Numéro d'ordre : 4340

Université des Sciences et Technologies de Lille

Thèse
présentée pour obtenir le titre de
docteur spécialité Informatique
par
Julien Taillard

Une approche orientée modèle pour la parallélisation d'un code de calcul éléments finis

Thèse soutenue le 9 Février 2009, devant la commission d'examen formée de :

Pierre Manneback	Professeur Faculté Polytechnique de Mons	**Rapporteur**
Christian Perez	Chargé de recherche INRIA	**Rapporteur**
Franck Cappello	Directeur de recherche INRIA	**Examinateur**
Laurent Nicolas	Directeur de recherche CNRS	**Examinateur**
Mamy Rakotovao	Chercheur Valeo Electrical Systems	**Examinateur**
Frédéric Guyomarc'h	Maitre de conférence LIFL	**Examinateur**
Jean-Luc Dekeyser	Professeur LIFL	**Directeur**
Francis Piriou	Professeur L2EP	**Co-Directeur**

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

© 2010 Tous droits réservés.

Remerciements

Ce manuscrit est l'aboutissement de plusieurs années de recherche, mais c'est également le fruit de discussions, d'apprentissages et de moments de détente avec différentes personnes que j'aimerais remercier ici.

Je tiens tout d'abord à exprimer ma profonde gratitude envers Jean-Luc Dekeyser et Francis Piriou pour m'avoir encadré, conseillé et soutenu tout au long de ma thèse et pour m'avoir guidé et aidé lors de l'écriture de ce manuscrit. Je suis également reconnaissant envers Frédéric Guyomarc'h avec qui j'ai collaboré durant deux ans et auprès duquel j'ai beaucoup appris. Je remercie également les rapporteurs de cette thèse, Pierre Manneback et Christian Perez ainsi que les examinateurs, Franck Cappello, Laurent Nicolas et Mamy Rakotovao, pour avoir lu, commenté ce manuscrit et pour leurs judicieux conseils lors de ma présentation. Je tiens en outre à remercier Valeo Electrical System et la région Nord-Pas de Calais pour avoir financé mes recherches au cours de ces trois années dans le cadre du projet CNRT Futurelec.

Les travaux de cette thèse s'inscrivent principalement dans le cadre des travaux de l'équipe-projet INRIA DaRT et plus particulièrement celui du développement de l'environnement Gaspard. Je remercie ici tous les membres du projet pour leurs collaborations et la bonne ambiance régnant dans l'équipe. Le développement de Gaspard étant un travail d'équipe, je salue ici l'ensemble des développeurs de l'environnement et notamment les anciens doctorants avec qui j'ai réalisé la majorité de mes développements : Éric, Sébastien, Huafeng et Rabie sans oublier Antoine, ancien ingénieur de l'équipe, qui a posé les bases du développement et avec qui j'ai réalisé une partie de mes études. Je souhaite bon courage aux doctorants actuels pour la rédaction de leur thèse : Adolf, Calin, Imran, Myriam, Nicolas, Vincent et Wendell. Enfin, je remercie l'ensemble des membres de l'équipe Modelisation du L2EP avec qui j'ai collaboré et plus particulièrement Thomas et Yvonnick pour leurs conseils et leurs différentes explications sur les problèmes liés à l'électromagnétisme.

Enfin, je souhaiterais remercier l'ensemble de ma famille et de mes amis pour m'avoir soutenu si souvent au cours de ma thèse.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Table des matières

Table des matières 3

1 Introduction 9
1.1 Contexte et problématique . . . 9
1.2 Contributions de cette thèse . . . 11
1.3 Plan de ce mémoire . . . 12

2 État de l'art 13
2.1 La conception de nouveaux langages parallèles . . . 15
2.1.1 Propriétés pouvant simplifier la programmation parallèle . . . 15
2.1.2 Les langages parallèles . . . 18
2.1.3 Bibliothèques de communications . . . 19
2.1.4 Langages à adressage global partitionné . . . 19
2.1.5 Global Arrays . . . 21
2.1.6 OpenMP . . . 21
2.1.7 High Performance Fortran (HPF) . . . 22
2.1.8 ZPL . . . 22
2.1.9 Les nouveaux langages parallèles . . . 22
2.1.10 Récapitulatif des langages parallèles . . . 24
2.2 Parallélisation automatique et génération de code parallèle . . . 24
2.2.1 Génération de code parallèle à partir de programme séquentiel . . . 25
2.2.2 D'un langage parallèle à un autre . . . 26
2.3 Ingénierie dirigée par les modèles . . . 27
2.3.1 Les modèles . . . 27
2.3.2 Les métamodèles . . . 28
2.3.3 Transformations de modèle . . . 30
2.3.4 Unified Modeling Language (UML) . . . 33
2.3.5 Modélisation d'applications parallèles . . . 33
2.4 L'environnement Gaspard . . . 35
2.4.1 Co-modélisation - modèle Y (le haut du Y) . . . 35
2.4.2 Langage de spécification : Array-OL . . . 36
2.4.3 Modélisation à haut niveau . . . 41
2.4.4 Un modèle exécutable . . . 46
2.4.5 Le métamodèle Gaspard . . . 48
2.5 Conclusion . . . 48

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

3 Méthodologie de compilation d'un modèle de haut niveau pour le calcul haute performance basée sur l'IDM **51**
3.1 Les apports d'une approche IDM pour le calcul haute performance 52
3.1.1 Le métamodèle : un langage parallèle 52
3.1.2 Les transformations de modèle : le compilateur 52
3.2 Définition d'une chaîne de compilation des applications de calcul haute performance 53
3.2.1 Cible de la chaîne de compilation : les machines à mémoires partagées 53
3.2.2 Stratégie globale d'une chaîne de compilation IDM partant d'une spécification Array-OL 54
3.2.3 Implémentation dans Gaspard 55
3.3 Modèle d'exécution en OpenMP des applications spécifiées en Array-OL et distribuées sur les différents processeurs et/ou cœurs 57
3.3.1 Génération de code SPMD : modèle d'exécution 57
3.3.2 Parallélisme de tâches 58
3.3.3 Parallélisme de données 59
3.3.4 Synchronisation des tâches 59
3.3.5 Gestion des tilers 59
3.4 Conclusion du chapitre 60

4 Extension de l'expressivité à haut niveau **61**
4.1 Introduction d'un composant itératif 62
4.1.1 Modélisation d'un composant itératif 63
4.1.2 Idées de compilation des composants itératifs 67
4.1.3 Utilisation d'un tel composant itératif 68
4.2 Spécification de port en entrée / sortie 68
4.2.1 Interface d'utilisation de fonctions de bibliothèques 69
4.2.2 Modélisation de port permettant de faire de l'entrée/sortie 70
4.2.3 Idées pour la compilation du concept 72
4.3 Utilisation de données du contexte d'exécution 73
4.3.1 Besoins 73
4.3.2 Modélisation 73
4.3.3 Compilation 75
4.4 Introduction de la généricité dans les modèles 75
4.4.1 Les templates UML 76
4.4.2 Un patron d'introduction de la généricité dans les métamodèles à base de composants 79
4.4.3 La généricité dans le métamodèle Gaspard 80
4.5 Conclusion du chapitre 84

5 Un métamodèle de langage procédural/impératif **85**
5.1 Motivations et objectifs du métamodèle 86
5.2 Grammaire YACC du C ANSI 86
5.3 Structures et concepts du métamodèle 87
5.3.1 Structure globale d'un modèle 87
5.3.2 Les fichiers 88
5.3.3 Les variables et les types de données 90

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

5.3.4 La définition des structures 91
5.3.5 Modélisation des expressions 94
5.3.6 Les procédures et les fonctions dans le métamodèle 95
5.3.7 Conclusion sur la définition du métamodèle 96
5.4 Extension du métamodèle pour les langages parallèles 96
5.4.1 Ajout de structure pour le support d'OpenMP 96
5.4.2 Modélisation de la bibliothèque MPI 98
5.5 Utilisation de ce métamodèle dans la chaîne de transformation 99
5.5.1 Appel de procédures 99
5.5.2 Génération du programme principal 99
5.5.3 Passage de paramètres par adresse 99
5.6 Génération de code à partir d'un modèle respectant ce métamodèle 99
5.6.1 Transformation *modèle vers texte* 101
5.6.2 Implémentation de la génération de code 101
5.7 Conclusion du chapitre 102

6 Transformation d'un modèle Gaspard vers un modèle dans le métamodèle de langage 105
6.1 Modèles intermédiaires de haut niveau 106
6.1.1 Le métamodèle Gaspard 106
6.1.2 Le métamodèle Polyhedron 107
6.1.3 Le métamodèle Loop 112
6.2 Un modèle complet pouvant être transformé 113
6.2.1 Déploiement des composants élémentaires 113
6.2.2 Allocation des éléments terminaux du modèle 113
6.2.3 Spécification du composant principal 113
6.3 Du métamodèle Gaspard vers le métamodèle Loop 114
6.3.1 Du métamodèle Gaspard vers le métamodèle Polyhedron 115
6.3.2 Du métamodèle Polyhedron vers le métamodèle Loop 117
6.4 Du métamodèle Loop vers le métamodèle de langage procédural contenant OpenMP 118
6.4.1 Exemple sur lequel les étapes de la transformation seront illustrées . . 118
6.4.2 Ordonnancement du graphe de tâches hiérarchiques 119
6.4.3 Synchronisation des tâches 120
6.4.4 Gestion des tilers 121
6.4.5 Gestion des variables 123
6.4.6 Optimisations des accès mémoires 124
6.5 Implémentation des transformations 125
6.5.1 Le moteur de transformation MoMoTE 125
6.5.2 Implémentation de la transformation métamodèle Loop vers métamodèle de langage procédural 129
6.6 Génération de code à partir de modèles et résultats d'exécution des codes générés 132
6.6.1 Génération de code à partir d'un modèle 133
6.6.2 Machines de tests 136
6.6.3 Test de différents algorithmes de multiplication de matrices 137
6.6.4 Effet de la distribution sur les performances 139

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

6.6.5 Méthodologie d'utilisation de Gaspard et de modélisation des applications afin d'obtenir des performances . . . 141
6.7 Conclusion du chapitre . . . 142

7 Étude de cas : application à la simulation de phénomènes électromagnétiques 143
7.1 Introduction . . . 144
7.2 Lois physiques régissant les phénomènes électromagnétiques . . . 144
7.2.1 Domaine continu . . . 144
7.2.2 Domaine discret . . . 148
7.3 CARMEL . . . 151
7.3.1 Présentation de CARMEL . . . 151
7.3.2 Parallélisation de CARMEL . . . 151
7.3.3 Structure de matrices creuses utilisées . . . 152
7.4 Patrons de modélisation de briques de bases d'une application basée sur la méthode des éléments finis . . . 153
7.4.1 Calcul d'une intégrale . . . 153
7.4.2 Calcul d'une contribution élémentaire . . . 155
7.5 Assemblage d'une matrice issue de la méthode des éléments finis . . . 157
7.5.1 Assemblage d'un problème de magnétostatique en deux dimensions en utilisant la formulation vectorielle (A) . . . 157
7.5.2 Résultats d'exécutions . . . 162
7.6 Résolutions des systèmes d'équations . . . 162
7.6.1 Les méthodes de résolution liées à CARMEL . . . 163
7.6.2 Modélisation des opérations classiques des solveurs itératifs de type méthode de Krylov . . . 163
7.6.3 Modélisation et résultats d'exécution du gradient conjugué non préconditionné . . . 165
7.6.4 Modélisation et résultats d'exécution du gradient conjugué préconditionné par SSOR . . . 169
7.7 Conclusion . . . 173

8 Vers la simulation de machines complexes 175
8.1 Pistes proposées par l'analyse numérique . . . 176
8.1.1 Utilisation d'autres méthodes de résolution . . . 176
8.1.2 Préconditionneurs parallèles . . . 177
8.1.3 Méthodes de décomposition de domaines . . . 178
8.1.4 Utilisation de la FIT . . . 180
8.2 Pistes proposées au niveau des modèles et de la compilation de ceux-ci . . . 181
8.2.1 Introduction des composants parallèles . . . 181
8.2.2 Expression du parallélisme sur des matrices creuses . . . 183
8.3 Conclusion . . . 184

9 Conclusions et perspectives 187

Bibliographie personnelle 191

Bibliographie 193

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

A Le métamodèle Gaspard **203**
A.1 Partie générale 203
A.2 Partie applicative 204
A.3 Partie architecture 204
A.4 Partie déploiement 205
A.5 Association 205

B Code généré pour l'algorithme du gradient conjugué **209**

Résumé/Abstract **216**

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

© 2010 Tous droits réservés.

http://doc.univ-lille1.fr

Chapitre 1

Introduction

1.1 Contexte et problématique 9
1.2 Contributions de cette thèse 11
1.3 Plan de ce mémoire 12

1.1 Contexte et problématique

Dans de nombreux secteurs de l'industrie, la conception et la réalisation de nouvelles machines sont basées sur un même processus composé de la conception et de la validation de la machine. La conception consiste à définir les besoins de la machine et à réaliser des plans de fabrications. Dans un second temps, un prototype est usiné afin de s'assurer que la machine remplit bien sa fonction et, le cas échéant, les défauts sont remontés à la phase de conception afin d'y apporter les modifications nécessaires aux plans et réusiner un nouveau prototype. Cette répétition d'étapes de conceptions et de validations est couteuse aussi bien au niveau financier que temporelle, c'est pourquoi il est aujourd'hui courant d'avoir recours à une phase intermédiaire entre la conception et la réalisation, il s'agit de la simulation numérique.

À partir d'une modélisation mathématique des phénomènes physiques en jeu et d'une représentation de la machine, il est possible de prédire sur ordinateur le comportement de la machine. De nombreux défauts peuvent être identifiés dès cette phase et les plans de conception modifiés avant même l'usinage d'un prototype. La précision de la prédiction dépend alors de la finesse de modélisation de la machine, cette finesse jouant sur le nombre d'inconnues influant lui-même la complexité de résolution du problème. Cette phase permet de réduire les temps de conception et les coûts : les temps de simulation étant petits par rapport au temps de construction d'un prototype et le nombre de prototypes nécessaires étant alors restreint. Les machines à réaliser sont néanmoins toujours plus complexes, plus puissantes et nécessitent plus de précisions et les temps de simulations augmentent avec cette complexité. Il se pose alors la nécessité de réduire les temps de simulation, ce qui est généralement effectué à l'aide de machines parallèles.

Les machines parallèles sont utilisées depuis de nombreuses années afin d'obtenir une grande puissance de calcul et ainsi de réduire les temps d'exécution des applications gourmandes en calculs. Le principe de base de ces machines est de réaliser les calculs parallèlement sur différents processeurs. Bien que l'évolution des processeurs soit constante et notamment

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

l'augmentation des puissances de calcul disponibles par processeurs, l'utilisation de ces machines est plus que jamais d'actualité.

En effet, l'évolution technologique constante dans le domaine des semi-conducteurs entraîne l'évolution des systèmes électroniques. L'une des principales évolutions technologiques est la finesse de gravures des puces qui diminue régulièrement. À l'aide de cette finesse de gravure, et suivant la loi de Moore, il est possible, tous les 18 à 24 mois, de placer deux fois plus de transistors sur une puce. Ce nombre de transistors croissant a permis aux fabricants de processeurs d'augmenter toujours la puissance de calcul des processeurs. L'augmentation de la fréquence des processeurs a entrainé des problèmes physiques de puissances électriques consommées, mais également de dissipation de la chaleur qui a contraint les constructeurs à ne plus augmenter la fréquence de leurs processeurs. Les fondeurs ont alors utilisé ce grand nombre de transistors disponibles afin de placer plusieurs cœurs de calculs dans un processeur. Cela permet, sans augmenter la fréquence, d'accroître la puissance des processeurs. L'apparition et la généralisation de ces processeurs multi-cœurs élargissent le spectre des machines parallèles : des simples ordinateurs classiques ayant des processeurs multi-cœurs aux machines massivement parallèles à la puissance pétaflopique.

Afin d'exploiter la puissance de calculs de ces machines, la programmation doit être effectuée en parallèle. Bien que des travaux soient menés afin de réaliser automatiquement la parallélisation des applications écrites en séquentiel, les applications parallèles sont encore majoritairement écrites manuellement. Il est alors nécessaire de faire appel à des spécialistes, ce type de programmation étant considéré comme dur, car demandant des connaissances informatiques pointues sur les mécanismes utilisés dans les architectures matérielles, mais également sur les outils et langages liés au parallélisme afin d'écrire des codes performants. Les programmes parallèles sont principalement développés en utilisant les langages C ou Fortran 90 agrémentés des bibliothèques OpenMPet/ou MPI. Ces langages sont de bas niveau et ne proposent pas ou peu de mécanismes provenant du génie logiciel. Lors du développement d'applications, deux aspects rentrent en jeu : une bonne structuration/écriture du code lui permettant d'être facilement compréhensible et modifiable et les performances de celui-ci. Dans le cadre du calcul scientifique, les performances sont souvent privilégiées au détriment des bonnes méthodes de développement. Avec la généralisation des architectures parallèles, il est important de rendre accessible la programmation parallèle en la simplifiant tout en ayant des performances. Dans ce but, de nouveaux langages parallèles et méthodologie de développement doivent être proposés. Il sera alors nécessaire d'avoir un compromis entre la méthode de développement de l'application parallèle et les performances.

Parallèlement à la généralisation des architectures parallèles, la complexité des applications informatiques croit également. Les utilisateurs demandent toujours plus de fonctionnalité, cela rend le développement plus complexe. Afin de répondre à cette demande, les entreprises ont alors comme solutions d'augmenter le temps accordé au développement de leurs applications ou d'améliorer la productivité des développeurs afin de produire les applications complexes dans le même temps (voir même plus court qu'auparavant). L'amélioration de la productivité des développeurs passe par la définition de nouvelles méthodes de développement. Parmi ces méthodes de développements se trouve l'ingénierie dirigée par les modèles (IDM). Cette méthodologie basée sur l'utilisation de modèles à tous les niveaux de conceptions et le raffinement de ces modèles entre les niveaux propose de rendre productifs les modèles, auparavant utilisés pour la réflexion et la documentation, en produisant du code à partir de ceux-ci.

Les travaux que nous présentons dans ce mémoire se situent au carrefour de ces différentes

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

problématiques : aider à exploiter les architectures parallèles et à en obtenir de la performance tout en préconisant de bonnes méthodes de développements. Cette thèse peut alors se résumer en le développement d'une approche orientée modèle pour les applications parallèles et son application à la simulation de machines électriques par la méthode des éléments finis.

1.2 Contributions de cette thèse

L'approche globale développée est d'utiliser l'ingénierie dirigée par les modèles afin de rendre plus accessible la programmation parallèle. La spécification des applications à l'aide de modèles devant notamment permettre une montée en abstraction de la spécification afin de ne pas avoir à gérer un grand nombre de détails d'implémentation ainsi que de s'abstraire totalement des langages de programmation qui seront ensuite utilisés. À partir de ces modèles de haut niveau des mécanismes produisant automatiquement du code parallèle dans le langage et pour l'architecture souhaités sont proposés.

Les principales contributions de cette thèse sont les suivantes :

1. *Une méthodologie d'IDM* permettant la compilation d'une spécification à haut niveau vers du code parallèle. L'expression du parallélisme est basée sur un langage de spécification Array-OLexprimant le parallélisme de tâches par un graphe de tâches et le parallélisme de données sur des dépendances de données entre tableaux. Les cibles de cette méthodologie lors de cette thèse seront les machines à mémoires partagées (classe dont font partie les machines ayant des processeurs multi-cœurs) programmées avec le standard OpenMP ;
2. *L'extension du domaine d'applications* pouvant être modélisées dans le langage de haut niveau. Le langage Array-OLétant restreint aux applications de traitement de signal intensif, il est nécessaire de l'étendre aux applications de calcul scientifique. Les extensions sont notamment le concept de modélisation de boucle dont le domaine d'itération n'est pas connu avant l'exécution, ainsi que des concepts permettant l'utilisation de données en entrée/sortie et la paramétrisation des modèles ;
3. *Un métamodèlede langage procédural* est défini afin de factoriser les transformations visant à produire des codes dans des langages procéduraux. Ce métamodèleautorise l'utilisation des bibliothèques tierces et notamment de bibliothèques de calculs parallèles. La génération de code, à partir d'un modèle dans ce métamodèle, est effectuée par une transformation de modèle vers texte qui ajoute une syntaxe au métamodèle. Les mêmes transformations sont ainsi utilisées pour produire aussi bien du Fortran et du C utilisant le langage OpenMP. D'autres métamodèlesnécessaires à la chaîne de transformations sont également définis ;
4. *Des transformations de modèles* permettant de passer automatiquement de la spécification de haut niveau basée sur un graphe de tâches vers les langages C et Fortran utilisant OpenMPsont proposées ;
5. *Des patrons de conceptions* des opérations classiques utilisées dans une application basée sur la méthode des éléments finis sont présentés. Ces patrons donnent un cadre à adapter aux différents calculs à effectuer et sont appliqués à la simulation de phénomènes électromagnétiques.

L'ensemble de ces contributions est implémenté dans l'environnement Gaspard développé dans le cadre du projet INRIA DaRT.

Thèse de Julien Taillard, Lille 1, 2009

1.3 Plan de ce mémoire

Les contributions citées précédemment sont décrites dans ce mémoire structuré en sept chapitres suivi de la conclusion générale.

Le chapitre 2 : État de l'art présente le contexte des domaines d'études couverts par cette thèse. Cela recouvre les langages parallèles existants et ceux en développement afin de simplifier la programmation parallèle, les approches de parallélisations automatiques, l'ingénierie dirigée par les modèles et son utilisation dans le cadre du parallélisme et enfin l'environnement Gasparddans lequel sont implémentées les différentes contributions de cette thèse.

Le chapitre 3 : Méthodologie de compilation d'un modèle haut niveau pour le calcul haute performance basé sur l'IDM présente la méthodologie IDM utilisée lors de cette thèse, le modèle d'exécution des applications modélisées et l'introduction de la méthodologie dans l'environnement Gaspard.

Le chapitre 4 : Extension de l'expressivité à haut niveau introduit les différentes extensions nécessaires dans le langage de haut niveau afin de traiter des applications de calculs scientifiques. Ces extensions vont de l'introduction des boucles d'itérations à la paramétrisation des modèles en passant par l'utilisation de données en entrée/sortie.

Le chapitre 5 : Un métamodèlede langage procédural/impératif décrit le métamodèlede langage procédural, le support de OpenMPdans ce métamodèleet la génération de code, via des transformations modèles vers textes, à partir d'un modèle conforme à ce métamodèle.

Le chapitre 6 : Transformation d'un modèle Gaspardvers un modèle dans le métamodèlede langage présente les transformations développées pour passer d'un modèle de haut niveau vers du code OpenMP, une discussion portant sur l'utilisation des moteurs de transformations est également présentée ainsi que des résultats illustrant la pertinence de l'approche.

Le chapitre 7 : Étude de cas : application à la simulation de machines électriques présente l'utilisation de cette méthodologie sur la simulation de phénomènes électriques. À partir de ces problèmes, des patrons de conceptions pour la modélisation d'une application basée sur les éléments finis sont extraits. Enfin des résultats d'exécutions de l'assemblage et de la résolution du système d'équations par la méthode du gradient conjugué sont présentés.

Le chapitre 8 : Vers la simulation de machines complexes présente les différentes améliorations pouvant être réalisées sur l'utilisation des méthodes d'analyse numérique ainsi que dans le domaine des modèles afin de réaliser la simulation de systèmes complexes à partir de modèle Gaspardtout en obtenant de bonnes performances.

Le chapitre 9 : Conclusions et perspectives conclut ces travaux de thèses en apportant une discussion sur l'utilisation d'une telle approche et dégage les perspectives de ces travaux.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Chapitre 2

État de l'art

2.1 La conception de nouveaux langages parallèles **15**
2.1.1 Propriétés pouvant simplifier la programmation parallèle 15
2.1.2 Les langages parallèles 18
2.1.3 Bibliothèques de communications 19
2.1.4 Langages à adressage global partitionné 19
2.1.5 Global Arrays 21
2.1.6 OpenMP 21
2.1.7 High Performance Fortran (HPF) 22
2.1.8 ZPL 22
2.1.9 Les nouveaux langages parallèles 22
2.1.10 Récapitulatif des langages parallèles 24
2.2 Parallélisation automatique et génération de code parallèle **24**
2.2.1 Génération de code parallèle à partir de programme séquentiel . . 25
2.2.2 D'un langage parallèle à un autre 26
2.3 Ingénierie dirigée par les modèles **27**
2.3.1 Les modèles 27
2.3.2 Les métamodèles 28
2.3.3 Transformations de modèle 30
2.3.4 Unified Modeling Language (UML) 33
2.3.5 Modélisation d'applications parallèles 33
2.4 L'environnement Gaspard **35**
2.4.1 Co-modélisation - modèle Y (le haut du Y) 35
2.4.2 Langage de spécification : Array-OL 36
2.4.3 Modélisation à haut niveau 41
2.4.4 Un modèle exécutable 46
2.4.5 Le métamodèle Gaspard 48
2.5 Conclusion **48**

Le calcul scientifique est un domaine qui utilise l'informatique comme outil pour résoudre des problèmes numériques provenant d'autres domaines scientifiques. Ce domaine nécessite toujours plus de puissance de calcul, les problèmes à simuler étant toujours plus complexes et demandant plus de précisions. Deux approches existent pour accroître la puissance des calculateurs. La première, dîte séquentielle, consiste à faire exécuter les instructions plus

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

rapidement par le processeur en augmentant la fréquence de celui-ci. Ainsi, en effectuant un simple remplacement de processeurs, la puissance de calculs augmentait pour la même application. La fréquence des processeurs n'augmente plus aussi rapidement qu'auparavant à cause de problèmes physiques, tels que la dissipation de la chaleur et la consommation d'énergie (la consommation augmentant avec un facteur au cube de la fréquence), rencontrés par les fondeurs. L'approche séquentielle ne permet plus d'augmenter très rapidement la puissance de calcul et ne peut plus être utilisée seule. Afin de continuer à augmenter la puissance de calcul, les constructeurs ont basé leurs nouvelles architectures de processeurs sur plusieurs cœurs de calculs. En effet, doubler le nombre de cœurs de calculs en divisant la fréquence par deux permet d'obtenir la même puissance de calculs qu'un seul cœur tout en réduisant la consommation d'énergie par quatre [24]. L'approche parallèle est donc prédominante aujourd'hui pour accroître la puissance et la programmation parallèle devient alors indispensable afin d'utiliser cette puissance de calcul.

En plus de ces processeurs multi-cœurs, il existe un grand nombre d'architectures de machines parallèles. Il y a ainsi des architectures « classiques » à mémoires partagées (comme les Symmetric Multi-Processors (SMP)) ou à mémoires distribuées (les clusters, les grilles de calculs ou encore les machines massivement parallèles) mais il existe également des architectures plus spécifiques pouvant être utilisées pour faire du calcul scientifique. Il y a notamment les accélérateurs matériels que sont les Application Specific Integrated Circuit (ASICs) et les Field Programmable Gate Arrays (FPGAs) qui produisent des circuits spécialisés pour un type de calcul [27, 69, 138]. Le calcul scientifique sur cartes graphiques est apparu récemment [114], il s'agit du GPGPU (General Purpose Graphical Processing Unit). L'intérêt des cartes graphiques est qu'elles sont composées d'un grand nombre de processeurs élémentaires et qu'elles proposent une grande puissance de calcul (518 Gigaflops pour une carte Nvidia Tesla C870).

La programmation de ces différentes architectures parallèles se fait via un certain nombre de langages parallèles : OpenMP pour les machines à mémoires partagées, les bibliothèques de communications pour les machines à mémoires distribuées, la bibliothèque CUDA pour les cartes graphiques en passant par les langages de descriptions d'architecture pour les accélérateurs. La bonne utilisation de tous ces langages et de leurs détails d'implémentation est complexe. En plus de cette connaissance des langages, la programmation parallèle est réputée pour être difficile par rapport à la programmation séquentielle, le style de programmation étant différent entre les deux. C'est pourquoi il est nécessaire de simplifier la programmation parallèle. Différentes voies sont explorées afin de réaliser cette simplification : la conception de nouveaux langages parallèles, mais également la parallélisation automatique. Ces deux voies seront respectivement présentées dans les paragraphes 2.1 et 2.2.

Parallèlement à ce développement des architectures et de la programmation parallèle, l'ingénierie logicielle propose des solutions afin de développer, de maintenir et de documenter les applications à partir de spécification de haut niveau : il s'agit de l'ingénierie dirigée par les modèles (IDM). La maintenance de code parallèle par des non-spécialistes étant également un problème, nous étudierons l'IDM au paragraphe 2.3.

Enfin, nous présenterons l'environnement Gaspard, développé au sein de l'équipe projet INRIA DaRT, et dans lequel seront implémentés les travaux de cette thèse (§ 2.4).

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

2.1 La conception de nouveaux langages parallèles

L'expansion de la programmation parallèle auprès des utilisateurs ayant besoin de puissance ne peut se faire que par une simplification de celle-ci. Une des voies pour la simplification de la programmation parallèle passe par la création de nouveaux langages permettant d'exprimer, facilement et sans détails techniques liés à l'architecture de la machine cible, le parallélisme de l'application. Dans cette optique, un certain nombre de nouveaux langages apparaissent et le point de départ du design de ces langages est une réflexion sur ce que l'on peut abstraire afin d'obtenir un langage parallèle plus simple tout en permettant l'obtention de bonnes performances [32, 33]. Les principaux nouveaux langages parallèles sont développés dans le cadre du projet DARPA HPCS (High Productivity Computing Systems) qui vise à multiplier par 10 la productivité des langages à l'horizon 2010. Dans le cadre de ce projet, la productivité est définie comme une combinaison entre la performance, la facilité de programmation, la portabilité et le côté robuste du langage. Nous allons voir ici les différents points pouvant simplifier la programmation parallèle. Ensuite, nous étudierons les langages parallèles existants ainsi que leurs avantages et inconvénients. Un récapitulatif des différents langages conclura ce paragraphe et dégagera les langages prometteurs pour simplifier la programmation parallèle.

2.1.1 Propriétés pouvant simplifier la programmation parallèle

Dans le cadre de la conception des nouveaux langages parallèles, des propriétés ont été identifiées comme permettant de simplifier l'utilisation de la programmation parallèle par les développeurs. Ces propriétés, principalement identifiées dans les travaux de Bradford L. Chamberlain, devraient donc être présentes dans les nouveaux langages parallèles. Quatre des propriétés présentées ci-après sont très importantes (la vue globale de l'application, l'expression de tout type de parallélisme, les communications simples et la séparation des algorithmes et de l'implémentation parallèle). Les autres permettraient aux utilisateurs de retrouver leurs habitudes ou au compilateur de réaliser certaines optimisations.

2.1.1.1 Vue globale de l'application

La manière de programmer des applications parallèles entraîne des types de vues différents selon le style de programmation. Selon le style de vue que le développeur a sur l'application, le développement et la compréhension de celle-ci sont plus ou moins compliqués. Deux types de vues de programmations peuvent être identifiés :

- Vue locale
- Vue globale

Une vue est appelée locale (mais également fragmentée ou partitionnée) si les programmeurs doivent exprimer leurs algorithmes en décomposant explicitement les données et les structures de contrôle de l'application en fonction des processeurs sur lesquels l'application va s'exécuter. Les programmes réalisés en SPMD sont tous à vue locale (le code dépendant du numéro du processeur). Une vue globale exprime tout l'algorithme sans découpage dû au parallélisme. Un programme ayant une vue globale est donc plus facilement compréhensible que la même application ayant une vue locale. Le nombre de lignes de code à écrire, dans le cas d'un programme utilisant une vue globale, est généralement plus petit que du code parallèle ayant une vue locale (entre 2 et 8 moins de lignes [34]). La vue globale peut

Thèse de Julien Taillard, Lille 1, 2009

également prévenir des bogues en évitant un certain nombre d'erreurs de programmation parallèle puisque la programmation est simplifiée [33]. La maintenance d'un tel code est plus simple puisque le code est plus compréhensible. Les deux types de vue sont illustrés sur la même application dans la figure 2.1. La vue globale est le code à gauche de la figure, on peut y voir l'expression d'une boucle parallèle en OpenMP. La vue locale est donnée à droite de la figure, on peut y voir une boucle parallèle écrite selon le numéro du processeur. Chaque processeur exécute ce code et selon le numéro du processeur n'effectue pas les mêmes index de la boucle. La vue globale est bien plus simple à comprendre et à développer puisque la boucle s'écrit simplement alors que dans la vue fractionnée, il faut être capable de répartir les répétitions de la boucle sur les processeurs (ici sur quatre processeurs).

```
!$OMP PARALLEL DO
do i=1,12
   !calcul a realiser
end do
!$OMP END PARALLEL DO
```

```
!p: numero du processeur
begin_index=4*p+1
end_index=4*(p+1)
do i=begin_index,end_index
   !calcul a realiser
end do
```

FIG. 2.1: Illustration d'une vue globale (à gauche) et d'une vue fractionnée de la même application.

2.1.1.2 Expression de tout type de parallélisme

L'expression de tout type de parallélisme est un point important d'un langage parallèle afin que le langage permette de programmer toutes les applications. Le parallélisme de données et de tâches doit donc être supporté, mais également le parallélisme imbriqué (du parallélisme de données dans du parallélisme de tâches par exemple). Ce parallélisme imbriqué permet au langage de s'adapter aux différentes architectures. Il peut être utilisé afin de l'associer aux différentes hiérarchies de la machine cible et, ainsi, d'utiliser au maximum les ressources de la machine. Dans le cadre de l'utilisation d'un cluster de SMPs, le parallélisme de tâches peut répartir les calculs sur les SMPs et le parallélisme de données répartit lui les calculs entre les différents processeurs d'un SMP. Les langages doivent également fournir des méthodes de synchronisations des processus si la synchronisation n'est pas déduite des constructions parallèles.

2.1.1.3 Séparation Algorithme/Implémentation

Lors de l'implémentation d'un algorithme dans un langage, cette implémentation est souvent guidée par le langage que l'on utilise. Ainsi, le même algorithme n'est pas implémenté de la même façon dans des langages différents afin d'utiliser les spécificités de chacun d'entre eux. Ces spécificités sont souvent liées aux stockages des données. Ainsi selon la façon dont sont linéarisés les tableaux, l'implémentation d'un algorithme tel que la multiplication de matrice doit être modifiée afin de parcourir les tableaux selon leur linéarisation (par ligne ou par colonne) afin d'obtenir des performances comme cela est illustré dans la figure 2.2. Il en est de même selon le type de matrice (dense ou creuse) que l'on utilise, cela modifie complètement l'implémentation de l'algorithme. Très peu de langage séquentiel supporte

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

une telle séparation, Matlab propose une telle séparation entre le stockage des données et l'algorithme [60].

Dans le cadre de l'étude des langages parallèles, nous nous intéresserons principalement à la séparation de l'algorithme et de l'implémentation de la distribution parallèle des tâches. La liaison des deux entraînant une complexité dans le code qui a pour effet que généralement peu de distributions différentes sont testées, cela nécessitant de trop nombreuses modifications dans le code.

```
1 Pour i Allant_de 1 A n
   Pour j Allant_de 1 A n
    tab1[i][j]=tab2[i][j]*tab3[i][j]
   FinPour
5 FinPour
```

```
1 Pour j Allant_de 1 A n
   Pour i Allant_de 1 A n
    tab1[i][j]=tab2[i][j]*tab3[i][j]
   FinPour
5 FinPour
```

FIG. 2.2: Illustration de l'adaptation de l'algorithme à la façon dont sont stockés les tableaux en mémoires.

2.1.1.4 Communications simples

Dans les langages parallèles, les communications entre processeurs peuvent être gérées différemment selon les langages. Ainsi, les communications peuvent être complètement cachées et gérées par le compilateur, mais elles peuvent être également visibles tout en restant à un haut niveau d'abstraction ou encore être réalisées manuellement à l'aide de bibliothèques de communications. La gestion des communications explicites, via des fonctions de bibliothèques, est un point difficile de la programmation parallèle. En effet, un mauvais recouvrement des communications par des calculs offre des performances moindres.

L'une des motivations de l'utilisation d'un langage de haut niveau pour le parallélisme est d'épargner les utilisateurs des détails d'implémentation des langages et notamment de ce qui est lié aux architectures des machines. Les communications étant difficiles, il est important qu'un langage parallèle se voulant simple propose une abstraction des communications.

2.1.1.5 Des types de données standard

Les codes de calculs scientifiques reposent sur un grand nombre de types de données dépendant des problèmes à traiter tels que les tableaux multidimensionnels, les tableaux creux, les ensembles, les graphes ou les tables de hachages. Ces types de données n'étant pas standard dans la plupart des langages, chaque utilisateur définit ses propres types de données. La maintenance des codes est donc liée à la compréhension des types de données utilisés par les concepteurs originaux. C'est pourquoi il est intéressant qu'un langage parallèle propose d'avoir un standard de type de données. En plus d'éviter à l'utilisateur d'avoir à créer son propre type de données, cela permettrait également aux compilateurs de réaliser des optimisations sur l'accès à ces types de données définies par le langage ou dans des bibliothèques. Ces optimisations ne pourraient pas être réalisées sur des types de données réalisés par l'utilisateur puisqu'il faut comprendre la structure du type de données pour permettre une optimisation. Des types de données standards dans un langage permettraient une compréhension aisée et des optimisations par les compilateurs.

Thèse de Julien Taillard, Lille 1, 2009

2.1.1.6 Concepts de programmations de haut niveau

Il est important qu'un programmeur retrouve rapidement ses habitudes de programmations dans un nouveau langage. C'est pourquoi, afin de faciliter l'adaptation à un langage parallèle, celui-ci devrait proposer des mécanismes de programmations de haut niveau des langages actuellement utilisés pour la programmation séquentielle. Ainsi, il serait intéressant qu'un langage de parallélisme propose ces mécanismes de haut niveau que sont la programmation orientée objet, la programmation par aspect et la programmation générique.

2.1.1.7 Performance

L'obtention de performances est la raison d'être du parallélisme. Un langage parallèle de haut niveau doit donc avoir de bonnes performances lors de son exécution. D'après Ngo [100], le modèle d'exécution d'un langage parallèle doit être clair afin que ce langage fournisse des performances. Un modèle d'exécution transparent et compréhensible aide l'utilisateur à écrire son code pour la performance. En effet, l'utilisateur connaissant ce modèle d'exécution saura comment écrire son code pour l'optimiser. Ce modèle d'exécution doit être lié au langage et non au compilateur afin que les codes soient performants sur les différentes architectures. Ce fut, d'après Ngo, un manque du langage HPF les compilateurs ne garantissant pas de prendre en compte les directives de parallélisation. Ce point bien qu'important pour un langage parallèle ne sera pas mis en avant durant l'étude des langages parallèles. La performance dépendant de la machine d'exécution ainsi que du compilateur.

2.1.1.8 Portabilité et inter-opérabilité

Ces deux points sont importants afin de favoriser l'extension du langage parallèle. Un langage portable sur tout type de systèmes et d'architectures matérielles est important. Il ne faut pas à avoir à réécrire ou à modifier le programme pour l'adapter à une nouvelle architecture. L'inter-opérabilité avec d'autres langages va également favoriser l'utilisation d'un langage en permettant de réutiliser des bibliothèques (notamment les bibliothèques de calculs scientifiques) et des codes déjà existants.

2.1.2 Les langages parallèles

Afin d'exploiter les différents types de machines parallèles existantes, il existe un certain nombre de langages parallèles[1] chacun ayant ces particularités. L'acceptation d'un langage parallèle est basée sur l'acceptation du marché qui dépend de la facilité de programmation et des performances du langage, mais également de la disponibilité d'architecture matérielle permettant d'exploiter pleinement les possibilités du langage [28]. Dans le cadre de la présentation des différents langages de programmation, nous allons utiliser la classification présentée par Chamberlain et al. dans [32]. Il y a tout d'abord les bibliothèques de communications entre processus, puis les langages permettant d'avoir un espace d'adressage global puis d'autres langages plus spécialisés.

[1] Par abus de langages, les bibliothèques de communications telles que MPI seront également nommées langages parallèles tant leurs utilisations dans un programme modifient le style de programmation

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

2.1.3 Bibliothèques de communications

Les bibliothèques de communications constituent la première famille de langages parallèles. Ces bibliothèques permettent aux processus parallèles de communiquer afin de s'échanger des données et de se synchroniser. Le modèle d'exécution principalement utilisé pour programmer en utilisant ces bibliothèques est le modèle SPMD. Chaque processus exécutant le même code paramétré par le numéro du processus. On a donc une vue locale de l'application. La programmation utilisant une bibliothèque de communications est considérée comme étant de bas niveau puisque toutes les communications doivent être explicitement gérées. Les propriétés de ces bibliothèques de communications sont celles des langages supportés par ces bibliothèques (Fortran, C et C++ principalement). L'expression du parallélisme de données et de tâches est réalisable. Par contre, le modèle d'exécution SPMD étant dépendant du numéro du processeur, le parallélisme imbriqué n'est pas directement exprimable. Le moyen d'exprimer le parallélisme imbriqué est de mixer les styles de programmations des bibliothèques de communications avec un autre langage (les bibliothèques de communications sont souvent mixées avec le langage OpenMP). Ces bibliothèques utilisent comme base des langages de bas niveau, c'est pourquoi la séparation algorithme/implémentation, les types de données standard et les concepts de programmation de haut niveau ne sont pas disponibles (le C++ propose tout de même la programmation orientée objet et des types de bases). Les bibliothèques de communications sont portées sur un grand nombre d'architectures, les codes les utilisant sont donc portables.

2.1.3.1 Message Passing Interface (MPI)

MPI est le langage de programmation parallèle le plus utilisé actuellement. Son succès vient de son standard clair, d'un grand nombre d'implémentations libres disponibles (tels que MPICH [65] ou Open MPI [64]), d'une bonne portabilité ainsi que de bonnes performances.

Cette librairie de communication est disponible en Fortran, en C et en C++ avec MPI-2 [93]. MPI propose un grand nombre de fonctions de communications (que ce soit point à point ou communication globale). La connaissance de toutes ces fonctions se révèle assez complexe et en général seul un nombre restreint de ces fonctions est utilisé.

2.1.3.2 Parallel Virtual Machine (PVM)

PVM [58] est également une bibliothèque de communications qui fut développée avant l'apparition MPI mais fut supplanté par celle-ci. PVM est disponible pour C, C++ et Fortran.

Les deux bibliothèques de communications présentées précédemment sont les plus connues et utilisées directement par les utilisateurs. Néanmoins, il existe d'autres bibliothèques de communications moins connues que les deux précédentes telles que SHMEM [11], ARMCI [101] et GASNet [18]. Celles-ci sont principalement utilisées en tant que cible de langages parallèles de haut niveau et par les compilateurs.

2.1.4 Langages à adressage global partitionné

Les langages à adressage global partitionné (Partitioned Global Adress Space (PGAS)) sont basés sur une abstraction de la distribution de la mémoire. La mémoire globalement adressable est partitionnée logiquement entre les processeurs afin que chacun ait une portion

locale de la mémoire. Ces langages se basent sur le fait que le modèle de programmation à mémoire partagée est plus simple que celui à mémoire distribuée. L'abstraction de la distribution de la mémoire entraîne une simplification des communications inter-processus, les communications réellement effectuées étant gérées par les compilateurs de langages. Bien qu'il y ait une montée du niveau d'abstraction entre ces langages et les bibliothèques de communications, le style de programmation de ces langages est toujours le modèle SPMD. On obtient alors une vue locale de l'application. Ces langages étant des extensions de langages de bas niveau, les nouveaux concepts de programmations, ainsi que la séparation des algorithmes et de l'implémentation ne sont pas présents (programmation SPMD). L'extension de langages bien connus des développeurs entraîne que les utilisateurs n'ont que quelques concepts à apprendre afin de faire du parallélisme. L'expression du parallélisme imbriqué est limitée par la programmation en SPMD. Ces langages qui reposent sur la librairie de communication de bas niveau ARMCI sont portables sur toutes les architectures supportées par cette bibliothèque de communication.

2.1.4.1 Co-Array Fortran

Co-Array Fortran [103] est un langage construit sur la norme Fortran pour répondre à la question « Que faut il ajouter à Fortran pour en faire un langage parallèle robuste et efficace ? ». Ce langage est une extension syntaxique de la grammaire Fortran permettant d'exprimer les deux concepts fondamentaux du parallélisme : la distribution des tâches et la distribution des données. Le parallélisme est géré par la programmation en SPMD. Le programme est donc répliqué sur un nombre fixé de processus (chaque réplication est appelée une *image*) et chacun de ces processus possède son propre ensemble de données. Une variable déclarée classiquement en Fortran est considérée comme privée et non accessible aux autres images. L'extension « co-array » permet aux programmeurs de gérer la distribution des données entre les processus en désignant des variables comme étant accessibles aux autres images. En effet, la déclaration d'une variable en « co-array » provoque l'affectation de la variable dans chacune des images. Une image pouvant accéder (en lecture comme en écriture) à sa copie locale, mais également aux copies des autres images via une notation particulière. L'accès aux « co-array » d'une variable se fait en indiquant dans quelle image on veut accéder à la variable. Les communications sont donc simplifiées (pas d'envoi/réception à gérer), mais encore explicites. Ce langage est aujourd'hui inclus dans la norme Fortran 2008.

2.1.4.2 Unified Parallel C

Unified Parallel C [28] (UPC) est une extension du langage C afin d'obtenir un langage PGAS. Un faible nombre de concepts sont ajoutés au langage C pour cela. Le principal concept introduit permet de spécifier quelles sont les données partagées entre les processus et comment sont distribuées celles-ci entre les processus. La gestion des communications est complètement dissimulée par le langage. En effet, une assignation simple telle que a=b peut être vue comme locale si les deux variables sont locales, mais également provoquer des communications si les variables sont distantes. UPC propose également une extension des boucles en permettant de spécifier les boucles parallèles et comment les itérations d'une boucle parallèle sont distribuées entre les processus selon la localité des données manipulées.

2.1.5 Global Arrays

Global Arrays [102] (GA) est une bibliothèque proposant un environnement de programmation à mémoire partagée dans le cadre des tableaux distribués (les *global arrays*). GA propose des fonctions gérant les structures de données correspondant aux tableaux distribués et effectuant des calculs sur ces tableaux. Un ensemble de primitives issues des bibliothèques de calcul scientifiques sont ainsi disponibles (comme la multiplication de matrices ou la résolution d'un système linéaire). À l'aide de ces primitives gérant les distributions, GA propose une vue globale de l'application permettant le parallélisme de tâches et de données. La séparation algorithme/implémentation est disponible dans le cas de la distribution puisque celle-ci est gérée par les primitives. Les communications sont explicites pour prendre et mettre des données dans les tableaux partagés. Cette bibliothèque est disponible pour les langages C, C++, Fortran et Python. L'un des points qui rend difficile l'utilisation de cette bibliothèque est le nombre de primitives à connaître et le fait qu'il est impossible de créer ses propres structures de données distribuées.

2.1.6 OpenMP

OpenMP [113] est un langage parallèle développé par un consortium (l'OpenMP Architecture Review Board) composé des principaux industriels impliqués dans ce standard. OpenMP est un langage conçu pour les machines à mémoires partagées. Il est constitué de directives de compilations exprimant le parallélisme d'une application et de quelques fonctions permettant notamment de connaître le nombre de processus légers (threads) sur lesquels est exécutée l'application et de connaître également le numéro de processus. OpenMP est disponible pour les langages C et Fortran. Les directives de compilations s'introduisent dans le code sous forme de commentaires en Fortran et sous forme de pragma en C. La distribution des tâches sur les threads est effectuée par le compilateur, il existe quelques directives indiquant une préférence, mais on ne peut pas spécifier de placement précis des calculs avec les directives. Ces directives sont un des points forts du langage OpenMP puisqu'elles permettent de rendre parallèle un programme séquentiel en y ajoutant simplement ces directives sans changer le code de l'application. Lors de l'utilisation des directives de compilations, OpenMP propose une vue globale de l'application. OpenMP permet également d'écrire du code SPMD en utilisant la seule directive indiquant les portions parallèles et utilisant le numéro du processus pour répartir les calculs, on obtient alors une vue locale de l'application comme dans tout code SPMD. OpenMP permet de spécifier tout type de parallélisme puisqu'il est possible d'imbriquer les directives de compilations. Lors de l'utilisation des directives, OpenMP propose une séparation algorithme/implémentation du parallélisme, ce qui n'est pas le cas lors de la programmation en SPMD. OpenMP se basant sur les langages C et Fortran, il a les défauts de ces langages : pas ou peu de concepts de programmation de haut niveau, pas de structures de données standard. Le principal inconvénient du langage OpenMP est la portabilité. Le langage, étant spécialisé pour les architectures à mémoires partagées, ne fonctionne principalement que sur ce type d'architecture. Des solutions industrielles (ClusterOMP d'Intel) ou académiques (ParADE [80]) proposent la conversion du code OpenMP vers des codes utilisant des bibliothèques de communications. Ces solutions sont néanmoins peu répandues.

Afin d'utiliser au mieux les architectures des supercalculateurs, une tendance est de mixer les langages MPI et OpenMP afin de gérer les différents niveaux de hiérarchies. Dans le cas

Thèse de Julien Taillard, Lille 1, 2009

d'un cluster de machines à mémoires partagées, MPI est utilisé pour la communication entre machines alors qu'OpenMP l'est pour faire les calculs parallèles sur chacun des noeuds à mémoires partagées. Les résultats [67, 47, 29] de ce type de programmation dépendent de l'architecture cible, mais également de l'application.

2.1.7 High Performance Fortran (HPF)

High Performance Fortran [70] est un langage data-parallèle proposant une vue globale de l'application. Ce langage est conçu comme une extension du langage Fortran. Le parallélisme en HPF est introduit à l'aide de directives de compilation permettant de spécifier la distribution des données entre processeurs. La distribution des calculs suit celle des données. En effet, la règle de « owner compute rule » indique que c'est au processeur possédant les données à traiter de réaliser le calcul. HPF ne propose pas de parallélisme de tâches, mais permet une séparation algorithme/implémentation des distributions puisqu'il suffit de modifier celles-ci sans avoir à modifier le code de l'algorithme. Les communications entre processeurs sont gérées par le compilateur. L'un des inconvénients identifiés dans le langage HPF est son manque de modèle d'exécution clairement défini, celui-ci variant selon les compilateurs.

2.1.8 ZPL

ZPL [31, 127] est un langage data-parallèle de haut niveau développé à l'université de Washington. Ce langage propose une vue globale de l'application dans laquelle le parallélisme est implicite et les communications sont explicites, mais simplifiées par rapport aux bibliothèques de communications. Le parallélisme implicite dépend des *régions* qui permettent de représenter les tableaux parallèles ainsi que les indices des tableaux sur lesquels les opérations sur les tableaux doivent être effectuées. Les communications entre régions sont explicites et spécifient des translations sur les tableaux parallèles nécessaires au calcul. Le compilateur transforme ensuite le code ZPL en code SPMD C utilisant pour les communications les librairies MPI, PVM ou SHMEM selon les choix de l'utilisateur. ZPL propose une forme d'abstraction du type de données, mais celle-ci ne s'applique uniquement qu'aux matrices creuses [35]. Il y a une séparation algorithme/implémentation puisque la distribution des calculs n'est pas explicite et est déduite lors de la compilation. La portabilité du langage dépend des bibliothèques de communications sur lesquelles il repose. Le langage, data-parallèle, ne propose pas de parallélisme de tâches.

2.1.9 Les nouveaux langages parallèles

Dans le cadre du projet DARPA HPCS dont le but est de développer des langages productifs permettant l'utilisation « simple » du parallélisme, plusieurs langages parallèles sont en développement : Chapel, Fortress et X10. Dans ce paragraphe, nous allons présenter les grandes lignes de chacun de ces langages qui seront, peut-être, les langages les plus utilisés demain.

Bien que développés par des équipes différentes, ces langages sont basés sur des concepts communs [126]. Ces trois langages se basent sur un haut niveau d'expression permettant de s'abstraire des détails d'implémentations liés à l'architecture de la machine sur laquelle va être exécuté le programme. Le travail d'optimisation permettant d'obtenir des performances sur l'architecture d'exécution revient aux compilateurs et aux environnements d'exécutions.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Dans ces trois langages, un processus de contrôle permet de lancer les autres processus. La mémoire est globalement adressable, afin d'avoir une vue globale de l'application, et des techniques permettant la distribution des données sont présentes. Les communications sont implicites et devront être gérées par les compilateurs. Ce sont des langages de haut niveau basés sur les langages orientés objets et sur la programmation générique. L'équilibrage de charge entre les processus devrait être géré par les langages, mais cela est encore un axe de recherche pour le moment.

2.1.9.1 Chapel

Chapel [26] est un langage développé par la société Cray Inc. C'est un langage proposant une vue globale des applications et qui supporte la programmation impérative ainsi que la programmation orientée objet. Le langage supporte tout type de parallélisme. Le parallélisme de tâches est géré par des threads concurrents alors que le parallélisme de données est géré par les itérations de boucles réalisées en parallèle. Ces deux types de constructions peuvent être imbriqués. Le langage propose une abstraction des types de données sur les tableaux (denses, creux) et sur les objets avec la programmation orientée objet. La distribution des tâches et données est réalisée sur des *locales* qui représentent des éléments d'architectures matérielles.

2.1.9.2 Fortress

Fortress [7] est un langage développé par Sun Microsystem. L'un des buts de Fortress est de détrôner Fortran dans le calcul mathématique. Dans ce but, Fortress propose notamment d'utiliser la syntaxe mathématique courante pour écrire les algorithmes. Fortress est un langage orienté bibliothèques. Un petit noyau du langage est étendu par des bibliothèques. Fortress est un langage parallèle par défaut. Les boucles sont parallèles par défaut, une boucle séquentielle devant être explicitement spécifiée. Le parallélisme et la distribution des boucles parallèles sont gérés par les *générateurs*. Ceux-ci expriment comment les itérations d'une boucle sont exécutées par les threads. Le changement de distribution revient donc à changer de *générateur*. Le parallélisme de tâches est réalisé en créant dynamiquement des threads que l'on peut placer manuellement sur différentes *régions* qui correspondent à un ensemble d'éléments matériels. Fortress propose une vue globale de l'application ayant une séparation algorithme/implémentation grâce aux *générateurs*. Une abstraction des types de données est également disponible, un algorithme écrit sur des matrices denses s'appliquant également sur des matrices creuses. La version 1.0 de la spécification de Fortress est disponible depuis Mai 2008.

2.1.9.3 X10

X10 [36] est un langage développé par IBM. Il est basé sur le langage Java et propose la programmation orientée objet bien que la programmation impérative soit également supportée. X10 reprend la base de Java en changeant la manière dont sont gérés les tableaux ainsi que la concurrence. Les tableaux X10 sont multidimensionnels et peuvent être distribués. Le parallélisme est implicite, comme dans le langage ZPL, et dépend des régions. Ce sont ensuite ces régions qui sont distribuées sur les différents endroits d'exécutions (les *places*). Le langage propose une vue globale de programmation dans laquelle les communications sont gérées par le compilateur du langage. Tous les types de parallélisme peuvent être exprimés,

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

les constructions parallèles pouvant s'imbriquer et créer le nombre de threads nécessaire à la bonne exécution de ce parallélisme. La séparation algorithme/implémentation est respectée grâce au mécanisme de distribution du langage. Celle-ci peut être changée sans modifier tout l'algorithme. Tous les concepts de programmations de haut niveau associés au langage Java sont disponibles dans ce langage.

2.1.10 Récapitulatif des langages parallèles

Récapitulons les propriétés des langages parallèles étudiés précédemment afin d'en distinguer les candidats permettant une simplification de la programmation parallèle. Nous nous intéresserons principalement aux quatre propriétés les plus importantes pour les langages parallèles que sont le type de vue proposé, l'expression du parallélisme, les communications simples et la séparation algorithme/implémentation de la distribution. Le tableau 2.1 récapitule ces propriétés sur les différents langages. La considération du type de vue proposée par les langages permet d'éliminer tous les langages s'utilisant en SPMD. Ainsi, les bibliothèques de communications (MPI, PVM), les langages à adressage global partitionné (Co Array Fortran et Unified Parallel C) ainsi qu'une partie du langage OpenMP ne sont plus en course pour simplifier la programmation parallèle. La séparation implémentation - distribution des tâches suit le type de vue de l'application. Un langage ayant une vue locale ne permettant pas une séparation de ces concepts. Il en est de même pour les communications qui sont implicites ou grandement simplifiées dans les langages ayant une vue globale. HPF et ZPL sont des langages data-parallèles, ils ne proposent donc pas tout type de parallélisme. OpenMP, quant à lui, est limité au niveau de la portabilité puisque permettant la programmation des machines à mémoires partagées. Les langages du programme DARPA HPCS, construits en ayant connaissance de ces propriétés, sont finalement les seuls bons candidats pour simplifier la programmation parallèle. Le fait que ceux-ci soient en développement entraîne qu'il n'est pas possible de prédire si l'un d'entre eux sera utilisé et performant. Il n'y a actuellement pas de langages disponibles permettant d'obtenir une simplification des développements et de la maintenance des applications parallèles. D'autres approches doivent donc être envisagées afin de permettre l'exploitation des machines parallèles actuelles.

2.2 Parallélisation automatique et génération de code parallèle

La génération de code parallèle ou parallélisation automatique est un sujet de recherche important depuis un certain nombre d'années. Considérant que le calcul parallèle est le seul moyen d'obtenir beaucoup de puissance de calcul, que la programmation parallèle est difficile (compte tenu qu'il faut actuellement gérer les spécificités de la machine pour obtenir des performances) et qu'il existe un grand nombre d'applications séquentielles déjà fonctionnelles, un effort de recherche a été mené sur la parallélisation automatique permettant de passer d'une application séquentielle vers une application parallèle. C'est le but principal de la parallélisation automatique ou génération de code parallèle (section 2.2.1). Un autre type d'approche envisage de passer d'un langage parallèle à un autre langage parallèle afin d'obtenir de meilleures performances sur des architectures particulières (section 2.2.2).

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Langages	*Vue*	*Séparation Algorithme Implémentation*	*Parallélisme*	*Communication*
MPI	locale	non	tâches et données	manuelle
PVM	locale	non	tâches et données	manuelle
CAF	locale	non	tâches et données	manuelle++
UPC	locale	non	tâches et données	invisible
GA	globale	non	tâches et données	manuelle++
OpenMP	les deux	selon type	imbriqués	-
HPF	globale	directives	données	invisible
ZPL	globale	régions	données	visible
Fortress	globale	générateurs	imbriqués	invisible
Chapel	globale	régions	imbriqués	visible
X10	globale	oui	imbriqués	invisible

TAB. 2.1: Tableau récapitulatif des propriétés importantes d'un langage parallèle de haut niveau simple. La colonne *Vue* indique si le type de vue proposé par le langage (locale, globale ou les deux pour OpenMP). La seconde colonne *Séparation Algorithme Implémentation* indique si le langage propose la séparation placement/implémentation et comment si celle-ci est disponible (via des directives de compilations ou des concepts spéciaux des langages). La troisième colonne indique le type de parallélisme pouvant être exprimé dans la langage (tâches, données, les deux ou imbriqués). Enfin, la quatrième colonne indique comment sont réalisées les communications (manuelle, manuelle++ (exprimée, mais simplifiés par rapport à manuelle), invisible, visible (lors de l'écriture du code, on sait ce qui va provoquer des communications mais elles ne sont pas écrites explicitement))

.

2.2.1 Génération de code parallèle à partir de programme séquentiel

Les principaux travaux de parallélisation ont été menés dans le cadre de la transformation d'un code séquentiel vers un code parallèle. La majorité de ces travaux sont basés sur la parallélisation des nids de boucles [84]. Leurs importances dans les applications sont clairement identifiées : le temps d'exécution passé dans ce type de boucles représente une grande partie du temps d'exécution total. Il est donc important de paralléliser de telles boucles. La parallélisation automatique commence toujours par une phase d'analyse des dépendances de données. Cette analyse permet de créer un graphe de tâches et de connaître les dépendances entre itérations d'un nid de boucle. Ces dépendances sont régulièrement représentées sous une forme mathématique à partir de laquelle des opérations peuvent être appliquées afin de déterminer comment le parallélisme peut intervenir avec de telles dépendances. Les polyèdres sont une des représentations mathématiques les plus utilisées dans ce domaine. Lors de la parallélisation, deux approches sont possibles : soit l'application parallélisant le code effectue tout le travail sans interaction avec l'utilisateur, soit l'application interagit avec l'utilisateur. Certains considèrent que l'utilisateur ne doit pas effectuer de choix lors de la parallélisation (celui-ci étant source d'erreurs), d'autres pensent que c'est grâce aux choix de celui-ci que la performance peut être obtenue. Le but principal pour la création de tels outils d'aide à la parallélisation de code séquentiel est d'embarquer l'expertise nécessaire à une parallélisation efficace dans des algorithmes automatiques. Cela est plus rapide à faire

Thèse de Julien Taillard, Lille 1, 2009

que de laisser l'expert paralléliser manuellement chacun des codes que l'on veut paralléliser. Les outils de parallélisation automatiques se concentrent principalement sur les langages les plus utilisés, c'est à dire OpenMP et MPI.

2.2.1.1 Génération de code OpenMP

Il existe un certain nombre de travaux portant sur la génération de code OpenMP. Le code OpenMP peut être généré de deux façons différentes : du code SPMD dépendant du numéro de thread ou du code séquentiel dans lequel sont insérées les directives OpenMP indiquant les parties parallèles du code. Parmi ces travaux, on peut citer les projets POST [6], CAPO [73].

Le projet POST (Programming with the OpenMP standard) [6] vise à créer un environnement d'aide à la programmation en OpenMP. Il aide ainsi l'utilisateur à savoir quelles boucles peuvent être parallélisées et quelle est la façon de les paralléliser. Il propose également un système d'identification de structures parallèles basé sur des scripts qui identifient les parties potentiellement parallèles et proposent la manière pour paralléliser la structure identifiée. Cela permet aux utilisateurs non spécialistes de bénéficier du savoir-faire des experts en parallélisme qui auront écrit et mis à dispositions ces scripts.

L'outil CAPO [73], basé sur CAPTools, est un outil d'aide à la parallélisation spécialisé dans la génération de code OpenMP. Cet outil limite au maximum les interactions avec les utilisateurs et génère des directives OpenMP dans le code séquentiel.

Certains travaux visent à utiliser différents outils pour réaliser la parallélisation. Ainsi, les travaux présentés dans [71] présentent un environnement de parallélisation interactif basé sur différents outils pour générer de l'OpenMP. Cet environnement regroupe notamment un outil pour l'analyse de dépendance, un outil d'insertion des directives OpenMP (CAPO) et un débugger.

2.2.1.2 Génération de code MPI

La génération de code MPI est un peu plus compliquée que celle de code OpenMP puisque dans ce cas, il faut générer tous les appels à la bibliothèque MPI permettant de gérer les communications entre processus. Le code généré sera du code SPMD, principale façon de programmer avec MPI. Il existe différents travaux générant du code MPI. Ainsi LooPo [37] propose d'utiliser les polyèdres afin de générer le code parallèle à partir du code séquentiel. KAP (Kit of Automatic Parallelization) [122] passe lui par un modèle abstrait du code dans lequel le parallélisme des boucles est extrait. Ensuite, les calculs et données sont décomposés et placés sur les processeurs. Enfin, une analyse de dépendances est réalisée afin d'identifier les accès aux données non locales et ainsi calculer les communications.

D'autres outils de parallélisation automatique ont également été produits pour cibler des langages moins répandus. C'est le cas de Bouclette [20] qui permet de générer du code HPF.

2.2.2 D'un langage parallèle à un autre

Une autre forme de génération de code parallèle est la conversion d'un langage parallèle à un autre ou d'un style de programmation vers un autre. Ce type de génération de code est principalement utilisé pour obtenir de meilleures performances sur des machines particulières. Ainsi Géraud Krawezik et *al.* dans [83] proposent de transformer du code parallèle MPI vers du code parallèle OpenMP afin de mieux utiliser les machines à mémoires partagées. En effet,

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

le code MPI peut être exécuté sur des machines à mémoires partagées, mais ces codes peuvent se révéler moins performants que du code OpenMP. Dans [92], les auteurs observent que la privatisation des tableaux dans du code OpenMP est importante dans les systèmes ccNUMA. C'est pourquoi ils proposent de convertir du code OpenMP classique (avec des directives) vers du code OpenMP SPMD permettant de faire des tableaux partagés, la méthodologie de conversion automatique est également proposée.

2.3 Ingénierie dirigée par les modèles

Alors que la puissance des machines augmente constamment, les utilisateurs attendent des logiciels de plus en plus complexes. Cette complexité des applications rend celles-ci plus difficiles à développer, mais également à maintenir. Cela appelle à une évolution des méthodes de développements de ces logiciels. C'est ainsi que sont apparues des méthodes de modélisation : Merise [129] (1970), SSADM (Structured Systems Analysis and Design Methodology) [50] (1980), UML (Unified Modeling Language) [106] (1995), etc.. Ces méthodes proposent des concepts et outils permettant de décrire le système à concevoir. À chaque étape du cycle de vie d'un système, un ensemble de documents sont ainsi produits afin de permettre à toutes les personnes impliquées dans le développement du logiciel (développeurs, chef de projet, décisionnaire, client) de partager leurs points de vue et leurs connaissances du système. Ces méthodes ont été critiquées pour leur lourdeur et leur manque de souplesse face à l'évolution du système. C'est ainsi que la notion de modèles *contemplatifs* a été introduite par Favre et al. [53]. L'ingénierie dirigée par les modèles (IDM) a été introduite afin de répondre à ces critiques et ne plus avoir que des modèles *contemplatifs*. Celle-ci a pour principal but de rendre les modèles *productifs* : en plus de servir à la compréhension et à la documentation du système informatique, les modèles doivent également servir à produire du code. Il existe de nombreuses variantes de l'IDM tels que *Model-Driven Engineering* (MDE), *Model-Driven Architecture* (MDA), *Model-Driven Development (MDD)*, etc. Seuls les principes de bases de ces différentes approches basées sur les modèles seront présentés dans cette section. Ces approches reposent sur trois concepts principaux : les modèles, les métamodèles et les transformations de modèles.

2.3.1 Les modèles

Les modèles sont la base de la récente méthodologie d'ingénierie dirigée par les modèles, mais le concept de modèle en lui-même est très ancien comme le dit Jean-Marie Favre dans [52]. Ainsi, toute représentation peut être considérée comme un modèle. Un portrait est un modèle de la personne qu'il représente tout comme une peinture est un modèle de ce qu'elle représente. La notion de modèle en informatique est présente depuis les années 70.

Il est important pour débuter de donner une définition de ce qu'est un modèle dans le cadre des systèmes informatiques. Un modèle est une vue abstraite d'un système informatique réalisée selon un (ou plusieurs) point(s) de vue. Pour réaliser un modèle, il est important d'identifier les caractéristiques pertinentes et intéressantes du système afin de les modéliser et ainsi de permettre l'étude du système selon ces caractéristiques. L'IDM repose sur une utilisation intensive des modèles tout au long du processus du développement.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

2.3.1.1 Séparation des aspects

L'un des intérêts principaux de l'utilisation des modèles est leur représentation graphique qui en facilite grandement la lecture et la compréhension. Afin de faciliter encore plus cette lecture, la notion de séparation des aspects a été introduite. En effet, lors du développement ou de l'étude d'un système informatique, les différents intervenants veulent que le modèle réponde à leurs différentes préoccupations. L'un des points importants des modèles est de permettre la séparation les aspects, cela est notamment le cas via la notion de *vue*. Un même système peut être vu selon différents points de vue permettant à chacun de répondre à différentes questions. Les modèles ont donc plusieurs représentations graphiques dans lesquelles seule une partie des caractéristiques du système est représentée dans chacune des représentations. Selon ce que veut voir l'utilisateur, seules certaines caractéristiques doivent être mises en avant. Il peut ainsi y avoir différentes vues du même système : une vue pour la structure, une vue pour les activités, etc.

2.3.2 Les métamodèles

Afin de rentre les modèles *productifs*, il est important que les machines puissent les lire, les comprendre. Comme le dit Jean-Marie Favre dans [53, p. 22] :

« Les modèles contemplatifs sont interprétés par l'homme alors que la préoccupation première de l'informaticien est de produire des artefacts interprétables par la machine »

En l'absence de syntaxe et de sémantique, il existe à peu près autant de modèles possibles pour un unique système que de modeleurs[2]. Il est donc important d'ajouter une sémantique et une syntaxe aux modèles afin qu'ils soient compréhensibles par les machines. C'est le rôle d'un métamodèle. Un modèle fait dans un métamodèle est dit *conforme à* ce métamodèle. Un métamodèle peut être vu comme un « *modèle d'un langage de modélisation* »[51]. Il est possible de faire une analogie entre les métamodèles et les grammaires des langages de programmation. En effet, si l'on compare le métamodèle à une grammaire d'un langage de programmation, alors un modèle conforme à ce métamodèle est équivalent à un programme dans le langage de programmation. Un métamodèle doit rendre possible la modélisation de toutes les caractéristiques pertinentes du type de système à modéliser.

2.3.2.1 Les niveaux de modélisations

Différents niveaux dans la modélisation peuvent être distingués. Ces niveaux représentent des niveaux d'abstractions dans les modèles. Ces niveaux sont présentés ici. Ainsi, le niveau M0 est le niveau le plus bas, il s'agit de l'instance : le système réel. Le niveau M1 correspond aux modèles que l'on peut faire du système réel. Le niveau M2 est le métamodèleutilisé afin de réaliser le modèle (niveau M1). Un métamodèle étant également un modèle, il est conforme à un autre métamodèle et ainsi de suite. Afin de ne pas avoir une infinité de niveaux, le métamodèle du niveau M3 permet de s'autodéfinir. On dit alors que le métamodèle est *méta-circulaire*. Il existe plusieurs métamodèles de niveaux M3 tels que Ecore [49] et MOF [105] (Meta-Object Facility).

La figure 2.3 illustre les différents niveaux et les relations entre ces différents niveaux.

Dans cette figure, le système informatique (au niveau M0) à modéliser est un compte bancaire ayant un numéro de compte et un solde. Au niveau modèle (M1), le concept COMPTE

[2]personnes réalisant des modèles dans ce contexte

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

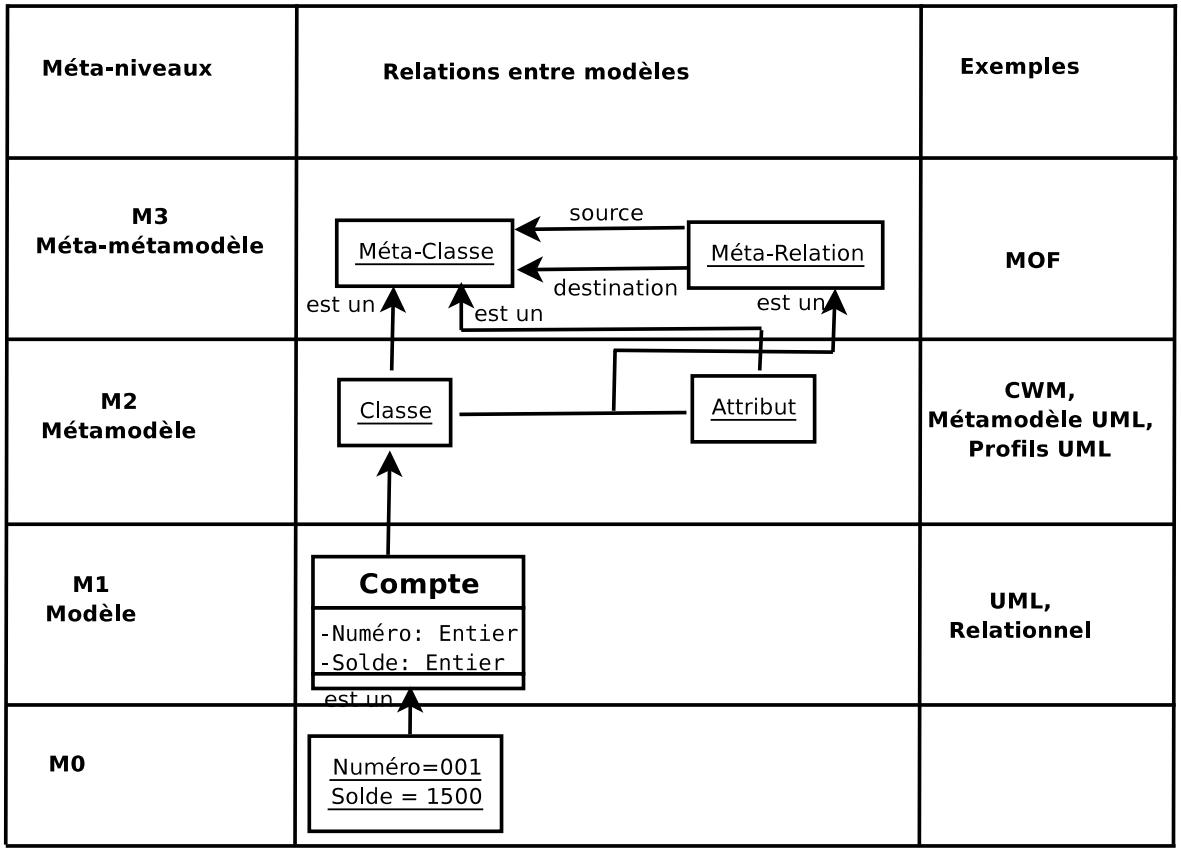


FIG. 2.3: Les différents niveaux de modélisation.

est donc modélisé. Le métamodèle (M2) utilisé ici est un métamodèle de langage objet ayant donc le concept de Classe et d'attribut. Le concept de COMPTE est donc défini comme une classe ayant deux attributs entiers : le numéro et le solde du compte. Ce métamodèle est lui-même défini par un métamodèle méta-circulaire, dans ce cas-ci le MOF.

2.3.2.2 Exemple de métamodèle

Les concepts de base de la création d'un métamodèle vont être présentés dans cette section. Cela permettra d'illustrer clairement ce qu'est un métamodèle mais fournira également au lecteur, les notions nécessaires à la compréhension des différents métamodèles utilisés et présentés au cours de ce manuscrit.

La figure 2.4 illustre un métamodèle générique. Ce métamodèle définit les concepts de MODEL, ELEMENT et CONNECTION. Un MODEL peut contenir aucun ou plusieurs ELEMENTS et aucune ou plusieurs CONNECTIONS. Les liens *connector* et *element* commençant par un losange noir et finissant par une flèche sont des liens de contenance : l'élément sur lequel pointe le losange peut contenir des éléments du type pointé par la flèche. Un ELEMENT possède trois attributs : un nom, un type et une valeur tous trois représentés sous la forme d'une chaîne de caractères. Un ELEMENT peut également contenir d'autres ELEMENTS via le lien de contenance *contains*. Il peut également y avoir des CONNECTIONS entre les éléments. Le concept CONNECTION exprime comment une connexion est réalisée entre deux ELEMENTS.

Les connecteurs nommés *source* et *target* sont des connecteurs exprimant une référence. Le concept CONNECTION possède une référence vers un ELEMENT qui sera source de la connexion et une référence vers un ELEMENT qui sera la cible de la connexion. Les ELEMENTS référencés seront contenus par le MODEL ou d'autres ELEMENTS.

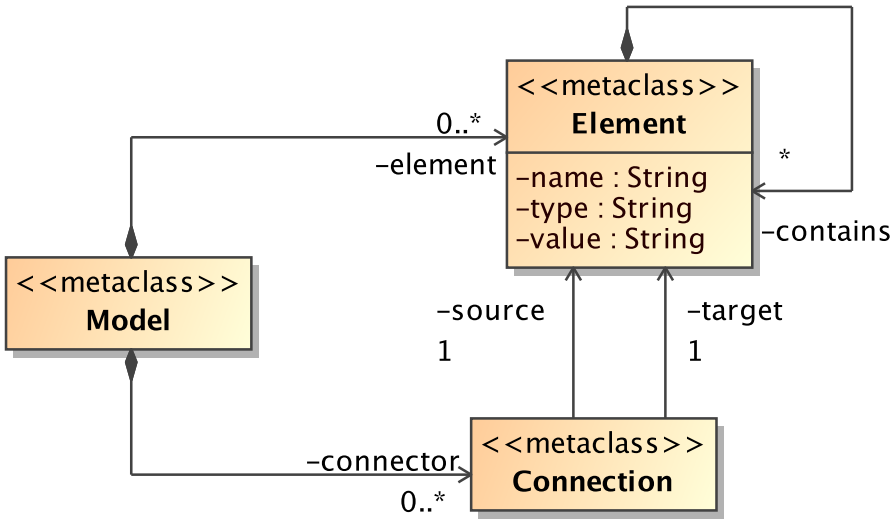


FIG. 2.4: Illustration d'un métamodèle générique.

Ce métamodèle est très générique puisqu'il permet de modéliser un grand nombre de systèmes, mais le faible nombre de concepts proposés fait qu'un modèle conforme à ce métamodèle peut être compris différemment par deux personnes utilisant ce métamodèle dans des cadres différents. La généricité d'un métamodèle n'est donc pas nécessairement un avantage. Les concepts manipulés doivent être clairement identifiés et nommés afin d'éviter ce problème.

2.3.3 Transformations de modèle

Rendre un modèle *productif* entraîne qu'il faut, comme son nom l'indique, produire. Les transformations de modèles permettent cette production. Une transformation prend un ou plusieurs modèles d'entrée (conforment à leurs métamodèles respectifs) et produit un ou plusieurs modèles de sortie (également conforment à leurs métamodèles respectifs). Le passage des modèles sources vers les modèles cibles est décrit à l'aide de *règles de transformations* qui sont exécutées par un outil nommé *moteur de transformation*. Le principe de base des transformations de modèle est illustré dans la figure 2.5 pour le modèle reposant sur le métamodèle de niveau 3 Ecore. On y retrouve le moteur de transformation (noté Transform) qui prend en entrée un ou plusieurs modèles et utilisent les règles (Rules) entre les métamodèles sources et les métamodèles cibles. Le moteur produit en sortie le ou les modèles conformes aux métamodèles de sortie.

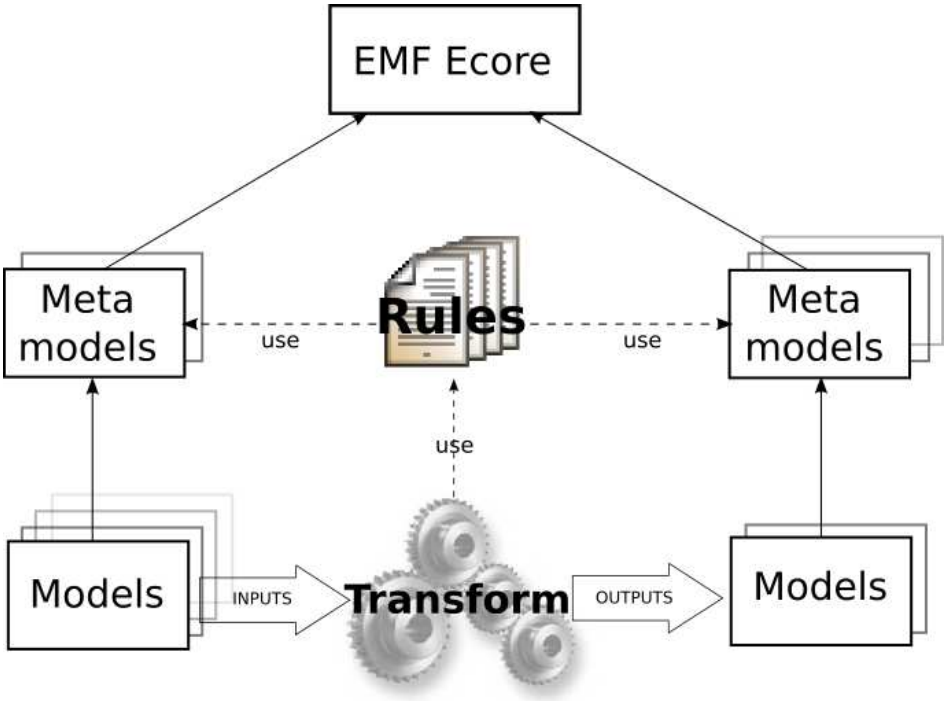


FIG. 2.5: Principe de fonctionnement d'un moteur de transformation.

2.3.3.1 Types de transformations de modèles

Différents types de transformations de modèles peuvent être réalisés [96] : les transformations *endogènes* et les transformations *exogènes*. Une transformation *endogène* est une transformation de modèles utilisant le même métamodèle en source et en cible. Ce type de transformation est principalement utilisé pour restructurer le modèle transformé afin de l'optimiser ou de l'adapter à une utilisation particulière (cela est également nommé le *refactoring*). Une transformation exogène est effectuée entre des métamodèles différents.

Dans le cadre d'un processus de production, certaines transformations peuvent être vues comme *horizontales* lorsque celles-ci restent au même niveau d'abstraction et d'autres comme *verticales* dans le cas où celles-ci provoquent un changement de niveau d'abstraction.

2.3.3.2 Les règles de transformations

Une transformation de modèle est basée sur le concept de *règles de transformations*. Une transformation étant basée sur un ensemble de règles de transformations qui permet de transformer les concepts et relations du (ou des) métamodèle(s) source vers ceux du (ou des) métamodèle(s) cible. Une transformation repose donc sur le principe de « diviser pour régner ». Il n'existe pas actuellement de définition formelle d'une règle de transformation. Selon les outils utilisés, une règle ne fera pas les mêmes choses. Néanmoins, la norme QVT (Query-View-Transformation), qui est un standard OMG pour l'écriture de transformations de modèle, définit les règles comme « spécifiant comment les éléments entre domaines sont reliés entre eux et comment les éléments d'un domaine sont calculés à partir des éléments des autres domaines ». On considère donc généralement qu'une règle permet de transformer un

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

ou plusieurs concepts sources vers le(s) concept(s) cible. QVT définit deux types de langages permettant d'écrire ces règles : les langages *impératifs* et les langages *déclaratifs*. Les langages *impératifs* vont décrire comment la règle est implémentée pour faire la transformation. Les langages *déclaratifs* déclarent quels sont les motifs qu'une règle sélectionne dans le modèle source et quels sont les motifs que la règle produit dans le modèle cible. Ce type de langage est le plus utilisé en IDM puisqu'il spécifie le but de la transformation sans se soucier de son exécution. L'exécution d'une règle peut être découpée en étapes. La première étape consiste à vérifier la condition de la règle. Cela permet d'identifier les motifs dont a besoin la règle pour s'exécuter. Pour chaque motif vérifiant cette condition, la règle sera exécutée. La seconde étape est la création du motif cible. Pour chacun des motifs sources pour lesquels la règle est exécutée, celle-ci va créer un motif cible. Ensuite, durant la dernière étape, la règle est traitée : les champs du motif cible sont remplis à partir des champs du motif source et les éventuelles sous-règles de la transformation sont exécutées. La structure d'une transformation est hiérarchique. Une règle principale transforme le concept principal source vers le concept cible, cette règle contient des sous-règles permettant de transformer d'autres concepts, et ainsi de suite. Cette hiérarchisation permet de transformer tout le modèle source vers le modèle cible.

Ce découpage en règles a plusieurs avantages. Les règles gérant un petit nombre de concepts sont relativement claires et cela clarifie la transformation globale également. Chaque règle peut être testée indépendamment des autres. Cela facilite l'élimination d'erreurs puisqu'il est plus simple de débugger des petits bouts de code plutôt qu'une application complète. Enfin, une règle peut être réutilisée dans plusieurs règles.

2.3.3.3 Les moteurs de transformations

Il existe de nombreux outils permettant la description et l'exécution de transformation de modèles (Kermeta [134, 97], ATL [75], ModelMorf [41],etc.). Ces outils ne sont pas standard et sont donc très spécifiques. Le seul standard de transformation de modèle est QVT [109] standardisé par l'OMG. Il s'agit d'une spécification permettant la description graphique et textuelle de transformation. Il n'existe pas à l'heure actuelle d'outils implémentant complètement cette spécification, il n'y a que des outils gérant une partie de cette spécification tels que (SmartQVT [130], QVTo [19]). Une approche répandue dans l'implémentation de transformation de modèle est d'utiliser les outils fournis par EMF [49] (Eclipse Modeling Framework), tels que EMF-Query permettant l'exécution de requête sur des modèles réalisé en Ecore. Ce type d'implémentation permet une manipulation complète des modèles en Java mais est réalisé à un niveau très bas.

Certaines notions de transformations de modèle ne sont pas encore bien développées. Parmi celles-ci, il y a la réversibilité d'une transformation. Dans cette section, nous avons toujours considéré qu'une transformation avait une direction. Il existe dans la littérature quelques approches permettant d'obtenir des transformations bidirectionnelles, les règles étant applicables dans un sens comme dans l'autre. Toutefois, cette réversibilité ne peut pas être garantie [136]. Un autre aspect des transformations de modèles intéressant est la traçabilité. Celle-ci consiste à retrouver à partir de quels motifs sources a été créé un motif cible. Cette notion est très intéressante dans l'optique de la correction des erreurs dans les modèles. Lorsqu'est constaté un problème dans un modèle cible, il faut être capable d'identifier d'où provient cette erreur afin de la corriger. Sans la traçabilité, il est très difficile pour un simple

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

utilisateur de retrouver d'où celle-ci provient. Afin de réaliser la traçabilité, une trace de l'exécution de la transformation doit être conservée pour être exploitée dans un second temps. À l'heure actuelle, aucun outil de transformation classique ne supporte la traçabilité bien que des travaux s'y intéressent [62].

2.3.4 Unified Modeling Language (UML)

Dans les sections précédentes, nous avons présenté les concepts de base de l'ingénierie dirigée par les modèles. Dans cette section, nous allons présenter le langage de modélisation le plus couramment utilisé, il s'agit de l'Unified Modeling Language [106] (UML). UML est un langage de modélisation graphique standardisé par l'OMG dans sa version 1.1 en Novembre 1997. Ce langage est basé sur un métamodèle auquel est associée une représentation graphique. Le principal inconvénient de ce langage est qu'il ne propose aucune sémantique. Chacun utilise donc UML à sa guise. La notion de *profils* a été introduit afin de pallier ce défaut. Un profil propose d'ajouter une sémantique à un modèle UML en ajoutant des *stéréotypes*. Un stéréotype ajoute une sémantique à l'élément UML sur lequel il est placé, et peut également ajouter des attributs à cet élément à l'aide des *taggued values*. Ce mécanisme d'extension d'UML est à la base de son succès. Une autre clé de son succès est que de nombreux outils proposent une implémentation de ce standard. Il s'agit aussi bien d'outils propriétaires (MagicDraw UML[3], Enterprise Architecture[4]) que d'outils libres tels que Papyrus[5].

2.3.5 Modélisation d'applications parallèles

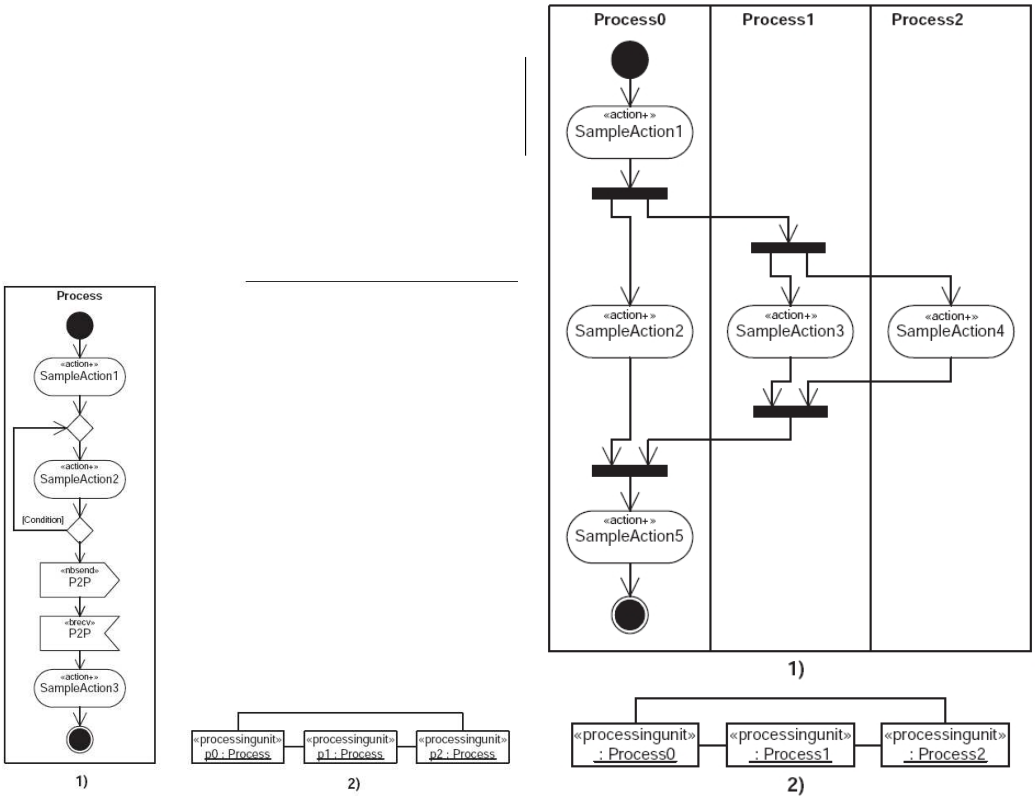
Il existe depuis les années 90 des travaux visant à définir un langage graphique pour spécifier des applications parallèles. Des environnements tels que GRAPNEL [77], HENCE [13], CODE [99], Paralex [10] ou encore TRAPPER [125] ont été développés dans ce but. La plupart de ces représentations étaient basées sur l'idée d'un graphe dans lequel les noeuds représentent les calculs et les arêtes des interactions entre noeuds (communication et autre). La modélisation pouvait être assez précise, ainsi GRAPNEL proposait deux niveaux de modélisation : la communication entre processus et le comportement du processus. La définition du comportement se faisant soit de manière graphique (des représentations des structures C étant disponibles), soit de manière textuelle ce qui peut être considéré comme plus pratique pour définir le comportement. Ces environnements visés à créer des programmes utilisant le paradigme de communications entre les processus (en utilisant les environnements MPI ou PVM). Ces approches ont eu un succès mitigé du fait que leur représentation n'était basée sur aucun standard.

L'apparition du standard UML a permis de disposer d'un environnement de modélisation graphique standard. Des travaux existent pour utiliser celui-ci comme langage de programmation parallèle graphique. Les travaux de Pllana et Fahringer [116, 118] visent à utiliser UML pour la modélisation d'applications hautes performances. Ils définissent un certain nombre de stéréotypes afin de modéliser les briques de bases de ce type d'applications. Ces briques de base proviennent des différents concepts qui peuvent être utilisés. Ainsi, il y a des concepts séquentiels (une tâche de calcul séquentiel), des concepts provenant de la programmation des machines à mémoires distribuées (les communications : *send*, *receive*,

[3] http ://www.nomagic.com
[4] http ://www.sparxsystems.com.au/
[5] http ://www.papyrusuml.org/

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

broadcast et des concepts provenant de la programmation des machines à mémoires partagées (boucle parallèle, travail partagé). Ces concepts s'appliquent à des activités afin de définir une application parallèle. Ensuite les applications sont placées sur des topologies de processus eux-mêmes étant ensuite déployés à l'aide d'un diagramme de déploiement sur la topologie physique de la machine. La figure 2.6 illustre la modélisation et le placement d'applications dans l'environnement proposé par Pllana et Fahringer. Dans la figure 2.6.a.1, un processus est modélisé (on peut d'ailleurs y voir les communications inter-processus nommé (P2P) et utilisant les stéréotypes « bsend » et « brecv »). Dans la partie de 2.6.a.2, ce process est placé sur une topologie de processus en boucle constituée de trois *processingunit*. Chaque *processingunit* va donc exécuter le même process, il s'agit d'une exécution SPMD. Dans la figure 2.6.b.1, on peut voir une application modélisée sur trois processus différents. Les *fork* et *join* y sont modélisés explicitement. La figure 2.6.b.2 présente le placement de cette application sur la topologie de processus en boucle. Chaque *processusunit* va exécuter un processus différent, il s'agit donc d'une exécution MIMD.



(a) Placement SPMD (b) Placement MIMD

FIG. 2.6: Modélisation proposée par Pllana et Fahringer.

Dans le cadre de l'outil Performance Prophet [117], ces modèles sont utilisés pour estimer les performances des applications ainsi modélisées.

Ce type d'approche est très proche des langages de programmation et nécessite donc une connaissance de ces langages afin d'utiliser l'approche. L'utilisation d'une telle approche ne semble pas rendre la programmation plus simple que la manipulation textuelle d'un langage

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

puisque l'utilisateur doit connaître certains aspects de la programmation parallèle comme les communications nécessaires pour l'exécution sur des machines à mémoires distribuées.

2.4 L'environnement Gaspard

Gaspard [39] est, à l'origine, un environnement de co-modélisation pour systèmes sur puces hautes performances développé dans le cadre des travaux du projet INRIA DaRT. Les travaux de cette thèse étant mené au sein du projet DaRT et intégré dans Gaspard, cette section vise à présenter les concepts généraux utilisés dans Gaspard et dont une bonne partie sera réutilisée dans la suite de ce manuscrit.

2.4.1 Co-modélisation - modèle Y (le haut du Y)

La méthodologie de Gaspard est inspirée par la méthodologie en « Y » définie par Gajski et Kuhn [57]. Le but de la méthodologie en Y de Gaspard est, à partir d'une spécification à très haut niveau du système, de générer des codes afin de faire de l'exploration d'architecture en simulant, de synthétiser, de vérifier et d'exécuter les systèmes. Pour cela, les différentes cibles de l'environnement Gaspard sont : le code VHDL permettant la synthèse d'accélérateurs matériels pour une partie du système embarqué ; le code SystemC permettant la co-simulation matériel/logiciel afin d'estimer différentes valeurs intéressantes du système (temps d'exécution, consommation d'énergie notamment) ; les langages synchrones afin de vérifier l'application décrite et les langages parallèles permettant l'exécution sur des calculateurs parallèles (c'est le sujet de cette thèse). La méthodologie de Gaspard est illustrée sous la forme des différents paquetages du profil UML Gaspard (figure 2.7). La méthodologie consiste à réaliser les modèles de l'application et de l'architecture indépendamment l'un de l'autre. Ces deux modèles sont « déployés », c'est-à-dire que les tâches élémentaires (les briques de base) des modèles seront réalisées par les fonctions logicielles ou matérielles (les Intellectual Properties (IPs)) sur lesquelles elles sont déployées. Enfin, l'application et l'architecture sont associées : les tâches applicatives sont placées sur les différents éléments de calculs de l'architecture et les données sont placées dans les mémoires.

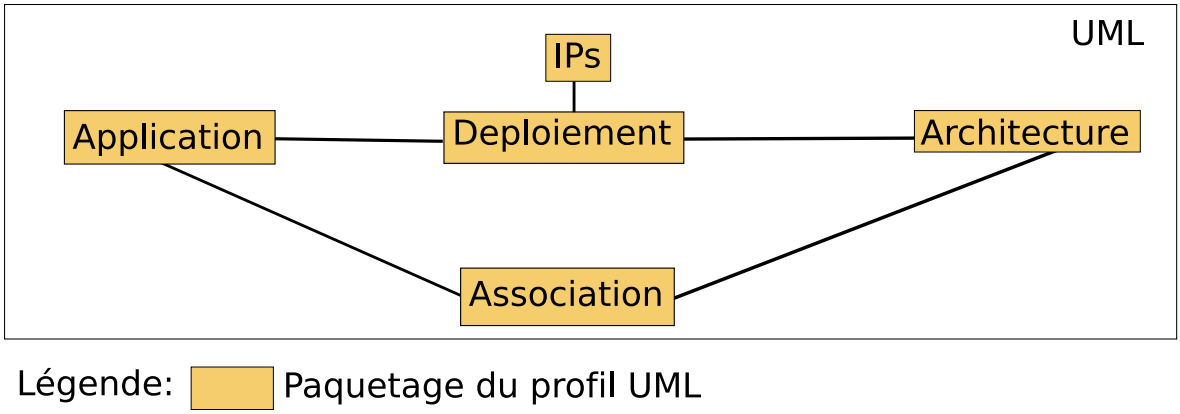


FIG. 2.7: Vue de l'organisation des paquetages du profil UML Gaspard.

Thèse de Julien Taillard, Lille 1, 2009

2.4.2 Langage de spécification : Array-OL

Array-OL (Array Oriented-Language) [21, 22] est un langage de spécification développé par Alain Demeure [40] chez Thales Underwater System. Ce langage vise à permettre la spécification d'applications de traitement de signal intensif manipulant un grand nombre de données tout en exprimant le parallélisme potentiel de l'application.

La constat de départ pour la création du langage était que la complexité de la spécification des applications provenait principalement de la façon dont les applications accèdent aux données. C'est pourquoi Array-OL se focalise sur la spécification des dépendances de données (que ce soit pour le parallélisme de tâches ou le parallélisme de données) sans se préoccuper de la fonctionnalité des tâches de calculs. Comme son nom l'indique les données manipulées par Array-OL sont représentées sous forme de tableaux dans lesquels les dimensions spatiales et temporelles sont banalisées. Les tableaux manipulés sont toriques afin de représenter certaines réalités. Les valeurs fournies par un réseau de sonar autour d'un sous-marin peuvent être représentées sous la forme d'un tableau à deux dimensions : une dimension pour les différents capteurs et une autre pour le temps (les sonars fournissant une donnée à chaque laps de temps). Un sous marin étant circulaire le premier capteur est voisin du dernier, cette propriété peut être utilisée pour réaliser des calculs. Les tableaux toriques permettent d'exprimer aisément des algorithmes utilisant les données provenant de « voisins ». Array-OL est basé sur un formalisme d'assignation unique : un élément ne peut pas être écrit deux fois, mais peut être lu plusieurs fois. Array-OL peut être considéré comme un langage du premier ordre.

Les spécifications du parallélisme sont présentées dans les paragraphes suivants : dans un premier temps la spécification du parallélisme de tâches (§ 2.4.2.1), puis dans un second temps la spécification du parallélisme de données (§ 2.4.2.2). Array-OL étant un langage de spécification, il n'est associé à aucun modèle d'exécution, le paragraphe 2.4.2.3 présente différents modèles d'exécutions pouvant être utilisés pour exécuter des applications spécifiées en Array-OL. Le langage Array-OL a également été étudié afin d'optimiser la façon dont il peut être exécuté. Certaines de ces optimisations sont présentées dans le paragraphe 2.4.2.4.

2.4.2.1 Spécification du parallélisme de tâches

Une application en Array-OL est représentée sous la forme d'un graphe de tâches dirigé acyclique. Dans ce graphe de tâches, les tâches sont les noeuds et les arêtes représentent les dépendances de données entre tâches.

La figure 2.8 illustre un graphe de tâches représentant le parallélisme de tâches en Array-OL. Ce graphe de tâches est composé de trois tâches (A, B et C). Les carrés blancs sur les bords des tâches, appelés ports, représentent les tableaux de données consommés et produits par la tâche. Par défaut, un tel graphe se lit de gauche à droite : les ports à gauche étant les données en entrée, et les ports à droite les données en sortie. Les dimensions à côté des ports expriment les dimensions de tableaux utilisés par les tâches. Ainsi, la tâche A prend en entrée un tableau en deux dimensions ayant comme dimension (1920,1080) et produit en sortie un tableau à deux dimensions (720,1080). Les flèches entre ports expriment les dépendances de données et leurs sens. La tâche C a donc besoin des données produites par la tâche A et par la tâche B pour pouvoir s'exécuter. Cela impose donc une certaine séquentialité. En revanche, les tâches A et B n'ont aucune dépendance de données entre elles, elles sont indépendantes et pourraient être exécutées en parallèle. Il n'y a pas de restrictions quant aux dimensions des tableaux et au nombre de tableaux que consomme et produit une tâche.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

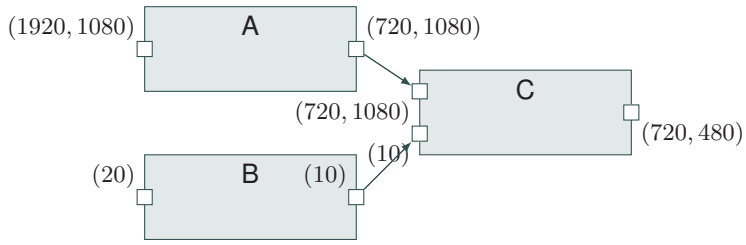


FIG. 2.8: Le parallélisme de tâches dans Array-OL. Trois tâches A,B et C sont représentées ainsi que les dépendances de données entre ces tâches. Les tâches A et B n'ayant pas dépendances de données entre elle, il est possible de les exécuter en parallèle.

2.4.2.2 Spécification du parallélisme de données

Le parallélisme de données est exprimé par une tâche répétée consommant (ou produisant en sortie) des motifs, les motifs étant constitués de points provenant des tableaux auxquels est connectée la tâche répétée. Le nombre de répétitions de la tâche est exprimé par un espace de répétition, cet espace pouvant être multidimensionnel. Les répétitions sont indépendantes les unes des autres, c'est pourquoi il est potentiellement possible d'exécuter ces répétitions en parallèle. La construction des motifs d'entrées (et de sorties) de la tâche répétée se fait à l'aide de *Tilers* qui sont des dépendances de données spéciales. Un *Tiler* représente les dépendances données de chacune des répétitions de la tâche et le tableau auquel il est connecté. Il exprime comment calculer un motif à partir de l'index de la répétition dans l'espace de répétition, du tableau d'entrée (ou de sortie) et de la dimension du motif. Un tiler est constitué des informations suivantes :

- **o** : l'origine ou élément de référence
- P : la matrice de pavage (Paving)
- F : la matrice d'ajustage (Fitting)

La construction d'un motif se déroule en deux étapes : le pavage qui permet de calculer le premier point d'un motif (l'élément de référence), et l'ajustage qui exprime comment calculer les autres points du motif à partir de l'élément de référence. Ces deux étapes sont présentées et illustrées dans les deux paragraphes suivants.

Avant de décrire ces deux étapes, fixons quelques notations :

- $s_{pattern}$: taille du motif
- s_{array} : taille du tableau
- $s_{repetition}$: espace de répétition de la tâche
- **r** : index d'une répétition dans l'espace de répétition
- **i** : index d'un point dans le motif

Ajustage L'ajustage consiste à reconstituer le motif à partir d'un élément de référence (qui sera calculé durant la phase de pavage). Pour cela, l'ajustage utilise la matrice d'ajustage (F), l'indice dans le motif (**i**) et l'élément de référence. Le calcul de l'ajustage est une combinaison linéaire de ces trois données :

$$\forall \mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e_i} = \mathbf{ref} + F \cdot \mathbf{i} \bmod \mathbf{s}_{\text{array}} \tag{2.1}$$

L'adresse (les coordonnées) dans le tableau de l'élément d'indice **i** dans le motif (noté $\mathbf{e_i}$) est la somme entre les coordonnées de l'élément de référence du motif, et le produit scalaire de la matrice d'ajustage (F) et l'indice de l'élément **i** le tout étant modulo la taille du tableau (s_{array}) afin de gérer l'aspect torique des tableaux Array-OL.

La construction de motifs est illustrée dans les figures 2.9 et 2.10. Dans ces figures, le point rouge représente l'élément de référence à partir duquel est calculé l'ajustage. La figure 2.9 représente la construction d'un motif unidimensionnel de taille (3). Le premier point du motif étant l'élément de référence, il reste à déterminer les deux autres points. La matrice d'ajustage exprime que l'on se déplace de 3 dans la première dimension du tableau lorsque l'on incrémente le vecteur **i**. Les deux autres points constituant le motif sont les deux autres points représentés sur la figure. La figure 2.10 illustre la construction d'un motif à 2 dimensions qui ne suit pas les axes du repère.

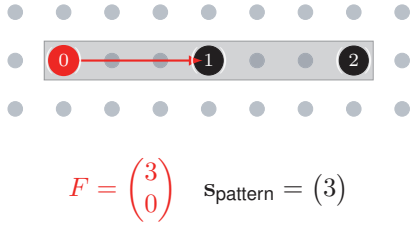


FIG. 2.9: Construction d'un motif simple.

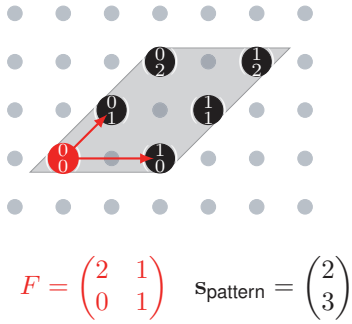


FIG. 2.10: Construction d'un motif à deux dimensions non parallèle aux axes.

Pavage Le pavage sert à calculer l'élément de référence de chacun des motifs spécifiés par le tiler. À partir de l'origine (ou élément de référence global), les éléments de référence des différents motifs sont calculés à l'aide de la matrice de Pavage. Ces points sont calculés

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

comme une combinaison linéaire entre la matrice de Pavage (P), l'index de la répétition (**r**) et l'origine.

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \mathbf{ref}_{\mathbf{r}} = \mathbf{o} + P \cdot \mathbf{r} \bmod \mathbf{s}_{\text{array}} \tag{2.2}$$

L'élément de référence pour le motif utilisé à la répétition **r** est noté $\mathbf{ref}_{\mathbf{r}}$. La construction de cet élément se fait par la somme des coordonnées de l'origine (l'élément de référence global) et du produit scalaire entre la matrice de pavage (P) et l'index de la répétition (**r**). Le calcul est effectué modulo la taille du tableau (s_{array}) afin de respecter la propriété torique des tableaux. La construction des différents éléments de référence pour un tiler est présentée à la figure 2.11. Cette figure illustre la construction des éléments de références et des motifs dans le cadre d'un espace de répétition de (5). Pour chaque indice (**r**) de l'espace de répétition, les vecteurs (en bleu) illustrent les décalages effectués à partir de l'élément de référence à l'aide de la matrice de pavage P. Le motif sélectionné correspond à une ligne du tableau.

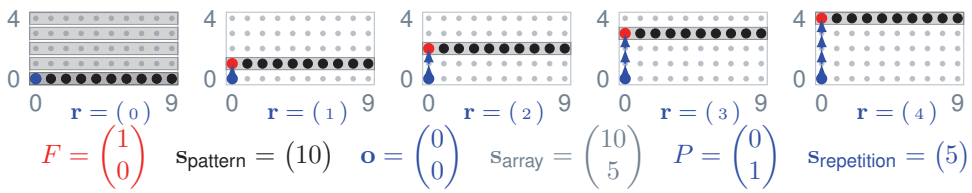


FIG. 2.11: Pavage d'un tableau à l'aide d'un motif de taille (10).

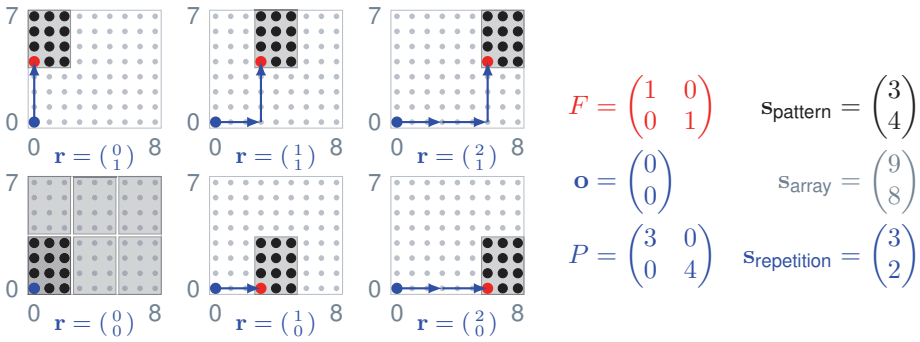


FIG. 2.12: Pavage complet d'un tableau deux dimensions à l'aide d'un motif à deux dimensions également (3,4).

La spécification graphique du parallélisme de données en Array-OL est donnée dans la figure 2.13. La tâche Hfilter prenant en entrée un motif de taille (13) et produisant un motif de taille (3) est répétée (240,1080) fois. Les tilers connectant les ports de la tâche Hfilter à ceux de son conteneur (Horizontal Filter) expriment comment les motifs sont pris (et placés) dans le tableau d'entrée de taille (1920,1080) (dans le tableau de sortie (720,1080)).

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

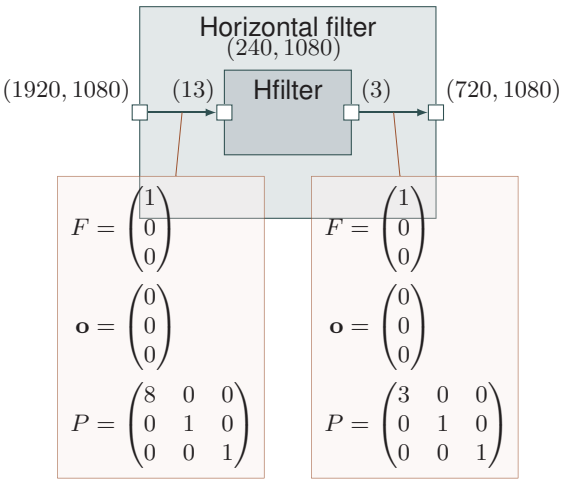


FIG. 2.13: Illustration du parallélisme de données dans Array-OL. La tâche Hfilter est répétée (240,1080) fois.

2.4.2.3 Modèle d'exécution

Array-OL étant un langage de spécification, il n'est associé à aucun modèle d'exécution. Au cours de sa thèse, Philippe Dumont a étudié les différents modèles d'exécutions qu'Array-OL peut supporter. Trois modèles d'exécutions viennent rapidement à l'esprit :

- séquentiel : les tâches du graphe de tâches sont exécutées dans un ordre respectant les dépendances de données, la répétition d'une tâche est effectuée séquentiellement répétition par répétition ;
- SPMD : les répétitions d'une tâche étant indépendantes, elles peuvent être exécutées en parallèle, deux tâches du graphe de tâches n'ayant pas de dépendances entre elles peuvent également l'être ;
- pipeline : l'idée de cette exécution est de créer un flux de tableau permettant de pipeliner l'exécution (dans le cadre du traitement de signal intensif). Cela nécessite une modification de l'application afin de l'adapter à ce modèle d'exécution. Des implémentations de ce modèle d'exécution ont été réalisées dans des modèles de calculs connus tels que Synchronous Data Flow (SDF) [89, 90] et les Kahn Process Network (KPN) [78] réalisés durant la thèse d'Abdelkader Amar [8].

Le choix d'un de ces modèles de calculs dépend de la machine cible sur laquelle on veut exécuter notre application. Dans le cadre d'une simple exécution, l'exécution séquentielle peut convenir. Dans le but d'obtenir de bonnes performances sur une machine parallèle, le modèle d'exécution SPMD semble adapté alors que dans le cadre du traitement de signal intensif utilisant des systèmes contenant des processeurs spécialisés pour chacune des tâches, le modèle d'exécution pipeline pourrait parfaitement convenir.

Thèse de Julien Taillard, Lille 1, 2009

2.4.2.4 Optimisations des applications Array-OL

Afin d'obtenir de meilleures performances lors de l'exécution des applications Array-OL, différentes optimisations peuvent être réalisées. Le but principal de ces optimisations est la modification de l'application afin de l'adapter à l'architecture d'exécution. Une application Array-OL refactorée reste une application Array-OL, il est donc possible d'appliquer différentes fonctions de refactoring à la suite sur la même application. Ces fonctions proposées par Julien Soula en 2001 et étendues par Philippe Dumont en 2005, sont encore actuellement étendues dans le cadre des travaux de thèse de Calin Glitia. La fonction de fusion permet, par exemple, de fusionner deux tâches en une seule qui, elle, contiendra deux tâches. Ce type d'optimisation permet notamment d'optimiser l'utilisation de la mémoire cache en utilisant complètement les données en caches avant de charger de nouvelles données. L'utilisation de ces fonctions est aujourd'hui manuelle et demande donc une certaine connaissance de l'architecture cible et de la manière dont est exécutée l'application afin de connaître quelles fonctions utiliser.

2.4.3 Modélisation à haut niveau

L'environnement Gaspard étant un environnement basé sur l'ingénierie dirigée par les modèles, la spécification à haut niveau se réalise en utilisant des modèles. UML étant le standard de modélisation le plus connu et le plus utilisé à l'heure actuelle, c'est naturellement sur ce langage de modélisation que la spécification Gaspard s'appuie. UML fournit un grand nombre de concepts de modélisation, mais aucune sémantique n'y est associée. Afin de donner une sémantique aux modèles Gaspard, un profil UML a été défini. Un profil permet d'ajouter de la sémantique aux concepts en y ajoutant des « stéréotypes ». Le profil Gaspard [15] propose donc les concepts sémantiques nécessaires afin de modéliser les applications, les architectures matérielles, mais également l'association de l'application sur l'architecture (allocation et distribution des tâches et des données). L'organisation du profil a été illustrée à la figure 2.7. Dans le cadre de cette thèse, le profil Gaspard a été utilisé, mais dans le futur le profil Gaspard est destiné à être remplacé par le profil standard OMG MARTE (Modeling and Analysis for Real-Time and Embedded systems) [120] auquel l'équipe a contribué et qui reprend les principaux concepts du profil Gaspard (seuls certains noms changent). Le profil Gaspard a pour origine les travaux d'Arnaud Cuccuru lors de sa thèse [38] et est basé sur le langage Array-OL présenté précédemment.

Afin d'illustrer la modélisation en UML utilisant le profil Gaspard, les différents concepts introduits seront utilisés pour modéliser une application facilement compréhensible : la multiplication de matrices en utilisant l'algorithme ligne / colonne.

2.4.3.1 Concepts généraux

Les modèles d'applications et d'architectures ont une base commune qui est ensuite spécialisée à l'aide des généralisations UML. La modélisation est basée sur les composants UML2 [107]. Dans le cadre de Gaspard, un composant est défini comme étant une tâche de calcul (dans le cadre d'une application) ou un élément d'architecture matériel (dans une architecture). Les informations entre les composants étant « véhiculés » via les ports et les connecteurs.

Les expressions Array-OL sont utilisées afin d'avoir une modélisation factorisée des applications et des architectures régulières (cela est introduit dans le standard MARTE via le

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

package RSM (Repetitive Structure Modeling) [23]). Cette expression autorise la modélisation d'une forme régulière de répétitions d'une instance[6]. Un modèle composé de N instances du même composant sera représenté par une instance du composant répétée N fois. On va ainsi utiliser le pouvoir d'expression d'Array-OL pour l'expression de tout le parallélisme potentiel des applications et des architectures. Le concept de *Tiler* sera utilisé dans nos modélisations. Une extension de ces expressions est notamment introduite afin de permettre l'expression de la répartition des éléments d'un tableau sur les éléments d'un autre tableau. Ce concept va notamment permettre l'expression de la distribution d'une tâche répétée sur un ensemble de processeurs, mais également exprimer comment un tableau peut changer de forme (*Reshape*). La distribution des tâches sur les processeurs sera présentée un peu plus tard (§ 2.4.3.4).

Les composants de base des modèles sont les composants élémentaires (utilisant le stéréotype « ElementaryComponent »). Ces composants n'ont pas de structure interne et sont considérés comme des « boites noires » qui sont ou seront déployées sur des fonctions de bibliothèques. Ces fonctions peuvent être des fonctions classiques de bibliothèques de calculs numériques, mais également des IPs (Intellectual Property) permettant de simuler la fonctionnalité d'éléments d'architecture.

2.4.3.2 Modélisation d'application

Une application Gaspard est semblable à une application Array-OL. Il s'agit d'un graphe de tâches hiérarchiques contenant le parallélisme potentiellement exploité de cette application. Afin de spécifier qu'il s'agit de composants applicatifs, les composants sont stéréotypés « ApplicationComponent ». Les connecteurs UML n'étant pas orientés, il est nécessaire de spécifier si les ports sont en entrée (« In »), en sortie (« Out ») ou les deux (« InOut ») afin de déterminer le sens des connecteurs. Les stéréotypes sur les ports apparaîtront dans les schémas UML sous forme de couleur : bleu pour « In », rose pour « Out » et la couleur par défaut (vert) pour « InOut ». Dans le cadre des schémas présentés dans ce manuscrit, nous nous imposerons que la circulation des données se fera toujours de gauche à droite et de haut en bas.

Modélisation du parallélisme de tâches La figure 2.14 illustre le parallélisme de tâches sur le composant principal de l'application de multiplication de matrice (il peut être comparé au *main* d'un programme C). Ce composant (*MatrixMultiplicationMain*) est composé de quatre instances de composants. Les deux instances de *initMatrix* (*iM1* et *iM2*) initialise les matrices (initialisation aléatoire ou lecture dans des fichiers par exemple). Ces matrices sont ensuite fournies à l'instance du composant *MatrixMultiplication* qui va calculer la multiplication de matrices. Et pour finir l'instance du composant *saveMatrix* peut stocker cette matrice dans un fichier, l'afficher ou encore la vérifier.

Modélisation du parallélisme de données Le parallélisme de données est basé sur l'expression du parallélisme de données en Array-OL. La figure 2.15 illustre l'expression de l'algorithme de multiplication ligne / colonne de deux matrices[7].

Le composant *MatrixMultiplication* est composé d'une tâche répétée (l'instance *dP* de *dotProduct*). Cette tâche répétée représente la tâche de calcul, qui prenant une ligne et une

[6]Le mot instance est abusivement utilisé afin de désigner les property UML

[7]Cet algorithme est illustré sur de petites matrices (4,4) afin de permettre une explication simple, mais peut évidemment être appliqué à des matrices beaucoup plus grandes

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

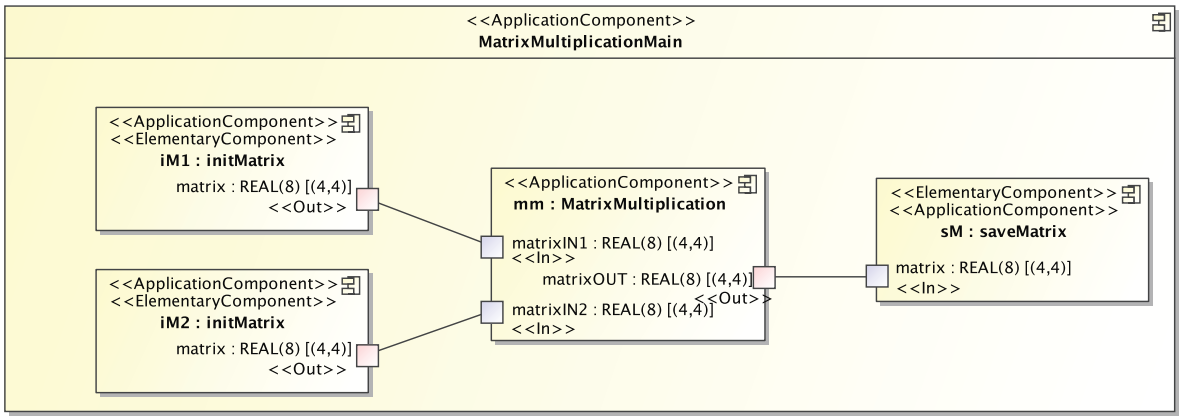


FIG. 2.14: Illustration du parallélisme de tâches modélisé à l'aide du profil Gaspard.

colonne, produit un point de la matrice. Les connecteurs stéréotypés « Tiler » expriment comment la tâche répétée consomme et produit les motifs selon l'indice de la répétition dans l'espace de répétition.

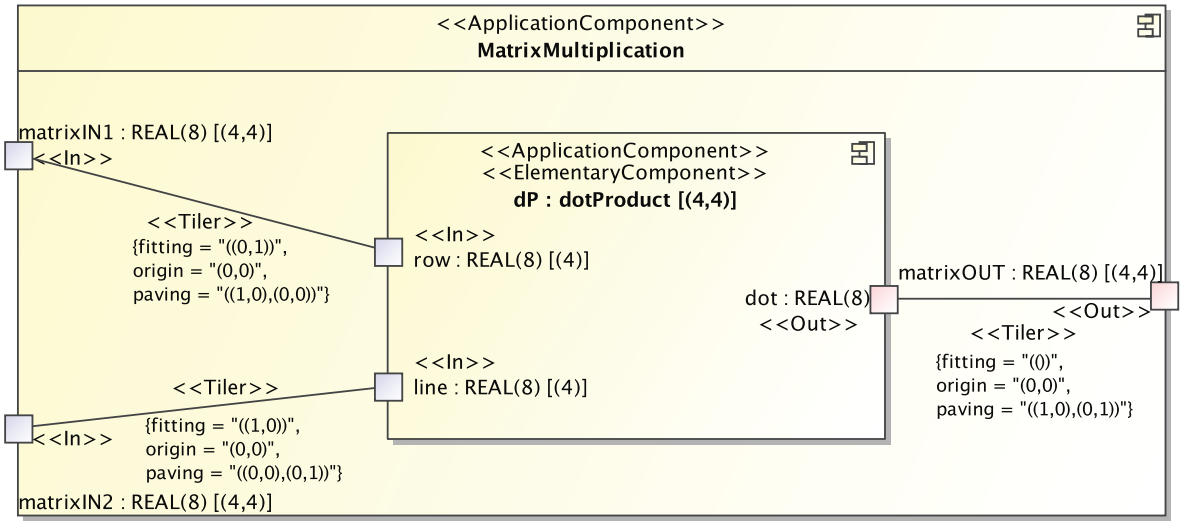


FIG. 2.15: Illustration de la modélisation du parallélisme de données en UML à l'aide du profil Gaspard.

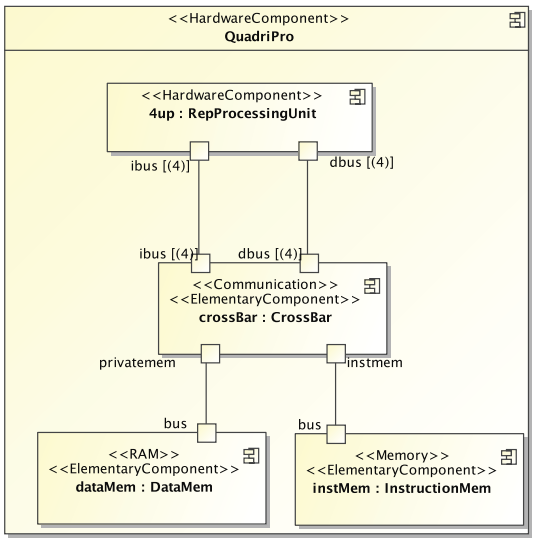
Les ports d'entrée *matrixIN1* et *matrixIN2* représentent les deux matrices d'entrée et le port de sortie *matrixOUT* représente la matrice de sortie. Le tiler allant de *matrixIN1* vers *line* permet de prendre des lignes dans *matrixIN1* selon l'indice de répétition alors que le tiler allant de *matrixIN2* vers *row* prend des colonnes.

2.4.3.3 Modélisation d'architecture

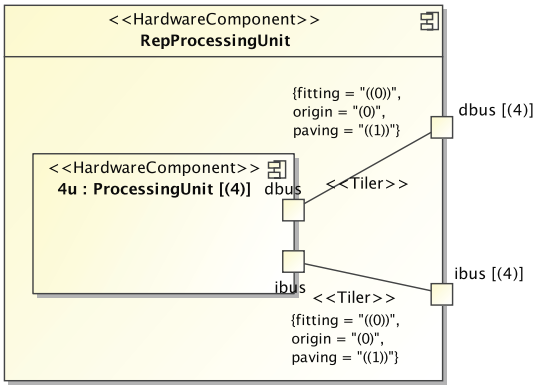
La modélisation de l'architecture est basée sur les concepts généraux de Gaspard et notamment sur la répétition des instances permettant de spécifier de façon factorisée les architectures régulières. Dans le cadre de la modélisation de l'architecture, les ports représentent

Thèse de Julien Taillard, Lille 1, 2009

différents connecteurs des composants matériels et les connecteurs des connexions physiques entre les composants.



(a) Modélisation d'une architecture de machine à mémoire partagée.



(b) Répétition des processeurs dans le composant *RepProcessingUnit*.

FIG. 2.16: Modélisation d'une architecture quadri-processeurs à mémoire partagée.

La figure 2.16 illustre la modélisation d'une architecture quadri-processeurs à mémoire partagée. L'architecture globale de la machine est modélisée à gauche de la figure (a). Celle-ci est composée d'une répétition de processeurs connectée à une mémoire d'instructions et à une mémoire de données via un crossbar. La répétition des processeurs est modélisée dans la partie droite de la figure (b). Elle est constituée de la répétition d'un même processeur, celui-ci étant répété quatre fois. Les tilers expriment ici la topologie de connexion entre les ports des processeurs et le crossbar (via la hiérarchie).

2.4.3.4 Modélisation d'association

L'association consiste à exprimer le placement de l'application sur l'architecture matérielle. Pour cela, les calculs doivent être placés sur les processeurs (à l'aide d'un ordonnancement statique) et les données sur les différentes mémoires. L'association est un point important pour obtenir des performances. Dans cette section, les concepts permettant l'allocation et la distribution des tâches sur les processeurs vont être présentés. Le placement des tâches sur les processeurs peut être vu de deux façons qui seront présentées dans les paragraphes suivants : l'allocation et la distribution.

Allocation L'allocation est l'opération consistant à allouer un élément sur un autre. Dans le cas d'une tâche, celle-ci est allouée sur un processeur qui aura la charge de l'exécuter. Dans le profil Gaspard, cette opération est réalisée à l'aide d'une dépendance partant de la tâche et allant vers le processeur et utilisant le stéréotype « TaskAllocation ». Cette opération fonctionne entre deux éléments ayant une multiplicité unitaire.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Distribution La distribution consiste (principalement) à répartir les répétitions d'un élément sur les répétitions d'un autre élément. Pour cela, une extension des concepts d'Array-OL a été effectuée afin d'exprimer cette distribution. Le principe de base de la distribution est de sélectionner un motif dans la répétition de la tâche répétée et de placer ce motif sur les processeurs répétés. Cette opération doit être répétée le nombre de fois suffisant permettant de placer au moins une fois chacune des répétitions de la tâche sur les processeurs. Afin d'effectuer cette opération, une « Distribution » est construite à l'aide de deux tilers, de la taille et la forme du motif ($pattern\_shape$) à construire et du nombre de répétitions à effectuer ($repetition\_space$). Le tiler $sourceTiler$ exprime comment remplir le motif intermédiaire à l'aide des indices de répétitions de la taĉhe. Le tiler $targetTiler$ exprime comment répartir les éléments composant le motif sur la répétition de processeurs.

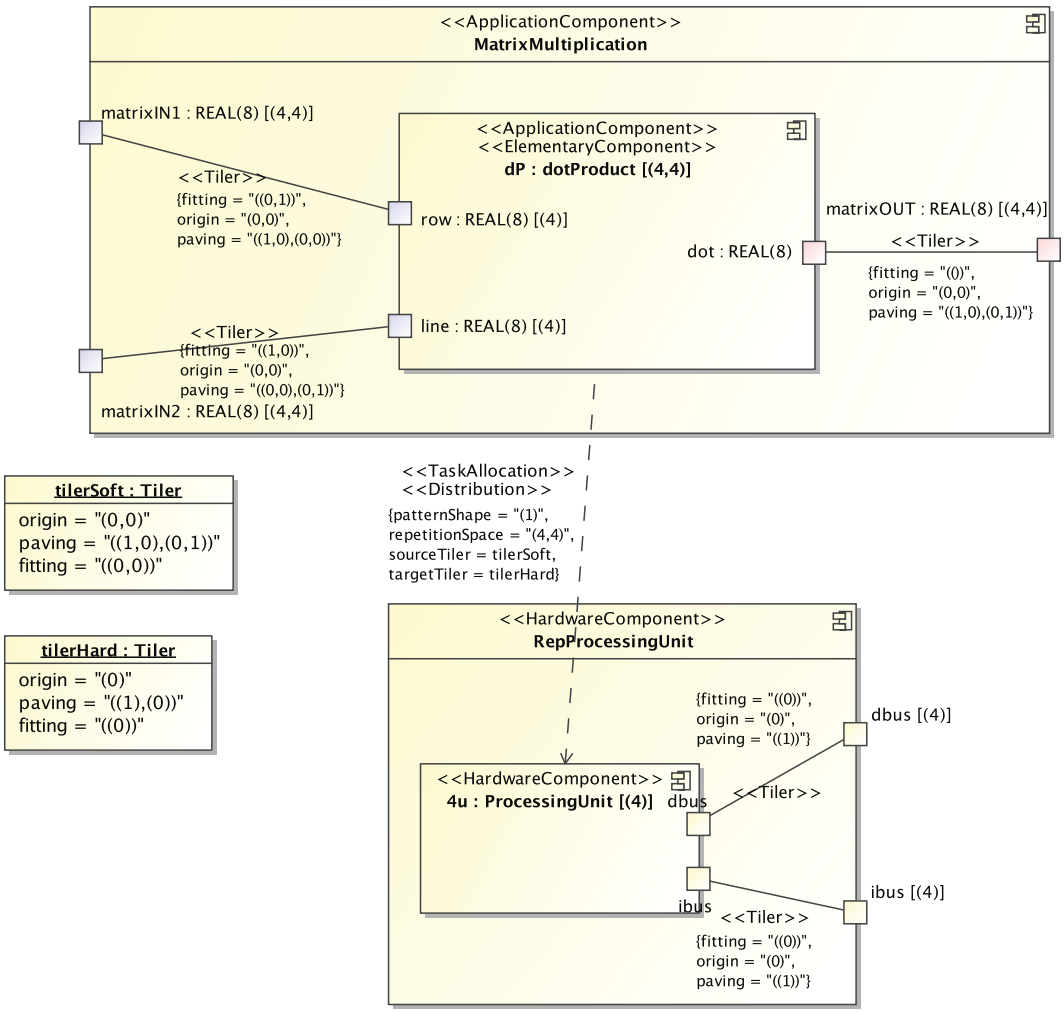


FIG. 2.17: Illustration du placement d'une tâche répétée sur une répétition de processeurs.

La figure 2.17 illustre l'association de la répétition de produit scalaire sur la répétition de processeurs de l'architecture présentée dans la section précédente. Cette distribution exprime un placement par colonne. Ainsi, chaque processeur se voit affecter une colonne de calcul à effectuer. Ce placement est illustré dans la figure 2.18.

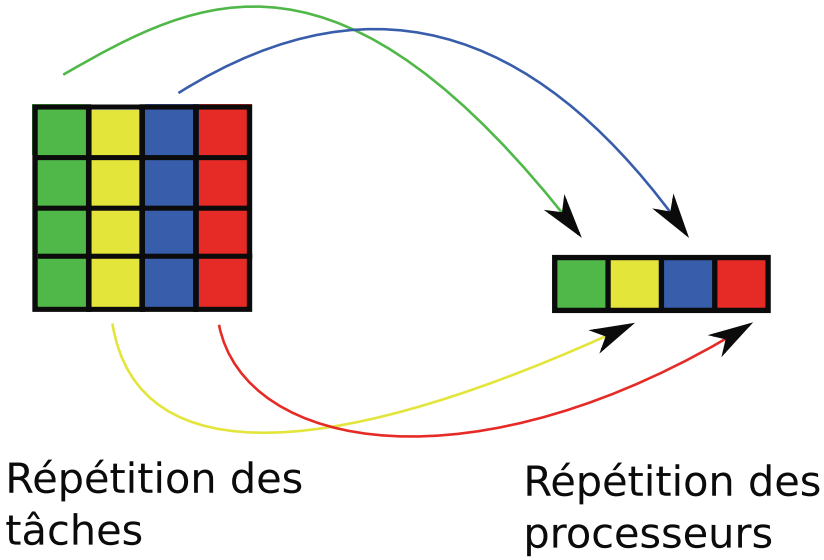


FIG. 2.18: Illustration du placement des répétitions des tâches sur les répétitions de processeurs.

L'allocation et la distribution des données sur les mémoires sont gérées de la même façon à l'aide du stéréotype « DataAllocation ».

Il est important de noter qu'actuellement le placement n'est pas contextuel, ce qui entraîne que chaque utilisation d'une instance sera allouée et distribuée de la même façon dans toute l'application. Reprenons la distribution illustrée à la figure 2.18, on peut imaginer une application effectuant plusieurs multiplications de matrices et ayant donc plusieurs instances du composant *MatrixMultiplication*. Le placement de l'instance *dP* composant la multiplication de matrice sera le même pour toutes les instances de *MatrixMultiplication* car celui n'a pas de notion de contexte. La réalisation de placement avec contexte serait possible en utilisant une notation de placement associant l'instance que l'on veut placer/distribuer avec l'instance du composant supérieur à laquelle correspond ce placement.

2.4.4 Un modèle exécutable

Dans la section précédente, nous avons présenté comment modéliser une application, une architecture et l'association entre les deux. Cette modélisation est indépendante de la plate-forme d'exécution : le langage que l'on va cibler avec Gaspard. Afin d'obtenir un code exécutable, il faut que le modèle soit « exécutable », c'est à dire que toutes les tâches élémentaires (les briques de base des modèles) soient liées à leur fonctionnalité dans le langage cible. Ainsi, les tâches applicatives doivent être liées aux routines effectuant le calcul nécessaire dans le langage cible et les composants matériels sont liés à des « bout de code » permettant la simulation de ces composants. Cette liaison est effectuée durant la phase de Déploiement. Le profil de déploiement a été développé par Éric Piel et Rabie Ben Attitalah lors de leurs thèses en collaboration avec les autres membres de l'équipe de développement.

Afin de simplifier la présentation des concepts, nous allons présenter uniquement le déploiement des composants applicatifs. Les notions pour le déploiement des composants matériels sont semblables.

Il peut exister plusieurs implémentations de la même fonction (différents langages et différents algorithmes). Le concept d'« AbstractSoftwareImplementation » fait ainsi référence à la fonctionnalité des fonctions. Celui-ci contient ensuite les implémentations réelles, les « SoftwareImplementations ». Le déploiement consiste donc à lier chaque tâche applicative élémentaire à un « SoftwareImplementation ».

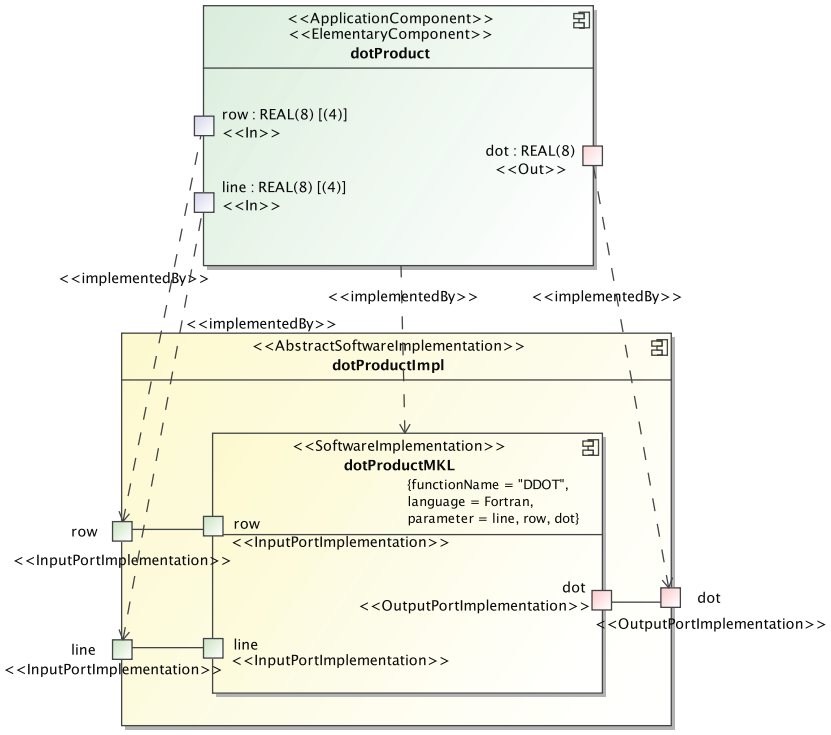


FIG. 2.19: Déploiement du composant élémentaire effectuant le produit scalaire sur une routine DDOT.

La figure 2.19 illustre le déploiement du composant élémentaire devant effectuer le produit scalaire entre une ligne et une colonne afin de produire une ligne. Le lien entre le composant élémentaire et la « SoftwareImplementation » stéréotypé « implementedBy » indique que le composant *dotProduct* est déployé sur l'implémentation *dotProductMKL* qui comme son nom l'indique est une routine appelant la routine DDOT de la MKL[8]. Les « portImplementedBy » indiquent comment les ports du composant sont déployés sur les ports d'implémentations. L'implémentation précise le nom de la routine (*DDOT*), le langage de celle-ci (*Fortran*) et les données à lui fournir en paramètres. Dans le cadre de composants applicatifs, les ports des composants sont considérés comme étant les paramètres de la fonction. Afin de générer un appel valide à la routine, il faut fournir les paramètres dans le bon ordre. C'est ce que précise le champ *parameter* de l'implémentation qui indique l'ordre dans lequel mettre les ports afin de générer l'appel valide.

[8]la bibliothèque de calcul optimisée de Intel

2.4.5 Le métamodèle Gaspard

La modélisation d'une application, d'une architecture et de leur association est exprimée en UML à l'aide du profil Gaspard. Afin de faciliter la transformation de modèles, un métamodèle Gaspard correspondant à ce profil a été créé. Ce métamodèle reprend tous les concepts du profil. Cela permet de commencer les transformations à partir d'un modèle dans lequel tous les éléments sont utiles et connus sans avoir tous les éléments UML qui peuvent être considérés comme inutiles et « pollueurs » du modèle.

La figure 2.20 illustre la base du métamodèle Gaspard : la définition des composants. Ces composants seront utilisés aussi bien dans la modélisation de l'application que de l'architecture puisque ces deux types de composants généraliseront le concept de composant ici présenté. Un COMPONENT est un élément ayant un nom et contient une structure interne qui représente le comportement du composant. La structure interne peut être ELEMENTARY quand on veut modéliser un composant élémentaire, COMPOUND quand un composant est composé par plusieurs instances d'autres composants ou REPETITIVE (concept non illustré dans la figure) qui indique que l'instance du composant interne est répétée un certain nombre de fois. Une description plus détaillée de ce métamodèle pourra être trouvée dans l'annexe A.

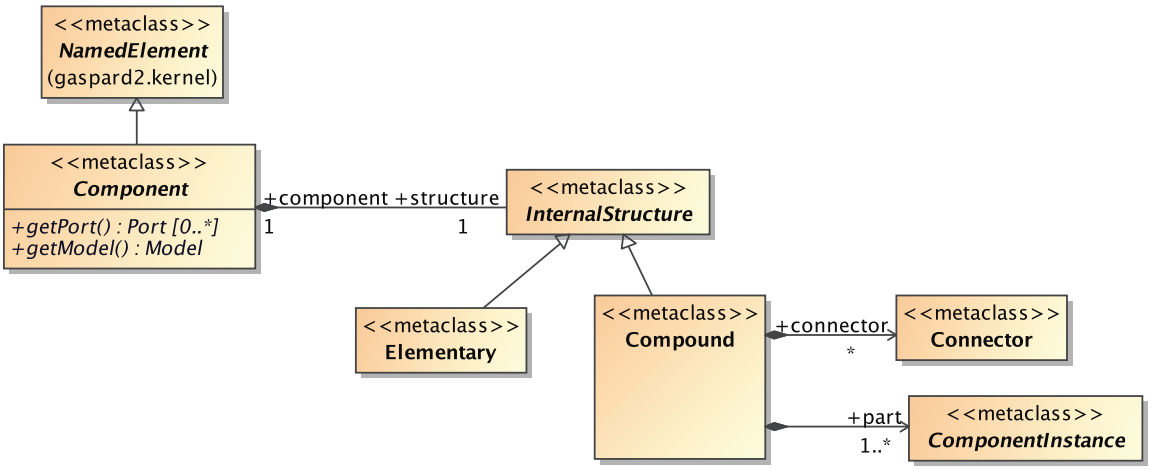


FIG. 2.20: Définition des composants dans le métamodèle Gaspard.

2.5 Conclusion

Dans ce chapitre, nous avons présenté les différents travaux et problématiques à la base de ces travaux de thèse. Ainsi, alors que les machines parallèles se généralisent notamment grâce à l'apparition des processeurs multi-cœurs, la programmation parallèle reste l'affaire de spécialistes. Bien que des travaux soient menés afin de la simplifier à travers le développement de nouveaux langages parallèles et de la parallélisation automatique, celle-ci est encore difficile notamment à cause de la diversité des langages parallèles et des applications. Le génie logiciel développé afin d'augmenter la productivité dans le développement de logiciel peut aider à simplifier la programmation parallèle. En effet, l'ingénierie dirigée par les modèles propose de s'abstraire des détails d'implémentations en modélisant à un haut niveau

Thèse de Julien Taillard, Lille 1, 2009

d'abstraction. Dans cette thèse, nous proposons d'utiliser une méthodologie basée sur les modèles afin de simplifier la programmation parallèle tout en bénéficiant de la réutilisation des modèles de haut niveau pour toute architecture cible. Enfin, l'environnement Gaspard a été introduit. La modélisation basée sur le langage Array-OL propose de modéliser tout le parallélisme potentiel des applications et des architectures avant d'effectuer l'association entre les deux. C'est dans cet environnement que seront implémentés les travaux de cette thèse.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

© 2010 Tous droits réservés.

http://doc.univ-lille1.fr

Chapitre 3

Méthodologie de compilation d'un modèle de haut niveau pour le calcul haute performance basée sur l'IDM

3.1 Les apports d'une approche IDM pour le calcul haute performance **52**
3.1.1 Le métamodèle : un langage parallèle 52
3.1.2 Les transformations de modèle : le compilateur 52
3.2 Définition d'une chaîne de compilation des applications de calcul haute performance **53**
3.2.1 Cible de la chaîne de compilation : les machines à mémoires partagées 53
3.2.2 Stratégie globale d'une chaîne de compilation IDM partant d'une spécification Array-OL 54
3.2.3 Implémentation dans Gaspard 55
3.3 Modèle d'exécution en OpenMP des applications spécifiées en Array-OL et distribuées sur les différents processeurs et/ou cœurs **57**
3.3.1 Génération de code SPMD : modèle d'exécution 57
3.3.2 Parallélisme de tâches 58
3.3.3 Parallélisme de données 59
3.3.4 Synchronisation des tâches 59
3.3.5 Gestion des tilers 59
3.4 Conclusion du chapitre **60**

Ce chapitre présente la méthodologie basée sur l'ingénierie dirigée par les modèles définie au cours de cette thèse. Le but de celle-ci est de proposer une stratégie visant à produire du code parallèle à partir de spécifications d'applications parallèles de haut niveau réalisées en UML. À partir de la spécification UML différents langages parallèles peuvent être générés. Dans le cadre de notre méthodologie, nous viserons l'exploitation des machines à mémoires partagées, catégorie dont font partie les machines classiques composées de processeurs multi-cœurs.

Avant de la présenter plus en détail, les apports potentiels de l'IDM dans le cadre du domaine du calcul haute performance sont présentés. Ces apports sont au niveau du langage parallèle ainsi qu'au niveau de la compilation de ce langage.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

Ce chapitre est décomposé en trois parties. Le premier paragraphe (3.1) illustre les apports potentiels des modèles pour la programmation parallèle. La méthodologie est ensuite définie à un niveau général avant d'illustrer son intégration dans l'environnement Gaspard (§ 3.2). Enfin, le modèle d'exécution des applications définies à haut niveau est introduit (§ 3.3).

3.1 Les apports d'une approche IDM pour le calcul haute performance

Le but de cette section est de distinguer les apports d'une approche IDM pour le calcul haute performance. L'utilisation d'une approche IDM sous-entend l'utilisation de modèles comme interfaces de spécification et de produire du code à partir de ces modèles. Les apports de l'IDM se font sentir aussi bien du côté de la spécification (§ 3.1.1) que du côté de la compilation de cette spécification vers du code exécutable (§ 3.1.2).

3.1.1 Le métamodèle : un langage parallèle

Dans le cadre d'une approche IDM pour le calcul haute performance, les modèles servent à spécifier une application parallèle. Le métamodèle spécialisé, ou le métamodèle UML utilisant un profil, peut être considéré comme un langage parallèle. La conception d'un langage parallèle bénéficie alors des avantages liés aux modèles. L'un des buts principaux des modèles est de se concentrer sur la fonctionnalité de l'application sans se soucier des détails liés à l'implémentation. Cela entraîne une hausse du niveau d'abstraction au niveau du langage de spécification, ce qui est nécessaire pour rendre plus accessibles les langages parallèles. Ce haut niveau d'abstraction devrait autoriser le fait d'avoir une vue globale de l'application spécifiant globalement celle-ci sans se soucier de sa future distribution et exécution.

La spécification à haut niveau à l'aide de modèles implique également la notion de programmation graphique, via les éditeurs UML classiques (dans le cas de modèles UML) ou via un éditeur spécialisé (dans le cas de l'utilisation d'un métamodèle quelconque). La programmation graphique est, à priori, plus conviviale que la programmation textuelle. Elle permet également une communication plus simple entre développeurs : un modèle lisible et dont les noms des tâches sont explicites étant plus compréhensible qu'un programme ayant une vue fractionnée.

Un très haut niveau d'abstraction est également un gage de portabilité du modèle. En effet, le modèle étant très abstrait par rapport à l'implémentation, il est tout à fait possible de cibler différents langages parallèles à partir d'un même modèle (les détails d'implémentations étant ajoutés lors de la transformation de ce modèle). Il est ainsi imaginable de, à partir d'un même modèle, générer du code pour machines à mémoires partagées et du code pour machines à mémoires distribuées.

3.1.2 Les transformations de modèle : le compilateur

L'IDM apporte au calcul haute performance, un nouveau type de langage parallèle (le modèle et son métamodèle) mais il apporte également une méthodologie permettant la compilation de ce langage afin de le rendre productif : la transformation de modèle.

Le principe de transformation de modèles promeut la décomposition de la transformation en plusieurs petites transformations successives. Cette décomposition de la transformation,

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

basée sur le principe de *diviser pour régner*, entraîne la production de petites transformations ayant chacune un but précis. Le développeur se concentre alors sur ce but précis de la transformation sans se soucier des autres et valide l'étape en vérifiant le modèle intermédiaire produit. Ces petites transformations sont plus simples à développer (une petite fonctionnalité est plus simple à implémenter qu'une grande). Cette décomposition permet d'identifier plus facilement les bugs puisqu'une fois que la transformation incriminée est détectée, le bug se trouve forcément dans la partie de code concernée, ce qui réduit le temps passé à chercher les bugs.

La décomposition des transformations entraîne également un autre avantage puisqu'il est possible de réutiliser une sous transformation dans un autre processus de compilation. Deux chaînes de transformations effectuant les mêmes transformations avant de diverger vers leur langage cible utilisent les mêmes codes de transformation. Cela limite les temps de développement et, si l'approche IDM est bien réalisée, permet une extension simple de l'environnement dans lequel les transformations sont implémentées. La notion de réutilisation est donc présente dans les modèles de haut niveau (les applications) mais également pour les transformations de modèles.

3.2 Définition d'une chaîne de compilation des applications de calcul haute performance

Dans ce paragraphe, nous allons présenter globalement la chaîne de compilation d'un modèle de haut niveau définie dans le cadre de cette thèse. Tout d'abord sera présenté le choix de la cible de compilation. Ensuite, la stratégie globale est illustrée et enfin son intégration dans l'environnement de développement Gaspard sera présentée.

3.2.1 Cible de la chaîne de compilation : les machines à mémoires partagées

La cible de la chaîne de compilation sera les machines à mémoires partagées. L'impossibilité de faire monter en fréquence les processeurs a entraîné l'apparition des processeurs multi-cœurs mais a également permis la généralisation des machines multiprocesseurs qui deviennent de plus en plus courantes. Cette tendance est également vraie pour les systèmes embarqués qui deviennent massivement parallèles en embarquant des processeurs multi-cœurs (tel que le Tile64 [131] de Tilera qui embarque 64 cœurs). Ces machines se démocratisant, il est de plus en plus intéressant de produire du code pour les utiliser. Il existe de plus des calculateurs performants à mémoire partagée (qui sont souvent des noeuds de calculateurs à mémoires distribuées) tels que les machines IBM équipées de processeurs Power5 ou Power6 possédant des réseaux d'interconnexions à la mémoire efficaces. Pour les applications ayant besoin de beaucoup de puissance de calcul sans pour autant nécessiter de supercalculateurs, ces cibles sont très intéressantes du fait de leurs disponibilités courantes et de leurs coûts relativement faibles.

Afin de cibler de telles architectures, nous allons générer du code OpenMP qui est un standard spécialisé pour les machines à mémoires partagées. Le standard OpenMP peut être introduit dans du code Fortran et du code C, il sera donc nécessaire de générer un des deux langages. Les machines de type cluster peuvent également être ciblées par du code OpenMP en utilisant des outils tels que ClusterOMP d'Intel qui permettent de faire de la mémoire virtuellement partagée. Des travaux sont également menés afin d'utiliser le standard

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

OpenMP pour la programmation de systèmes embarqués multiprocesseurs hétérogènes. Pour cela, Liu et Chaudhary dans [91] proposent une extension du standard OpenMP permettant de gérer l'hétérogénéité des systèmes sur puces. En gérant ces extensions, la chaîne vers OpenMP pourrait également être utilisée pour les systèmes sur puces.

En plus de cibler les nouvelles architectures de machines, la génération de code OpenMP-permet de faciliter la génération puisque la gestion des communications entre processeurs est réalisée automatiquement par la mémoire partagée. Si l'on devait générer du code pour architectures à mémoires distribuées en MPI notamment, il aurait fallu gérer la communication des données entre processeurs en essayant de recouvrir la communication par les calculs.

3.2.2 Stratégie globale d'une chaîne de compilation IDM partant d'une spécification Array-OL

Le point de départ d'une approche IDM est un modèle conforme à un métamodèle. Dans le but de cibler des applications de calculs scientifiques et de faire du parallélisme, le métamodèle utilisé à haut niveau doit rendre possible l'expression du parallélisme de l'application tout en étant indépendant d'un langage d'implémentation. Comme nous l'avons vu dans le paragraphe 2.4.2, le langage Array-OL est un langage de spécification exprimant tout le parallélisme potentiel d'une application. Celui-ci n'étant lié à aucun modèle d'exécution, il est indépendant des langages d'implémentation et peut être implémenté dans n'importe lequel d'entre eux. De plus, le langage propose une notation aisément utilisable en modèle et compacte qui rend le modèle plus lisible. Le langage Array-OL est donc bien adapté pour être utilisé dans une approche IDM

Le point de départ de notre chaîne de compilation pour le calcul haute performance sera donc un modèle UML dans lequel tout le parallélisme potentiel est exprimé à l'aide de la spécification Array-OL. Ce modèle en plus d'exprimer le parallélisme de l'application doit également permettre de cibler une architecture matérielle et de spécifier une distribution des calculs sur cette architecture cible. C'est pourquoi le parallélisme potentiellement exploitable de l'architecture sera également exprimé ainsi que la distribution des calculs sur cette architecture. Array-OL ne décrivant pas le comportement des tâches de calculs, il sera également nécessaire que le modèle exprime le déploiement des tâches élémentaires applicatives sur des routines dans le langage cible. Lors de la distribution, seul le placement des tâches sur les processeurs sera pris en compte, OpenMP ne disposant pas de directives pour placer les données. Le placement des données est en effet « réalisé » par le placement des calculs, les données suivant les calculs : une tâche ayant besoin de données les fait monter dans la mémoire cache du processeur sur lequel elle est exécutée. Le placement des tâches est donc important afin de réutiliser au maximum les données qui seront dans la mémoire cache avant de changer ces données. À partir de ce point de départ, la compilation démarre afin de produire du code OpenMP. La première étape définie est de traduire les informations d'associations afin que celles-ci soient utilisables dans le code produit. C'est pourquoi la chaîne de transformation passe par un modèle polyédrique permettant d'exprimer l'association dans une forme plus facilement utilisable dans du code que la notation Array-OL. Ensuite, la transformation produit un modèle de boucle dans lequel les polyèdres sont traduits en boucles. Ces boucles expriment le placement des tâches répétées sur les différents processeurs. Après cette étape dans un modèle de boucles, un modèle de code OpenMP est généré. Ce modèle contient toute la structure du code OpenMP qui sera produit sans la syntaxe des langages puisque le métamodèle utilisé est indépendant d'un langage cible. Il s'agit d'un

métamodèle de langage procédural, celui-ci étant indépendant des langages cibles, il permet de générer du code dans un grand nombre de langages procéduraux. Enfin à partir de ce modèle de code OpenMP, du code OpenMP/C ou du code OpenMP/Fortran est généré. Le modèle d'exécution d'une application Array-OL en OpenMP est présenté plus tard (§ 3.3.1).

3.2.3 Implémentation dans Gaspard

Dans le cadre du projet INRIA DaRT, l'environnement de conception Gaspard est développé, notre chaîne de génération de code s'intègre naturellement dans cet environnement lui aussi étant basé sur le langage de spécification Array-OL à haut niveau. La figure 3.1 présente l'environnement Gaspard tel qu'il est aujourd'hui. La chaîne de compilation vers le langage OpenMP est la partie entourée dans cette figure. Le haut du modèle Y représente les modèles devant être fournis afin d'être compilés (Application, Architecture, Association, Déploiement). À partir de ces modèles fournis en UML, différentes chaînes de compilations permettent d'aller vers différentes cibles. Nous allons tout d'abord présenter la chaîne OpenMP dans Gaspard puis les différentes cibles de compilation seront introduites.

3.2.3.1 Chaîne de transformation vers OpenMP

Dans la chaîne de transformation pour OpenMP introduite dans l'environnement Gaspard, on retrouve les différentes étapes définies dans la méthodologie générale. Les différents niveaux d'abstractions sont représentés par chacun des métamodèles correspondants et des transformations de modèles automatiques s'opèrent entre ces niveaux d'abstractions. La génération des codes OpenMP/Fortran et OpenMP/C est réalisée via une phase de génération de code qui va écrire en texte le modèle OpenMP.

3.2.3.2 Les autres chaînes de transformations dans Gaspard

Ce paragraphe présente les autres chaînes de transformations actuellement disponibles dans Gaspard et positionne nos travaux par rapport à ces chaînes et à leurs cibles. Une des principales différences de ces chaînes avec celle vers OpenMP est le fait que les autres chaînes de Gaspard sont spécialisées pour les systèmes embarqués.

Génération de langages synchrones La chaîne ciblant les langages synchrones (*Synchronous*) utilise un métamodèle intermédiaire *Synchronous Equational* afin de générer du code dans les langages Lustre [66], Lucid Synchrone [119] et Signal [88]. Le but de cette chaîne développée dans le cadre de la thèse de Huafeng Yu [137] est de faire de la vérification de l'application modélisée en UML. Ce but est totalement différent du nôtre puisqu'ici seule une vérification basée sur les équations synchrones est effectuée alors que notre but est de permettre l'exécution performante des applications.

Génération d'accélérateurs matériels La chaîne ciblant le VHDL utilise un métamodèle intermédiaire *RTL* afin de produire du code permettant de synthétiser des accélérateurs matériels. Cette chaîne, développée dans le cadre de la thèse de Sébastien Le Beux [87], prend comme entrée un modèle d'application déployé sur des tâches élémentaires et génère le code VHDL permettant la synthèse de l'application sur FPGA. Cela produit des applications très performantes, mais le domaine d'application pouvant être généré pour cette cible est

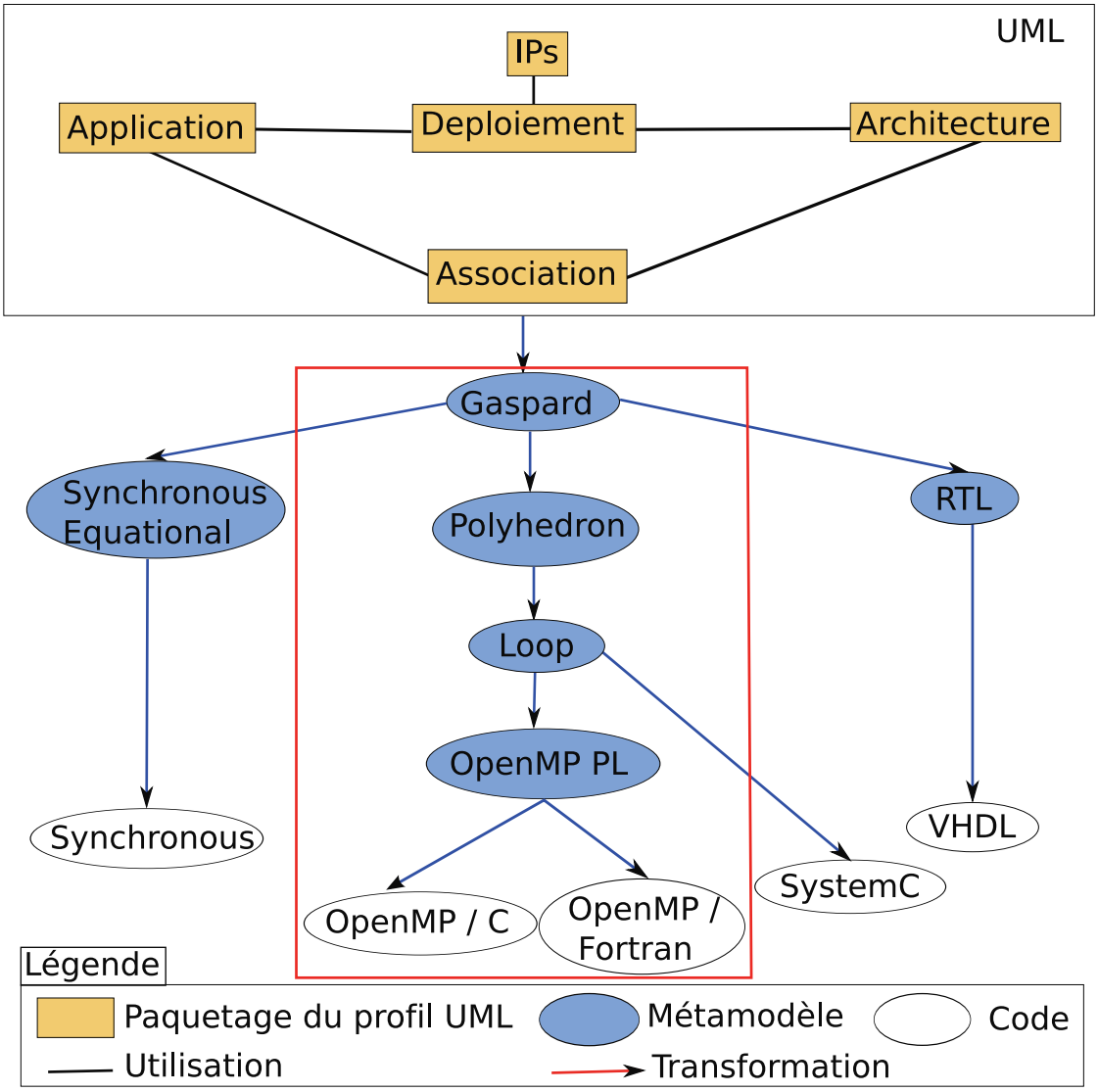


FIG. 3.1: L'environnement de développement Gaspard.

restreint par le nombre de cellules du FPGA. Cela contraint la taille des données pouvant être manipulées par cette cible. Cependant dans le cadre d'un calculateur hétérogène composé de processeurs et de FPGA, il pourrait être intéressant d'utiliser ces FPGAs pour faire des parties de calcul très particulières, des applications de calcul scientifique, pouvant être fortement accélérées par ce type de matériel.

Génération de code SystemC pour la simulation de systèmes sur puces La dernière chaîne de compilation de l'environnement Gaspard est celle permettant la génération de code SystemC [112]. Celle-ci, développée dans le cadre des thèses de Éric Piel [115] et de Rabie Ben Atitallah [14], permet la simulation conjointe logiciel/matériel d'un système sur puce. Le point de départ de cette chaîne est un modèle d'application associé à un modèle d'architecture dans lesquels les tâches élémentaires sont déployées sur des IPs matériels ou logiciels. Cette

chaîne utilise une partie de la chaîne de transformation permettant de cibler OpenMP, car au niveau de l'application les problèmes sont les mêmes que dans le cadre de OpenMP (il faut être capable de placer les tâches et les données sur les différents processeurs et mémoires). Ensuite à partir d'un modèle dans le métamodèle Loop, le code SystemC représentant le système sur puce est produit. Bien qu'il y ait une partie commune entre la chaîne OpenMP et cette chaîne, les buts de ces chaînes sont différents : le but de la chaîne SystemC est de simuler le système sur puce afin de vérifier les contraintes du système à produire (temps d'exécution, surface, consommation d'énergie ou encore contrainte temps réels) et ainsi permettre de faire de l'exploration d'architecture alors que la chaîne OpenMP produit du code utilisé pour exécuter efficacement des applications de calcul haute performance et simplifier la programmation de telles applications.

3.3 Modèle d'exécution en OpenMP des applications spécifiées en Array-OL et distribuées sur les différents processeurs et/ou cœurs

La spécification Array-OL n'étant pas liée à un modèle d'exécution, il faut en définir un afin de pouvoir exécuter ce type d'application en OpenMP. Le modèle d'exécution utilisé dans le cadre de notre chaîne de compilation va être explicité dans ce paragraphe. Le modèle d'exécution est basé sur une exécution SPMD (§ 3.3.1). La gestion des dépendances de données *tilers* est expliquée ensuite.

3.3.1 Génération de code SPMD : modèle d'exécution

Le point de départ de la chaîne de compilation est un modèle Gaspard complètement spécifié (contenant la spécification de l'application, de l'architecture, de l'association entre l'application et l'architecture ainsi que le déploiement des composants élémentaires sur des routines). Il est intéressant d'utiliser toutes les informations disponibles. Une des informations importantes à utiliser est la distribution des tâches. En effet, le modèle Gaspard exprime le placement des tâches répétées sur les différents processeurs (des threads dans le cadre de OpenMP) et ce placement joue un rôle important dans l'optique d'obtenir des codes performants. Une bonne distribution des calculs doit assurer une bonne localité des données afin d'utiliser pleinement les données montées dans le cache du processeur avant de recharger d'autres données [72]. Cette localité des données va ainsi réduire les défauts de caches dus au fait que des données nécessaires au calcul ne sont pas présentes dans la mémoire du processeur, ce qui va provoquer un accès à la mémoire centrale qui est une mémoire beaucoup plus lente que la mémoire cache. Ce transfert de données peut également provoquer des conflits d'accès au réseau d'interconnexion, phénomènes qui vont encore ralentir l'exécution du programme.

OpenMP propose des directives de compilations (exprimant les parties parallèles de l'application) ainsi qu'une clause *schedule* exprimant la manière dont sont exécutées les répétitions d'une boucle par les différents threads. Les types d'ordonnancements disponibles (*static*,*dynamic*,*guided* et *runtime*) expriment principalement des placements par blocs, cycliques ou cycliques par blocs. Le placement du parallélisme de tâches sur les threads ne peut pas être exprimé dans les constructions OpenMP(tels que les *sections* ou les *tasks* d'OpenMP 3.0). C'est le compilateur ou l'environnement d'exécution qui placera les tâches sur les threads

Thèse de Julien Taillard, Lille 1, 2009

à sa guise. Les performances liées à la localité seront donc liées à l'implémentation du compilateur et de l'environnement d'exécution utilisé ce qui est peut être gênant pour la portabilité de l'application : un code généré dans un style de programmation performant avec un compilateur/environnement pourrait ne pas l'être avec un autre. Afin d'obtenir un code ayant un placement des tâches tel que spécifié dans le modèle de haut niveau, il faut réaliser cette distribution « à la main » dans le code de l'application. C'est pourquoi nous allons générer du code dans le style SPMD (Single Program Multiple Data). Ce code, qui sera exécuté par tous les threads, contient des informations d'exécutions dépendantes du numéro des threads. Le placement des tâches est explicite dans un tel code. La vue de l'application est fractionnée sur les threads, ce qui en complexifie la lecture. Le code généré n'étant pas destiné à être lu et modifié par l'utilisateur ce n'est pas un grand inconvénient.

En plus de permettre de respecter l'association, le code SPMD est le modèle d'exécution classique en MPI. Une partie des travaux (transformations et métamodèles intermédiaires) réalisés pour la génération de code OpenMP pourrait donc être utilisée pour cibler les machines à mémoires distribuées en utilisant la bibliothèque de communications MPI.

Il est également à noter que la génération de code OpenMP aurait pu également ne prendre en compte que le modèle d'application et générer le code à partir de ce modèle en incluant aux bons endroits les directives de compilations et notamment des directives !$OMP PARALLEL DO spécifiant les boucles parallèles lorsqu'un composant répété exprimant le parallélisme de données est identifié dans le modèle. Les directives OpenMP ne garantissant pas le placement des calculs sur les threads, le placement explicite a été préféré.

3.3.2 Parallélisme de tâches

Le parallélisme de tâches en Gaspard est exprimé sous la forme d'un graphe. Deux tâches n'ayant pas de dépendances de données entre pouvant être exécutée en parallèle. Les tâches élémentaires étant réalisées par des routines dans le langage cible (C ou Fortran pour OpenMP), le point essentiel est la gestion du placement de ces tâches. La figure 3.2 illustre le placement en SPMD de deux tâches sur des threads différents. p0 étant une variable privée contenant, pour chacun des threads, son numéro, le code contenu dans la section conditionnelle vérifiant que p0 vaut 0 ne sera exécuté que par le thread ayant le numéro 0.

```
1   !p0 est le numero du thread

    if (p0 == 0) then
         !code gerant la tache affectee au thread 0
5   end if

    if (p0 == 1) then
         !code gerant la tache affectee au thread 1
    end if
```

FIG. 3.2: Code SPMD du parallélisme de tâches.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

3.3.3 Parallélisme de données

Le parallélisme de données est exprimé à l'aide de tâches répétées. Le code généré pour de telles tâches doit donc gérer la répartition des répétitions de la tâche sur les répétitions des processeurs. Cela revient à générer une boucle dont les indices dépendent de l'indice du thread. Un tel code est illustré dans la figure 3.3. Dans ce programme, chaque thread va exécuter la boucle, mais les indices de boucles utilisés par les threads sont dépendant du numéro de chaque thread. Ainsi, le thread 0 va exécuter la boucle de l'indice 0 à l'indice 1 alors que le thread 1 exécutera la boucle de l'indice 2 à l'indice 3 et ainsi de suite.

```
1   !p0 est le numero du thread

    do i=2*p0,2*(p0+1)-1
      !code de la tache
5   end do
```

FIG. 3.3: Code SPMD pour du parallélisme de données.

3.3.4 Synchronisation des tâches

Afin d'obtenir un résultat cohérent, une synchronisation des calculs est nécessaire lorsqu'une tâche doit utiliser les données produites par une autre sur un autre processeur. Il faut alors s'assurer que les données ont bien été produites avant de commencer le calcul suivant. En respectant le graphe de tâches hiérarchique modélisé dans le modèle d'application et en analysant les dépendances de données, des barrières de synchronisations seront introduites lorsque deux tâches ayant une dépendance de données ne sont pas exécutées dans le même thread.

3.3.5 Gestion des tilers

Un autre point important du modèle d'exécution est la gestion des tilers provenant du modèle Array-OL. Pour rappel, ces connecteurs expriment la manière dont est réalisé l'accès et la production des données dans les tableaux d'entrée et de sortie par les tâches parallèles répétées. Une implémentation naïve de ces connecteurs est de créer un tableau intermédiaire pour chaque motif utilisé. Cela revient donc à remplir un motif à chaque niveau de hiérarchie utilisant un tiler. Cette solution naïve provoque l'allocation potentielle d'un grand nombre de variables (dans le cadre d'une hiérarchie profonde) et de beaucoup de recopies mémoires puisqu'à chaque niveau de hiérarchie, une recopie du motif dans le tableau source est effectuée dans la variable cible. Une telle implémentation est illustrée dans la figure 3.4.

Toutes ces variables et recopies mémoires diminuent les performances, c'est pourquoi le concept de court-circuit a été développé dans les thèses de Philippe Dumont [48] et Julien Soula [128]. Le but du court-circuit est de ne pas déclarer les motifs internes inutiles (ils prennent de la mémoire et engendrent des recopies alors qu'ils ne sont pas utilisés directement pour le calcul). Pour cela, on court-circuite les motifs intermédiaires. Dans le cas de deux hiérarchies imbriquées, le but est de calculer le motif interne (le plus profond dans la hiérarchie) à partir du motif de plus haut niveau sans utiliser le motif intermédiaire. C'est pourquoi le court-circuit produit un seul tiler à partir des deux qui permet de calculer

Thèse de Julien Taillard, Lille 1, 2009


```
do i=1,4
  !Remplissage d'un motif intermediaire
  do j=1,4
    !Remplissage d'un motif intermediaire
    !Gestion de la tache repetee
  end do
end do
```

FIG. 3.4: Implémentation naïve des tilers.

les points du motif interne dans le motif externe sans passer par le motif intermédiaire. Une implémentation de tiler ayant utilisé le court-circuit est illustrée à la figure 3.5.

```
do i=1,4
  do j=1,4
    !Remplissage d'un motif
    !Gestion de la tache repetee
  end do
end do
```

FIG. 3.5: Implémentation d'un tiler à l'aide du court-circuit.

On peut remarquer que cette implémentation n'utilise qu'une variable intermédiaire à la place de deux (figure 3.4). Les formules expliquant la construction du court-circuit seront données plus tard, lors de l'explication de la transformation du modèle Loop vers un modèle de langage procédural (Chapitre 6, § 6.4.4.1).

Comme cela sera signifié dans la phase de transformation de code, il est possible, selon les langages d'implémentation, de ne pas avoir à utiliser de variables en spécifiant le motif directement dans le tableau. Cela est valable dans le cas où le motif à construire est régulier et n'utilise pas la propriété torique des tableaux Array-OL.

3.4 Conclusion du chapitre

Dans ce chapitre, nous avons présenté la méthodologie introduite lors de cette thèse. Cette méthodologie d'ingénierie dirigée par les modèles permet de bénéficier des avantages de cette dernière pour le calcul haute performance. Ces bénéfices se situent aussi bien au niveau du langage parallèle de haut niveau que du compilateur. En effet, le langage parallèle (le métamodèle) et la représentation graphique des modèles autorisent une montée en niveau d'abstraction de la spécification qui cache un grand nombre de détails d'implémentations et rend ainsi les modèles indépendants des langages d'implémentations qui seront utilisés ensuite. Pour le compilateur, la décomposition de la transformation simplifie celle-ci et permet de réutiliser des parties de ce compilateur pour d'autres cibles de compilation. Cette méthodologie a été introduite dans le cadre de l'environnement Gaspard et un modèle d'exécution des applications Array-OL modélisées en Gaspard a été défini afin d'exploiter les machines à mémoires partagées.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Chapitre 4

Extension de l'expressivité à haut niveau

4.1 Introduction d'un composant itératif **62**
4.1.1 Modélisation d'un composant itératif 63
4.1.2 Idées de compilation des composants itératifs 67
4.1.3 Utilisation d'un tel composant itératif 68
4.2 Spécification de port en entrée / sortie **68**
4.2.1 Interface d'utilisation de fonctions de bibliothèques 69
4.2.2 Modélisation de port permettant de faire de l'entrée/sortie 70
4.2.3 Idées pour la compilation du concept 72
4.3 Utilisation de données du contexte d'exécution **73**
4.3.1 Besoins 73
4.3.2 Modélisation 73
4.3.3 Compilation 75
4.4 Introduction de la généricité dans les modèles **75**
4.4.1 Les templates UML 76
4.4.2 Un patron d'introduction de la généricité dans les métamodèles à base de composants 79
4.4.3 La généricité dans le métamodèle Gaspard 80
4.5 Conclusion du chapitre **84**

À l'origine, Array-OL et Gaspard étaient prévus pour un domaine d'application restreint : le traitement de signal intensif. Une des particularités de ce domaine est que les applications sont très répétitives et très régulières. La spécification Array-OL, pensée pour ce domaine dès le départ, permet de spécifier les applications de ce domaine.

Dans le cadre de cette thèse, nous étendons le domaine d'application visé aux applications de calculs scientifiques ce qui étend considérablement le domaine d'application. Cette extension du domaine d'application implique que de nouveaux types d'algorithmes doivent pouvoir être spécifiés (tels que les algorithmes de résolution de systèmes linéaires). C'est pourquoi il est nécessaire d'étendre le pouvoir d'expression dans le modèle de spécification, dans notre cas UML avec le profil Gaspard, afin de modéliser ces nouveaux types d'algorithmes, mais également de spécifier des comportements permettant d'obtenir des codes générés performants et ne pouvant pas être déduit de la spécification.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

Afin de cibler les applications du domaine du calcul scientifique haute performance, il est nécessaire d'augmenter le pouvoir de modélisation en Gaspard. Une des notions essentielles pour le calcul haute performance non présent dans le profil Gaspard est le fait de pouvoir modéliser des boucles itératives (dont le domaine d'itération n'est pas connu à l'avance, celui-ci dépendant des données utilisées à l'exécution), alors que dans la spécification inspirée d'Array-OL seule la répétition d'une tâche dans un domaine connu peut être modélisée. L'introduction d'une telle modélisation est présentée dans le paragraphe 4.1.

Le langage Array-OL est un langage à assignation unique, cela implique que les données peuvent être écrites une seule et unique fois dans chaque espace mémoire. Ainsi, un composant élémentaire possède des ports d'entrée sur lesquels arrivent les données, et produit ses données dans les ports de sortie. Avec cette spécification, il est interdit d'écrire des données dans les ports d'entrée. Cela limite l'utilisation directe des fonctions de bibliothèques scientifiques, celles-ci prenant souvent des données en entrée/sortie. Une modélisation permettant de spécifier des ports en entrée/sortie est présentée dans le paragraphe 4.2.

Dans le cadre de certains algorithmes, certaines tâches ont besoin d'avoir une certaine connaissance du contexte d'exécution afin de réaliser leurs calculs. Cela est notamment vrai dans le cadre de tâches répétées ayant besoin de connaitre l'indice de répétition qu'elles effectuent afin de réaliser le calcul. L'introduction d'un mécanisme autorisant les tâches à connaitre l'indice de répétition qu'elles exécutent est présentée dans le chapitre 4.3.

L'un des arguments pour l'utilisation de modèles est de permettre la réutilisation d'un modèle complet ou de certains composants dans d'autres modèles. En Gaspard, un composant possède des ports en entrée et sortie ayant des dimensions fixes et peut être composé par des instances de composants répétées un nombre de fois fixe également. Ces valeurs fixes limitent fortement la réutilisation des modèles dans le sens où les problèmes ont un nombre d'éléments qui diffère pour chacun des problèmes. Cela restreint au minimum la réutilisation. Afin de permettre une plus grande réutilisation, le paragraphe 4.4 présente une introduction de la généricité dans le profil Gaspard ainsi que dans le métamodèle Gaspard autorisant la réutilisation d'un même modèle pour des nombres d'inconnues et d'éléments différents. Ce point augmente grandement la possibilité de réutilisation des modèles.

Lors de l'introduction de ces concepts de modélisation, des idées permettant la génération de code et/ou la transformation vers un modèle permettant la génération de code seront présentées. Ces idées seront utilisées dans les chaînes de transformation de l'environnement Gaspard pour la génération de code OpenMP.

4.1 Introduction d'un composant itératif

La spécification Array-OL, basée sur le fait que les applications de traitement de signal intensif sont régulières et répétitives, restreint fortement les possibilités de spécification à des algorithmes dans lesquels le nombre de répétitions d'une tâche et les tailles (des données) sont fixés à la spécification. Le fait que les tailles de données soient fixées ne pose pas de problème particulier (pour une application) mais limite la réutilisation, c'est pourquoi la généricité des composants est présentée dans le paragraphe 4.4. Le fait que le nombre de répétitions d'une tâche soit fixé ne convient pas à un grand nombre d'algorithmes de calcul scientifique, notamment à tous les algorithmes itératifs de résolution de système linéaire qui sont couramment utilisés en calcul scientifique.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

Ces méthodes de résolution itératives, utilisées lorsque le problème ne peut pas être résolu en utilisant des méthodes directes, ont la particularité (pour Gaspard) d'avoir un domaine d'itération non connu à l'avance, celui-ci étant dépendant des valeurs calculées. De plus, itération signifie également que des données produites à une itération seront réutilisées en entrée de l'itération suivante. Il est donc également nécessaire de spécifier le retour en entrée des données produites en sortie en indiquant de quel port de sortie proviennent les données sur un port d'entrée.

4.1.1 Modélisation d'un composant itératif

Comme le terme « itératif » le spécifie, un composant itératif va imposer une séquentialité entre les répétitions de pas d'itération. Les données produites à l'itération N sont nécessaires au calcul de l'itération N+1 : la séquentialité est évidente. Les répétitions du composant sur lequel on itère ne pourront donc pas être distribuées sur différents processeurs, mais les tâches constituant le pas d'itération pourront éventuellement être exécutées en parallèle.

Avant de présenter l'introduction de composant itératif dans le métamodèle Gaspard et dans le profil Gaspard, nous allons résumer ce que doit permettre de modéliser précisément ce type de composant et présenter le choix de modélisation effectué. Ensuite l'introduction de ce type de composant dans le métamodèle Gaspard (paragraphe 4.1.1.3) et dans le profil Gaspard (paragraphe 4.1.1.4) sont présentés.

4.1.1.1 Besoins de modélisation

L'objectif de ce paragraphe est de présenter les différents points importants que doit offrir la modélisation mise en place. Le premier point à prendre en compte lors de la spécification d'un algorithme itératif est que l'on veut modéliser la répétition séquentielle d'une tâche jusqu'à ce qu'une condition d'arrêt de la répétition soit atteinte. Il est ainsi nécessaire de modéliser cette répétition et cette condition d'arrêt. Afin que la condition puisse être évaluée au début de l'itération et à chaque pas d'itération, il est nécessaire que la condition porte sur des valeurs initialisées avant le début de l'itération et recalculées à chaque pas d'itération (afin que l'itération puisse se terminer). L'autre besoin essentiel d'une telle modélisation est de permettre à des valeurs utilisées en entrée de l'itération et mises à jour à chaque pas d'itération de revenir en entrée de l'itération suivante. Il faudra donc être capable de spécifier que des données produites sur les ports de sortie d'une instance de composant doivent être prises en entrée pour l'itération suivante.

Afin de pouvoir utiliser ces concepts lors de la modélisation et de la compilation, il est nécessaire d'introduire ces concepts dans le profil Gaspard (pour la modélisation en UML) et dans le métamodèle Gaspard.

4.1.1.2 Choix de modélisation

Des travaux dans l'équipe envisagent et formalisent des dépendances inter-répétitions [22, 61]. Celles-ci expriment une dépendance de données entre les répétitions d'une tâche afin d'exprimer un semi-ordre d'exécution entre répétitions. Cette modélisation permet d'exprimer des algorithmes dont une partie peut être réalisée en parallèle, mais dans lesquels certaines répétitions ont besoin du résultat produit par d'autres afin d'être calculées. Ce type de dépendance inter-répétition est illustré dans la figure 4.1 qui représente le calcul d'une intégrale discrète sur un flux arrivant. Une tâche effectuant la somme de deux nombres est

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

répétée une infinité de fois. Pour chaque valeur arrivant sur le flux, le composant effectue la somme de cette valeur avec celle calculée précédemment et sort une valeur à chaque temps. Le connecteur partant du port de sortie du composant + et revenant sur le port d'entrée spécifie la dépendance inter-répétition. Cette dépendance inter-répétition contient un vecteur d exprimant les dépendances entre répétitions. Ici d vaut (1), ce qui signifie un comportement séquentiel. Lors de la première exécution de la tâche + aucune valeur ne peut venir de la répétition précédente, il y a donc un mécanisme permettant d'initialiser le système en utilisant une valeur par défaut pour la première répétition. Cette valeur d'initialisation est souvent la valeur neutre de l'opération (dans le cas d'une addition, il s'agit de 0).

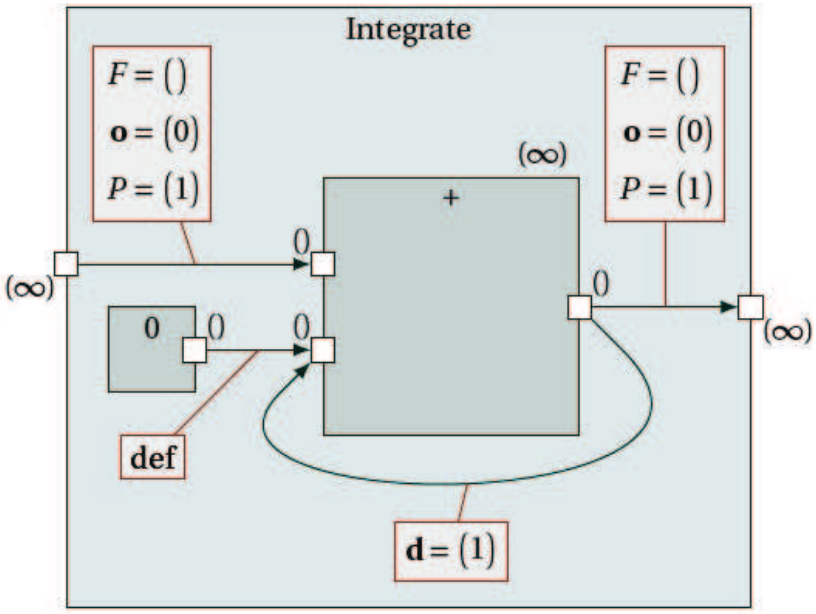


FIG. 4.1: Illustration d'une dépendance inter-répétition introduite dans la spécification Array-OL. Ici est représentée une intégration discrète sur un flux de données.

Une approche de ce type aurait pu être utilisée, mais cette solution a l'inconvénient que le domaine d'itération de la tâche répétée est toujours fixé avant l'exécution. Le nombre d'itérations d'un algorithme itératif est, par nature, inconnu avant l'exécution puisque ce nombre dépend principalement de la valeur des données utilisées. L'utilisation de dépendances inter-répétition n'est donc pas valide dans notre cas.

Le choix de modélisation réalisé repose sur le fait que la modélisation doit être simplement compréhensible afin d'être utilisé par toute personne ayant besoin de ce concept sans forcément être un informaticien. Il est important que cela puisse être utilisé simplement afin de réaliser des simulations numériques. Un composant itératif est défini comme étant un composant contenant une seule tâche (la tâche sur laquelle celui-ci va itérer), et ayant une condition d'arrêt (l'invariant du calcul) permettant l'arrêt des itérations une fois que la condition est vérifiée. La condition d'arrêt doit dépendre de données entrantes dans le composant itératif puisque celle-ci doit pouvoir être testée avant même la première itération. La condition dépend donc d'au moins une donnée provenant de l'entrée du composant itératif. Une fois que l'on est capable d'exprimer le composant itératif et la condition d'arrêt de l'itération, il reste à exprimer comment les données produites par une itération peuvent revenir en

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

entrée de l'itération suivante. Ces dépendances entre les données produites et les données consommées sont exprimées par des connecteurs entre les ports de sortie et ceux d'entrée. Un nouveau type de connecteur va donc exprimer cette dépendance. Ce type de connecteur peut être vu comme une dépendance inter-répétition ayant un vecteur de dépendance de (1). Le comportement des données est spécifié comme suit. Lors de la première itération, les données en entrée sont toutes prises sur les ports du composant itératif. Ensuite, si le port d'entrée du composant itéré est relié à un port de sortie de ce même composant alors on utilise les données provenant du port de sortie, sinon on réutilise les données spécifiées sur le composant itératif. En sortie, le composant itératif ne sort des valeurs sur ses ports de sortie que lorsque l'itération sur son composant interne sera finie.

Le principe d'assignation unique, lié à Array-OL, n'est pas entièrement respecté par ce concept. En effet, lors de chaque pas d'itération, le composant sur lequel on itère va écrire les données produites sur ses ports de sortie et certaines de ces sorties seront recopiées dans les ports d'entrée pour être utilisées au pas d'itération suivant. Le composant sur lequel on itère écrit donc autant de fois dans ses ports de sortie que de fois où l'on itère dessus. À l'intérieur d'un composant itératif l'assignation unique n'est donc pas respectée. Par contre le composant itératif n'écrira dans ses ports de sortie que lorsque l'invariant de calcul ne sera plus vrai, il écrira donc une unique fois dedans. Au niveau global de la spécification, l'instance d'un composant itératif pourra donc être vue comme étant à assignation unique. Ce non-respect local de l'assignation unique ne devrait donc pas poser de problème.

4.1.1.3 Vue domaine : introduction dans le métamodèle Gaspard

Ce paragraphe présente la vue domaine de la modélisation d'un composant itératif et son intégration dans le métamodèle Gaspard. Afin de respecter au maximum le métamodèle Gaspard tel qu'il fut défini, mais également de conserver les transformations de modèles déjà existantes sur des modèles Gaspard, il est nécessaire que la base du métamodèle reste inchangée. L'ajout d'un concept, tel que nos composants itératifs, doit donc étendre le métamodèlesans le modifier.

Dans le métamodèle Gaspard (présenté dans le chapitre 2 paragraphe 2.4.2), un composant contient une structure interne permettant de spécifier son comportement. Ainsi, un composant peut être ELEMENTARY, COMPOUND ou encore REPETITIVE. Un composant itératif étant un composant au comportement différent des trois types présents, l'introduction d'une nouvelle structure interne s'impose. L'ajout de cette structure interne est illustré dans la figure 4.2.

La structure interne introduite pour les composants itératifs est nommée ITERATIVE, elle hérite de la notion de INTERNALSTRUCTURE de Gaspard (tout comme les concepts de base ELEMENTARY et COMPOUND). Cette structure interne contient en attribut une chaîne de caractère représentant l'expression de la condition d'arrêt d'itération. Une structure interne ne peut contenir qu'une seule instance de composant (COMPONENTINSTANCE). Cette instance de composant représente la tâche sur laquelle le calcul va itérer. Une structure ITERATIVE contient également des connecteurs qui peuvent être les CONNECTORS classiques représentant les dépendances de données, mais également les connecteurs spéciaux permettant de spécifier quelles sont les données produites par le pas d'itération et qui doivent être réutilisées en tant qu'entrées du pas d'itération suivant. Ce nouveau type de connecteur est illustré dans la figure 4.3.

Le connecteur spécifiant les données produites par le pas d'itération et devant revenir en entrée de l'itération suivante est nommé COPY. Il spécifie que les données produites doivent

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

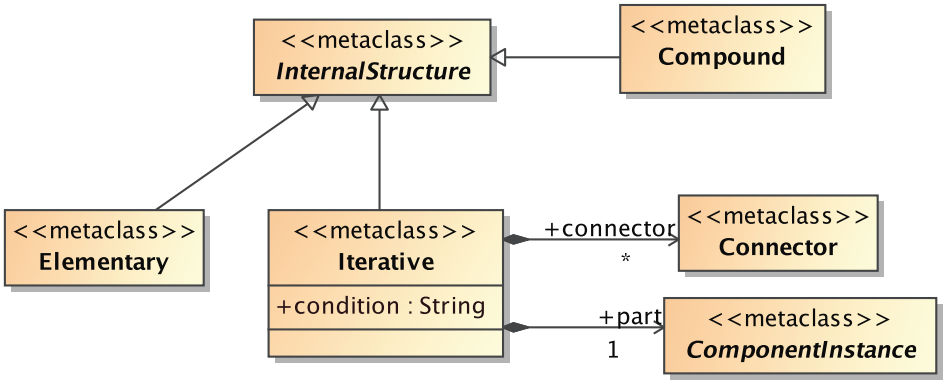


FIG. 4.2: Structure interne d'un composant itératif.

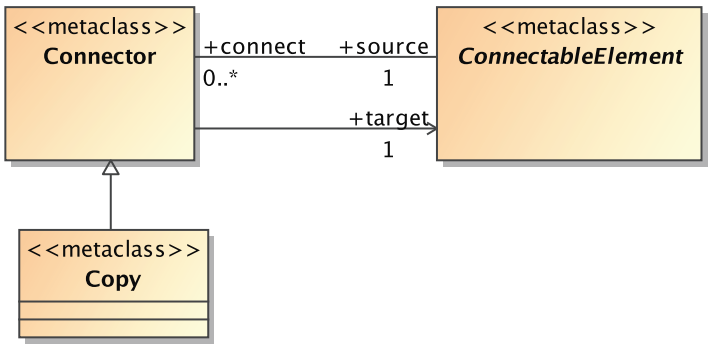


FIG. 4.3: Connecteur copy.

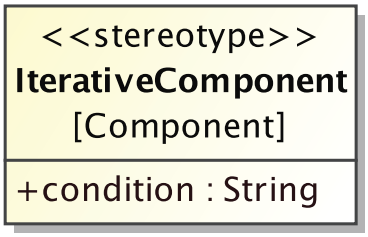
être « copiées » dans les ports d'entrée avant de recommencer une itération. Ce concept de COPY généralise le concept de CONNECTOR du métamodèle Gaspard : son rôle est celui d'un connecteur (relié deux ports) mais ayant une sémantique différente des connecteurs classiques.

4.1.1.4 Vue UML : le profil

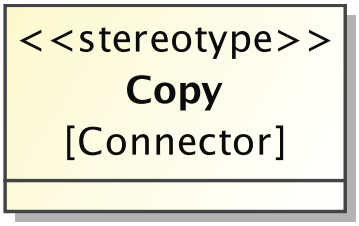
L'interface de saisie de l'environnement Gaspard est un modeleur UML, il est donc nécessaire d'introduire les composants itératifs dans le profil Gaspard afin de pouvoir les modéliser dans le métamodèle UML (le profil étant le moyen d'ajouter de la sémantique aux concepts UML), cela se fait via la définition de stéréotypes. Afin de spécifier un composant itératif, un stéréotype « ITERATIVECOMPONENT » s'appliquant aux composants UML (COMPONENT) a été introduit dans le profil. Ce stéréotype contient une *tagged value* (qui est un attribut d'un stéréotype) représentant la condition stockée sous forme de chaîne de caractères. Afin de modéliser les connecteurs COPY, un stéréotype « COPY » s'appliquant aux connecteurs a également été créé. Ces concepts sont représentés dans la figure 4.4.

L'utilisation du profil défini ici est illustrée dans le paragraphe 4.1.3 sur un exemple simple. La modélisation d'un algorithme itératif de résolution de système, le gradient conjugué, est présentée dans le chapitre 7.

Thèse de Julien Taillard, Lille 1, 2009



(a) Stéreotype IterativeComponent



(b) Stéreotype Copy

FIG. 4.4: Les stéréotypes introduit dans le profil UML Gaspard.

4.1.2 Idées de compilation des composants itératifs

La première transformation réalisée dans l'environnement est de passer du métamodèle UML avec profil vers le métamodèle Gaspard. L'ajout de ce concept ne pose pas de problème à cette transformation, puisque qu'un « IterativeComponent » sera transformé en composant Gaspard ayant une structure interne de type ITERATIVE. Les connecteurs stéréotypés « Copy » seront eux transformés en connecteurs COPY du métamodèle Gaspard. Dans les modèles intermédiaires de la chaîne de compilation de Gaspard, le concept de ITERATIVE reste inchangé puisque celui-ci étant séquentiel, il n'y pas de besoin d'effectuer d'opérations afin de le compiler. La compilation de ce concept est relativement simple puisque directe. La phase de passage d'un modèle à base de composant vers un modèle de code change toute la structuration. La structuration d'un composant ayant une structure ITERATIVE change également. Un composant itératif représentant une boucle *Tant que*, il faut alors créer une telle boucle dans le langage souhaité ayant en condition d'arrêt celle spécifiée dans le composant. Les références aux ports d'entrée du composant dans la condition d'arrêt doivent être remplacées par les variables affectées à ces ports afin que la condition soit valide. Dans le corps de cette boucle doivent être introduites les structures de code nécessaires à la gestion de l'instance du composant itéré ainsi que les copies de données produites en sortie sur les données prises en entrée par le composant itératif afin que l'itération suivante prenne en compte les bonnes valeurs.

La schématisation d'un code Fortran généré à partir d'une telle compilation du concept de composant itératif est illustrée dans la figure 4.5.

```
...
do while (condition)
  !Code genere pour le pas d'iteration
  !Copies des donnees produite par le pas dans les entrees du pas suivant
end do
..
```

FIG. 4.5: Structure d'un code généré pour un composant itératif.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

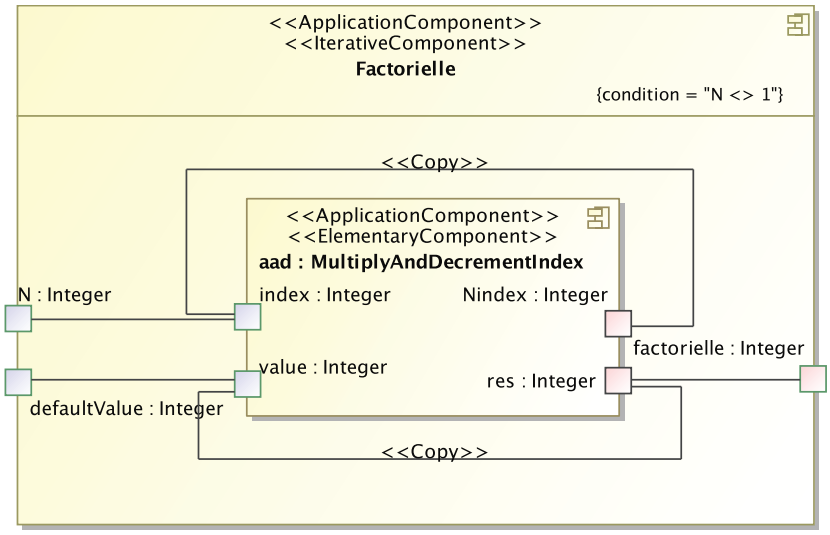


FIG. 4.6: Modélisation du calcul d'une factorielle.

4.1.3 Utilisation d'un tel composant itératif

Afin d'illustrer la modélisation d'un composant itératif, nous avons modélisé une opération simple : le calcul d'une factorielle. La modélisation est donnée dans la figure 4.6. L'algorithme modélisé consiste à multiplier par N avant de multiplier par N-1 et ainsi de suite jusqu'à arriver à 1, il faut alors arrêter le calcul. Le composant *Factorielle* est un composant itératif dont la condition est que les itérations doivent continuer tant que N est différent de 1. L'instance de composant sur laquelle on itère (*aad*) prend en entrée la valeur de N (l'index) et la valeur calculée précédemment et produit en sortie N-1 ainsi que $res = value * N$. Les valeurs fournies en sortie sur les ports *Nindex* et *res* doivent revenir en entrée de l'itération suivante, c'est pourquoi les connecteurs « Copy » sont présents. À la fin des itérations (i.e quand N=1), la valeur de la factorielle est produite sur le port *factorielle*. Lors de l'instanciation du composant *Factorielle*, il faut donner sur les ports la valeur de N sur le port *N* et la valeur par défaut sur *defaultValue*. Cette dernière est, en temps normal, la valeur neutre de la multiplication, c'est à dire 1.

4.2 Spécification de port en entrée / sortie

Le calcul scientifique est un domaine largement étudié et les fonctions classiques de ce domaine ont été optimisées afin d'obtenir des programmes performants. Dans les premiers temps du calcul scientifique, il était important de permettre aux scientifiques du monde entier d'utiliser des codes existants. C'est pourquoi des initiatives ont été prises afin de mettre à disposition les différents codes (Netlib en 1984 [44]). Ces initiatives ont conduit à la définition de bibliothèques de calcul regroupant des fonctions classiques. Ces fonctions classiques, regroupées dans des bibliothèques de calculs, ont des interfaces (signatures) standards afin qu'un code puisse utiliser la bibliothèque optimisée pour l'architecture sur laquelle il s'exécute sans avoir à en modifier son code. L'une de ces bibliothèques classiques est la bibliothèque BLAS (Basic Linear Algebra Subprogram) qui fournit les routines de bases travaillant sur les vecteurs et les matrices. On y trouve les routines sur les vecteurs,

les routines matrices-vecteurs et les routines matrices-matrices. Cette bibliothèque possède de nombreuses implémentations optimisées. L'implémentation sur architecture Intel est contenue dans la MKL [3] et l'implémentation pour architecture IBM dans ESSL [2]. D'autres implémentations de cette bibliothèque sont libres et sont optimisées selon le fonctionnement du TLB (GotoBLAS [63]) ou encore selon la taille des mémoires caches en cherchant, lors de la compilation, les tailles de blocs optimales à utiliser en fonction de la machine sur laquelle la bibliothèque est compilée (ATLAS [42]). Les interfaces de fonctions fournies par les bibliothèques BLAS utilisent fortement les variables en entrée/sortie afin de limiter les copies mémoires. Dans l'optique de pouvoir utiliser proprement ce type de librairie optimisée dans le cadre de Gaspard, il faut être capable de spécifier des ports (variables) en entrée/sortie des composants élémentaires Gaspard. Dans cette section, nous présentons une façon permettant de spécifier ce type de composant tout en conservant la sémantique d'assignation unique associée aux modèles.

4.2.1 Interface d'utilisation de fonctions de bibliothèques

L'utilisation des fonctions de bibliothèques dans l'environnement Gaspard est régulièrement effectuée via la réalisation d'une interface. Cette interface est chargée de faire le lien entre la signature du composant élémentaire (ses ports d'entrée et de sortie) et la signature de la fonction de bibliothèque, les deux n'ayant pas souvent la même interface. Dans le cas de simples ajustements, tel que la dissimulation de paramètres de la fonction standard dont la valeur est toujours fixée dans notre utilisation, ce type d'interface ne pénalise pas les performances du code généré, bien que cela puisse être pénible d'écrire au moins une interface par fonction de bibliothèque. Dans le cas d'utilisation de paramètres en entrée/sortie, l'interface devra alors gérer cela en effectuant des copies mémoires et ces copies auraient alors pour effet de pénaliser les performances.

Prenons l'exemple d'une fonction de la bibliothèque BLAS : le DAXPY. Le DAXPY réalise une mise à jour de vecteur en effectuant l'opération suivante : Y = Y + αX dans laquelle Y et X sont des vecteurs de taille N de type double et α un scalaire. En respectant la sémantique d'assignation unique liée à Array-OL, on obtient alors un composant élémentaire ayant la signature présentée à la figure 4.7.

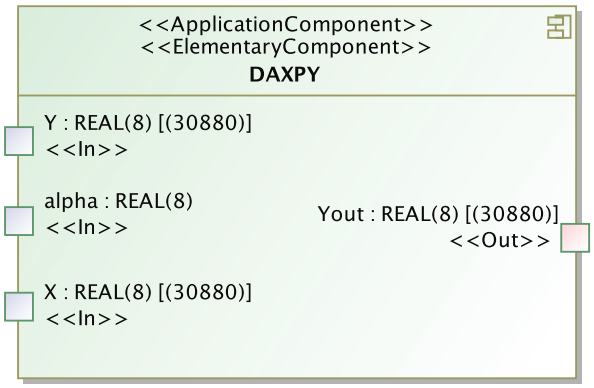


FIG. 4.7: Modélisation de l'interface classique DAXPY.

Le composant DAXPY prend alors en entrée trois valeurs (les vecteurs X et Y et le scalaire α) et produit un vecteur Yout qui doit valoir Yout = Y + αX. La signature standard de la

Thèse de Julien Taillard, Lille 1, 2009

fonction DAXPY dans la bibliothèque BLAS est la suivante :

```
1 subroutine DAXPY(N,alpha,X,XLN,Y,YLN)
```

Dans cette signature X et Y sont les vecteurs utilisés, N est un entier spécifiant la taille des vecteurs Y et X et alpha est le scalaire par lequel X est multiplié. Le vecteur Y est spécifié comme étant en entrée/sortie.

Les ports en entrée/sortie n'étant pas supportés dans les applications dans Gaspard, cela nécessite actuellement d'écrire une interface permettant d'utiliser la fonction optimisée. Cette interface, dont le code est illustré dans la figure 4.2.1, doit effectuer la copie du vecteur d'entrée Y dans le vecteur de sortie Yout avant d'appeler la fonction standard avec Yout en entrée/sortie. Cela provoque donc une copie mémoire, ce qui n'est pas bon dans l'optique d'obtenir des performances satisfaisantes. De plus, cela provoque la montée dans la mémoire cache d'un troisième vecteur (alors que seuls deux peuvent être nécessaires). Cela remplit un peu plus le cache et peut provoquer des défauts de cache qui n'auraient pas eu lieu sans la montée en cache de ce troisième vecteur.

```
1   subroutine monDAXPY(Y,alpha,X,Yout)
        Yout=Y
        call DAXPY(30880,X,1,Yout,1)
    end subroutine monDAXPY
```

Il est donc nécessaire de définir un mécanisme de modélisation exprimant des ports en entrée/sortie.

4.2.2 Modélisation de port permettant de faire de l'entrée/sortie

Il existe en Gaspard des ports en entrée/sortie qui sont actuellement utilisés principalement dans les modèles d'architecture (les données circulant dans les deux sens sur les pattes d'un composant matériel). Il serait possible d'utiliser ces ports dans un modèle applicatif. La spécification serait alors simple et le déploiement vers les interfaces de fonctions directes. L'introduction de port en entrée/sortie dans un modèle d'application entraîne le non-respect du mécanisme d'assignation unique provenant de Array-OL et le langage perd ce que permet de prouver ce mécanisme d'assignation unique (tels que l'ordonnancement possible du graphe de tâches). C'est pourquoi, nous proposons de garder le modèle d'application inchangé (spécifiant les ports en entrée et sortie sans entrée/sortie) afin de garder le modèle de spécification basé sur le mécanisme d'assignation unique. L'information que, finalement, un port est en entrée/sortie n'a besoin d'être connue que lors de l'implémentation de la spécification. Le modèle de spécification Array-OL à assignation unique ne restreint pas le modèle d'exécution à un tel modèle. Nous proposons de spécifier ces données en entrée/sortie lors du déploiement des tâches élémentaires sur les implémentations.

Les implémentations des composants présentent la signature des routines utilisées en implémentation. Un port utilisé en entrée/sortie doit être spécifié sur l'implémentation. Nous allons donc utiliser le concept de INPUTOUTPUTPORTIMPLEMENTATION qui indique que les données utilisant un tel port sont utilisées par la routine en entrée/sortie. La signature de l'interface de DAXPY comporte deux INPUTPORTIMPLEMENTATIONS en entrée (X et α) et un INPUTOUTPUTPORTIMPLEMENTATION en entrée/sortie. Cette fonction élémentaire est illustrée dans la figure 4.8.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

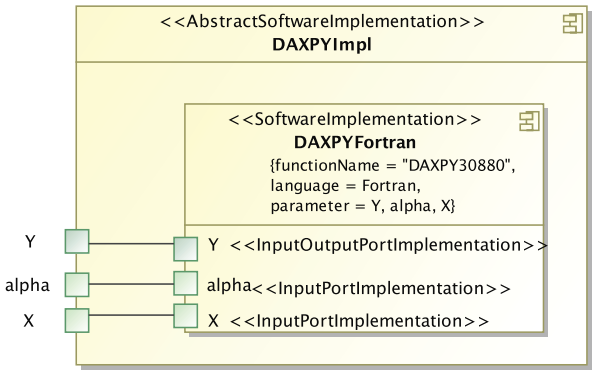


FIG. 4.8: Signature de l'implémentation de la routine DAXPY utilisant un port en entrée/sortie.

Le composant applicatif contient quatre ports alors que l'implémentation n'en spécifie que trois. Le principe pour utiliser ces ports en entrée/sortie est de déployer deux ports du composant élémentaire (un port IN et un port OUT) sur le même port d'implémentation (qui doit être un INPUTOUTPUTIMPLEMENTATION). Il faut donc être capable de déployer quatre PORTS sur trois PORTIMPLEMENTATIONS. Le déploiement d'un port d'entrée et d'un port de sortie sur le même PORTIMPLEMENTATION ne peut se faire que lorsque celui-ci est en entrée/sortie.

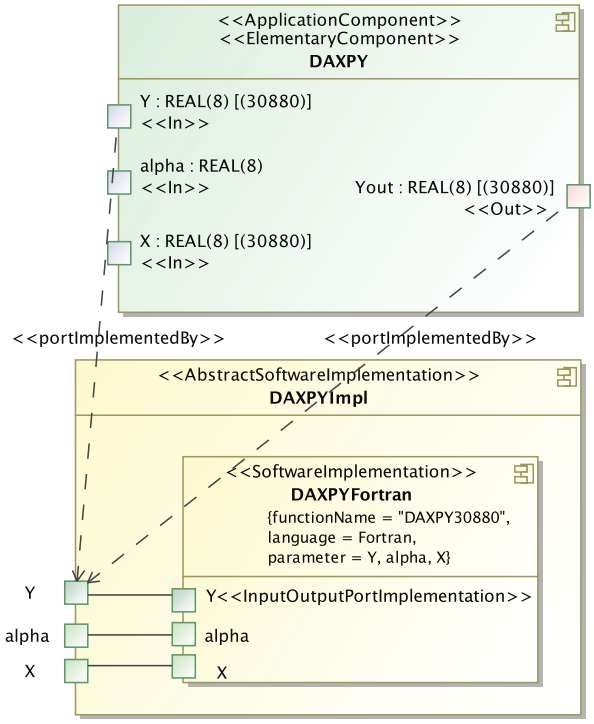


FIG. 4.9: Modélisation du deploiement de deux ports sur un même port d'implémentation.

La figure 4.9 illustre ce principe de modélisation. La modélisation du composant **DAXPY** reste inchangée avec ses trois ports en entrée et son port en sortie. C'est lors du déploiement

Thèse de Julien Taillard, Lille 1, 2009

qu'est exprimé le fait que les ports **Y** et **Yout** sont déployés sur le même PORTIMPLEMENTATION. Cette information est inutile dans la vue globale de l'application et n'est nécessaire que lors de la compilation du modèle.

4.2.3 Idées pour la compilation du concept

Les modèles sont basés sur la spécification Array-OL à assignation unique. Cette spécification n'étant pas liées à un modèle d'exécution celui-ci ne doit pas nécessairement avoir cette spécificité d'assignation unique. La spécification que deux ports sont déployés sur le même INPUTOUTPUTPORTIMPLEMENTATION doit être utilisée lors de la déclaration des variables du programme. En effet, lors de la déclaration des variables, deux ports déployés sur le même PORTIMPLEMENTATION doivent être associés à une seule et même variable. Des précautions doivent être prises en compte lors de la compilation de ce concept. En effet, selon la manière dont celle-ci est effectuée, il est possible que des cas d'incohérences soient introduits par cette utilisation de port en entrée/sortie. Un exemple de ces possibles incohérences peut provenir de l'exemple illustré dans la figure 4.10. On peut y voir trois instances du composant DAXPY, celui-ci étant déployé comme à la figure 4.9. Si l'on considère qu'à la compilation, les ports Y et Yout d'une instance de DAXPY sont associés à la même variable et que les ports d'entrée d'une instance réutilisent la variable associée au port de sortie auquel ils sont connectés via le CONNECTOR. Il est alors possible sur cet exemple que toutes les instances des ports Y et Yout soient associées à la même variable. Cela produirait un résultat faux puisque le résultat de l'instance *d3* pourrait être calculé en utilisant la valeur produite par *d2* alors que les valeurs Yout de ces deux instances doivent être différentes. Dans ce cas, il faut que la transformation déclare une variable dont le type et la dimension seront les même ceux du port *Yout*. Cette nouvelle variable se verra affecter la valeur de la variable liée au port *Yout* de l'instance *d1*. Les instances *d2* et *d3* pourront alors travailler chacune sur leur copie de la donnée.

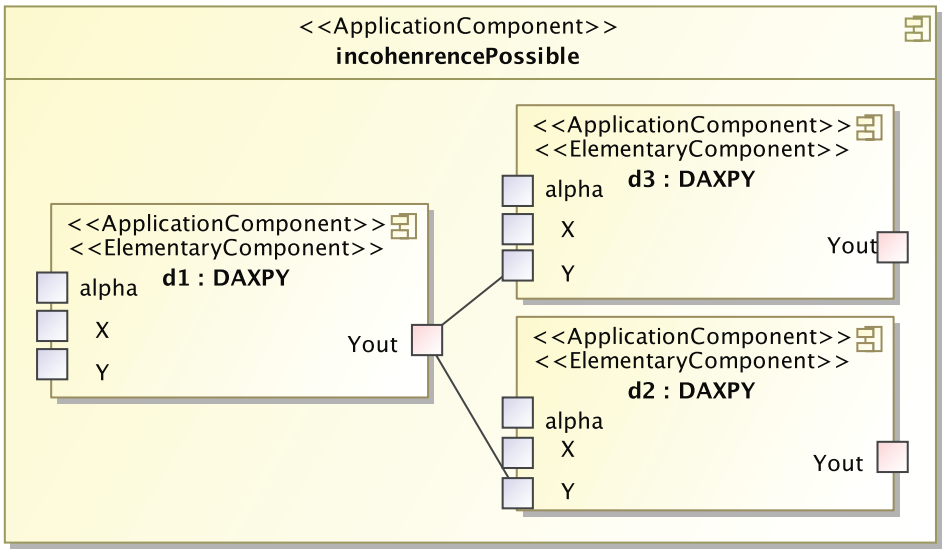


FIG. 4.10: Modèle pouvant occasionner des incohérences. La compilation de ce modèle nécessite de réaliser une copie de la variable affectée à *Yout* afin que chacune des instances *d2* et *d3* puisse travailler sur leurs copies.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

4.3 Utilisation de données du contexte d'exécution

4.3.1 Besoins

La modélisation du parallélisme de données dans la spécification Array-OL, et donc dans l'environnement Gaspard, est basée sur la répétition de tâches sur des données différentes. La tâche répétée n'a aucune connaissance du contexte dans lequel elle est et principalement ne connait pas l'indice de la répétition qu'elle exécute. Dans le cadre de tâches régulières et répétitives effectuant toujours les mêmes calculs quelque soit les données, ce mécanisme est bien adapté, mais il existe des algorithmes de tâches répétées dont le comportement (i.e les calculs effectués) dépend de la valeur d'une ou de plusieurs données. Cette donnée peut être l'indice de répétition que celle-ci exécute. L'indice de répétition n'étant pas une donnée comme une autre, elle n'est pas produite par un autre composant et ne passe donc pas par les ports, un mécanisme doit alors être introduit afin de donner la connaissance de l'indice de répétition à une tâche répétée.

4.3.2 Modélisation

Lors de la modélisation d'un composant élémentaire toutes les données consommées et produites par celui-ci sont passées via les ports. Il en est de même dans le déploiement dans lequel les « SoftwareImplementations » représentent les fonctions réalisant les composants élémentaires et les « PortImplementations » sont les paramètres passés à la fonction. La donnée modifiant le calcul effectué pouvant avoir été produite par un autre composant aussi bien que pouvant être l'indice de répétition de la tâche, la modélisation élémentaire du composant ne peut pas être changée et le composant élémentaire doit pouvoir être utilisé dans les deux cas.

La connaissance qu'une donnée provient de l'indice de répétition n'apparaît que lors de la modélisation de l'algorithme. C'est lors de la création d'une instance répétée qu'il est nécessaire que cela soit spécifiée. Cette spécification doit indiquer que l'instance va avoir besoin de la connaissance de l'indice de répétition, mais également sur quel port la donnée doit arriver. La modélisation proposée doit donc être associée aux instances de composants et non aux composants eux-mêmes qui doivent rester indépendants de la façon dont leurs arrivent les données.

Un point important à noter est que le domaine de répétition d'une tâche dépend du contexte dans lequel celle-ci se trouve. En effet, selon les tableaux d'entrée et de sortie desquels proviennent les données fournies à la tâche, le domaine de répétition varie. Les espaces de répétitions utilisés pour différentes instances du même composant peuvent avoir des dimensions différentes. L'index de la répétition que traite une tâche a le même nombre de dimensions que l'espace de répétition. Lors de la modélisation d'une tâche, la taille des ports est fixée. Dans le cadre de la modélisation présentée ici, nous considérerons que la dimension de l'indice de répétition est égale à la taille du port sur lequel la valeur doit arriver. Cette propriété doit être prise en compte par la modélisation.

4.3.2.1 Vue domaine

Le fait qu'une tâche de calcul a besoin de l'indice de répétition est contextuel et dépend de l'algorithme, c'est pourquoi le déploiement des indices sur un port se réalise lors de l'utilisation des instances de composants. L'ajout de ce mécanisme s'effectue sur le concept

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

de COMPONENTINSTANCE. Une instance de composant peut donc se voir attribuer un déploiement des indices de la répétition sur les instances de ports. Pour cela, le concept de INDICESONPORTS est introduit. Il permet de spécifier qu'une instance a besoin de la connaissance des indices de répétition et indique sur quel(s) port(s) ceux-ci sont passés. INDICESONPORTS pointe vers les différents PORTINSTANCES sur lesquels les indices doivent être passés. Ces concepts sont illustrés dans la figure 4.11.

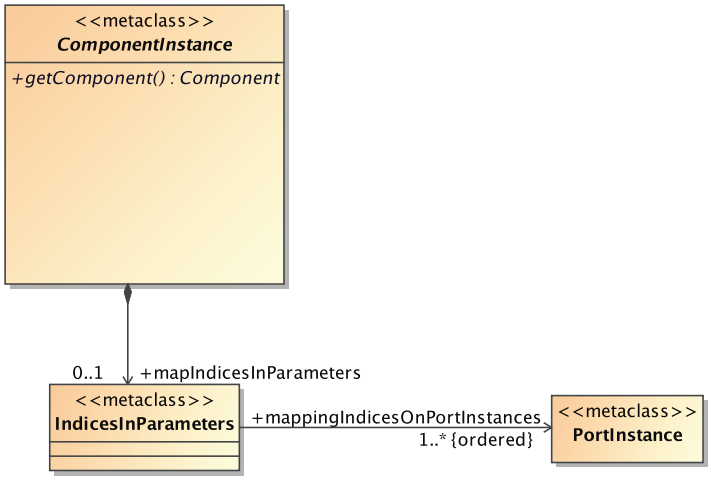


FIG. 4.11: Mécanisme permettant de spécifier le ou les ports sur lesquels le ou les indices seront déployés

4.3.2.2 Vue profil

Dans le métamodèle UML, les instances de composants n'existent pas et ce sont en fait des PROPERTYS UML (des instances potentielles) qui sont utilisées. L'ajout de ce concept se limite donc à créer un nouveau stéréotype s'appliquant sur les PROPERTYS comme cela est présenté à la figure 4.12. Le stéréotype « IndicesOnPorts » contient une taggued value indiquant le ou les ports sur lesquels les indices doivent être indiqués.

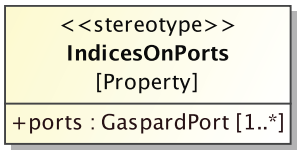


FIG. 4.12: Stéréotype introduit afin de spécifier le déploiement des indices de la répétition sur les ports.

4.3.2.3 Exemple

Afin d'illustrer l'utilisation de concept, nous avons modélisé un comportement très simple : le calcul des 10 premières exponentielles d'une valeur. Le calcul des exponentielles peut être réalisé indépendamment les unes des autres, mais la fonction exponentielle doit connaître

Thèse de Julien Taillard, Lille 1, 2009

l'indice de l'exponentielle à calculer. Afin de réaliser ce calcul, on dispose d'un composant élémentaire *exponentielle* calculant $value^N$. Le composant calculant les dix premières exponentielles est illustré à la figure 4.13. À partir d'une valeur, le composant calcule les dix premières exponentielles et les stocke dans un tableau de dix valeurs. L'instance du composant réalisant l'exponentielle est répétée dix fois. Chacune de ces répétitions doit avoir la connaissance de l'exponentielle qu'elle doit calculer sur son port N. Pour cela, l'instance *e* est stéréotypée « IndicesOnPorts » et sa taggued value pointe vers le port N.

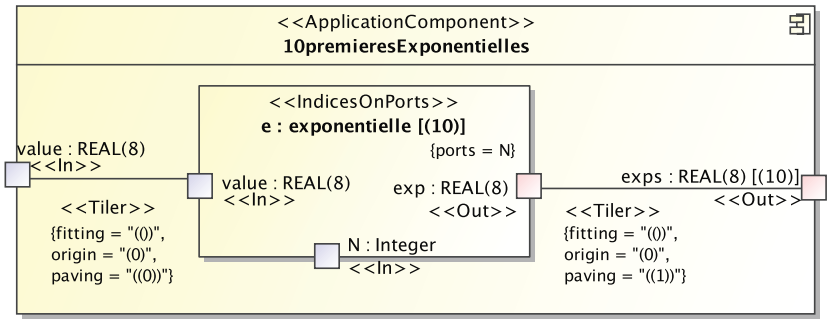


FIG. 4.13: Exemple d'utilisation du stéréotype « IndicesOnPort ».

L'utilisation de ce stéréotype sur un cas réel de modélisation parallèle d'un produit matrice creuse - vecteur sera illustrée ultérieurement.

4.3.3 Compilation

Lors de la compilation de ce concept, il sera nécessaire de retrouver les variables générées et représentant les indices de répétitions de la tâche afin de les passer en paramètre de la fonction de la tâche élémentaire ou de les associer aux instances de ports du composant si celui-ci n'est pas élémentaire.

4.4 Introduction de la généricité dans les modèles

L'un des buts de l'utilisation d'une approche IDM est l'augmentation de la productivité des développeurs. Celle-ci est obtenue par la simplification de la spécification à haut niveau, permettant de s'abstraire des détails d'implémentations, mais également par la réutilisation de modèles (ou de composants) entre différents projets. Une fois une fonctionnalité spécifiée dans un composant, tous les projets nécessitants une telle fonctionnalité doivent pouvoir réutiliser le dit-composant. Malheureusement, il arrive souvent dans le cadre de l'environnement Gaspard, qu'un composant ne puisse pas être réutilisé parce qu'une partie de la spécification ne correspond pas exactement aux besoins d'un problème donné. En effet, dans un modèle Gaspard, toutes les valeurs sont fixées à l'avance. Ainsi, des valeurs telles que la multiplicité des ports (i.e la taille des données manipulées) et le nombre de répétitions des tâches sont fixées. Or, en calcul scientifique, il arrive rarement que deux problèmes aient les mêmes spécifications (telles que le nombre d'inconnues d'un système d'équations ou le nombre d'éléments non nul dans une matrice creuse). La réutilisation de composants est alors compliquée et se limite souvent à copier le composant réalisant la fonctionnalité demandée, à le modifier manuellement afin d'adapter la taille des données et des répétitions des tâches au

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

problème particulier que l'on traite. Ce travail, en plus d'être source d'erreurs, est fastidieux pour l'utilisateur notamment dans le cas où celui-ci doit modifier toute une hiérarchie de composants. Cela limite fortement la réutilisation des composants ainsi que la productivité de l'approche IDM. L'introduction de la généricité dans les modèles est donc importante afin de rendre possible la réutilisation des modèles et composants. Cela repose sur le fait de proposer des composants paramétrables, la valeur de ces paramètres étant connue plus tard lors de l'utilisation réelle des composants.

La notion de généricité est présente dans un grand nombre de langages de programmation tels que le Java (avec les Java Generics) ou le C++ (les templates). L'utilisation de la généricité rend possible le développement de code de calcul scientifique maintenable à un niveau plus abstrait que le code classique. Il est ainsi possible de rendre abstrait le type de données manipulé (entier ou double), mais également la manière dont sont stockées les données (matrices pleines ou creuses).

La première section présente la généricité en UML basée sur les *templates*. Ensuite une méthodologie permettant d'ajouter la généricité aux métamodèles de composants est présentée. Celle-ci sera appliquée sur le métamodèle Gaspard.

4.4.1 Les templates UML

Le premier environnement de spécification d'une application Gaspard étant les modeleurs UML, il est nécessaire de définir les composants paramétrables en UML. La norme UML propose un mécanisme de généricité nommé *template*. Ce mécanisme qui va nous servir à spécifier des composants génériques dans les modèles UML Gaspard est présenté dans ce paragraphe. Il est à noter que ce mécanisme de *template* ne se limite pas à la paramétrisation de composants, mais s'applique à un grand nombre d'éléments de la norme UML. Durant cette présentation, le mécanisme est uniquement illustré dans le cadre des composants afin d'être clair et d'exprimer également ce qui doit pouvoir être modélisé dans les modèles Gaspard.

4.4.1.1 Présentation des templates UML existants

La norme UML définie les éléments pouvant être paramétrés en UML : les TEMPLATEELEMENTS. Un élément paramétrable « expose », pour reprendre le jargon UML, ces parties internes qui doivent être paramétrées via des paramètres (les TEMPLATEPARAMETER). L'élément paramétré diffère ainsi la définition de ces éléments internes à plus tard lorsque ces éléments internes se verront affecter une valeur.

La spécification d'éléments paramétrables se déroule en deux étapes : la *définition* des éléments paramétrables et le *binding* (l'affectation) qui affecte une valeur à chacun des paramètres formels.

4.4.1.2 Définition d'un composant paramétrable

La définition d'un composant paramétrable consiste à déclarer quels sont les éléments paramétrables contenus par ce composant. Pour cela, un élément paramétrable possède une TEMPLATESIGNATURE permettant de spécifier ces paramètres, de les exposer. Les paramètres peuvent être utilisés pour paramétrer des valeurs (multiplicité et taille de ports dans le cas de Gaspard), mais également des types permettant, par exemple, de laisser la définition du type d'une propriété (PROPERTY UML) à plus tard.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

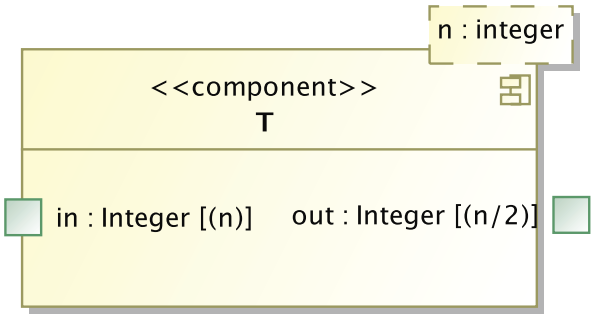
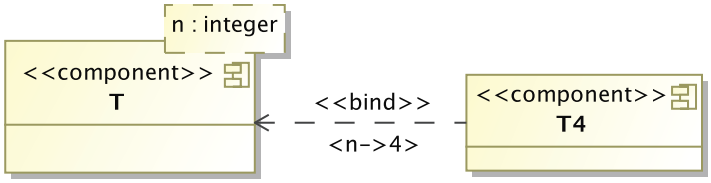


FIG. 4.14: Définition d'un composant paramétré en UML.

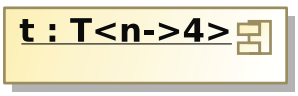
La figure 4.14 présente un composant T ayant deux ports[1]. Ce composant T possède un paramètre n. Ce paramètre expose les multiplicités des ports *in* et *out*. Ainsi lorsque le paramètre n se sera vu affecter une valeur, la multiplicité du port *in* devrait être cette valeur alors que celle du port *out* sera cette valeur divisée par 2.

4.4.1.3 Affectation de valeurs aux paramètres

L'affectation de valeurs aux paramètres se fait via le *binding*. Il consiste à affecter à chacun des paramètres une valeur. UML propose deux mécanismes afin d'effectuer cette affectation. La première solution est d'utiliser le lien d'abstraction « bind ». Cette solution, présentée à la figure 4.15 (a), nécessite de définir un nouveau composant (T4 dans l'exemple) et à modéliser le lien « bind » pour chaque ensemble de valeurs affectées aux paramètres. L'utilisation de cette solution demande de créer, pour chaque ensemble de valeurs de paramètres, un nouveau composant et de faire le lien « bind ». Cela va encombrer les diagrammes et nécessite beaucoup de manipulations. Cette solution ne convient donc pas puisque la paramétrisation de toute une hiérarchie de composant sera longue et contraignante. En effet, pour chaque ensemble de valeur, il faudra recréer une hiérarchie complète en ayant réalisé le « binding » pour chaque composant utilisé dans la hiérarchie.



(a) Affectation utilisant le lien bind



(b) Affectation utilisant l'affectation anonyme

FIG. 4.15: Les différentes options permettant l'affectation des paramètres en UML.

La seconde solution, disponible dans la dernière spécification UML et nommée affectation anonyme (*anonymous binding*), simplifie la notation d'affectation en permettant, tout comme dans la notation des *generics* de Java, d'effectuer l'affectation à la suite du nom du composant

[1] À la suite d'une erreur d'affichage de l'outil utilisé pour réaliser ces figures, le port de droite est décalé du bord du composant. Il ne faut pas y chercher de signification particulière.

Thèse de Julien Taillard, Lille 1, 2009

lors de son utilisation. Cela évite de devoir modéliser le nouveau composant et d'effectuer le lien spécifiant le *binding*. Cette notation, illustrée dans la partie (b) de la figure simplifie grandement l'utilisation des templates UML.

4.4.1.4 Problèmes de spécification et d'implémentations

Lors de la modélisation de composants paramétrables en UML, nous avons constaté deux problèmes principaux liés soit à la norme elle-même, soit à son implémentation dans les modeleurs UML. Ces problèmes limitent la manière dont peuvent être utilisés les concepts de généricité en UML. Tout en explicitant ces différents problèmes, nous donnerons une manière de contourner ces problèmes en attendant que ceux-ci soient résolus.

Le premier problème, le plus important, est un problème provenant de la spécification UML elle-même. En effet, dans le métamodèle UML, un paramètre ne peut exposer qu'un seul élément interne. Il est ainsi impossible d'exposer la multiplicité de deux ports via le même paramètre (tel que cela avait été fait dans le composant T de la figure 4.14). Ce problème, lié à la manière dont sont référencés les paramètres, est reconnu comme étant une erreur par l'OMG et devrait changer dans la future version du métamodèle UML. En attendant cette correction et son implémentation dans les modeleurs UML, les paramètres exposés sont référencés par leur nom. La multiplicité UML est écrite sous forme de texte, il est donc possible d'écrire le nom du paramètre dans la multiplicité et celui-ci devra être interprété lors de la transformation de modèle afin de récupérer la valeur qui lui aura été affectée.

L'autre problème rencontré est dû aux outils UML qui n'implémentent pas entièrement la norme. La flexibilité du métamodèle UML se paie par une complexité relative de ces spécifications qui poussent les outils à n'en implémenter qu'une partie. La notation d'affectation anonyme qui rend possible une utilisation plus simple de la généricité en UML n'est pas implémentée dans les différents outils. Une solution temporaire a été mise en place afin de pouvoir utiliser les composants paramétrables en UML en attendant que la nouvelle spécification et son implémentation dans les outils UML permettent de les utiliser proprement. Cette solution est illustrée dans la figure 4.16. La référence entre les éléments « exposés » et les paramètres de composants est effectuée via le nom des paramètres. De même, l'utilisation de la notation de « binding » nous intéressant n'étant pas encore possible, nous utilisons le même type de notation que proposé dans la norme, mais dans le nom de la PROPERTY et non dans le type de celle-ci.

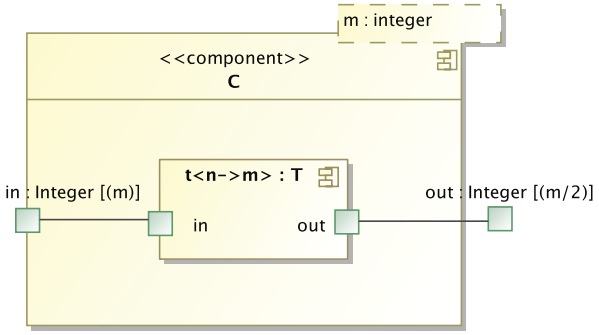


FIG. 4.16: Illustration de la solution temporaire retenue.

L'interprétation de toutes ces notations temporaires sera effectuée par nos transforma-

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

tions de modèles. Le passage à la notation standard ne devrait pas provoquer de grands changements dans les transformations du fait des notations proches qui ont été choisies.

4.4.2 Un patron d'introduction de la généricité dans les métamodèles à base de composants

La notion de réutilisation est implicite avec l'utilisation d'une approche à base de composants, l'introduction de la généricité dans les métamodèles à base de composants semble donc toute choisie afin de favoriser cette réutilisation. Le métamodèle Gaspard est un métamodèle basé sur la modélisation de composants. Le fait d'introduire la généricité dans le métamodèle Gaspard n'est pas différent que de l'introduire dans un autre métamodèle basé sur les composants. Nous proposons donc de définir un patron d'introduction de la généricité dans les métamodèles à base de composants. Ce pattern pourra être appliqué aussi bien au métamodèle Gaspard qu'au métamodèle MARTE.

Lors de la présentation des *templates* UML, nous avons vu deux étapes dans leurs utilisations : la *définition* et l'*affectation*. Ces deux étapes sont importantes et doivent être gérées avec l'introduction de la généricité dans les métamodèles à base de composants. Le patron doit être le moins intrusif possible afin de ne pas impacter les travaux déjà existants et utilisant le métamodèle sans généricité. C'est pourquoi un ensemble d'éléments génériques sont définis. Ceux-ci seront accrochés au métamodèle cible via l'utilisation de généralisations et éventuellement de références.

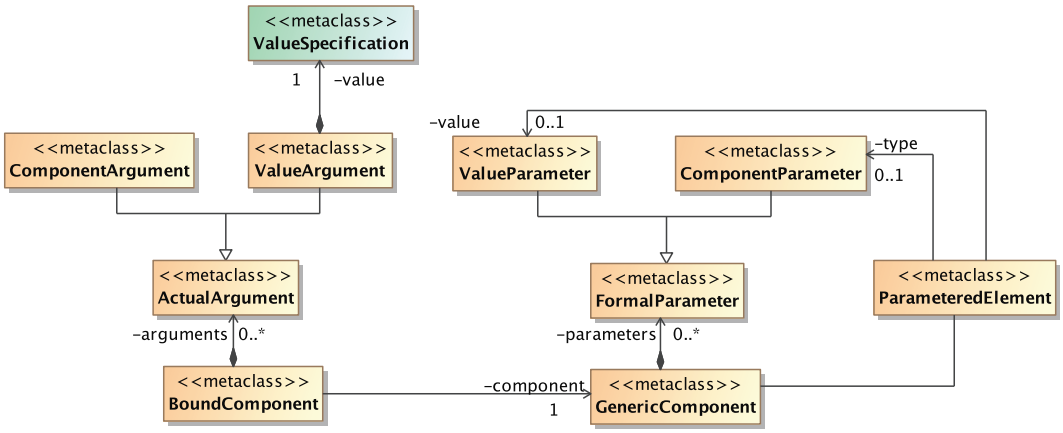


FIG. 4.17: Formalisation de l'introduction de la généricité dans un métamodèle à base de composants.

La figure 4.17 illustre le patron de conception permettant l'introduction de la généricité dans un métamodèle à base de composants. Dix concepts sont présents dans ce patron de conception. Deux types de composants , qui devront étendre le concept de composant du métamodèle cible, sont définis. Il s'agit des éléments GENERICCOMPONENT et BOUNDCOMPONENT. Ces deux concepts représentent les composants ayant des paramètres pour le premier et les composants ayant les paramètres affectés à des valeurs pour le second. La figure peut être coupée en deux selon l'objectif des éléments la composant.

La partie droite de la figure représente les concepts utiles à la définition des composants génériques. Un composant paramétrable contient des paramètres formels

Thèse de Julien Taillard, Lille 1, 2009

(FORMALPARAMETER). Ces paramètres peuvent être une valeur (VALUEPARAMETER) ou un type de composant (COMPONENTPARAMETER). Les PARAMETEREDELEMENTS représentent les éléments du métamodèle pouvant être paramétrés. Ces éléments du métamodèle de base pouvant être paramétrés seront étendus par des éléments étendant également PARAMETEREDELEMENT afin de pouvoir référencer les paramètres introduit.

La partie gauche de la figure représente les concepts utilisés pour l'affectation. Ainsi, un BOUNDCOMPONENT représente le nouveau composant créé par le mécanisme d'affectation dans lequel les paramètres ont une valeur. Un BOUNDCOMPONENT possède des ACTUALARGUMENTS qui représentent les valeurs que vont prendre les paramètres du composant paramétrable auquel fait référence le BOUNDCOMPONENT. Ces arguments donnent une valeur aux paramètres définis dans le composant paramétrable référencé. Les valeurs peuvent être soit un type de composant (COMPONENTARGUMENT) afin de spécifier le type d'une propriété paramétré, soit une valeur (VALUEARGUMENT). Une valeur peut être une expression mathématique permettant de calculer une valeur à partir de la valeur d'un des paramètres du composant contenant. Il est donc nécessaire d'avoir un langage d'expression. Un tel langage d'expression est déjà défini dans le standard de modélisation MARTE, il s'agit du Value Specification Language (VSL), nous avons donc décidé de le réutiliser. Un VALUEARGUMENT peut ainsi contenir une expression venant du VSL.

Le paragraphe suivant présente l'utilisation de ce patron de conception afin d'introduire la généricité dans le métamodèle Gaspard.

4.4.3 La généricité dans le métamodèle Gaspard

Le besoin de composants génériques s'est fait sentir lors de la modélisation de composants Gaspard. Nous allons donc l'introduire dans le métamodèle Gaspard à l'aide du patron de conception présenté dans la section précédente. L'introduction de la généricité est effectuée afin de maintenir la généricité le plus longtemps possible afin d'éventuellement la conserver dans les métamodèles inférieurs à Gaspard. Il pourrait, en effet, être intéressant de conserver les paramètres assez bas dans les chaînes de transformations. La première étape est donc d'introduire, dans un premier temps, la généricité dans le métamodèle Gaspard avant d'envisager de la conserver dans les métamodèles plus bas dans les chaînes de transformations. Ce type de composant générique pourrait notamment être utilisé afin de définir des composants ayant un comportement récursif et dont la condition d'arrêt (de la récursion) dépend de la valeur d'un paramètre à un moment donné.

Le premier paragraphe 4.4.3.1 présente la stratégie d'intégration des composants génériques dans l'environnement Gaspard, puis l'application du patron de composants génériques sur le métamodèle Gaspard est présenté. Enfin, les idées de bases de la transformation d'un modèle UML avec composants génériques vers un modèle Gaspard complètement spécifié permettant l'utilisation des chaînes de transformations de Gaspard sont introduites (§ 4.4.3.3).

4.4.3.1 Stratégie d'introduction

Les chaînes de compilation actuellement disponibles dans l'environnement Gaspard prennent en entrée des composants complètement spécifiés (i.e dont toutes les valeurs de tailles de données et de répétitions de tâches sont spécifiées). Afin de réutiliser ces chaînes de compilations, nous avons défini une stratégie permettant la production de modèles complètement spécifiés pour ces chaînes de compilation. À partir d'un modèle UML paramétré,

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

un modèle dans le métamodèle Gaspard générique (G2P), métamodèle Gaspard dans lequel la généricité a été introduite, est automatiquement produit. Ensuite l'affectation des valeurs aux paramètres est réalisée afin de produire un modèle dans le métamodèle Gaspard original (G2).

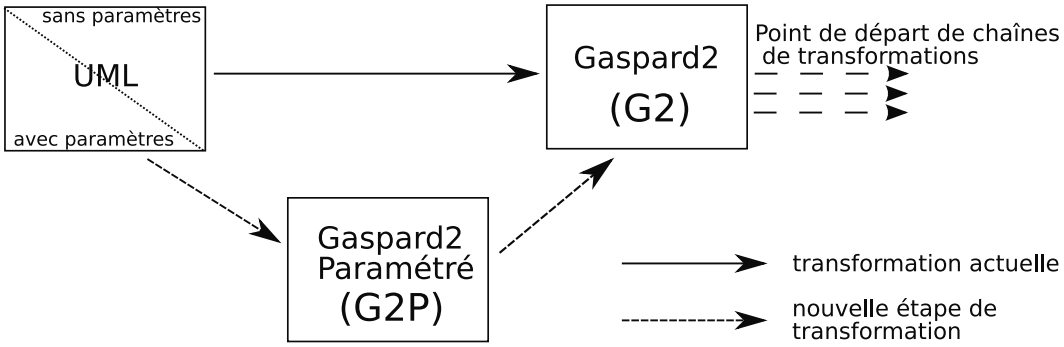


FIG. 4.18: Introduction d'un nouveau niveau de transformation afin de réutiliser les chaînes de transformations existantes.

Cette stratégie d'introduction est illustrée dans la figure 4.18. La transformation passant d'un modèle UML sans paramètres vers un modèle Gaspard est la transformation actuelle. Le nouveau niveau de métamodèle générique introduit deux nouvelles transformations : la transformation d'un modèle UML avec paramètres vers un modèle Gaspard paramétré (G2P), puis une transformation affectant réellement les valeurs aux paramètres (Gaspard paramétré (G2P) vers Gaspard (G2)).

4.4.3.2 Introduction de la généricité dans le métamodèle

L'introduction de la généricité dans le métamodèle Gaspard est basée sur le patron de conception présenté dans le paragraphe 4.4.2. Les concepts actuellement importants à paramétrer sont les multiplicités des ports et des tâches répétées. Le type d'une instance est également intéressant à paramétrer afin de changer facilement de composant juste en changeant de type. Le concept de base représentant un composant dans le métamodèle Gaspard est le concept de COMPONENT. C'est donc sur ce concept que le patron va se brancher.

La figure 4.19 illustre l'introduction de la généricité dans le métamodèle Gaspard. Le concept de base des composants Gaspard est le COMPONENT. C'est donc ce concept qu'étend le GENERICCOMPONENT. Les éléments pouvant être paramétrés sont les composants (le type d'une instance de composant) ainsi que les multiplicités. La paramétrisation des multiplicités entraîne qu'il est intéressant de paramétrer les tilers puisque ceux-ci dépendent de la taille des ports et du nombre de répétition de la tâche répétée. La métamodèle Gaspard ayant été conçu de façon très spécialisé, il est nécessaire de modifier celui-ci afin d'autoriser la généricité dans les multiplicités et les tilers.

Dans le métamodèle Gaspard original, les vecteurs (INTVECTOR), matrices (MATRIX) et multiplicités (SHAPE) sont spécifiés comme n'étant composés que d'entiers. La définition de ces concepts est illustrée dans la figure 4.20, on peut voir qu'il n'est possible de spécifier les vecteurs que comme des vecteurs d'entiers et que le concept de SHAPE comprend un tableau d'entier également. Un paramètre n'étant pas un entier, mais une référence formelle il est impossible de paramétrer ces concepts importants. Afin de permettre la paramétrisation,

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

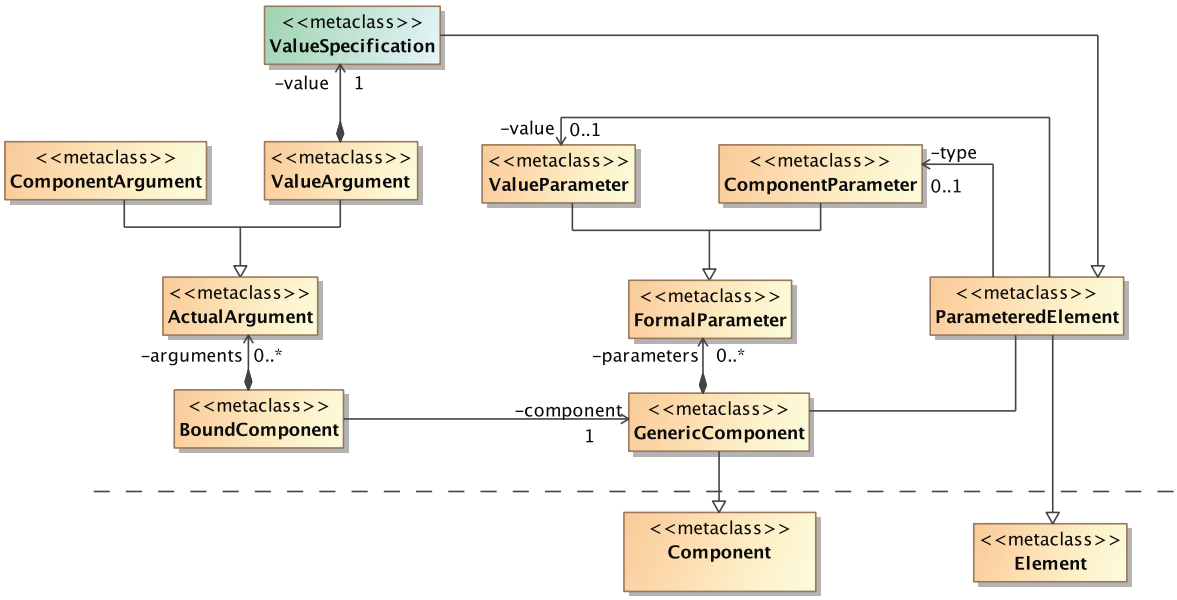


FIG. 4.19: Introduction de la généricité dans le métamodèle Gaspard à l'aide du template de généricité.

nous introduisons une légère modification de ces concepts afin de les rendre générique. Dans la nouvelle version du métamodèle (figure 4.21), une matrice (MATRIX) et une multiplicité (SHAPE) contiennent des vecteurs (un pour la multiplicité, un nombre indéterminé pour la matrice). Ces vecteurs (VECTORS) sont composés de VALUESPECIFICATION, provenant du package VSL du métamodèle MARTE, qui peuvent être des entiers, mais également des expressions basées sur les paramètres notamment.

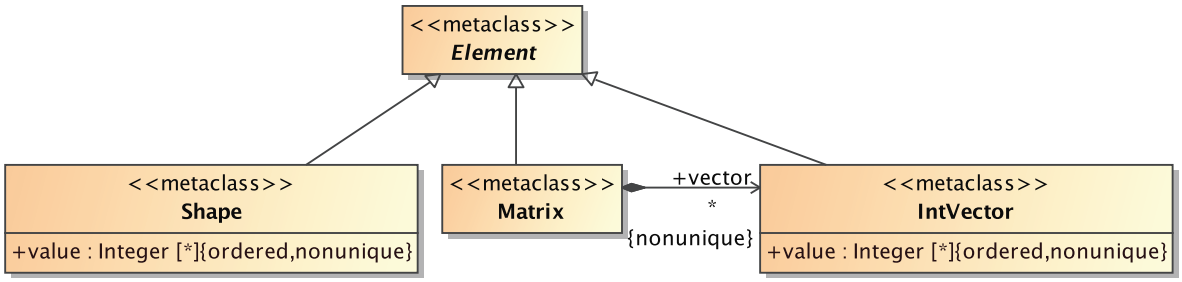


FIG. 4.20: Spécification des vecteurs, matrices et multiplicité dans la version originale (G2).

L'introduction de ces différents concepts rend le métamodèle Gaspard générique. Ce métamodèle, qui est actuellement exploité pour convertir un modèle paramétré en modèle dans lequel toutes les valeurs sont fixées, pourra être exploité ultérieurement comme une étape permettant d'emmener les paramètres plus bas dans les transformations.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

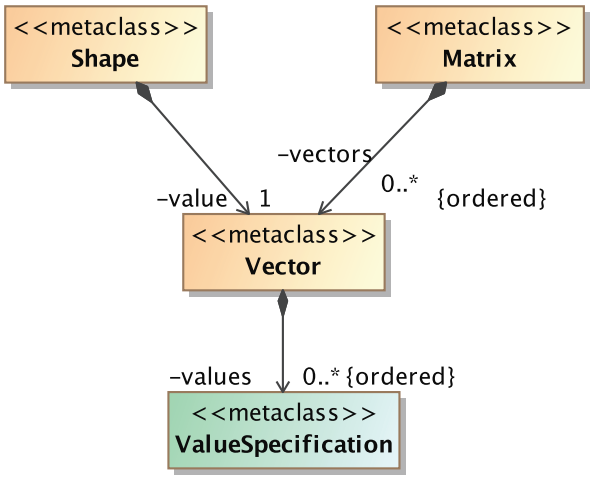


FIG. 4.21: Les versions différentes des matrices, des vecteurs et des multiplicités.

4.4.3.3 D'un modèle UML paramétré à un modèle Gaspard complètement spécifié

Comme défini dans la stratégie d'introduction des composants génériques dans l'environnement Gaspard, deux étapes sont nécessaires avant d'arriver à un modèle Gaspard complètement spécifié.

La première étape de la transformation consiste à passer d'un modèle UML avec template vers un modèle dans le métamodèle Gaspard Paramétré. Chaque COMPONENT UML va créer un COMPONENT Gaspard. Un composant paramétré sera transformé en GENERICCOMPONENT. De même, chaque TEMPLATEPARAMETER appartenant à la TEMPLATESIGNATURE est transformé en FORMALPARAMETER. Il est également nécessaire d'interpréter les différents éléments faisant référence aux paramètres afin de faire référence aux FORMALPARAMETERS créés à la transformation.

La seconde étape de la transformation consiste à instancier effectivement les composants paramétrés avec leurs valeurs de paramètres afin d'obtenir un modèle Gaspard complètement spécifié. Pour cela, il faut passer d'un modèle dans le métamodèle Gaspard paramétré vers un modèle dans le métamodèle Gaspard. Cette transformation consiste à identifier les FORMALPARAMETERS et ACTUALARGUMENTS afin de créer les composants idoines. Une correspondance directe entre éléments de base des métamodèlesest implicite (puisque le métamodèle paramétré n'est pas autre chose qu'une version étendue de la version « normale »). À chaque instance de composant utilisant des ACTUALARGUMENTS, il faut créer une copie du composant auquel il fait référence en remplaçant les références aux FORMALPARAMETERS par les valeurs provenant des ACTUALARGUMENTS. Une fois ce composant créé, il faut que l'instance de composant fasse référence à ce nouveau composant afin qu'elle pointe vers un composant ayant les bonnes valeurs de paramètres.

4.4.3.4 Conclusion de la généricité

Dans cette partie, nous avons présenté l'introduction de la généricité des composants dans l'environnement Gaspard. Celle-ci est basée sur la généricité d'UML (les *templates*) mais également sur un patron de conception permettant l'introduction de la généricité dans les métamodèles à base de composants. Ce patron, appliqué ici sur le métamodèle Gaspard, devrait

Thèse de Julien Taillard, Lille 1, 2009

être applicable sur n'importe quel métamodèle. Une application sur le métamodèle MARTE a déjà été réalisée [54]. L'introduction de la généricité dans le profil et le métamodèle Gaspard est un début d'introduction de la généricité dans l'environnement Gaspard. Elle permet principalement la réutilisation des modèles de haut niveau. La perspective de ces travaux d'introduction de la généricité dans l'environnement Gaspard est la gestion des paramètres tout le long du processus de compilation. La « descente » des paramètres dans le processus de compilation offre de nouvelles possibilités. Il est imaginable de générer du code paramétré pouvant s'adapter à différents problèmes. Il est également possible d'optimiser les applications en appliquant les fonctions de refactoring en agissant sur certains des paramètres et en réalisant des exécutions tests validant ou non les tailles de blocs testés.

4.5 Conclusion du chapitre

Dans ce chapitre, nous avons présenté l'extension de l'expressivité au plus haut niveau d'abstraction permettant de cibler et de faciliter l'usage de modèle.

Un point crucial pour permettre la modélisation d'algorithme de calcul scientifique a été abordé : la modélisation de structure itérative. La modélisation de telle structure n'était pas possible dans la spécification Gaspard de base.

La spécification à assignation unique d'Array-OL réduisait les possibilités d'utiliser des ports en entrée/sortie sur des composants. Or cette possibilité est un point important afin d'obtenir des codes performants puisque cela évite la déclaration de trop de variables et permet de ne pas faire un nombre excessif de copies de données en mémoire.

Enfin, le manque de réutilisation a été abordé. La généricité pose un vrai problème pour le calcul scientifique puisque les problèmes traités dans ce domaine sont tous différents les uns des autres. C'est pourquoi les composants génériques ont été introduits afin de pallier le manque de réutilisation des modèles.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Chapitre 5

Un métamodèle de langage procédural/impératif

5.1 Motivations et objectifs du métamodèle **86**
5.2 Grammaire YACC du C ANSI **86**
5.3 Structures et concepts du métamodèle **87**
5.3.1 Structure globale d'un modèle 87
5.3.2 Les fichiers 88
5.3.3 Les variables et les types de données 90
5.3.4 La définition des structures 91
5.3.5 Modélisation des expressions 94
5.3.6 Les procédures et les fonctions dans le métamodèle 95
5.3.7 Conclusion sur la définition du métamodèle 96
5.4 Extension du métamodèle pour les langages parallèles **96**
5.4.1 Ajout de structure pour le support d'OpenMP 96
5.4.2 Modélisation de la bibliothèque MPI 98
5.5 Utilisation de ce métamodèle dans la chaîne de transformation **99**
5.5.1 Appel de procédures 99
5.5.2 Génération du programme principal 99
5.5.3 Passage de paramètres par adresse 99
5.6 Génération de code à partir d'un modèle respectant ce métamodèle **99**
5.6.1 Transformation *modèle vers texte* 101
5.6.2 Implémentation de la génération de code 101
5.7 Conclusion du chapitre **102**

Afin de réaliser la génération de code OpenMP, il convient de transformer le graphe de tâches du modèle Gaspard vers un modèle textuel de celui-ci contenant les informations de parallélisme du code. Ce passage d'un graphe de tâches vers un modèle textuel n'est pas trivial et demande un certain temps de développement. Souhaitant pouvoir produire aussi bien du code Fortran que du code C, il est intéressant de passer par un modèle intermédiaire avant de générer le code lui-même. Cela permet de factoriser la transformation du graphe de tâches en un modèle textuel. C'est pourquoi un métamodèleintermédiaire avant la réalisation de la génération de code est introduit.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

Ce chapitre présente le métamodèle de langages procéduraux introduit afin de factoriser les transformations du métamodèle loop vers les langages C et Fortran.

La première partie de ce chapitre présente les motivations et objectifs de ce métamodèle. La seconde partie de ce chapitre est consacrée à la présentation de la grammaire YACC du langage C ANSI qui a inspiré en partie ce métamodèle. La troisième partie de ce chapitre est consacrée à la définition des structures de bases du métamodèle permettant de modéliser un programme écrit dans un langage procédural. La quatrième partie de ce chapitre présente l'extension du métamodèle afin de permettre la modélisation de programme parallèle. L'extension consiste en l'introduction des structures OpenMP dans le métamodèle et la définition de la librairie MPI. La cinquième partie présente la méthodologie d'utilisation du métamodèledans le cadre du développement de Gaspard. La sixième partie explique comment du code peut être produit à partir d'un modèle dans le métamodèle.

5.1 Motivations et objectifs du métamodèle

La transformation de modèle vers du code parallèle OpenMP suppose de cibler soit le langage C, soit le langage Fortran qui sont les deux langages gérés par la norme OpenMP. La première solution qui vient à l'esprit est de générer directement l'un des deux langages (solution par ailleurs retenue pour la chaîne de génération de code SystemC). Cette solution a l'inconvénient de provoquer l'écriture d'une autre transformation si l'on veut cibler l'autre langage. Ainsi, deux transformations quasi semblables (à la syntaxe près et quelques optimisations possibles dans l'un ou l'autre des langages) doivent être écrites à partir du métamodèle Loop. Cette transformation est complexe puisqu'il faut transformer le graphe de tâches vers un modèle textuel tout en ajoutant les informations de parallélisme (directives OpenMP et barrières de synchronisation) et en déclarant les variables nécessaires au programme et leurs comportements (publiques ou privées). Écrire plusieurs fois cette transformation serait compliquée et source d'erreurs. De plus, développer plusieurs fois la même transformation (ou tout du moins des transformations très proches) n'est pas conforme à l'esprit de l'IDM qui promeut la réutilisation de code et la productivité. Pour toutes ces raisons, cette option de génération directe de code à partir du métamodèle Loop ne semble pas la bonne solution. C'est pourquoi nous n'avons pas choisi de générer directement le code à partir du métamodèle Loop, mais de passer par un métamodèle intermédiaire avant de cibler l'un ou l'autre des langages.

Le C et le Fortran étant des langages de mêmes classes (langages procéduraux ou impératifs), nous avons choisi de définir un métamodèle intermédiaire générique de ces langages permettant de produire ensuite aussi bien le code C que le code Fortran. L'introduction de ce métamodèle permet d'utiliser la même transformation entre le métamodèle Loop et le métamodèle de langage procédural. C'est lors de la génération de code à partir de ce métamodèle que se fera le choix du langage cible. Ce métamodèle n'est pas propre à nos besoins, il a été développé afin d'être générique, c'est à dire de pouvoir être réutilisé dans toute approche modèle nécessitant un tel métamodèle.

5.2 Grammaire YACC du C ANSI

La structure d'un langage de programmation est basée sur une grammaire. Cette grammaire définit la syntaxe à utiliser lors de l'écriture d'un programme dans ledit langage. En

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

comparant le monde des langages et le monde des modèles, il est évident de faire l'analogie entre une grammaire et un métamodèle : les deux concepts donnent une syntaxe et une sémantique aux éléments les utilisant. Ainsi, un programme respectant une grammaire peut être vu comme un modèle conforme à un métamodèle. Il est donc intéressant d'étudier les grammaires de langages procéduraux afin d'en déduire un métamodèle de ces langages.

YACC (*Yet Another Compiler Compiler*) [74] est un analyseur syntaxique permettant d'analyser la syntaxe d'un programme à partir d'une définition de sa grammaire afin de la vérifier notamment. Le métamodèle de langage impératif est notamment basé sur la grammaire YACC du C ANSI [76] qui définit la structure d'un programme C. Afin de permettre une meilleure compréhension du métamodèle de langage procédural, la structure de cette grammaire va être ici illustrée afin d'en extraire une base pour la métamodélisation de langages procéduraux.

L'extrait de la grammaire du C présenté à la figure 5.1 illustre la construction des structures du langage C. Une structure (*statement*) peut ainsi être six structures différentes : une structure ayant un label (*labeled_statement*) ; une structure composée (*compound_statement*) ; une expression (*expression_statement*) ; une sélection (*selection_statement*) ; une itération (*iteration_statement*) ou un saut (*jump_statement*).

La déclaration de chaque type de structure illustre la manière dont celle-ci est construite. Ainsi, une structure de sélection (*selection_statement*) peut être une structure de *Si .. alors ..*, de *Si .. alors .. sinon* ou un switch. Une structure *Si .. alors ..* est constituée par un token (ou jeton en français) *IF* suivit d'une *expression* entre parenthèses constituant la condition, et une structure constituant le corps du alors.

Cette grammaire illustre toutes les possibilités offertes par le langage C. Toutes ces structures ne sont pas disponibles dans tous les langages, c'est pourquoi il faudra, afin de construire le métamodèle, extraire les structures communes à tous les langages afin de les modéliser. Les structures propres à un langage ne doivent pas être introduites dans un métamodèle général de langage. Ces constructions spécifiques peuvent cependant être introduites comme une extension du métamodèle (c'est ce qui sera fait pour OpenMP).

5.3 Structures et concepts du métamodèle

Ce métamodèle est introduit afin de permettre la modélisation d'un programme dans les langages Fortran et C puisque ce sont les langages supportés par la norme OpenMP. Ce métamodèle a donc été inspiré par la grammaire de ces deux langages et notamment la grammaire YACC du C ANSI illustrée précédemment. Les constructions de bases communes aux langages impératifs sont extraites de ces grammaires afin de les modéliser. Le métamodèle doit notamment permettre la modélisation de concepts de bases des langages tels que les boucles, les structures conditionnelles, les expressions, l'appel de fonctions et de procédures. Cette modélisation doit être faite de manière générique afin de pouvoir être transposée dans les différents langages procéduraux.

5.3.1 Structure globale d'un modèle

La base d'une application est le programme principal accompagné des bibliothèques nécessaires lui permettant de remplir ses fonctionnalités. Cela va constituer la base d'un modèle dans le métamodèle.

La base d'un modèle, dans le métamodèle, est l'espace de travail (le WORKSPACE). Ce concept représente la racine du modèle, c'est-à-dire le concept qui va contenir tous les

```
statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        ;

statement_list
        : statement
        | statement_list statement
        ;

expression_statement
        : ';'
        | expr ';'
        ;

selection_statement
        : IF '(' expr ')' statement
        | IF '(' expr ')' statement ELSE statement
        | SWITCH '(' expr ')' statement
        ;

iteration_statement
        : WHILE '(' expr ')' statement
        | DO statement WHILE '(' expr ')' ';'
        | FOR '(' ';' ';' ')' statement
        | FOR '(' ';' ';' expr ')' statement
        | FOR '(' ';' expr ';' ')' statement
        | FOR '(' ';' expr ';' expr ')' statement
        | FOR '(' expr ';' ';' ')' statement
        | FOR '(' expr ';' ';' expr ')' statement
        | FOR '(' expr ';' expr ';' ')' statement
        | FOR '(' expr ';' expr ';' expr ')' statement
        ;
```

FIG. 5.1: illustration des structures dans la grammaire YACC du C ANSI.

autres concepts du modèle. Ainsi, le WORKSPACE va rassembler les concepts de base d'une application, c'est pourquoi il contient le programme principal (PROGRAM) définissant la fonction principale (le *main*), ainsi que les bibliothèques (LIBRARY et LIBRARYINTERFACE) utilisées et si nécessaire définies avec le programme (figure 5.2).

5.3.2 Les fichiers

Le programme ainsi que les bibliothèques sont des fichiers (FILE). Le programme et les bibliothèques ont en commun de pouvoir contenir des variables (VARIABLE) et des routines (ROUTINE qui peuvent être des fonctions FUNCTIONS ou des procédures PROCEDURES). Un

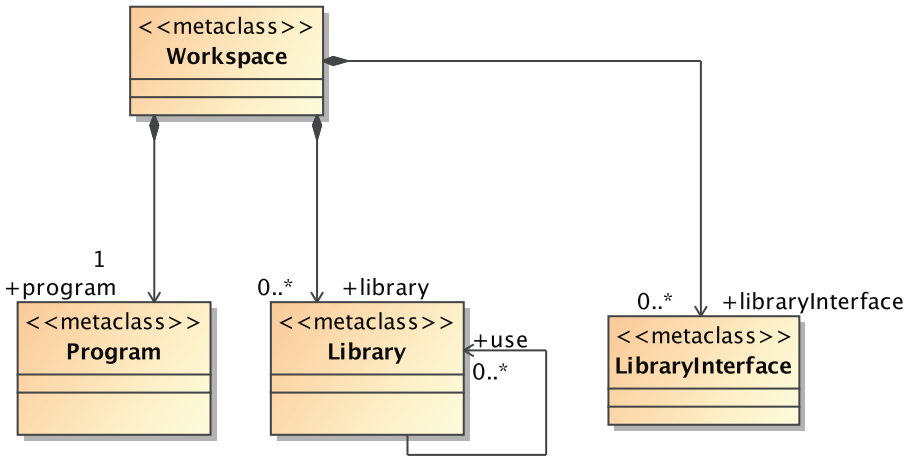


FIG. 5.2: Illustration d'un WORKSPACE : la racine d'un modèle dans le métamodèle de langage procédural.

programme contient en plus un *main* qui est en fait une suite de structures définissant le programme principal (STATEMENT_LIST). La figure 5.3 illustre ces notions dans le métamodèle.

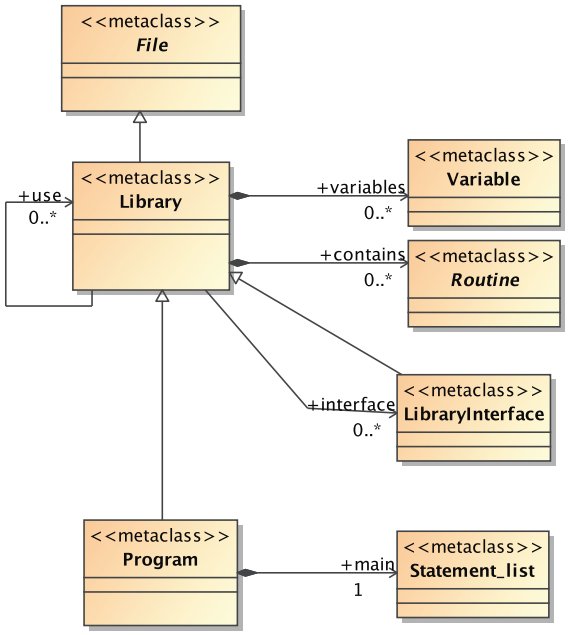


FIG. 5.3: Illustration de la structuration des fichiers.

La notion de LIBRARYINTERFACE permet d'exprimer l'interface d'une bibliothèque de fonctions. Ainsi, une LIBRARYINTERFACE ne doit contenir que des routines ayant une signature et pas de corps : c'est la déclaration des routines disponibles dans la bibliothèque. Cela peut être vu comme le fichier .h d'une bibliothèque en langage C.

Le concept de LIBRARY représente quant à lui la bibliothèque de fonctions. Une LIBRARY va donc contenir l'implémentation des différentes fonctions qu'elle déclare. Pour rester dans la comparaison avec le langage C, c'est le fichier .c d'une bibliothèque C implémentant l'interface définie dans le fichier .h illustré précédemment.

5.3.3 Les variables et les types de données

Les variables et les types de données sont très liés puisque ce sont principalement ces premières qui utilisent ces seconds.

5.3.3.1 Les types de données

Un programme peut utiliser des types de données quelconques, c'est pourquoi il est important de pouvoir déclarer des types de données dans le modèle. Tous les métamodèles de l'environnement Gaspard utilisent la même base pour modéliser les types de données. La structuration des types de données dans le métamodèle est présentée dans la figure 5.4.

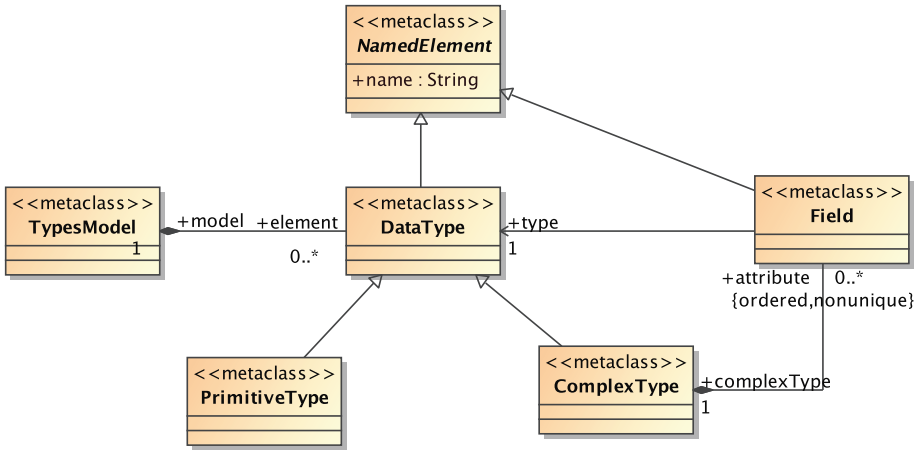


FIG. 5.4: Les types dans le métamodèle.

Le concept de base des types de données est le concept abstrait de DATATYPE qui regroupe les différents types de données que l'on peut avoir. Un DATATYPE généralise le concept de NAMEDELEMENT permettant à l'élément d'avoir un nom. Les types élémentaires des langages dans le métamodèle sont représentés par le concept de PRIMITIVETYPE. Il permet la représentation des entiers, des booléens, des doubles, etc. Dans les applications de calculs hautes performances, ce sont généralement les types de données primitifs qui seront utilisés. Il est cependant également possible de définir des structures de données (COMPLEXTYPE). Une structure de données est composée de champs nommés (FIELD) qui ont un type. Ce mécanisme permet la modélisation de tout type de structure de données puisqu'il n'y a aucune contrainte sur le nombre de champs d'une structure ou le type de données de ceux-ci. Le concept de type énuméré est également présent bien que non illustré ici.

5.3.3.2 Les variables

Dans le métamodèle de langage procédural, toutes les variables doivent être déclarées pour être utilisées, il n'y a pas de déclaration implicite, tout doit être explicite. Les variables peuvent être déclarées dans un programme, une librairie ou une routine. La portée des variables est semblable à celle des langages : une variable déclarée dans une routine ne peut être utilisée que dans cette routine. La figure 5.5 présente les variables dans le métamodèle. Une variable (VARIABLE) a un type et peut avoir une dimension (ARRAY) afin de représenter un tableau de données.

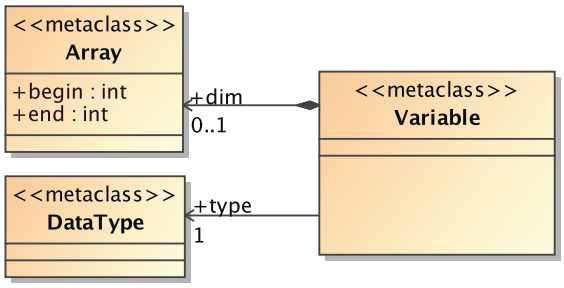


FIG. 5.5: Les variables dans le métamodèle.

5.3.4 La définition des structures

Les structures des langages de programmation sont illustrées par le concept de STATEMENT dans le métamodèle. Ce concept permet ainsi de définir le corps du programme et des différentes routines déclarées dans le modèle. Un programme est défini par une suite ordonnée de structures. La structure STATEMENT_LIST permet de construire une telle suite de structure. La fonction principale d'un programme est ainsi constitué par un STATEMENT_LIST.

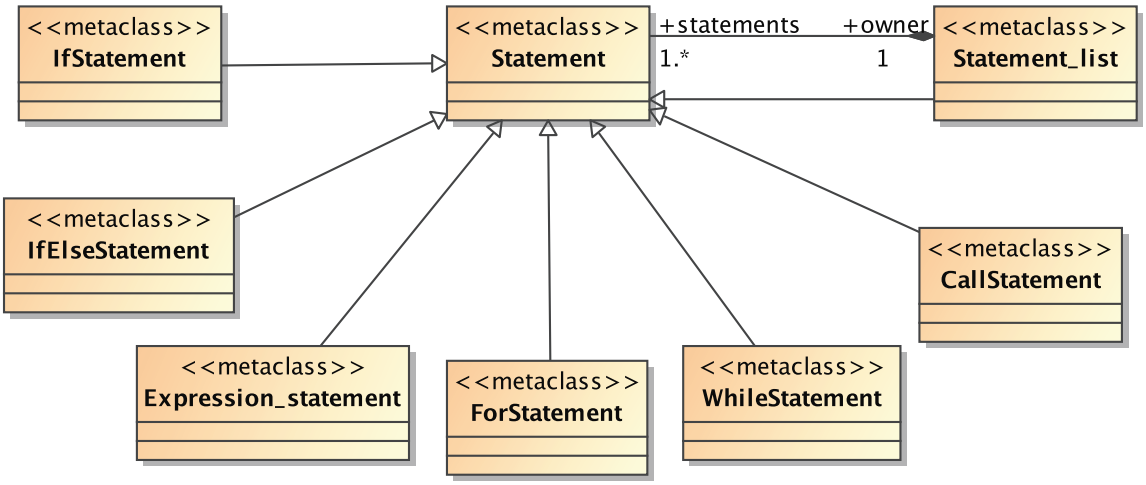


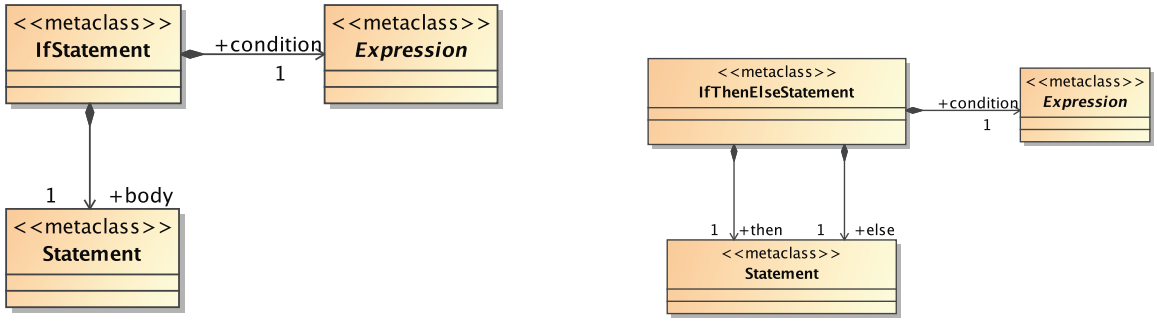
FIG. 5.6: Illustration des différentes constructions disponibles.

Thèse de Julien Taillard, Lille 1, 2009

Les différentes constructions classiques des langages sont disponibles, elles sont illustrées dans la figure 5.6. Toutes les structures généralisent le concept STATEMENT. On peut remarquer que les structures présentées ici sont équivalentes à celles illustrées lors de la présentation de la grammaire YACC du C ANSI. Ces structures sont détaillées dans les sous-sections suivantes.

5.3.4.1 Les structures conditionnelles

Les structures conditionnelles permettent de construire les structures dont l'exécution dépend de la valeur d'un prédicat. La construction de ces structures de sélection reprend la structure classique de la grammaire C qui est semblable à la grammaire Fortran. Les structures conditionnelles sont illustrées à la figure 5.7. Un IFSTATEMENT contient une condition qui est une EXPRESSION (les expressions sont présentées dans la section 5.3.5). Si cette condition est vraie alors le programme exécutera le corps du if qui est une structure (STATEMENT). Un IFELSESTATEMENT est également composé par une condition. Si cette condition est vérifiée alors le programme exécutera la partie du programme correspondant au lien de composition *then* (qui est un STATEMENT) sinon c'est la structure correspondant au lien de composition *else* qui sera exécutée.



(a) construction d'un if

(b) construction if then else

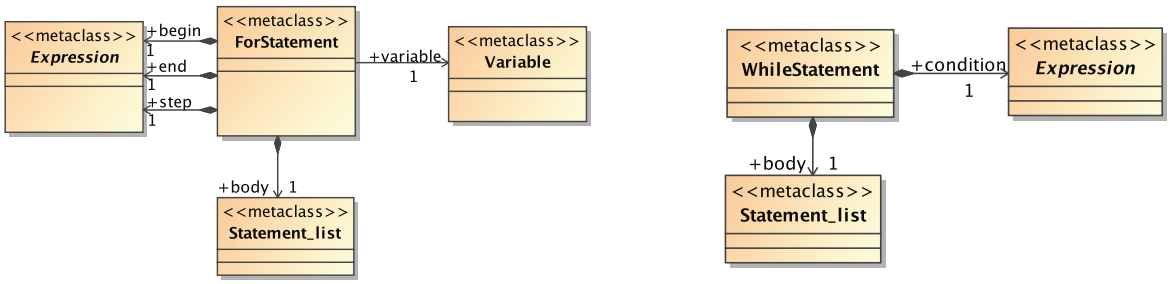
FIG. 5.7: Structures des structures conditionnelles.

5.3.4.2 Les structures itératives

Les structures itératives permettent de répéter un nombre de fois défini ou non la structure d'un programme : ces structures sont généralement appelées *boucles*. Elles sont illustrées dans la figure 5.8.

Deux types de boucles sont communs aux différents langages. Il y a les boucles *Pour* (*do* ou *for* en anglais) représentées par le concept de FORSTATEMENT. Ces boucles itèrent d'un indice d'itération de début à un indice de fin avançant d'un pas *step* à chaque itération. Dans le métamodèle, un FORSTATEMENT est constitué de trois expressions représentées par les liens de compositions (*begin*, *end* et *step*). Ces expressions expriment respectivement l'indice de début d'itération, de fin d'itération ainsi que le pas d'itération. Le corps de la boucle est une suite de structure STATEMENT_LIST. Les boucles *Tant que* (*while*), qui ne s'arrêtent que lorsque la condition d'arrêt devient fausse, constituent le second type de boucle. Ce type de boucle est régulièrement utilisé pour les boucles de convergence. Dans le métamodèle,

© 2010 Tous droits réservés. http://doc.univ-lille1.fr



(a) Une boucle Pour

(b) Une boucle Tant que

FIG. 5.8: Les boucles dans le métamodèle de langage procédural.

les boucles *tant que* sont représentées par le concept de WHILESTATEMENT qui est composé d'une expression exprimant la condition d'arrêt et d'une suite de structure exprimant le corps de la boucle.

5.3.4.3 L'appel de procédures

Bien que ne constituant pas une structure dans la grammaire du C, les appels de procédures sont introduits sous la forme de structure dans le cadre du métamodèle. En effet, un appel ne renvoyant pas de valeurs, il ne peut pas être introduit dans les expressions. C'est pourquoi, le concept de CALLSTATEMENT est implanté dans le métamodèle. Celui autorise l'appel de procédures dans un modèle. Ce concept est présenté dans la figure 5.9.

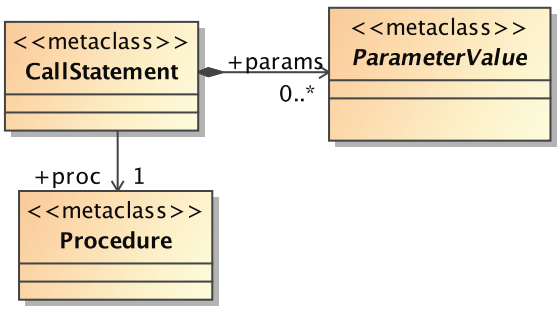


FIG. 5.9: Structure permettant l'appel d'une procédure.

Un CALLSTATEMENT contient une référence vers la procédure appelée, via le lien d'association *proc*, et contient une liste ordonnée de PARAMETERVALUE représentant les paramètres à passer à la procédure pour son exécution. Un PARAMETERVALUE peut être soit une référence vers une variable, soit une expression.

5.3.4.4 Les expressions

Afin de réaliser les différents calculs nécessaires au programme, il est important de pouvoir placer des expressions dans la structure du programme. C'est pourquoi le concept d'EXPRESSION_STATEMENT a été introduit.

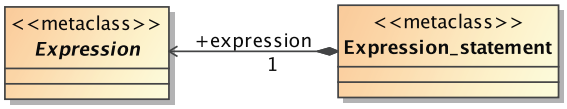


FIG. 5.10: Structure permettant l'introduction d'expression.

Ce concept contient une expression, via son lien de composition *expression* et permet ainsi de placer une expression dans le programme. La modélisation des expressions est présentée dans le paragraphe suivant (§ 5.3.5).

5.3.5 Modélisation des expressions

Les expressions sont les constructions de base permettant d'exprimer les calculs à effectuer, mais également les conditions des structures conditionnelles.

Une expression permet d'appliquer un opérateur entre des expressions (deux ou plus selon les opérateurs). Le concept EXPRESSION est la métaclasse de base permettant la modélisation et l'utilisation des expressions dans un modèle conforme au métamodèle. La combinaison des imbrications entre les structures (STATEMENT) et les expressions (EXPRESSION) va exprimer l'algorithme du programme ou d'une fonction. La structuration des concepts disponibles pour modéliser des expressions est présentée dans la figure 5.11.

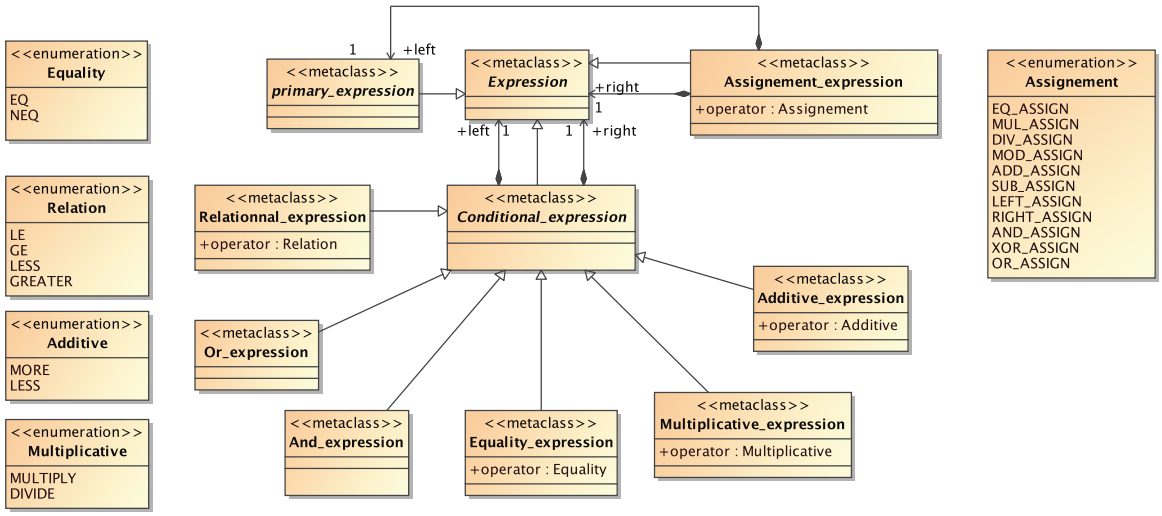


FIG. 5.11: Construction des expressions dans le métamodèle.

La construction d'une expression dans le métamodèle va représenter un arbre syntaxique abstrait. Ce type de construction est couramment utilisé dans les processus de compilations afin, notamment, de réaliser des optimisations. Dans ce métamodèle, l'arbre syntaxique abstrait est un arbre dans lequel les noeuds sont des opérateurs et les feuilles des expressions primaires (concept PRIMARY_EXPRESSION). Le concept d'expression primaire regroupe l'utilisation de variables dans une expression, de constantes, de chaînes de caractères. La construction de l'arbre syntaxique se fait par les expressions de type CONDITIONAL_EXPRESSION. Ce

concept regroupe les expressions ayant en membre gauche et en membre droit des expressions quelconques et qui appliquent entre ces membres un opérateur. La place des expressions (à gauche ou à droite) est importante dans le cas d'opérateur non commutatif tel que la division, la division de a par b ne donnant pas le même résultat que la division de b par a. Les CONDITIONAL_EXPRESSIONS peuvent être des expressions relationnelles (concept RELATIONNAL_EXPRESSION), des expressions exprimant des *et* logiques et *ou* logiques (concepts AND_EXPRESSION et OR_EXPRESSION), des expressions portant sur l'égalité entre deux expressions (concept EQUALITY_EXPRESSION) et des expressions multiplicatives ou additives (concepts MULTIPLICATIVE_EXPRESSION et ADDITIVE_EXPRESSION)

Les affectations sont représentées à l'aide du concept ASSIGNATION_EXPRESSION. Le membre gauche d'une affectation doit être la référence à une variable ou élément d'une variable (case d'un tableau), c'est pourquoi le membre gauche sera toujours une expression primaire (PRIMARY_EXPRESSION). Le membre droit d'une affectation peut être le résultat d'une expression quelconque.

L'appel à une fonction est également une expression (concept FUNCTIONCALL non illustré ici) puisque la valeur de retour peut être utilisée dans une affectation, mais aussi dans tout type d'expression.

5.3.6 Les procédures et les fonctions dans le métamodèle

Les procédures et fonctions regroupées dans le concept de ROUTINE peuvent être déclarées et contenues dans les librairies et le programme (figure 5.12). Une routine possède une signature (concept SIGNATURE) qui est la déclaration des paramètres et du type de la valeur de retour. Une routine peut être une procédure (concept PROCEDURE) qui est une routine qui ne renvoie rien et une fonction (concept FUNCTION) qui renvoie une valeur. Une routine peut contenir des variables locales (VARIABLE) et contient un corps de type STATEMENT afin de permettre la définition complète des fonctionnalités de la routine.

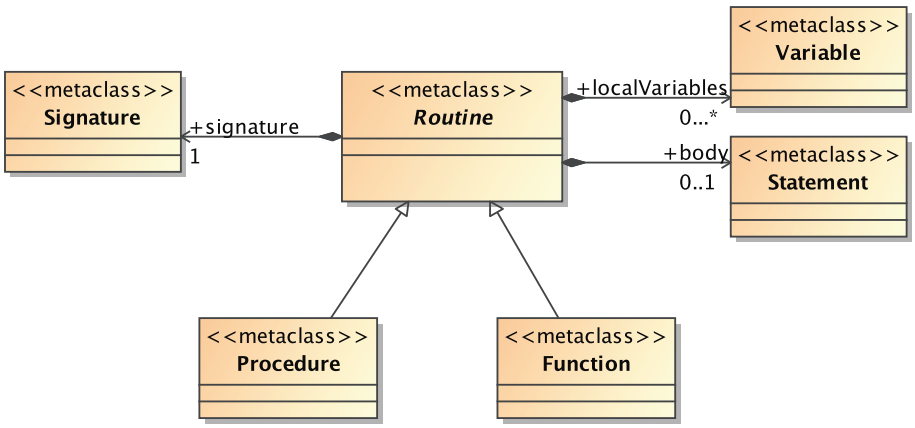


FIG. 5.12: Structuration des routines dans le métamodèle.

Thèse de Julien Taillard, Lille 1, 2009

5.3.7 Conclusion sur la définition du métamodèle

Cette partie a présenté la définition du métamodèle de langages procéduraux. Ce métamodèle a été défini d'une manière générique et aucunement restreinte à l'utilisation que nous allons en faire dans notre de flot de conception Gaspard. Il est ainsi possible de réutiliser ce métamodèle dans toute approche orientée modèle ayant besoin d'un tel métamodèle. Bien que non testé sur tous les langages procéduraux existants (tels que la partie procédurale de Ada ou Pascal), ce métamodèle devrait pouvoir être utilisé pour cibler ces différents langages, cela ne devrait pas provoquer de grands bouleversements. Tout au plus, cela nécessiterait de petits ajustements pour tenir compte d'une spécificité oubliée d'un langage. Ce métamodèle peut être étendu afin de gérer une spécificité particulière d'un langage, mais également afin d'ajouter des constructions aux langages procéduraux afin de gérer d'autres langages tels que OpenMPque nous allons étudier dans le paragraphe suivant.

5.4 Extension du métamodèle pour les langages parallèles

Le métamodèle, décrit dans la section précédente, permet de modéliser des programmes séquentiels dans des langages procéduraux/impératifs. Afin de faire de la génération de code parallèle, le métamodèle doit être étendu afin de permettre la génération de code OpenMP. Dans le but d'illustrer la généricité de l'approche, l'introduction de MPI est également discutée.

5.4.1 Ajout de structure pour le support d'OpenMP

La norme OpenMP [113] définit des directives de compilation à introduire dans le code afin d'indiquer les parties parallèles du code. Ces directives sont introduites sous la forme de commentaires en Fortran et sous forme de *pragma* en C. La norme OpenMP définit également une librairie permettant principalement d'accéder aux informations sur les processus légers. Afin de permettre la modélisation de programme OpenMP, il est nécessaire de gérer ces deux aspects.

5.4.1.1 Ajout des directives de compilation dans le métamodèle

Les directives de compilation imposent d'ajouter des concepts dans le métamodèle permettant de les exprimer. Les directives n'étant pas ajoutées dans les mêmes concepts (commentaires en Fortran et pragmas en C) selon le langage utilisé, il n'est pas possible de simplement ajouter le concept dans lequel sont mises ces directives. Les directives de compilations peuvent être considérées comme des structures quelconques dans le programme, c'est pourquoi elles vont généraliser le concept STATEMENT. La figure 5.13 illustre l'introduction d'une partie des directives de compilations dans le métamodèle. C'est lors de la génération de code que sera gérée dans quels types de structures sont mises les directives.

Les directives de compilations OpenMP sont regroupées dans le concept de OMP_STATEMENT. Une directive OpenMP peut avoir des clauses qui peuvent notamment être PRIVATE, SHARED,DEFAULT. Les directives OpenMP contiennent les structures du métamodèle : les STATEMENTS. On retrouve notamment la structure classique OMP_PARALLEL, ainsi que OMP_SECTIONS qui contient des OMP_SECTION qui eux-mêmes contiennent des STATEMENTS. Certaines directives de compilations ne peuvent contenir qu'un type de

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

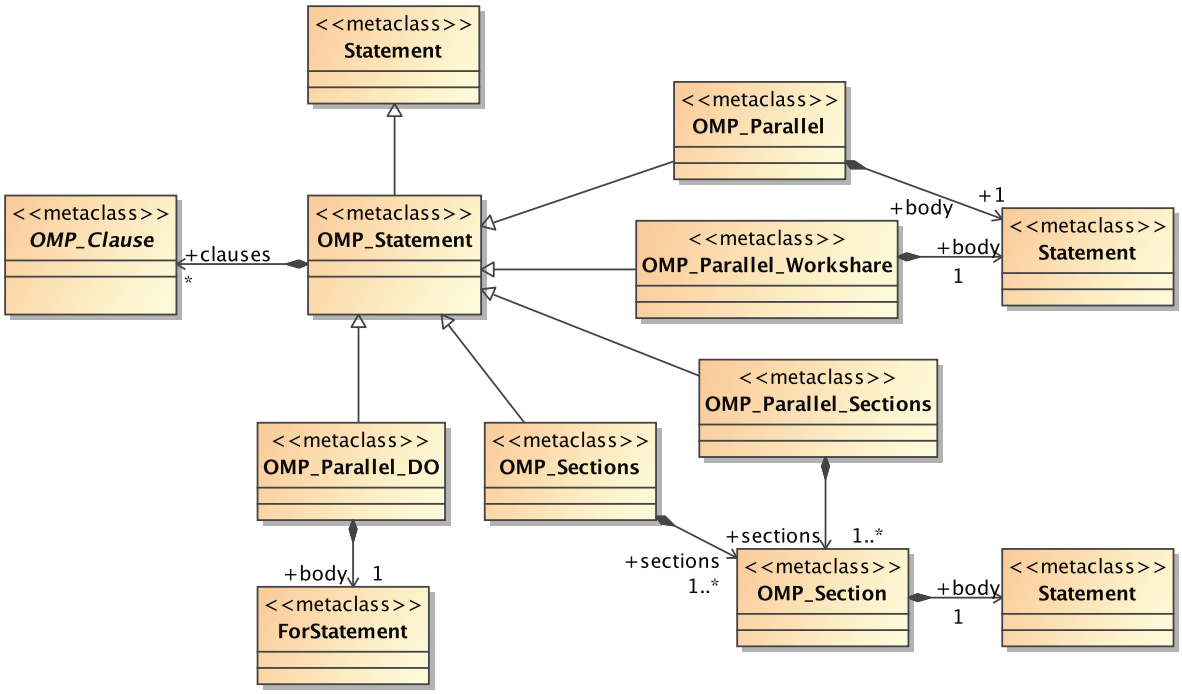


FIG. 5.13: Introduction des directives de compilations dans le métamodèle.

structure. C'est pourquoi un OMP_PARALLEL_DO qui signifie qu'une boucle est parallèle ne peut contenir qu'une structure de boucle *Pour* (concept FORSTATEMENT).

5.4.1.2 Modélisation de la bibliothèque OpenMP

Afin d'utiliser toute la norme OpenMP, il est également nécessaire de pouvoir utiliser les fonctions de la bibliothèque OpenMP. Cette bibliothèque contient des fonctions et des procédures permettant d'agir sur le nombre de threads, de connaître ce nombre ou encore de récupérer le numéro du thread.

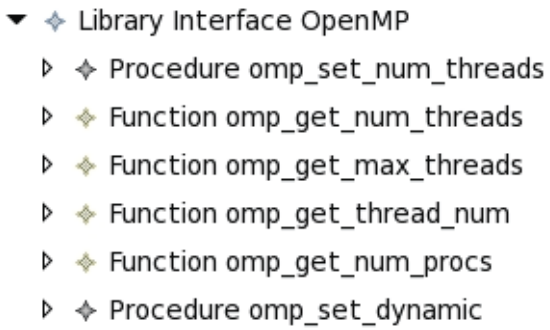


FIG. 5.14: Extrait de la bibliothèque OpenMP modélisée.

La bibliothèque contenant ces routines OpenMP a donc été modélisée (figure 5.14) afin de permettre la modélisation complète d'un programme OpenMP dans le métamodèle de

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

langage procédural étendu. Le modèle est représenté sous forme d'arbre puisque, par défaut, il n'existe pas de représentation graphique associée à un métamodèle. On utilise donc ici l'éditeur arborescent associé à EMF/Ecore.

5.4.2 Modélisation de la bibliothèque MPI

MPI (Message Passing Interface) [43] est une bibliothèque introduisant des fonctions de communications entre processus, elle est notamment utilisée pour programmer les machines à mémoires distribuées. Permettre l'utilisation de MPI dans un modèle conforme au métamodèle de langage procédural devrait revenir à modéliser cette bibliothèque. Malheureusement, la norme MPI définit des interfaces différentes selon le langage que l'on veut utiliser. Ainsi pour le langage C, l'ensemble des routines sont des fonctions renvoyant en retour le code d'erreur. En Fortran, les routines sont des procédures dont le dernier paramètre est le code d'erreur (celui renvoyé en retour de la fonction C). En considérant que la génération de code n'est qu'une écriture du modèle en texte (c'est à dire sans calcul durant la génération), il n'est donc, à priori, pas possible d'utiliser un seul modèle de bibliothèque quelque soit le langage ciblé ensuite. Si un seul langage est ciblé, il est possible de modéliser la bibliothèque correspondant au langage et de l'utiliser. Un extrait des bibliothèques C et Fortran pour MPI est présenté dans la figure 5.15.

(a) Bibliothèque Fortran (b) Bibliothèque C

FIG. 5.15: Modélisation d'une petite partie de la bibliothèque MPI.

Une solution permettant l'utilisation de la même interface de bibliothèque pour le MPI afin de permettre de cibler aussi bien le Fortran que le C à partir d'un même modèle et en utilisant la même transformation serait de faire un adaptateur entre l'interface C et l'interface Fortran afin d'utiliser toujours l'une des deux. Une autre possibilité serait également de charger la bonne bibliothèque dans la chaîne de transformation selon le langage cible et la transformation s'adapterait selon la bibliothèque chargée. En considérant que la génération de code peut effectuer des calculs, il serait alors possible de modifier le métamodèle afin de signaler lors de la création de la bibliothèque où se met le code de retour dans chacun des langages (dans les paramètres ou en retour de fonction).

Le métamodèle de langage procédural autorise l'utilisation de la librairie MPI dans les modèles afin de générer du code utilisant cette bibliothèque. Cela n'est actuellement pas

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

utilisé dans l'environnement Gaspard.

5.5 Utilisation de ce métamodèle dans la chaîne de transformation

Le métamodèle définit précédemment est généraliste. Dans le cas de son utilisation dans l'environnement Gaspard, nous allons définir un cadre d'utilisation contraignant que les transformations générant un modèle dans le métamodèle devront respecter. Ce cadre d'utilisation doit permettre la génération de code OpenMP(en C et en Fortran) sans modification de la transformation générant le modèle à ce niveau (la transformation du métamodèle loop vers le métamodèle de langage).

5.5.1 Appel de procédures

Les composants élémentaires sont liés dès le modèle de haut niveau vers une implémentation dans le langage ciblé. Les implémentations doivent avoir autant d'entrées et de sorties que les composants élémentaires. Aussi, afin de faciliter le travail de génération de code, il a été décidé que les composants élémentaires ne devaient être liés uniquement qu'à des procédures (qui ne renverront donc rien en sorties). Dans le cas de l'utilisation de fonctions, il aurait fallu définir un mécanisme de déploiement spécifiant à quel port correspond la valeur de sortie. Cela va permettre de n'avoir à générer que des appels à des procédures qui effectueront les calculs nécessaires au programme.

5.5.2 Génération du programme principal

Comme rappelé précédemment, les composants élémentaires des modèles sont associés à une procédure dans le langage cible, il n'est donc pas nécessaire de générer ces briques élémentaires. La génération du modèle dans le métamodèle de langage doit donc se concentrer sur la génération du liant entre ces briques de base afin de produire du code parallèle efficace. C'est pourquoi seul le programme principal sera généré par la transformation.

5.5.3 Passage de paramètres par adresse

La transformation doit produire un modèle compatible aussi bien avec le C qu'avec du Fortran. Une des grandes différences entre ces deux langages est le mécanisme de passage de paramètres aux routines. Le Fortran passe les paramètres par adresse alors que le C le fait par valeur. Afin de permettre l'utilisation de procédures Fortran dans le code C ou l'inverse, il a été décidé que tous les paramètres seraient passés par adresse. Il faudra alors évidemment utiliser des procédures C prenant des pointeurs en paramètres. Pour la génération de Fortran, cela ne changera rien, mais pour la génération de code C, il faudra passer l'adresse des différentes variables ainsi, à la génération de code, le symbole & sera placé devant chacun des paramètres d'une fonction.

5.6 Génération de code à partir d'un modèle respectant ce métamodèle

La chaîne de transformation ciblant le langage OpenMP passe par différents niveaux intermédiaires avant d'arriver dans le métamodèle de langages procéduraux. Ce métamo-

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

dèle est le dernier utilisé avant la génération de code. L'étape à laquelle nous sommes dans la chaîne de transformation de modèle est entourée en rouge dans la figure 5.16 représentant la méthodologie générale.

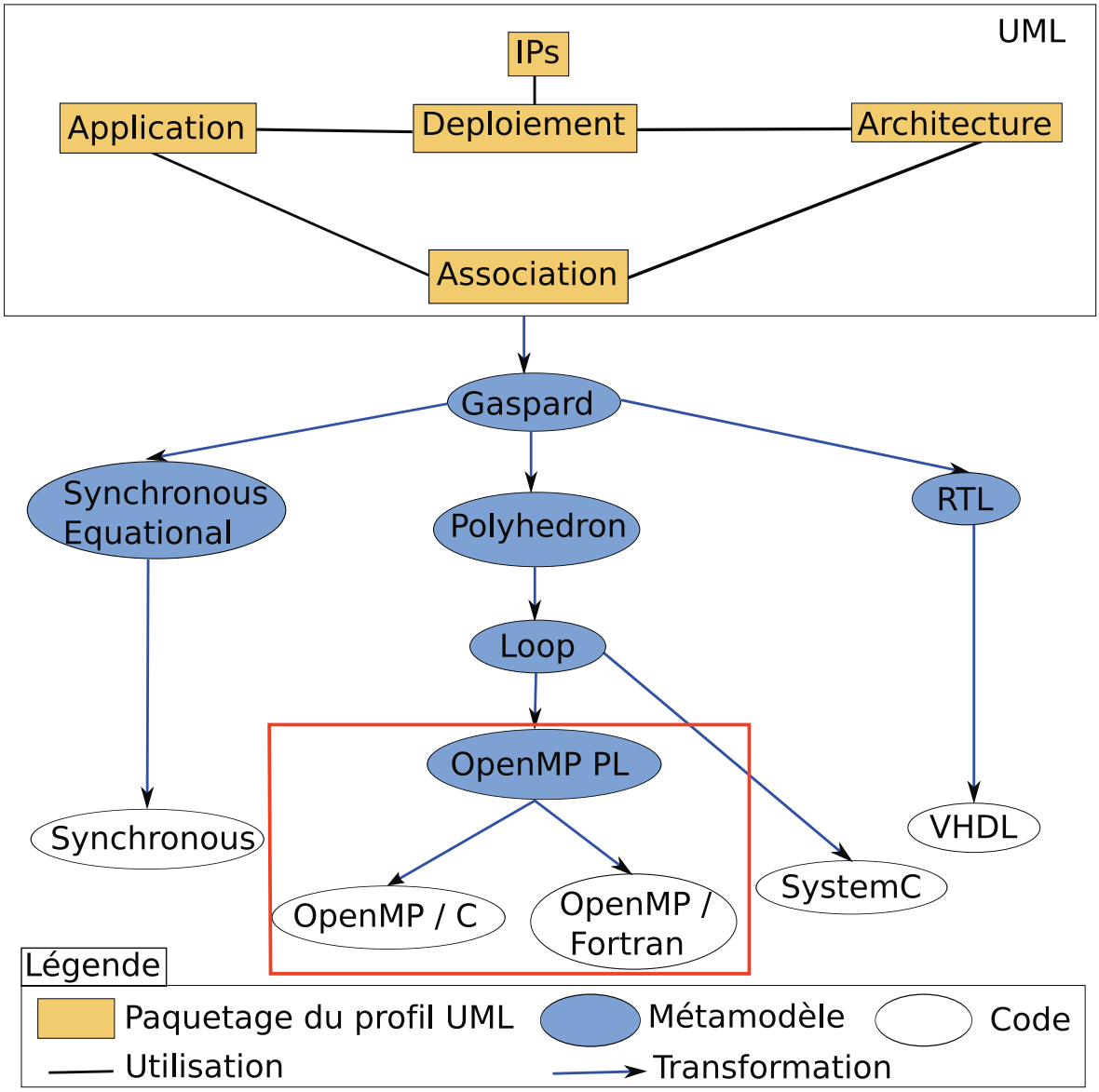


FIG. 5.16: La génération de code dans la méthodologie Gaspard.

La génération de code n'est pas vraiment une transformation de modèle vers modèle, c'est ce que l'on appelle couramment une transformation de *modèle vers texte*. Dans cette section, nous allons tout d'abord présenter le moteur de transformation utilisé pour cette génération de code puis nous verrons comment cette génération est implémentée.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

5.6.1 Transformation *modèle vers texte*

Selon les principes de l'IDM, la génération de code est vue comme l'écriture du modèle sous la forme de texte. Ainsi, les concepts ne doivent pas être transformés, mais traduits en texte. La génération de code peut donc être vue comme une transformation «un vers un» dans laquelle un concept génère une certaine structure de texte. C'est exactement dans ce cas que nous nous trouvons puisque la génération de code consiste à générer pour chaque concept la syntaxe correspondant à ce concept dans le langage cible.

De nombreux outils visant à faire de la transformation modèle vers texte existent. Un standard de l'OMG existe pour ce type de transformation : il s'agit de MOF2Text [104]. Dans ce standard est défini un métamodèle permettant de spécifier les transformations modèles vers textes à l'aide de patrons de génération de code. Les outils respectant ce standard doivent ainsi être capable de lire un modèle dans ce métamodèle et de l'interpréter afin de produire du code. Il n'existe pas d'outils implémentant complètement ce standard. Mais les outils disponibles en reprennent des idées, notamment sur l'utilisation de patron de génération de code. Ainsi des outils tels que Acceleo [5], M2T [4] et MOFScript [111] sont basés sur les patrons de code.

Lors de l'implémentation de l'environnement Gaspard, l'équipe a choisi l'utilisation de JET (Java Emitter Templates) provenant du projet M2T. Celui-ci a l'avantage d'avoir une excellente intégration à l'environnement Eclipse sur lequel repose Gaspard. JET est basé sur les patrons de générations de code comme la plupart des générateurs de code, mais possède l'avantage de pouvoir y inclure du code Java permettant d'effectuer des calculs durant la génération. À partir de ces patrons de génération de code, JET va générer des classes Java correspondant aux patrons. L'exécution d'un patron de génération se fait grâce à la méthode `generate()` générée pour chacun des patrons. Afin de faire la génération de code, il faut écrire un patron de génération pour chacun des concepts et appeler la méthode `generate()` sur la classe générée par JET pour le concept identifié. Afin de simplifier l'utilisation de JET, l'équipe à introduit une interface à JET appelée MoCodE (Model to Code Engine). Ce moteur de génération de code va simplifier l'appel des patrons en trouvant automatiquement (selon la classe de l'objet Java à traiter) le patron JET à appeler pour les objets que l'on passe à la méthode `generate()` de l'interface MoCodE.

5.6.2 Implémentation de la génération de code

Les générations de code Fortran et de code C ont été implémentées à partir du métamodèle. La génération de code est, comme dit précédemment, une écriture directe des concepts sous forme de texte.

Le patron de génération de code Fortran est présenté dans la figure 5.17. Les cinq premières lignes définissent le nom du paquetage, les différentes classes Java à importer, et le nom de la classe que le patron va générer (*IfElseStatement*). Les lignes suivantes (de 5 à 11) permettent de récupérer le moteur de transformation qui permettra d'appeler automatiquement les autres patrons et l'instance du concept concerné, ici une instance du concept IFELSESTATEMENT. Les lignes 13 et 14 provoquent la génération de l'expression qui représente la condition. Les lignes suivantes génèrent la structure d'un Si alors sinon. Ainsi un **If** suivit de l'expression entre parenthèses est généré, suivit de la structure du **Then** contenant la structure correspondante au **Then** dans le modèle, puis la structure correspondante au **Else** et la fin du if **end if**.

Thèse de Julien Taillard, Lille 1, 2009


```
<%@ jet package="fortranProgram"
   imports="fr.lifl.west.gaspard2.metamodel.ompprocedurallanguage.* java.util.*
   org.eclipse.emf.ecore.EObject"
   class="IfElseStatement" %>
<%
fr.lifl.west.mocode.engine.Engine ts =
         ((fr.lifl.west.mocode.engine.Argument) argument).getEngine();
fr.lifl.west.gaspard2.metamodel.ompprocedurallanguage.IfElseStatement iES =
         (fr.lifl.west.gaspard2.metamodel.ompprocedurallanguage.IfElseStatement)
         ((fr.lifl.west.mocode.engine.Argument) argument).getElement();
%>

<%String exp=ts.generate((EObject)iES.getCondition());%>
<%exp=exp.replace("\n","");%>
      IF ( <%=exp%> )
            THEN
                  <%=ts.generate((EObject)iES.getIfBody())%>
            ELSE
                  <%=ts.generate((EObject)iES.getElseBody())%>
      END IF
```

FIG. 5.17: Patron de génération JET/MoCodE pour le Fortran.

Le patron de génération de code C pour le même concept est présenté dans la figure 5.18. Les seuls changements observés par rapport au patron pour le Fortran sont dus à la syntaxe. Le **then** a disparu au profit des accolades et le **end if** a également disparu.

5.7 Conclusion du chapitre

Dans ce chapitre, nous avons présenté le métamodèle de langage procédural défini et utilisé dans le cadre du développement de l'environnement Gaspard. Bien que défini afin d'être utilisé dans Gaspard et afin de factoriser les transformations de modèles vers le Fortran et le C, le métamodèle est générique : il peut être utilisé dans d'autres approches modèles et pour d'autres langages que ceux citer précédemment. Il reprend les concepts généraux des langages de programmation impératifs afin de permettre la modélisation complète d'un programme.

Ce métamodèle est ensuite étendu afin de permettre la modélisation de programme parallèle que ce soit avec OpenMP ou MPI.

Les restrictions d'utilisations du métamodèle dans le cadre de Gaspard sont également définies.

La génération de code Fortran et C à partir d'un modèle conforme au métamodèle est introduite. Cette génération est réalisée à l'aide de patron de génération de code et est de forme «un pour un» puisque les concepts du métamodèle sont très proche des langages.

Ce métamodèle est le dernier utilisé dans notre environnement de conception basé sur les modèles. Le chapitre suivant présente les métamodèles et les transformations de modèles utilisés afin de produire un modèle dans le métamodèle de langages procéduraux à partir d'une spécification de haut niveau.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009


```
<%@ jet package="CProgram"
   imports="fr.lifl.west.gaspard2.metamodel.ompprocedurallanguage.* java.util.*
   org.eclipse.emf.ecore.EObject"
   class="IfElseStatement" %>
<%
fr.lifl.west.mocode.engine.Engine ts =
         ((fr.lifl.west.mocode.engine.Argument) argument).getEngine();
fr.lifl.west.gaspard2.metamodel.ompprocedurallanguage.IfElseStatement iES =
         (fr.lifl.west.gaspard2.metamodel.ompprocedurallanguage.IfElseStatement)
         ((fr.lifl.west.mocode.engine.Argument) argument).getElement();
%>

<%String exp=ts.generate((EObject)iES.getCondition());%>
<%exp=exp.replace("\n","");%>
      if ( <%=exp%> )
      {
            <%=ts.generate((EObject)iES.getIfBody())%>
      }
      else
      {
            <%=ts.generate((EObject)iES.getElseBody())%>
      }
```

FIG. 5.18: Patron de génération JET/MoCodE pour le C.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

© 2010 Tous droits réservés.

http://doc.univ-lille1.fr

Chapitre 6

Transformation d'un modèle Gaspard vers un modèle dans le métamodèle de langage

6.1 Modèles intermédiaires de haut niveau **106**
6.1.1 Le métamodèle Gaspard 106
6.1.2 Le métamodèle Polyhedron 107
6.1.3 Le métamodèle Loop 112
6.2 Un modèle complet pouvant être transformé **113**
6.2.1 Déploiement des composants élémentaires 113
6.2.2 Allocation des éléments terminaux du modèle 113
6.2.3 Spécification du composant principal 113
6.3 Du métamodèle Gaspard vers le métamodèle Loop **114**
6.3.1 Du métamodèle Gaspard vers le métamodèle Polyhedron 115
6.3.2 Du métamodèle Polyhedron vers le métamodèle Loop 117
6.4 Du métamodèle Loop vers le métamodèle de langage procédural contenant OpenMP **118**
6.4.1 Exemple sur lequel les étapes de la transformation seront illustrées 118
6.4.2 Ordonnancement du graphe de tâches hiérarchiques 119
6.4.3 Synchronisation des tâches 120
6.4.4 Gestion des tilers 121
6.4.5 Gestion des variables 123
6.4.6 Optimisations des accès mémoires 124
6.5 Implémentation des transformations **125**
6.5.1 Le moteur de transformation MoMoTE 125
6.5.2 Implémentation de la transformation métamodèle Loop vers métamodèle de langage procédural 129
6.6 Génération de code à partir de modèles et résultats d'exécution des codes générés **132**
6.6.1 Génération de code à partir d'un modèle 133
6.6.2 Machines de tests 136
6.6.3 Test de différents algorithmes de multiplication de matrices 137

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

6.6.4 Effet de la distribution sur les performances 139
6.6.5 Méthodologie d'utilisation de Gaspard et de modélisation des applications afin d'obtenir des performances 141
6.7 Conclusion du chapitre **142**

Comme présentée précédemment, la transformation d'un modèle de haut niveau vers un modèle dans le métamodèle de langage est réalisée en plusieurs étapes via différents métamodèles. Le but de ce chapitre est de présenter les grandes lignes de ces transformations en se focalisant particulièrement sur la transformation passant du métamodèle Loop vers le métamodèle de langage procédural, car elle représente le passage d'une application représentée sous la forme d'un graphe de tâches vers une application ayant une certaine séquentialité et utilisant des directives permettant le parallélisme. Les transformations dont nous allons discuter sont encadrées dans la figure 6.1.

Lors de ce chapitre seront présentés brièvement les métamodèles utilisés dans cette chaîne de transformation. Ensuite, sera introduit le concept de modèle exécutable qui définit tout ce que doit avoir un modèle afin de pouvoir être transformé et exécuté en utilisant la chaîne générant du code OpenMP. Puis les principes de compilation utilisés lors des transformations sont introduits : les transformations du métamodèle Gaspard vers le métamodèle Loop et enfin du métamodèle Loop vers le métamodèle de langage procédural ayant l'extension OpenMP. L'implémentation de ces transformations en règles de transformation est alors discutée. Pour finir, un exemple de transformation et des résultats d'exécutions de codes générés sont donnés afin d'en déduire une méthodologie d'utilisation de l'environnement Gaspard dans le cadre du calcul haute performance.

6.1 Modèles intermédiaires de haut niveau

Cette section présente les différents métamodèles utilisés dans la chaîne de transformation utilisée pour les applications de calcul haute performance. Ces trois métamodèles représentent des vues différentes d'un graphe de tâches basé sur des composants, c'est pourquoi les métamodèles utilisent la même base de métamodélisation de composants. Certains concepts changent entre ces métamodèles selon l'objectif de chacun d'entre eux. L'aspect de modélisation de l'architecture est présent dans les trois métamodèles. L'architecture n'est utile que lors de la distribution au niveau Gaspard dans la chaîne ciblant OpenMP. Celle-ci n'est plus utile pour OpenMP dans les métamodèles Polyhedron et Loop. C'est pourquoi le détail de la modélisation de l'architecture ne sera pas présenté ici. De plus amples informations pourront être trouvées dans les thèses d'Éric Piel [115], de Rabie Ben Atitallah [14] et d'Arnaud Cuccuru [38].

6.1.1 Le métamodèle Gaspard

Le métamodèle Gaspard a été présenté et illustré dans le paragraphe 2.4.5 du chapitre 2. Pour rappel, un modèle applicatif est représenté comme un graphe de tâches dans lequel tout le parallélisme potentiel (qu'il s'agisse de parallélisme de données ou de parallélisme de tâches) est exprimé à l'aide du langage de spécification Array-OL. Ce métamodèle exprime les mêmes concepts que le métamodèle UML et le profil Gaspard mais sans tous les concepts de UML qui nous sont inutiles. La description détaillée peut être trouvée dans l'annexe A de ce manuscrit.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

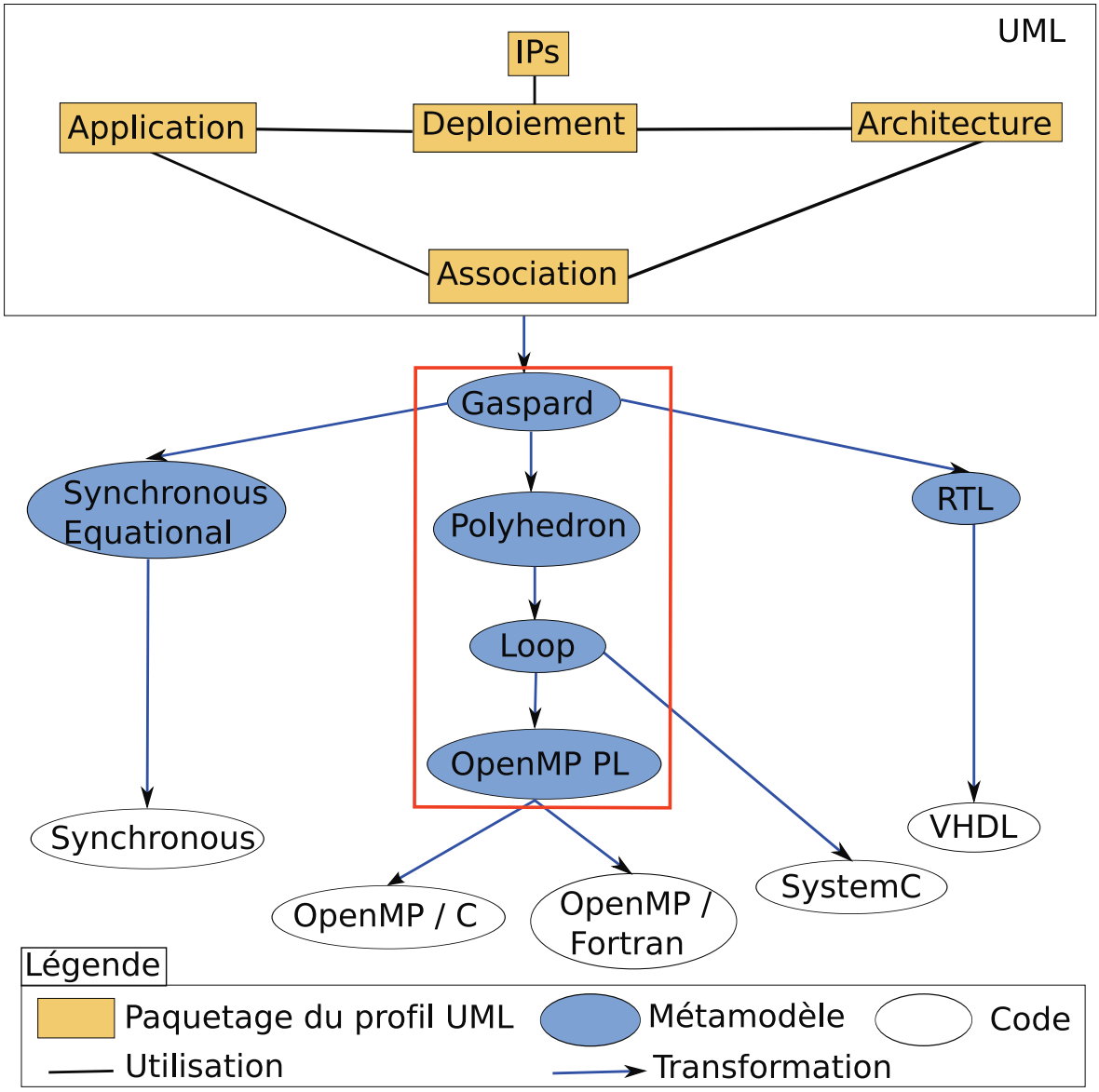


FIG. 6.1: Positionnement des transformations à gérer afin d'autoriser la génération de code OpenMP.

6.1.2 Le métamodèle Polyhedron

Ce métamodèle est commun aux chaînes de transformations vers le SystemC et OpenMP. Il a été réalisé en collaboration avec Éric Piel afin de convenir aux deux cibles. Certaines informations de ce métamodèle ne sont pas nécessaires à la génération de code OpenMP. Néanmoins, ces informations, telles que la distribution des données, sont utiles à la génération de code SystemC. Elles pourraient cependant être utilisées ultérieurement afin de générer du code pour machines à mémoires distribuées.

Ce métamodèle est dérivé du métamodèle Gaspard, un certain nombre de concepts sont donc communs aux deux métamodèles. Le but premier de ce métamodèle est de commencer à

se rapprocher des langages et notamment de consommer les informations d'associations et de distributions contenues dans le métamodèle Gaspard qui sont difficilement exploitables sous cette forme dans les langages. C'est pourquoi les tâches applicatives vont être « contenues » dans les processeurs sur lesquels elles doivent être exécutées (en réalité, moins que contenues, les tâches vont faire références aux processeurs sur lesquelles elles doivent être exécutées). La répétition des tâches distribuées sur les répétitions de processeurs est exprimée sous forme de polyèdres entiers. Cette représentation permet une simplification de la génération du code exprimant le placement des différentes répétitions d'une tâche.

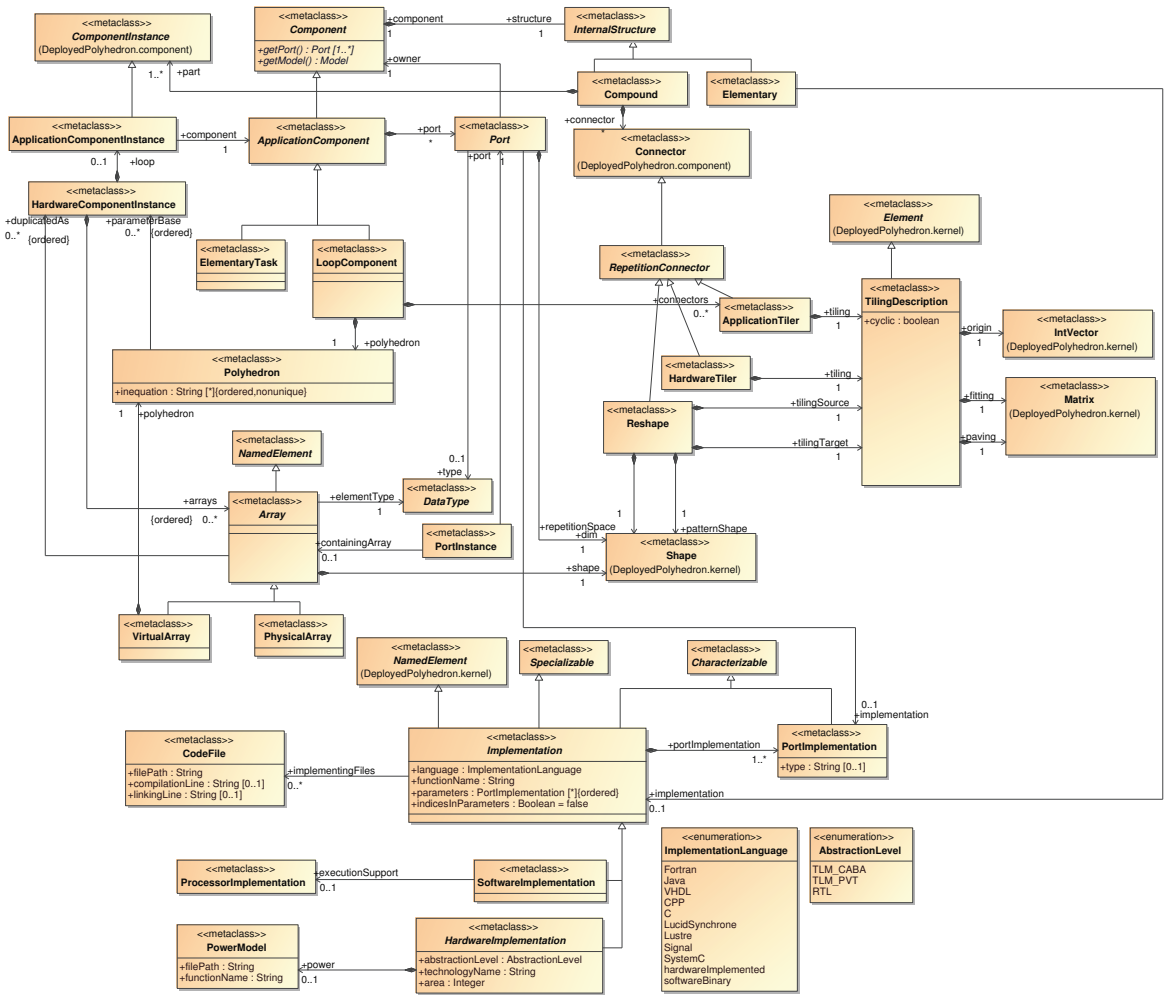


FIG. 6.2: Aperçu global du métamodèle Polyhedron.

La figure 6.2 représente une vue globale de la partie applicative du métamodèlePolyhedron. On peut notamment noter l'absence de toutes notions d'associations et de distributions qui ne font pas parties de ce métamodèle.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

6.1.2.1 Les composants applicatifs

Tout comme dans le métamodèle Gaspard, la modélisation à ce niveau est basée sur la notion de composants et d'instances de composants. A ce niveau, il ne reste que deux types de composants applicatifs : les ELEMENTARYTASKS qui correspondent aux composants élémentaires de Gaspard, et les LOOPCOMPONENTS qui contiennent des sous composants ainsi que la répétition. La figure 6.3 présente la structure des composants applicatifs dans le métamodèle. Les LOOPCOMPONENTS différent des composants répétitifs de Gaspard de par le fait que la répétition et la distribution ne sont plus associées à l'instance du composant contenu, mais à la structure du composant contenant. Cela est plus proche du code généré puisque les *Tilers* entourant l'instance répétée dépendent également de l'espace de répétition.

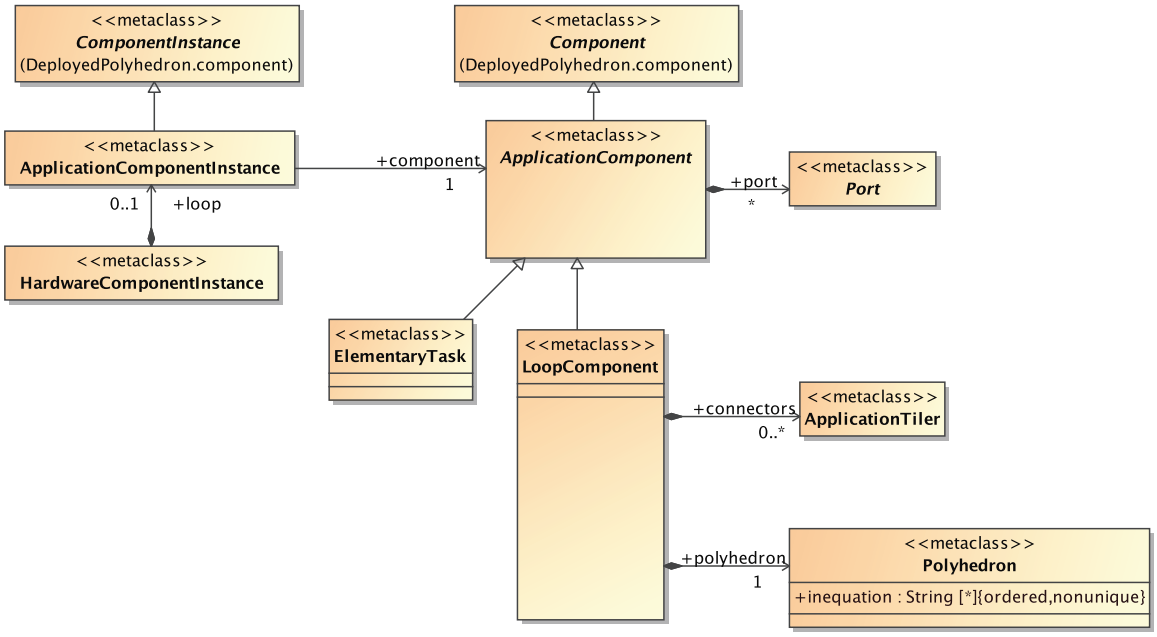


FIG. 6.3: Les composants applicatifs dans le métamodèle Polyhedron.

Une autre différence significative entre le modèle applicatif de Gaspard et l'application dans le métamodèle Polyhedron sont les connecteurs. Deux types de connecteurs sont différenciés : les connecteurs utilisés pour les accès mémoires et les connecteurs exprimant une dépendance de données entre des tâches. Les connecteurs utilisés pour les accès mémoires sont les APPLICATIONTILERS. Ils sont utilisés même si l'accès aux données est direct (i.e il n'y a pas besoin de construire de motifs intermédiaires). Les APPLICATIONTILERS sont placés dans un LOOPCOMPONENT et connectent les PORTS du LOOPCOMPONENT aux instances de ports des instances de composants contenues par le LOOPCOMPONENT. Les CONNECTORS sont eux utilisés pour exprimer les dépendances de données entre instances de composants. Cette notion d'accès mémoire, qui est importante afin de générer les appels matériels permettant l'accès aux données dans une simulation SystemC, sera vue comme des dépendances de données à la génération de code OpenMP.

6.1.2.2 Répétitions des tâches

Le fait que les tâches soient directement associées aux processeurs sur lesquels elles vont être exécutées modifie l'expression de la répétition et de la distribution. Dans le métamodèle Gaspard, la répétition est exprimée par une SHAPE représentant l'espace de répétition alors que la distribution exprime, grâce à une dépendance, comment des parties de cet espace de répétition sont associées aux répétitions des processeurs à l'aide de deux tilers, un espace de répétition de la distribution (*repetitionSpace*) et de la dimension du motif manipulé (*patternShape*). Cette notion de distribution est intéressante à utiliser pour les concepteurs, mais n'est pas simple à compiler. En effet, afin d'exprimer les placements de différentes répétitions d'une tâche sur la répétition de processeurs, il faut parcourir tout l'espace de répétition et calculer pour chaque indice dans l'espace de répétition sur quel processeur cette répétition doit être exécutée. Pour cela, nous proposons l'utilisation de polyèdres entiers paramétrés par les indices des processeurs. Les polyèdres sont des structures mathématiques couramment utilisées en programmation parallèle que ce soit pour la parallélisation automatique ou pour l'analyse de dépendance.

En mathématique, un polyèdre est défini comme un ensemble dans un espace affine réel de dimension quelconque. Cet espace peut être borné ou non. Nous utilisons des polyèdres convexes qui sont des ensembles définis par un nombre fini de demi-espaces. Ces demi-espaces sont exprimés par des inéquations affines. Un polyèdre est donc défini par un nombre fini d'inéquations. Un polyèdre simple est illustré dans la figure 6.4. Dans cette figure, dans un espace à deux dimensions, les inéquations représentent des droites (en vert) et forment un polyèdre en forme de triangle (l'espace compris entre les droites). Un polyèdre paramétré est un polyèdre, comme défini précédemment, contenant en plus des équations dépendantes de variables (les paramètres). Ces paramètres permettent de réduire le polyèdre à un ensemble plus petit dépendant de ceux-ci.

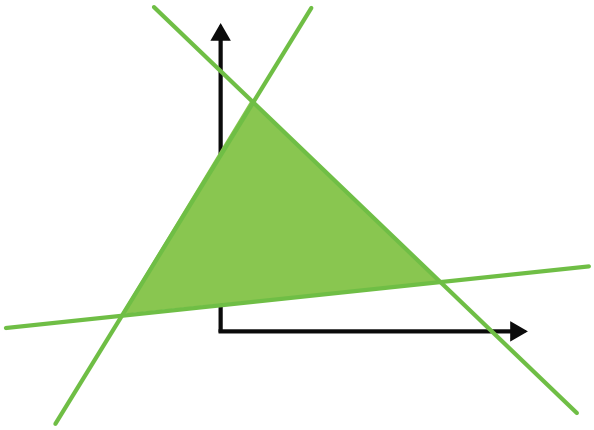


FIG. 6.4: Un polyèdre.

Dans le cadre de Gaspard, nous allons créer des polyèdres paramétrés par le numéro du processeur. Ces polyèdres paramétrés vont exprimer quelles parties de l'espace de répétition d'une tâche doit effectuer chacun des processeurs. Des notations sont définies afin que le polyèdre soit compréhensible lors de la transformation, mais également par un utilisateur/-développeur humain. Ainsi, les indices dans l'espace de répétitions d'une tâche sont notés x. Ces indices sont notamment utilisés pour le calcul des tilers. Considérant un espace de

répétition d'une tâche en deux dimensions, l'indice de répétition dans la première dimension de cet espace sera noté *x0* alors que l'indice dans la seconde dimension sera noté *x1*. Les indices dans l'espace de répétition des processeurs sera noté *p*. Dans le cas de la génération de code OpenMP, l'espace de répétition étant à une dimension (on exprime en fait le nombre de threads), l'indice de répétition sur les processeurs sera toujours *p0*. Un polyèdre tel qu'utilisé dans Gaspard est présenté dans la figure 6.5. Dans cette figure, un espace de répétition à deux dimensions d'une tâche est représenté. Les points représentent chacun un indice dans l'espace de répétition. Les deux types de points représentent chacun un processeur. Le type associé aux indices de l'espace de répétition exprime sur quel processeur la répétition associée aux indices doit être exécutée.

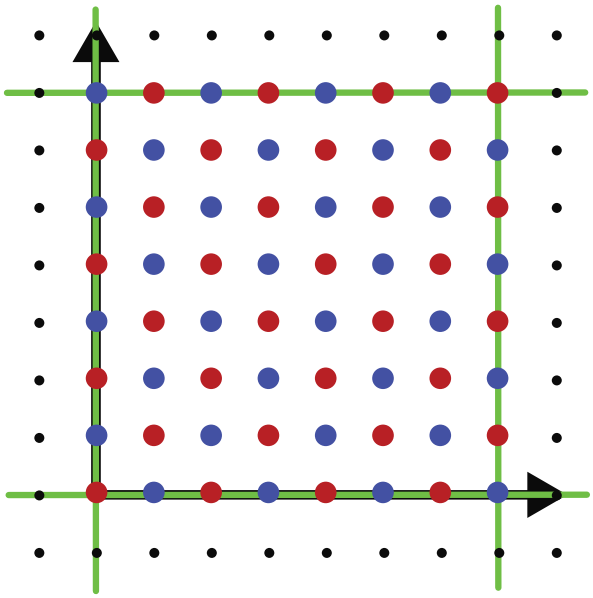


FIG. 6.5: Un polyèdre paramétré.

Dans le métamodèle Polyhedron, chaque LOOPCOMPONENT possède son polyèdre. Un polyèdre (concept POLYHEDRON) est représenté par une suite de chaînes de caractères exprimant les équations et inéquations du polyèdre dans un format spécifique. Ce format est proche de celui utilisé dans la transformation suivante afin de ne pas avoir à modifier la chaîne pour l'appel de cet outil. La figure 6.6 illustre le concept de POLYHEDRON dans le métamodèle.

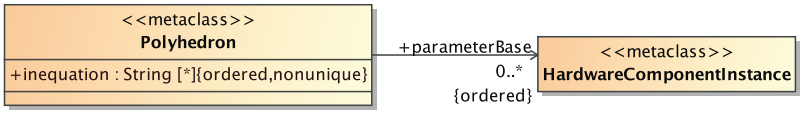


FIG. 6.6: Le concept de Polyhedron dans le métamodèle.

En terme de métamodèlisation, il aurait été possible de modéliser plus finement les équations et inéquations que par une chaîne de caractères néanmoins cela n'aurait pas été d'une grande aide puisque, comme on le verra plus tard, le polyèdre est traité d'un bloc par

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

une application externe. Il aurait donc fallu reconstituer les inéquations et équations afin d'appeler l'application externe.

La construction des polyèdres est expliquée dans la section 6.3.1.1 expliquant la transformation d'un modèle Gaspard vers un modèle Polyhedron.

6.1.3 Le métamodèle Loop

Le but du métamodèle Loop est de se rapprocher toujours un peu plus des langages. Ce métamodèle est très proche du métamodèlePolyhedron. L'unique différence entre les deux métamodèles est l'expression de la répétition et de la distribution, des boucles vont remplacer les polyèdres. Ces boucles vont parcourir les polyèdres selon les indices de répétitions de la tâche associée et elles seront réutilisées dans le code généré.

Les boucles sont représentées par le concept LOOPSTATEMENT dans le métamodèle. Dans l'environnement Gaspard, la génération des boucles est effectuée par un outil externe appelé CLooG (Chunky Loop Generator) [1]. La création de deux métamodèles proches (Polyhedron et Loop) permet de bien différencier l'appel à un outil externe dans la transformation.

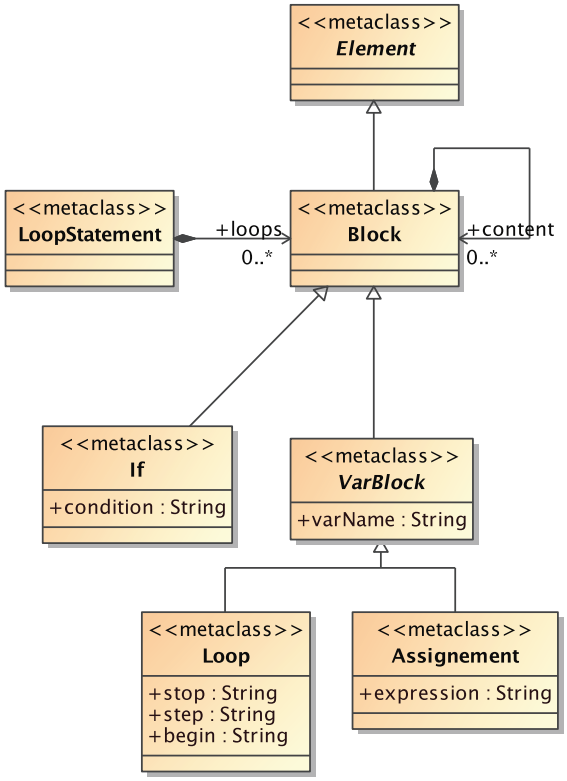


FIG. 6.7: Le représentation des boucles dans le métamodèle Loop.

La figure 6.7 illustre la construction d'un LOOPSTATEMENT dans le métamodèle Loop. Chaque composant LOOPCOMPONENT voit son polyèdre remplacé par une boucle parcourant ce polyèdre. Un LOOPSTATEMENT est composé par une suite ordonnée de BLOCK. Chacun des BLOCK peut être composé d'autres BLOCKS imbriqués. Un BLOCK représente une structure de la boucle. Trois types de structures sont possibles : une condition (IF), une boucle (LOOP) et

Thèse de Julien Taillard, Lille 1, 2009

l'affectation d'une valeur à une variable (ASSIGNEMENT). Chacune des structures contient des chaînes de caractères représentant les expressions utilisées en pseudo-code. Ces trois types de structures suffisent à modéliser toutes les boucles parcourant les différents polyèdres : ce sont les structures fournies par l'outil externe appelé.

6.2 Un modèle complet pouvant être transformé

La compilation (transformation) d'un modèle de haut niveau (Gaspard) ne peut se faire que lorsque la définition de celui-ci est complète. Pour la compilation vers du code OpenMP, les spécifications (sur la partie applicative du modèle) suivantes sont impératives :

- déploiement de tous les composants élémentaires ;
- allocation de chaque élément terminal du modèle à un processeur ;
- spécification du composant principal.

6.2.1 Déploiement des composants élémentaires

Dans les modèles utilisés, les composants élémentaires sont considérés comme des boîtes noires correspondant à une fonction dans le langage cible. Afin d'être capable de faire l'appel à la fonction correspondant au composant élémentaire lors de la phase de génération de code, il faut que celui-ci soit déployé sur une implémentation (concept SOFTWAREIMPLEMENTATION). Il faut donc impérativement que chacun des composants élémentaires soit déployé afin de connaître la fonction qu'il faudra appelée dans le code généré.

6.2.2 Allocation des éléments terminaux du modèle

L'exécution d'une tâche ne peut se faire que sur un processeur ou processus léger. La compilation devant générer du code SPMD, l'allocation de chaque tâche doit être spécifiée. Cette spécification peut être faite directement sur l'instance terminale dans le modèle (via TASKALLOCATION et si nécessaire DISTRIBUTION), mais également par l'intermédiaire de la hiérarchie. Une instance de tâche peut être associée via les instances de composants la contenant, la règle étant que dans l'arbre de la hiérarchie, une instance ne peut être placée qu'une seule et unique fois.

La figure 6.8 illustre une application totalement allouée. Les éléments ronds représentent les instances dans le modèle. Les connecteurs entre ces éléments représentent l'appartenance d'une instance à une autre dans la hiérarchie. Les éléments vert représentent une machine quadri-processeurs : l'élément C représente la machine qui contient (4) instances de CPU. Les éléments jaunes représentent une application. L'instance T1 contient deux instantes (T2 et T3). La tâche (et feuille) T2 est distribuée directement sur les processeurs. Les tâches T4 et T5 ne sont pas allouées directement, mais via la tâche les contenants, T3. La tâche T3 est distribuée sur les processeurs et les tâches T4 et T5 suivent cette distribution, elles sont donc bien allouées.

6.2.3 Spécification du composant principal

Afin de générer le bon programme, il faut connaître le composant principal (le *main*). Pour cela, une instance dans le modèle UML doit être stéréotypé « MainInstance ». Il doit y avoir

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

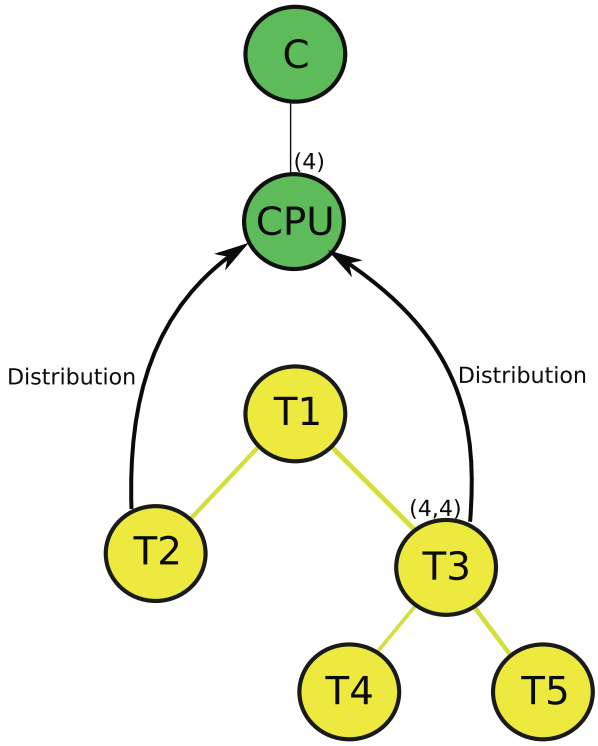


FIG. 6.8: Exemple d'une application dans laquelle les tâches sont toutes allouées.

une seule « MainInstance » par modèle (tout comme il ne doit y avoir qu'une seule fonction principale par programme C).

6.3 Du métamodèle Gaspard vers le métamodèle Loop

Cette section présente brièvement les transformations allant du métamodèle Gaspard vers le métamodèle Loop. Cette description est assez brève d'une part parce que la première partie de la transformation (Gaspard vers Polyhedron) a été définie en collaboration avec Éric Piel lors de sa thèse et développée par celui-ci. D'autre part parce que la transformation du métamodèle Polyhedron vers le métamodèle Loop est assez simple du fait que les métamodèles sont très proche et consiste en l'appel d'un programme externe : une boîte noire.

Durant la description de ces transformations, la transformation des concepts d'architecture ne sera pas explicitée puisque les concepts d'architecture ne changent pas entre le métamodèle Gaspard et le métamodèleLoop. Cette transformation est de type «un vers un» : un composant matériel dans le métamodèle source va générer le même type de composant dans le métamodèle cible.

Les transformations explicitées dans les sections suivantes interviennent uniquement sur les parties Application et Association du modèle Gaspard.

Thèse de Julien Taillard, Lille 1, 2009

6.3.1 Du métamodèle Gaspard vers le métamodèle Polyhedron

Cette transformation est la première transformation de modèle de la chaîne de compilation. Le but de la transformation est de rassembler les modèles d'architecture et d'application en un seul modèle en consommant les informations d'associations et de distributions de l'application sur l'architecture afin de se rapprocher d'un modèle d'un code exécutable. Les deux étapes essentielles de cette transformation sont la génération des polyèdres afin de gérer la répétition des tâches et le placement (association et/ou distribution) des tâches sur les processeurs. Cette partie de la transformation est explicitée dans le paragraphe 6.3.1.1. La seconde étape de la transformation est la gestion de la hiérarchie des modèles afin que chaque tâche soit placée dans un contexte particulier, cela va être géré par une duplication des composants nécessaires et l'instanciation du bon composant dans le bon contexte. Cette partie est présentée dans le paragraphe 6.3.1.2.

6.3.1.1 Génération des polyèdres

Dans le métamodèle Polyhedron, il y deux types de composants applicatifs : les ELEMENTARYTASKS et les LOOPCOMPONENTS. Un composant élémentaire Gaspard est directement transformé en ELEMENTARYTASK. Les LOOPCOMPONENTS sont eux crées à la gestion des instances de composants puisqu'ils représentent la répétition et la distribution des instances qu'ils contiennent. Un des points principaux de la génération des LOOPCOMPONENTS est la création des polyèdres exprimant la répétition et la distribution des tâches.

Lors de la transformation des instances de composants, les informations disponibles sont : la répétition de la tâche, sa distribution et en suivant la distribution on obtient le groupe de processeurs sur lequel doivent être exécutées les répétitions de la tâche. En obtenant ce groupe de processeurs, on obtient la répétition des processeurs. Ce sont les informations nécessaires à la génération du polyèdre associé au concept de LOOPCOMPONENT. Fixons les notations pour tous ces concepts :

- informations provenant de la distribution :
  - tiler sur la répétition de l'application (O_{sw},P_{sw},F_{sw}) ;
  - espace de répétition de la distribution ($s_{pattern}$) ;
  - motif utilisé par la distribution (s_{array}) ;
  - tiler sur la répétition de processeurs (O_{hw},P_{hw},F_{hw}) ;
- espace de répétition de la tâche applicative (s_{Sw}) ;
- espace de répétition des processeurs (s_{Hw}).

Le polyèdre doit permettre de calculer quels indices ($\mathbf{T_p}$) de l'espace de répétition de l'application doivent être exécutés sur le processeur d'indice ($\mathbf{P}$).

Pour cela, le polyèdre constitué d'équations et d'inéquations est construit comme suit :

$$\begin{cases} \mathbf{T_p} + s_{sw}.\mathbf{m_s} = O_{sw} + P_{sw}.\mathbf{r} + F_{sw}.\mathbf{i} \\ \mathbf{P} + s_{hw}.\mathbf{m_h} = O_{hw} + P_{hw}.\mathbf{r} + F_{hw}.\mathbf{i} \\ 0 \leq \mathbf{r} < s_{pattern} \\ 0 \leq \mathbf{i} < s_{array} \\ 0 \leq \mathbf{P} < s_{hw} \\ 0 \leq \mathbf{T_p} < s_{sw} \end{cases} \tag{6.1}$$

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Les deux équations proviennent de l'expression de calcul des tilers du côté applicatif et matériel de la distribution. Les vecteurs $\mathbf{m_s}$ et $\mathbf{m_h}$ sont introduit dans les équations afin de gérer l'aspect torique pouvant être utilisé dans les tilers et peuvent avoir une valeur positive quelconque. Les inéquations permettent de borner le polyèdre. Ainsi $\mathbf{P}$ doit être compris dans l'espace de répétition des processeurs. De même, $\mathbf{T_p}$ est compris dans l'espace de répétition de la tâche applicative. Par construction, le polyèdre utilise toutes les informations de la distribution. Aucune limitation n'est posée sur la construction du polyèdre, un polyèdre peut donc être construit grâce à l'équation (6.1) à partir de n'importe quelle distribution.

L'équation 6.1 exprime la construction des polyèdres dans le cas d'une instance placée sur un groupe de processeurs. Dans ce cas, le polyèdre permet de parcourir pour chacun des processeurs, les indices de l'espace de répétition de la tâche qu'ils doivent exécuter. Cela ne représente que quelques cas dans le modèle puisqu'une fois qu'une instance de composant est placée sur un processeur, la hiérarchie du composant suit le même placement. La construction du polyèdre est plus simple quand l'instance n'est pas directement placée : le polyèdre exprime alors comment parcourir l'espace de répétition de la tâche. Pour cela le polyèdre généré est simple et consiste en un ensemble d'inéquations : $0 \leq \mathbf{T_p} \leq s_{sw}$.

Nous allons illustrer la construction d'un polyèdre à partir d'un exemple. La figure 6.9 illustre une distribution d'une tâche répétée (4,4) fois sur un ensemble de quatre processeurs. La distribution présentée ici est une distribution par colonne : chaque processeur va avoir à effectuer une colonne de calcul.

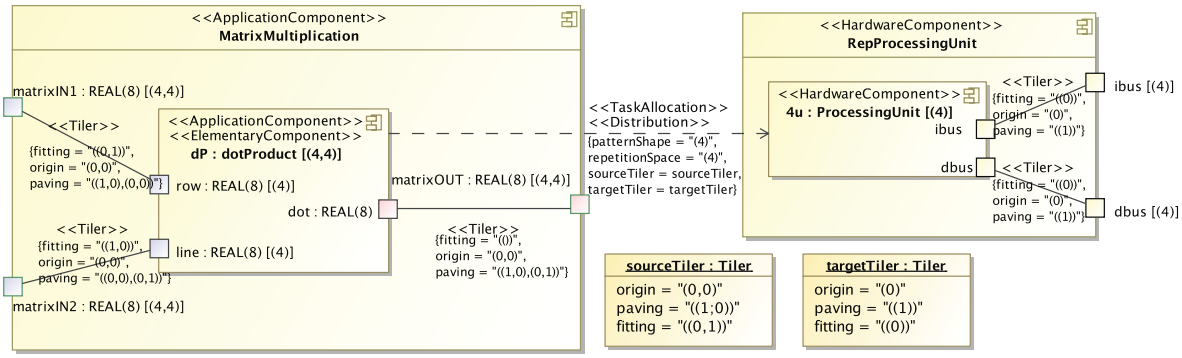


FIG. 6.9: Exemple de distribution d'une tache répétée sur un ensemble de processeurs.

Définissons les vecteurs dont nous allons avoir besoin pour définir le polyèdre de cette distribution. L'espace de répétition de la tâche contient deux dimensions, le vecteur $\mathbf{T_p}$ va alors contenir deux valeurs. L'espace de répétition des processeurs n'a qu'une seule dimension, $\mathbf{P}$ ne contient alors qu'une valeur. De même l'espace de répétition de la distribution (*repetitionSpace*) et la taille du motif géré (*patternShape*) n'ont qu'une dimension, les vecteurs $\mathbf{r}$ et $\mathbf{i}$ n'ont alors qu'une seule valeur. Il n'y aura de vecteurs $\mathbf{m_s}$ et $\mathbf{m_h}$ puisque les propriétés toriques sur les tableaux ne sont pas utilisées.

Les notations des vecteurs sont alors fixées :

- $\mathbf{T_p} = \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$;
- $\mathbf{P} = \begin{pmatrix} p_0 \end{pmatrix}$;
- $\mathbf{r} = \begin{pmatrix} d_0 \end{pmatrix}$;

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

$$\begin{cases} p_0 \geq 0, 3 - p_O \geq 0 \\ -p_0 + q_0 = 0 \\ -x_0 + q_0 = 0 \\ -x_1 + d_0 = 0 \\ q_0 \geq 0, 3 - q_0 \geq 0 \\ d_0 \geq 0, 3 - d_0 \geq 0 \\ x_0 \geq 0, 3 - x_0 \geq 0 \\ x_1 \geq 0, 3 - x_1 \geq 0 \end{cases} \quad (6.2) \qquad \begin{cases} q_0 = p_0 \\ x_0 = q_0 \\ x_1 = d_0 \\ 0 \leq p_0 \leq 3 \\ 0 \leq q_0 \leq 3 \\ 0 \leq d_0 \leq 3 \\ 0 \leq x_0 \leq 3 \\ 0 \leq x_1 \leq 3 \end{cases} \quad (6.3)$$

– $\mathbf{i} = (q_0)$.

Le polyèdre automatiquement construit pour cet exemple en appliquant l'équation 6.1 est illustré dans l'équation 6.2, sa simplification est donnée dans l'équation 6.3.

Les inéquations bornent les valeurs qui peuvent être prises par les vecteurs. Ainsi les valeurs x_0 et x_1 sont comprises dans l'espace de répétition de la tâche qui est de (4,4).

6.3.1.2 Arbre de tâches

Le second effet de la suppression des concepts d'associations est que l'arbre d'application est partagé entre différents processeurs. Afin de placer les tâches dans le bon contexte, un refactoring (une réorganisation) du graphe de tâches est effectué afin de réaliser ce travail : on modifie le modèle dans le métamodèle Polyhedron afin de s'adapter à cette exigence. L'algorithme utilisé commence par repérer chaque tâche à l'origine d'une branche : la tâche la plus basse d'une branche de l'arborescence qui soit explicitement allouée sur un processeur. Pour chacune de ces tâches, la règle de transformation parcourt le chemin allant de la racine de l'application à la tâche d'origine et chaque noeud est recopié sur dans la branche allouée au processeur d'origine. Si ce noeud a déjà été dupliqué dans la branche, car une autre tâche partage une partie du chemin alors le noeud déjà présent sera utilisé sans recopie. On va donc obtenir un graphe de tâches dans lequel chaque branche d'application placée sur des processeurs sera clairement identifiée.

6.3.2 Du métamodèle Polyhedron vers le métamodèle Loop

Cette transformation est effectuée entre deux métamodèles assez proche puisque l'unique différence entre les deux métamodèles est que le polyèdre du métamodèle Polyhedron va être transformé dans une boucle parcourant ce polyèdre. Cette transformation va faire appel à un programme externe qui va produire les boucles à partir des polyèdres, c'est pourquoi le niveau intermédiaire a été défini.

La génération de tous les éléments du modèle est de type «un pour un» excepté pour les polyèdres. La génération des boucles parcourant les polyèdres est effectuée par un outil nommé CLooG (Chuncky Loop Generator) écrit par Cédric Bastoul [1, 12]. Cet outil prend en entrée un polyèdre dans un format spécial et génère les boucles parcourant ce polyèdre. L'outil est écrit en langage C, il est donc important que le moteur de transformation utilisé pour faire les transformations permette de faire des appels externes. Afin d'appeler CLooG

Thèse de Julien Taillard, Lille 1, 2009


```
do d0=0,3
    q0=p0
    x0=p0
    x1=d0

end do
```

FIG. 6.10: Boucle Fortran générée pour le polyèdre présenté dans l'équation 6.2.

facilement, une interface JNI (Java Native Interface) a été écrite. Cette interface permet de récupérer, dans une structure Java, les informations fournies par l'outil.

Reprenons le polyèdre généré précédemment dans l'équation 6.2. En passant ce polyèdre dans l'outil CLooG et en spécifiant que le numéro du processeur est un paramètre, on obtient la boucle illustrée dans la figure 6.10. Dans cette boucle, on peut noter que x_0 est toujours égal à p_0 alors que x_1 varie avec d_0 de 0 à 3, chaque processeur va donc exécuter une colonne de calcul.

6.4 Du métamodèle Loop vers le métamodèle de langage procédural contenant OpenMP

La transformation du métamodèle Loop vers le métamodèle OpenMP est une transformation radicale puisque l'on doit passer d'un graphe de tâche hiérarchique à un modèle proche d'une forme textuelle dans lequel l'ordre des éléments a une signification. Dans cette transformation, il faut donc être capable d'effectuer le calcul des tilers, de déclarer des variables, d'ordonnancer les différentes tâches afin d'obtenir un programme valide, de synchroniser les tâches parallèles et de générer les appels aux tâches élémentaires avec les bons paramètres. La génération doit produire un programme parallèle utilisant le style SPMD.

6.4.1 Exemple sur lequel les étapes de la transformation seront illustrées

Afin d'expliquer les différentes étapes de transformation à réaliser entre le métamodèle de Loop et le métamodèlede langage procédural, celles-ci seront illustrées sur un même exemple. La figure 6.11 présente une représentation graphique de la partie applicative de la multiplication de matrice dans le métamodèle Loop. Dans cette représentation sur laquelle seront illustrées les différentes transformations, les instances de composants sont illustrées à l'aide de forme ovale alors que les composants sont représentés sous forme de rectangle à bord arrondi. Les petits rectangles qui surmontent les instances représentent les boucles associées aux composants instanciés. Pour rappel, les valeurs importantes dans les boucles sont les x_N qui représentent l'indice de la répétition de la tâche a exécuter et les p_N qui sont les numéros des processeurs dans l'espace de répétition (pour OpenMP, il n'y a qu'une seule dimension de thread, c'est pourquoi on utilisera toujours p_0). Les dépendances de données sont représentées par les flèches orientées illustrant le sens de circulation des données (de gauche à droite).

En haut de la figure, on retrouve l'instance principale *MainLC* (représentant la fonction principale). La boucle associée au composant instancié est très simple et permet d'exécuter une seule fois le composant puisque l'instance principale n'est pas répétée. Le composant principal instancié est illustré par la partie rectangulaire. Ce composant est composé par

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

quatre instances. Les deux premières instances (dans l'ordre du graphe de tâches, c'est à dire de gauche à droite) sont destinées à créer et initialiser les matrices (lecture dans un fichier ou initialisation aléatoire). Celles-ci sont placées sur des processeurs. En effet, l'instance *initMatrixLC1* possède une boucle (*if (p0 ==0)*) qui exprime qu'elle ne doit être exécutée que par le thread 0, l'instance *initMatrixLC2* devant être exécutée par le thread 1. Ensuite, la multiplication de matrice est réalisée par l'instance *MatMulLC* dont les boucles expriment une distribution de la répétition des instances internes sur les différents processeurs (les valeurs de X_0 et X_1 dépendent de P_0, la boucle exprime alors une distribution). Enfin, une vérification de la matrice est effectuée sur le thread 0 par l'instance *checkMatrixLC*.

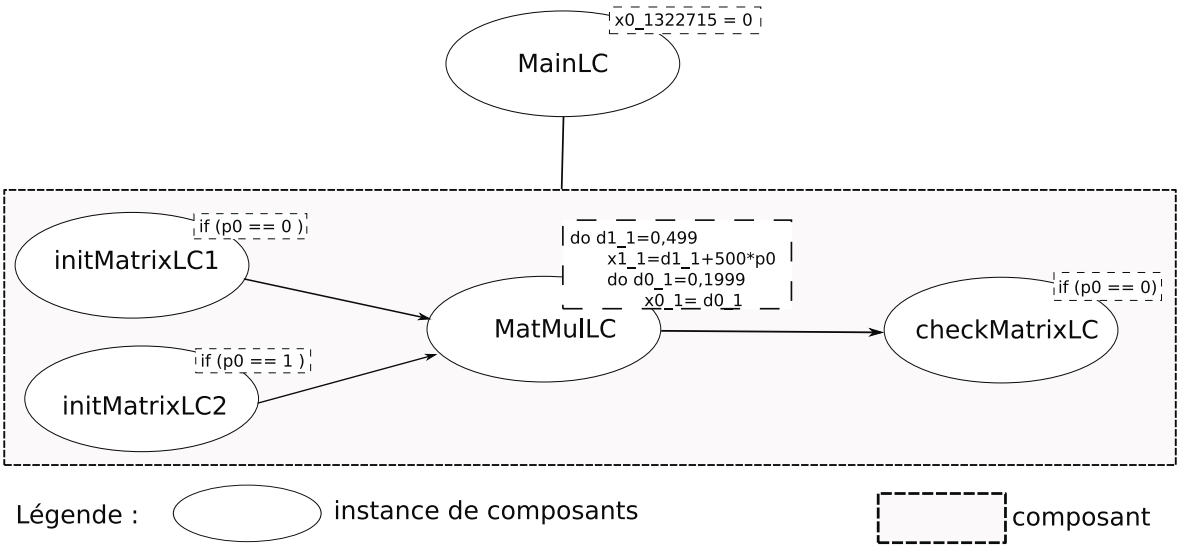


FIG. 6.11: La multiplication de matrice au niveau Loop.

Sur cet exemple seront expliquées les différentes étapes à effectuer lors de la transformation. Lors des illustrations, nous ne prendrons que le composant instancié par *MainLC* (l'instance principale ne servant qu'à savoir qu'il faut générer un programme).

6.4.2 Ordonnancement du graphe de tâches hiérarchiques

Une des étapes de la transformation d'un modèle Loop vers un modèle de langage procédural est d'ordonnancer les tâches constituants le graphe de tâches afin d'obtenir un programme produisant le bon résultat. En effet, dans un programme, les structures sont ordonnées de façon à définir un ordre d'exécution entre ces structures. À la transformation, il faut donc produire les structures dans le métamodèlede langage procédural dans un ordre respectant l'ordonnancement calculé entre les tâches du métamodèle Loop. Cet ordonnancement est également utile lors de déclaration des variables (section 6.4.5).

Afin d'écrire la transformation sous forme de règles, l'ordonnancement est calculé localement à un niveau de hiérarchie (un LOOPCOMPONENT). La hiérarchie étant respectée par la transformation, ce type d'ordonnancement va produire un ordonnancement global valide. En effet, si un ordonnancement est calculé à un niveau de hiérarchie alors cela produit une structure dans le modèle cible assurant le respect de cet ordonnancement. Ensuite est calculé un ordonnancement pour chacunes des sous-tâches (si celles-ci sont composées) et les

Thèse de Julien Taillard, Lille 1, 2009

structures crées par celles-ci sont placées dans les structures correspondantes à la tâche créée dans le niveau de hiérarchie supérieur. Un ordonnancement calculé localement à chaque niveau de hiérarchie produira ainsi un ordonnancement global du graphe de tâche valide.

Chaque LOOPCOMPONENT va produire une structure dans le programme. Un ordonnancement est calculé pour chaque LOOPCOMPONENT afin d'établir un ordre d'exécution entre les instances contenues par le LOOPCOMPONENT. Le principe de base de l'ordonnancement est qu'une tâche ne peut être exécutée que lorsque toutes les données dont elle a besoin en entrée ont été produites. Cela revient ainsi à une analyse du graphe de tâches internes à un LOOPCOMPONENT en suivant les dépendances de données. Le principe de l'algorithme d'ordonnancement est illustré à la figure 6.12.

```
DonneesProduites=Port d'entree du LoopComponent
Li=liste des instances contenues par le LoopComponent
ordonnancement=liste ordonnee vide
Tant_que Li n'est pas vide
  Pour chaque instance i de Li
     //Si les donnees sur les ports d'entree de la tache ont ete produit
     Si i peut etre ordonnance
       Ajouter i a liste ordonnancement
       Ajouter les portsInstances de i a la liste DonneesProduites
       Enlever i de la liste Li
     FinSi
  FinPour
FinTant_que
```

FIG. 6.12: Algorithme calculant un ordonnancement local à un niveau de la hiérarchie des composants.

L'exécution de la règle hiérarchique va produire une hiérarchie de STATEMENT_LIST dans lesquels chaque instance va produire ensuite le code nécessaire à son exécution. Cette étape de la transformation est décrite dans la figure 6.13 sur l'exemple de la multiplication, on peut y voir l'ordonnancement du composant principal. Un ordonnancement valide de ces instances, respectant les dépendances de données, est le suivant : *initMatrixLC1*, *initMatrixLC2*, *MatMulLC*, *checkMatrix*. À droite de la figure, on peut observer la structure de programme créée pour un tel composant. On peut y observer des différents STATEMENT_LIST crées pour chacune des instances et l'ordre de l'ordonnancement qui est respecté.

6.4.3 Synchronisation des tâches

Lors de l'exécution d'un programme parallèle, les processeurs ne vont pas forcément à la même vitesse d'exécution (des accès à la mémoire peuvent notamment provoquer ces décalages entre processeurs). C'est pourquoi il est important de synchroniser les processeurs afin de s'assurer que toutes les données devant être produites par une tâche parallèle ont bien été produites avant d'exécuter la tâche suivante qui prend en entrée ces données.

La synchronisation est calculée en même temps que l'ordonnancement des tâches puisque tout comme pour l'ordonnancement celle-ci a besoin d'une analyse des dépendances de données pour être calculée. Deux types de cas peuvent être distingués : les tâches ayant des dépendances entre elles et celles n'en ayant pas. Les tâches n'ayant pas de dépendances de

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

données entre elles ne nécessitent pas de synchronisations, puisque celles-ci sont totalement indépendantes les unes des autres. Les tâches ayant des dépendances de données entre elles peuvent nécessiter des synchronisations entre elles, cela dépend du contexte dans lequel elles sont. Deux tâches placées sur des processeurs différents doivent être synchronisées puisque les temps d'exécutions sur les processeurs ne seront pas forcément les mêmes. En revanche, deux tâches ayant une dépendance de donnée, mais étant placées sur le même processeur (dans le même thread en OpenMP), ne nécessitent pas de synchronisation puisque la séquentialité de l'exécution sur un processeur va garantir que la première tâche ait fini son calcul avant que la seconde ne commence. En OpenMP, la synchronisation des threads se fait à l'aide de la directive `!$OMP Barrier` (en Fortran). Lors de la génération de la structure de code, il faudra donc générer de telles structures entre des tâches devant être synchronisées. Les synchronisations calculées pour la multiplication de matrices sont illustrées dans la figure 6.13. Toutes les tâches du composant principal étant exécutées sur des processeurs différents (leurs boucles dépendent de p0), il est important d'effectuer des synchronisations dès qu'il y a une dépendance de données. Les instances *initMatrixLC1* et *initMatrixLC2* n'ayant pas dépendances entre elles, il n'y a pas de synchronisation. Ensuite, ces deux instances ont une dépendance avec l'instance *MatMulLC*, il est donc nécessaire d'effectuer une synchronisation entre ces tâches, c'est pourquoi une barrière de synchronisation est générée dans le modèle OpenMP. Une barrière est également nécessaire entre les instances *MatMulLC* et *checkMatrixLC*. Dans les structures générées pour chacune des instances, il ne sera pas nécessaire de générer de synchronisation puisque celles-ci seront exécutées dans un thread et auront donc un comportement séquentiel.

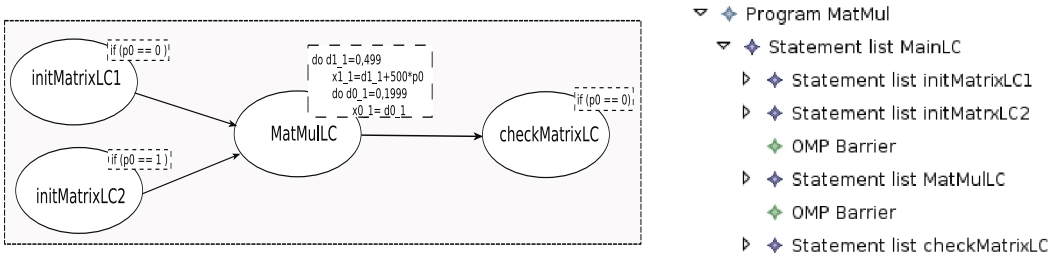


FIG. 6.13: Ordonnancement hiérarchique des tâches d'un modèle Loop et modèle produit dans le métamodèle de langage procédural par cet ordonnancement.

6.4.4 Gestion des tilers

Un tiler exprime la manière dont est extrait un motif d'un tableau pour un indice dans l'espace de répétition d'une tâche. Afin de produire du code exécutable, il faut, dans le code produit, construire à un moment donné ces motifs afin d'effectuer les calculs sur les bons motifs. Il faut donc déterminer quand et où construire ces motifs.

Dans un modèle du métamodèle Loop, les tilers sont présents entre les ports d'un LOOPCOMPONENT et les instances de ports d'une instance de composant contenue dans ce LOOPCOMPONENT. Le respect de la spécification des tilers amène dans un premier temps à construire les motifs à chaque niveau de hiérarchie. Cela consiste donc à déclarer, pour chacun des niveaux de hiérarchie, les motifs nécessaires quand un tiler est identifié et à produire le code permettant de copier les bonnes valeurs du tableau dans le motif (ou l'inverse si c'est

Thèse de Julien Taillard, Lille 1, 2009

un tiler de sortie). Cette solution a comme inconvénient de provoquer beaucoup de copies mémoires ainsi que de prendre beaucoup de place en mémoire. C'est pourquoi la notion de court-circuit a été introduite lors des thèses de Julien Soula et de Philippe Dumont. Le court-circuit réduit le nombre de motifs déclarés, en court-circuitant ces recopies, en permettant d'exprimer les éléments d'un motif d'une sous tâche selon les coordonnées des éléments du tableau de la tâche. Pour simplifier, cela permet de construire un tiler à partir de deux tilers et ainsi d'éviter la déclaration du motif intermédiaire qui aurait été nécessaire entre les deux tilers. Le principe du court-circuit est expliqué dans le paragraphe 6.4.4.1, son utilisation est elle présentée dans le paragraphe 6.4.4.2.

6.4.4.1 Le court-circuit

Le court-circuit construit un tiler à partir de deux tilers consécutifs. Nommons les tilers afin d'illustrer la construction du nouveau à tiler à partir des deux tilers. Le tiler au niveau supérieur de la hiérarchie sera noté T_{up} (O_{up},P_{up},F_{up}) et le tiler inférieur sera noté T_{sub} (O_{sub},P_{sub},F_{sub}).

Les équations Array-OL donnent l'expression suivante pour calculer les points de la tuile d'indice $\mathbf{r}$. Pour le point d'indice $\mathbf{i}$ dans le motif, les coordonnées du point correspondant dans le tableau sont :

$$\mathrm{O} + \begin{pmatrix} \mathrm{P} & \mathrm{F} \end{pmatrix} \times \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \tag{6.4}$$

En reprenant les équations Array-OL, nous avons donc pour nos deux tilers les équations suivantes :

$$\mathrm{O_{up}} + \begin{pmatrix} \mathrm{P_{up}} & \mathrm{F_{up}} \end{pmatrix} \times \begin{pmatrix} \mathbf{r_{up}} \\ \mathbf{i_{up}} \end{pmatrix} \tag{6.5}$$

$$\mathrm{O_{sub}} + \begin{pmatrix} \mathrm{P_{sub}} & \mathrm{F_{sub}} \end{pmatrix} \times \begin{pmatrix} \mathbf{r_{sub}} \\ \mathbf{i_{sub}} \end{pmatrix} \tag{6.6}$$

Considérant que le tiler inférieur (T_{sub}) est construit à partir du motif rempli par le tiler supérieur (T_{up}). On peut remplacer l'indice i_{up} par l'équation du tiler inférieur. On obtient alors :

$$\mathrm{O_{up}} + \begin{pmatrix} \mathrm{P_{up}} & \mathrm{F_{up}} \end{pmatrix} \times \begin{pmatrix} \mathbf{r_{up}} \\ \mathrm{O_{sub}} + \begin{pmatrix} \mathrm{P_{sub}} & \mathrm{F_{sub}} \end{pmatrix} \times \begin{pmatrix} \mathbf{r_{sub}} \\ \mathbf{i_{sub}} \end{pmatrix} \end{pmatrix} \tag{6.7}$$

En effectuant les multiplications, on obtient alors :

$$\mathrm{O_{up}} + \mathrm{P_{up}}\,\mathbf{r_{up}} + \mathrm{F_{up}}\,\mathrm{O_{sub}} + \begin{pmatrix} \mathrm{F_{up}}\,\mathrm{P_{sub}} & \mathrm{F_{up}}\,\mathrm{F_{sub}} \end{pmatrix} \times \begin{pmatrix} \mathbf{r_{sub}} \\ \mathbf{i_{sub}} \end{pmatrix} \tag{6.8}$$

$$\mathrm{O_{up}} + \mathrm{F_{up}}\,\mathrm{O_{sub}} + \begin{pmatrix} \mathrm{P_{up}} & \mathrm{F_{up}}\,\mathrm{P_{sub}} & \mathrm{F_{up}}\,\mathrm{F_{sub}} \end{pmatrix} \times \begin{pmatrix} \mathbf{r_{up}} \\ \mathbf{r_{sub}} \\ \mathbf{i_{sub}} \end{pmatrix} \tag{6.9}$$

Dans l'équation (6.9), on peut identifier les différentes parties du nouveau tiler construit. L'origine ne dépend ni de l'espace de répétition des tâches, ni de l'indice du point dans le

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

motif. Le pavage (*paving*) se calcule avec les indices dans l'espace de répétition des tâches (**r**). L'ajustage dépend lui de l'indice du point dans le motif construit (**i**). Le nouveau tiler construit est donc le suivant :

- $O = O_{up} + F_{up}\, O_{sub}$
- $P = \begin{pmatrix} P_{up} & F_{up}\, P_{sub} \end{pmatrix}$
- $F = \begin{pmatrix} F_{up}\, F_{sub} \end{pmatrix}$

6.4.4.2 Utilisation lors de la transformation

Le court-circuit est utilisé à la génération du modèle dans le métamodèle de langage procédural. Afin de limiter les copies mémoires, le motif produit par un tiler n'est alloué que lorsque ce motif va être utilisé par une tâche élémentaire : la tâche élémentaire devant effectuer les calculs, il lui faut le bon motif pour s'exécuter et produire le bon résultat. Le court-circuit est calculé comme suit : à chaque niveau de hiérarchie, si il existait des tilers reliés à ce niveau de hiérarchie alors on doit calculer le court-circuit pour chacun des connecteurs reliés à un tiler au niveau de hiérarchie précédent.

6.4.5 Gestion des variables

Pour s'exécuter, le programme a besoin de variables afin d'y stocker les valeurs nécessaires à son exécution. Celles-ci n'étant pas présentes dans le niveau d'abstraction utilisé dans le métamodèle Loop, il faut les insérer lors de la création du modèle de code procédural. La première idée venant à l'esprit est de déclarer une variable par port utilisé dans le graphe de tâches. Mais cela va représenter un grand nombre de variables et beaucoup d'entre elles ne seraient que des copies d'autres puisqu'un grand nombre des connecteurs (exceptés les tilers) ne sont que des dépendances de données n'effectuant aucun calcul sur les données. Une partie de l'efficacité du programme générée va dépendre du nombre de variables déclarées et des copies mémoires. Il faut donc réduire au maximum le nombre de variables déclarées et réutiliser les variables qui ne sont plus utilisées.

La base de la gestion des variables est que les ports servant à exprimer les dépendances de données ne doivent pas provoquer la déclaration d'une variable. Ainsi, les ports intermédiaires utilisés par les niveaux de hiérarchie n'ont pas à être déclarés. Seuls certains ports des tâches élémentaires doivent être déclarés afin de permettre l'exécution correcte de la tâche. Les principes de base de la déclaration des variables dans le cadre d'un modèle Array-OL à assignation unique sont les suivants (nous discuterons ensuite des différentes optimisations réalisées (§ 6.4.6). Dans le cas où il s'agit de tâches non répétées (i.e non connectées par un *tiler*), les ports de sorties des tâches élémentaires doivent être affectés à une variable puisqu'une tâche élémentaire va produire des données dans les variables correspondantes aux ports de sorties alors que les ports d'entrées (des ports élémentaires) n'ont pas à être déclarés puisque les données proviennent forcément d'un port de sortie on peut donc réutiliser directement la variable déclarée pour ce port de sortie. Dans le cas de tâches répétées, le nombre de variables à déclarer est modifié. En effet, dans le cadre d'un tiler entrant, il est nécessaire de déclarer le motif qui sera rempli à partir du tableau source et qui sera utilisé par la tâche. Dans le cadre d'un tiler sortant, le motif est déjà déclaré puisque comme dit précédemment les ports de sortie d'une tâche élémentaire doivent être déclarés sous forme de variable. En revanche, il faudra déclarer le tableau cible dans lequel le motif produit sera copié.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

L'implémentation de cela en règles de transformation n'est pas aussi simple puisqu'il faudrait parcourir toute la hiérarchie d'un seul coup et alors il n'y aurait qu'une unique règle de transformation. Pour résoudre cela, la déclaration de variables est effectuée par niveau de hiérarchie. Les variables sont affectées aux instances de port à un niveau donné. Lors de l'affectation des variables au niveau inférieur, il faut être capable de connaître ce qu'il se passe au niveau hiérarchique supérieur afin de réutiliser les bonnes variables dans lesquelles se trouvent les données. Ce type de transformation peut ne pas être réalisable avec n'importe quel type de règle de transformation.

6.4.6 Optimisations des accès mémoires

L'optimisation des accès mémoires est un point important pour les performances. Une bonne optimisation de ceux-ci permet notamment d'éviter un grand nombre de défauts de caches en minimisant le nombre de variables utilisées. Cela est particulièrement important pour les machines « classiques » qui n'ont pas de réseau d'interconnexion spécialisé et n'ayant donc pas d'accès parallèle à la mémoire. Parmi ces optimisations, certaines sont déterminées uniquement à la compilation, mais d'autres ont besoin d'être spécifiées dans la modélisation à haut niveau.

La première optimisation à réaliser est de réduire au maximum la déclaration de variables. En effet, si l'on prend un modèle Array-OL, il arrive très régulièrement (si ce n'est toujours) que des tableaux ne soient écrits et lus qu'une seule fois. Array-OL n'imposant pas de modèle d'exécution à assignation unique, une variable peut être réutilisée plus d'une fois. L'optimisation consiste donc à réutiliser les variables quand celles-ci ne sont plus utilisées. Pour cela, il faut, avant de déclarer une variable correspondant au port que l'on veut déclarer, chercher dans les variables qui ne sont plus utilisées si une variable ne pourrait pas correspondre à nos besoins (même type de données et même taille du tableau). Si c'est le cas, on peut alors réutiliser celle-ci sans en déclarer une nouvelle.

L'autre optimisation importante est l'utilisation des ports en entrée/sortie tel que défini dans le chapitre 4 paragraphe 4.2. L'utilisation de ces ports nécessite une gestion à la compilation puisqu'il ne faut pas déclarer tous les ports. Deux ports déployés sur le même *PortImplementation* doivent utiliser une seule et unique variable et non deux. Il faut donc identifier, lors de la compilation, les ports qui sont déployés sur la même implémentation afin de ne pas déclarer de variables inutiles et de réutiliser la bonne.

Enfin, il est également possible de réaliser une optimisation dans le cadre de la génération de code Fortran. En effet, en Fortran, il est possible de spécifier des sections d'un tableau en utilisant la notation suivante :

$[limite_1]$:$[limite_2]$[:pas]

Cette notation permet de définir une section dans chaque dimension d'un tableau. La section débute à $limite_1$ et se termine à ou avant $limite_2$ et prend un élément tous les pas. Le pas est optionnel et si celui-ci n'est pas explicitement donné, il est de un. Elle permet de ne pas avoir à déclarer de variables pour le motif intermédiaire ainsi que de ne pas générer les boucles remplissant le motif. Dans le cas de tilers réguliers et parallèles aux axes, il est tout à fait possible de générer cela sans déclarer de tableaux intermédiaires. Actuellement, cette optimisation est réalisée à ce niveau de la transformation (métamodèle Loop vers métamodèle OpenMP) mais cela réduit le fait de pouvoir générer du code C puisqu'une telle notation n'est pas supportée dans le langage C. Dans le cadre d'une refonte du code des transformations, il vaudrait mieux ne pas réaliser cette optimisation à ce niveau, mais la

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

réaliser dans le cadre d'un refactoring du modèle de code utilisé uniquement dans le cadre de la chaine allant vers le code OpenMP/Fortran.

6.5 Implémentation des transformations

Comme cela a été décrit lors de la présentation de l'IDM (paragraphe 2.3.3 du chapitre 2), le passage d'un modèle à un autre se fait par une transformation de modèle. Une transformation de modèle est constituée de règles de transformation qui exprime comment un (des) concept(s) du métamodèle source produit un(des) concept(s) dans le métamodèle cible. Bien que l'OMG ait standardisé un langage (QVT [108]) permettant d'exprimer les transformations de modèles, le concept de *règle de transformation* n'est jamais clairement défini. Cela a comme conséquence que chaque moteur de transformation possède sa propre définition d'une règle de transformation impliquant le fait que la même transformation ne sera pas exprimée de la même façon selon le moteur. Une définition de règle de transformation impose une certaine rigidité sur l'expression de la transformation qui implique que selon la transformation un moteur sera plus ou moins adapté.

Il existe différents moteurs de transformation respectant plus ou moins la norme QVT : ATL [75], Kermeta [134, 97], SmartQVT [130], QVTO [19]. Lors du début de l'implémentation de l'environnement Gaspard, nous avons dû choisir un moteur de transformation. Des contraintes techniques, telles que l'impossibilité d'appeler du code externe ainsi que la version limitée des outils à l'époque, ont rendu impossible l'utilisation de moteur existant et nous ont poussé à définir notre propre moteur de transformation répondant à nos besoins.

Le paragraphe suivant présente MoMoTE, le moteur de transformation que nous avons défini. Ensuite l'implémentation des règles de transformation à l'aide de MoMoTE est illustrée et discutée.

6.5.1 Le moteur de transformation MoMoTE

MoMoTE (Model to Model Transformation Engine) a été défini dans l'équipe projet INRIA DaRT afin de permettre l'implémentation des transformations pour l'environnement Gaspard.

MoMoTE est constitué d'un moteur de transformation et d'une API (Application Programming Interface) Java permettant la définition de règles de transformations. MoMoTE repose sur les librairies Ecore et EMFT Query permettant la définition et la manipulation de modèles. Ces librairies sont distribuées sous forme de plugin Eclipse, ce qui assure à MoMoTE une intégration facile à cet environnement (sur lequel est basé Gaspard).

Ce moteur permet la transformation de N modèles d'entrée en M modèles de sortie.

6.5.1.1 Écriture des règles de transformation

Les règles de transformations sont écrites dans une classe Java étendant la classe abstraite `Rule` définie par MoMoTE. La classe `Rule` déclare des méthodes que la règle doit implémenter. Ces méthodes sont utilisées par le moteur de transformation afin d'exécuter la règle.

Les méthodes qu'une règle de transformation doit implémenter sont :

- `getCondition();`
- `create();`
- `process();`

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

– `processReference()`.

Une autre méthode peut être surchargée, il s'agit de :
– `processQueryResult()`.

Un des avantages de MoMoTE sur les autres moteurs de transformation est qu'il est écrit en Java. Cela offre aux utilisateurs de disposer de toute la puissance de Java, notamment le mécanisme d'héritage de classe qui peut permettre la factorisation de différentes règles.

Les rôles des méthodes d'une règle MoMoTE vont être expliqués sur une règle de transformation illustrée figure 6.14. Cette règle de transformation permet la transformation d'un LOOPCOMPONENT du métamodèle Polyhedron vers un LOOPCOMPONENT du métamodèle Loop.

Le constructeur sert principalement à déclarer les sous-règles de la règle de transformation. La méthode `addRule()` ajoute une sous-règle à la règle en indiquant le point d'attache des éléments par les règles. Le constructeur `PolyhedronLoopComponent2LoopLoopComponent()` ajoute ici les sous-règles permettent la gestion des différents types de port du composant, des connecteurs et de la structure interne du composant.

`getCondition` indique la requête EMFT à effectuer sur le modèle pour trouver les motifs d'entrées de la règle. Le but de la règle illustrée ici est de transformer tous les LOOPCOMPONENTS du modèle source (le métamodèle Polyhedron) en LOOPCOMPONENTS du modèle cible (le métamodèle Loop). La condition qui est requise ici est donc d'être un LOOPCOMPONENT du métamodèle Polyhedron.

`create` indique quel élément, dans le métamodèle cible, doit être crée pour chacun des éléments respectant la condition de `getCondition()`. Cette méthode est donc appelée pour chacun des éléments renvoyés par l'exécution de la condition sur le modèle. Dans la règle présentée ici, un LOOPCOMPONENT dans le métamodèle Loop doit être crée pour chacun des LOOPCOMPONENT du métamodèle Polyhedron présent dans le modèle source. On utilise donc l'objet fabrique (Factory) fourni par EMF.

`process` est appelé pour chacun des couples (élément source, élément cible) obtenus. Cette méthode permet de remplir, si nécessaire via des calculs, les attributs de l'élément créé. Dans le cas de la création d'un LOOPCOMPONENT dans le métamodèle Loop, deux opérations sont à réaliser : donner au composant crée le nom du composant source, et transformer le polyèdre en boucle (cela est fait par l'appel à la méthode externe `Polyhedron2LoopStatement.convert(dpm.getPolyhedron())`.

`processReference` permet de résoudre les références dans les modèles. Elle est appelée pour chaque élément créé. Les références à résoudre dans la règle illustrée sont les références de la boucle pour connaître à quelle répétition de processeur elle correspond. Cela est fait à l'aide de la table renvoyée par la méthode `getGlobalRefs()` qui contient les correspondances entre éléments sources et éléments cibles. Dans la règle illustrée ici, la résolution de référence consiste à faire pointer les LOOPCOMPONENTS crées vers les processeurs sur

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009


```
1  public class PolyhedronLoopComponent2LoopLoopComponent extends Rule{

       public PolyhedronLoopComponent2LoopLoopComponent()
       {
5      super();
       setQueryFromContext(true);
       addRule(DeployedloopPackage.eINSTANCE.getApplicationComponent_Port(),
             new PolyhedronInputPort2LoopInputPort());
       addRule(DeployedloopPackage.eINSTANCE.getApplicationComponent_Port(),
10           new PolyhedronOutputPort2LoopOutputPort());
       addRule(DeployedloopPackage.eINSTANCE.getApplicationComponent_Port(),
             new PolyhedronInputOutputPort2LoopInputOutputPort());
       addRule(DeployedloopPackage.eINSTANCE.getLoopComponent_Connectors(),
             new PolyhedronApplicationTiler2LoopApplicationTiler());
15     addRule(DeployedloopPackage.eINSTANCE.getComponent_Structure(),
             new PolyhedronCompound2LoopCompound());
       }

       protected EObject create(EObject srcElement)
20     {
       return DeployedloopFactory.eINSTANCE.createLoopComponent();
       }


25     protected EObjectCondition getCondition(EObject srcElementContext)
       {
       return new EObjectTypeRelationCondition(
                Deployedpolyhedron Package.eINSTANCE.getLoopComponent());
       }
30
       protected void process(EObject srcElement, EObject tgtElement)
       {
       LoopComponent dpm=(LoopComponent) srcElement;
        fr.lifl.west.gaspard2.metamodel.deployedloop.LoopComponent smlm=
35      ( fr.lifl.west.gaspard2.metamodel.deployedloop.LoopComponent) tgtElement;
       smlm.setName(dpm.getName());
       smlm.setLoop(Polyhedron2LoopStatement.convert(dpm.getPolyhedron()));
       }

40     protected void processReferences(EObject srcElement, EObject tgtElement)
       {
       LoopComponent dpm=(LoopComponent) srcElement;
       fr.lifl.west.gaspard2.metamodel.deployedloop.LoopComponent smlm=
         ( fr.lifl.west.gaspard2.metamodel.deployedloop.LoopComponent) tgtElement;
45
       List parameterBaseList=dpm.getPolyhedron().getParameterBase();
       for (Object o:parameterBaseList)
          smlm.getLoop().getParametersBase().add(getTransformation()
             .getGlobalRefs().get((EObject) o));
50     }
   }
```

FIG. 6.14: Une règle de transformation en MoMoTE.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

lesquels ils doivent s'exécuter. L'architecture étant transformée par une autre règle, on ne peut faire pointer les boucles sur les processeurs que lorsque ceux-ci auront été créés.

processQueryResult n'est pas utilisé dans la règle de la figure 6.14. Lors de l'exécution d'une règle, MoMoTE va placer dans une liste tous les éléments du modèle respectant la condition. Puis, le moteur de MoMoTE va prendre chacun des éléments de la liste (dans l'ordre) et exécuter les autres méthodes de la règle pour chacun de ces éléments. Dans certaines règles de transformation, il peut être important de pouvoir fixer l'ordre dans lequel les éléments doivent être transformés. C'est pourquoi cette méthode existe. Appelée avant de commencer l'exécution des règles sur les éléments, elle permet à l'utilisateur de définir un ordre des éléments dans la liste afin que la règle transforme dans un ordre précis les éléments. Cette méthode est notamment utilisée afin de calculer un ordonnancement des tâches entre le métamodèle Loop et le métamodèle de langage : cette transformation sera illustrée dans le paragraphe suivant.

6.5.1.2 Exécution des règles

L'exécution des règles de transformation est faite par le moteur de transformation de MoMoTE. Afin de bien comprendre comment cela fonctionne, l'algorithme d'exécution des règles va être présenté ici.

```
1  transformation(){
   Initialisation des regles

   Pour chaque regle r dans la liste des regles
5       //Execution de la requete sur le modele en utilisant la methode
        //getCondition()
        queryResult=rule.executeQuery()
        orderedQueryResult=rule.processQueryResult(queryResult)

10      Pour chaque element srcElt de la liste orderedQueryResult
            tgtElt=r.create(srcElt) // creation de l'element cible
            Ajout de l'element dans le model cible
            Ajout du couple (src,tgtElt) dans la table de reference global
            rule.process(srcElt,tgtElt)
15          Execution des sous regles selon le meme principe
        FinPour
   FinPour

   Execution de processReference pour chacune des regles avec chacun des
20 couples

   }
```

FIG. 6.15: Pseudo-code du moteur de MoMoTE pour l'exécution des règles.

La figure 6.15 illustre le fonctionnement simplifié du moteur de transformation de MoMoTE. Après une initialisation des règles qui comprend notamment l'instanciation des sous-règles pour chacune des règles. Le moteur exécute toutes les règles d'un même niveau.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

L'exécution d'une règle se déroule comme suit : la première étape est de faire une requête sur le modèle source afin d'obtenir tous les éléments du modèle respectant la condition donnée par la méthode `getCondition()`, cela renvoie une liste d'éléments valides. Ensuite, le moteur offre la possibilité à la règle d'ordonnancer les différents éléments avec la méthode `processQueryResult()` qui, si elle n'est pas implémentée par la règle, ne fait rien et renvoie la même collection que celle qu'on lui a fournie. Ensuite, pour chacun des éléments respectant la condition, l'élément cible est crée (avec la méthode `create()`), ajouté au modèle cible et à la table globale des références et la méthode `process()` est appelée afin de réaliser les calculs nécessaires et de remplir les attributs de l'élément crée. Les sous-règles de la règle sont exécutées en respectant le même principe pour chacun des éléments créés. Le moteur réalise donc un parcours en profondeur de la hiérarchie.

Une fois que toutes les règles ont été exécutées une fois, il reste à résoudre les références (recréer les liens entre des éléments créés par des règles différentes). Pour cela, la méthode `processReference()` est appelée sur chacune des règles et pour chacun des couples (élément source, élément cible) que la règle a créé.

Cette description du cœur du moteur de transformation MoMoTE permet d'illustrer qu'il y a un ordre d'exécution entre les règles et entre les éléments gérés par une même règle et que cet ordre peut être important selon les calculs réalisés lors de l'exécution de la méthode `process()` notamment.

6.5.2 Implémentation de la transformation métamodèle Loop vers métamodèle de langage procédural

Les règles de transformations présentées classiquement sont des cas d'étude simples tel que la transformation *Classe vers Table*. Ces exemples présentent des transformations d'un (de) concept(s) vers un (ou des) concept(s) ne nécessitant pas le contexte de la transformation.

La réalisation de règles de transformations complexes et contextuelles n'est pas présentée et cela provoque que l'on ne sait pas ce qui peut être réalisé ou non via les règles de transformation et par quels types de règles de transformation de modèle. En nous basant sur la description de l'implémentation de la règle transformant un modèle dans le métamodèleLoop vers le métamodèle de langage procédural, nous discuterons de ce qui a permis son implémentation en MoMoTE et qui n'est pas forcément réalisable en utilisant un autre moteur de transformation de modèle.

6.5.2.1 Implémentation de la transformation des composants

Les transformations des deux types de composants du métamodèle Loop (les LOOPCOMPONENTS ou les ELEMENTARYTASKS) ont des points communs. En effet, une instance de composant est transformée dans une structure de code (STATEMENT_LIST) qui va contenir le code correspondant au composant instancié. Lors de ces transformations, il est nécessaire de déterminer localement pour chaque instance si certains de ces ports doivent être déclarés en variables ainsi que de calculer les courts-circuits sur les éventuels tilers. Tous ces calculs ne peuvent pas être réalisés pour une instance sans connaître le contexte dans lequel celle-ci se trouve. En effet, afin de calculer le court-circuit d'un tiler, il faut connaître le court-circuit calculé précédemment en fonction de la hiérarchie dans laquelle est notre instance. De même, la déclaration de variables dépend de quelles variables ont été déclarées précédemment et dans le cas de réutilisation de variables, la connaissance qu'une variable est utilisée ou non

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

dépend de l'application globale. La connaissance du contexte dans lequel est l'instance que l'on traite est donc primordiale dans cette transformation.

Les transformations des différents types de composants ayant des points communs, il semble logique pour tout développeur de factoriser le code de ces transformations. Une règle de transformation en MoMoTE étant une classe Java, il est possible d'utiliser le mécanisme d'héritage de ce langage. Ainsi, une règle abstraite contenant les différentes fonctionnalités nécessaires aux transformations de composants est définie (il s'agit de la règle `ApplicationComponent2Statementlist`). Afin d'avoir une connaissance du contexte dans lequel une règle traitant une instance de composant est appelée, celle-ci contient un certain nombre d'attributs. Parmi ces attributs, on trouve des tables de hachages permettant d'associer les ports à des variables, de connaître les courts-circuits calculés et sur quels ports ceux-ci sont connectés, mais également une référence vers la règle mère de cette règle (une sous règle connaît ainsi quelle est la règle qui l'a créé). Cette référence vers la règle mère est l'un des points essentiels de la connaissance du contexte de la transformation. Grâce à celle-ci la règle peut aller récupérer les informations de contexte du composant hiérarchiquement supérieur afin de connaître de quelle manière les ports de celui-ci ont été affectés aux variables ainsi que de retrouver les courts-circuits qui ont été calculés par ce composant vers les ports de l'instance que la règle traite.

La réalisation de ce type de transformation est possible dans le cas où le moteur de transformation parcourt le modèle en profondeur et qu'il est possible de guider, d'ordonnancer le parcours en largeur. Cela est le cas de MoMoTE.

La figure 6.16 illustre la déclaration de la règle `ApplicationComponent2Statementlist` et les méthodes privées que celle-ci définit. La règle définit une méthode prévue pour gérer les déclarations de ports en variables selon le contexte (`variablesDeclaration()`) ainsi que les fonctions permettant la gestion des tilers (calculant le court-circuit ou générant les structurant permettant le calcul de ceux-ci).

La figure 6.17 illustre l'utilisation de la règle `ApplicationComponent2StatementList` lors de la réalisation de la règle traitant les ELEMENTARYTASKS (règle `ElementaryTask2StatementList`) et l'utilisation des méthodes définies dans celle-ci lors l'implémentation de la méthode `process()` d'une règle de transformation MoMoTE. La transformation d'une tâche élémentaire consiste donc à créer une structure STATEMENT_LIST. Dans cette structure, après la gestion des variables à déclarer et la gestion des tilers en entrée, il faut introduire un appel à la routine sur laquelle est déployée la tâche élémentaire (lignes 14 à 21). Il faut ensuite gérer les tilers de sortie si ceux-ci n'ont pu être gérés par une optimisation auparavant.

6.5.2.2 L'implémentation est-elle possible avec n'importe quel moteur de transformation ?

Après avoir étudié comment la transformation est réalisée, il est intéressant de se demander si celle-ci aurait pu être réalisée dans un autre moteur de transformation.

Comme nous l'avons vu dans la section précédente, la transformation du métamodèle Loop vers le métamodèle de langage procédural a pu être possible grâce à la flexibilité de MoMoTE basée sur le langage Java ainsi qu'à la possibilité d'indiquer au moteur de transformation un ordre d'exécution des règles. Se baser sur un langage tel que Java permet de bénéficier de toute la puissance de celui-ci. Une grande partie des moteurs de transformations

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009


```
1   public abstract class ApplicationComponent2StatementList extends Rule {

      //reference vers la regle mere
      protected ApplicationComponent2StatementList ownerRule;
5     protected Map inputTilerMap;//key = tgt portInstance, value=tiler
      protected Map outputTilerMap;//key = src portInstance, value = List of tiler
      //...

      //methode gerant la declaration des variables
10    public void variablesDeclaration(Statement_list sL)
      {
        ..
      }

15    //gestion des tilers en entree
      public void gestionTilerIn(EObject tgtElement)
      {
        ..
      }
20
      //gestion des tilers en sortie dans le cas d'optimisation possible
      public void gestionTilerOutBefore(EObject tgtElement)
      {
        ..
25    }

      //gestion des tilers en sortie dans le cas classique
      public void gestionTilerOut(EObject tgtElement)
      {
30      ..
      }

    }
```

FIG. 6.16: Déclaration de la règle abstraite de gestion des composants.

n'ont pas cette flexibilité puisque ceux-ci sont basés sur leur propre langage spécialisé dans l'expression de règles et non dans la programmation généraliste. Cette flexibilité a notamment permis d'obtenir des informations sur le contexte de la transformation à l'aide d'attributs dans la règle tel que l'objet de la règle mère. De plus, ayant développé le moteur de transformation, nous avons pu l'adapter à nos besoins notamment en permettant de spécifier l'ordre d'exécution des règles de transformation (à l'aide de la méthode `processQueryResult()`).

L'implémentation d'une telle transformation dans un moteur de transformation est conditionnée par ce que celui-ci permet. Un moteur de transformation rigide, basé sur des règles de transformations transformant un concept en un autre sans offrir possibilité de communiquer avec le moteur et d'avoir des informations du contexte, ne semble pas adapté à ce type de transformation. L'utilisation d'un moteur souple avec lequel on peut interagir via des surcharges de méthodes et la flexibilité des règles semble indispensable pour réaliser ce type de transformation. L'appel de code externe et l'expression de boites noires sont également

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

```
public ElementaryTask2StatementList
       extends ApplicationComponent2StatementList{

  ..

  public void process(EObject srcElt,EObject tgtElt)
  {
    //Declaration des variables necessaire
    variablesDeclaration((Statement_list) tgtElt);

    //Gestion des tilers en entree
    gestionTilerIn(tgtElt);

    //Optimisation des tilers en sortie si possible
    gestionTilerOutBefore(tgtElt);

    //Generation de l'appel a la tache elementaire
    ElementaryTask eT=(ElementaryTask) srcElt;
    Statement_list sL=(Statement_list) tgtElt;
    CallStatement cS=OmpprocedurallanguageFactory
                   .eINSTANCE.createCallStatement();

    //Gestion du nom de la fonction et des bibliotheques
    //Gestion des parametres de la routine
    //Ajout de l'appel de la routine a la structure
    //cree pour la tache elementaire
    sL.getStatements().add(cS);

    //Gestion des tilers en sortie si ils n'ont pas ete optimise
    gestionTilerOut(tgtElt);
  }

}
```

FIG. 6.17: Méthode `process()` de la règle transformant une tâche élémentaire en structure permettant l'appel de celle-ci.

indispensables.

6.6 Génération de code à partir de modèles et résultats d'exécution des codes générés

Le but de cette partie est d'illustrer la transformation complète d'un modèle Gaspard vers du code parallèle ainsi que présenter les résultats de différents tests effectués afin d'en déduire une méthodologie d'utilisation de l'environnement Gaspard dans le cadre du calcul haute performance.

La chaîne ciblant OpenMP dans l'environnement Gaspard étant développée afin de cibler les applications de calcul haute performance, nous avons testé celle-ci sur une opération

Thèse de Julien Taillard, Lille 1, 2009

classique dans les applications de calcul scientifique, à savoir la multiplication de matrice.

Les fonctions « classiques » de calcul scientifique ont été très largement étudiées et sont regroupées sous forme de bibliothèque de fonctions dans des bibliothèques BLAS (Basic Linear Algebra Subprograms) ayant une interface standard pour permettre de changer facilement d'implémentation selon la machine utilisée. Ces fonctions de bases sur les vecteurs et les matrices sont étudiées depuis une trentaine d'année [85, 86, 46, 45]. Ces fonctions sont regroupées par niveau de complexité. La multiplication de matrice fait partie des fonctions de niveau 3 (sur 3). Il existe un grand nombre d'implémentations de la bibliothèque BLAS. Celles-ci peuvent être libre telles que ATLAS [42] ou GoToBLAS [63] ou propriétaires (ESSL d'IBM ou MKL d'Intel). Ces bibliothèques sont plus ou moins optimisées selon les architectures sur lesquelles elles sont utilisées. Des travaux de parallélisations de multiplications de matrices avec OpenMP ont également été menés. Dans [17], Bentz et *al.* compare un algorithme de multiplication de matrices par bloc utilisant la fonction DGEMM des différentes implémentations disponibles et obtiennent de bonnes accélérations.

6.6.1 Génération de code à partir d'un modèle

Nous allons illustrer la génération de code sur l'algorithme de multiplication de matrice ligne colonne distribué sur quatre processeurs. Nous allons tout d'abord présenter le modèle UML, puis ensuite illustrer ce qu'il se passe dans l'environnement Gaspard et enfin détailler le code généré par cet exemple.

6.6.1.1 Modèle UML de la multiplication de matrice

La figure 6.18 présente le modèle UML de la multiplication de matrice placée sur quatre processeurs. Au centre de la figure se situe le composant d'architecture contenant une répétition de quatre éléments de calcul (*ProcessingUnit*), c'est sur ces éléments que seront placées les tâches applicatives. Le composant applicatif situé en haut de la figure est le composant principal de l'application, le *main*. Celui-ci est composé par quatre instances : deux d'entre elles réalisent l'initialisation des matrices puis vient la réalisation de la multiplication de matrice, et enfin une instance de composant vérifie le résultat du calcul effectué. Le composant applicatif en bas de la figure modélise l'algorithme de multiplication de matrice ligne colonne sous une forme de parallélisme de données : l'opération du produit scalaire est répétée (2000,2000) fois. Les liens entre les instances d'applications et l'instance de *ProcessingUnit* sont l'association de l'application sur l'architecture. Les tâches non répétées (les instances iM et cM) sont placées sur un seul processeur, mais il est nécessaire d'effectuer une distribution afin de spécifier sur quel processeur chacune d'entre elles sera exécutée. La distribution des répétitions de produits scalaires est effectuée par blocs : des blocs de calcul de taille(2000,500) devront être effectués par chacun des processeurs. Enfin, l'instance du composant vérifiant la matrice est placée sur un processeur.

6.6.1.2 Vue de l'environnement Gaspard

Une fois le modèle UML effectué dans l'outil de modélisation UML (dans notre cas MagicDraw UML), nous allons transformer ce modèle vers du code à l'aide de l'environnement Gaspard. Gaspard étant développé comme un plugin Eclipse, il faut être capable de lire le modèle UML dans Eclipse à l'aide des outils fournis par EMF. Pour cela, la première étape consiste à exporter le modèle UML de MagicDraw vers un modèle UML de EMF (il

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

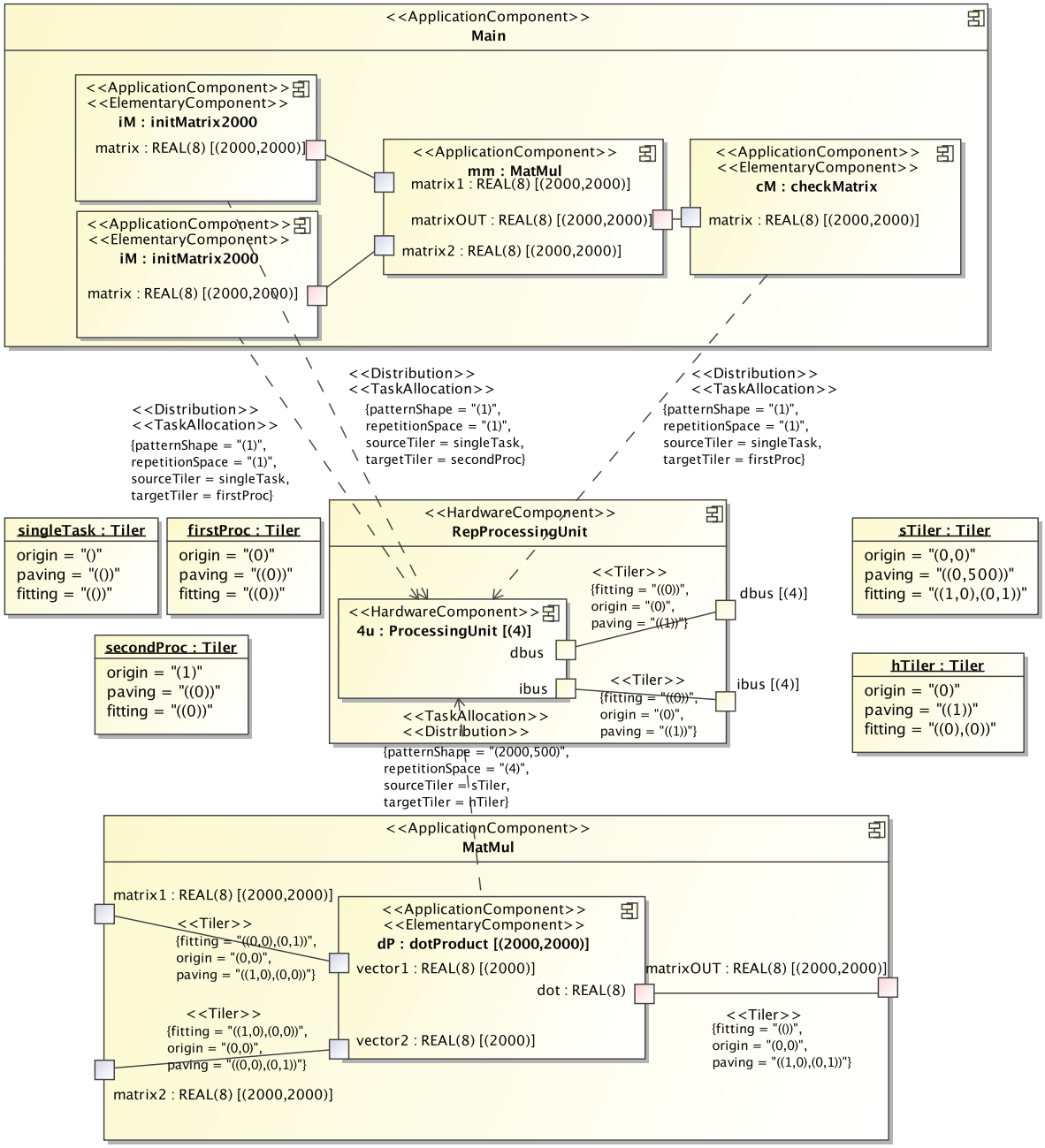


FIG. 6.18: Modélisation de la multiplication de matrice placé sur quatre processeurs.

s'agit d'une transformation de modèle du métamodèle UML de MagicDraw vers le métamodèle UML d'EMF) qui sera notre point de départ dans l'environnement Gaspard. À partir de ce modèle UML dans EMF, les chaînes de compilations de Gaspard peuvent être exécutées. Celles-ci vont produire tous les modèles intermédiaires avant de réaliser la génération de code. L'environnement Gaspard est illustré dans la figure 6.19. On peut y distinguer le modèle UML de départ puis les différents modèles intermédiaires générés avant la génération de code C et Fortran. Dans cet exemple, il est possible de générer les deux codes à partir du

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

même modèle puisqu'il est possible d'appeler des fonctions Fortran en C et inversement.

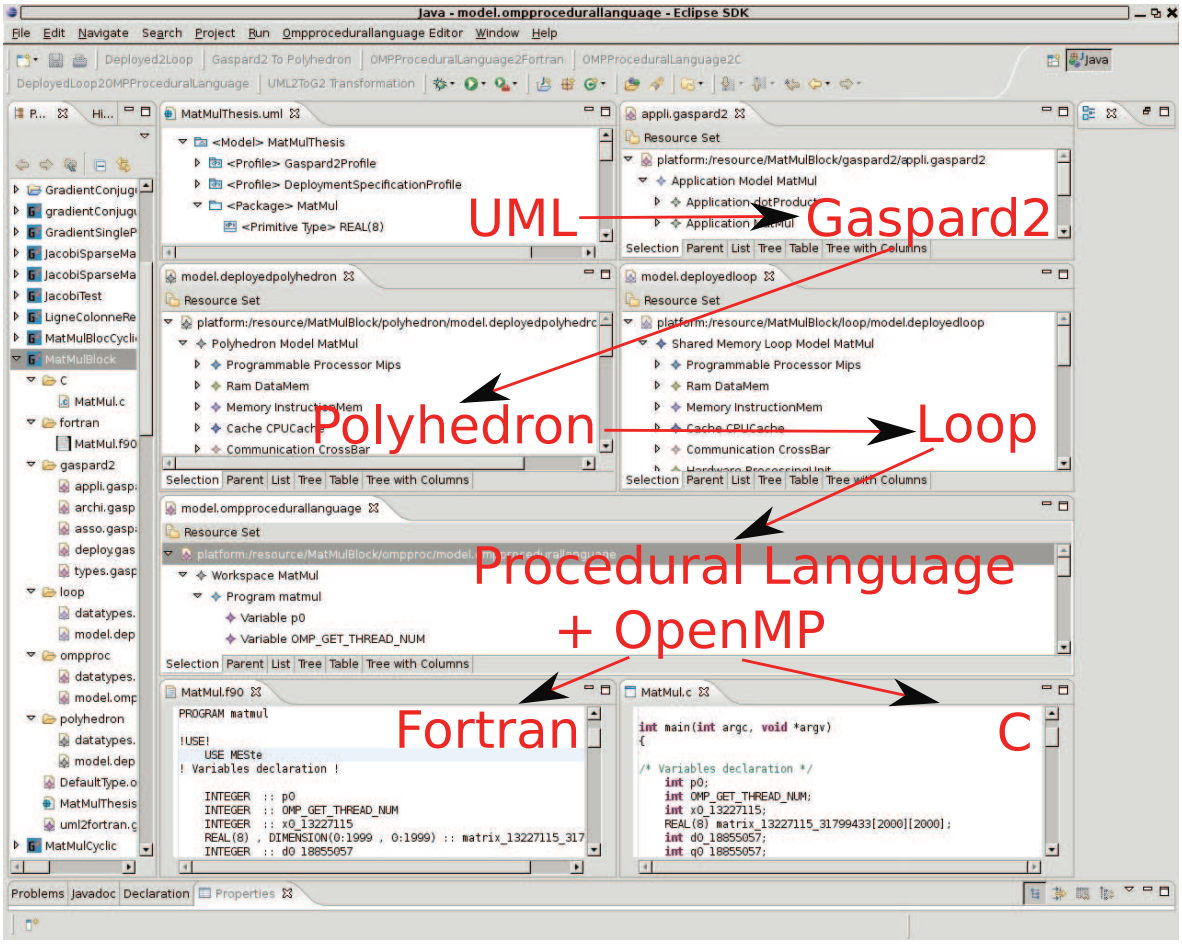


FIG. 6.19: Transformation dans l'environnement Gaspard.

6.6.1.3 Code généré pour cet exemple

La figure 6.20 présente la structure du code généré. Le code est simplifié afin d'être lisible. Dans ce code on peut noter la présence de la directive OpenMP !$OMP PARALLEL qui indique que le code interne est parallèle. Ensuite, on peut y retrouver la structure du composant principal illustré précédemment. Les deux premiers appels de routine permettent d'initialiser la matrice, ensuite sont réalisés les produits scalaires (avec la routine *dotProduct*). Les boucles entourant cette routine sont générées automatiquement à partir de la distribution. Enfin, la vérification est effectuée. Dans ce code, on peut distinguer les informations de placements provenant des boucles générées automatiquement à partir des informations de distribution exprimées en UML. Les deux premières tâches sont chacune placées sur deux threads différents c'est pourquoi on retrouve les deux constructions conditionnelles Si (dépendant de p_0). Les boucles générées autour de la boucle du produit scalaire exprime le placement par blocs (x_0 et/ou x_1 sont exprimés en fonction de p_0). Puis la routine vérifiant la matrice est exécutée sur le thread 0.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

```
PROGRAM matmul

!USE!
     USE MESte
! Variables declaration !

     INTEGER :: p0
     INTEGER :: OMP_GET_THREAD_NUM
     REAL(8) , DIMENSION(0:1999 , 0:1999) :: matrix_13227115_31799433
     REAL(8) , DIMENSION(0:1999 , 0:1999) :: matrix_13227115_17840791
     REAL(8) , DIMENSION(0:1999 , 0:1999) :: matrixOUT_13227115_8888398
     //Toutes les autres variables sont egalement declare ici
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(matrix_13227115_31799433 ,matrix_13227115_17840791 ,matrixOUT_13227115_8888398)

          p0=omp_get_thread_num()
          x0_13227115= 0
      IF ( p0 == 1 ) THEN
          d0_18855057= 0
          q0_18855057= 0
          CALL initMatrix2000(matrix_13227115_31799433)
      END IF

      IF ( p0 == 0 ) THEN
          q0_13150556= 0
          d0_13150556= 0
          CALL initMatrix2000(matrix_13227115_17840791)
      END IF
      !$OMP BARRIER

      DO d1_18722457=0,499

          x1_18722457=d1_18722457+500*p0
          DO d0_18722457=0,1999

          x0_18722457= d0_18722457
          q0_18722457= p0
          //Calcul des tilers ou sont calculer les valeurs de originPattern
     CALL dotProduct(matrix_13227115_17840791(originPattern_11210989(0):originPattern_11210989(0)+1999,
                                 originPattern_11210989(1)),
               matrix_13227115_31799433(originPattern_19291211(0),
                                 originPattern_19291211(1):originPattern_19291211(1)+1999),
               matrixOUT_13227115_8888398(originPattern_27181780(0),originPattern_27181780(1)))

          END DO
     END DO

     !$OMP BARRIER
      IF ( p0 == 0 ) THEN
          q0_2984495= 0
          d0_2984495= 0
          CALL checkMatrix(matrixOUT_13227115_8888398)
      END IF

      !$OMP BARRIER
!$OMP END PARALLEL
END PROGRAM matmul
```

FIG. 6.20: Structure du code généré pour la multiplication de matrice.

6.6.2 Machines de tests

Les résultats présentés dans cette thèse ont été réalisés sur plusieurs machines ayant des architectures différentes. Afin de permettre une meilleure compréhension des résultats, nous allons ici présenter brièvement l'architecture des machines de tests (de l'architecture la plus classique à l'architecture spécialisée pour le calcul).

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

Goelo est une machine de travail de bureau classique ayant une architecture PC. Cette machine est équipée de deux processeurs Intel Xeon composés chacun de deux cœurs de calcul cadencés à 3Ghz se partageant 2Go de mémoire.

Cleep est un noeud d'un cluster de calcul. Celui-ci est composé de quatre processeurs ayant chacun 2 cœurs de calcul cadencés à 2Ghz. Ces processeurs se partagent 48Go de mémoire via une architecture PC.

Decrypthon1 est un noeud du supercalculateur de l'université de Lille 1. Il s'agit d'une machine P575 d'IBM composée de huit processeurs Power5 cadencés à 1.9Ghz se partageant 16Go de mémoire. Il s'agit d'une machine spécialisée pour les calculs parallèles disposant d'un réseau d'interconnexion spécifique d'accès à la mémoire.

6.6.3 Test de différents algorithmes de multiplication de matrices

Afin de pouvoir en déduire comment utiliser l'environnement Gaspard dans le cadre du calcul haute performance, nous avons comparé différents codes de multiplications de matrices avec une implémentation optimisée. Nous avons choisi GotoBLAS qui propose une version séquentielle et une version parallèle de l'algorithme.

Les différents algorithmes dont le code est généré sont les suivants :

- Algorithme ligne x colonne ;
- Algorithme de multiplication par blocs allant jusqu'aux opérations scalaires ;
- Algorithme de multiplication par blocs utilisant la fonction GotoBLAS dès que le code est sur un thread.

Ces algorithmes ont été choisis afin de vérifier si leurs propriétés sont respectées. En effet, l'algorithme de base est l'algorithme de multiplication ligne colonne. Celui-ci est connu pour ne pas utiliser très bien les mémoires caches. L'algorithme de multiplication par blocs est plus évolué puisque sa structure doit permettre une meilleure utilisation des mémoires caches. Enfin, il est probable que l'environnement Gaspard ne puisse pas gérer les optimisations de code séquentiel, c'est pourquoi nous avons également modélisé l'algorithme de multiplication par blocs utilisant la tâche GotoBLAS pour les parties de code séquentielles.

Les résultats présentés ici ont été réalisés sur la machine de bureau Goelo. Les temps d'exécutions sont fournis en secondes. À l'exception du code séquentiel, les codes sont exécutés sur quatre threads afin d'utiliser tous les cœurs de calculs présents sur la machine. Le même comportement a été observé sur les autres machines.

Les temps d'exécutions de référence de la fonction GotoBLAS en séquentiel et en parallèle sur des tableaux de taille {2000,2000} sont données dans le tableau 6.1. Ils sont d'environ 3 secondes en séquentiel et d'environ 1 seconde en parallèle.

Algorithme	Temps d'exécution moyen
GotoBLAS séquentiel	0 :03.39
GotoBLAS parallèle	0 :01.05

TAB. 6.1: Temps de référence d'exécution des fonctions GotoBLAS en séquentiel et en parallèle sur quatre threads sur des matrices carrées.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Le tableau 6.2 présente les temps d'exécutions des différents algorithmes sur des matrices carrées {2000,2000}. Comme prévu lors de sa modélisation, le code de multiplication ligne par colonne donne de mauvais temps d'exécution puisque celui-ci met 21 secondes à s'exécuter. Le code de multiplication par blocs allant jusqu'aux opérations scalaires donnent de meilleures performances puisqu'il met 12s à s'exécuter. Cela est probablement dû aux nombres de défauts de caches réduits par rapport à l'algorithme précédent. Il est à noter que ces deux algorithmes sont mauvais puisque bien plus lent que le temps d'exécution de la tâche optimisée en séquentiel. L'algorithme de multiplication de matrices par blocs utilisant des fonctions GotoBLAS est lui bien meilleur que les autres codes générés. En effet son temps d'exécution est de 1,3 seconde, ce qui est plus rapide que le code optimisé séquentiel et très proche du code optimisé en parallèle.

Algorithme	Temps d'exécution moyen
Ligne x colonne	0 :21.60
Multiplication par blocs	0 :13.17
Multiplication par blocs utilisant des tâches GotoBLAS	0 :01.26

TAB. 6.2: Matrices carrés (2000x2000) - Temps d'exécution sur quatre processeurs - Moyenne réalisée sur 100 exécutions.

Le comportement des algorithmes sur des matrices rectangles de taille {3000,2000}×{4000,3000} est également observé. Les tableaux 6.3 et 6.4 donnent respectivement les temps d'exécutions de référence de la fonction GotoBLAS et des codes générés par Gaspard pour les matrices rectangulaires. On peut y observer le même comportement que pour les matrices carrées.

Algorithme	Temps d'exécution moyen
GotoBLAS séquentiel	0 :10.03
GotoBLAS parallèle	0 :02.96

TAB. 6.3: Temps de référence d'exécution des fonctions GotoBLAS en séquentiel et en parallèle sur quatre threads sur des matrices rectangulaires.

Algorithme	Temps d'exécution moyen
Ligne x colonne	1 :57.87
Multiplication par blocs	0 :33.52
Multiplication par blocs utilisant des tâches GotoBLAS	0 :04.04

TAB. 6.4: Matrices rectangulaires - Temps d'exécution sur quatre processeurs - Moyenne réalisée sur 100 exécutions.

Certaines conclusions peuvent être extraites de ces résultats. La première est que les algorithmes gardent les propriétés connues sur ceux-ci lors de la génération de code. Le code généré à partir du modèle d'un algorithme connu pour ne pas bien utiliser les mémoires caches aura également cette propriété et aura ainsi des temps d'exécutions mauvais. La

Thèse de Julien Taillard, Lille 1, 2009

seconde conclusion que l'on peut tirer de ces premiers tests est que, lors de la génération, l'environnement Gaspard n'est actuellement pas capable d'optimiser les parties de code séquentielles. C'est pourquoi il n'est pas nécessaire de modéliser trop finement les applications (comme aller jusqu'aux opérations scalaires) mais utiliser des tâches séquentielles optimisées dès que l'on sait que le niveau de parallélisme que l'on veut utiliser est atteint.

6.6.4 Effet de la distribution sur les performances

La distribution des tâches sur les threads est un point important dans le but d'obtenir des performances. Afin de tester l'influence de celle-ci, nous avons distribué la même application différemment. L'algorithme utilisé pour ce test est l'algorithme de multiplication de matrice ligne colonne utilisant la tâche élémentaire de produit scalaire. La modélisation est réalisée avec des matrices carrées de tailles 2000x2000. Cet algorithme a été choisi, car il est constitué d'un grand nombre de tâches à distribuer (4 millions de produits scalaires), et que son temps d'exécution relativement long permet d'observer plus facilement les effets de la distribution sur ce temps d'exécution.

Les codes générés seront exécutés sur les différentes cibles d'exécutions. Cela va permettre d'observer l'impact du réseau d'interconnexion à la mémoire dans l'exécution.

Les différents placements testés sont les placements classiques que l'on pouvait trouvé en HPF :

- bloc ;
- cyclique ;
- bloc-cyclique ;
- cyclique-cyclique.

Le placement par bloc consiste à faire effectuer par un processeur un bloc de calcul. Dans le cadre de la multiplication de matrice, cela revient à découper l'espace de répétition du produit scalaire en quatre parties égales et à faire exécuter chaque partie par un processeur. C'est le placement illustré dans l'exemple de la figure 6.18. Dans un programme HPF, ce type de distribution se serait exprimé de la manière suivante : `!$HPF DISTRIBUTE(BLOCK)` . La figure 6.21 illustre cette distribution qui répartit des blocs de calcul de taille (2000,500) sur chacun des processeurs.

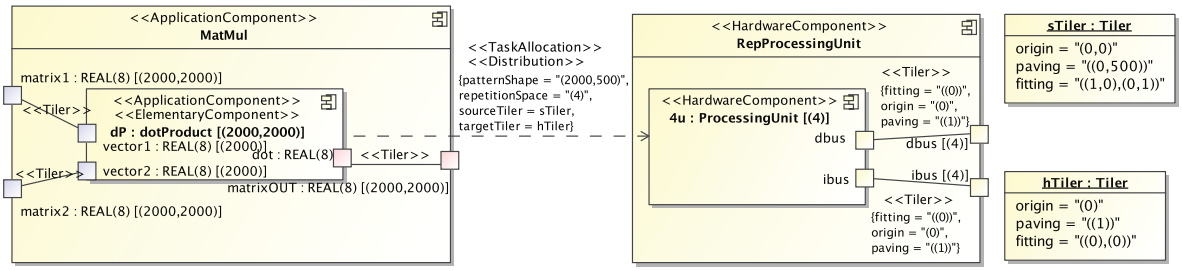


FIG. 6.21: Distribution par bloc.

Le placement cyclique consiste à répartir le calcul d'une manière cyclique. Pour la multiplication de matrice, cela consiste à prendre une colonne de calcul à le placer sur un processeur puis à prendre une seconde colonne de calcul est à la placer sur le processeur

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

suivant et ainsi de suite. Le placement HPF serait exprimé de la façon suivante : `!$HPF DISTRIBUTE(CYCLIC)` . La figure 6.22 représente la distribution d'une colonne de calcul sur les processeurs cycliquement.

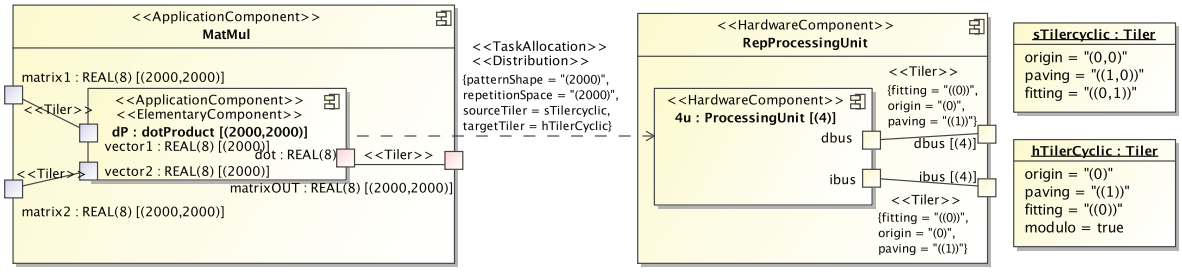


FIG. 6.22: Distribution cyclique.

Le placement bloc-cyclique consiste à répartir des blocs de calcul de manière cyclique. Une telle distribution exprimée en HPF de la façon suivante : `!$HPF DISTRIBUTE(BLOCK,CYCLIC)` est modélisée dans la figure 6.23. Cette distribution va placer cycliquement sur les processeurs des blocs de calcul de taille (2000,50).

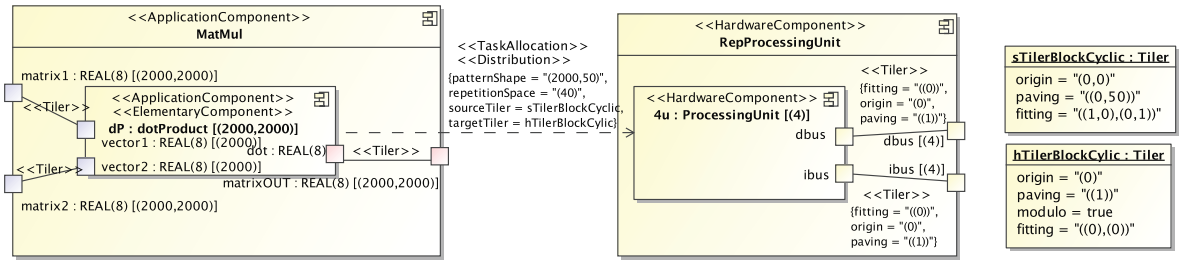


FIG. 6.23: Distribution cyclique par blocs.

Le placement cyclique-cyclique consiste à répartir les calculs de manière cyclique dans les deux dimensions de l'espace de répétition. La distribution présentée à la figure 6.24 réalise un tel placement. En effet, la distribution sélectionne des motifs de calcul de taille (4) dans les calculs et place chacun de ces calculs sur des processeurs différents. Le pavage du tiler *hTilerCyclicCyclic* qui est de valeur ((4),(1)) exprime un décalage de un sur les processeurs quand l'index dans l'espace de répétition augmente de un dans la seconde dimension. Le placement HPF serait exprimé de la façon suivante : `!$HPF DISTRIBUTE(CYCLIC,CYCLIC)` .

La figure 6.25 présente les résultats d'exécution de ces différents placements sur les différentes machines d'exécutions. Dans ces résultats, les codes sont exécutés sur quatre threads. On peut noter que les distributions par blocs (que ce soit bloc ou bloc-cyclique) ont les meilleurs résultats sur toutes les machines d'exécutions. Dans la mesure où ces placements permettent d'utiliser au maximum les données présentes dans la mémoire cache des processeurs, ce sont les résultats qui étaient attendus. Les distributions cycliques sont moins bonnes sur toutes les machines. En effet, ce type de distribution va imposer à chaque processeur de changer les données sur lesquelles il travaille très régulièrement. Il y a donc

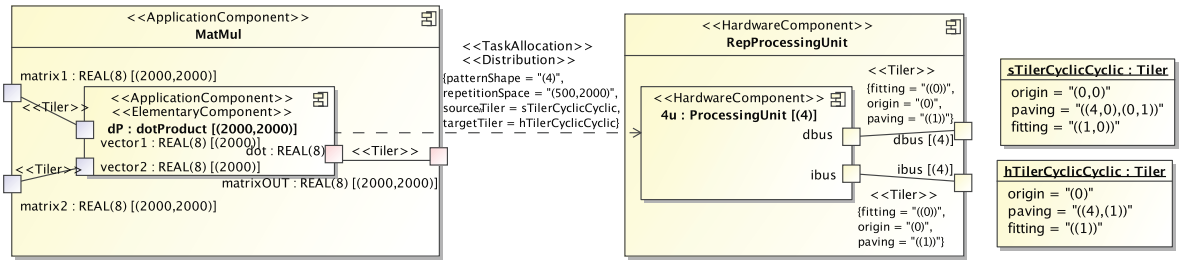


FIG. 6.24: Distribution cyclique cyclique.

un grand nombre de défauts de caches qui peuvent entraîner des contentions sur le réseau d'interconnexion à la mémoire. On peut d'ailleurs constater que celui-ci joue un rôle important dans les temps d'exécutions. En effet, alors que sur Goelo et Cleep (machines à architecture PC) les temps d'exécution passent du simple au double entre une distribution par blocs et une distribution cyclique, le temps d'exécution n'augmente que très légèrement sur le supercalculateur équipé d'un réseau d'interconnexion spécialisé.

	Bloc	Bloc cyclique	Cyclique	Cyclique Cyclique
Goelo	36.01	36.00	78.57	77.80
Cleep	25.16	25.23	48.34	48.35
Decrypthon1	37.66	37.74	40.53	40.32

FIG. 6.25: Test de différents placements sur les différentes architectures de test - Exécution sur 4 threads - Moyenne sur 10 exécutions.

Les résultats présentés dans cette section illustrent l'importance de la distribution pour obtenir de la performance et cela est principalement vrai sur les architectures PC classiques qui n'ont pas d'accès parallèles à la mémoire.

6.6.5 Méthodologie d'utilisation de Gaspard et de modélisation des applications afin d'obtenir des performances

Des différents résultats présentés précédemment on peut extraire des généralités permettant de dégager une méthodologie d'utilisation de l'environnement Gaspard dans le cadre du calcul haute performance.

Comme les résultats des tests sur les différents algorithmes de multiplication de matrice le montrent, les codes générés par l'environnement Gaspard quand un modèle est modélisé très finement (jusqu'aux opérations scalaires) ne sont pas compétitifs par rapport aux codes optimisés manuellement. Il est en effet difficile d'optimiser les parties séquentielles d'un code automatiquement généré alors que cela dépend fortement de l'architecture.

La méthodologie générale d'utilisation de l'environnement Gaspard dans le but d'obtenir des performances devrait alors être la suivante : laisser l'environnement Gaspard gérer le parallélisme en utilisant des tâches élémentaires séquentielles optimisées dès que le grain de modélisation le permet.

En plus de cette méthodologie générale, il est important d'avoir à l'esprit différentes informations afin de réaliser des modèles permettant d'obtenir de bonnes performances.

Thèse de Julien Taillard, Lille 1, 2009

Celles-ci sont utiles lors de la création des modèles d'applications et d'associations. La modélisation de l'architecture n'étant utilisée pour le moment qu'afin de connaître le nombre de processeurs, elle ne nécessite pas de conseils de modélisation.

Création d'un modèle d'application Lors de la modélisation d'une application, il est important de savoir comment l'algorithme choisi va se comporter par rapport aux accès mémoires. En effet, comme on a pu le constater dans les résultats de multiplication de matrices, un algorithme utilisant mal les caches des processeurs provoquera un grand nombre d'accès mémoire qui pénaliseront les performances. Il est également important de ne pas descendre à un grain de modélisation trop fin et modéliser en utilisant les tâches séquentielles optimisées. Une dernière chose importante à faire et d'utiliser les ports en entrée/sortie dès que nécessaire afin de réduire le nombre de données en mémoires.

Création de l'association Lors de la distribution des tâches, il faut le plus possible placer les tâches utilisant les mêmes données sur le même processeur afin toujours de réduire les accès mémoires. Pour cela, une distribution par blocs est souvent adaptée.

6.7 Conclusion du chapitre

Dans ce chapitre, nous avons présenté selon une vue conceptuelle puis selon une vue implémentation les transformations passant d'un modèle de haut niveau vers le métamodèle de langage procédural agrémenté des constructions OpenMP. Ces transformations ont été complètement implémentées dans l'environnement Gaspard à l'aide du moteur de transformation MoMoTE. Après la présentation de ces transformations, l'implémentation de celles-ci dans les moteurs de transformations est discutée. Les résultats présentés ensuite montrent que les effets de la modélisation de l'application et de la distribution sur les performances des codes générés. En effet, la modélisation de l'application et de la distribution influent sur les performances du code généré, une mauvais choix dans l'un ou l'autre modèle dégradant les performances. Ces résultats permettent d'en déduire une méthodologie générale d'utilisation de Gaspard.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Chapitre 7

Étude de cas : application à la simulation de phénomènes électromagnétiques

7.1 Introduction **144**
7.2 Lois physiques régissant les phénomènes électromagnétiques **144**
7.2.1 Domaine continu 144
7.2.2 Domaine discret 148
7.3 CARMEL **151**
7.3.1 Présentation de CARMEL 151
7.3.2 Parallélisation de CARMEL 151
7.3.3 Structure de matrices creuses utilisées 152
7.4 Patrons de modélisation de briques de bases d'une application basée sur la méthode des éléments finis **153**
7.4.1 Calcul d'une intégrale 153
7.4.2 Calcul d'une contribution élémentaire 155
7.5 Assemblage d'une matrice issue de la méthode des éléments finis **157**
7.5.1 Assemblage d'un problème de magnétostatique en deux dimensions en utilisant la formulation vectorielle (A) 157
7.5.2 Résultats d'exécutions 162
7.6 Résolutions des systèmes d'équations **162**
7.6.1 Les méthodes de résolution liées à CARMEL 163
7.6.2 Modélisation des opérations classiques des solveurs itératifs de type méthode de Krylov 163
7.6.3 Modélisation et résultats d'exécution du gradient conjugué non préconditionné 165
7.6.4 Modélisation et résultats d'exécution du gradient conjugué préconditionné par SSOR 169
7.7 Conclusion **173**

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

7.1 Introduction

Ce chapitre présente l'application de la méthodologie d'IDM introduite lors de cette thèse à des problèmes de calculs scientifiques, plus précisément la simulation des phénomènes électriques. Les phénomènes simulés peuvent être le calcul simple d'un champ magnétique induit par un champ électrique ou la simulation de machines électriques bien plus complexe telles que des alterno-démarreurs de voitures. Le code sur lequel nous nous sommes basés est celui de CARMEL (Code Avancé de Recherche pour les Machines ÉLectriques) développé au Laboratoire d'Électrotechnique et d'Électronique de Puissance de Lille (L2EP). Ce code, en perpétuelle évolution, est développé en Fortran 90.

Avant de présenter le contexte de CARMEL, les lois physiques régissant les problèmes d'électromagnétisme seront présentées dans le domaine continu et dans le domaine discret. Une grande partie du temps d'exécution de CARMEL est passé dans deux parties de code : l'assemblage et la résolution du système. Nous avons donc étudié la modélisation de ces parties. Des patrons de modélisations des calculs à effectuer pour l'assemblage sont ensuite présentés (§ 7.4). L'application de ces patrons de conception sur un problème de magnétostatique et les résultats d'exécutions des codes ainsi obtenus sont donnés. Enfin, la modélisation de la méthode de résolution du gradient conjugué est effectuée et des codes sont générés pour résoudre les problèmes.

7.2 Lois physiques régissant les phénomènes électromagnétiques

Afin de situer les problèmes à résoudre dans le cadre de la simulation de phénomènes électromagnétiques, nous allons présenter les équations régissant ces phénomènes : il s'agit des équations de Maxwell. Ces équations sont définies dans le domaine continu qui sera le premier introduit. La résolution de ces équations n'est pas possible dans ce domaine, c'est pourquoi le passage dans le domaine discret sera ensuite introduit.

7.2.1 Domaine continu

Les phénomènes électromagnétiques dans le domaine continu reposent sur les équations de Maxwell qui décrivent la répartition spatiale et temporelle des champs électriques et des champs magnétiques. Ces champs sont également dépendants des milieux dans lesquels ils circulent, c'est pourquoi les lois de comportements des milieux vis-à-vis des champs sont introduites. Ces lois lient les champs électriques et magnétiques. Afin d'obtenir un problème solvable, des conditions sur les champs aux limites des domaines d'études sont imposées. À partir de ces différentes équations, plusieurs problèmes d'électromagnétismes peuvent être déduits. Nous étudierons plus précisément le problème magnétostatique.

7.2.1.1 Les équations de MAXWELL

La répartition spatiale et temporelle des champs magnétiques et électriques est obtenue par les équations de Maxwell [55, 121] qui régissent l'ensemble des phénomènes électromagnétiques :

$$\mathbf{rot}\ \mathbf{H} = \mathbf{J} + \partial_t \mathbf{D} \tag{7.1}$$

$$\mathbf{rot}\ \mathbf{E} = -\partial_t \mathbf{B} \tag{7.2}$$

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

$$div\ \mathbf{B} = 0 \tag{7.3}$$

$$div\ \mathbf{J} = 0 \tag{7.4}$$

$$div\ \mathbf{D} = \rho \tag{7.5}$$

avec :
$\mathbf{H}$: le champ magnétique (A.m^{-1}) ;
$\mathbf{B}$: l'induction magnétique (T) ;
$\mathbf{E}$: le champ électrique (V.m^{-1}) ;
$\mathbf{D}$: l'induction électrique (C.m^{-2}) ;
$\mathbf{J}$: la densité de courant de conduction (A.m^{-2}) ;
ρ : la densité volumique de charges (C.m^{-3}) ;
t : le temps.

Ces équations expriment des comportements physiques. Ainsi, l'équation (7.2) exprime que la variation d'un champ magnétique entraîne la création d'un champ électrique.

On se place dans le cadre de l'hypothèse de la quasi-statique, ce qui entraîne que l'on néglige la propagation des ondes. La composante $\partial_t \mathbf{D}$ devient alors nulle, l'équation (7.1) s'écrit alors sous la forme suivante :

$$\mathbf{rot}\ \mathbf{H} = \mathbf{J} \tag{7.6}$$

7.2.1.2 Lois de comportement des milieux

Les équations de MAXWELL décrivent géométriquement les distributions des différentes grandeurs électromagnétiques quelque soit le milieu considéré. À ces équations, on introduit des fonctions permettant de relier les champs électromagnétiques. Ces fonctions prennent en compte les propriétés des milieux rencontrés. Le comportement électrique et magnétique des milieux est pris en compte par les lois de comportement qui lient les champs magnétiques et électriques entre eux. Ces relations mettent également en jeu des variables telles que la température. Ces variables sont considérées comme constantes afin que les lois ne dépendent que des champs. Les lois de comportements des milieux sont les suivantes :

$$\mathbf{J} = \sigma \mathbf{E} \tag{7.7}$$

$$\mathbf{D} = \epsilon \mathbf{E} \tag{7.8}$$

$$\mathbf{B} = \mu_{\mathbf{H}} \mathbf{H} \tag{7.9}$$

avec σ la conductivité électrique ($\Omega^{-1} m^{-1}$) (aptitude du matériau à laisser passer le courant électrique), ϵ la permittivité électrique ($\mathbf{F}/m$) (la réponse d'un milieu donné à un champ électrique) et $\mu_{\mathbf{H}}$ la perméabilité magnétique ($\mathbf{H}.m^{-1}$) (faculté d'un matériau à modifier un champ magnétique). Le comportement de $\mu_{\mathbf{H}}$ dépend du type de matériau. Si le matériau n'est pas ferromagnétique, le modèle est linéaire, $\mu_{\mathbf{H}}$ est donc constant. Pour les matériaux ferromagnétiques, il faut prendre en compte le phénomène d'hystérésis. Bien qu'il soit possible d'utiliser des modèles complexes prenant en compte ce phénomène, il est souvent utilisé, dans le cadre de matériaux ferromagnétiques doux, une valeur de $\mu_{\mathbf{H}}$ dépendante ou non de $\mathbf{H}$.

Thèse de Julien Taillard, Lille 1, 2009

7.2.1.3 Conditions aux limites

Le système composé par les équations de Maxwell et les lois de comportements possède une infinité de solutions. Afin d'assurer l'unicité de la solution, des conditions sont imposées aux limites du domaine. L'imposition d'une condition limite consiste à ajouter une information concernant la fonction inconnue sur tout ou partie de la frontière du domaine d'étude.

Deux types de conditions aux limites sont couramment utilisés :
- les conditions aux limites de Dirichlet (valeur de la fonction) ;
- les conditions aux limites de Neumann (valeur de la dérivée normale de la fonction).

L'utilisation de l'une ou l'autre condition limite dépend du type de problème que l'on traite.

7.2.1.4 Les problèmes d'électromagnétismes

À partir des équations de Maxwell et des lois de comportements, il est possible de distinguer des problèmes indépendants. Quatre problèmes peuvent être distingués :
- le problème électrostatique ;
- le problème électrocinétique ;
- le problème magnétostatique ;
- le problème magnétodynamique.

La magnétostatique revient à étudier les phénomènes créés par un courant électrique stationnaire. La magnétodynamique traite les phénomènes magnétiques créés par un courant électrique variable. L'électrostatique étudie les systèmes de charges électriques à l'équilibre. Enfin, l'électrocinétique étudie les interactions dynamiques entre courants électriques dans le cadre de l'utilisation de tensions de courants moyens à élevés. Nous ne détaillerons dans ce manuscrit que le problème magnétostatique, les problèmes traités dans la suite utilisant ce problème.

7.2.1.5 Problème magnétostatique

La magnétostatique est l'étude des phénomènes magnétiques en régime stationnaire. Les équations à considérer sont alors :

$$\mathbf{rot}\ \mathbf{H} = \mathbf{J_S} \tag{7.10}$$

$$div\ \mathbf{B} = 0 \tag{7.11}$$

avec la loi de comportement liant ces champs :

$$\mathbf{B} = \mu_{\mathbf{H}}\mathbf{H} \tag{7.12}$$

$\mathbf{J}_S$ est une densité de courant source connue.

Formulation en potentiel scalaire magnétique Afin de prendre en compte les sources de courant présentes dans le domaine d'étude, le champ magnétique peut être décomposé en deux parties $\mathbf{H} = \mathbf{H}_s - grad\Omega$. Le champ source $\mathbf{H}_s$ vérifie l'équation (7.10) tandis que le

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

potentiel scalaire magnétique Ω introduit les champs à rotationnel nul. À partir des équations (7.11) et (7.12), on obtient alors :

$$div(\mu_{\mathbf{H}}(\mathbf{H}_s - grad\Omega)) = 0 \tag{7.13}$$

Formulation en potentiel vecteur magnétique D'après (7.11), $\mathbf{B}$ est un champ de vecteur à divergence nulle. Cela entraîne que l'induction dérive d'un potentiel vecteur magnétique $\mathbf{A}$ tel que

$$\mathbf{B} = \mathbf{rot}\mathbf{A} \tag{7.14}$$

À partir de l'équation (7.10), on obtient alors :

$$\mathbf{rot}(\nu_{\mathbf{H}}\ \mathbf{rot}\mathbf{A}) = \mathbf{J} \tag{7.15}$$

avec ν_H l'inverse de la perméabilité magnétique $\nu_H = 1/\mu_{\mathbf{H}}$.

Conditions aux limites En supposant que la frontière Γ du domaine d'étude D ait été décomposée en deux parties complémentaires $\Gamma_{\mathbf{B}}$ et $\Gamma_{\mathbf{H}}$ ($\Gamma_{\mathbf{B}} \cap \Gamma_{\mathbf{H}} = \Gamma$ et $\Gamma_{\mathbf{B}} \cup \Gamma_{\mathbf{H}} = \emptyset$), les conditions limites peuvent s'énoncer sous la forme suivante :

$$\mathbf{B} \cdot \mathbf{n}|_{\Gamma_{\mathbf{B}}} \ = \ 0 \tag{7.16}$$

$$\mathbf{H} \wedge \mathbf{n}|_{\Gamma_{\mathbf{H}}} \ = 0 \tag{7.17}$$

où $\mathbf{n}$ est la normale sortante à la frontière Γ.

7.2.1.6 Formulation intégrale

Les différents problèmes de l'électromagnétisme constitués de deux équations différentielles, de lois de comportements et de conditions aux limites du domaine peuvent se ramener, moyennant l'utilisation de potentiels, à une équation différentielle, munie de conditions aux limites. Quelque soit la formulation du problème et le problème traité, les potentiels scalaires ou vecteurs conduisent à la résolution d'un système sous la forme :

$$\begin{cases} \mathcal{L} \cdot \mathbf{u} + \mathbf{f} = 0 \\ \mathcal{C} \cdot \mathbf{u} = 0 \end{cases} \tag{7.18}$$

dans lequel :
$\mathcal{L}$ est un opérateur linéaire ;
$\mathbf{u}$ est le vecteur à déterminer ;
$\mathbf{f}$ représente un champ donné du problème ;
$\mathcal{C}$ est un opérateur associé aux conditions limites.

Dans la suite, le problème sera résolu sous cette forme.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

7.2.2 Domaine discret

Les formulations dans le domaine continu ne peuvent être résolues de manière analytique à cause de la complexité géométrique des dispositifs. Afin de résoudre ces équations, nous allons avoir recours aux méthodes de résolutions numériques, c'est pourquoi le problème doit être discrétisé. La discrétisation se fait à l'aide de la méthode des éléments finis. Les discrétisations, données ici, sont admises et proviennent des travaux menés sur la résolution des problèmes d'électromagnétismes par la méthode des éléments finis. De plus amples détails sur la discrétisation des problèmes du domaine continu se trouvent dans [25, 68, 95, 94].

7.2.2.1 Méthode des éléments finis

La méthode des éléments finis consiste à réaliser une approximation du domaine d'étude à l'aide d'éléments et à approximer les différentes grandeurs sur ces éléments. Les éléments utilisés peuvent avoir différentes formes (tétraèdres, prismes ou hexaèdres). Les grandeurs électromagnétiques (connues et inconnues) seront projetées dans les espaces fonctionnels discrets à l'aide de fonctions d'interpolations.

Méthode des résidus pondérés Afin de résoudre le système, les grandeurs électromagnétiques sont discrétisées. Celles-ci sont projetées sur une base de dimension connue. L'inconnue $\mathbf{u}$ s'écrit alors comme une combinaison linéaire des fonctions de base de cet espace. Afin de vérifier les équations en potentiel introduites, la méthode des résidus pondérés est utilisée. Cette méthode consiste à multiplier l'équation du système 7.18 par une fonction test, notée $\mathbf{v}$:

$$\int_D [\mathcal{L} \cdot \mathbf{u}] \cdot \mathbf{v} = f \cdot \mathbf{v} \tag{7.19}$$

Dans le but de réduire l'ordre de dérivabilité de l'inconnue $\mathbf{u}$, on effectue une intégration par parties en utilisant l'opérateur adjoint de $\mathcal{L}$ (noté $\mathcal{L}^*$), on obtient alors :

$$\int_D [\mathcal{L}^* \cdot \mathbf{v}] \cdot \mathbf{u} = f' \cdot \mathbf{v} \tag{7.20}$$

Il existe plusieurs choix de fonctions tests $\mathbf{v}$. Dans le cadre de la méthode de Galerkin, nous utiliserons les fonctions d'interpolations associées à $\mathbf{u}$. Ces fonctions sont définies dans le paragraphe suivant.

Fonctions d'interpolations Les fonctions d'interpolations permettent de discrétiser le champ $\mathbf{u}$ dans le maillage construit pour la discrétisation du domaine. Dans le maillage, on retrouve des noeuds, des arêtes, des facettes et des éléments. Une fonction de forme est associée à chacune de ces formes. Celles-ci seront notées ω_n, $\mathbf{w}_a$, $\mathbf{w}_f$ et ω_e respectivement pour les noeuds, les arêtes, les facettes et les éléments.

Pour chaque noeud n du maillage, une fonction scalaire nodale ω_n est introduite. Cette fonction vaut 1 au nœud n et est nulle sur tous les autres nœuds du maillage.

$$\sum_{n \in \mathcal{N}} \omega_n = 1 \tag{7.21}$$

De même, pour les arêtes et les facettes, on définit les fonctions d'interpolations $\mathbf{w}_a$ et $\mathbf{w}_f$ telles que :

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

$$\mathbf{w}_a = \omega_n \cdot \mathbf{grad}\omega_m - \omega_m \cdot \mathbf{grad}\omega_n \tag{7.22}$$

où le couple (n,m) désigne les indices des nœuds constituant les extrémités de l'arête a.

$$\mathbf{w}_f = \sum_{n,m,p \in \mathbf{f}} \omega_n \cdot \mathbf{grad}\omega_m \wedge \mathbf{grad}\omega_p \tag{7.23}$$

où le triplet (n,m,p) désigne les nœuds composant la facette $\mathbf{f}$.

Enfin la fonction d'interpolation sur les éléments ω_e est égale à l'inverse du volume de e, nulle sur les autres éléments.

Chacune de ces fonctions engendre un espace des éléments de sa forme de dimension finie ($\mathcal{W}_n$, $\mathcal{W}_\mathbf{a}$, $\mathcal{W}_f$ et $\mathcal{W}_e$ respectivement). Les opérateurs vectoriels (le gradient, le rotationnel et la divergence) ainsi que les grandeurs électromagnétiques étudiées sont également discrétisés dans les espaces précédemment définis. Le tableau 7.1 donne les espaces dans lesquels sont discrétisées les grandeurs.

$\mathcal{W}_n$	$\mathcal{W}_a$	$\mathcal{W}_f$	$\mathcal{W}_e$
Ω	**A**,**H**	**B**,**J**	
	E	**D**	ρ

TAB. 7.1: Espaces discrétisés associés aux grandeurs électromagnétiques

À partir de ces discrétisations des grandeurs sur les éléments, des fonctions d'interpolations définies et en utilisant la méthode des résidus pondérés, on peut en déduire les formulations discrètes des différents problèmes de l'électromagnétisme.

Opérateurs adjoints des opérateurs vectoriels Les opérateurs vectoriels sont également discrétisés dans les différents espaces des fonctions d'interpolations. Les opérateurs adjoints de ces opérateurs sont les suivants :

- le rotationnel est son propre opérateur adjoint ;
- l'opérateur adjoint de div est $-grad$ et inversement.

7.2.2.2 Formulation discrète du problème magnétostatique

Formulation en potentiel scalaire Le potentiel scalaire Ω est décomposé dans l'espace associé aux nœuds. On a donc :

$$\Omega = \sum_{n \in N} \omega_n \cdot \Omega_n \tag{7.24}$$

où Ω_n est la valeur du potentiel scalaire au noeud n.

Le potentiel est décomposé dans l'espace associé aux arêtes. On a donc :

$$\mathbf{H}_s = \sum_{a \in A} \omega_a \cdot \mathbf{H}_{sa} \tag{7.25}$$

où $\mathbf{H}_{sa}$ est la valeur du champ magnétique à l'arête a calculée à partir de $\mathbf{J}_S$.

La forme intégrale de la formulation en potentiel scalaire magnétique s'écrit, en appliquant la méthode des résidus pondérés, à partir de l'équation(7.13) :

$$\int_D \psi \cdot [div\ \mu_\mathbf{H}(\mathbf{H}_s - grad\Omega)] = 0 \tag{7.26}$$

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

où ψ est la fonction de test. La méthode de Galerkin entraine que les fonctions d'interpolations sont les fonctions d'interpolations liées au maillage. On obtient alors :

$$\int_D \omega_n \cdot [div\ \mu_{\mathbf{H}}(\mathbf{H}_s - grad\Omega)] = 0 \qquad \forall\, \omega_n \in \mathcal{W}_n \tag{7.27}$$

À l'aide de la formule de Green, on obtient alors la formulation faible suivante :

$$\int_D \mu_{\mathbf{H}}(grad\omega_n\Omega - grad\omega_n \cdot \mathbf{H}_s) + \int_\Gamma \omega_n \mu_H \mathbf{H}_s + \int_\Gamma \omega_n(\mu_{\mathbf{H}} grad\Omega)\cdot \mathbf{n} = 0 \qquad \forall\, \omega_n \in \mathcal{W}_n \tag{7.28}$$

L'intégrale de surface Γ s'annule grâce aux conditions limites du problème, l'équation (7.28) s'écrit alors :

$$\int_D \mu_{\mathbf{H}}(grad\omega_n\Omega) = \int_D \mu_{\mathbf{H}} grad\omega_n \cdot \mathbf{H}_s \qquad \forall \omega_n \in \mathcal{W}_n \tag{7.29}$$

Formulation en potentiel vecteur Le potentiel vecteur $\mathbf{A}$ est projeté dans l'espace des arêtes :

$$\mathbf{A} = \sum_{a \in A} \mathbf{w}_a \cdot A_a \tag{7.30}$$

où A_a est la valeur du potentiel vecteur A sur l'arête a. La densité de courant est discrétisée dans l'espace des facettes, on a alors :

$$\mathbf{J}_s = \sum_{f \in F} \mathbf{w}_f j_s \tag{7.31}$$

La forme intégrale de la formulation à résoudre, déduite de l'équation (7.15), est donc :

$$\int_D \mathbf{w}_a[\mathbf{rot}(\nu_H \mathbf{rot}\mathbf{A} - \mathbf{J}_s)] = 0 \qquad \forall\, \mathbf{w}_a \in \mathcal{W}_a \tag{7.32}$$

En effectuant une intégration par parties et en annulant l'intégrale de surface à l'aide des conditions limites, l'équation 7.32 devient alors :

$$\int_D \mathbf{rot}\mathbf{w}_a \nu_H \mathbf{rot}\mathbf{A} = \int_D \mathbf{w}_a J \qquad \forall\, \mathbf{w}_a \in \mathcal{W}_a \tag{7.33}$$

7.2.2.3 Résolution numérique des problèmes

Les problèmes discrétisés seront résolus numériquement. Nous allons étudier ici les différentes méthodes mathématiques dont nous aurons besoin afin de construire un système à résoudre par les méthodes numériques. Pour cela, le problème va être écrit sous une forme matricielle. L'inconnu $\mathbf{u}$ pouvant s'écrire en combinaison linéaire des fonctions d'interpolations. On peut déduire un système matriciel à partir de l'équation (7.20) :

$$[S][\mathbf{U}] = [\mathbf{F}] \tag{7.34}$$

dans lequel U est un vecteur contenant les inconnues u_k. S la matrice de raideur dans laquelle on retrouve les termes élémentaires [95]. La construction de systèmes d'équations matriciels à partir des équations est nommée l'assemblage.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Calcul de l'intégrale sur le domaine L'intégrale sur le domaine D est difficile à calculer, il est possible de ramener cette intégrale en la somme des intégrales sur les éléments du maillage. Nous avons donc, pour le membre gauche de l'équation (7.20), l'équation suivante :

$$\int_D [\mathcal{L}^* \cdot \mathbf{v}] \cdot \mathbf{u} = \sum_{e=1}^{Ne} \int_e [\mathcal{L}^* \cdot \mathbf{v}] \cdot \mathbf{u} \tag{7.35}$$

où N_e est le nombre d'éléments maillant le domaine D.

Intégration numérique Les intégrales à calculer sont généralement complexes, c'est pourquoi il est préférable de les calculer numériquement. Pour cela, la méthode des points de Gauss est utilisée. Le calcul à réaliser est alors le suivant :

$$\int_E [\mathcal{L}^* \cdot \mathbf{v}] \cdot \mathbf{u} \approx \sum_{i=1}^{nbPtGauss} \rho_i [\mathcal{L}^* \cdot \mathbf{v}] \cdot \mathbf{u}(z_i) \tag{7.36}$$

dans laquelle ρ_i et z_i sont respectivement les poids et points de Gauss. Ces points et poids de Gauss sont des points bien choisis dans l'élément et des coefficients numériques tels que la formule intègre exactement un polynôme de degré n.

7.3 CARMEL

7.3.1 Présentation de CARMEL

CARMEL (Code Avancé de Recherche en Modélisation ÉLectromagnétique) est un code initialement développé dans l'équipe Modélisation du laboratoire L2EP. Ce code est aujourd'hui développé dans le cadre d'un laboratoire commun entre cette équipe du L2EP et l'équipe de modélisation en électromagnétisme d'EDF R&D : le LAMEL (Laboratoire Avancé de Modélisation du Matériel Électrique). C'est un code basé sur la méthode des éléments finis permettant la résolution de problème en électrocinétique, en magnétostatique ou encore en magnétodynamique. Le code est actuellement écrit en Fortran 90, et est en perpétuelle évolution.

7.3.2 Parallélisation de CARMEL

La parallélisation du code de CARMEL a déjà fait l'objet de la thèse d'Emmanuel Cagniot [25]. Durant sa thèse, celui-ci a développé, en utilisant le langage HPF et la librairie de communication Halos [16], deux versions parallèles de CARMEL selon des approches « légères » et « lourdes ». L'approche « légère » consistait alors à introduire, sans modifier la structure du code, des directives de compilations HPF afin de paralléliser les morceaux de code le pouvant (les directives introduites n'excédaient alors pas 1% du nombre de lignes de la version séquentielle). L'approche « lourde » consistait, elle, à modifier grandement le code en changeant d'algorithme global. L'approche « lourde » était basée sur une méthode de décomposition de domaine (la méthode du complément de Schur [30]). Le principe d'une telle méthode est de confier à chaque processeur un domaine du problème à résoudre, les communications entre processeurs n'intervenant que pour la gestion des interfaces entre les domaines. Ce type de méthode est principalement utilisé pour programmer les machines

à mémoires distribuées afin de résoudre des problèmes demandant beaucoup de mémoire et bénéficiant ainsi du stockage d'une partie du problème dans chacune des mémoires distribuées. Ces codes n'ont pu être maintenus au L2EP en raison de l'abandon de HPF par la communauté du calcul haute performance. L'intérêt de l'utilisation d'une approche orientée modèles se fait alors sentir puisque la maintenance des modèles doit être plus simple que celle du code (le modèle ne contenant pas les détails d'implémentation) et le changement de langage parallèle utilisé impacte éventuellement les tâches élémentaires, mais pas le modèle global.

7.3.3 Structure de matrices creuses utilisées

Les matrices obtenues par la méthode des éléments finis comportent par nature beaucoup de valeurs nulles. C'est pourquoi les matrices sont stockées sous forme de matrices creuses. Afin de faciliter la compréhension des différents modèles utilisant des matrices creuses, nous allons présenter brièvement la structure de matrices creuses utilisée dans les modèles suivants. Les structures de matrices creusent sont classiquement composées de trois vecteurs. Deux d'entre eux sont utilisés afin de réaliser des indirections sur les lignes et colonnes alors que le troisième vecteur est utilisé pour stocker les valeurs non nulles de la matrice. Dans le code de CARMEL, la structure de matrice creuse utilisée est le MORSE également nommé SCSR (Symmetric Compressed Sparse Row). Cette structure utilise la propriété de symétrie de la matrice en ne stockant que la partie triangulaire inférieure de la matrice. Ce type de matrice creuse rend la programmation des algorithmes sur les matrices plus complexes. Les algorithmes nécessitent alors de parcourir deux fois la matrice (par ligne et par colonne) afin d'avoir la partie triangulaire inférieure et la partie supérieure. Dans le cadre du calcul parallèle, cela va entraîner des réductions dans les calculs. La notion de réduction n'étant pas présente en Gaspard, ce stockage va pénaliser le parallélisme. C'est pourquoi nous allons utiliser la version non symétrique de ce stockage, le stockage CSR (Compressed Sparse Row). Le stockage va organiser par ligne toutes les valeurs non nulles de la matrice. Dans le stockage, illustré à la figure 7.1, le vecteur iA indique les indices dans les tableaux jA et A à laquelle sont stockés chaque ligne. En considérant que l'indice des tableaux commence à 1, les valeurs non nulles de la ligne i sont comprises dans jA (qui indique la colonne de l'élément non nul) et A (qui donne les valeurs) entre les indices iA(i) et iA(i+1)-1. L'élément ayant comme coordonnées (i,jA(k)) a comme valeur la valeur A(k), k étant un indice compris entre iA(i) et iA(i+1)-1.

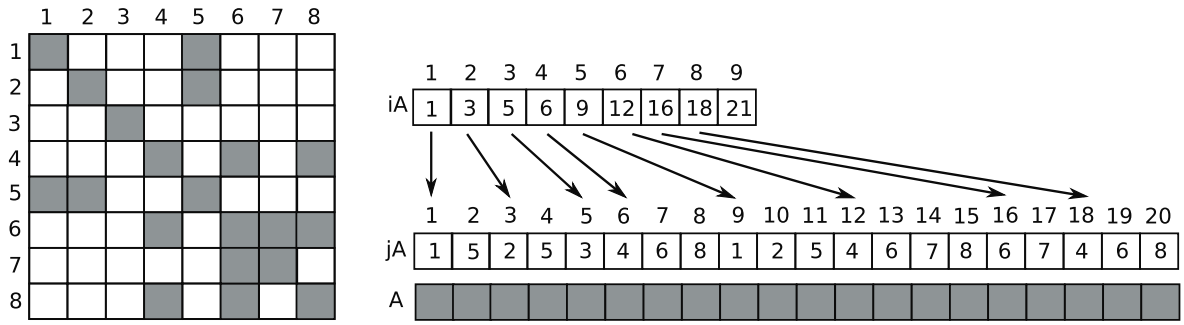


FIG. 7.1: Structure de matrice creuse (CSR) utilisée dans ce chapitre.

En observant les temps d'exécutions de CARMEL, on peut constater que le temps d'exécution est principalement passé dans deux opérations : la résolution du système par le gradient conjugué (≈70% du temps d'exécution) et l'assemblage de la matrice (≈20% du temps d'exécution), le reste du temps étant passé dans l'initialisation de l'application et le post-traitement des données. Afin de réduire les temps d'exécution de l'application, nous nous sommes intéressés à modéliser et paralléliser ces deux principales opérations.

7.4 Patrons de modélisation de briques de bases d'une application basée sur la méthode des éléments finis

Dans le paragraphe 7.2.2.3, nous avons étudié les méthodes mathématiques souvent utilisées afin de construire, dans le cadre de la méthode des éléments finis, le système matriciel à résoudre. Lors de la modélisation d'applications utilisant ces méthodes, nous avons constaté que, quelque soit le problème, les modèles auraient la même forme, la même base. Des patrons de modélisation ont donc été extraits, ceux-ci seront à adapter selon les calculs à réaliser. Dans les patrons donnés ci-dessous, la correspondance entre les tailles des ports des différents composants devra être vérifiée par l'utilisateur qui choisira quels sont les composants à instancier dans le cadre de son problème. De même, le nombre de ports d'entrée des composants réalisant ces fonctionnalités dépendra des problèmes à calculer.

7.4.1 Calcul d'une intégrale

Deux patrons d'intégrations peuvent être distingués, l'intégration d'une fonction par la méthode d'intégration des points de Gauss et l'intégration par la méthode des éléments finis, utilisant la méthode précédente, en passant dans l'élément de référence afin de la calculer.

7.4.1.1 Calcul d'une intégrale par la méthode d'intégration de Gauss

L'intégration d'une fonction dans un domaine se calcule régulièrement numériquement par la méthode d'intégration de Gauss (équation (7.36)) en évaluant la fonction sur les points de Gauss et en pondérant ces évaluations par les poids de Gauss. La décomposition des calculs est alors logique : tout d'abord, va être effectuée la détermination des points et poids de Gauss. Ensuite, la fonction va être évaluée sur les points de Gauss et enfin la pondération est effectuée.

Considérons que l'on veut effectuer l'intégration d'une fonction telle que :

$$f(x, y) \ : \mathbb{R}^p \times \mathbb{R}^r \rightarrow \mathbb{R}^q \tag{7.37}$$

dans laquelle on intègre sur x et où y est une variable nécessaire au calcul.

Le patron de modélisation d'une telle fonction est présenté à la figure 7.2 dans laquelle $NBPointsGauss$ représente le nombre de points de Gauss utilisés et p, q et r les dimensions des domaines.

L'instance du composant *pointsEtPoidsGauss* produit les points et poids de Gauss sur lesquels l'intégration est réalisée. Ensuite, la fonction est évaluée sur les différents points de Gauss (composant *EvaluationFonction*). Enfin, la pondération est réalisée par l'instance du composant *Multiplication*.

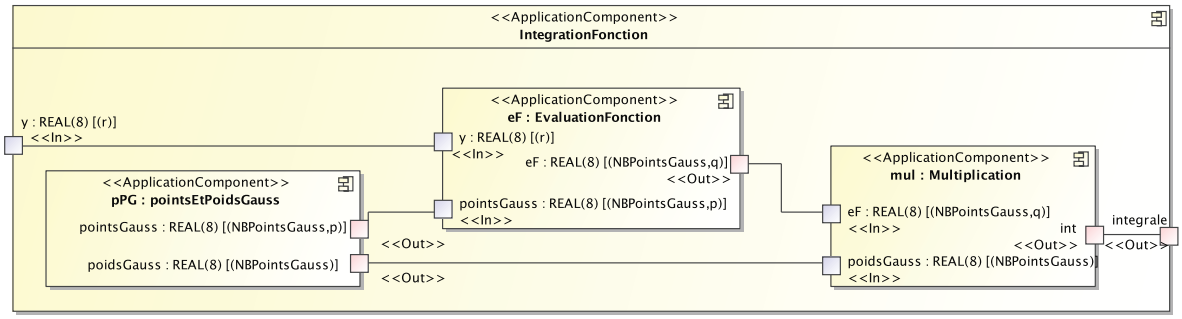


FIG. 7.2: Patron de calcul d'une intégrale par la méthode d'intégration de Gauss.

7.4.1.2 Calcul d'une intégrale dans la méthode des éléments finis

Le calcul d'une intégrale sur l'élément de référence E_{ref} se fait à l'aide de la méthode d'intégration de Gauss dont le patron a été présenté au paragraphe précédent. Les fonctions d'interpolations étant connues sur cet élément de référence, les calculs sont plus simples à réaliser sur cet élément. Le calcul de l'intégrale d'un élément quelconque du maillage nécessite alors de réaliser une transformation de l'élément courant vers l'élément de référence. Cela est alors réalisé par la matrice Jacobienne. On doit alors réaliser le calcul suivant pour chacun des éléments du maillage :

$$\int_E [\mathcal{L}^* \cdot \mathbf{v}] \cdot \mathbf{u} = \int_{E_{ref}} [\mathcal{L}^* \cdot \mathbf{v}] \cdot \mathbf{u} |J| \tag{7.38}$$

dans lequel J représente la matrice Jacobienne de la transformation. Cette intégrale est ensuite calculée à l'aide de la méthode des points de Gauss. L'adaptation du patron de l'intégration de Gauss pour le calcul dans la méthode des éléments finis est illustrée à la figure 7.3. La modélisation de ce patron introduit le calcul de la jacobienne (l'instance du composant *Jacobienne*)et du déterminant (instance du composant *Determinant*) de celle-ci.

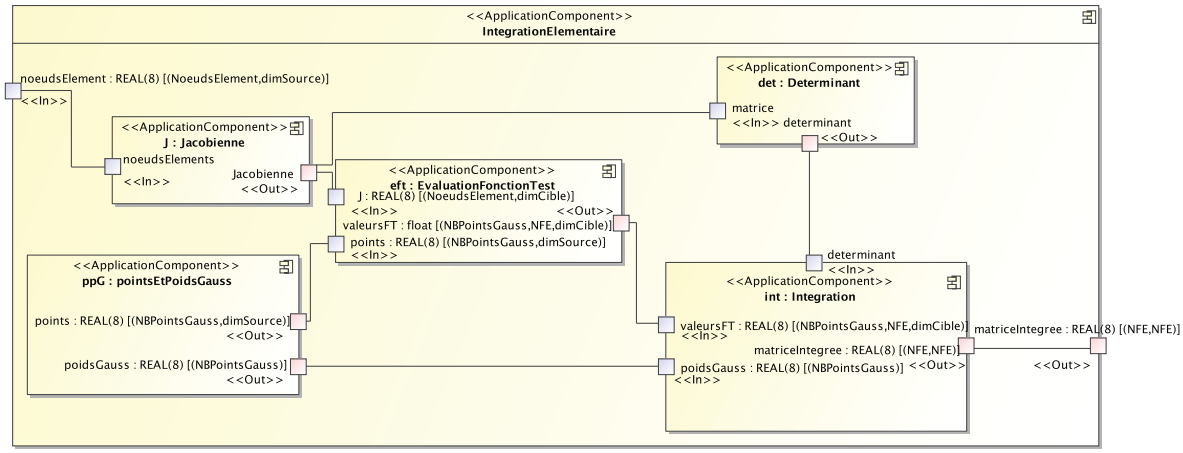


FIG. 7.3: Patron de calcul d'une intégrale avec la méthode des points de Gauss

Le patron et les composants contenus par celui-ci sont paramétrés par les valeurs suivantes :

- NBPointsGauss : le nombre de points de Gauss utilisés ;
- dimSource : le nombre de dimensions du domaine source ;
- dimCible : le nombre de dimensions du domaine cible ;
- NFE : dimension de la fonction de forme utilisée ;
- NoeudsElement : le nombre de noeuds de l'élément.

7.4.2 Calcul d'une contribution élémentaire

L'intégrale sur le domaine se calcule en effectuant la somme des intégrales sur les éléments maillant le domaine (7.35). Ainsi, lorsque toutes les intégrales sur les éléments du maillage sont calculées, il faut additionner ces contributions afin d'obtenir la valeur de l'intégrale. Les intégrales sur les éléments contribuent à l'intégrale du domaine sur les différents noeuds, arêtes ou facettes composant chaque élément selon la fonction d'interpolation utilisée dans le calcul. Le calcul de la contribution élémentaire d'une intégrale revient alors à obtenir les données dont on a besoin pour le calcul, à calculer l'intégrale sur l'élément et enfin à placer la contribution de l'élément dans l'intégrale sur le domaine. Le comportement d'un tel patron sur des éléments triangulaires est illustré à la figure 7.4. La première étape est de récupérer les coordonnées des nœuds composant l'élément. Ensuite avec ces coordonnées des nœuds et d'autres données nécessaires au calcul (la perméabilité magnétique de l'élément par exemple), l'intégrale sur l'élément est alors déterminée. Ensuite, une fonction est chargée, selon les éléments de forme utilisés lors du calcul, de répartir les contributions obtenues lors de l'intégration sur l'élément dans l'intégration sur le domaine.

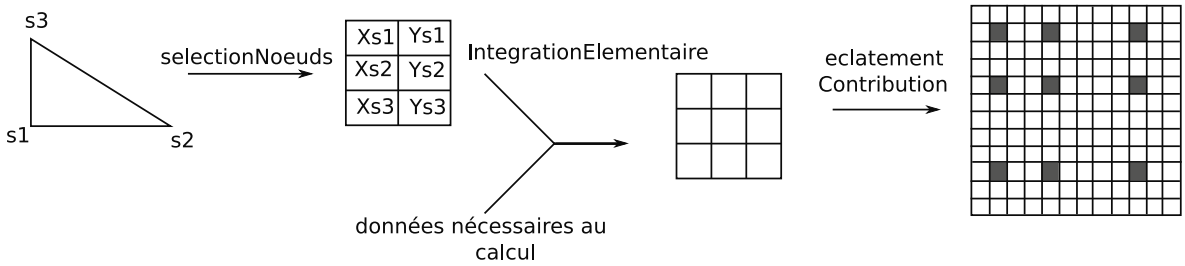


FIG. 7.4: Principe du calcul de la contribution d'un élément et de sa mise dans la matrice représentant l'intégrale

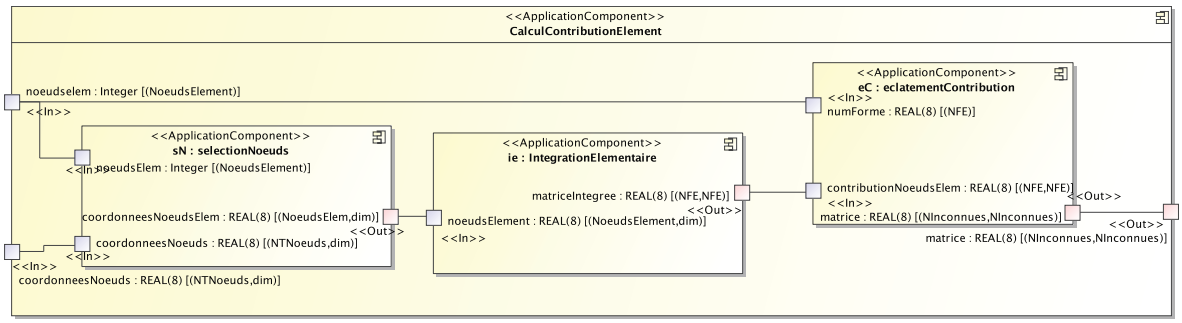
Le patron de modélisation présenté ici va dépendre de la manière dont est modélisée l'application utilisant la méthode des éléments finis. En effet, dans ce calcul, il faut accéder à des données du maillage (coordonnées des nœuds, arêtes appartenant à un élément, etc..). L'accès à ces données change selon la manière dont celles-ci sont stockées. L'utilisation d'un type de données générique permettrait de modéliser ici un composant « générique » réutilisable quelque soit la manière dont sont stockées les données. Ainsi, le patron prendrait comme entrée une donnée de type « Maillage » et produirait une donnée de type « Matrice ». L'environnement Gaspard ne supporte pas les types de données génériques. Nous allons donc illustrer ce patron en utilisant une représentation particulière du maillage. Le maillage utilisé dans le patron est stocké sous forme de différents tableaux. Chaque donnée du maillage est numérotée (les nœuds, les arêtes, les facettes et les éléments). Des tableaux permettent de connaître quels sont les nœuds, arêtes et facettes appartenant à un élément (ces tableaux

sont nommés *connectivite*, le tableau *connectiviteElementNoeuds* stocke les numéros de nœuds constituant un élément). D'autres tableaux donnent les coordonnées des nœuds, les nœuds composant une arête ou encore les nœuds appartenant à une facette. À partir de ces différents tableaux, on retrouve toutes les informations du maillage.

Dans les paragraphes suivants, nous présenterons les patrons de modélisations pour les matrices denses et creuses sur un exemple n'utilisant que les noeuds de l'élément.

7.4.2.1 Assemblage d'une matrice dense

L'assemblage d'une matrice dense permet d'illustrer le principe de base vu à la figure 7.4, le patron d'un tel assemblage est présenté à la figure 7.5. Le composant principal *CalculContributionElement* prend sur ces ports d'entrée la liste des noeuds composants l'élément ainsi que la liste des coordonnées de tous les noeuds et produit une matrice dense ayant les contributions de l'élément réparties dans la matrice globale. L'instance du composant *selectionNoeuds* produit un tableau contenant, dans l'ordre des numéros des nœuds, les coordonnées des différents noeuds de l'élément. Ces coordonnées des noeuds de l'élément sont ensuite utilisées pour calculer l'intégrale sur l'élément (il s'agit d'une instance du patron calculant une intégrale avec la méthode des points de Gauss). Le composant *eclatementContribution* répartit ensuite cette contribution selon les noeuds sur lesquels elle a été calculée.



où les paramètres sont les suivants :

NoeudsElem : nombre de nœuds dans un élément ;
NTNoeuds : nombre total de nœuds dans le maillage ;
NFE : dimension des fonctions de formes ;
NInconnues : nombre d'inconnues du système que l'on construit ;
dim : dimension dans laquelle est le problème.

FIG. 7.5: Calcul et répartition d'une contribution élémentaire dans la matrice.

7.4.2.2 Assemblage d'une matrice creuse

La taille des problèmes traités est souvent grande et le stockage dense d'une matrice prend beaucoup de place. Or il arrive très régulièrement que les matrices comportent un grand nombre de valeurs nulles, c'est pourquoi les matrices creuses sont utilisées : on ne stocke que les valeurs non nulles des matrices. Les matrices creuses sont classiquement stockées dans trois vecteurs : deux vecteurs d'indirection et un vecteur de valeurs. L'utilisation de ce type de matrice provoque alors un changement du patron précédemment défini. La structure du maillage contient toutes les informations permettant de déterminer où seront les valeurs non

nulles de la matrice. La structure de la matrice creuse est ainsi déterminée avant de connaître les valeurs qui y seront. Nous considérerons, dans le patron de conception donné, que la structure de la matrice a été déterminée précédemment. L'adaptation aux matrices creuses du patron de modélisation de l'ajout d'une contribution élémentaire à l'intégrale sur le domaine est présentée à la figure 7.6. Dans cette figure, le nouveau paramètre *NValeursNonNulles* exprime le nombre de valeurs non nulles qui seront comprises dans la matrice. Parmi les tâches de calculs de ce patron, seul le composant effectuant la distribution des contributions dans la matrice est modifiée. Il s'agit en effet de la tâche qui produit la matrice de sortie, le code de la fonction sur laquelle le composant est déployé devra donc gérer la nature creuse de la matrice.

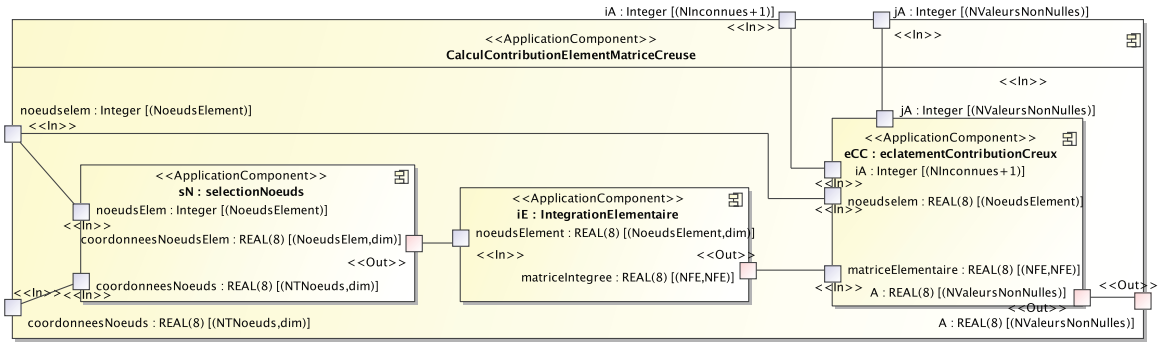


FIG. 7.6: Calcul et répartition d'une contribution élémentaire dans une matrice creuse.

7.5 Assemblage d'une matrice issue de la méthode des éléments finis

7.5.1 Assemblage d'un problème de magnétostatique en deux dimensions en utilisant la formulation vectorielle (A)

On veut calculer le champ magnétique dans un T ferromagnétique ayant deux sources électriques. La figure 7.7 présente une vue schématique de ce dispositif. Le domaine est maillé à l'aide d'éléments triangulaire. Le maillage de ce domaine, que nous utiliserons dans les exemples, comprend 480 000 éléments pour 241 001 sommets. La matrice creuse à construire est constituée de 1 687 007 valeurs non nulles. L'extrême finesse du maillage est uniquement utilisée pour évaluer les performances de la parallélisation.

7.5.1.1 Formulation discrétisée

Le problème général à résoudre est le suivant :

$$\int_D \mathbf{rot}\ \mathbf{w}_a\ \nu_H\ \mathbf{rot}\mathbf{A} = \int_\Omega \mathbf{w}_a\ \mathbf{J} \tag{7.39}$$

dans lequel $\mathbf{w}_a$ est la fonction de formes sur les arêtes, ν_H l'inverse de la perméabilité magnétique, et $\mathbf{J}$ la densité de courant.

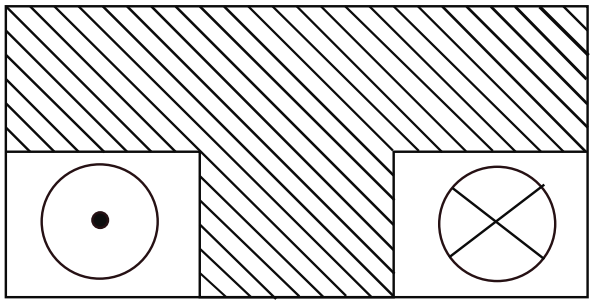


FIG. 7.7: Dispositif de T férromagnétique.

Dans le cadre d'un problème en deux dimensions, l'équation (7.39) peut se simplifier. En effet dans un problème bi-dimentionnel, le rotationnel est forcément perpendiculaire au plan, A sera alors calculé sur les noeuds. L'équation se transforme alors en :

$$-\int_D \mathbf{w}_a \; div \; \nu_H \; grad \; A = \int_D \mathbf{w}_a \; \mathbf{J} \tag{7.40}$$

En appliquant l'opérateur adjoint, on obtient :

$$\int_D grad^T \; \mathbf{w}_a \; \nu_H \; grad \; A = \int_D \mathbf{w}_a \; \mathbf{J} \tag{7.41}$$

L'intégrale sur le domaine D est égale à la somme des intégrales sur les éléments. On a donc :

$$\int_D grad^T \; \mathbf{w}_a \; \nu_H \; grad \; A = \sum_{i=1}^{Ne} \int_{e_i} grad^T \mathbf{w}_a \nu_H \; gradA \tag{7.42}$$

On ramène l'intégrale sur un élément à une intégrale sur l'élément de référence, on obtient donc :

$$\int_{e_i} grad^T \mathbf{w}_a \nu_H \; gradA = \int_{e_{ref}} grad^T \mathbf{w}_{a_{ref}} J^{-T} \nu_H J^{-1} gradA |J| \tag{7.43}$$

où J est la jacobienne.

7.5.1.2 Calcul de l'intégration élémentaire

L'intégration élémentaire doit effectuer le calcul décrit à l'équation (7.43). En appliquant le patron de modélisation du calcul d'une intégrale avec la méthode d'intégration de Gauss décrit au paragraphe 7.4.1, on obtient pour notre problème, le composant présenté à la figure 7.8. La fonction à intégrer nécessite la perméabilité magnétique de l'élément, c'est pourquoi celle-ci est en entrée du composant réalisant l'intégration.

Le composant *PointGaussTriangleRef* produit les points et poids de Gauss utilisés dans le triangle de référence. Ensuite le composant *EvaluationFonctionMagnetostatique2D* évalue la fonction sur les différents points de Gauss, effectuant notamment la multiplication des gradients par la jacobienne et l'inverse de la perméabilité magnétique. Ce composant est modélisé à la figure 7.9.

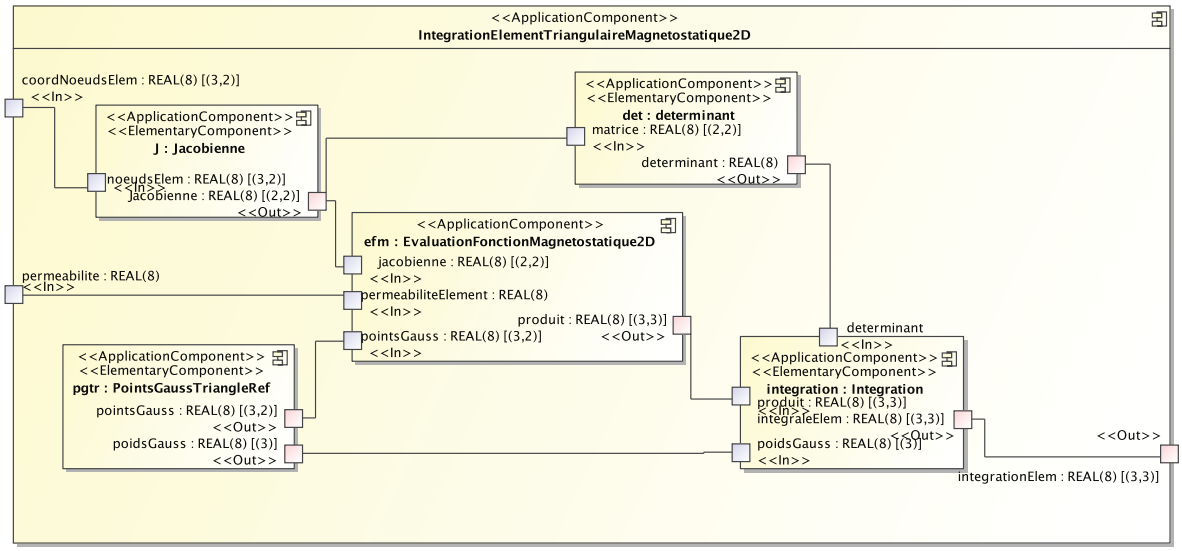


FIG. 7.8: Modélisation du calcul d'une intégrale sur un élément pour le problème magnétostatique en deux dimensions.

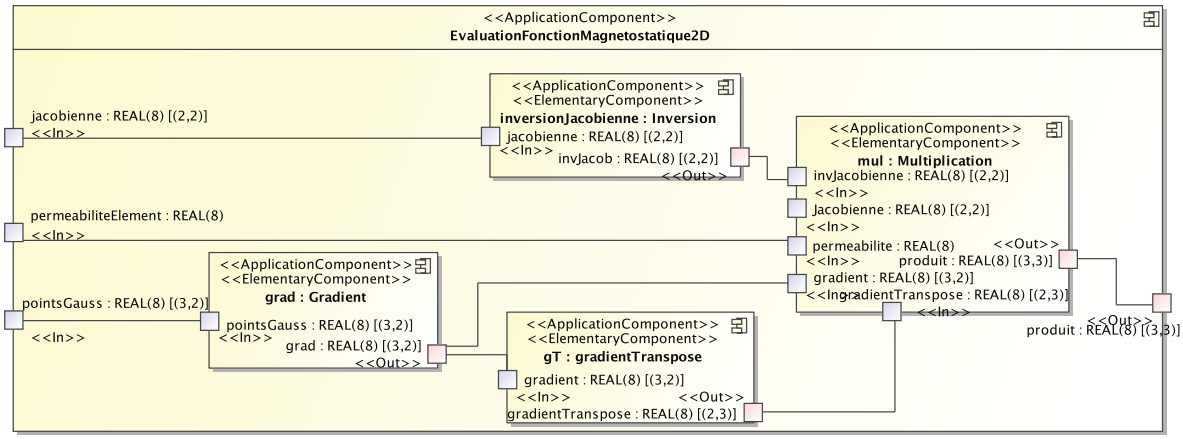


FIG. 7.9: Modélisation de l'évaluation de la fonction du problème magnétostatique.

7.5.1.3 Contribution d'une intégrale élémentaire à l'intégrale du domaine

En appliquant le patron de modélisation de la contribution d'une intégrale à l'intégrale du domaine sur cet exemple, on obtient alors le modèle illustré à la figure 7.10.

7.5.1.4 Calcul sur tous les éléments

Nous avons précédemment modélisé le calcul de l'intégrale sur un élément et vu comment celle-ci contribuait à l'intégrale du domaine. Il reste maintenant à modéliser le calcul global de l'intégrale du domaine, c'est-à-dire le calcul des intégrales sur chacun des éléments du domaine et l'addition de leurs différentes contributions. Le fait que le concept de réduction parallèle ne soit pas présent dans l'environnement Gaspard entraîne une réflexion du

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

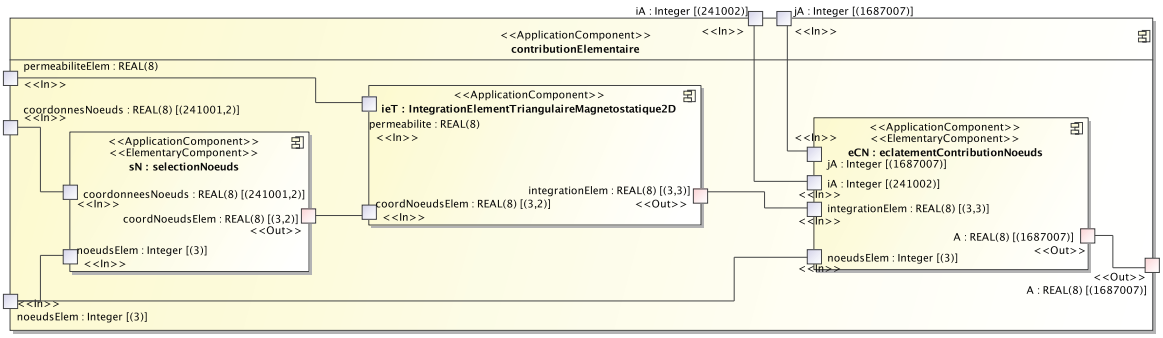


FIG. 7.10: Modélisation de la contribution d'un élément à l'intégrale du domaine.

modeleur[1] sur la modélisation de l'addition des différentes contributions.

Les calculs des intégrales sur chacun des éléments étant totalement indépendants, la première idée pour modéliser cela est de modéliser du parallélisme de données et d'effectuer ensuite l'addition des différentes contributions des éléments. La modélisation du parallélisme de donnée est présentée à la figure 7.11. Dans le composant *CalculContributionsDesElements*, le calcul de l'intégrale sur un élément et la répartition de la contribution dans l'intégrale de domaine est effectuée pour chacun des éléments (l'instance du composant *ContributionElementaire* qui est répétée 480 000 fois). Les ports d'entrées *connectiviteElementNoeuds* et *permeabiliteElements* fournissent respectivement, selon le numéro de l'élément, les numéros des nœuds le composant et sa perméabilité magnétique.

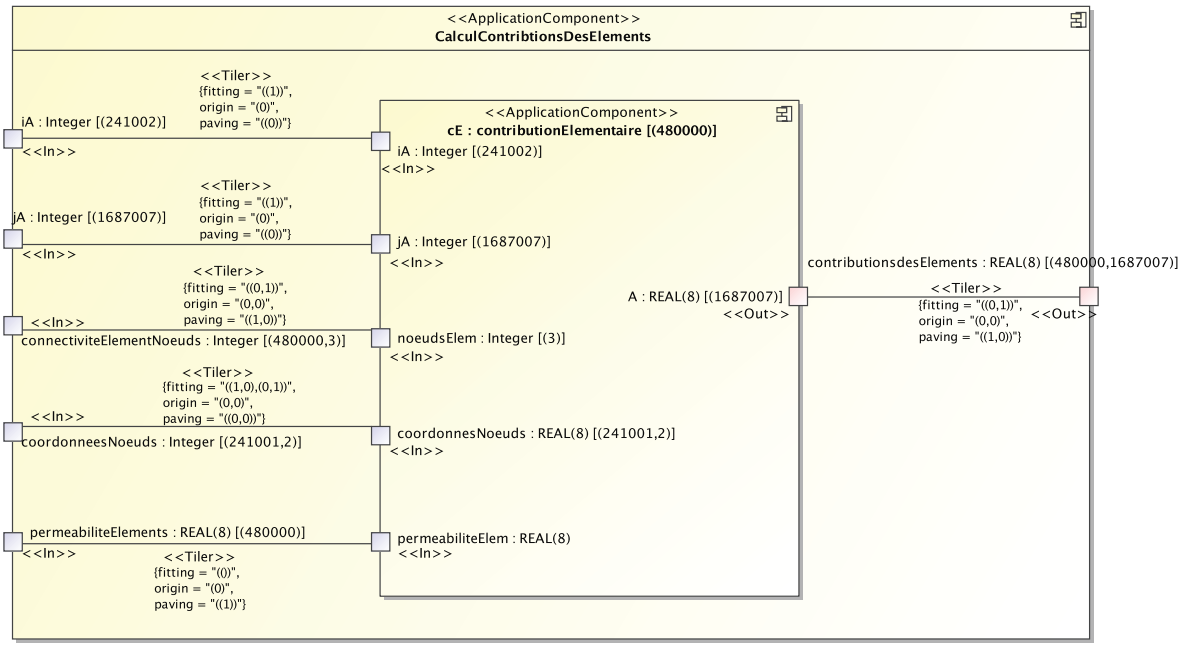


FIG. 7.11: Parallélisme de données calculant la contribution élémentaire de chacun des éléments du domaine.

[1]Personne réalisant les modèles.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Cette modélisation permet d'être indépendante de l'architecture en répartissant les 480 000 calculs selon le nombre de processeurs que l'architecture possède, mais nécessite un grand espace mémoire pour s'exécuter. En effet, durant l'intégration sur les différents éléments et en l'absence de réduction, il faut stocker toutes les contributions avant de les additionner. Cela correspond au port de sortie *contributionsdesElements* du composant *CalculContributionsDesElements* qui représente un très grand tableau (480 000 * 1 687 007 éléments). Cette solution n'est pas envisageable. L'addition des différentes contributions serait alors soit une tâche élémentaire, soit un composant dépendant du nombre de processeurs afin d'être parallélisé (chaque processeur effectuant une somme partielle avant que l'un d'entre eux additionne ces sommes partielles).

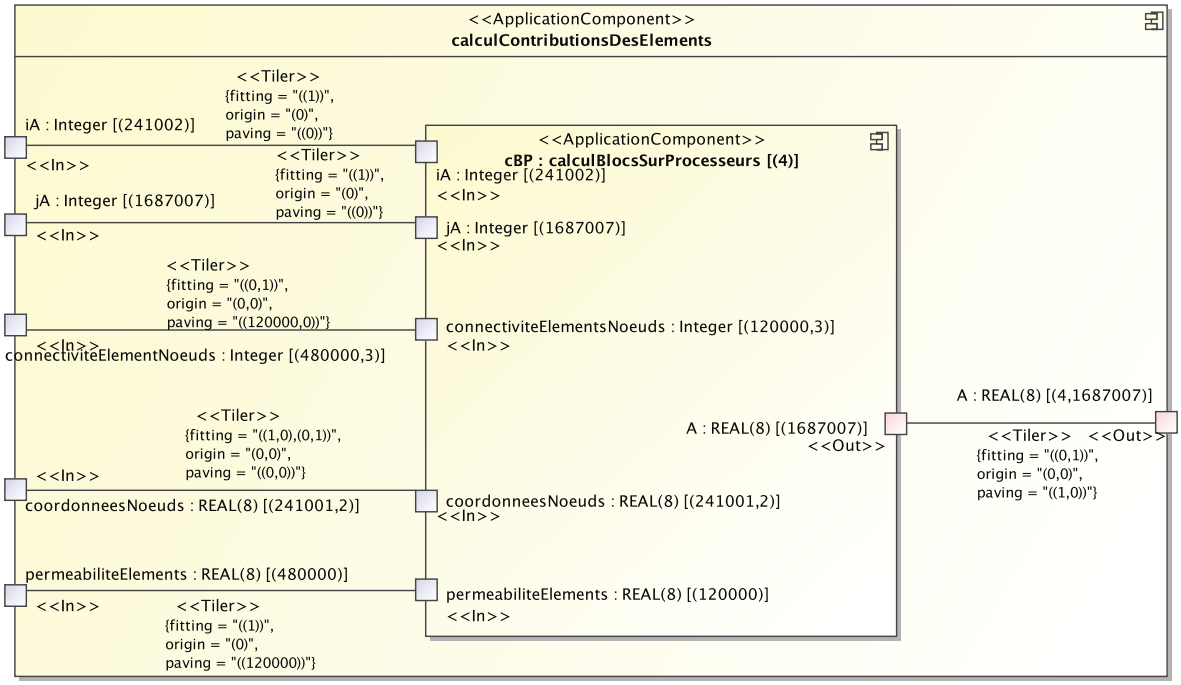


FIG. 7.12: Modélisation du calcul des contributions des éléments par blocs sur les processeurs.

La réduction parallèle des différentes contributions ne pouvant pas être modélisée « proprement » et indépendamment de l'architecture d'exécution, la modélisation utilisée dépendra de celle-ci. Le principe de la modélisation est alors la suivante : selon le nombre de processeurs, chacun se voit confier un bloc d'éléments sur lequel il devra calculer les intégrations sur les éléments, mais également réduire les différentes contributions. Chaque processeur va alors produire une contribution partielle de son bloc d'éléments à l'intégrale du domaine. Il suffit alors de sommer ces différentes contributions partielles afin d'obtenir l'intégrale du domaine. Le composant réalisant un tel comportement est modélisé à la figure 7.12. Le calcul des 480 000 contributions est réparti sur 4 processeurs, chacun d'entre eux calculera 120 000 contributions. Le composant est modélisé pour s'exécuter sur quatre processeurs, l'instance du composant effectuant le calcul sur un bloc d'éléments et additionnant les contributions est donc répétée quatre fois. Lors de la distribution, chaque répétition de l'instance sera placée sur un processeur.

La notion de réduction dans la modélisation Gaspard devrait entraîner la génération d'un

Thèse de Julien Taillard, Lille 1, 2009

code utilisant peu de tableaux intermédiaires en effectuant des sommes partielles sur chacun de processeurs avant d'additionner ces sommes partielles.

7.5.2 Résultats d'exécutions

Le code généré effectuant l'assemblage du problème magnétostatique pour le T ferromagnétique, présenté précédemment (figure 7.2) et composé de 480 000 éléments et 241 000 nœuds, a été exécuté sur le supercalculateur decrypthon1. Les temps d'exécutions sont donnés dans le tableau 7.2. On peut constater dans ces résultats que l'accélération est quasi linéaire. En effet, alors que le temps séquentiel est d'une demi-seconde, le temps sur 2 processeurs est moitié moindre que celui-ci. Le comportement est quasiment le même sur quatre et huit processeurs, il est même supra linéaire sur 8 processeurs.

	Séquentiel	2 Threads	4 Threads	8 Threads
Temps d'exécution (s)	0.50	0.2544	0.1327	0.056563
Accélération	-	1. 96	3.77	8.77
Efficacité	-	0.95	0.94	1.09

TAB. 7.2: Résultats sur le supercalculateur (decrypthon1).

Ces résultats sont néanmoins à nuancer. En effet, l'assemblage est une opération demandant peu de calcul mais utilisant beaucoup de données. Pour être efficace, il est nécessaire d'avoir un réseau d'interconnexion performant. Sur des calculateurs plus classiques, les performances ne sont pas bonnes. Le tableau 7.3 présente les résultats sur le calculateur Cleep. On peut y observer des performances décevantes puisque seule une très petite accéleration est constatée sur quatre processeurs.

	Séquentiel	2 Threads	4 Threads	8 Threads
Temps d'exécution (s)	0.213	0.358	0.1911	0.28

TAB. 7.3: Résultats sur le calculateur Cleep.

L'assemblage est le calcul de la structure de la matrice et des valeurs la composant. Les résultats présentés ici correspondent aux calculs des valeurs. La création de la structure de matrice creuse n'est pas parallélisable, mais est peu coûteuse en temps.

7.6 Résolutions des systèmes d'équations

Une fois que le problème est assemblé, on doit alors le résoudre sous la forme :

$$\mathbf{A} \cdot u = b \tag{7.44}$$

dans laquelle

A : une matrice carrée de taille (Ni,Ni) où Ni est le nombre d'inconnues du système ;
u : vecteur inconnu que l'on cherche ;
b : vecteur connu.

La résolution d'un système d'équations linéaires peut être effectuée via une méthode directe ou via une méthode itérative. Le choix de la méthode de résolution du système

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

linéaire dépend des problèmes, du nombre d'inconnues ainsi que de la forme de la matrice. Les méthodes directes fonctionnent généralement bien avec des matrices denses alors qu'avec des matrices creuses ces méthodes utilisent beaucoup (trop) de mémoires. Les méthodes itératives sont alors utilisées afin de s'approcher progressivement d'une solution convenable pour le système. Dans le cadre de CARMEL, les problèmes sont résolus à l'aide de méthodes itératives qui vont trouver une solution approchée du problème.

7.6.1 Les méthodes de résolution liées à CARMEL

Le choix d'une méthode de résolution est un point important de la programmation d'une application basée sur les éléments finis puisque, selon les problèmes, les méthodes de résolutions sont plus ou moins efficaces. Les méthodes de résolutions pour les problèmes de CARMEL ont été étudiées durant le stage de Daniel Morosan de l'école polytechnique de Bucarest dans l'équipe-projet DaRT et les travaux d'Abdellatif Tinzefte [133] au L2EP. Différentes méthodes de résolutions itératives telles que Jacobi, Gauss-Seidel, multi-grille ou encore gradient conjugué, avec différents préconditionneurs (Cholesky Incomplet (IC0) et Symmetric Successive Over Relaxation (SSOR) notamment), ont été étudiées, mais également implémentées et exécutées sur des problèmes provenant de CARMEL. Les résultats montrent que le gradient conjugué est la méthode itérative la mieux adaptée au problème de CARMEL. Couplée avec des préconditionneurs de type IC0 ou SSOR, la convergence de la méthode de résolution sur les problèmes d'électromagnétismes est bonne, c'est pourquoi nous avons modélisé le gradient conjugué afin de résoudre les équations.

7.6.2 Modélisation des opérations classiques des solveurs itératifs de type méthode de Krylov

Les solveurs itératifs de type méthode de Krylov sont connus pour être principalement une combinaison de produits matrices vecteurs, de mises à jour de vecteurs et de produits scalaires [124], c'est le cas du gradient conjugué. Ces opérations sont parallélisables en utilisant le parallélisme de données [98]. En effet, dans le produit matrice vecteur, chaque point du vecteur de sortie peut être calculé indépendamment des autres. La mise à jour de vecteurs peut être décomposée en mise à jour de morceaux de vecteurs alors que le produit scalaire peut être décomposé en produit scalaire de morceaux de vecteurs dont on doit effectuer la somme ensuite. Avant de présenter la modélisation du gradient conjugué, nous allons introduire une modélisation des fonctions de bases utilisées dans cet algorithme.

Afin d'utiliser les fonctions de base des bibliothèques de calcul scientifique, les modèles de mise à jour de vecteur et de produit scalaire sont réalisés en utilisant le nombre de processeurs. Chaque processeur se verra confier des blocs de vecteurs sur lesquels il devra effectuer les calculs, un composant élémentaire déployé sur la fonction de bibliothèque adaptée calculant le bloc d'un seul coup sans descendre aux opérations scalaires. Il faut, en effet, profiter des fonctions optimisées des bibliothèques BLAS et leurs bonnes utilisations des mémoires caches pour limiter les conflits d'accès mémoires.

Produit matrice vecteur La modélisation du produit matrice vecteur dans le cadre de matrice creuse est illustrée dans la figure 7.13. Le principe du calcul en parallèle du produit matrice vecteur est de calculer chacun des points du vecteur de sortie indépendamment les uns des autres. Cela revient à effectuer pour chaque point, un produit scalaire entre la ligne

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

correspondant au point que l'on calcule et le vecteur. La structure de la matrice creuse n'étant pas connue et fixe (le nombre d'éléments non nuls par ligne varie), il n'est pas possible de sélectionner une seule ligne de la matrice à l'aide d'un tiler. C'est pourquoi pour le calcul de chaque point, la tâche élémentaire *CSRdotproduct* prend en entrée toute la matrice (les trois vecteurs iA, jA et A), le vecteur par lequel la matrice doit être multipliée ainsi qu'un *index*. Cet index indique quelle ligne doit être multipliée par le vecteur afin de produire un point du vecteur de sortie. Afin de récupérer l'indice de répétition de la tâche, le stéréotype « IndicesOnPort » est appliqué sur l'instance et indique que l'indice doit être passé sur le port *index*.

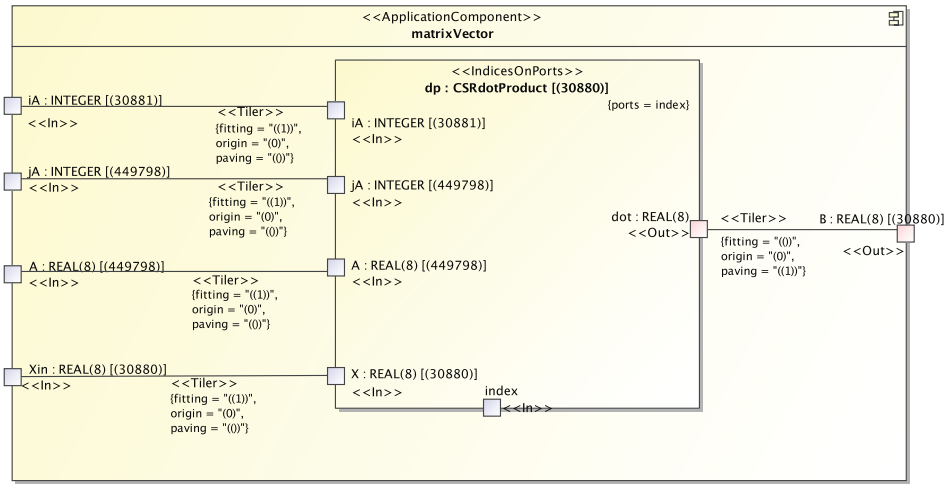


FIG. 7.13: Modélisation de la multiplication matrice creuse - vecteur.

Mise à jour de vecteur La figure 7.14 illustre la mise à jour de vecteur en parallèle. Chaque point du vecteur peut être mis à jour indépendamment des autres. Le principe de la modélisation de la figure 7.14 est de mettre à jour plusieurs blocs d'un vecteur en même temps. Ici, voulant mettre à jour un vecteur de taille (30880) sur quatre processeurs, nous avons divisé ce vecteur en quatre blocs de taille (7720) qui seront mis à jour en parallèle.

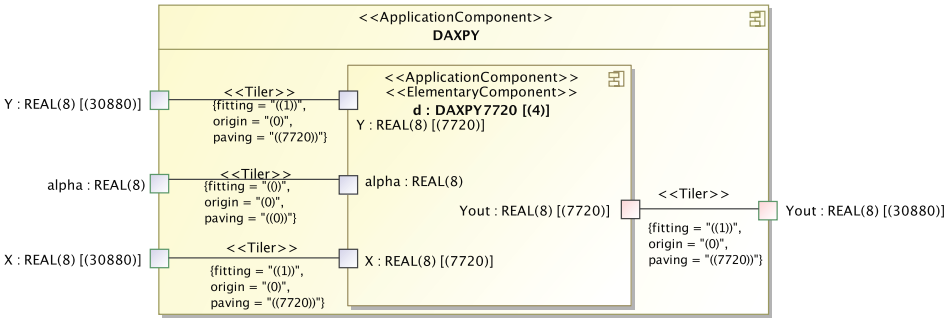


FIG. 7.14: Modélisation de la mise à jour de vecteurs potentiellement parallèle.

Produit scalaire La figure 7.15 illustre la modélisation d'un produit scalaire parallèle. Le produit scalaire peut être parallélisé en réalisant des produits scalaires sur des blocs de vecteurs avant d'additionner les résultats de ces produits par une réduction parallèle ou une somme séquentielle, c'est ce principe qui est utilisé dans la modélisation. Quatre produits scalaires sont potentiellement réalisés en parallèle et rangent leurs résultats dans un tableau. Ensuite une tâche (*sum*) est chargée de réaliser la réduction.

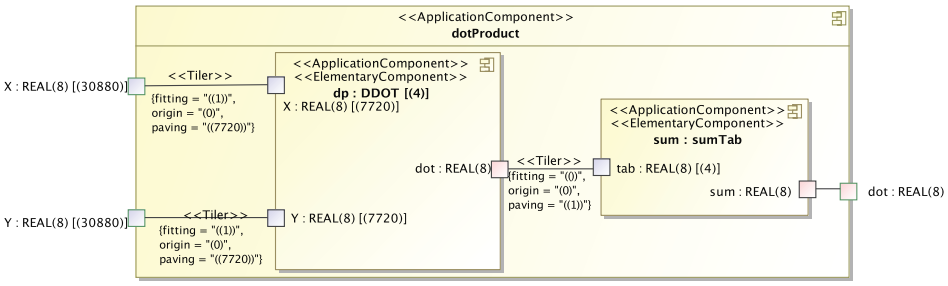


FIG. 7.15: Modélisation d'un produit scalaire potentiellement parallèle.

7.6.3 Modélisation et résultats d'exécution du gradient conjugué non préconditionné

7.6.3.1 Modélisation du gradient conjugué non préconditionné

L'algorithme classique du gradient conjugué est illustré dans la figure 7.16. Après une initialisation du système, l'algorithme itère jusqu'à ce que la précision requise soit atteinte. Une fois le nombre d'itérations maximum ou la précision requise atteinte, il faut alors sortir de la boucle avec l'instruction adéquate du langage cible. Ce type de sortie d'une boucle n'est pas disponible dans le langage de haut niveau de l'environnement Gaspard. C'est pourquoi il est nécessaire de changer la forme de l'algorithme afin de l'adapter à la modélisation. Cela est illustré dans la figure 7.16. Comme l'algorithme le montre, le gradient conjugué est une suite d'opérations fortement séquentielles. On trouve très peu de parallélisme de tâches dans cet algorithme (seules les mises à jour de x et r peuvent être effectuées en parallèle). Lors de la modélisation de cet algorithme, les patrons de modélisations en parallèle des opérations classiques d'algorithmes itératifs présentés au chapitre précédent seront utilisés. Cette modélisation est présentée à la figure 7.17. On peut constater que le composant réalisé est « en longueur », ce qui illustre bien la suite séquentielle d'opérations à réaliser.

7.6.3.2 Résultats d'exécution du gradient conjugué non précontionné

Problème résolu Ces résultats sont effectués sur un exemple académique. Il s'agit d'un barreau de fer dans un cube d'air dans lequel circule du courant et on calcule la distribution du champ dans le domaine complet (figure 7.18). Le problème est résolu dans le cas de la magnétostatique et selon les deux formulations (potentiel scalaire ou potentiel vecteur).

La distribution des tâches du gradient est effectuée par blocs : chaque processeur réalise une partie des mises à jour des vecteurs et calcule un bloc du produit matrice-vecteur.

Thèse de Julien Taillard, Lille 1, 2009

$r_o = b - Ax_0$
$p_0 = r_0$
$Pour \;\; k = 1, 2, ...$
  $q_k = Ap_k$
  $Si \;\; k = 1$
    $p_1 = r_O$
  $Sinon$
    $\beta_{k-1} = \frac{r_{k-1}^T r_{k-1}}{r_{k-2}^T r_{k-2}}$
    $p_k = r_{k-1} + \beta_{k-1} p_{k-1}$
  $Fin \;\; Si$
  $\alpha_k = \frac{r_{k-1}^T r_{k-1}}{p_k^T q_k}$
  $x_k = x_{k-1} + \alpha_k p_k$
  $r_k = r_{k-1} - \alpha_k q_k$
$Fin \;\; Pour$

$r_o = b - Ax_0$
$p_0 = r_0$
$q_1 = Ap_0$
$p_1 = r_O$
$\alpha_1 = \frac{r_0^T r_0}{p_1^T q_1}$
$x_1 = x_0 + \alpha_1 p_1$
$r_1 = r_0 - \alpha_1 q_1$
$erreur = r_1^T r_1$
$k = 1$
$Tant \;\; que \;\; (erreur > tol)$
  $k = k + 1$
  $\beta_{k-1} = \frac{r_{k-1}^T r_{k-1}}{r_{k-2}^T r_{k-2}}$
  $p_k = r_{k-1} + \beta_{k-1} p_{k-1}$
  $\alpha_k = \frac{r_{k-1}^T r_{k-1}}{p_k^T q_k}$
  $x_k = x_{k-1} + \alpha_k p_k$
  $r_k = r_{k-1} - \alpha_k q_k$
  $erreur = r_k^T r_k$
$Fin \;\; Tant \;\; que$

FIG. 7.16: Algorithme du gradient conjugué et son adaptation pour obtenir une boucle itérative.

Résultats sur une matrice de 30880 inconnues (Formulation en potentiel scalaire) Les figures 7.19 et 7.20 présentent les résultats obtenus par les codes de gradient conjugué générés afin de résoudre un problème ayant 30880 inconnues.

La figure 7.19 illustre les temps d'exécutions des codes sur les différentes machines. On peut constater que l'on gagne des performances sur la machine de bureau jusqu'à 4 threads bien que cela ne soit pas linéaire. La perte de performance sur 8 threads s'explique par le fait que la machine ne possède que 4 coeurs de calculs (il y a donc deux threads par coeurs). La perte de performance peut notamment s'expliquer par une perte de temps à réaliser des changements de contexte alors que la mémoire cache est déjà utilisée par un thread. Les temps d'exécutions sur les calculateurs (decrypthon1 et Cleep) sont bons jusqu'à quatre threads. Sur Cleep, on obtient en effet une accélération proche de deux (1.78) sur deux threads, et de 3.10 sur quatre threads. Sur le supercalculateur, on obtient des performances bonnes sur deux et quatre threads (accélération respective de 2.12 et de 3.94). Les performances augmentent encore sur 8 threads pour Cleep (4.66) bien que loin d'être une accélération linéaire. Sur le supercalculateur, les performances diminuent légèrement sur 8 threads.

Résultats sur une matrice de 184844 inconnues (Formulation en potentiel vecteur) Les figures 7.21 et 7.22 présentent les résultats obtenus sur la formulation A non jaugée. Le problème contient 184 844 inconnues.

Les résultats obtenus montrent également un gain de performance sur deux et quatre processeurs. Sur les machines « classiques » (Goelo et Cleep), les accélérations sont moins bonnes que précédemment alors que sur le supercalculateur IBM les performances sont toujours aussi bonnes (même meilleures). Cela peut s'expliquer par le fait que le problème est plus gros et nécessite donc plus de mémoire. Les caches étant petits, l'exécution du code provoque

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

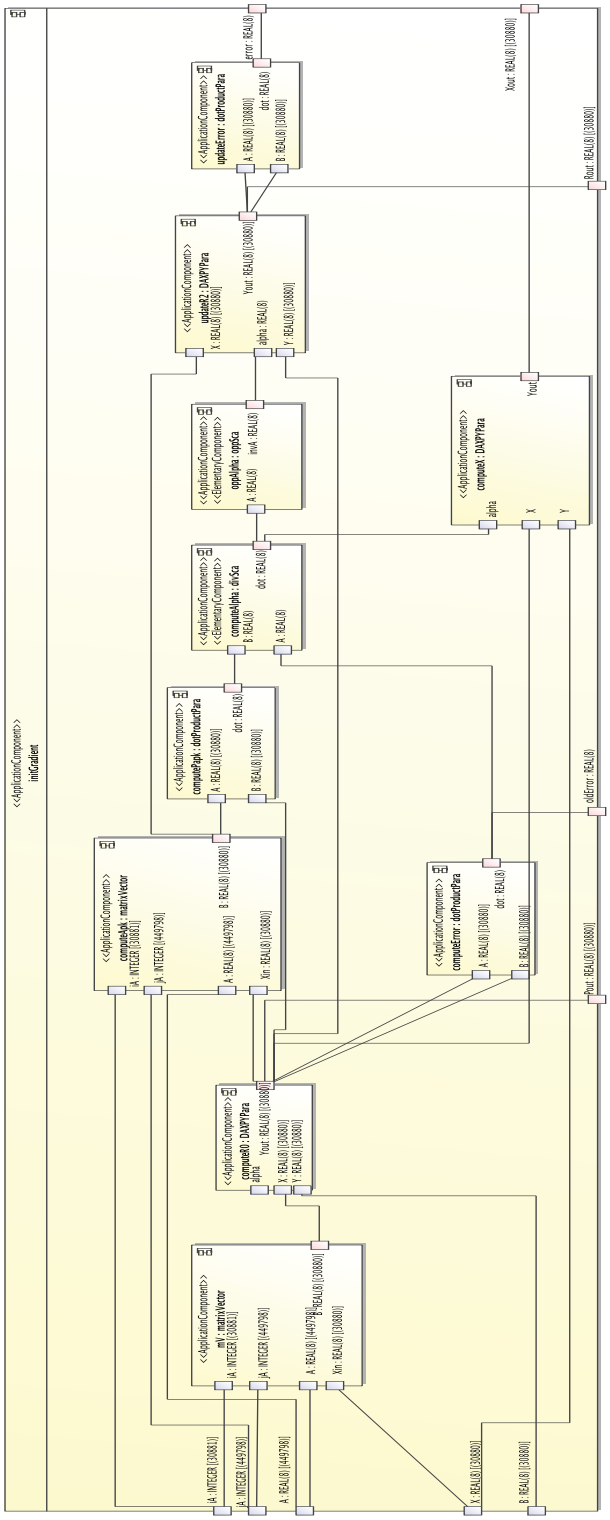


FIG. 7.17: Modèle du gradient conjugué.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

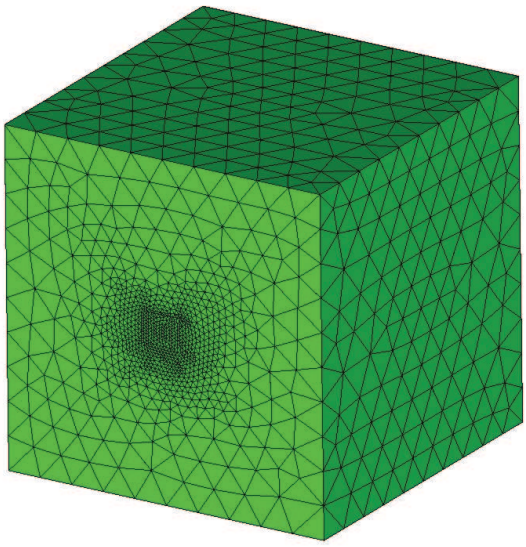


FIG. 7.18: Maillage du barreau de fer dans l'air.

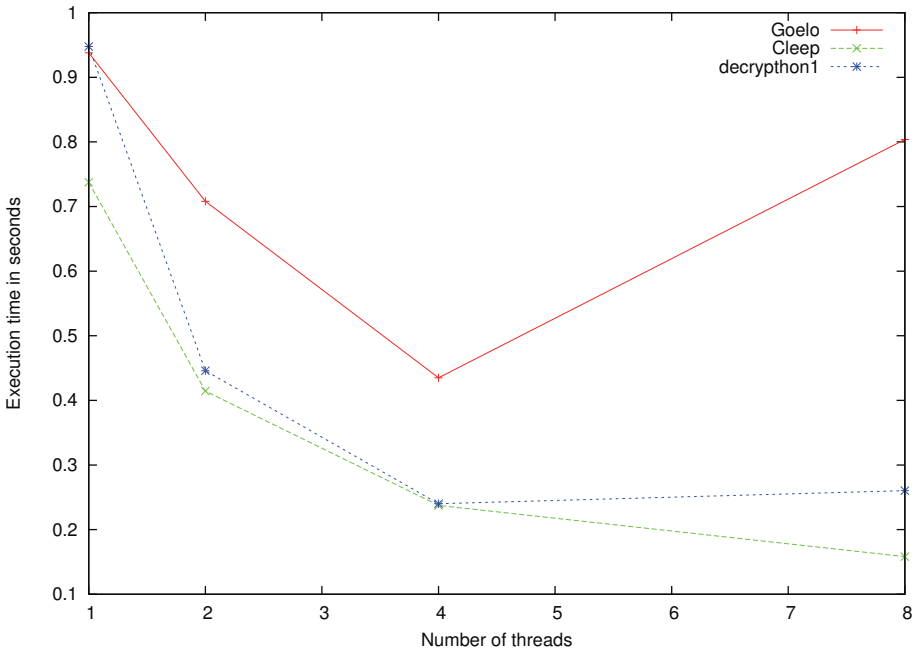


FIG. 7.19: Résultats d'exécution sur un problème contenant 30880 inconnues.

plus de défauts de caches que dans le problème ayant moins d'inconnues. Ces défauts de caches provoquent des accès mémoires qui vont jouer sur les réseaux d'interconnexions des processeurs à la mémoire. Les machines ayant le moins bon réseau d'interconnexion sont pénalisées. Les deux machines ayant un réseau d'interconnexion « normal » ont donc des résultats moins bons que le supercalculateur ayant un réseau d'interconnexion spécialisé.

	Goelo		Cleep		decrypthon1	
Nombre de threads	Acc	Eff	Acc	Eff	Acc	Eff
1	1	-	1	-	1	-
2	1.32	0.66	1.78	0.89	2.12	1.06
4	2.15	0.54	3.10	0.77	3.94	0.98
8	1.166	0.15	4.6630	0.58	3.63	0.45

FIG. 7.20: Accélérations et efficacité des codes générés sur les différentes machines. Problème ayant 30880 inconnues.

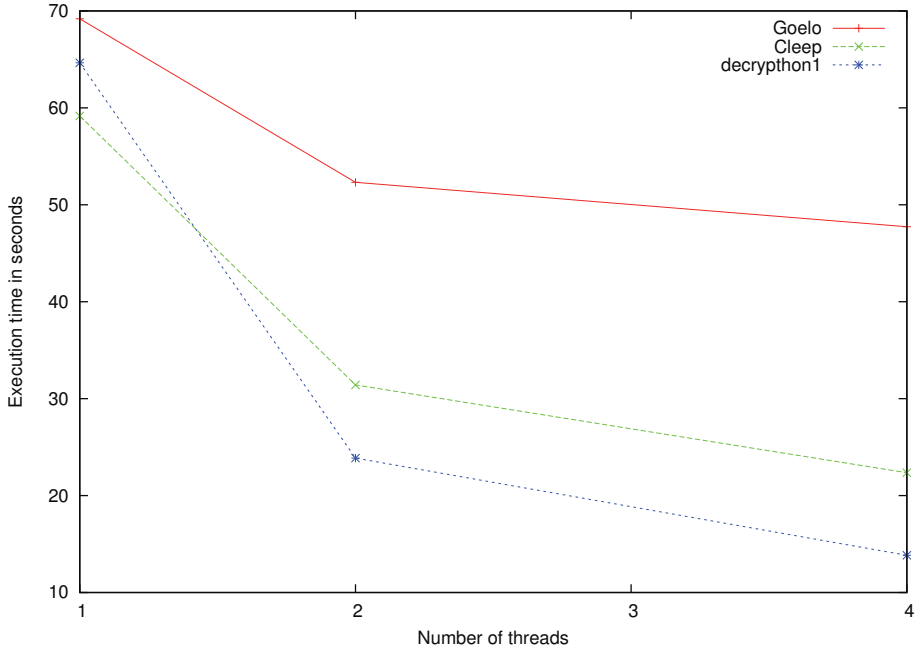


FIG. 7.21: Résultats d'exécution sur un problème contenant 184844 inconnues.

	Goelo		Cleep		decrypthon1	
Nombre de threads	Acc	Eff	Acc	Eff	Acc	Eff
1	1	-	1	-	1	-
2	1.3223	0.66	1.88	0.94	2.7083	1.35
4	1.44	0.36	2.65	0.66	4.66	1.16

FIG. 7.22: Accélérations et efficacité des codes générés sur les différentes machines. Problème ayant 184844 inconnues.

7.6.4 Modélisation et résultats d'exécution du gradient conjugué préconditionné par SSOR

L'utilisation d'un préconditionneur dans les méthodes itératives permet d'améliorer notablement la convergence du problème. Cela réduit donc le nombre d'itérations réalisées, mais peut également permettre de faire converger un problème qui divergerait sans le

préconditionnement. Il s'agit donc d'une étape indispensable afin d'obtenir une application convergente sur tous les problèmes.

Le principe du préconditionnement consiste à introduire une matrice régulière M, facilement inversible, qui soit une bonne approximation de A. Le problème à résoudre est alors $M^{-1}Au = M^{-1}b$ à la place de $Au = b$. L'utilisation des préconditionneurs dans les méthodes de résolution de type Krylov entraîne l'introduction de solveurs directs afin de résoudre le problème sur la matrice M. Nous allons étudier ici la modélisation du gradient conjugué préconditionné par la méthode SSOR. Afin d'être modélisé la forme de l'algorithme doit également être modifiée afin d'obtenir un algorithme utilisant une boucle *Tant que*, cela est effectuée comme dans le cas du gradient conjugué non préconditionné.

7.6.4.1 Modélisation

Le préconditionnement par la méthode SSOR utilise la propriété de symétrie de la matrice A pour construire M. Ainsi la matrice M est factorisée à l'aide d'une matrice triangulaire inférieur L telle que $M = LL^T$. La résolution du problème préconditionné est alors effectuée en deux étapes : une descente résolvant $Ly = b$ et une remontée résolvant $L^T x = y$. Deux étapes de calculs sont nécessaires pour un tel préconditionnement : la construction d'une matrice de préconditionnement à partir de la matrice du problème, puis lors de l'algorithme du gradient conjugué des résolutions triangulaires sont introduites. Dans notre modélisation, nous considérons que la matrice de préconditionnement est construite avant le gradient conjugué. L'algorithme du gradient conjugué préconditionné par SSOR est donné à la figure 7.23 dans lequel L représente la triangulaire inférieure issue de la factorisation de la matrice de préconditionnnement M.

1 $r_o = b - Ax_0$
2 $\tilde{r_0} = L^{-1} r_0$
3 $p_o = L^{-T} \tilde{r_0}$
4 $Pour \;\; k = 0, 1, 2, ...$
5 $\quad \alpha_k = \frac{\tilde{r_k}^T \tilde{r_k}}{p_j^T A p_j}$
6 $\quad x_{k+1} = x_k + \alpha_k p_k$
7 $\quad \tilde{r_{k+1}} = \tilde{r_k} - \alpha_k L^{-1} A p_k$
8 $\quad \beta_{k+1} = \frac{\tilde{r_{k+1}}^T \tilde{r_{k+1}}}{\tilde{r_k}^T \tilde{r_k}}$
9 $\quad p_{k+1} = L^{-T} \tilde{r_{k+1}} + \beta_{k+1} p_k$
10 $Fin \;\; Pour$

FIG. 7.23: Algorithme du gradient conjugué préconditionné par SSOR.

La parallélisation de cet algorithme repose toujours sur la parallélisation de chaque opération indépendamment des autres. Les modèles des opérations de bases du gradient conjugué sont réutilisés. Il faut alors paralléliser les résolutions triangulaires (L et L^T), ce sont les opérations prenant le plus de temps dans cet algorithme. La parallélisation de tels algorithmes s'effectue à l'aide de technique d'arbres d'éliminations. L'idée est de construire un arbre permettant de connaître de quelles inconnues dépendent les autres inconnues afin de déterminer les inconnues pouvant être résolues indépendamment des autres et donc de déterminer le parallélisme de la résolution. Ces arbres sont calculés dynamiquement selon la matrice. L'efficacité de ces techniques dépend alors fortement de la matrice, une matrice

Thèse de Julien Taillard, Lille 1, 2009

fortement couplée n'offrant que très peu de parallélisme. Ces algorithmes, dépendant de la structure, de la matrice entraînent qu'il n'est pas possible de les modéliser dans l'environnement Gaspard. Deux solutions, reposant toutes deux sur un composant élémentaire, restent alors pour modéliser l'algorithme. Ces deux solutions sont étudiées dans les paragraphes suivants.

Déploiement sur une fonction séquentielle La première solution est d'utiliser un composant élémentaire déployé sur une fonction séquentielle effectuant les résolutions triangulaires. C'est une situation classique de l'environnement Gaspard. Toutes les opérations de l'algorithme du gradient conjugué ne seront alors pas parallélisées et le gain de temps d'exécution devrait alors suivre la loi d'Amdahl.

Déploiement sur une fonction parallèle La seconde solution est d'utiliser un composant élémentaire déployé sur une fonction intrinsèquement parallèle. Cette solution, non supportée actuellement, permettrait d'obtenir un code de gradient conjugué préconditionné complètement parallèle qui devrait fournir des performances meilleures que dans le cas de la solution précédente. La méthodologie d'utilisation définie précédemment (dans le chapitre 6) était de « laisser Gaspard gérer le parallélisme en se reposant sur des fonctions élémentaires optimisées ». Cette méthodologie ne serait plus respectée dans le cadre de cette solution. L'utilisation de fonctions parallèles dans le cadre de l'environnement Gaspard est étudiée plus en détail dans le paragraphe 8.2.1 du chapitre 8.

L'utilisation de l'un ou l'autre type de déploiement sur une fonction élémentaire ne bouleversera pas la modélisation d'un gradient conjugué. Le modèle du gradient conjugué préconditionné par SSOR est présenté à la figure 7.24. On y retrouve la structure linéaire du gradient conjugué et les résolutions triangulaires associées à SSOR (L et L^T).

7.6.4.2 Résultats

Afin d'observer le comportement des temps d'exécutions du gradient conjugué préconditionné dans le seul cas actuellement supporté par l'environnement Gaspard (l'utilisation d'une fonction séquentielle), nous avons généré deux codes de gradient conjugué préconditionné par SSOR : l'un séquentiel, l'autre pour quatre processeurs. Ces codes sont générés pour résoudre le problème de magnétostatique en potentiel scalaire ayant 30880 inconnues.

L'étude des temps d'exécutions[2] du code séquentiel donne le pourcentage du temps pour les différentes parties de l'application (tableau 7.4). Comme prévu, le préconditionnement par SSOR est très coûteux puisque L et L^T totalisent 58% du temps d'exécution du code séquentiel.

La loi de Amdahl [9] prédit le gain (R) offert par le parallélisme en fonction du nombre de processeurs (N) et de la proportion d'activité parallélisable (s), celui-ci se calcule de la manière suivante :

$$R = \frac{1}{(1-s) + \frac{s}{N}} \tag{7.45}$$

[2]obtenus par profilage à l'aide de gprof

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

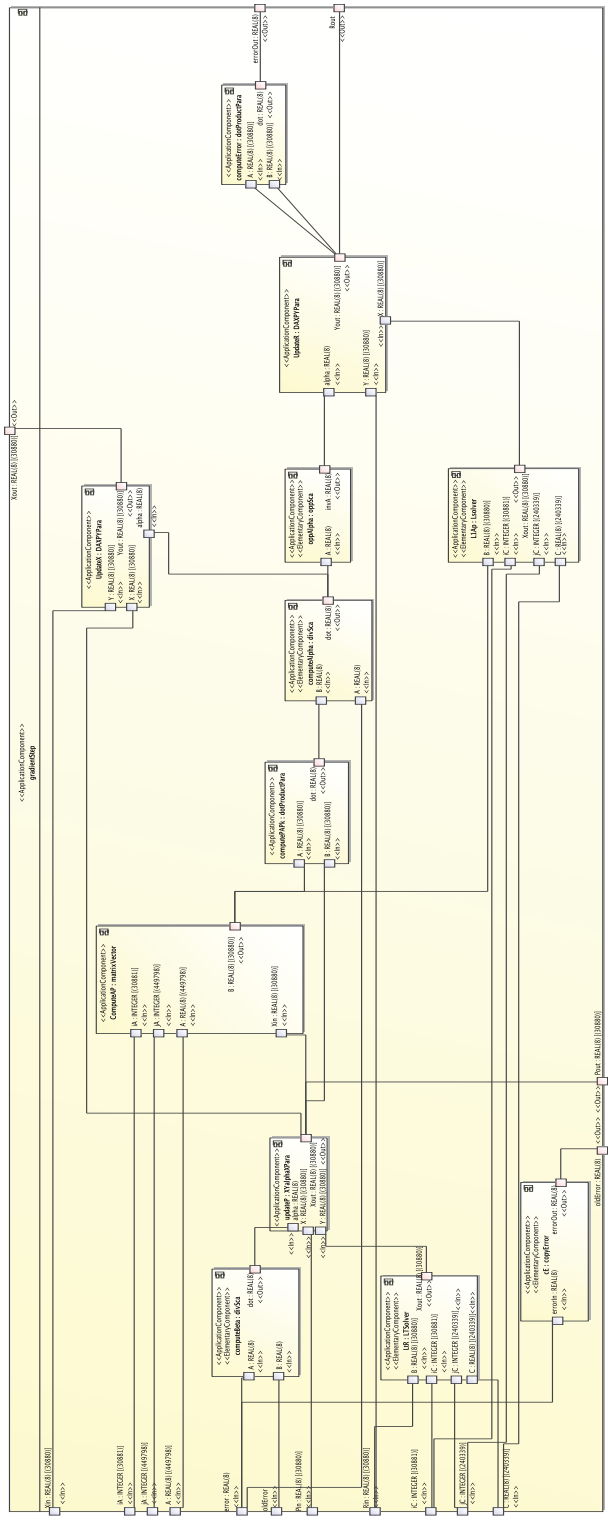


FIG. 7.24: Modélisation du gradient conjugué préconditionner par SSOR

Thèse de Julien Taillard, Lille 1, 2009

Fonction	% du temps d'exécution
Multiplication matrice-vecteur	31
L	29
L^T	29
produit scalaire	6
mise à jour de vecteurs	4
autres fonctions	1

TAB. 7.4: Proportion de temps passé dans les fonctions utilisées pour le gradient conjugué préconditionné par SSOR.

Considérons cette loi sur le code du gradient conjugué préconditionné en voulant le paralléliser sur quatre processeurs. Les fonctions pouvant être parallélisées représentent environ 41% du temps d'exécution (la multiplication matrice vecteur, le produit scalaire et la mise à jour de vecteurs). D'après la loi de Amdahl, le gain théorique maximal du parallélisme est alors de 1,44.

Les temps d'exécutions des gradients conjugués préconditonnés par SSOR sont présentés à la figure 7.5. Ces résultats ont été effectués sur la machine de bureau goelo.

Le temps d'exécution du code séquentiel est d'environ une seconde, il est comparable au temps d'exécution du gradient conjugué non préconditionné. Le temps d'exécution minimal pouvant être obtenu sans paralléliser les résolutions triangulaires est, selon la loi d'Amdahl, d'environ 0.69 s. Le temps observé sur quatre processeurs est de 0.72s. Ce temps est acceptable par rapport au temps d'exécution théorique pouvant être obtenu et montre bien l'importance de la non-parallélisation de SSOR dans cet algorithme.

Séquentiel	0.99s
4 Threads	0.72s

TAB. 7.5: Temps d'exécutions du gradient conjugué préconditionné par SSOR sur goelo.

7.7 Conclusion

Dans ce chapitre, nous avons étudié l'application de la méthodologie d'IDM définie lors de cette thèse à un code de recherche en électromagnétisme nommé CARMEL. Après avoir introduit les lois physiques régissant les phénomènes électromagnétiques dans le domaine continu et leurs discrétisations, nous avons modélisé les opérations nécessitant beaucoup de temps de calcul dans CARMEL : l'assemblage et la résolution du système. De ces modélisations ont été extraits des patrons de modélisations pour le calcul d'une intégration par la méthode de Gauss ainsi que le calcul de la contribution d'un élément à l'intégrale du domaine dans une méthode basée sur les éléments finis. L'application de ces patrons de conceptions pour l'assemblage d'un problème académique de magnétostatique a permis d'observer de bons résultats. Enfin, la modélisation de la méthode de résolution itérative du gradient conjugué a été introduite. Cette modélisation et parallélisation repose sur la parallélisation des opérations vectorielles de l'algorithme, celui-ci imposant une séquentialité entre ces opérations. Un gradient conjugué non préconditionné permet d'obtenir des per-

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

formances parallèles raisonnables. La modélisation du gradient conjugué préconditionné expose un algorithme parallèle non modélisable dans l'environnement Gaspard : la résolution triangulaire, cet algorithme reposant sur la structure de la matrice étudiée. Deux solutions à ce problème ont été proposées. Il s'agit de l'utilisation d'une tâche élémentaire réalisant ce calcul, cette tâche pouvant être séquentielle ou parallèle (cette seconde solution n'étant actuellement pas supportée par l'environnement). Le code généré effectuant la résolution triangulaire en parallèle voit ses performances limitées comme la loi de Amdahl permettait de le déterminer.

En conclusion à ces expériences, on peut en déduire que l'utilisation de l'approche modèle définie dans cette thèse est réalisable dans le cadre de calculs réguliers et pouvant être définis statiquement (il s'agit en fait des principales limitations du langage Array-OL). Les possibilités de modélisations des applications et la compilation de ces modèles sont encore à étendre vers des concepts tels que les réductions et l'utilisation de tâches élémentaires parallèles.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Chapitre 8

Vers la simulation de machines complexes

8.1 Pistes proposées par l'analyse numérique **176**
8.1.1 Utilisation d'autres méthodes de résolution 176
8.1.2 Préconditionneurs parallèles 177
8.1.3 Méthodes de décomposition de domaines 178
8.1.4 Utilisation de la FIT 180
8.2 Pistes proposées au niveau des modèles et de la compilation de ceux-ci . . **181**
8.2.1 Introduction des composants parallèles 181
8.2.2 Expression du parallélisme sur des matrices creuses 183
8.3 Conclusion **184**

Comme cela a été présenté dans le chapitre précédent, l'environnement Gaspard permet la modélisation et la génération de code réalisant l'assemblage et la résolution des systèmes linéaires résolvant les équations de Maxwell dans le cadre de la méthode des éléments finis. Bien que les performances sur l'assemblage soient satisfaisantes, l'accélération de l'algorithme du gradient conjugué est limitée par le calcul du préconditionnement qui prend une grande partie du temps d'exécution et dont la parallélisation n'est pas modélisable en Gaspard, l'algorithme pour les matrices creuses étant dépendant du graphe de dépendance de la matrice. Dans le cas de simulation prenant un long temps d'exécution, il est alors important d'améliorer encore cette accélération.

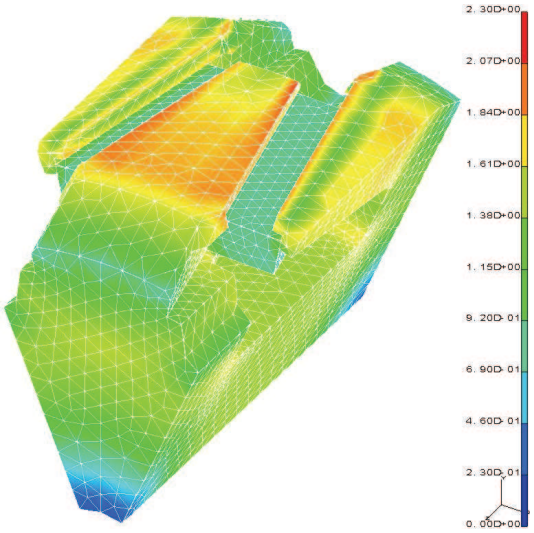
Les systèmes électromagnétiques pouvant être traités par CARMEL peuvent être complexes et nécessiter un grand nombre de calculs. Parmi ces systèmes, on peut notamment citer un alternateur à griffe dont le maillage est illustré dans la figure 8 (pour des raisons de symétrie, le domaine d'étude est ramené à un huitième de la machine). Bien que deux maillages (plus ou moins fin) aient été réalisés pour ce système, les temps de résolutions sont grands dans les deux cas. Ainsi, pour le maillage grossier (composé de 216 292 éléments et 38 699 nœuds), le temps d'exécution pour déterminer l'induction magnétique à vide pour une période électrique est d'environ 55 heures (il y a 72 pas de calcul par période). Sur un maillage plus fin (545 212 éléments et 98 545 nœuds), le temps de calcul du même problème (pour 48 pas de calcul par période électrique) monte à 139 heures[1].

[1]Résultats séquentiels provenant des travaux d'Antoine Bruyère

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

(a) Maillage d'un alternateur à griffe.



(b) Carte d'induction magnétique **B** à vide pour I = 15A.

Une amélioration de ces temps peut être obtenue avec une parallélisation de la méthode du gradient conjugué préconditionné par SSOR bien que l'accélération soit limitée par la loi d'Amdhal, le préconditionnement par SSOR étant séquentiel dans le modèle réalisé précédemment. Une meilleure parallélisation et/ou une meilleure convergence de la méthode de résolution sont alors nécessaires afin d'obtenir des temps d'exécutions réduits.

Le but de ce chapitre est d'étudier les différentes pistes (qu'elles soient dans l'analyse numérique ou dans le domaine des modèles et de la compilation de ceux-ci) qui permettraient de réduire grandement ces temps d'exécutions. Nous présenterons tout d'abord les différentes pistes provenant de l'analyse numérique pouvant aider à obtenir plus de parallélisme et à diminuer les calculs nécessaires à résoudre le problème. Ensuite, nous évoquerons l'aspect modèle avec notamment l'introduction de l'utilisation de fonctions parallèles dans l'environnement Gaspard. Enfin, nous conclurons ce chapitre.

8.1 Pistes proposées par l'analyse numérique

L'analyse numérique propose de nombreuses méthodes pouvant être utilisées dans le but de résoudre les problèmes provenant de la simulation des phénomènes électromagnétiques. Parmi celles-ci se trouvent l'utilisation d'autres méthodes de résolution (§ 8.1.1), la parallélisation du préconditonneur utilisé dans les méthodes de Krylov (§ 8.1.2) et l'utilisation des méthodes de décomposition de domaines pour la résolution, mais également pour le préconditionnement (§ 8.1.3).

8.1.1 Utilisation d'autres méthodes de résolution

Il existe un certain nombre de solveurs itératifs de type Krylov (dont le gradient conjugué fait parti). Parmi ces solveurs, certains d'entre eux pourraient donner de meilleurs résultats que le gradient conjugué. Les candidats potentiels sont notamment les algorithmes MINRES

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

(Minimum Residual Method [110]), SQMR (Symmetric quasi-minimal residual method [56]) et MRTR(Minimized Residual method based on the three-term recurrence formula of CG type [82]). Ces méthodes sont basées sur les opérations classiques que sont les produits scalaires et les mises à jour de vecteurs. Elles sont alors modélisables et parallélisables de la même façon que la méthode du gradient conjugué. La convergence de ces méthodes sur les problèmes d'électromagnétismes est proche de celle du gradient conjugué, mais on peut observer une meilleure stabilité de la convergence [133]. Le préconditionnement est également important pour la convergence de ces méthodes, la parallélisation de cette étape est donc importante.

8.1.2 Préconditionneurs parallèles

Différents préconditionneurs peuvent être utilisés afin d'améliorer notablement la convergence des problèmes. Parmi les préconditionneurs classiques figurent notamment SSOR (utilisé précédemment) qui repose sur une factorisation ou un procédé itératif (processus de relaxation) et la factorisation incomplète de Cholesky (noté IC0). Le calcul des préconditionneurs est découpé en deux phases : une phase de précalcul qui correspond à la construction de la matrice de préconditionnement et qui est effectuée une fois avant le gradient conjugué et les résolutions triangulaires qui sont effectuées à chaque itération du gradient conjugué. Ces résolutions triangulaires accaparent une majeure partie du temps d'exécution du gradient conjugué que nous avons modélisé, il est donc très important de les paralléliser afin d'obtenir de meilleures accélérations.

8.1.2.1 Les résolutions triangulaires

Les factorisations utilisées par les préconditionneurs décomposent la matrice de préconditionnement en deux matrices : une matrice triangulaire inférieure et une matrice triangulaire supérieure. Dans le cas du préconditionneur SSOR, la matrice M est factorisée en utilisant la propriété symétrique de la matrice, on a alors M=LL^T. On résout alors le système préconditionné $Mx = b$ exprimé de la manière suivante :

$$LL^T x = b \tag{8.1}$$

à l'aide de la double résolution suivante :

$$Ly = b \tag{8.2}$$

puis,

$$L^T x = y \tag{8.3}$$

Une résolution triangulaire dans le cas de matrice pleine est séquentielle. L'équation (8.4) illustre la phase de descente lors de la résolution de l'équation (8.2). La valeur de y_1 doit être déterminée avant de pouvoir calculer y_2 et ainsi de suite, une séquentialité est donc imposée.

$$\begin{pmatrix} l_{1,1} & 0 & \dots & \dots & 0 \\ l_{2,1} & l_{2,2} & \ddots & & 0 \\ \vdots & & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & 0 \\ l_{n,1} & \dots & \dots & \dots & l_{n,n} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{pmatrix} \tag{8.4}$$

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

Dans le cas de matrices creuses, les facteurs sont également creux. La valeur nulle de certains éléments entraîne l'élimination de certaines dépendances entre inconnues et il est alors possible de calculer des inconnues en parallèle. Si l'on considère que l'élément $l_{2,1}$ de L est nul, alors les valeurs des inconnues y_1 et y_2 peuvent être calculées en parallèle. L'étude de dépendance entre les inconnues est réalisée à l'aide de graphes de dépendances et d'arbres d'éliminations. La parallélisation des résolutions triangulaires repose sur ces structures de données dépendantes des valeurs de la matrice.

8.1.2.2 Parallélisation

Afin d'obtenir une bonne parallélisation de ces méthodes de résolutions triangulaires, les méthodes de renumérotation des inconnues sont utilisées afin de réduire les dépendances entre inconnues et ainsi obtenir plus de parallélisme dans les arbres d'élimination. Ces renumérotations sont utilisées dans la phase de construction de la matrice de préconditionnement. La renumérotation des inconnues est effectuée afin de réduire le remplissage de la matrice (le nombre d'éléments nuls dans la matrice A devenant non nuls) lors de la factorisation. La renumérotation peut être effectuée par les algorithmes du degré minimum [132] ou de dissections emboitées [59] pour citer les plus connus. L'algorithme du degré minimum consiste en une heuristique locale qui choisit la prochaine inconnue à éliminer en prenant un nœud du graphe de plus petit degré, cette méthode est fortement séquentielle. L'algorithme de dissections emboitées est une heuristique globale séparant le graphe en deux parties et renumérotant les inconnues et ainsi de suite de manière récursive jusqu'à ce que la taille des graphes devienne inférieure à un seuil. Cette méthode peut être parallélisée puisque les calculs sur les sous graphes sont indépendants, mais les graphes et l'expression de leurs découpages ne peuvent être modélisés dans l'environnement Gaspard.

Ces algorithmes permettent de construire des arbres d'éliminations plus courts et ayant plus de branches afin d'obtenir du parallélisme utilisé lors de la résolution directe des systèmes préconditionnés, les différentes branches de l'arbre pouvant être calculées indépendamment les unes des autres. Ce parallélisme serait alors utilisé à chaque itération du gradient conjugué. La parallélisation des résolutions directes est dépendante de la structure de la matrice, les résultats obtenus par cette parallélisation ne seraient alors pas nécessairement très bons. La modélisation de ces algorithmes parallèles n'est pas réalisable dans l'environnement Gaspard du fait de l'utilisation de structure d'arbre.

8.1.3 Méthodes de décomposition de domaines

Ces méthodes de calculs sont basées sur la décomposition du domaine en un certain nombre de sous domaines. Chaque processeur traite alors un sous domaine avant d'éventuellement réaliser des synchronisations afin de rassembler les calculs réalisés sur les différents sous domaines. Ces méthodes peuvent être divisées en deux classes : les méthodes effectuant des résolutions locales (les méthodes de Schwarz) et les méthodes effectuant une résolution globale du système (méthode du complément de Schur), les sous domaines utilisés pouvant être avec ou sans recouvrements dans les deux cas. Ces méthodes sont régulièrement utilisées dans le cadre de la programmation de machines à mémoires distribuées afin de permettre de traiter des problèmes ne pouvant pas être stockés dans une seule mémoire. Ces méthodes peuvent être utilisées en tant que solveur, mais également en tant que préconditionneur des méthodes de Krylov.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

8.1.3.1 Méthode du complément de Schur

Le problème à résoudre est $Au = b$ sur le domaine Ω. On considère que le domaine Ω est décomposé en n sous domaines Ω_1,..,Ω_n. La renumérotation des inconnues numérotant d'abord les inconnues locales aux sous domaines avant de numéroter les inconnues aux interfaces permet d'obtenir une matrice angulaire, on a alors la représentation suivante du problème :

$$\begin{pmatrix} B_1 & & & & F_1 \\ & B_2 & & & F_2 \\ & & \ddots & & \vdots \\ & & & B_n & F_n \\ E_1 & E_2 & \dots & E_n & C \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \\ g \end{pmatrix} \; avec\; u = \begin{pmatrix} x \\ y \end{pmatrix} \; et\; b = \begin{pmatrix} f \\ g \end{pmatrix} \tag{8.5}$$

dans laquelle les blocs B_i correspondent aux données locales à un sous domaine, E_i, F_i et C_i les données dépendantes des inconnues aux interfaces. Le vecteur x contient les inconnues locales aux sous domaines et y contient les inconnues aux interfaces. Pour simplifier, le problème peut être écrit sous la forme suivante :

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} f \\ g \end{pmatrix} \tag{8.6}$$

Le problème associé à un sous domaine i est :

$$\begin{pmatrix} B_i & F_i \\ E_i & C_i \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} f_i \\ g_i \end{pmatrix} \tag{8.7}$$

où le vecteur x_i contient les inconnues locales au domaine et y_i les inconnues à l'interface du domaine.

L'élimination des inconnues locales dans (8.7) conduit à l'équation suivante portant sur y_i :

$$S_i y_i = g_i - E_i B_i^{-1} F_i \; avec\; S_i = C_i - E_i B_i^{-1} F_i \tag{8.8}$$

où S_i est la forme locale du complément de Schur.

L'algorithme du complément de Schur consiste à résoudre le problème à l'interface selon l'équation (8.8) puis à résoudre le problème local avec les inconnues aux interfaces calculées de la manière suivante :

$$B_i x_i = f_i - F_i y_i \tag{8.9}$$

Le préconditionnement de la méthode du complément de Schur est réalisé sur le problème aux interfaces :

$$MSy = Mg \tag{8.10}$$

et peut être réalisé localement au domaine :

$$M_i S_i y_i = M_i g_i \tag{8.11}$$

sur le domaine i.

La matrice du complément de Schur n'étant généralement pas calculée explicitement, le système réduit aux interfaces des domaines est calculé à l'aide de méthodes itératives. Le nombre d'itérations nécessaires à résoudre le système réduit étant proche du nombre

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

d'itérations pour résoudre le système global à la différence qu'une itération pour résoudre le système réduit coûte plus cher que lors de la résolution du système global, Youssef Saad préconise alors de résoudre le système global en construisant un préconditionneur pour A à partir d'un préconditionneur pour C [123]. Il s'agit de l'approche utilisée par Emmanuel Cagniot lors de sa thèse sur la parallélisation de CARMEL [25] pour ce qui était nommé la parallélisation lourde.

L'intérêt de cette méthode dans le cadre de l'utilisation de machines à mémoires distribuées est évident puisque celle-ci permet de stocker de gros problèmes en répartissant les sous domaines sur les différentes mémoires. La méthode du complément de Schur, dans le cadre de l'utilisation d'une machine à mémoire partagée, ne permet pas de stocker des problèmes plus grands puisqu'il n'y a qu'une mémoire centrale. En revanche, le point intéressant de cette méthode est le fait que le préconditionnement peut être calculé localement à un domaine. Les processeurs vont alors pouvoir réaliser les résolutions directes (si les problèmes locaux sont suffisants petits) sur leurs domaines avant de communiquer entre eux les informations des inconnues. Les différentes résolutions seront ainsi calculées en parallèle sur les différents processeurs ce qui devrait moins affecter les performances que les résolutions triangulaires du gradient conjugué préconditionné par SSOR.

8.1.3.2 Préconditionnement par méthodes de Schwarz

Les méthodes de Schwarz (additives et multiplicatives) peuvent être utilisées en solveurs (comme précédemment avec la méthode du complément de Schur), mais du fait de leurs convergences lentes, elles sont souvent utilisées en préconditionneurs [79] des méthodes de Krylov où elles donnent de bonnes performances. Le principe des méthodes de Schwarz additives est de résoudre en parallèle les problèmes sur les sous domaines avant d'effectuer une synchronisation forte afin de communiquer les données des inconnues aux interfaces. Les méthodes de Schwarz multiplicatives sont, quant à elles, fortement séquentielles, le résultat de la résolution du premier sous domaine étant nécessaire à la résolution du second sous domaine et ainsi de suite. La méthode de parallélisation de ces méthodes consiste alors à commencer le calcul de l'itération suivante sur les domaines le pouvant durant le calcul séquentiel sur les sous domaines. Le calcul de la seconde itération sur le premier sous domaine peut être réalisé durant le calcul sur le troisième sous domaine de la première itération, on obtient ainsi un front de parallélisation.

8.1.4 Utilisation de la FIT

Nous avons étudié les problèmes de CARMEL discrétisés en utilisant la méthode des éléments finis. Cette méthode, qui est probablement la plus utilisée, n'est pas la seule méthode permettant de discrétiser et de résoudre les équations régissant les phénomènes physiques. La technique d'intégration finie (Finite Integration Technique : FIT) est une méthode alternative à la méthode des éléments finis développée en 1977 par Weiland [135] et dont l'étude et l'intégration dans CARMEL sont réalisées dans le cadre de la thèse de Julien Korecki [81]. Cette méthode est connue pour permettre une résolution plus rapide des problèmes que lors de l'utilisation de la méthode des éléments finis.

La méthode des éléments finis permet l'utilisation de maillages quelconques, ainsi les maillages réalisés dans le cadre de cette méthode sont souvent grossiers dans les parties très régulières et très raffinés dans les parties nécessitant de la précision. Cette irrégularité

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

des mailles a pour effet de créer des matrices ayant des structures creuses et irrégulières. Le maillage ne peut alors pas être représenté sous la forme d'un tableau. La FIT, quant à elle, nécessite l'utilisation d'un maillage régulier du domaine. Cette propriété permet d'envisager de travailler directement sur le maillage (pour des mailles de formes carrées ou cubiques) qui pourrait être représenté sous la forme d'un tableau multidimensionnel. L'application devrait alors être prévue pour travailler directement sur le maillage. La régularité du maillage permettrait alors de sélectionner des motifs réguliers dans le maillage à l'aide de la spécification Array-OL liée aux modèles et ainsi d'exprimer le parallélisme de l'application. La régularité du maillage est un avantage dans le cadre de l'environnement Gaspard mais est également un inconvénient puisque la régularité des mailles entraine que le domaine est, par endroits, grossièrement modélisé. Des études sont actuellement menées afin de déformer les mailles aux endroits nécessaires.

8.2 Pistes proposées au niveau des modèles et de la compilation de ceux-ci

Alors que des améliorations peuvent être réalisées afin d'utiliser d'autres méthodes d'analyse numérique, d'autres peuvent être réalisées du côté des modèles et de la compilation de ceux-ci. C'est ce qui va être présenté dans ce paragraphe. Du point de vue modèle, les modifications présentées visent principalement à obtenir plus de parallélisme dans les applications modélisées. Les modèles réalisés dans l'environnement Gaspard permettent d'exprimer le parallélisme de données réguliers basés sur des tableaux, mais cela ne permet pas d'exprimer tout le parallélisme potentiel que ce soit à l'aide d'autres types de données (des graphes ou des arbres par exemple) mais également sur des matrices creuses qui, stockées sous forme de vecteurs, ne sont pas supportées par le concept de tiler. Les modifications à apporter portent alors sur l'utilisation des fonctions parallèles dans l'environnement Gaspard et l'abstraction du stockage des matrices permettant d'obtenir plus de parallélisme potentiel.

8.2.1 Introduction des composants parallèles

Comme cela a été évoqué précédemment, de nombreux algorithmes de calcul numérique reposent sur des structures de données autres que des tableaux, les graphes et les arbres ont notamment été évoqués. Ainsi, les algorithmes de parallélisation des résolutions triangulaires effectuées pour préconditionner le gradient conjugué utilisent des arbres d'éliminations afin de déterminer les inconnues pouvant être calculées indépendamment et ainsi obtenir du parallélisme. Un arbre d'élimination étant dépendant de la structure de la matrice à laquelle il est associé, il ne peut être modélisé, c'est pourquoi il serait nécessaire d'introduire l'utilisation de composants parallèles dans l'environnement Gaspard. Cela permettrait d'avoir le maximum de parallélisme possible dans les applications modélisées. Cette introduction rendrait en partie obsolète la première méthodologie d'utilisation définie qui était de laisser l'environnement Gaspard gérer le parallélisme en utilisant des fonctions séquentielles optimisées dès que nécessaire. L'extension de l'environnement Gaspard à ce type de déploiement permettrait de réaliser tout type d'algorithme (un algorithme non modélisable serait alors réalisé sous forme de code parallèle). Dans la modélisation du déploiement sur une telle fonction, la connaissance qu'une tâche élémentaire est déployée sur du code parallèle est importante afin d'adapter le code parallèle généré (afin de synchroniser tous les processus sur cette opération

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

par exemple). Nous proposons ici une manière d'introduire dans le modèle de haut niveau des concepts autorisant l'utilisation de fonctions parallèles. La modélisation présentée doit être utilisable non seulement par la chaîne OpenMP mais également par les autres chaînes de transformations de modèles de l'environnement Gaspard. Deux étapes peuvent être dégagées de cette modélisation : la spécification qu'une tâche élémentaire est réalisée par une fonction parallèle et la distribution de celle-ci sur les processeurs. Nous allons étudier ces deux étapes.

8.2.1.1 Spécification d'une tâche parallèle

La spécification qu'une tâche élémentaire est parallèle ne peut être réalisée qu'au déploiement puisque celui-ci associe la tâche élémentaire à la fonction (séquentielle ou parallèle) l'implémentant. C'est donc lors de la spécification de la fonction qu'il faut indiquer si celle-ci est parallèle ou non. Pour cela, nous proposons un stéréotype « ParallelSoftwareImplementation » étendant « SoftwareImplementation » et indiquant si l'implémentation est parallèle. Ce stéréotype, qui est illustré à la figure 8.1, possède deux taggued values afin de permettre la compilation d'un tel modèle. La première taggued value exprime le langage parallèle utilisé dans l'implémentation parallèle. Idéalement, seul le concept de programmation de la fonction (pour mémoire partagée ou mémoire distribuée) serait nécessaire au haut niveau de la modélisation, la connaissance du langage n'intervenant qu'à la génération du code et indiquant qu'il faut initialiser et finaliser les communications MPI par exemple. La seconde taggued value indique le nombre de processeurs pour lequel est prévu la fonction (si la fonction s'adapte au nombre de processeurs disponibles, le nombre de processeurs sera indiqué en indéterminé (*)).

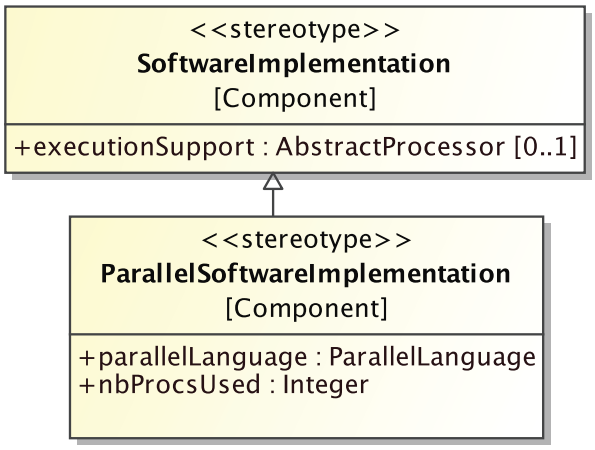


FIG. 8.1: Définition du stéréotype « ParallelSoftwareImplementation »

Le déploiement est alors réalisé normalement en utilisant le stéréotype « implementedBy ».

8.2.1.2 Distribution de la tâche parallèle

Bien que les fonctions parallèles gèrent elles-mêmes le parallélisme, il est important de spécifier la distribution de celles-ci afin de permettre la validation du modèle et/ou la simulation en SystemC mais également de réaliser éventuellement des migrations de données dans le cas d'utilisation de machines à mémoires distribuées. Pour cela, le mécanisme

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

d'allocation/distribution existant peut être utilisé afin de distribuer la tâche élémentaire sur le nombre de processeurs prévu. Le cas où la fonction parallèle s'adapte au nombre de processeurs est plus délicat à gérer puisque cela dépend de la tâche. Dans un premier temps, nous considérerons que la tâche utilise tous les processeurs. La spécification d'une telle distribution d'une tâche élémentaire parallèle est illustrée dans la figure 8.2. Considérant que la tâche de résolution triangulaire *Lsolver* est déployée sur une fonction parallèle utilisant 4 processeurs, la distribution spécifie le placement de l'instance *L1AP* sur les quatre processeurs de l'architecture (le modèle est en fait une vue simplifiée d'un pas de gradient conjugué préconditionné).

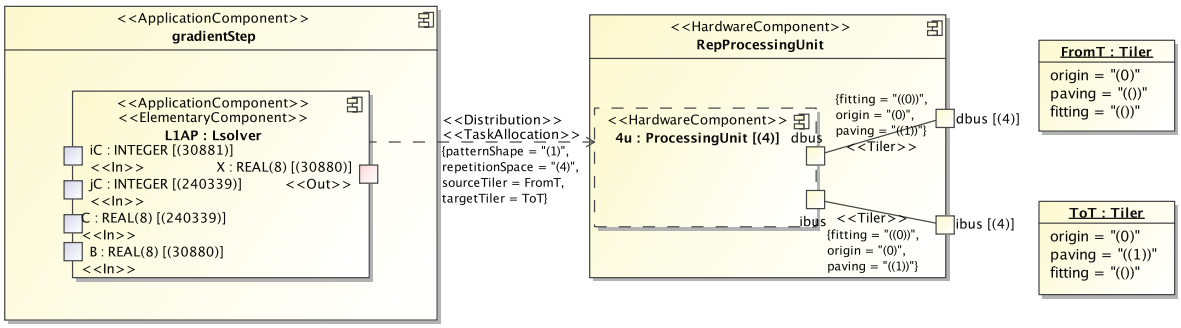


FIG. 8.2: Spécification de la distribution d'une tâche élémentaire parallèle sur les processeurs.

8.2.2 Expression du parallélisme sur des matrices creuses

Le concept mathématique de matrice est utilisé dans un grand nombre d'algorithmes numériques. L'expression de l'algorithme utilisant ce concept de matrice dépend alors du stockage de celle-ci. Dans l'environnement Gaspard, une matrice creuse est actuellement représentée comme trois vecteurs indépendants provoquant des indirections dans les différents vecteurs. La sélection de motifs dans des matrices utilisant un tel type de données n'est pas possible à l'aide de tilers puisque ceux-ci ne peuvent être utilisés que sur des matrices pleines et ne supportent pas les indirections. Le parallélisme pouvant être exprimé en utilisant ce type de matrice est alors très limité et la sélection des motifs n'est alors réalisée que dans la tâche élémentaire (comme c'est le cas dans la multiplication matrice vecteur présentée figure 7.13 à l'aide d'un index). On aimerait différencier le concept mathématique de matrice du stockage de celle-ci afin d'avoir un unique modèle dont le code généré s'adapte selon le stockage utilisé pour la matrice mais également permettre d'exprimer plus de parallélisme sur les matrices creuses. Pour cela, deux solutions sont alors envisageables. La première est de définir des structures de haut niveau des matrices dans lequel il y a du parallélisme potentiel. La seconde est plus générale et consiste à exprimer un tiler sur les matrices creuses et selon le stockage de celles-ci le tiler effectuera les calculs nécessaires afin de retrouver le motif demandé. Ces solutions sont discutées dans les paragraphes suivants.

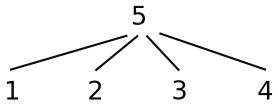
8.2.2.1 Organisation des matrices par blocs

L'organisation de la matrice par blocs peut être réalisée à l'aide d'une renumérotation des inconnues. On pourrait alors obtenir des structures de haut niveau des matrices telle qu'une matrice angulaire représentée ci-dessous :

Thèse de Julien Taillard, Lille 1, 2009

$$\begin{pmatrix} B_1 & & & F_1 \\ & B_2 & & F_2 \\ & & \ddots & \vdots \\ E_1 & E_2 & \dots & C \end{pmatrix}$$

(a)



(b)

Une telle organisation en blocs permet une représentation de la matrice globale en tableau de blocs qui pourraient alors être sélectionnés à l'aide de tilers. Le composant réalisant le produit matrice vecteur prendrait alors ce tableau de blocs représentant la matrice et le vecteur par lequel le multiplier et produirait le vecteur de sortie. La granularité du parallélisme est alors au niveau des blocs. L'arbre d'élimination de ces matrices structurées est connu et propose du parallélisme pour les résolutions directes. L'arbre d'élimination des colonnes d'une matrice angulaire ayant 4 sous domaines est illustré à la figure 8.2.2.1. Les quatre feuilles de l'arbre peuvent être calculées en parallèle.

8.2.2.2 Introduction de tilers spécifiques liés à un type de données

Une autre solution pour exprimer du parallélisme sur des matrices creuses serait de créer des tilers spécifiques à un type de données. En effet, en ayant connaissance du stockage de la matrice, il est possible de sélectionner des motifs « réguliers » dans une matrice creuse (un bloc de lignes, de colonnes ou une partie de la matrice). L'expression du tiler resterait la même qu'actuellement et c'est selon le stockage de matrice creuse utilisé que celui-ci construirait le motif. Afin de ne pas créer des motifs contenant un grand nombre de zéro, il serait nécessaire de créer des motifs creux afin de ne stocker que les données non nulles. La connaissance de la structure pleine de la matrice (i.e la taille des dimensions) serait nécessaire dans un tel cas afin de permettre le calcul du tiler. La figure 8.3 illustre le fonctionnement éventuel d'un tiler sur une matrice creuse stockée avec le format CSR et construisant un motif creux également stocké également en CSR. Un bloc de deux lignes (les lignes 3 et 4) de la matrice est sélectionné dans la matrice creuse pour l'indice de répétition 1.

L'efficacité du calcul du tiler dépend du motif accédé et du stockage de la matrice creuse. En effet alors que l'accès aux lignes dans le format CSR est rapide et simple, l'accès aux colonnes dans ce format est plus complexe puisque cela nécessite de parcourir entièrement le tableau jA à la recherche de l'indice de la colonne recherchée. La complexité de ce calcul est alors supérieure à celui de l'accès aux lignes. La performance des codes générés dépendrait alors du stockage de matrice choisi.

8.3 Conclusion

La simulation de problèmes d'électromagnétismes de grandes tailles à l'aide de l'environnement Gaspard demande encore des améliorations que ce soit du côté de l'utilisation de méthodes plus fines d'analyse numérique, mais également du côté de la modélisation. Du côté de l'analyse numérique, nous avons évoqué le changement de méthodes de résolution pour des méthodes peut être plus efficace, la parallélisation des préconditionneurs et l'utilisation de méthodes de décompositions de domaines proposant potentiellement du parallélisme exploitable dans l'environnement Gaspard. Au niveau modélisation, l'introduction de l'uti-

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

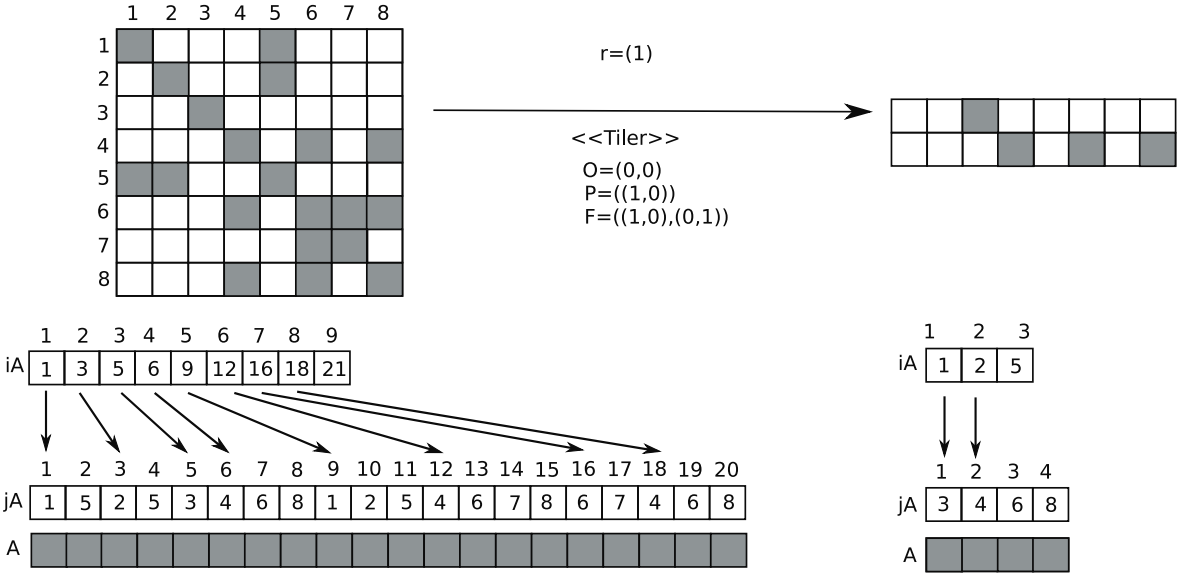


FIG. 8.3: Illustration du fonctionnement d'un tiler sur des matrices creuses.

lisation de fonctions parallèles dans l'environnement Gaspard semble indispensable afin d'autoriser l'utilisation d'algorithmes parallèles non modélisables dans les modèles. Nous avons proposé une modélisation de l'utilisation de telles fonctions parallèles, la compilation doit encore être implantée. La modélisation du parallélisme sur des matrices creuses que ce soit par une réorganisation de la matrice ou l'introduction d'un nouveau type de tiler permettrait d'obtenir plus de parallélisme.

© 2010 Tous droits réservés.

http://doc.univ-lille1.fr

Chapitre 9

Conclusions et perspectives

Résumé

Les travaux de cette thèse s'inscrivent dans le cadre du développement d'applications parallèles : de la simplification de leur spécification et de leur maintenance vers la génération de code parallèle pour l'architecture cible. Il résulte de ces travaux une méthodologie d'ingénierie dirigée par les modèles proposant la modélisation des applications à un haut niveau d'abstraction en UML avant de générer automatiquement du code parallèle pour la machine cible via des transformations de modèles.

La première contribution de cette thèse est la définition de la méthodologie dirigée par les modèles à laquelle est associé un modèle d'exécution des applications modélisées en Gaspard. Ce modèle d'exécution de type SPMD est défini en utilisant le langage OpenMP afin de gérer les machines à mémoires partagées dont les machines de bureau ayant des processeurs multi-cœurs font partie.

L'extension du profil UML Gaspard est ensuite présentée. Ce profil, originellement basé sur le langage d'expression Array-OL, vise la modélisation d'applications de traitement de signal intensif en exprimant le parallélisme potentiel de ces applications. L'extension au domaine des applications de calcul scientifique nécessite l'ajout de concepts de modélisation. L'un des concepts introduits vise à modéliser des boucles dont le domaine d'itération n'est pas connu initialement. Ce type de boucle (« Tant que ») étant souvent utilisé dans les algorithmes itératifs de résolutions de systèmes d'équations linéaires. L'ajout de la possibilité d'utiliser des paramètres en entrée/sortie est également défini afin de réduire l'utilisation mémoire du code généré et ainsi améliorer les performances. Enfin, l'ajout de l'utilisation des paramètres dans le modèle de haut niveau et sa transformation vers un modèle *exécutable* est introduit dans le but de favoriser la réutilisation des modèles.

Dans le chapitre suivant est défini un métamodèle de langage procédural à partir duquel des générateurs de code produiront des langages tels que le Fortran ou le C. Le principal but de la définition d'un tel métamodèle est la réutilisation de la même transformation de modèle pour générer différents langages. Que l'on veuille générer du C ou du Fortran la transformation générera le même modèle, seul le générateur de code ajoutera la syntaxe du langage au modèle.

Le développement de la chaîne de compilation d'un modèle de haut niveau est ensuite discuté. Cette chaîne de compilation, basée sur le concept de transformations de modèles lié à l'IDM, est décomposée en « petites » transformations. Ces « petites » transformations

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

produisent chacune un modèle dans un métamodèle intermédiaire. L'implémentation de ces transformations dépend de l'outil dans lequel elles sont développées (le moteur de transformation) et la question de savoir si tout peut être réalisé en transformations de modèles est discutée. Les résultats d'exécutions de codes générés sur différents algorithmes de l'opération classique de la multiplication de matrices sont donnés et analysés.

La méthodologie est ensuite appliquée sur une application de calcul scientifique simulant les phénomènes électromagnétiques nommée CARMEL et développée au L2EP. Cette application passant principalement son temps dans deux opérations coûteuses (l'assemblage et la résolution du système), nous nous sommes intéressés à modéliser et paralléliser ces opérations. À partir de ces modélisations, des patrons de conceptions ont été extraits pour les opérations classiques de calcul d'une méthode basée sur les éléments finis. Des modèles de la phase d'assemblage de la matrice et de la résolution du système avec la méthode du gradient conjugué ont été réalisés et des codes générés à partir de ces modèles. Les résultats montrent que l'on peut obtenir des résultats intéressants, mais que la non-parallélisation d'une des opérations de l'algorithme (la résolution triangulaire liée au préconditionnement), car non modélisable dans l'environnement, pénalise l'accélération du gradient conjugué.

Enfin, les perspectives permettant le passage à l'échelle de la méthodologie pour les simulations de phénomènes électromagnétiques complexes sont évoquées. Les changements à apporter peuvent être du côté des méthodes de l'analyse numérique mais également du côté des modèles. De nouvelles méthodes d'analyse numérique peuvent permettre d'obtenir plus de parallélisme et/ou une convergence plus rapide. Du côté des modèles, l'autorisation de l'utilisation de fonctions parallèles comme tâches élémentaires dans l'environnement Gaspard servirait à utiliser des algorithmes parallèles non modélisables dans l'environnement. Une représentation de plus haut niveau des matrices permettrait d'exprimer plus de parallélisme, la génération de code devant gérer le stockage associé à la matrice afin de générer un code valide.

Discussion

L'utilisation quasi quotidienne des outils UML et de l'approche orientée modèle introduite dans Gaspard peut amener à émettre certaines critiques. La première est que le langage Array-OL est relativement difficile à appréhender, le côté simplification de la programmation parallèle n'est pas total. La seconde porte principalement sur les outils UML et de la résolution des problèmes rencontrés.

Adaptation au langage Array-OL

La modélisation du parallélisme dans Gaspard est basée sur le langage Array-OL. L'appropriation des concepts de ce langage demande une certaine pratique. Le langage de haut niveau proposé dans le cadre de l'environnement Gaspard demande alors un certain temps afin d'être maîtrisé. Le calcul des différentes matrices (pavage et ajustage) permettant d'extraire les motifs des tableaux est une tâche complexe, fastidieuse et source d'erreurs. Cela limite pour le moment le nombre d'utilisateurs potentiels d'un environnement tel que Gaspard. L'intégration d'outils permettant la saisie graphique des tilers et de leur fonctionnement devrait être bénéfique pour améliorer ce point.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Utilisation des outils UML et résolutions des bugs

UML est un standard de modélisation recouvrant beaucoup de domaines. La spécification offre donc beaucoup de liberté permettant de modéliser les différents domaines. On retrouve cette liberté de modélisation dans les outils. L'utilisation de ces outils UML n'est pas toujours simple. De plus, UML repose sur un format soi-disant standard (le XML) afin de pouvoir échanger les modèles entre outils UML. Dans la réalité, chaque modeleur utilise son propre format de XML ce qui est décevant et réduit l'interopérabilité entre les outils.

Dans le cadre de l'environnement Gaspard, nous utilisons un nombre très restreint de notions UML mais les outils UML laissent une grande liberté. Il arrive régulièrement de faire des erreurs de modélisations (souvent liées à un clic de souris non précis) sans que cela ne soit visible graphiquement et qui a une signification pour un des domaines couverts par UML, l'utilisateur pense alors que son modèle est bon. La compilation d'un tel modèle, à travers l'environnement Gaspard, provoque une erreur, mais la résolution de ce bug est difficile puisque ne sachant pas d'où provient l'erreur (de la transformation ou du modèle), l'utilisateur doit la chercher par lui même. Le développeur de la transformation connaissant le code identifie assez rapidement le problème, mais pour un utilisateur cela sera quasiment impossible. Deux solutions indépendantes et complémentaires devraient alors être envisagées afin de limiter les erreurs et d'aider la recherche des erreurs. Ainsi, l'utilisation d'un éditeur graphique spécialisé au domaine que l'on traite serait intéressant puisque l'éditeur pourrait limiter les erreurs en empêchant des connexions entre éléments incompatibles dans le cadre de notre démarche. La disponibilité d'outils de validation du modèle de départ et/ou d'outils indiquant clairement de quel élément provient l'erreur est également indispensable pour permettre l'adoption d'une telle approche. Ces deux points devraient alors permettre une utilisation plus simple de l'environnement Gaspard.

Perspectives

De ces travaux, plusieurs perspectives peuvent être distinguées.

Optimisation et génération de code pour de nouvelles architectures

Le code OpenMP généré peut être optimisé pour gérer au mieux les processeurs multi-cœurs. Il pourrait notamment être intéressant de réaliser des copies privées des variables partagées afin que chacun des cœurs travaille sur sa copie privée et limiter ainsi les accès à la mémoire partagée.

La génération de code vers de nouvelles architectures parallèles est une perspective de ces travaux. À partir d'un même modèle de haut niveau, il est en effet intéressant de pouvoir cibler différentes architectures matérielles. Deux cibles sont actuellement intéressantes : les machines à mémoires distribuées et les cartes graphiques. Les architectures à mémoires distribuées sont intéressantes du fait qu'elles sont très répandues (du simple cluster aux machines massivement parallèles). Les cartes graphiques sont une autre cible très intéressante du fait de leur faible coût et de la puissance de calcul proposée par celles-ci. Les architectures ayant une architecture définie et très sensible à la régularité des calculs, la structure des applications modélisées devra être modifiée automatiquement afin de s'adapter au mieux à la structure des cartes graphiques. Il s'agit du sujet de thèse de Wendel Rodrigez débuté dans l'équipe-projet INRIA DaRT en octobre 2008.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

Extension du modèle Gaspard de haut niveau

Bien que certaines extensions aient été apportées au modèle de haut niveau de Gaspard afin de cibler les applications de calcul scientifique, certaines limitations existent toujours et contraignent le pouvoir d'expression des modèles. Ainsi, le concept de réduction devrait être ajouté au métamodèle afin de permettre de réaliser des modèles d'applications indépendants de l'architecture cible. Nous avons en effet pu constater, lors de modélisations d'opérations telles que l'assemblage d'un problème dans la méthode des éléments finis, que des modèles devaient être réalisés selon le nombre de processeurs utilisés afin de ne pas utiliser trop de mémoires et obtenir des performances.

Le concept de distribution des tâches sur les processeurs devrait également être amélioré afin de prendre en compte le contexte. Le problème se pose lorsque plusieurs instances d'un même composant sont utilisées. S'il est intéressant de placer les instances internes à ce composant, alors toutes les instances de ce composant auront la même distribution. Or, selon le contexte de l'application, il peut être intéressant à deux moments différents de ne pas utiliser la même distribution. Une notion de distribution contextuelle devrait donc être introduite.

Génération de code paramétré

Les codes parallèles actuellement générés sont les programmes principaux. Il pourrait être intéressant de produire des bibliothèques de fonctions parallèles à partir des modèles qui pourraient être réutilisées par les développeurs ne désirant pas utiliser toute l'approche modèle et/ou cherchant juste une fonction parallèle adaptée à leurs besoins. L'environnement de modélisation serait alors réservé à quelques spécialistes modélisant et générant les fonctions parallèles utilisables par les développeurs. La génération de code paramétré nécessite l'introduction des paramètres dans les différents métamodèles intermédiaires et la modification en conséquence des transformations pour garder ces paramètres entre les différents modèles.

Extensions pour la simulation de phénomènes électromagnétiques complexes

Nous avons évoqué dans le chapitre 8 les différentes évolutions nécessaires afin d'obtenir des performances lors de la simulation de phénomènes électromagnétiques complexes. Pour cela, il faudrait dans un premier temps implémenter la compilation de l'utilisation de fonctions parallèles puis, si les performances ne sont pas suffisantes, changer de méthodes d'analyse numérique en utilisant notamment les méthodes de décomposition de domaine proposant du parallélisme sur les sous domaines. Les performances de ces méthodes sur les architectures à mémoires partagées risquent de provoquer de nombreux conflits d'accés à la mémoire qui pourrait nécessiter d'utiliser des machines à mémoires distribuées afin d'être efficace.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Bibliographie personnelle

[p1] Julien Taillard, Frédéric Guyomarc'h, Yvonnick Le Menach, Francis Piriou, and Jean-Luc Dekeyser. Modélisation de l'assemblage d'une matrice issue de la méthode des éléments finis. In *Les méthodes numériques en Électromagnétisme (NUMELEC 2008)*, Liège, Belguim, décembre 2008.

[p2] César Olavo De Moura Filho, Anne Etien, Julien Taillard, Cédric Dumoulin, and Frédéric Guyomarc'h. Component-based models going generic : the marte case-study. Rapport de Recherche 6632, INRIA, septembre 2008.

[p3] Julien Taillard, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. OpenMP code generation based on a Model Driven Engineering approach. In *The 2008 High Performance Computing Simulation Conference (HPCS 2008)*, Nicosia, Cyprus, juin 2008.

[p4] César Olavo De Moura Filho, Julien Taillard, Frédéric Guyomarc'h and Cédric Dumoulin. Adaptation des Templates UML pour la modélisation de composants paramétrables : application à Gaspard2. In *4èmes Jounées sur l'Ingénierie Dirigée par les Modèles (IDM 08)*, Mulhouse, France, juin 2008.

[p5] Julien Taillard, Frédéric Guyomarc'h, and Jean-Luc Dekeyser. A graphical framework for high performance computing using an MDE approach. In *16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, Toulouse, France, février 2008.

[p6] Rabie Ben Atitallah, Éric Piel, Julien Taillard, Smail Niar, and Jean-Luc Dekeyser. From High Level MPSoC description to SystemC Code Generation. In *International ModEasy'07 Workshop in conjunction with Forum on specification and Design Languages (FDL'07)*, Barcelona, Spain, septembre 2007.

[p7] Pierre Boulet, Éric Piel, Philippe Marquet and Julien Taillard. Repetitive Allocation Modeling with MARTE. In *Forum on specification and design languages (FDL'07)*, Barcelona, Spain, septembre 2007.

[p8] Rabie Ben Atitallah, Pierre Boulet, Arnaud Cuccuru, Jean-Luc Dekeyser, Antoine Honoré, Ouassila Labbani, Sébastien Le Beux, Philippe Marquet, Éric Piel, Julien Taillard, and Huafeng Yu. Gaspard2 uml profile documentation. Rapport technique 0342, INRIA, septembre 2007.

[p9] Julien Tailard, Jean-Luc Dekeyser, and Francis Piriou. A design of a UML profile for meta-computing. In *14th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2006)*, Montbéliard-Sochaux, France, février 2006.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

[p10] Julien Taillard, Philippe Marquet, and Jean-Luc Dekeyser. Embedded Linux co-simulation. In *IP Based SoC Design Conference (IP-SoC 2005)*,, Grenoble, France, décembre 2005.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Bibliographie

[1] Cloog home page. http://www.cloog.org. 112, 117

[2] *Engineering and Scientific Subroutine Library Guide and Reference*. 69

[3] *Intel Math Kernel Library Reference Manual*. 69

[4] Model To Text (M2T) Home Page. http ://www.eclipse.org/modeling/m2t. 101

[5] Acceleo. Générateur de code Acceleo. http://www.acceleo.org/pages/accueil/fr. 101

[6] L. Adhianto, B. Chapman, D. Lancaster, and I. Wolton. Tools for OpenMP Application Development : The POST Project, 2000. 26

[7] Eric Allen, David Chase, Joe Hallet, Victor Lucgangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Tobin-Hochstdat. The fortress language specification version 1.0. http ://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf. 23

[8] Abdelkader Amar, Pierre Boulet, Jean-Luc Dekeyser, and Frans Theeuwen. Distributed process networks using half FIFO queues in CORBA. In *ParCo'2003, Parallel Computing*, Dresden, Germany, septembre 2003. 40

[9] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings (30)*, page 483. 171

[10] O. Babaoglu, L. Alvsi, A. Amoroso, R. Davoli, and L. G. Giachini. Paralex : An environment for parallel programming in distributed systems. In *ACM Int'l Conf. on Supercomputing (ICS'92)*, pages 178–187, Washington, DC, juillet 1992. (Also as TR UBLCS-92-4, University of Bologna, Italy). 33

[11] Ray Barriuso and Allan Knies. SHMEM user's guide for c. Technical report, Cray Research Inc., June 1994. 19

[12] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *13th IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, pages 7–16, Juan-les-Pins, september 2004. 117

[13] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam. Visualization and debugging in a heterogeneous environment. *Computer*, 26(6) :88–95, 1993. 33

[14] Rabie Ben Atitallah. *Modèles et simulation de systèmes sur puce multiprocesseurs – Estimation des performances et de la consommation d'énergie*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, décembre 2007. 56, 106

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

[15] Rabie Ben Atitallah, Pierre Boulet, Arnaud Cuccuru, Jean-Luc Dekeyser, Antoine Honoré, Ouassila Labbani, Sébastien Le Beux, Philippe Marquet, Éric Piel, Julien Taillard, and Huafeng Yu. Gaspard2 uml profile documentation. Rapport technique 0342, INRIA, septembre 2007. 41

[16] Siegfried Benker. Optimizing irregular HPF applications using Halos. In *Parallel and Distributed Processing, Proceedings of IPPS/SPDP Workshops*, volume 1586 of *Lecture Notes in Computer Science*, pages 1015–1024, San Juan, Puerto Rico, USA, avril 1999. Springer. 151

[17] Jonathan L. Bentz and Ricky A. Kendall. Parallelization of General Matrix Multiply Routines Using OpenMP. In Springer / Heidelberg, editor, *Shared Memory Parallel Programming with OpenMP*, volume 3349/2005 of *Lecture Notes in Computer Science*, pages 1–11, 2005. 133

[18] Dan Bonachea. GASNet specification. Technical Report UCB/CSD-02-1207, U.C. Berkeley, 2002. 19

[19] Borland. Qvt - o, 2007. http://www.eclipse.org/m2m/qvto/doc. 32, 125

[20] Pierre Boulet. *Outils pour la parallélisation automatique*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996. 26

[21] Pierre Boulet. Array-OL revisited, multidimensional intensive signal processing specification. Rapport de recherche RR-6113, INRIA, février 2007. 36

[22] Pierre Boulet. Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing. Rapport de recherche RR-6467, INRIA, mars 2008. 36, 63

[23] Pierre Boulet, Philippe Marquet, Éric Piel, and Julien Taillard. Repetitive Allocation Modeling with MARTE. In *Forum on specification and design languages (FDL'07)*, Barcelona, Spain, septembre 2007. Invited Paper. 42

[24] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov. The impact of multicore on math software. In B.Kagström et al., editor, *PARA 06*, volume LNCS 4699, pages 1–10, 2007. 14

[25] Emmanuel Cagniot. *Algorithmes Data-parallèles Irréguliers Appliqués à la Simulation Éléctromagnétique Tridimensionelle*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, décembre 2000. 148, 151, 180

[26] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The Cascade High Productivity Language. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60. IEEE Computer Society, avril 2004. 23

[27] Owen Callanan, David Gregg, Andy Nisbet, and Mike Peardon. High performance scientific computing using fpgas with ieee floating point and logarithmic arithmetic for lattice qcd. *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, pages 1–6, Aug. 2006. 14

[28] François Cantonnet, Yiyi Yao, Mohammed Zahran, and Tarek El-Ghazawi. Productivity analysis of the upc language. In *IEEE International Symposium on Parallel and Distributed Processing 2004 (IPDPS 2004)*, 2004. 18, 20

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

[29] Franck Cappello and Daniel Etiemble. Mpi versus mpi+openmp on ibm sp for the nas benchmarks. In *Supercomputing '00 : Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 12, Washington, DC, USA, 2000. IEEE Computer Society. 22

[30] L.M. Carvalho. *Méthodes du complément de Schur préconditionnées dans des environnements à mémoires distribuée*. PhD thesis, Institut National Polytechnique de Toulouse, 1997. 151

[31] B. Chamberlain, S. Choi, E. Lewis, C. Lin, L. Snyder, and W. Weathersby. Zpl's wysiwyg performance model. In *HIPS '98 : Proceedings of the High-Level Parallel Programming Models and Supportive Environments*, page 50, Washington, DC, USA, 1998. IEEE Computer Society. 22

[32] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3) :291–312, 2007. 15, 18

[33] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. 2004. 15, 16

[34] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the nas mg benchmark across parallel languages and architectures. In *Supercomputing '00 : Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 46, Washington, DC, USA, 2000. IEEE Computer Society. 15

[35] Bradford L. Chamberlain and Lawrence Snyder. Array language support for parallel sparse computation. In *ICS '01 : Proceedings of the 15th international conference on Supercomputing*, pages 133–145, New York, NY, USA, 2001. ACM. 22

[36] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10 : an object-oriented approach to non-uniform cluster computing. In *20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press. 23

[37] Michael Classen and Martin Griebl. Automatic code generation for distributed memory architectures in the polyotope model. In *20th International Parallel And Distributed Processing Symposium, IPDPS 2006*, apr. 26

[38] Arnaud Cuccuru. *Modélisation unifiée des aspects répétitifs dans la conception conjointe logicielle/matérielle des systèmes sur puce à hautes performances*. PhD thesis, Université des sciences et technologies de Lille, Laboratoire d'informatique fondamentale de Lille, novembre 2005. 41, 106

[39] DaRT Team LIFL/INRIA, Lille, France. Graphical Array Specification for Parallel and Distributed Computing (GASPARD2). https://gforge.inria.fr/projects/gaspard2/, 2008. 35

[40] Alain Demeure, Anne Lafarge, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, septembre 1995. 36

[41] Tata Research Development and Design Centre. Modelmorf – a model transformer. http://www.tcs-trddc.com/ModelMorf. 32

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

[42] World Wide Web document. Atlas homepage. URL : http://math-atlas.sourceforge.net/. 69, 133

[43] World Wide Web document. Message passing interface. URL : http://www-unix.mcs.anl.gov/mpi/. 98

[44] Jack Dongarra, Gene H. Golub, Eric Grosse, Cleve Moler, and Keith Moore. Netlib and na-net : Building a scientific computing community. *IEEE Annals of the History of Computing*, 30(2) :30–41, 2008. 68

[45] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. Algorithm 656 : an extended set of basic linear algebra subprograms : model implementation and test programs. *ACM Trans. Math. Softw.*, 14(1) :18–32, 1988. 133

[46] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1) :1–17, 1988. 133

[47] Nikolaos Drosinos and Nectarios Koziris. Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. *ipdps*, 01 :15a, 2004. 22

[48] Philippe Dumont. *Spécification Multidimensionnelle pour le traitement du signal systématique*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, décembre 2005. 59

[49] Eclipse Consortium. EMF. http://www.eclipse.org/emf, 2007. 28, 32

[50] Malcon Eva. *SSADM Version 4 : A User's Guide*. McGraw-Hill Publishing Co, avril 1994. 27

[51] Jean-Marie Favre. Foundations of meta-pyramids : Languages vs. metamodels – episode ii : Story of thotus the baboon1. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http ://drops.dagstuhl.de/opus/volltexte/2005/21> [date of citation : 2005-01-01]. 28

[52] Jean-Marie Favre. Foundations of model (driven) (reverse) engineering : Models – episode i : Stories of the fidus papyrus and of the solarus. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http ://drops.dagstuhl.de/opus/volltexte/2005/13> [date of citation : 2005-01-01]. 27

[53] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles, au-delà du MDA*. Hermès Science, Lavoisier, janvier 2006. 27, 28

[54] César Olavo De Moura Filho, Anne Etien, Julien Taillardand Cédric Dumoulin, and Frédéric Guyomarc'H. Component-based models going generic : the marte case-study. Rapport de recherche 6632, INRIA, 2008. 84

[55] G. Fournet. *Électromagnétisme à partir des Équations Locales*. MASSON, Paris, 1985. 144

[56] Roland W. Freund and Noel M. Nachtigal. A new krylov-subspace method for symmetric indefinite linear systems. In *Proceedings of the 14th IMACS World Congress on Computational and Applied Mathematics*, pages 1253–1256. IMACS, 1994. 177

[57] D. D. Gajski and R. Kuhn. Guest editor introduction : New VLSI-tools. *IEEE Computer*, 16(12) :11–14, décembre 1983. 35

[58] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM : Parallel Virtual Machine : A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994. 19

[59] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference, 1981. 178

[60] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab : Design and implementation. *SIAM J. Matrix Anal. Appl*, 13 :333–356, 1992. 17

[61] Calin Glitia and Pierre Boulet. High Level Loop Transformations for Multidimensional Signal Processing Embedded Applications. In *International Symposium on Systems, Architectures, MOdeling, and Simulation (SAMOS VIII)*, Samos, Greece, juillet 2008. 63

[62] Flori Glitia, Anne Etien, and Cedric Dumoulin. Traceability for an MDE Approach of Embedded System Conception. In *Fourth ECMDA Tracibility Workshop*, Berlin, Germany, juin 2008. 33

[63] Kazushige Goto and Robert van de Geijn. On Reducing TLB Misses in Matrix Multiplication. Technical Report TR-2002-55, The University of Texas at Austin, Departement of Computer Sciences, novembre 2002. FLAME Working Note #9. 69, 133

[64] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI : A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006. 19

[65] William Gropp, Ewing Lusk, David Ashton, Pavan Balaji, Darius Butinas, Ralph Butler, Anthony Chan, David Goodell, Jayesh Krishna, Rob Ross, Rajeev Thakur, and Brian Toonen. *MPICH2 User's Guide*. Mathematics and Computer Science Division, Argonne National Laboratory, apr 2008. http ://www.mcs.anl.gov/research/projects/mpich2/documentation/files/mpich2-doc-user.pdf. 19

[66] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language . *Proceedings of the IEEE*, 79(9), September 1991. 55

[67] Yun He and Chris H. Q. Ding. Mpi and openmp paradigms on cluster of smp architectures : the vacancy tracking algorithm for multi-dimensional array transposition. In *Supercomputing '02 : Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. 22

[68] Thomas Henneron. *Contribution à la prise en compte des grandeurs globales dans les problèmes d'électromagnétisme résolus avec la méthode des éléments finis*. PhD thesis, Université de Sciences et Technologies de Lille (USTL), dec 2004. 148

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

[69] Martin C. Herbordt, Tom VanCourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug DiSabello. Achieving high performance with fpga-based computing. *Computer*, 40(3) :50–57, 2007. 14

[70] High Performance Fortran Forum. High performance fortran language specification version 2.0. Rapport technique, Department of Computer Science, Rice University, 1997. 22

[71] C. S. Ierotheou, H. Jin, G. Matthews, S. P. Johnson, and R. Hood. Generating OpenMP code using an interactive parallelization environment. *Parallel Comput.*, 31(10-12) :999–1012, 2005. 26

[72] William Jalby and Ulrike Meier. Optimizing matrix operations on a parallel multiprocessor with a memory hierarchical system. In *ICPP*, pages 429–432, 1986. 57

[73] Haoqiang Jin, Michael Frumkin, and Jerry Yan. Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes. pages 440–456, 2000. 26

[74] Stephen C. Johnson. http ://dinosaur.compilertools.net/#yacc. 87

[75] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Proceedings of the Model Transformation in Practice Workshop*, Montego Bay, Jamaica, octobre 2005. 32, 125

[76] Jutta Degener. Ansi c yacc grammar, 1995. URL : http://www.lysator.liu.se/c/ANSI-C-grammar-y.html. 87

[77] Péter Kacsuk, Gábor Dózsa, and Tibor Fadgyas. Designing parallel programs by the graphical language grapnel. *Microprocess. Microprogram.*, 41(8-9) :625–643, 1996. 33

[78] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74 : Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland, août 1974. 40

[79] Guy Atenekeng Kahou, Emmanuel Kamgnia, and Bernard Philippe. An explicit formulation of the multiplicative schwarz preconditionner. Technical Report RR-5685, INRIA, may 2006. 180

[80] Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha. Parade : An openmp programming environment for smp cluster systems. In *SC '03 : Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 6, Washington, DC, USA, 2003. IEEE Computer Society. 21

[81] Julien Korecki. *Contribution au développement de la technique d'intégration finie appliquée à l'électromagnétisme*. PhD thesis, Université des sciences et technologies de Lille, Soutenance prévue en 2009. 180

[82] Koji Koyamada, Shinsuke Tamura, and Osamu Ono. *Application of MRTR Method for Solving Complex Symmetric Linear Systems*, chapter 2, pages 108–112. Systems Modeling and Simulation. Springer, jul 2007. 177

[83] Géraud Krawezik and Franck Capello. Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In *SPAA '03 : Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127, New York, NY, USA, 2003. ACM. 26

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

[84] Leslie Lamport. The parallel execution of DO loops. In *Communications of the ACM, vol. 17, n° 2*, page 83, février 1974. 25

[85] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3) :308–323, 1979. 133

[86] C. L. Lawson, R. J. Hanson, F. T. Krogh, and D. R. Kincaid. Algorithm 539 : Basic linear algebra subprograms for fortran usage [f1]. *ACM Trans. Math. Softw.*, 5(3) :324–325, 1979. 133

[87] Sébastien Le Beux. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, décembre 2007. 55

[88] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9) :1321–1336, Sep 1991. 55

[89] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, janvier 1987. 40

[90] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1) :24–35, janvier 1987. 40

[91] Feng Liu and Vipin Chaudhary. Extending OpenMP for Heterogeneous Chip Multiprocessors. volume 00, page 161, Los Alamitos, CA, USA, 2003. IEEE Computer Society. 54

[92] Zhenying Liu, Barbare Chapman, Tien-Hsiung Weng, and Oscar Hernandez. Improving the Performance of OpenMP by Array Privatization. In M.J Voss, editor, *OpenMP Shared Memory Parallel Programming*, volume LNCS 2716 of *WOMPAT 2003*, pages 244–259, 2003. 27

[93] Ewing Lusk and al. MPI-2 : Extensions to the message-passing interface. Technical report, Message Passing Interface Forum, 1997. http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm. 19

[94] Georges Marques. *Contribution à l'Estimation de l'Erreur Numérique en 3D : Application aux Problèmes d'Électromagnétisme Statique*. PhD thesis, Université des Sciences et Technologies de Lille (USTL), 2000. 148

[95] Yvonnick Le Menach. *Contribution à la modélisation numérique tridimensionnelle des systèmes électrotechniques : Prise en compte des inducteurs*. PhD thesis, Université des sciences et technologies de Lille, 1999. 148, 150

[96] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. http://drops.dagstuhl.de/opus/volltexte/2005/11. 31

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009

[97] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, octobre 2005. 32, 125

[98] Kengo Nakajima and Hiroshi Okuda. Parallel iterative solvers for unstructured grids using an openmp/mpi hybrid programming model for the geofem platform on smp cluster architectures. In *ISHPC '02 : Proceedings of the 4th International Symposium on High Performance Computing*, pages 437–448, London, UK, 2002. Springer-Verlag. 163

[99] Peter Newton and James C. Browne. The code 2.0 graphical parallel programming language. In *ICS '92 : Proceedings of the 6th international conference on Supercomputing*, pages 167–177, New York, NY, USA, 1992. ACM. 33

[100] Ton Anh Ngo. *The Role of Performance Models in Parallel Programming and Languages*. PhD thesis, University of Washington, 1997. 18

[101] Jarek Nieplocha and Bryan Carpenter. ARMCI : A portable remote memory copy library for distributed array libraries and compiler run-time systems. In J.Rolim et al., editor, *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, volume LNCS 1586, pages 533–546, San Juan, Porto Rico, april 1999. 19

[102] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays : a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94 : Proceedings of the 1994 conference on Supercomputing*, pages 340–ff., Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 21

[103] Robert W. Numrich and John Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24(2) :4–17, 2005. 20

[104] Object Management Group. *MOF Models to Text Transformation Language Final Adopted Specification*, 2007. 101

[105] Object Management Group, Inc., editor. *MOF 2.0 Core Final Adopted Specification*. http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 2003. 28

[106] Object Management Group, Inc., editor. *UML 2 Infrastructure (Final Adopted Specifcation)*. http://www.omg.org/cgi-bin/doc?ptc/2003-09-15, septembre 2003. 27, 33

[107] Object Management Group, Inc., editor. *UML 2 Superstructure (Available Specification)*. http://www.omg.org/cgi-bin/doc?ptc/2004-10-02, octobre 2004. 41

[108] Object Management Group, Inc. MOF Query / Views / Transformations. http://www.omg.org/docs/ptc/05-11-01.pdf, novembre 2005. OMG paper. 125

[109] Object Management Group, Inc. MOF Query / Views / Transformations. http://www.omg.org/docs/ptc/07-07-07.pdf, juillet 2007. OMG paper. 32

[110] Masashi Okamoto and Masamori Ihara. A new algorithm for the least-squares solution in factor analysis. *Psychometrika*, 48(4) :597–605, dec 1983. 177

[111] Jon Oldevik. *MOFScript User Guide*. 101

[112] Open SystemC Initiative. SystemC. http://www.systemc.org/, 2002. 56

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

[113] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*, 2008. http ://www.openmp.org/mp-documents/spec30.pdf. 21, 96

[114] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5) :879–899, may 2008. 14

[115] Éric Piel. *Ordonnancement de systèmes parallèles temps-réel, De la modélisation à la mise en œuvre par l'ingénierie dirigée par les modèles*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, France, décembre 2007. 56, 106

[116] Sabri Pllana and Thomas Fahringer. On customizing the uml for modeling performance-oriented applications. In *« UML » 2002 – The Unified Modeling Language*, pages 83–102. Springer-Verlag, 2002. 33

[117] Sabri Pllana and Thomas Fahringer. Performance prophet : A performance modeling and prediction tool for parallel and distributed programs. In *International Conference on Parallel Processing (ICPP-05) ; Performance Evaluation of Networks for Parallel, Cluster and Grid Computing Systems*, juin 2005. 34

[118] Sabri Pllana, Jun Qin, and Thomas Fahringer. Teuta : A tool for uml based composition of scientific grid workflows. In *in 1st Austrian Grid Symposium. Schloss Hagenberg*. Springler Verlag, 2005. 33

[119] M. Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at : www.lri.fr/∼pouzet/lucid-synchrone. 55

[120] ProMarte partners. UML Profile for MARTE, Beta 1. http://www.omg.org/cgi-bin/doc?ptc/2007-08-04, août 2007. 41

[121] J.-P. Pérez, R. Carles, and R. Fleckinger. *Électromagnétisme. Fondements et Applications.* MASSON, Paris, 1997. 144

[122] Xue rong Gong, Yong hong Sheng, Lin sheng Lu, and Ping Zhang. An improved automatic mpi code generation algorithm for distributed memory machine. In *PDCAT '06 : Proceedings of the Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 132–137, Washington, DC, USA, 2006. IEEE Computer Society. 26

[123] Yousef Saad and Masha Sosonkina. Distributed schur complement techniques for general sparse linear systems. Technical report, SIAM J. Sci. Comput, 1997. 180

[124] Youssef Saad. *Iterative methods for sparse linear systems*. PWS, 1996. 163

[125] Christian Scheidler, Lorenz Schafers, and Ottmar Kramer-Fuhrmann. TRAPPER : A graphical programming environment for industrial high-performance applications. In *Parallel Architectures and Languages Europe*, pages 403–413, 1993. 33

[126] Aniruddha G. Shet, Wael R. Elwasif, Robert J. Harrison, and David E. Bernholdt. Programmability of the HPCS Languages : A Case Study with a Quantum Chemistry Kernel. In *IEEE International Symposium on Parallel and Distributed Processing 2008 (IPDPS 2008)*, pages 1–8, 2008. 22

[127] Lawrence Snyder. *Programming Guide to ZPL*. MIT Press, Cambridge, MA, 1999. 22

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

[128] Julien Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. Thèse de doctorat, Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, décembre 2001. 59

[129] Hubert Tardieu, Arnold Rochfeld, and René Colletti. *La Méthode Merise : Principes et outils*. Editions d'Organisation, 1991. 27

[130] France Telecom. Smartqvt documentation, 2007. http ://smart-qvt.elibel.tm.fr/doc/fr.tm.elibel.smartqvt.doc/SmartQVT_documentation.pdf. 32, 125

[131] TILE64 Processor Family. http://www.tilera.com/products/processors.php. 53

[132] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *roceedings of the IEEE*, 55 :1801 – 1809, nov 1967. 178

[133] Abdelattif Tinzefte, Yvonnick Le Menach, and Francis Piriou. Iterative solvers for singular symmetric linear systems in electromagnetism low frequency. In *Thirteenth Biennial IEEE Conference on Electromagnetic Field Computation (CEFC 2008)*, may 2008. 163, 177

[134] INRIA Triskell. Kermeta. http://www.kermeta.org/. 32, 125

[135] T. Weiland. A discretization method for the solution of maxwell's equations for six-component fields. In *Electronics and Communications AEÜ*, volume 31, pages 116–120, 1977. 180

[136] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. A bidirectional transformation approach towards automatic model synchronization. In *Summer school on Generative and Transformational Techniques in Software Engineering 2007 (GTTSE'07)*, Braga, Portugal, juillet 2007. 32

[137] Huafeng Yu. *A MARTE-Based Reactive Model for Data-Parallel Intensive Processing : Transformation Toward the Synchronous Model*. PhD thesis, Université des Sciences et Technologies de Lille, 2008. 55

[138] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *FPGA '05 : Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, New York, NY, USA, 2005. ACM. 14

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Annexe A

Le métamodèle Gaspard

Cette annexe présente le métamodèle Gaspard plus précisément que durant la thèse. Les métamodèles Polyhedron et Loop utilisant la même base que ce métamodèle, sa présentation doit permettre de mieux comprendre ces métamodèles.

A.1 Partie générale

Le métamodèle Gaspard définit des parties communes à toutes les autres parties du métamodèle. Il s'agit de concepts de base communs entre ces parties. La figure A.1 définit le noyau de Gaspard. Celui-ci définit le concept d'élément nommé (NAMEDELEMENT), de type de données (que ce soit les types primitifs ou les types composés) et les concepts de matrices (MATRIX), de vecteurs (VECTOR) et de multiplicités (SHAPE) nécessaires pour exprimer les répétitions des tâches et les tilers associés.

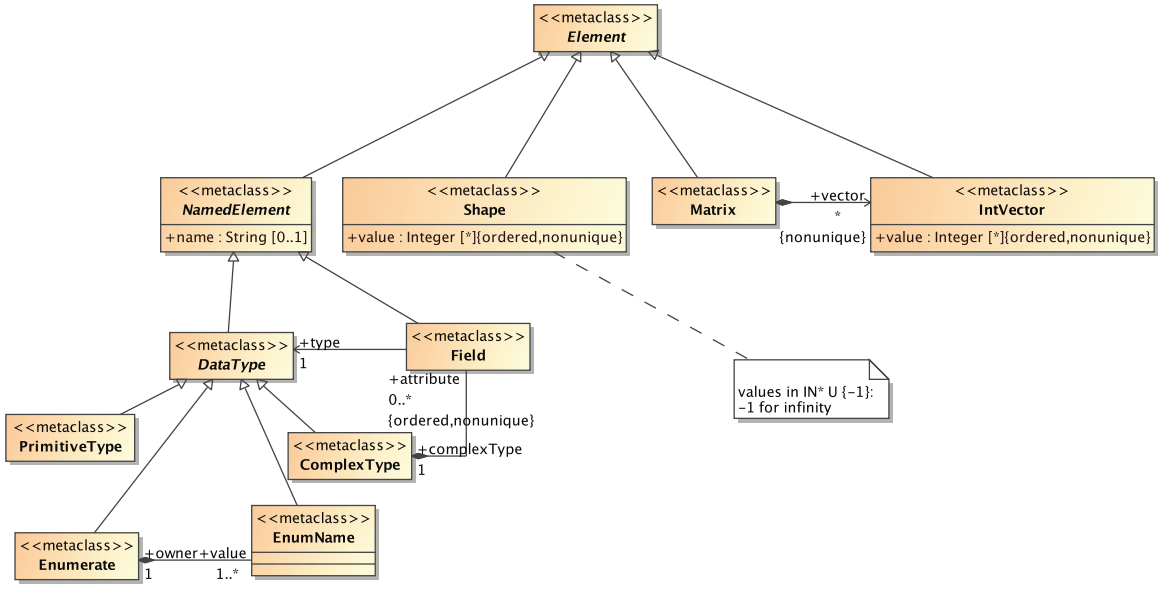


FIG. A.1: Noyau de base du le métamodèle Gaspard.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Le métamodèle Gaspard est un métamodèle basé sur les composants. Les parties applicatives et architectures des modèles utilisent la même base de composant. La figure A.2 présente la base des composants Gaspard. Un composant est ainsi un élément nommé qui contient une structure. Cette structure interne peut être élémentaire (ELEMENTARY),composé (COMPOUND) ou répétitive (REPETITIVE).

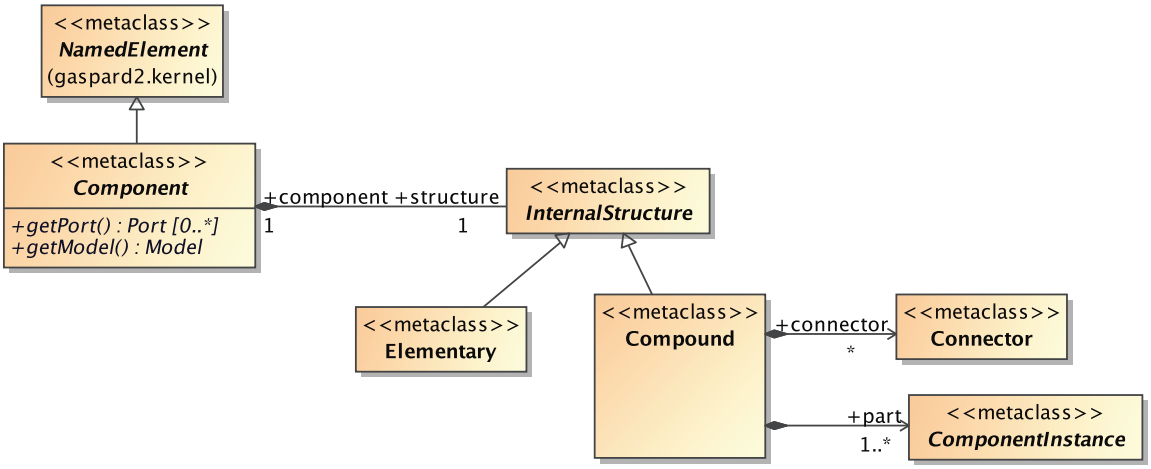


FIG. A.2: La base des composants dans le métamodèle Gaspard.

Les composants Gaspard contiennent des ports qui sont définis comme présentés à la figure A.3. Un PORT peut avoir un type de données (spécifiant le type de données circulant sur ce port) ainsi qu'une multiplicité exprimant la taille du tableau de données passant par ce port. Un port peut être spécialisé en port en entrée (INPUTPORT), en sortie (OUTPUTPORT) ou dans les deux sens (INPUTOUTPUTPORT).

A.2 Partie applicative

La partie applicative du métamodèle Gaspard est simple. Il s'agit de spécialiser un type de composant afin de pouvoir identifier facilement qu'il s'agit d'un composant applicatif. C'est pourquoi le concept de APPLICATIONCOMPONENT est introduit (figure A.4). Ce concept est spécialisé en différents types de composant applicatif. Le composant APPLICATION est le plus utilisé de ces concepts.

A.3 Partie architecture

Les composants utilisés lors de la modélisation des architectures matérielles sont illustrés à la figure A.5. Un HARDWARECOMPONENT généralise le concept COMPONENT. Ensuite ce concept est spécialisé en différents concepts permettant de représenter les différents types de matériels (Mémoire, Processeurs, Communication).

Thèse de Julien Taillard, Lille 1, 2009

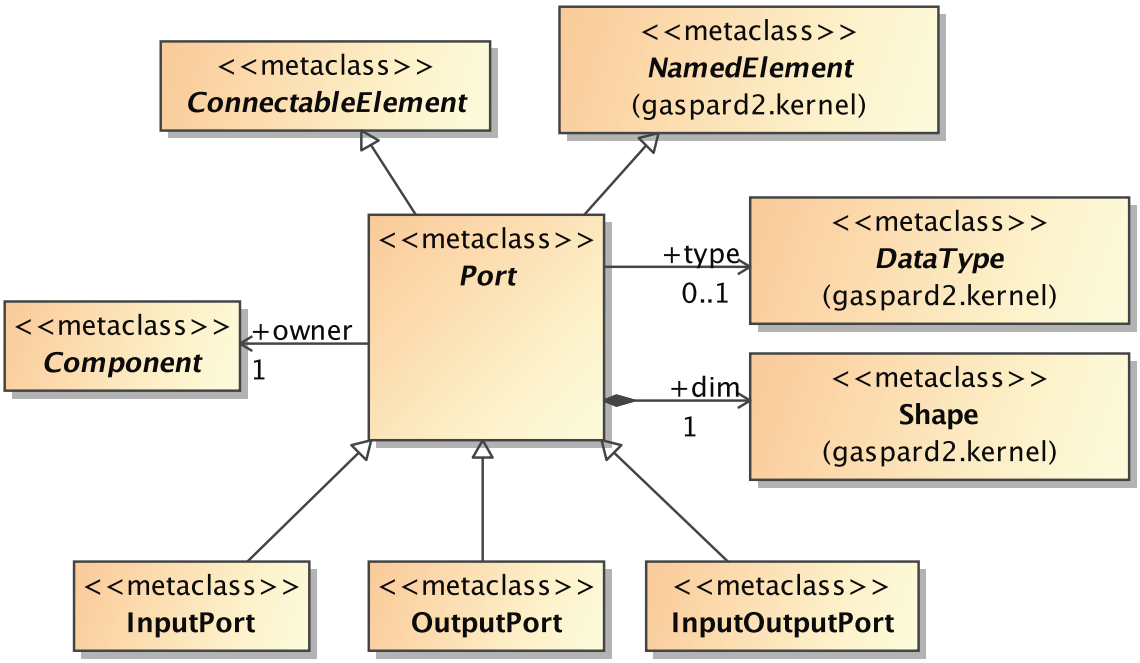


FIG. A.3: Les ports dans le métamodèle Gaspard.

A.4 Partie déploiement

Le déploiement dans le métamodèle Gaspard est le reflet des concepts présents dans le profil de déploiement UML. On y retrouve ainsi les concepts de ABSTRACTIMPLEMENTATION et d'IMPLEMENTATION qui sont spécialisés pour les parties applicatives et architectures (figure A.6).

A.5 Association

Les concepts utilisés lors de l'association sont illustrés dans la figure A.7. Le concept principal est l'ASSOCIATION qui part d'une instance de composant (*source*) vers une autre instance de composant (*target*). Ce concept est spécialisé en TASKALLOCATION et DATAALLOCATION afin de différencier le placement des tâches et des données. Un placement simple d'une instance vers une instance devient une distribution d'une instance répétée vers une instance répétée en contenant une DISTRIBUTION. Cette distribution contient les deux tilers TILINGDESCRIPTION , ainsi que la taille du motif (*patternShape*) et l'espace de répétition (*repetitionSpace*) exprimant la façon dont sont placées les répétitions de la tâche applicative sur la répétition de processeurs.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

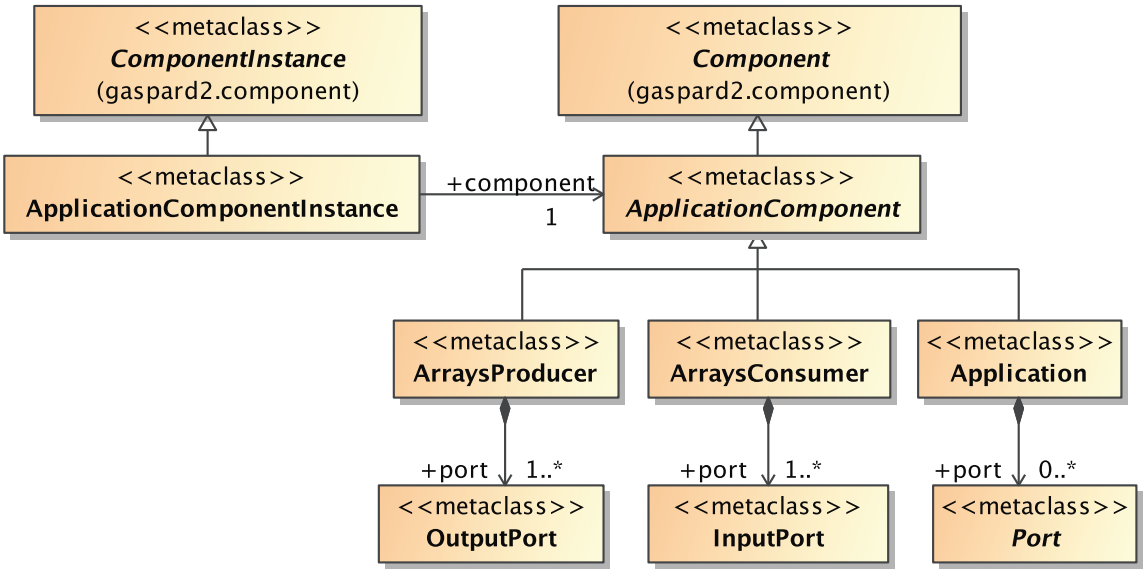


FIG. A.4: Modélisation des applications dans le métamodèle Gaspard.

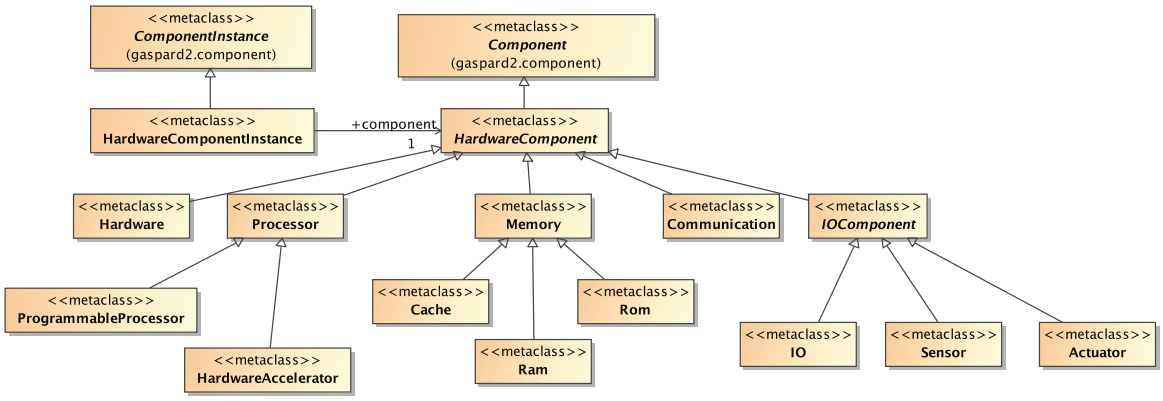


FIG. A.5: Modélisation des architectures matérielles dans le métamodèle Gaspard.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

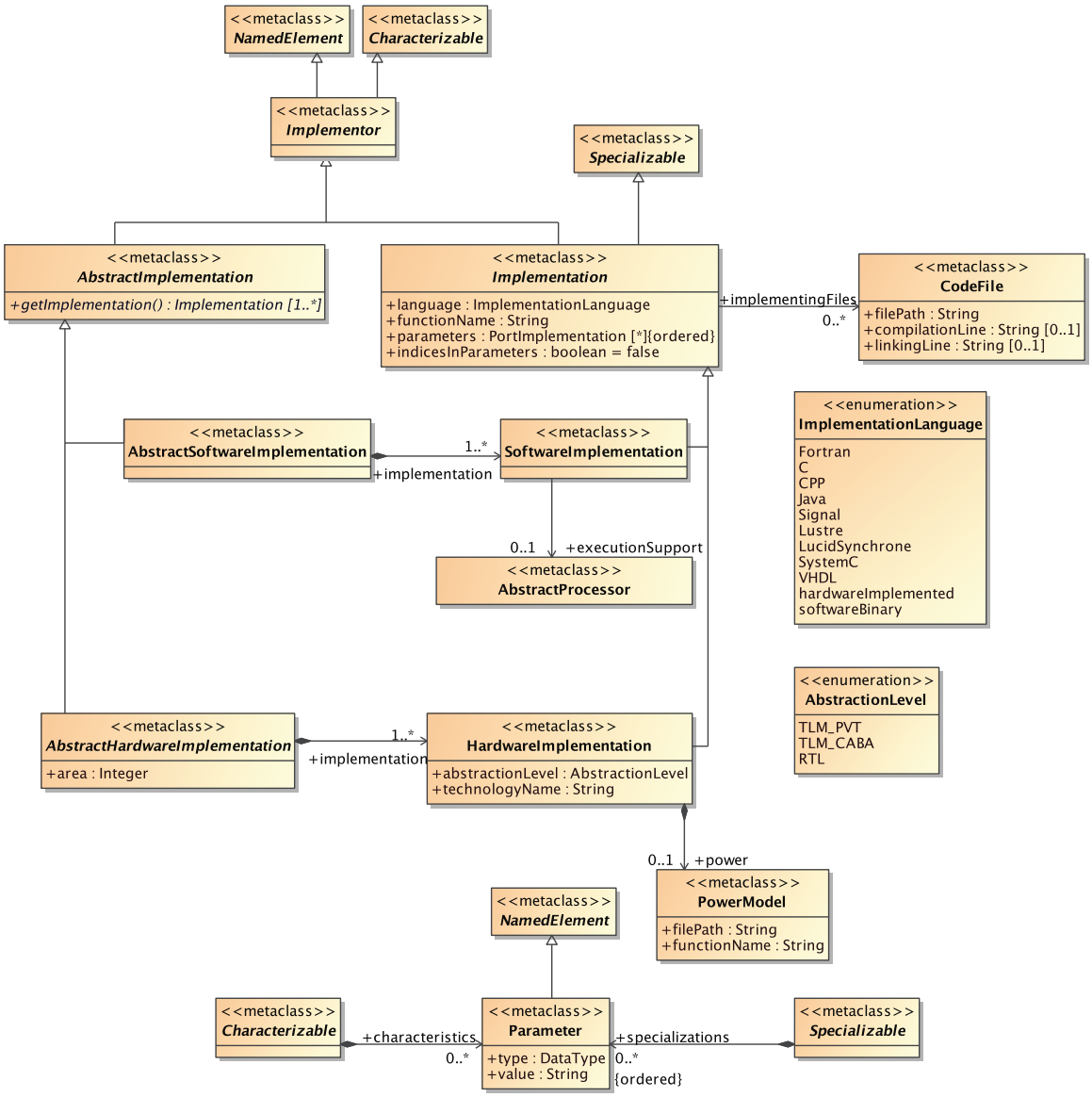


FIG. A.6: Modélisation du déploiement dans le métamodèle Gaspard.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

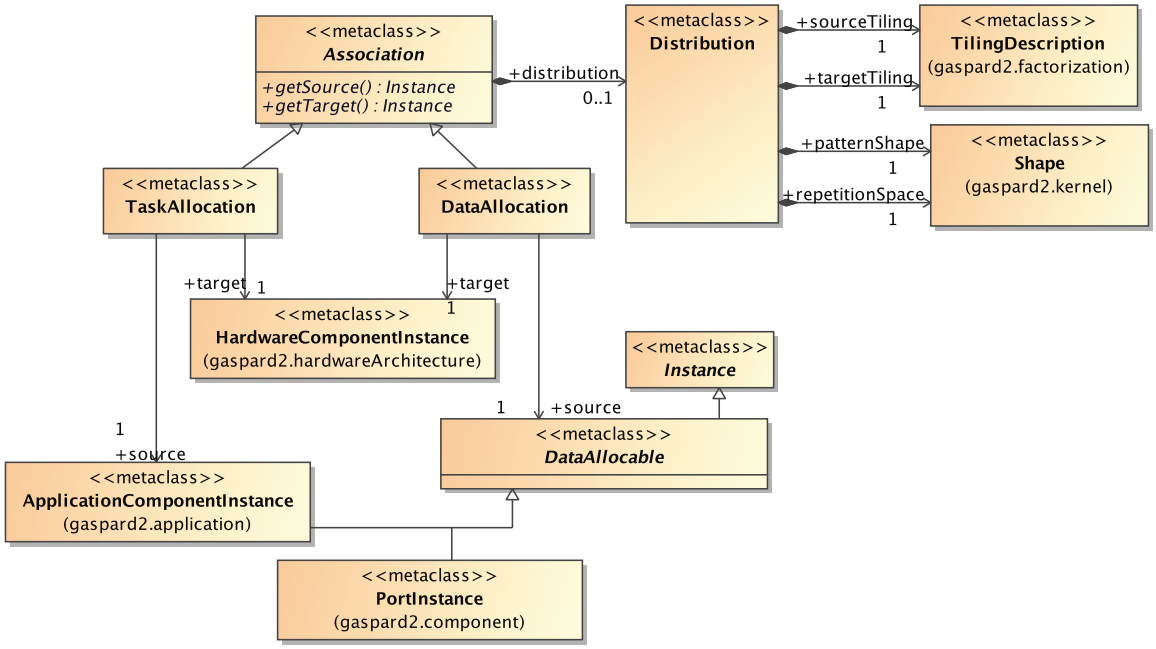


FIG. A.7: L'association et la distribution dans le métamodèle Gaspard.

© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Annexe B

Code généré pour l'algorithme du gradient conjugué

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!                             !!
!!        ConjugateGradient     !!
!!                             !!
!!                             !!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

PROGRAM ConjugateGradient

!USE!
  USE CGToolkit
  USE carmel2CSR
  USE util
      USE BLASInterface
      USE timers
! Variables declaration !

  INTEGER :: p0
  INTEGER :: OMP_GET_THREAD_NUM
  INTEGER :: x0_19580496
  REAL(8) , DIMENSION(0:30879) :: X_19580496_4954880
      !..!

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X_19580496_4954880 ,jA_19580496_17043614,A_19580496_31390655 ,B_19580496_6405019 ,
iA_19580496_6551495 ,errorOut_19580496_12444834 ,XOut_19580496_18129118 ,dot_10255813_9359099,dot_10255813_33528192 ,
Yout_10255813_1822152 ,Yout_10255813_7255679 ,Yout_10255813_28593650 ,B_2682767_15442943,dotTab_32948673_27617421 ,
dot_2682767_10615994 ,dot_2682767_9514761 ,invA_2682767_30662059 ,errorOut_22985369_20484646,Yout_22985369_4787716 ,
dot_25420205_7514753 ,B_25420205_16380850 ,dot_25420205_9717983 ,dot_25420205_3143077 ,invA_25420205_13680254)

     p0=OMP_GET_THREAD_NUM()
     x0_19580496= 0
   IF ( p0 == 0 ) THEN

     q0_9033125= 0
     d0_9033125= 0

      CALL initX(X_19580496_4954880)
   END IF


   IF ( p0 == 1 ) THEN

     q0_25285802= 0
     d0_25285802= 0

     CALL readCSRMatrix(iA_19580496_6551495,jA_19580496_17043614,A_19580496_31390655,B_19580496_6405019)
   END IF

             !$OMP BARRIER
```


© 2010 Tous droits réservés. http://doc.univ-lille1.fr

```
  x0_10255813= 0
  x0_2682767= 0
DO d0_10627996=0,7719

  x0_10627996=d0_10627996+7720*p0
  q0_10627996= p0
  originPattern_27963762(0)=0
  originPattern_27963762(0)=originPattern_27963762(0) + (1)*(x0_10627996)
CALL
   CSRdotProduct(iA_19580496_6551495,jA_19580496_17043614,A_19580496_31390655,X_19580496_4954880,&
             x0_10627996,B_2682767_15442943(originPattern_27963762(0)))

END DO

         !$OMP BARRIER
  x0_26353324= p0
  q0_26353324= p0
  d0_26353324= 0
  originPattern_7557841(0)=0
  originPattern_7557841(0)=originPattern_7557841(0) + (7720)*(x0_26353324)
  originPattern_33417151(0)=0
  originPattern_33417151(0)=originPattern_33417151(0) + (7720)*(x0_26353324)
  originPattern_4990114(0)=0
  originPattern_4990114(0)=originPattern_4990114(0) + (7720)*(x0_26353324)
CALL DAXPY7720(B_19580496_6405019(originPattern_7557841(0):originPattern_7557841(0)+7719),-1.0D0,&
              B_2682767_15442943(originPattern_33417151(0):originPattern_33417151(0)+7719),&
              Yout_10255813_1822152(originPattern_4990114(0):originPattern_4990114(0)+7719))

             !$OMP BARRIER
  x0_32948673= 0
  d0_12455534= 0
  q0_12455534= p0
  x0_12455534= p0
  originPattern_14755337(0)=0
  originPattern_14755337(0)=originPattern_14755337(0) + (7720)*(x0_12455534)
  originPattern_12344036(0)=0
  originPattern_12344036(0)=originPattern_12344036(0) + (7720)*(x0_12455534)
  originPattern_3019973(0)=0
  originPattern_3019973(0)=originPattern_3019973(0) + (1)*(x0_12455534)
CALL dotProduct7720(Yout_10255813_1822152(originPattern_12344036(0):originPattern_12344036(0)+7719),&
                 Yout_10255813_1822152(originPattern_14755337(0):originPattern_14755337(0)+7719),&
                 dotTab_32948673_27617421(originPattern_3019973(0)))

             !$OMP BARRIER
 IF ( p0 == 0 ) THEN

  q0_10876018= 0
  d0_10876018= 0
CALL sum4Tab(dotTab_32948673_27617421,dot_10255813_33528192)

 END IF

         !$OMP BARRIER
DO d0_90226=0,7719

  q0_90226= p0
  x0_90226=d0_90226+7720*p0
  originPattern_12093347(0)=0
  originPattern_12093347(0)=originPattern_12093347(0) + (1)*(x0_90226)
CALL CSRdotProduct(iA_19580496_6551495,jA_19580496_17043614,A_19580496_31390655,Yout_10255813_1822152,&
             x0_90226,B_2682767_15442943(originPattern_12093347(0)))

END DO

         !$OMP BARRIER
  x0_24559049= 0
  q0_7977946= p0
  x0_7977946= p0
  d0_7977946= 0
  originPattern_15181810(0)=0
  originPattern_15181810(0)=originPattern_15181810(0) + (7720)*(x0_7977946)
  originPattern_17976414(0)=0
  originPattern_17976414(0)=originPattern_17976414(0) + (7720)*(x0_7977946)
  originPattern_18238737(0)=0
```


© 2010 Tous droits réservés. http://doc.univ-lille1.fr

```
  originPattern_18238737(0)=originPattern_18238737(0) + (1)*(x0_7977946)
CALL dotProduct7720(B_2682767_15442943(originPattern_17976414(0):originPattern_17976414(0)+7719),&
                 Yout_10255813_1822152(originPattern_15181810(0):originPattern_15181810(0)+7719),&
                 dotTab_32948673_27617421(originPattern_18238737(0)))

             !$OMP BARRIER
 IF ( p0 == 0 ) THEN

  d0_31909984= 0
  q0_31909984= 0
CALL sum4Tab(dotTab_32948673_27617421,dot_2682767_10615994)

 END IF

         !$OMP BARRIER
 IF ( p0 == 0 ) THEN

  d0_6909598= 0
  q0_6909598= 0
CALL divSca(dot_10255813_33528192,dot_2682767_10615994,dot_2682767_9514761)

 END IF

         !$OMP BARRIER
 IF ( p0 == 0 ) THEN

  d0_4740668= 0
  q0_4740668= 0
CALL invSca(dot_2682767_9514761,invA_2682767_30662059)

 END IF


  x0_29126465= p0
  q0_29126465= p0
  d0_29126465= 0
  originPattern_26073227(0)=0
  originPattern_26073227(0)=originPattern_26073227(0) + (7720)*(x0_29126465)
  originPattern_30809147(0)=0
  originPattern_30809147(0)=originPattern_30809147(0) + (7720)*(x0_29126465)
  originPattern_13979899(0)=0
  originPattern_13979899(0)=originPattern_13979899(0) + (7720)*(x0_29126465)
CALL DAXPY7720(X_19580496_4954880(originPattern_26073227(0):originPattern_26073227(0)+7719),dot_2682767_9514761,&
              Yout_10255813_1822152(originPattern_30809147(0):originPattern_30809147(0)+7719),&
              Yout_10255813_7255679(originPattern_13979899(0):originPattern_13979899(0)+7719))

             !$OMP BARRIER
  d0_20545290= 0
  x0_20545290= p0
  q0_20545290= p0
  originPattern_30195134(0)=0
  originPattern_30195134(0)=originPattern_30195134(0) + (7720)*(x0_20545290)
  originPattern_11606232(0)=0
  originPattern_11606232(0)=originPattern_11606232(0) + (7720)*(x0_20545290)
  originPattern_12351725(0)=0
  originPattern_12351725(0)=originPattern_12351725(0) + (7720)*(x0_20545290)
CALL DAXPY7720(Yout_10255813_1822152(originPattern_11606232(0):originPattern_11606232(0)+7719),invA_2682767_30662059,&
              B_2682767_15442943(originPattern_30195134(0):originPattern_30195134(0)+7719),&
              Yout_10255813_28593650(originPattern_12351725(0):originPattern_12351725(0)+7719))
             !$OMP BARRIER
  x0_31068632= 0
  d0_17946175= 0
  q0_17946175= p0
  x0_17946175= p0
  originPattern_8803921(0)=0
  originPattern_8803921(0)=originPattern_8803921(0) + (7720)*(x0_17946175)
  originPattern_1888506(0)=0
  originPattern_1888506(0)=originPattern_1888506(0) + (7720)*(x0_17946175)
  originPattern_7861810(0)=0
  originPattern_7861810(0)=originPattern_7861810(0) + (1)*(x0_17946175)
CALL dotProduct7720(Yout_10255813_28593650(originPattern_1888506(0):originPattern_1888506(0)+7719),&
                 Yout_10255813_28593650(originPattern_8803921(0):originPattern_8803921(0)+7719),&
                 dotTab_32948673_27617421(originPattern_7861810(0)))

             !$OMP BARRIER
```


© 2010 Tous droits réservés. http://doc.univ-lille1.fr

```
 IF ( p0 == 0 ) THEN

  q0_10140736= 0
  d0_10140736= 0
CALL sum4Tab(dotTab_32948673_27617421,dot_10255813_9359099)

 END IF

         !$OMP BARRIER        !$OMP BARRIER
DO WHILE (dot_10255813_9359099 > 1.0D-12)

  x0_25420205= 0
 IF ( p0 == 0 ) THEN

  d0_29636311= 0
  q0_29636311= 0
CALL divSca(dot_10255813_9359099,dot_10255813_33528192,dot_25420205_7514753)

 END IF


 IF ( p0 == 0 ) THEN

  d0_6186238= 0
  q0_6186238= 0
CALL copySca(dot_10255813_9359099,errorOut_22985369_20484646)

 END IF

         !$OMP BARRIER
  q0_11712808= p0
  d0_11712808= 0
  x0_11712808= p0
  originPattern_6909646(0)=0
  originPattern_6909646(0)=originPattern_6909646(0) + (7720)*(x0_11712808)
  originPattern_27043516(0)=0
  originPattern_27043516(0)=originPattern_27043516(0) + (7720)*(x0_11712808)
  originPattern_10317815(0)=0
  originPattern_10317815(0)=originPattern_10317815(0) + (7720)*(x0_11712808)
CALL DAXPY7720(Yout_10255813_28593650(originPattern_6909646(0):originPattern_6909646(0)+7719),dot_25420205_7514753,&
              Yout_10255813_1822152(originPattern_27043516(0):originPattern_27043516(0)+7719),&
              B_2682767_15442943(originPattern_10317815(0):originPattern_10317815(0)+7719))
             !$OMP BARRIER
DO d0_25871309=0,7719

  x0_25871309=d0_25871309+7720*p0
  q0_25871309= p0
  originPattern_32628883(0)=0
  originPattern_32628883(0)=originPattern_32628883(0) + (1)*(x0_25871309)
CALL CSRdotProduct(iA_19580496_6551495,jA_19580496_17043614,A_19580496_31390655,B_2682767_15442943,&
                 x0_25871309,B_25420205_16380850(originPattern_32628883(0)))

END DO

         !$OMP BARRIER
  x0_16952132= 0
  x0_20730842= p0
  q0_20730842= p0
  d0_20730842= 0
  originPattern_2486573(0)=0
  originPattern_2486573(0)=originPattern_2486573(0) + (7720)*(x0_20730842)
  originPattern_26431625(0)=0
  originPattern_26431625(0)=originPattern_26431625(0) + (7720)*(x0_20730842)
  originPattern_32181625(0)=0
  originPattern_32181625(0)=originPattern_32181625(0) + (1)*(x0_20730842)
CALL
    dotProduct7720(B_25420205_16380850(originPattern_26431625(0):originPattern_26431625(0)+7719),&
              B_2682767_15442943(originPattern_2486573(0):originPattern_2486573(0)+7719),&
              dotTab_32948673_27617421(originPattern_32181625(0)))

             !$OMP BARRIER
 IF ( p0 == 0 ) THEN

  q0_7191053= 0
```


© 2010 Tous droits réservés. http://doc.univ-lille1.fr

```
  d0_7191053= 0
CALL sum4Tab(dotTab_32948673_27617421,dot_25420205_9717983)
 END IF

         !$OMP BARRIER        !$OMP BARRIER
 IF ( p0 == 0 ) THEN

  q0_18707104= 0
  d0_18707104= 0
CALL divSca(dot_10255813_9359099,dot_25420205_9717983,dot_25420205_3143077)

 END IF

         !$OMP BARRIER
  q0_7924344= p0
  d0_7924344= 0
  x0_7924344= p0
  originPattern_13694334(0)=0
  originPattern_13694334(0)=originPattern_13694334(0) + (7720)*(x0_7924344)
  originPattern_27889577(0)=0
  originPattern_27889577(0)=originPattern_27889577(0) + (7720)*(x0_7924344)
  originPattern_29680427(0)=0
  originPattern_29680427(0)=originPattern_29680427(0) + (7720)*(x0_7924344)
CALL DAXPY7720(Yout_10255813_7255679(originPattern_27889577(0):originPattern_27889577(0)+7719),dot_25420205_3143077,&
          B_2682767_15442943(originPattern_13694334(0):originPattern_13694334(0)+7719),&
          XOut_19580496_18129118(originPattern_29680427(0):originPattern_29680427(0)+7719))

 IF ( p0 == 0 ) THEN

  d0_21023874= 0
  q0_21023874= 0
CALL invSca(dot_25420205_3143077,invA_25420205_13680254)

 END IF

         !$OMP BARRIER
  q0_10626698= p0
  x0_10626698= p0
  d0_10626698= 0
  originPattern_28129784(0)=0
  originPattern_28129784(0)=originPattern_28129784(0) + (7720)*(x0_10626698)
  originPattern_10658746(0)=0
  originPattern_10658746(0)=originPattern_10658746(0) + (7720)*(x0_10626698)
  originPattern_29679045(0)=0
  originPattern_29679045(0)=originPattern_29679045(0) + (7720)*(x0_10626698)
CALL DAXPY7720(Yout_10255813_28593650(originPattern_10658746(0):originPattern_10658746(0)+7719),invA_25420205_13680254,
              B_25420205_16380850(originPattern_28129784(0):originPattern_28129784(0)+7719),&
              Yout_22985369_4787716(originPattern_29679045(0):originPattern_29679045(0)+7719))

             !$OMP BARRIER
  x0_3620953= 0
  x0_29533738= p0
  q0_29533738= p0
  d0_29533738= 0
  originPattern_30967308(0)=0
  originPattern_30967308(0)=originPattern_30967308(0) + (7720)*(x0_29533738)
  originPattern_18621340(0)=0
  originPattern_18621340(0)=originPattern_18621340(0) + (7720)*(x0_29533738)
  originPattern_32828414(0)=0
  originPattern_32828414(0)=originPattern_32828414(0) + (1)*(x0_29533738)
CALL dotProduct7720(Yout_22985369_4787716(originPattern_18621340(0):originPattern_18621340(0)+7719),&
              Yout_22985369_4787716(originPattern_30967308(0):originPattern_30967308(0)+7719),&
              dotTab_32948673_27617421(originPattern_32828414(0)))
             !$OMP BARRIER
 IF ( p0 == 0 ) THEN

  d0_32556040= 0
  q0_32556040= 0
CALL sum4Tab(dotTab_32948673_27617421,errorOut_19580496_12444834)

 END IF

         !$OMP BARRIER        !$OMP BARRIER
  dot_10255813_9359099=errorOut_19580496_12444834
```


© 2010 Tous droits réservés. http://doc.univ-lille1.fr

Thèse de Julien Taillard, Lille 1, 2009


```
    dot_10255813_33528192=errorOut_22985369_20484646
    Yout_10255813_1822152=B_2682767_15442943
    Yout_10255813_28593650=Yout_22985369_4787716
    Yout_10255813_7255679=XOut_19580496_18129118
  END DO
                !$OMP BARRIER
!$OMP END PARALLEL
```


© 2010 Tous droits réservés. http://doc.univ-lille1.fr

© 2010 Tous droits réservés.

http://doc.univ-lille1.fr

Une approche orientée modèle pour la parallélisation d'un code de calcul éléments finis

Résumé :

La généralisation des processeurs multi-cœurs rendant la plupart des machines disponibles parallèles, l'utilisation de ce parallélisme est important afin d'obtenir des performances. Les travaux de cette thèse s'inscrivent dans le cadre du développement d'applications parallèles afin de bénéficier du parallélisme des architectures d'exécution actuelles.

Dans cette thèse, nous proposons une méthodologie de développement d'applications parallèles basée sur l'ingénierie dirigée par les modèles (IDM). La spécification des applications est réalisée à l'aide du langage de modélisation UML. Après avoir défini cette méthodologie d'IDM, l'extension d'un profil UML basé sur le langage Array-OL est ensuite proposé afin d'étendre le domaine des applications modélisables aux applications de calcul scientifique.

La partie suivante est la définition d'un métamodèle de langage de procédural. Ce métamodèle est indépendant des langages cibles et permet l'utilisation de la même transformation afin de produire ces différentes cibles. Ce métamodèle autorise l'utilisation des bibliothèques parallèles (OpenMP ou MPI) dans le but de produire du code parallèle.

Ensuite la chaîne de compilation reposant sur le concept de transformations de modèles est développée et introduite dans le cadre de l'environnement Gaspard. La cible de cette compilation est les machines à mémoires partagées programmées à l'aide du langage OpenMP.

Enfin, cette méthodologie est utilisée dans le cadre du développement d'applications de simulations de phénomènes électromagnétiques. À partir de ces modélisations, des patrons de conceptions génériques de calculs classiques sont extraits. Les résultats obtenus sur les applications modélisées statiquement sont satisfaisants.

Mots clés : UML, IDM, Parallélisme, Transformations de modèle, Compilation, Électromagnétisme

A Model Driven Approach to parallelize finite element method

Abstract :

The multicore processors generalization has led to the fact that most of the available computers are parallel, and utilization of this parallelism is important to obtain performance. This thesis deals with the development of parallel applications in order to benefit the available parallelism.

In this thesis, we propose a methodology to develop parallel application based on the Model Driven Engineering (MDE). Applications specification is realized using the Unified Modeling Language (UML). After defining this methodology, an extension of the UML profile based on the Array-OL language is proposed to extend the application domain to scientific computing applications.

The following part is the definition of a metamodel of procedural languages. This metamodel, independent of the targeted languages, allows to use the same model transformations for different targets. The use of a parallel library (such as OpenMP or MPI) makes possible the production of parallel code.

Then, the transformation chain based on the model transformation concept is developped and introduced in the Gaspard framework. Shared memory machines are the target of this compilation using the OpenMP language.

Finally, this methodology is used in the development of an electromagnetism simulation. Starting from models, generic design patterns of classical computations are extracted. Results given on the modeled applications are satisfying.

Keywords : UML, MDE, Parallelism, Models transformation, Compilation, Electromagnetism

© 2010 Tous droits réservés.