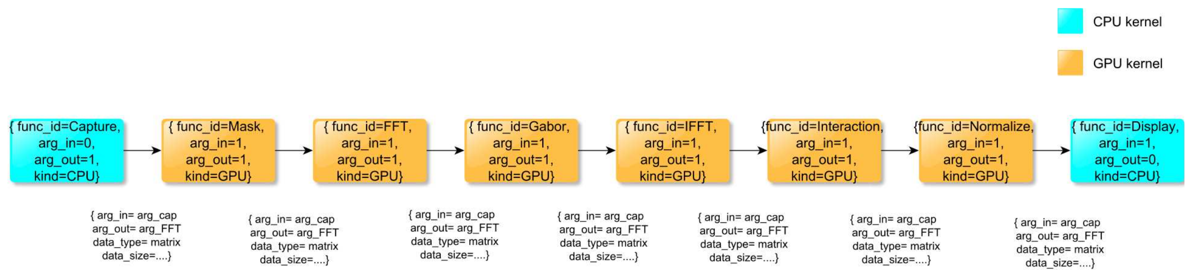

FIGURE 8.3 – Illustration du DFG-XML de l'application de calcul de cartes de saillance avec l'environnement SignalPU

8.3 Expérimentations et résultats

Dans cette section, nous présentons différentes expérimentations sur les deux implémentations PACCO et SignalPU de l'application de calcul de cartes de saillance. Le but étant de comparer et valider les performances de chaque modèle. Dans chaque expérimentation, nous présentons les résultats obtenus et les analysons. L'architecture cible des expériences est un cluster hétérogène CPU-GPU composé de quartes nœuds connectés via InfiniBand. Les caractéristiques des éléments de calcul sont présentées dans le tableau 7.1.

8.3.1 Performances globales

Dans cette expérience, nous traitons 1000 images de même taille 512x512 avec les deux implémentations sur des configurations matérielles différentes. Pour l'implémentation SignalPU nous déplions le graphe avec un degré J=10. La distribution MPI est effectuée de façon à équilibrer les charges entre les nœuds. Dans l'implémentation PACCO, le placement manuel est obtenu après une exploration exhaustive afin d'équilibrer au mieux les charges. L'objectif de cette expérience est de mesurer la performance obtenues pour chaque implémentation, son passage à l'échelle et sa capacité à exploiter les différents niveaux de parallélisme.

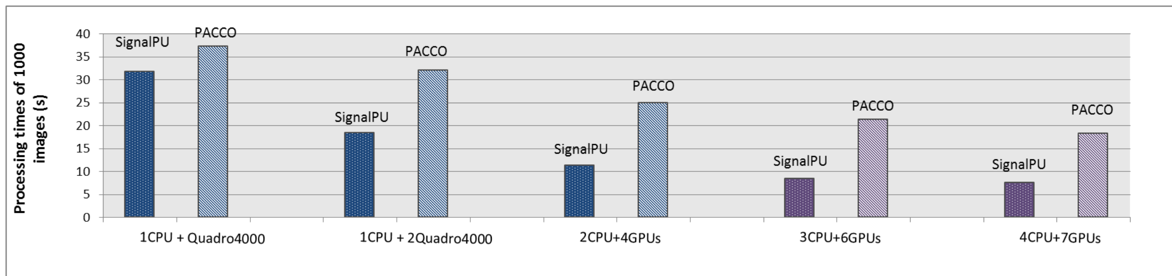

FIGURE 8.4 – Comparaison des résultats des performances globales SignalPU Vs PACCO

La figure 8.4 présente les temps de calcul globaux en secondes des deux implémentations pour 3 configurations différentes. Dans la première configuration incluant 1 cœur CPU+ 1 Quadro 4000, nous obtenons pour SignalPU un temps de calcul d'environ 32 secondes et seulement 39 secondes pour l'implémentation PACCO, soit une accélération de 1.2x en faveur de SignalPU. Cela est du à la synchronisation des tâches. En effet, dans le modèle d'exécution de l'environnement PACCO, les tâches au sein d'une itération sont synchronisées par barrière globale. Les tâches de la prochaine itération ne peuvent être traitées qu'après que l'itération courante soit terminée. Cela signifie que les éléments de calcul les moins performants ou ayant une charge de travail supérieure retardent les autres éléments et rallongent l'itération. Cependant, avec SignalPU, nous utilisons seulement des dépendances de données pour synchroniser les tâches. Les tâches de l'itération suivante au sein du pipline sont traitées dès que leur données sont disponibles.

Dans la configuration avec 1 cœur CPU+ 2 Quadro 4000, les performances sont améliorées dans les deux implémentations. Le temps de calcul de SignalPU est réduit à 19 secondes, ce qui correspond à une accélération de 1.7x. Cependant, avec PACCO le temps de calcul est de 31 secondes, ce qui correspond à une accélération de 1.2x. Nous expliquons cette différence de résultats par l'ordonnancement "scheduling" des tâches. En effet, l'implémentation PACCO comporte un placement statique des tâches alors que le graphe de l'application comporte des acteurs ayant des temps de calcul disparates avec une faible granularité (8 nœuds) relativement à celle de l'architecture (1CPU+2GPU). Cela fait qu'une répartition des charges au sein de l'itération n'est pas atteinte. Par conséquent, l'itération est retardée par les éléments de calcul les plus chargés et le passage à l'échelle des performances est réduit. Contrairement, avec l'implémentation SignalPU, l'ordonnancement dynamique des tâches du pipline accompagné d'un dépliement d'un facteur J=1 permet de distribuer efficacement le calcul entre les 2 GPU.

Ce point est discuté plus amplement dans l'expérience suivante.

Dans la dernière expérience effectuée sur plusieurs nœuds : Archi1+Archi2+Archi3+Archi4 et, les performances (4CPUs+7GPUs) des deux implémentations sont améliorées. Nous obtenons sur 2 nœuds (2CPUs+4GPUs) un temps de 13 secondes pour SignalPU ce qui correspond à une accélération de 2.3x. Pour PACCO nous obtenons 25 secondes ce qui correspond à un accélération d'environ 1.6x. Pour 3 nœuds (3CPUs+6GPUs), nous obtenons pour SignalPU des performances de 9 secondes et une accélération d'environ 3.6x. Pour PACCO, nous obtenons 21 secondes et une accélération d'environs 1.85x. Pour une configuration de 4 nœuds (4CPUs+7GPUs), nous obtenons pour SignalPU des temps de calcul de 8 secondes, soit une accélération d'environ 4x. Pour PACCO, nous obtenons 19 secondes et une accélération d'environ 2.1x. Ici également, les raisons pour expliquer les différences de résultats sont la synchronisation des tâches et le "scheduling" dynamique. En effet, avec SignalPU, la répartition des charges de calcul est effectuée via la fonctionnalité de distribution $MPI_data_distribute()$ proposée par l'environnement. Cependant, dans l'environnement PACCO, l'itération est non uniformément répartie sur les éléments de calcul des deux nœuds. En outre, le traitement est synchronisé pour chaque itération ce qui retarde les autres itérations.

8.3.2 Ordonnancement statique vs "Scheduling" Dynamique

Dans cette seconde expérience, nous étudions les différences de performance entre la combinaison de l'ordonnancement dynamique avec le dépliement du graphe "unfolding" et un placement statique sans dépliement. Nous traitons avec l'implémentation SignalPU 1000 images de taille $512x512$ sur une configuration matérielle hétérogène CPU-GPU (2 cœurs CPU+ 2 Quadro 4000) avec des implémentations avec ou sans les 2 optimisations en question :

1. **Ordonnancement dynamique** : labellisée "Dynamique scheduling", c'est une implémentation sans dépliement de graphe (J=1) mais avec une distribution dynamique des tâches au sein d'une itération. Une répartition des charges est effectuée sur les acteurs de chaque niveau du graphe DFG mais non pas sur l'ensemble du graphe.
2. **Ordonnancement statique** : labellisée "Static scheduling", c'est une implémentation itérative sans dépliement de graphe (J=1) où chaque acteur est associé statiquement à un élément de calcul. Le placement le plus efficace trouvé grâce à une exploration exhaustive qui consiste à traiter tout les acteurs sur un seul GPU.
3. **Ordonnancement statique avec dépliement du graphe** : labellisée "Static scheduling with unfolding", dans cette implémentation nous déplions la boucle principale de l'application sur dix niveaux (J=10). Cependant, le placement des acteurs est statique comme dans la première implémentation.
4. **Ordonnancement dynamique avec dépliement du graphe** : labellisée "Dynamique scheduling with unfolding", dans cette implémentation nous combinons les deux optimisations. Nous déplions le graphe avec J=10 et utilisons le scheduling dynamique pour répartir les charges des acteurs sur les 10 niveaux de DAG.

Dans la figure 8.5, sont illustrés les temps de calcul obtenus pour chaque implémentation sous la forme de 4 groupes de 4 barres. Au sein d'un groupe, chaque barre correspond à un élément de calcul (2 cœurs CPU + 2 Quadro 4000). Chaque barre comporte le cumul de 3 temps : un temps d'exécution effectif appelé "Execution", un temps d'attente appelé "Sleep" et un temps de sur coût appelé "Overhead". La somme des 3 temps représente le temps global de traitement de l'application.

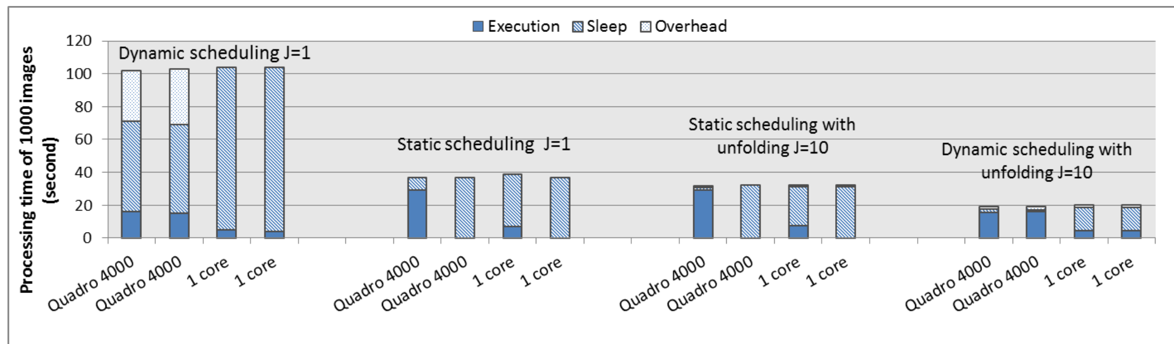

FIGURE 8.5 – Comparaison des performances Scheduling dynamique Vs Scheduling statique d'une implémentation SignalPU

Le premier groupe de barres à gauche du graphique représente les temps d'exécution de l'implémentation (Ordonnancement dynamique) sur l'architecture. Cette implémentation n'est pas efficace pour les raisons suivantes : premièrement, les temps d'attente labellisés "Sleep" sont élevés. Cela est du à l'aspect itératif de l'application. En effet, les éléments de calcul les plus performants ou ayant moins de tâches à traiter doivent attendre les plus longs avant d'entamer une nouvelle itération. Deuxièmement, les temps de surcoût labellisés "Overhead" sont importants. En effet, l'execution étant sans dépliement J=1, des allocations et libérations sur charge le coût des itérations ce qui augmente les temps de calcul.

Dans le deuxième groupe de 4 barres de la figure 8.5, nous montrons les temps de calcul de l'implémentation avec ordonnancement statique. Les performances de cette implémentation sont meilleures que celles de l'implémentation précédente. En effet, grâce à la distribution statique des acteurs sur un seul GPU les communications entre les éléments de calcul sont nulles ce qui réduit les temps d'attente "Sleep". En outre, le placement statique réduit les temps de surcoût "Overhead" du support exécutif.

Dans le troisième groupe de 4 barres, nous montrons les temps de calcul de l'implémentation ordonnancement statique avec dépliement du graphe. Les performances sont améliorées par rapport aux précédentes implémentations. Cela est du au fait que l'occupation de l'élément de calcul (GPU) est plus importante puisque les temps d'attente (Sleep) sont encore plus faibles. En effet, grâce au dépliement du graphe (J=10), le GPU sollicité n'attend pas l'execution de la dernière tâches d'un niveau de graphe pour commencer l'exécution d'une autre itération. Cependant, les temps de traitement labellisés "Execution" restent inégaux entre les 2 GPUs à cause de la distribution statique des acteurs.

Le dernier groupe de 4 barres représente le temps de calcul de l'implémentation avec ordonnancement dynamique et dépliement du graphe. Ici les performances obtenues sont meilleures que dans les autres implémentations. Les temps d'exécution sont améliorés et pratiquement égaux sur les deux GPUs et les temps d'attente sont faibles. Cela est obtenu grâce à une combinaison de la distribution dynamique des acteurs avec un dépliement du graphe. En effet, ce dernier permet de fournir plus de tâches à l'ordonnanceur (Scheduler) afin de mieux répartir les charges entre les éléments de calcul. Par conséquent, comme proposé dans notre MdPP, il est intéressant et efficace de combiner la distribution dynamique avec un dépliement de graphe afin d'augmenter les performances.

8.3.3 Surcoûts introduits par PACCO et SignalPU

Dans cette expérience nous nous intéressons à l'évaluation du surcoût introduit par l'utilisation de PACCO et SignalPU. Pour cela, nous traitons avec les deux implémentations un nombre d'images croissant et mesurons l'évolution des surcoûts engendrés par rapport au temps de calcul utile. Dans la figure 8.6, nous montrons l'évolution en pourcentage du temps moyens du surcoût relativement au temps global de calcul utile. L'évolution de ce pourcentage se stabilise vers 7% pour SignalPU et seulement 2% pour PACCO. Cette différence s'explique par le fait que PACCO est un support exécutif statique qui ne comporte pas d'ordonnanceur en ligne. En revanche, SignalPU est basé sur un support exécutif dynamique permettant une distribution dynamique des tâches. Cela nécessite, entre autres, de réaliser des allocations de différentes tailles sur les mémoires des éléments de calcul ciblés. Cependant, il est à noter que ce temps "Perdu" par SignalPU reste inférieur au gain apporté par la distribution dynamique car les éléments de calcul sont mieux exploités.

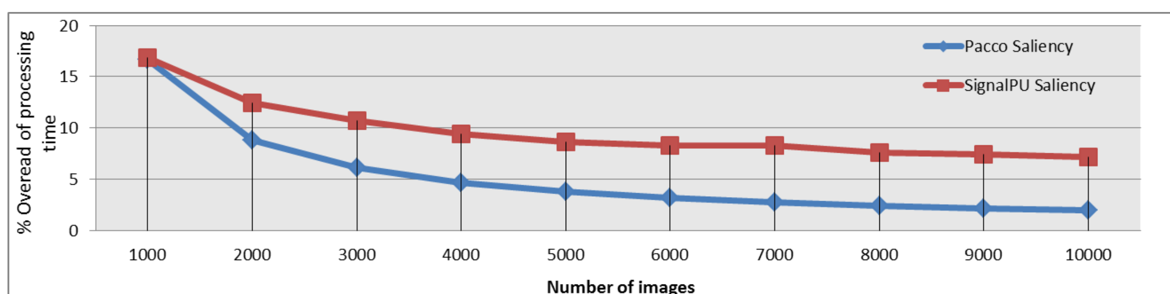

FIGURE 8.6 – Évolution des surcoûts dans les implémentations SignalPU et PACCO

8.3.4 Apport de la persistance des initialisations et de l'auto-tuning

La dernière expérience que nous proposons vise à évaluer le gain en performance offert des tâches produite par l'utilisation des fonctionnalités de persistance des initialisations et de l'auto-tuning dans l'implémentation SignalPU. La fonctionnalité de persistance des initialisations permet l'exécution de la partie d'initialisation d'un acteur qu'une seule fois sur chaque élément de calcul. Tandis que la fonctionnalité d'auto-tuning permet de trouver les paramètres

de Threads par blocs des "Kernel" GPUs par exploration à l'exécution. Dans cette expérience nous mesurons le temps de calcul de l'acteur *Interaction*() sur deux GPU différents : GTX 780 et Quadro 4000. Les résultats sont donnés dans la figure 8.7. Premièrement, nous notons sur la première itération, l'execution de la partie d'initialisation représentée en ligne bleu hachée. Cette partie est exécutée une seule fois pour chaque GPU. Deuxièmement, les temps de calcul des kernels varient avec un facteur supérieur à 2 selon le paramétrage des Kernels. En outre, l'exploration converge vers un paramétrage différents pour chaque GPU. En effet, nous obtenons respectivement les paramètres (32,8,1) et (32,16,1) pour la GTX 780 et la Quadro 4000 correspondant aux temps 0.4 ms et 2 ms. Par conséquent, grâce aux deux fonctionnalités de persistance de l'initialisation et d'auto-tuning, nous réduisons les temps de calcul et augmentons les performances.

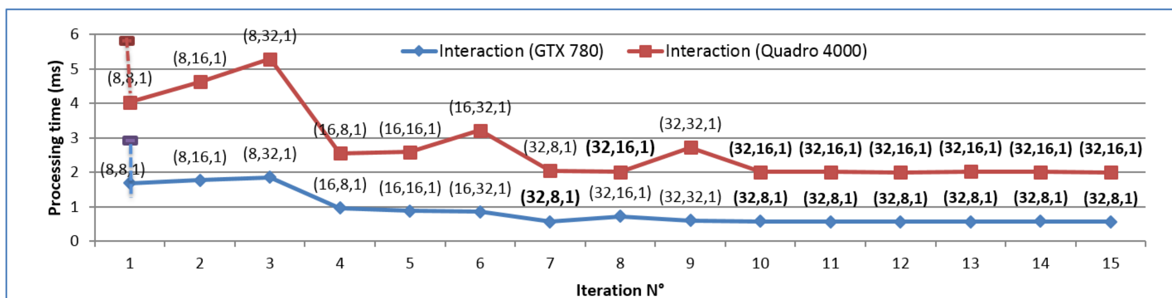

FIGURE 8.7 – Évolution des performances des tâches dans une implémentation SignalPU

8.4 Conclusion

Dans ce chapitre, nous avons validé et comparé les deux modèles SignalPU et PACCO sur une application de traitement d'image du monde réel, l'application de calcul de cartes de saillance visuelle. Nous avons présenté en premier l'implémentation de cette application avec les deux modèles. Ces deux implémentations bénéficient d'un haut niveau d'abstraction et nécessitent un effort de programmation réduit. Cependant, l'implémentation SignalPU bénéficie d'une abstraction supplémentaire concernant le placement des acteurs sur l'architecture. En effet, dans l'implémentation PACCO, le programmeur doit placer manuellement et statiquement les acteurs sur les éléments de calcul, alors que SignalPU offre un placement dynamique. Nous avons présenté par la suite différentes expériences visant à valider et comparer les performances des deux modèles. Nous avons montré que SignalPU permet d'atteindre de meilleures performances que PACCO. En effet, grâce à son modèle d'exécution basé sur la distribution dynamique des tâches combiné à un dépliement du graphe (unfolding) et à la distribution MPI, les éléments de calcul sont utilisés plus efficacement. En outre, dans l'environnement SignalPU, des fonctionnalités comme la réutilisation de buffer et le pipelining de tâches permettent de réduire les surcoûts. Dans l'environnement PACCO l'exécution est basée sur un modèle BSP itératif avec placement statique. Ce modèle engendre des surcoûts limités par rapport à SignalPU. Cependant, il peut générer des temps d'attente importants dans les éléments de calcul les moins sollicités de l'itération. Finalement, dans SignalPU, les fonctionnalités de

persistance des initialisations et d'auto-tuning augmentent les performances au niveau de la tâche et permettent d'obtenir des temps d'exécution effectifs les plus réduits possible.

Chapitre 9

L'application de suivi

9.1 Introduction

Nous proposons de valider et de comparer, dans ce chapitre, les modèles PACCO et SignalPU sur l'implémentation d'une application du monde réel de traitement de vidéo pour le suivi de personnes. Pour cela nous présentons en premier dans la section 9.2 cette application et décrivons son implémentation sur cluster hétérogène avec chaque modèle : PACCO et SignalPU. Ensuite, nous présentons dans la section 9.3, différentes expérimentations et résultats obtenus avec les deux implémentations que nous analysons et comparons en vue de valider les deux modèles.

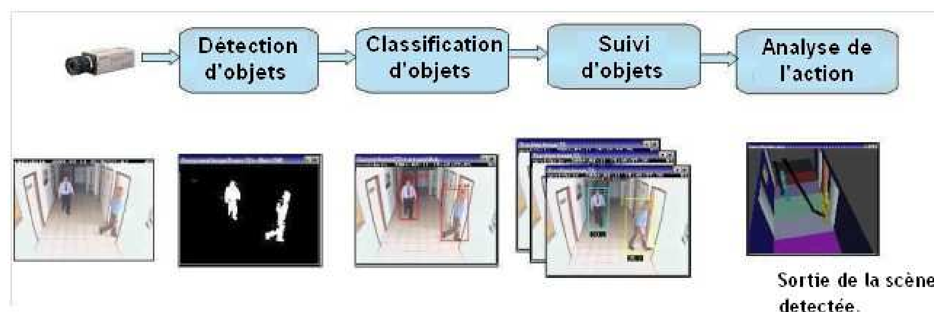

Figure 9.1 – Illustration des étapes de traitement d'une vidéo pour le suivi de personne

9.2 Application de suivi et implémentations

L'application de suivi de personne est une application classique du domaine du traitement de la vidéo. C'est une étape préliminaire pour des applications d'identification et d'analyse du comportement des mobiles dans le domaine de la vision par ordinateur. Ces applications sont fortement utilisées depuis l'evolution des techniques de capture des images en haute résolution dans l'industrie, avec diverses perspectives, par exemple le suivi de personnes dans un magasin à des fins de marketing. Cela motive leur accélération sur des nœuds de calcul parallèles et hétérogènes tant le volume de données capturées croit et que les algorithmes utilisés sont de plus en plus complexes et coûteux. En effet, une application comme celle de l'identification des trajectoires des clients dans une grande surface à partir des images

d'un réseau de caméras, nécessite un suivi continu des personnes sur une quantité importante d'images à grande résolution. L'implémentation de ces applications sur cluster hétérogène offre des perspectives d'accélération tout particulièrement intéressantes.

Les étapes de calcul de l'application de suivi choisies pour notre étude sont décrites dans la figue 9.1. Une vidéo est traitée successivement par plusieurs blocs de différentes complexité dans une boucle principale. Chaque itération traite une seule image pour produire les contours et les positions successives des personnes. Une description plus détaillée de l'application est donnée à travers l'algorithme 6. Premièrement, les images successives extraites de la video d'entrée sont traitées par une fonction d'extraction du fond $Foreground_extraction$. Cette fonction est basée sur une approche de modélisation statistique de l'image avec des techniques de mixture de gaussiennes. Le résultat de cette fonction est une image binaire où les pixels blancs représentent les pixels en mouvement. A l'image binaire sont appliqués ensuite un filtre d'érosion $Erosion$ et un filtre de $dilatation$ pour favoriser le regroupement des pixels d'un objet de la scène. La fonction suivante, $Connected_Component_labeling$, se base sur un algorithme d'étiquetage des pixels connectés pour extraire les blobs représentants les objets en mouvement de la scène. Dans la dernière fonction, $Tracking$, les blobs sont caractérisés par des identificateurs géométrique et de texture, ils sont ensuite suivis à travers les images successives en se basant sur une approche de couplage dans un graphe biparti orienté.

Algorithm 6 Application de Suivi

Input : A video r_vid of size $w \cdot l$
Output : The tracking of each person
1: **for** $r_im \leftarrow$ image number i of r_vid **do**
2: $r_fg \leftarrow Foreground_extraction(r_im)$
3: $e_fg \leftarrow Erosion(r_fg)$
4: $d_fg \leftarrow Dilatation(e_fg)$
5: $ccl_fg \leftarrow Connected_Component_labeling(d_fg)$
6: $track_fg \leftarrow Tracking(d_fg, r_im)$
7: **end for**

Dans ce qui suit nous présentons les implémentations avec PACCO et SignalPU de cette application. Cependant, nous nous fixons deux contraintes d'exécution : nous souhaitons, d'une part garantir un débit stable pour les images de suivi en sortie afin de réduire la taille des tampons d'affichage. D'autre part, l'accès aux images de la vidéo d'entrée est ordonné i.e. la lecture des images se fait successivement image par image.

9.2.1 implémentation PACCO

Pour implémenter l'application de suivi avec l'environnement PACCO, il faut premièrement la modéliser avec un DAG à taux unique. On représente les fonctions de l'algorithme (Capture, FG, Erode, Dilate, CCL, Tracking, Display) avec des nœuds et les données échangées avec des arcs. Deuxièmement, il faut représenter l'architecture d'exécution avec un graphe orienté où les nœuds représentent les éléments de calcul et les arcs les supports de communication

les reliant. Les fonctions sont ensuite associées statiquement aux éléments de calcul. Les différentes fonctions sont encapsulées sous forme de classes. Chaque classe comporte des attributs représentant les données internes, et trois méthodes : une méthode d'initilisation, une méthode d'execution, et une méthode d'attente. La classe de chaque fonction est incluse dans le code principal de l'application. Par exemple, la classe de la fonction $Foreground_extraction$ comporte un état interne : l'arriére plan. Ces données sont mises à jour à chaque itération et sont sauvegardées dans la mémoire associée à l'element de calcul de placement. Dans cette implémentation, il faut également définir les structure de données utilisées dans la description des arcs. En effet, le type *matrix* correspond à une image en niveau de gris ou binaire de taille égale à l'image d'entrée. Le type *blobs* correspond à un vecteur de blobs de taille fixe représentant les pixels des objets mobiles dans la scène.

Comme déjà présenté dans le chapitre 4, à l'exécution, le graphe de l'application et le graphe d'architecture sont analysés pour construire le graphe d'implémentation. Des buffers sont introduit et les allocations nécessaires effectuées. L'exécution se déroule itérativement selon le modèle BSP. Chaque élément de calcul communique ses données et exécute les fonctions qui lui sont associées en faisant ainsi progresser les résultats à travers le pipeline de tâche tout au long des itérations. A noter que le modèle d'execution de PACCO permet de respecter les contraintes de stabilité de débit et de lecture ordonnée des images.

9.2.2 Implémentation SignalPU

Dans l'implémentation SignalPU, l'algorithme est modélisé cette fois avec un DFG synchrone à taux unique. Les fonctions de l'algorithme sont toutes représentées avec un nœud dans le graphe. Les dépendances de données sont représentées par des arcs reliant les acteurs. Les arcs contiennent des informations comme le type et la taille des données. L'acteur $Foreground_extraction$ comporte dans cette description un arc avec auto-dépendance permettant de mettre à jour l'arriére plan pour chaque image en fonction de l'image courante. Les fonctions sont représentées avec deux sous-fonctions : "Init" pour initialiser les données statiques et "exec" pour traiter les données changeantes. Pour respecter les contraintes additionnelles sur la stabilité du débit et la lecture ordonnée des images, nous avons rajouté des dépendances d'exécution dans le graphe. L'itération i d'un acteur ne peut s'exécuter jusqu'à ce que son itération précédente $(i-1)$ soit terminée. En outre, pour respecter une stabilité du débit de l'exécution, le placement des acteurs sur les éléments de calcul doit être statique. Par conséquent, la répartition des calculs sur les nœuds MPI exploite le modèle Pipeline et non pas SIMD.

A l'exécution, le graphe d'application DFG synchrone est analysé et déplié selon le degré spécifié par l'utilisateur. Ce dépliement permet aussi de transformer les dépendances inter-itérations incluant les auto-dépendances pour produire un DAG de tâches. Dans ce cas d'implémentation, les deux modèles PACCO et SignalPU offrent une abstraction similaire. En effet, la description de l'application est similaire et dans les deux cas le placement est explicite. Cependant, le modèle d'exécution reste différent. Dans la section suivante, nous présentons les résultats obtenus par exécution des deux implémentations sur un cluster hétérogène.

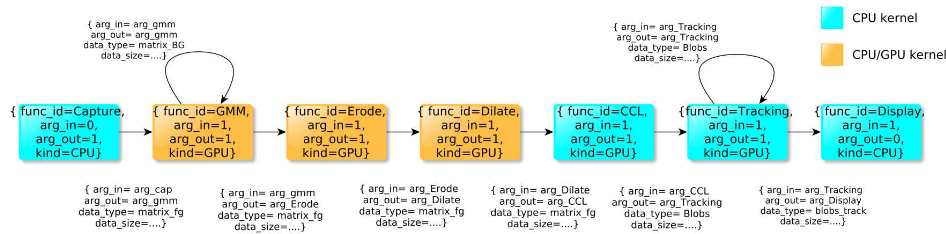

FIGURE 9.2 – XML-DFG modélisant l'application de tracking dans SignalPU. Seules les dépendances de données sont présentes

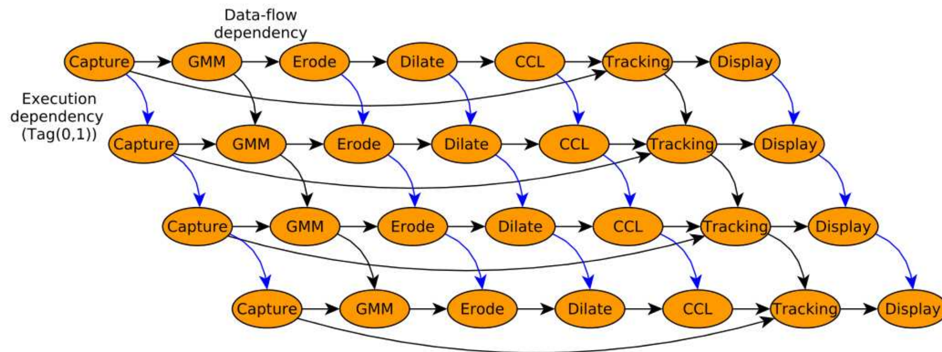

FIGURE 9.3 – Dépliement du graphe (J=4) et ajout des dépendances d'exécution additionnelles (arcs bleu) sur le graphe d'application de l'application de suivi

9.3 Expérimentations et résultats

Dans cette section nous nous intéressons à la validation des performances des deux modèles SignalPU et PACCO sur deux expérimentations : une expérimentation visant à mesurer les performances globales de chaque implémentation et à les comparer et une deuxième expérimentation restreinte à SignalPU, visant à mesurer l'impact du dépliement de graphe sur les performances.

9.3.1 Performances globales

Nous allons tout d'abord procéder à une comparaison des performances globales des deux implémentations SignalPU et PACCO sur différentes architectures. A travers ces mesures, il est possible de déterminer l'aptitude de chaque modèle à exploiter les différents niveaux de parallélisme. Dans la figure 9.4, nous présentons le débit en image par seconde obtenu avec les deux implémentations sur des configurations CPU+GPU différentes.

Pour la configuration à 1 cœur CPU, l'implémentation PACCO produit un débit légèrement supérieur à celui produit par l'implémentation SignalPU. Cela est du à son modèle d'exécution statique. En effet, même si le placement est statique dans les deux implémentations, SignalPU engendre des surcoûts plus importants que le support exécutif PACCO.

Dans les exécutions sur 2 et 3 Cœurs CPU, les performances sont améliorées pour les deux implémentations grâce à l'exploitation du parallélisme de graphe (pipeline). En effet, SignalPU et PACCO produisent respectivement des accélérations d'environ 1.8x et 2.7x pour le premier et 1.2x et 1.6x pour le second. Nous expliquons ces différences par la différence de méthode de synchronisation dans les deux modèles d'exécution. En effet, dans PACCO, les itérations successives sont synchronisées par une barrière globale. Un élément de calcul ne peut entamer le traitement d'une tâche de l'itération BSP suivante que lorsque toutes les tâches de l'itération BSP précédente sont terminées. Or, avec SignalPU et grâce à la fonctionnalité de dépliement de graphe, la synchronisation est assurée par les dépendances de données. Les tâches d'itérations successives sont exécutées dans l'ordre dès que leurs données d'entrée sont disponibles.

Dans la configuration 1 cœur CPU+ 1 GPU, les performances sont améliorées dans les deux implémentations par rapport à une exécution avec 1 seul cœur CPU. En effet, une exploitation sur le GPU du parallélisme de données des acteurs : GMM, Erode, Dilate permet de réduire le temps de traitement d'une image. Nous obtenons avec SignalPU une accélération d'environ 2.4x comparée à une exécution sur 1 cœur CPU. Cependant, avec PACCO, l'accélération obtenue est de seulement 1.9x. Pour expliquer cette différence, nous nous basons sur les mêmes raisons citées précédemment : la synchronisation des itérations successives par barrière globale dans PACCO limite le taux d'occupation des éléments de calcul. A noter que dans les deux implémentations, cette performance est inférieure à la performance obtenue par l'utilisation de 3 cœurs CPU. Cela est du au fait que les acteurs CPUs représentent la charge principale de l'application.

Dans les configurations avec 2 cœurs CPU+ 1GPU, et 3 cœurs CPU+1GPU les performances sont améliorées dans les deux implémentions. En effet, nous obtenons en comparaison à l'exécution 1 Cœur CPU des accélérations respectivement d'environ 3.3x et 4.2x pour SignalPU et seulement 2.0x et 2.2x pour PACCO. Également ici PACCO montre une scalabilité moindre que celle produite avec SignalPU. Encore une fois, cela s'explique par son modèle d'exécution BSP à synchronisation globale. Par conséquent SignalPU exploite mieux le parallélisme de graphe et de données dans un contexte d'optimisation du débit d'une application de traitement d'image.

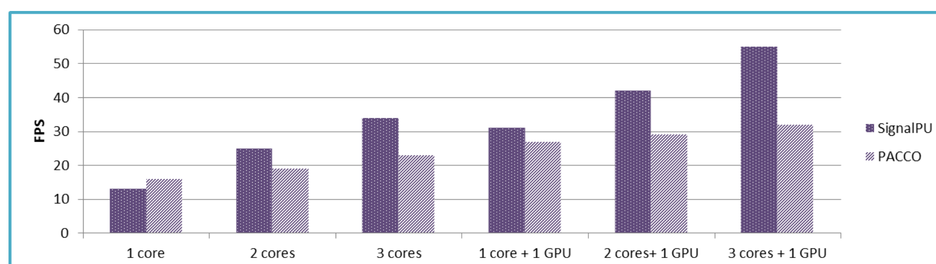

FIGURE 9.4 – Performances globales de SignalPU vs PACCO sur différentes architectures

9.3.2 Performances et dépliement du graphe

Dans cette expérimentation, nous nous intéressons à la validation de la fonctionnalité de dépliement du graphe dans l'approche SignalPU utilisée avec un contexte de placement statique. En effet, dans l'application traitée dans ce chapitre, le placement des acteurs est statique sur les éléments de calcul. Cependant, nous souhaitons évaluer l'impact du dépliement du graphe sur le débit. Pour cela, nous exécutons l'implémentation SignalPU de l'application de suivi avec différents degrés de dépliement "J" sur une architecture composée de 4 cœurs CPU. Nous mesurons pour chaque exécution le débit et le taux moyen de non utilisation dans les éléments de calcul par rapport au temps global de traitement. Dans la figure 9.5, nous présentons ces mesures que nous discutons comme dans la suite :

Pour le degré de dépliement $J = 1$, l'exécution présente les moins bonnes performances. En effet, le FPS atteint seulement 14 images par seconde. En outre, le taux d'inoccupation moyen dans les 4 cœurs utilisés est élevé et atteint 72%. Nous expliquons cela par une synchronisation forte entre les itérations successives. En effet, avec un dégrée de dépliement "J=1", pour être exécutées, les tâches d'une itération doivent attendre que toutes les tâches de l'itération précédente soient terminées.

Pour les dégrées de dépliement $J = 2, J = 3, et J > 4$, les exécutions respectives produisent des améliorations sur les performances. En effet, même si le placement des acteurs est statique, les résultats obtenus montrent que les éléments de calcul sont mieux occupés, les taux de non utilisation sont respectivement de $(65\%, 54\%, 50\%)$ ce qui produit un meilleur FPS $(28, 34, 36)$. En effet, grâce à la synchronisation par dépendances de données entre les tâches des itérations successives, chaque élément de calcul peut exécuter autant de tâches dés que leurs données d'entrée sont disponibles. Par conséquent l'exécution de l'application est seulement limitées par ses dépendances de données. A noter qu'au delà d'un certain seuil de dépliement du graphe, dans notre cas $J > 4$, l'impact sur les performances est nul. Cela est du aux dépendances de données et d'ordre d'exécution dans l'application, et au placement statique.

Unfolding	1	2	3	>4
FPS	14	28	34	36
% Sleep	0,74	0,64	0,54	0,5

Figure 9.5 – Évolution du FPS et du temps de non utilisation en faisant varier le degré de dépliement du graphe, "J", dans SignalPU. L'architecture utilisée est composée de 4 Cœurs CPUs

9.4 Conclusion

Dans ce chapitre, nous avons validé l'utilisation des environnements PACCO et SignalPU sur l'application de suivi des personnes dans une vidéo. Cette application a été implémenté

avec ces deux modèles dans un contexte "streaming". Nous avons en premier, dans la section 9.2, présenté l'application de suivi et son implémentation avec PACCO et SignalPU en prenant en compte ses contraintes d'exécution. Ces deux implémentations présentent un niveau d'abstraction équivalent, avec un léger avantage pour SignalPU qui évite au programmeur de représenter l'architecture matérielle. Ensuite, dans la section 9.3, nous avons présenté deux expérimentations pour comparer et valider les performances des deux modèles. Les résultats obtenus montrent que SignalPU exploite plus efficacement le parallélisme de graphe et de données dans ce contexte "streaming" grâce notamment à la fonctionnalité de dépliement de graphe.

Conclusion et perspectives

9.5 Conclusion

Dans cette thèse, le premier chapitre (chapitre 1) nous a permis de présenter l'évolution des machines de calcul vers des topologies parallèles et hétérogènes à l'image des grilles de calcul. Ces grilles sont en effet capables de produire de hautes performances grâce à leur architectures parallèles incluant des composants hétérogènes. Nous avons présenté quelques un de ces composants (CPU multi-coeurs, GPU, Xeon Phi, FPGA) et les optimisations à exploiter dans chacune d'entre elles.

Dans le chapitre 2, nous avons introduit les modèles de programmation permettant de faciliter l'implémentation sur ces architectures. Nous avons cité différents environnements d'implémentation illustrant les MdPP en les évaluant selon plusieurs critères, comme le niveau d'abstraction qu'ils offrent, la flexibilité et la finesse de contrôle de l'architecture, ou la couverture possible de différentes architectures.

Toutefois nous avons vu qu'il est compliqué pour un modèle avec un haut niveau d'abstraction d'offrir un grand contrôle sur l'architecture et inversement. Pour répondre à cette problématique, nous avons proposé l'idée de spécifier les applications avec un modèle de programmation spécifique à un domaine applicatif. En effet, en se basant sur les caractéristiques communes d'un type d'application donné, il est possible de définir un modèle d'abstraction leur correspondant et ainsi de concilier les objectifs de productivité et d'efficacité.

Pour mettre cette idée en pratique, nous nous sommes intéressés aux applications de type DSP que nous avons présentées dans le chapitre 3. Nous avons discuté leurs caractéristiques et les modèles permettant de les décrire, à savoir le modèle de calcul à réseaux de processus ou plus précisément le modèle graphe de flot de données DFG. Dans ce même chapitre nous avons présenté quelques environnements d'implémentation et de simulation spécifiques aux applications DSP sur architectures parallèles et hétérogènes. Ensuite, nous avons abordé l'implémentation de ces applications avec différents environnements existants à travers un état de l'art général. Nous avons étudié dans cet état de l'art les avantages et inconvénients de chaque type d'environnement et avons situé notre contribution que nous avons présentées dans les parties suivantes.

Ces trois chapitres ont constitué la partie I qui introduit le contexte et l'objectif de la thèse.

Dans la deuxième partie (partie II), développée en deux chapitres, nous avons entamé la première contribution de ce travail de recherche. Le chapitre 4 à été consacré aux les modèles de programmation BSP. Nous avons présenté quelques environnements basés sur ce modèle. Nous avons ensuite présenté PACCO, un modèle de programmation développé dans notre équipe pour implémenter des applications de type DSP sur un cluster hétérogène. Le support

exécutif de PACCO est basé sur le modèle BSP itératif avec pipeline. Nous avons relevé ses avantages et inconvénients. En effet, PACCO est un environnement statique. Or, pour être performant, son modèle d'exécution nécessite une répartition de charge équilibrée au sein de la super étape. Pour remédier à cet inconvénient et pour enrichir l'environnement PACCO, nous avons proposé une fonctionnalité de migration de tâches en cherchant à réduire l'impact de la migration sur le déroulement de l'execution. Cette approche a été présentée dans le chapitre 5. L'idée que nous avons retenue est de répartir les actions de cette migration sur plusieurs itérations. En effet, cela permet de ne pas arrêter le calcul pendant la migration réduisant ainsi son impact. Nous avons ensuite présenté son implémentation dans l'environnement PACCO. Finalement, nous avons validé notre approche sur une application de traitement d'image du monde réel en se fixant différents scénarios : un contexte de répartition des charges pour augmenter le débit, un scénario de réduction de la consommation d'énergie, et un scénario de libération d'un élément de calcul.

La troisième partie (partie III) est consacrée à notre deuxième contribution. Elle est également articulée autour de deux chapitres : le chapitre 6 a été consacré en premier à l'introduction de l'environnement StarPU. Nous avons présenté ses composants et montré son utilisation à travers un exemple d'implémentation d'une application de type DSP synthétique. Ensuite, nous avons présenté notre contribution principale, SignalPU, un modèle de programmation basé sur StarPU et spécifique à l'implémentation des applications de type DSP sur cluster hétérogène. L'utilisateur modélise son application sous forme de DFG synchrone à taux unique en décrivant le graphe de l'application. Nous avons ensuite montré son intérêt en comparant une implémentation SignalPU à l'implémentation StarPU. En effet, avec SignalPU, l'utilisateur n'a pas besoin de manipuler la librairie C de StarPU pour décomposer son application. En outre, il bénéficie de plusieurs fonctionnalités spécifiques au domaine applicatif visé avec une meilleur abstraction : déploiement dynamique du graphe, réutilisation automatique des buffers, pipelining des tâches, persistance des initialisations et l'auto-tuning des kernels GPU. Dans le chapitre 7 nous avons validé SignalPU à travers la comparaison de l'implémentation d'une application synthétique avec une implémentation basée sur les environnements MPI+OpenMP+CUDA. Nous avons présenté en premier une bibliothèque d'opérateurs de charge. Ensuite, nous avons construit une application synthétique ayant une structure représentative d'une application de type DSP. Nous avons comparé l'implémentation SignalPU de cette application à son implémentation MPI+OpenMP+CUDA en terme d'expression et d'abstraction. En terme de performance, nous avons comparé les résultats de l'exécution de ces deux implémentations sur cluster composé de deux nœuds MPI. SignalPU permet de produire de meilleures performances, notamment grâce à son modèle d'execution basé sur une distribution dynamique et efficace du DAG de tâches sur l'architecture.

Dans la partie IV, nous avons évalué les deux modèles PACCO et SignalPU sur l'implémentation des applications du monde réel. Dans le chapitre 8, nous avons présenté une validation à travers la comparaison de résultats des implémentations PACCO et SignalPU d'une application de calcul de cartes de saillance visuelle sur un cluster hétérogène CPU-GPU. L'objet est ici de traiter un grand nombre d'images. En terme d'expressivité, les deux modèles ont offert un haut niveau d'abstraction. En effet, pour implémenter cette application de Saillance, il a suffit de décrire son algorithme avec un graphe de flot de données, introduire les fonctions

dans le code et décrire les types et tailles des structures de données échangées entre les acteurs. Cependant, nous avons vu que SignalPU permet d'abstraire le placement des tâches. En effet, l'implémentation PACCO a nécessité la description de l'architecture de calcul avec un graphe orienté et un placement statique et manuel de chaque acteur. En terme de performance, l'environnement SignalPU a présenté de meilleurs résultats comparé à PACCO, notamment grâce à la combinaison de la fonctionnalité de dépliement avec l'ordonnancement dynamique permettant ainsi d'exploiter différents niveaux de parallélisme : tâche, donnée, graphe. En outre, des expériences sur la mesure surcoût ont montré une stabilisation à 7% du temps global de traitement grâce aux fonctionnalités de réutilisation de buffer et de pipelining de tâche. Finalement, les fonctionnalités de persistance des initialisations et de l'auto-tunning des "kernels" ont montré une réduction des temps de calcul des tâches. Dans le chapitre 9, nous avons également présenté des résultats de comparaison entre les implémentations PACCO et SignalPU de l'application de suivi de mobiles dans une vidéo. Le contexte de cette application est de maximiser le débit de traitement en respectant des contraintes de stabilité du débit et de production ordonnée des images résultat. En terme d'abstraction, les deux modèles ont permis un haut niveau d'abstraction dans la décomposition de l'algorithme et la gestion des communications et synchronisations. Cependant, pour respecter la contrainte de stabilité de débit, le placement des tâches a été effectué manuellement et statiquement dans les deux implémentations. En outre, des dépendances d'execution entre les tâche ont été rajoutées dans l'implémentation SignalPU pour respecter une exécution ordonnée et pour obtenir un débit stable. En terme de performance, SignalPU a montré de meilleurs résultats de passage à l'échelle que PACCO. En effet, grâce à la fonctionnalité de dépliement du graphe, et la synchronisation selon les dépendances de données, l'occupation des éléments de calcul a été plus efficace.

Ces différentes contributions ont été publiées dans plusieurs conférences internationales et un journal avec comité de lecture international. La publication [1] porte sur notre première contribution concernant la migration de tâches dans l'environnement PACCO. Les publications [2,3,4] portent sur notre deuxième contribution concernant l'environnement SignalPU et ses fonctionnalités.

9.6 Perspectives

Le travail effectué durant ces travaux de thèse étant loin d'être terminé, plusieurs perspectives de recherche et d'amélioration restent à creuser.

Concernant notre première contribution sur la migration des tâches dans l'environnement PACCO. Il est intéressant de développer les mécanismes de déclenchement de la migration afin de pouvoir l'exploiter plus largement : celui-ci pourrait avoir comme objectif une meilleure répartition des charges, la tolérance aux fautes ou une réduction de la consommation d'énergie. Une deuxième perspective serait de développer l'approche pour une migration de plusieurs tâches en parallèle. En effet, cela est possible dans certaines conditions qu'il faudrait étudier pour distinguer les cas de figures selon les dépendances entre les tâches à migrer. Une troisième

perspective pour l'environnement PACCO est d'ajouter à son support exécutif une fonctionnalité de gestion des dépendances de données avec des files FIFO. Cette amélioration pourrait augmenter ses performances dans le sens où les calculs seront synchronisés seulement sur les disponibilités des données. La quatrième perspective que nous proposons concernant l'environnement PACCO est d'y introduire les différentes fonctionnalités de SignalPU pouvant améliorer son niveau d'abstraction et ses performances. Nous citons, par exemple, la fonctionnalité de dépliement automatique du graphe (unfolding) qui permettrait d'éviter que l'utilisateur le fasse manuellement. La fonctionnalité d'auto-tuning qui permettrait d'optimiser les Kernels GPUs automatiquement selon l'élément de calcul de placement. Finalement, il serait intéressant pour l'utilisateur de l'environnement PACCO de ne pas avoir à représenter manuellement l'architecture ciblée. Cela le doterait d'un niveau d'abstraction supérieur et d'une meilleure portabilité. Pour mettre en pratique cette perspective, il est possible de se baser sur l'environnement HWLOC [Bro+10] permettant de représenter finement la topologie matérielle des clusters.

Concernant notre deuxième contribution SignalPU, une première perspective d'amélioration consisterait à étendre le modèle de calcul du DFG synchrone à taux unique vers d'autres modèles dynamiques et Turing-complet comme le DFG booléen. Cela permettrait d'exprimer plus d'algorithmes avec SignalPU. La deuxième perspective concernant ce modèle serait de rajouter une politique d'ordonnancement spécifique aux contraintes d'implémentation des applications de type DSP : contrainte de cadence ou de latence, contrainte d'ordre d'exécution, etc. Cela permettrait de soulager le modèle de conception des dépendances d'exécutions additionnelles ajoutées pour garantir ces contraintes. La troisième perspective concernant l'environnement SignalPU serait d'étudier les implémentations d'applications de type DSP avec des temps d'exécution variant dans le temps (calculs data dépendant) et d'évaluer dans ce contexte des estimateurs basés sur la connaissance du comportement de l'application. Cela pourrait aider l'ordonnancement à mieux équilibrer les charges en prédisant plus efficacement leurs variations dans le temps. Une quatrième perspective pourrait porter sur l'approfondissement des travaux sur la distribution des calculs sur les nœuds MPI. En effet, en l'état actuel, l'utilisateur de SignalPU est chargé de distribuer statiquement les calculs en précisant pour chaque nœud les itérations à exécuter via la fonction $MPI_data_schedule()$. Or, il serait intéressant d'abstraire d'avantage cette fonctionnalité en permettant une affectation enligne des itérations vers les nœuds MPI. Dans ce cadre, il est possible de s'appuyer sur les travaux de l'équipe TADAAM de l'INRIA de Bordeaux concernant l'environnement TREEMATCH [Jea+13] qui permet d'équilibrer les charges entre processus MPI en réduisant les communications de l'application.

Annexe A

Exemples de différents programmes pour l'implémentation d'une addition de vecteurs

Listing A.1 – Exemple d'un programme cuda

```
breaklines
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4
5   // Kernel CUDA. Chaque thread traite un element de c
6  __global__ void vecAdd(double *a, double *b, double *c, int n)
7  {
8
9      int id = blockIdx.x*blockDim.x+threadIdx.x;   // Calculer l'index pour chaque thread
10
11     if (id < n)
12         c[id] = a[id] + b[id];    // Condition de non depassement  des bords
13 }
14
15 int main( int argc, char* argv[] )
16 {
17    int n = 100000;     // Taille des vecteurs
18
19     double *h_a,*h_b,*h_c;  // Vecteurs host d entree et de sortie
20     double *d_a,*d_b,*d_c;  // Vecteurs device d entree et de sortie
21
22     h_a = (double*)malloc(n*sizeof(double)); // Allocation host
23     h_b = (double*)malloc(n*sizeof(double));
24     h_c = (double*)malloc(n*sizeof(double));
25
26     cudaMalloc(&d_a, n*sizeof(double));  // Allocation device GPU
27     cudaMalloc(&d_b, n*sizeof(double));
28     cudaMalloc(&d_c, n*sizeof(double));
29
30     for( int i = 0; i < n; i++ ) // Initialisation des vecteurs
31     {  h_a[i] = sin(i)*sin(i);
32        h_b[i] = cos(i)*cos(i);}
33
34     cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice); // Copie host to device
35     cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
36
37     int blockSize=1024 // Nombre de threads dans un block
38     int gridSize=(int)ceil((float)n/blockSize); // Nombre de block dans la grille
39
40     vecAdd<<<gridSize, blockSize>>>(d_a,d_b,d_c,n); // Appel du kernel
41
42     cudaMemcpy(h_c,d_c,bytes,cudaMemcpyDeviceToHost); // Copie device->host
43
44     cudaFree(d_a);cudaFree(d_b);cudaFree(d_c);  // Liberation des memoires device
45
46
47     free(h_a);free(h_b);free(h_c); // Liberation des memoires host
48
49     return 0;
50 }
```

Listing A.2 – Exemple d'un programme OpenCL

breaklines

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <OpenCL/opencl.h>


#define DATA_SIZE 10

using namespace std;

const char *ProgramSource =
"__kernel␣void␣add(__global␣float␣*inputA,␣__global␣float␣*inputB,␣__global␣float␣*output)\n"\
"{\n"\
"␣␣size_t␣id␣=␣get_global_id(0);\n"\
"␣␣output[id]␣=␣inputA[id]␣+␣inputB[id];\n"\
"}\n";

int main(void)
{
cl_context context;
cl_context_properties properties[3];
cl_kernel kernel;
cl_command_queue command_queue;
cl_program program;
cl_int err;
cl_uint num_of_platforms=0;
cl_platform_id platform_id;
cl_device_id device_id;
cl_uint num_of_devices=0;
cl_mem inputA, inputB, output;

size_t global;

float inputDataA[DATA_SIZE]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float inputDataB[DATA_SIZE]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
float results[DATA_SIZE]={0};

int i;

// retreive a list of platforms avaible
if (clGetPlatformIDs(1, &platform_id, &num_of_platforms)!= CL_SUCCESS)
{
printf("Unable␣to␣get␣platform_id\n");
return 1;
}

// try to get a supported GPU device
if (clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU, 1, &device_id, &num_of_devices) != CL_SUCCESS)
{
printf("Unable␣to␣get␣device_id\n");
return 1;
}

// context properties list - must be terminated with 0
properties[0]= CL_CONTEXT_PLATFORM;
properties[1]= (cl_context_properties) platform_id;
properties[2]= 0;

// create a context with the GPU device
context = clCreateContext(properties,1,&device_id,NULL,NULL,&err);

// create command queue using the context and device
command_queue = clCreateCommandQueue(context, device_id, 0, &err);

// create a program from the kernel source code
program = clCreateProgramWithSource(context,1,(const char **) &ProgramSource, NULL, &err);

// compile the program
if (clBuildProgram(program, 0, NULL, NULL, NULL, NULL) != CL_SUCCESS)
{
printf("Error␣building␣program\n");
return 1;
}

// specify which kernel from the program to execute
kernel = clCreateKernel(program, "add", &err);

// create buffers for the input and ouput

```

```
80  inputA = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * DATA_SIZE, NULL, NULL);
81  inputB = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * DATA_SIZE, NULL, NULL);
82  output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * DATA_SIZE, NULL, NULL);
83
84  // load data into the input buffer
85  clEnqueueWriteBuffer(command_queue, inputA, CL_TRUE, 0, sizeof(float) * DATA_SIZE, inputDataA, 0, NULL, NULL);
86  clEnqueueWriteBuffer(command_queue, inputB, CL_TRUE, 0, sizeof(float) * DATA_SIZE, inputDataB, 0, NULL, NULL);
87
88  // set the argument list for the kernel command
89  clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputA);
90  clSetKernelArg(kernel, 1, sizeof(cl_mem), &inputB);
91  clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);
92
93  global=DATA_SIZE;
94
95  // enqueue the kernel command for execution
96  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global, NULL, 0, NULL, NULL);
97  clFinish(command_queue);
98
99  // copy the results from out of the output buffer
100 clEnqueueReadBuffer(command_queue, output, CL_TRUE, 0, sizeof(float) *DATA_SIZE, results, 0, NULL, NULL);
101
102 // print the results
103 printf("output:␣");
104
105 for(i=0;i<DATA_SIZE; i++)
106 {
107 printf("%f␣",results[i]);
108 }
109
110 // cleanup - release OpenCL resources
111 clReleaseMemObject(inputA);
112 clReleaseMemObject(inputB);
113 clReleaseMemObject(output);
114 clReleaseProgram(program);
115 clReleaseKernel(kernel);
116 clReleaseCommandQueue(command_queue);
117 clReleaseContext(context);
118
119 return 0;
120
121 }
```

Annexe B

Codes de l'application synthétique

Listing B.1 – Code MPI+OpenMP de l'application synthétique

```
breaklines
#include <cmath>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <time.h>
#include <unistd.h>
#include <omp.h>
#include <mpi.h>


void pixelprocessSimu_cuda( float* A,float* A_dev,float* B,float* B_dev, float k, int n , int device);
void cuda_alloc_wrapper ( float * A, int n , int device) ;
void cuda_free_wrapper ( float * A, int device) ;

 using namespace std;

 void producer( float* A, float*B , int n )
 {
         timespec  current_time,last_time;

          clock_gettime(CLOCK_REALTIME, &last_time);
               for (unsigned i = 0; i < n; i++)
               {
                       A[i] = i;
                       B[i] = -i;
               }

        usleep((1)*1000);


         clock_gettime(CLOCK_REALTIME, &current_time);

  long time_temp;

        if ( (current_time.tv_nsec-last_time.tv_nsec) < 0 )   time_temp = (long)((1000000000 + current_time.
             tv_nsec-last_time.tv_nsec)) ;
       else  time_temp = (long)((current_time.tv_nsec-last_time.tv_nsec)) ;

         printf("Capture␣duration␣time␣:␣%lld.%.9ld␣\n", (long long)(current_time.tv_sec-last_time.tv_sec),
              time_temp);

 }


 void pixelprocessSimu(float* A,float* B, float k, int n)
 {
        timespec  current_time,last_time;

          clock_gettime(CLOCK_REALTIME, &last_time);

        for (int j=0;j<1000*(k);j++)
        {
               for (unsigned i = 0; i < n; i++)
               {

                       B[i] = A[i]; /// just a simulation
               }

        }

```

```
59
60           clock_gettime(CLOCK_REALTIME, &current_time);
61
62    long time_temp;
63
64          if ( (current_time.tv_nsec-last_time.tv_nsec) < 0 )   time_temp = (long)((1000000000 + current_time.
              tv_nsec-last_time.tv_nsec)) ;
65         else  time_temp = (long)((current_time.tv_nsec-last_time.tv_nsec)) ;
66
67            printf("PixelProcess␣duration␣time␣:␣%lld.%.9ld␣\n", (long long)(current_time.tv_sec-last_time.
                 tv_sec), time_temp);
68   }
69
70   void ForkJoin( float* A, float* B, float* C, float* D, float* E, int n )
71   {
72           timespec  current_time,last_time;
73
74             clock_gettime(CLOCK_REALTIME, &last_time);
75
76
77           for (unsigned i = 0; i < n; i++)
78                   {
79                           C[i] = A[i];
80                           D[i] = A[i];
81                           E[i] = A[i];
82                   }
83
84           cout << "␣Resultat␣Fork_Join␣de␣A[55]:␣" << C[55] << endl;
85           cout << "␣Resultat␣Fork_Join␣de␣B[55]:␣" << D[55] << endl;
86           cout << "␣Resultat␣Fork_Join␣de␣C[55]:␣" << E[55] << endl;
87
88                   clock_gettime(CLOCK_REALTIME, &current_time);
89
90    long time_temp;
91
92          if ( (current_time.tv_nsec-last_time.tv_nsec) < 0 )   time_temp = (long)((1000000000 + current_time.
              tv_nsec-last_time.tv_nsec)) ;
93         else  time_temp = (long)((current_time.tv_nsec-last_time.tv_nsec)) ;
94
95            printf("ForkJoin␣duration␣time␣:␣%lld.%.9ld␣\n", (long long)(current_time.tv_sec-last_time.tv_sec),
                 time_temp);
96   }
97
98   float * Consumer(float *A, float* B, float* C)
99   {
100          timespec  current_time,last_time;
101
102            clock_gettime(CLOCK_REALTIME, &last_time);
103
104          cout << "␣Resultat␣de␣A[55]:␣" << A[55] << endl;
105          cout << "␣Resultat␣de␣B[55]:␣" << B[55] << endl;
106          cout << "␣Resultat␣de␣C[55]:␣" << C[55] << endl;
107
108
109          usleep((1.5)*1000);
110
111
112          clock_gettime(CLOCK_REALTIME, &current_time);
113
114   long time_temp;
115
116         if ( (current_time.tv_nsec-last_time.tv_nsec) < 0 )   time_temp = (long)((1000000000 + current_time.
              tv_nsec-last_time.tv_nsec)) ;
117        else  time_temp = (long)((current_time.tv_nsec-last_time.tv_nsec)) ;
118
119           printf("Consumer␣duration␣time␣:␣%lld.%.9ld␣\n", (long long)(current_time.tv_sec-last_time.tv_sec),
                 time_temp);
120  }
121
122
123
124
125
126
127  ///////**** Fonction target ****/////////
128
129   int main(int argc, char **argv)
130   {
131
132
133           timespec  current_time,last_time;
134
135           int high = 512 ; int with=512;
```

```
136             int size_img = high*with;
137      const int size_loop = (int) atoi((argv[1]));
138      const int unfolding = (int) atoi((argv[2]));
139      const int chunk = (int) atoi((argv[3]));
140     // double sinTable[size];
141
142      cout << size_loop << endl;
143      clock_gettime(CLOCK_REALTIME, &last_time);
144      //cout << "Max threads 2: " << omp_get_max_threads() << endl;
145
146       int n=0;
147       int rank;
148       MPI_Init(&argc,&argv);
149       MPI_Comm_rank(MPI_COMM_WORLD, &rank);
150
151
152      #pragma omp  parallel num_threads(unfolding)
153     {
154
155      float *img_in,*Var1_1,*Var1_2, *Var2_1, *Var2_2, *Var3_1,*Var3_2,*Var3_3, *Var4_1,*Var4_2,*Var4_3,*img_out
           ;
156
157      Var1_1 = (float*) malloc ( size_img*sizeof(float));
158       Var1_2 = (float*) malloc ( size_img*sizeof(float));
159        Var2_1 = (float*) malloc ( size_img*sizeof(float));
160         Var2_2 = (float*) malloc ( size_img*sizeof(float));
161         Var3_1 = (float*) malloc ( size_img*sizeof(float));
162         Var3_2 = (float*) malloc ( size_img*sizeof(float));
163         Var3_3 = (float*) malloc ( size_img*sizeof(float));
164         Var4_1 = (float*) malloc ( size_img*sizeof(float));
165         Var4_2 = (float*) malloc ( size_img*sizeof(float));
166         Var4_3 = (float*) malloc ( size_img*sizeof(float));
167
168
169
170
171      int thread_loop=0;
172
173
174
175      cout << "␣Threads␣:␣" << omp_get_num_threads() << endl;
176
177
178      float *A_dev, *B_dev;
179      int dev;
180
181      int n1,n2;
182      if (rank == 0) {n1=0; n2=size_loop/4;}     /// scheduling over MPI nodes
183      else if (rank==1) {n1=(size_loop/4)+1; n2=size_loop/2;}
184      else if (rank==2) {n1=(size_loop/2)+1; n2=(size_loop/2)+size_loop/4;}
185      else if  (rank==2) {n1=(size_loop/2)+(size_loop/4)+1; n2=size_loop;}
186
187      #pragma omp  for private(n) private(A_dev,B_dev)  //schedule(dynamic,chunk )
188       for(n=n1; n<n2; ++n)       /// loop sur le nombre d'images
189      {
190          A_dev=0;
191                   B_dev=0;
192
193           thread_loop++;
194
195                  //dev= omp_get_thread_num() %3;  /// scheduling over gpus
196                  dev= n%3;
197
198
199                  #pragma omp task depend(out:Var1_1,Var1_2)
200                  producer( Var1_1, Var1_2 , size_img );
201
202                  #pragma omp task depend(in:Var1_1) depend(out:Var2_1)
203                  {
204                  pixelprocessSimu_cuda(Var1_1,A_dev,Var2_1,B_dev, 8, size_img,dev);
205              }
206
207                  #pragma omp task depend(in:Var1_2) depend(out:Var2_2)
208                  pixelprocessSimu_cuda(Var1_2,A_dev,Var2_2,B_dev, 4, size_img,dev);
209
210                  #pragma omp task depend(in:Var2_1,Var2_2) depend(out:Var3_1,Var3_2,Var3_3)
211                  ForkJoin(Var2_1,Var2_2,Var3_1,Var3_2,Var3_3,size_img);
212
213                  #pragma omp task depend(in:Var3_1) depend(out:Var4_1)
214                  pixelprocessSimu_cuda(Var3_1,A_dev,Var4_1,B_dev, 6,size_img,dev );
215
216                  #pragma omp task depend(in:Var3_2) depend(out:Var4_2)
217                  pixelprocessSimu_cuda(Var3_2,A_dev,Var4_2,B_dev, 3,size_img,dev );
```

```
218
219                 #pragma omp task depend(in:Var3_3) depend(out:Var4_3)
220                 pixelprocessSimu_cuda(Var3_3,A_dev,Var4_3,B_dev, 2,size_img,dev );
221
222                 #pragma omp task depend(in:Var4_1,Var4_2,Var4_3)
223                 img_out=Consumer(Var4_1,Var4_2,Var4_3);
224
225
226
227         }
228
229  }
230
231    MPI_Finalize();
232
233   clock_gettime(CLOCK_REALTIME, &current_time);
234
235   long time_temp;
236
237         if ( (current_time.tv_nsec-last_time.tv_nsec) < 0 )   time_temp = (long)((1000000000 + current_time.
              tv_nsec-last_time.tv_nsec)) ;
238        else  time_temp = (long)((current_time.tv_nsec-last_time.tv_nsec)) ;
239
240           printf("Duration␣time␣:␣%lld.%.9ld␣\n", (long long)(current_time.tv_sec-last_time.tv_sec), time_temp
                );
241     // the table is now initialized
242   }
```

Listing B.2 – Code CUDA de l'implémentation MPI+OpenMP+CUDA de l'application synthétique

```
    breaklines
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <iostream>
4  #include <time.h>
5
6  #define size 512*512
7
8  using namespace std;
9
10
11 static __global__ void image_inverse_cuda(float *in, float *out,unsigned n,unsigned time)
12 {
13         unsigned i =  blockIdx.x*blockDim.x + threadIdx.x;
14
15 if ( i < n )
16         for (int j=0;j<(1000*time);j++) {
17            out[i]=in[i]+1;
18                 __syncthreads();
19         }
20
21 }
22
23
24 void pixelprocessSimu_cuda( float* A,float* A_dev,float* B,float* B_dev, float k, int n , int device)
25
26 {
27
28   timespec  current_time,last_time;
29
30            clock_gettime(CLOCK_REALTIME, &last_time);
31 // C and CUDA code, then
32     cudaSetDevice(device);
33
34     cout << "␣A_dev:␣:␣" << A_dev << endl;
35
36      if ( A_dev==NULL) {
37                  cudaMalloc( (void**)&A_dev, n*sizeof(float) );
38      cudaMalloc( (void**)&B_dev, n*sizeof(float) );
39          } else {
40
41
42       }
43
44     unsigned threads_per_block = 128;  // without optim
45         unsigned nblocks = (n + threads_per_block-1) / threads_per_block;
```

```
46
47        cudaMemcpy( A_dev, A, n, cudaMemcpyHostToDevice );
48        // cudaMemcpyAsync( A_dev, A, n, cudaMemcpyHostToDevice , stream_copy_in);
49           image_inverse_cuda<<<nblocks,threads_per_block,0 >>>(A_dev,B_dev, n, k);
50
51        cudaMemcpy( B, B_dev, n, cudaMemcpyDeviceToHost );
52       // cudaMemcpyAsync( B, B_dev, n, cudaMemcpyDeviceToHost , stream_copy_out);
53
54         cout << "␣Resultat␣pixelProcessing_cuda␣de␣B[55]:␣" << B[55] << endl;
55
56
57      // cudaFree(A_dev);  //not free, reuse buffers
58      // cudaFree(B_dev);
59
60   clock_gettime(CLOCK_REALTIME, &current_time);
61
62    long time_temp;
63
64          if ( (current_time.tv_nsec-last_time.tv_nsec) < 0 )   time_temp = (long)((1000000000 + current_time.
                 tv_nsec-last_time.tv_nsec)) ;
65         else  time_temp = (long)((current_time.tv_nsec-last_time.tv_nsec)) ;
66
67            printf("PixelProcess_CUDA␣duration␣time␣:␣%lld.%.9ld␣\n", (long long)(current_time.tv_sec-last_time.
                   tv_sec), time_temp);
68   /**/
69
70  }
```

Listing B.3 – Code exemple d'une implémentation utilisateur de la fonction prédéfinie $MPI_data_distribute()$ de l'application synthétique

```
    breaklines
1  int sched_data(int i, int size)
2  {
3          if (size == 1) return(0);
4
5
6          else if  (size == 2){
7
8          if ( (i%4 ==0 ) || (i%4 ==1 ) || (i%4 ==2 ) ) return(0);  /// archi-005
9
10         else if ( (i%4 ==3 ) ) return(1);  /// archi-004
11
12            }
13
14         else if  (size == 3){
15
16         if ( (i%7 ==0 ) || (i%7 ==1 ) || (i%7 ==2 ) ) return(0);  /// archi-005
17
18         else if ( (i%7 ==3 ) || (i%7 ==4 ) || (i%7 ==5 ) ) return(1);  /// archi-004
19
20         else if ( (i%7 == 6 ) ) return(2);
21
22         }
23
24       else if  (size == 4){
25
26         if ( (i%9 ==0 ) || (i%9 ==1 ) || (i%9 ==2 ) ) return(0);  /// archi-005
27
28         else if ( (i%9 ==3 ) || (i%9 ==4 ) || (i%9 ==5 ) ) return(1);  /// archi-004
29
30         else if ( (i%9 == 6 ) || (i%9 == 7 ) ) return(2);  /// archi-003
31
32         else if ( (i%9 == 8 ) ) return(3);  ///archi-002
33
34       }
35
36 }
```

Bibliographie de l'auteur

[1] Farouk Mansouri, Sylvain Huet, Vincent Fristot, Dominique Houzet. *Task migration of DSP application specified with a DFG and implemented with the BSP computing model on a CPU-GPU cluster*. DASIP 2013 : 326-333

[2] Farouk Mansouri, Sylvain Huet, Dominique Houzet. *SignalPU : A Programming Model for DSP Applications on Parallel and Heterogeneous Clusters*. HPCC/CSS/ICESS 2014 : 937-944

[3] Farouk Mansouri, Sylvain Huet, Dominique Houzet. *A Visual Programming Model to Implement Coarse-Grained DSP Applications on Parallel and Heterogeneous Clusters*. Euro-Par Workshops (1) 2014 : 141-152

[4] Farouk Mansouri, Sylvain Huet, Dominique Houzet. *A domain-specific high-level programming model*. CONCURRENCY AND COMPUTATION : PRACTICE AND EXPERIENCE

Bibliographie

[All+08] Eric Allen et al. *The Fortress Language Specification*. Rap. tech. Version 1.0. Sun Microsystems, Inc., 2008, p. 262 (cf. p. 25).

[ATN10] Cédric Augonnet, Samuel Thibault et Raymond Namyst. *StarPU : a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines*. Anglais. Research Report RR-7240. INRIA, 2010, p. 33 (cf. p. 26, 81, 90).

[Aug+12] Cédric Augonnet et al. “StarPU-MPI : Task Programming over Clusters of Machines Enhanced with Accelerators”. Dans : *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*. EuroMPI’12. Vienna, Austria : Springer-Verlag, 2012, p. 298–299 (cf. p. 92).

[Bal+98] Felice Balarin et al. “Scheduling for Embedded Real-Time Systems”. Dans : *IEEE Design & ; Test of Computers* 15.1 (1998), p. 71–82 (cf. p. 42).

[Blu+95] Robert D. Blumofe et al. “Cilk : An Efficient Multithreaded Runtime System”. Dans : *SIGPLAN Not.* 30.8 (août 1995), p. 207–216 (cf. p. 40).

[Bou+12] Vincent Boulos et al. “Efficient implementation of data flow graphs on multi-gpu clusters”. Anglais. Dans : *Journal of Real-Time Image Processing* (oct. 2012), n/c (cf. p. 43, 53).

[Bro+10] François Broquedis et al. “hwloc : a Generic Framework for Managing Hardware Affinities in HPC Applications”. Dans : *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. Sous la dir. d’IEEE. Pisa, Italy, fév. 2010 (cf. p. 134).

[Bsp] *http ://bsponmpi.sourceforge.net/* (cf. p. 52).

[Bue+11] Javier Bueno et al. “Productive Cluster Programming with OmpSs”. Dans : *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*. Euro-Par’11. Bordeaux, France : Springer-Verlag, 2011, p. 555–566 (cf. p. 25).

[Cam08] C.B. Cameron. “Parallel Ray Tracing Using the Message Passing Interface”. Dans : *Instrumentation and Measurement, IEEE Transactions on* 57.2 (2008), p. 228–234 (cf. p. 37).

[CCZ07] B.L. Chamberlain, D. Callahan et H.P. Zima. “Parallel Programmability and the Chapel Language”. Dans : *Int. J. High Perform. Comput. Appl.* 21.3 (août 2007), p. 291–312 (cf. p. 25).

[Cha+01] Robit Chandra et al. *Parallel Programming in OpenMP*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2001 (cf. p. 25).

[Che07] Wei-Yu Chen. “Optimizing Partitioned Global Address Space Programs for Cluster Architectures”. Thèse de doct. EECS Department, University of California, Berkeley, 2007 (cf. p. 24).

[Col12] Mathew Colgrove. *5x in 5 Hours : Porting a 3D Elastic Wave Simulator to GPUs Using PGI Accelerator*. PGI. 2012 (cf. p. 39).

[DG08] Jeffrey Dean et Sanjay Ghemawat. "MapReduce : Simplified Data Processing on Large Clusters". Dans : *Commun. ACM* 51.1 (jan. 2008), p. 107–113 (cf. p. 51).

[EP96] Ralf Ebner et Alexander Pfaffinger. "Transformation of Functional Programs into Data Flow Graphs Implemented with PVM". Dans : *Proceedings of the Third European PVM Conference on Parallel Virtual Machine*. EuroPVM '96. London, UK, UK : Springer-Verlag, 1996, p. 251–258 (cf. p. 63).

[Fly72] M. Flynn. "Some Computer Organizations and Their Effectiveness". Dans : *Computers, IEEE Transactions on* C-21.9 (1972), p. 948–960 (cf. p. 11).

[FOL11] D. Fober, Y. Orlarey et S. Letz. "FAUST Architectures Design and OSC Support." Dans : *Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11)*. Sous la dir. d'IRCAM. 2011, p. 231–216 (cf. p. 36).

[Fri99] Matteo Frigo. "A fast Fourier transform compiler". Dans : *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*. T. 34. 5. ACM, 1999, p. 169–180 (cf. p. 38).

[GAA10] Michael I Gordon et Saman Adviser-Amarasinghe. "Compiler techniques for scalable performance of stream programs on multicore architectures". Dans : (2010) (cf. p. 35, 36, 40).

[GBP07] Thierry Gautier, Xavier Besseron et Laurent Pigeon. "KAAPI : A thread scheduling runtime system for data flow computations on cluster of multi-processors". Anglais. Dans : *2007 international workshop on Parallel symbolic computation*. Waterloo, Canada : ACM, 2007, p. 15–23 (cf. p. 26, 40).

[Gro02] Thorsten Grotker. *System Design with SystemC*. Norwell, MA, USA : Kluwer Academic Publishers, 2002 (cf. p. 41).

[Hal+91] N. Halbwachs et al. "The synchronous dataflow programming language LUSTRE". Dans : *Proceedings of the IEEE*. 1991, p. 1305–1320 (cf. p. 36).

[HHR10] Dominique Houzet, Sylvain Huet et Anis Rahman. "SysCellC : a data-flow programming model on multi-GPU". Dans : *procedia computer science* 1.1 (mai 2010), p. 1029–1038 (cf. p. 41).

[Hil+98] Jonathan M. D. Hill et al. "BSPlib : The BSP programming library." Dans : *Parallel Computing* 24.14 (1998), p. 1947–1980 (cf. p. 51).

[Hir12] E.C. Hiram. *Openhmpp*. Placpublishing, 2012 (cf. p. 25).

[HK08] Ali R. Hurson et Krishna M. Kavi. "Dataflow Computers : Their History and Future". Dans : *Wiley Encyclopedia of Computer Science and Engineering*. 2008 (cf. p. 16).

[HOR+10] Patrick HORAIN et al. "Perceiving and rendering users in a 3D interaction". Dans : *IHCI 2010 : Second IEEE International Conference on Intelligent Human Computer Interaction*. Allahabad, India : Springer, jan. 2010, p. 42–53 (cf. p. 38).

[Huy+12] Huynh Phung HUYNH et al. “Poster : Automated Mapping Streaming Applications onto GPUs”. Dans : *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion :* 2012, p. 1490 (cf. p. 41).

[HZG08] Qiming HOU, Kun ZHOU et Baining GUO. “BSGP : bulk-synchronous GPU programming”. Dans : *ACM Trans. Graph.* 27.3 (2008) (cf. p. 52).

[Jea+13] Emmanuel JEANNOT et al. “Communication and Topology-aware Load Balancing in Charm++ with TreeMatch”. Dans : *IEEE Cluster 2013*. Indianapolis, United States : IEEE, sept. 2013 (cf. p. 134).

[JW96] Ben H. H. JUURLINK et Harry A. G. WIJSHOFF. “The E-BSP Model : Incorporating General Locality and Unbalanced Communication into the BSP Model”. Dans : *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Lyon, France, August 26-29, 1996, Proceedings, Volume II*. 1996, p. 339–347 (cf. p. 51).

[KH10] David B. KIRK et Wen-mei W. HWU. *Programming Massively Parallel Processors : A Hands-on Approach*. 1st. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2010 (cf. p. 25).

[Kim+10] Seokhyun KIM et al. “A PC-based fully-programmable medical ultrasound imaging system using a graphics processing unit”. Dans : *Ultrasonics Symposium (IUS), 2010 IEEE*. 2010, p. 314–317 (cf. p. 37).

[KLH14] T. KJELDSEN, L. LASSEN et M.C. HEMMSEN. “Synthetic Aperture Sequential Beamforming implemented on multi-core platforms”. Dans : *Ultrasonics Symposium (IUS), 2014 IEEE International*. 2014, p. 2181–2184 (cf. p. 37).

[LB+13] Olivier LE BOT et al. “Separation of odontocete click trains by rhythmic analysis”. Dans : *The 6th International Workshop on Detection, Classification, Localization, and Density Estimation (DCLDE) of Marine Mammals using Passive Acoustics*. Saint Andrews, United Kingdom, juin 2013, p. 52 (cf. p. 29).

[Lee89] Edward Ashford LEE. “Scheduling strategies for multiprocessor real-time DSP”. Dans : 1989, p. 1279–1283 (cf. p. 42).

[LP95] Edward A. LEE et Thomas PARKS. “Dataflow Process Networks”. Dans : *Proceedings of the IEEE*. 1995, p. 773–799 (cf. p. 33).

[Mah+11] Ahmed MAHMOUDI Sidi et al. “Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes”. Dans : *Rencontres francophones du parallélisme*. Saint-Malo, France, mai 2011 (cf. p. 40).

[Mal+10] Grzegorz MALEWICZ et al. “Pregel : A System for Large-scale Graph Processing”. Dans : *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA : ACM, 2010, p. 135–146 (cf. p. 51).

[Mey+12] B. MEYER et al. “Convey vector personalities - FPGA acceleration with an openmp-like programming effort ?” Dans : *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. 2012, p. 189–196 (cf. p. 16).

[MKM14] Sidi Ahmed MAHMOUDI, Michal KIERZYNKA et Pierre MANNEBACK. “Real-Time GPU-Based Motion Detection and Tracking Using Full HD Videos”. Dans : *Intelligent Technologies for Interactive Entertainment.* Springer, 2014 (cf. p. 38).

[Mok00] Mohand MOKHTARI. *MATLAB 5.2 and 5.3 et SIMULINK 2 and 3 pour etudiants et ingenieurs.* Springer, 2000, p. I–XX, 1–662 (cf. p. 36).

[Mun+11] A. MUNSHI et al. *OpenCL Programming Guide.* OpenGL. Pearson Education, 2011 (cf. p. 24).

[Ngu07] Hubert NGUYEN. *Gpu gems 3.* Addison-Wesley Professional, 2007 (cf. p. 94).

[Pac96] Peter S. PACHECO. *Parallel Programming with MPI.* San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1996 (cf. p. 24).

[Pel+14a] M. PELCAT et al. “Preesm : A dataflow-based rapid prototyping framework for simplifying multicore DSP programming”. Dans : *EDERC, 2014 6th European Embedded Design in.* 2014, p. 36–40 (cf. p. 41).

[Pel+14b] Maxime PELCAT et al. “PREESM : A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming”. Dans : *EDERC.* Italy, sept. 2014, p. 36 (cf. p. 35, 36).

[Pha13] T.Q. PHAM. “Parallel Implementation of Geodesic Distance Transform with Application in Superpixel Segmentation”. Dans : *Digital Image Computing : Techniques and Applications (DICTA), 2013 International Conference on.* 2013, p. 1–8 (cf. p. 39).

[PM91] K.K. PARHI et D.G. MESSERSCHMITT. “Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding”. Dans : *Computers, IEEE Transactions on* 40.2 (1991), p. 178–195 (cf. p. 90).

[Pol13] Antony POLUKHIN. *Boost C++ application development cookbook.* Birmingham : Packt Publ., 2013 (cf. p. 90).

[Pto14] Claudius PTOLEMAEUS, éd. *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014 (cf. p. 35, 36).

[Puj+12] C. PUJARA et al. “Real-time stereo video decoding and rendering on multi-core architecture”. Dans : *Communications (NCC), 2012 National Conference on.* 2012, p. 1–4 (cf. p. 37).

[Rei07] James REINDERS. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O’Reilly, 2007, p. I–XXV, 1–303 (cf. p. 26, 40).

[Rev09] G. E. REVESZ. *Lambda-calculus, Combinators and Functional Programming.* 1st. New York, NY, USA : Cambridge University Press, 2009 (cf. p. 33).

[RHP11] Anis RAHMAN, Dominique HOUZET et Denis PELLERIN. “Visual Saliency Model on Multi-GPU”. Anglais. Dans : *GPU Computing Gems Emerald Edition.* Elsevier, 2011, p. 451–472 (cf. p. 113, 114).

[Ros+11] Christopher J. ROSSBACH et al. “PTask : Operating System Abstractions to Manage GPUs As Compute Devices”. Dans : *Proceedings of SOSP ’11.* SOSP ’11. Cascais, Portugal : ACM, 2011, p. 233–248 (cf. p. 40).

[RR+09] R. da ROSA RIGHI et al. “MigBSP : A Novel Migration Model for Bulk-Synchronous Parallel Processes Rescheduling”. Dans : *High Performance Computing and Communications, 2009. HPCC ’09. 11th IEEE International Conference on.* 2009, p. 585–590 (cf. p. 63).

[Sar+12] Vijay SARASWAT et al. *X10 Language Specification.* Rap. tech. IBM, 2012 (cf. p. 25).

[Sch+13] L. SCHOR et al. “Exploiting the parallelism of heterogeneous systems using dataflow graphs on top of OpenCL”. Dans : *ESTIMedia, 2013 IEEE 11th Symposium.* 2013, p. 41–50 (cf. p. 41).

[Seo+10] Sangwon SEO et al. “HAMA : An Efficient Matrix Computation with the MapReduce Framework”. Dans : *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on.* 2010, p. 721–726 (cf. p. 51).

[Sin07] Oliver SINNEN. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing).* Wiley-Interscience, 2007 (cf. p. 32).

[SK10] Jason SANDERS et Edward KANDROT. *CUDA by Example : An Introduction to General-Purpose GPU Programming.* 1st. Addison-Wesley Professional, 2010 (cf. p. 24, 38).

[ST98] David B. SKILLICORN et Domenico TALIA. “Models and Languages for Parallel Computation”. Dans : *ACM Comput. Surv.* 30 (1998), p. 123–169 (cf. p. 20, 22).

[TK96] Pilar de la TORRE et Clyde P. KRUSKAL. “Submachine Locality in the Bulk Synchronous Setting (Extended Abstract)”. Dans : *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II.* Euro-Par ’96. London, UK, UK : Springer-Verlag, 1996, p. 352–358 (cf. p. 51).

[Top] *http ://www.top500.org* (cf. p. 37).

[Val90] Leslie G. VALIANT. “A bridging model for parallel computation”. Dans : *Commun. ACM* 33.8 (août 1990), p. 103–111 (cf. p. 49).

[Ver+03] Alex VERSTAK et al. “BSML : A binding schema markup language for data interchange in problem solving environments.” Dans : *Scientific Programming* 11.3 (2003), p. 199–224 (cf. p. 52).

[Whi09] Tom WHITE. *Hadoop : The Definitive Guide.* 1st. O’Reilly Media, Inc., 2009 (cf. p. 51).

[Yed] *http ://yed.yworks.com/support/manual/index.html* (cf. p. 95).

[Yze14] Bisseling R. H. Roose D. Meerbergen K. YZELMAN A. N. “MulticoreBSP for C : A High-Performance Library for Shared-Memory Parallel Programming”. eng. Dans : *International Journal of Parallel Programming* 42.4 (2014), p. 619–642 (cf. p. 52).

[UPC05] UPC CONSORTIUM. *UPC Language Specifications, v1.2.* Tech Report LBNL-59208. Lawrence Berkeley National Lab, 2005 (cf. p. 25).

Résumé — Depuis une dizaine d'année, l'évolution des machines de calcul tends vers des architectures parallèles et hétérogènes à l'image des grilles de calcul. Composés de plusieurs nœuds connectés via le réseaux et incluant chacun des unités de traitement hétérogènes, elles permettent une grande performance de calcul. Pour programmer ces architectures, l'utilisateur doit s'appuyer sur des modèles de programmation comme MPI, OpenMP, ou CUDA. Ces modèles visent deux objectifs : augmenter la productivité du programmeur en offrant une abstraction des spécificités de l'architecture, et préserver l'efficacité en exploitant ses performances. Cependant, ces deux objectifs sont paradoxaux, les atteindre en même temps est une mission difficile. En effet, augmenter l'abstraction d'an modèle sans réduire son efficacité est vrai défit pour les concepteurs. Dans cette thèse, nous proposons d'exploiter l'idée qu'un modèle de programmation spécifique à un domaine particulier peut atteindre les deux objectifs. Nous proposons deux modèles spécifiques aux traitements du signal et de l'image sur cluster hétérogène. Le premier modèle est statique que nous enrichissons avec une fonctionnalité de migration de tâches. Le deuxième modèle est dynamique basé sur le moteur d'exécution StarPU. Les deux modèles offre d'une part un haut niveau d'abstraction en modélisant les applications sous forme de graphe de flot de données. D'autre part, ils permettent d'exploiter efficacement les différents niveaux de parallélisme (tâches, données, graphe). Nous validons ces deux modèles par l'implémentation de deux applications de traitement de l'images du monde réel sur cluster CPU-GPU.
Mots clés : Modèle de programmation. Grille de calcul parallèle et hétérogène. Modèle de calcul DFG. Parallélisme de tâches, parallélisme de données. Traitement de signal et d'images.

Abstract — For ten years, the evolution of computing machines tend to parallel and heterogeneous architectures such as clusters. Composed of several nodes connected via the network and including heterogeneous processing units, they allow a high performance computation. To program these architectures, the user must rely on programming models such as MPI, OpenMP or CUDA. These models have two objectives : to increase programmer productivity by providing an abstraction of the architectural specificities, and preserve the efficiency by exploiting its performances. However, these two goals are paradoxical, achieving at the same time is a difficult mission. Indeed, increasing the model's level of abstraction without reducing its effectiveness is a real challenge for designers. In this thesis, we propose to exploit the idea that a specific programming model to a particular area can achieve both goals. We offer two specific models to signal and image processing on heterogeneous cluster. The first model is static that we enrich with a task migration feature. The second model is based on the dynamic engine StarPU. Both models offer firstly a high level of abstraction in modeling applications as data flow graph. And secondely, they allow to effectively operate the various levels of parallelism (task, data, graph). We validate these models by the implementation of two real-world applications of images processing on CPU-GPU cluster.
Keywords : Parallel programming models. Parallel and heterogeneous cluster. DFG Model of computation. Task parallelism, data parallelism. Digital signal processing.

GIPSA-lab, 11 rue des Mathématiques, Grenoble Campus BP46, F-38402
SAINT MARTIN D'HERES CEDEX