

Sûreté : de l'analyse à l'instrumentation et à la synthèse de code

Corinne Ancourt

▶ **To cite this version:**

Corinne Ancourt. Sûreté : de l'analyse à l'instrumentation et à la synthèse de code. Calcul parallèle, distribué et partagé [cs.DC]. Université Pierre et Marie Curie (Paris 6), 2015. tel-01518362

HAL Id: tel-01518362

https://minesparis-psl.hal.science/tel-01518362

Submitted on 4 May 2017

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

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

Université Pierre et Marie Curie

École doctorale

SÛRETÉ: DE L'ANALYSE À L'INSTRUMENTATION ET À LA SYNTHÈSE DE CODE

Par Martine dite Corinne ANCOURT

pour obtenir

l'HABILITATION À DIRIGER DES RECHERCHES

Présentée et soutenue publiquement le 18 mai 2015

Devant un jury composé de :
Paul Feautrier, Professeur émérite à l'ENS Lyon, Rapporteur
Patrice Quinton, Professeur à l'ENS Rennes, Rapporteur
Sanjay Radopadhye, Professeur à Colorado State University, Rapporteur
Cédric Bastoul, Professeur à l'université de Strasbourg, Examinateur
Emmanuelle Encrenaz, Maître de conférences HDR à l'UPMC, Examinatrice
François Irigoin, Directeur de recherche à MINES ParisTech, Examinateur

Remerciements

Je tiens à exprimer toute ma gratitude à François Irigoin, directeur du Centre de Recherches Informatique de MINES ParisTech, pour la confiance qu'il m'a toujours témoignée. Déjà directeur de ma thèse de doctorat, il a toujours soutenu mes projets. Ses conseils scientifiques, critiques et constructifs, ont joué un rôle déterminant dans ce travail.

Je remercie vivement l'ensemble des membres du jury d'avoir accepté d'examiner ce mémoire, et plus particulièrement, les professeurs Cédric Bastoul et Sanjay Radopadhye d'avoir accepté la charge de rapporteur. J'ai apprécié leurs commentaires avisés et critiques.

Je tiens à remercier chaleureusement les professeurs Paul Feautrier et Patrice Quinton d'avoir accepté d'être examinateurs et pour l'intérêt qu'ils ont toujours porté à mes travaux.

Un grand merci à Emmanuelle Encrenaz, membre du comité scientifique de l'université Paris VI, qui a accueilli favorablement ma demande de présentation d'habilitation à diriger des recherches. Nos premiers échanges ont été encourageants et m'ont invitée à poursuivre la dynamique de mes travaux.

Le travail présenté dans ce mémoire est un travail collaboratif. Mes remerciements vont à tous les étudiants avec lesquels j'ai partagé cette aventure. J'ai apprécié nos échanges scientifiques et culturels, et leur capacité à relever les défis.

Mes derniers remerciements s'adressent à toutes les personnes qui ont contribué de près ou de loin au bon déroulement de ce travail, et en particulier à l'ensemble des membres de l'équipe du CRI pour leurs encouragements et conseils.

Table des matières

1 Logiciel sûr et parallèle **7**
- 1.1 Un peu d'histoire... . . . 7
- 1.2 Qualité d'un logiciel . . . 8
- 1.3 Les architectures parallèles . . . 9
- 1.4 Contexte de mes activités de recherche . . . 11

2 Les bons outils **15**
- 2.1 Un environnement de développement PIPS . . . 16
- 2.2 Définition : *$\mathcal{Z}$-polyèdre* . . . 19
- 2.3 Choix de l'abstraction . . . 20
- 2.4 Abstraction pour les dépendances . . . 24
- 2.5 Régions convexes de tableaux . . . 26
  - 2.5.1 Le calcul des régions de tableaux READ et WRITE . . . 27
  - 2.5.2 Le calcul des régions de tableaux IN et OUT . . . 30
- 2.6 Conclusion . . . 32

3 Sûreté par analyse, transformation et instrumentation de programmes **35**
- 3.1 Vérification des déclarations de tableaux . . . 36
- 3.2 Vérification des accès aux éléments de tableaux . . . 37
- 3.3 Détection des variables non initialisées . . . 39
- 3.4 Expériences . . . 41
  - 3.4.1 Objectif: traiter des programmes de taille réelle. . . . 42
  - 3.4.2 Vérifier les codes . . . 42
  - 3.4.3 Optimisations . . . 43
- 3.5 Conclusion . . . 45

4 Synthèse de code **47**
- 4.1 Synthèse de code à partir d'un *Z-polyèdre* . . . 49
  - 4.1.1 Exemple de transformation d'un nid de boucles: le *tiling* . . . 49
  - 4.1.2 Énumérer les points entiers d'un polyèdre . . . 53
- 4.2 Code distribué pour une mémoire partagée émulée . . . 57
- 4.3 Code distribué pour HPF . . . 62
  - 4.3.1 Formalisation des distributions HPF . . . 62
- 4.4 Optimisations des communications pour des transferts par accès direct aux mémoires (DMA) . . . 66
  - 4.4.1 Formalisation des redistributions SPEAR-DE . . . 67
  - 4.4.2 Les contraintes liées aux DMAs . . . 69
  - 4.4.3 Utilisation d'outils mathématiques . . . 70
  - 4.4.4 Algorithme de génération de code pour DMA . . . 73

4.4.5 Cas des distributions avec éléments redondants 74
4.5 Conclusion 75

5 Placement d'applications embarquées à l'aide de la programmation concurrente par contraintes 77
5.1 Une approche globale 78
5.2 Les modèles 79
5.3 Un modèle de programmation logique concurrent par contraintes (PLCC) . 82
5.4 Conclusion 84

6 Contributions et perspectives 85
6.1 Axe 1: Vérifications des applications 86
6.2 Axe 2: Synthèse de code 87
6.3 Axe 3: Cadre algébrique - le modèle polyédrique 89
6.4 Bilan 91

7 Liste de mes publications scientifiques 93

Chapitre 1

Logiciel sûr et parallèle

Old conventional wisdom : Increasing clock frequency is the primary method of improving processor performance.
New conventional wisdom : Increasing parallelism is the primary method of improving processor performance.

K. Asanovic & *al.*, *The Landscape of Parallel Computing Research : A View from Berkeley*

1.1 Un peu d'histoire...

Les premiers articles sur la sûreté des logiciels datent des **années 70**. L'informatique envahit les bureaux, les applications sont de plus en plus sophistiquées, écrites au kilomètre. Les cartes perforées tombent aux oubliettes et il est plus rapide d'écrire du code et d'observer les résultats produits que de générer un code sécurisé. Les temps de programmation sont moins longs que les temps de production du matériel, les coûts de développement du logiciel semblent négligeables comparés aux coûts de production du matériel.

Puis les **années 90** arrivent annonçant le vrai boum de l'internet [1] ! Certaines applications sont largement diffusées. Tout le monde prend conscience du danger des applications non sécurisées. Les vulnérabilités informatiques font rapidement l'objet de défis à la fois de la part des *hackers* et des fournisseurs. Les développeurs cherchent à obtenir un code *propre* et de *qualité*. Les clients souhaitent des applications sécurisées, protégées, car la vulnérabilité est grande et cause de perte d'argent et de temps. Tout le monde a encore en mémoire l'échec du lancement de la fusée Ariane V en juin 1996 lié à la reprise d'une partie du logiciel d'Ariane IV, sans nouvelle validation, et qui a coûté 6 milliards d'euros (et 10 années de travail) ou encore le *bug* de l'an 2000 qui a coûté 76 milliards d'euros à la France.

Mais le logiciel est partout ! *L'électronique embarquée représente un tiers du coût global d'un avion, 20 % de celui d'une voiture, et plus de 70 % de celui d'un téléphone mobile ! Mais surtout, ces produits s'appuient davantage sur des logiciels que du matériel. Ainsi, dans un avion, c'est le logiciel qui freine à l'atterrissage. Dans la voiture, il contrôle la*

1. Création du Web par T. Berners-Lee en 1989 et des premiers navigateurs *Mosaic, Netscape, Internet Explorer, Mozilla* à partir de 1995.

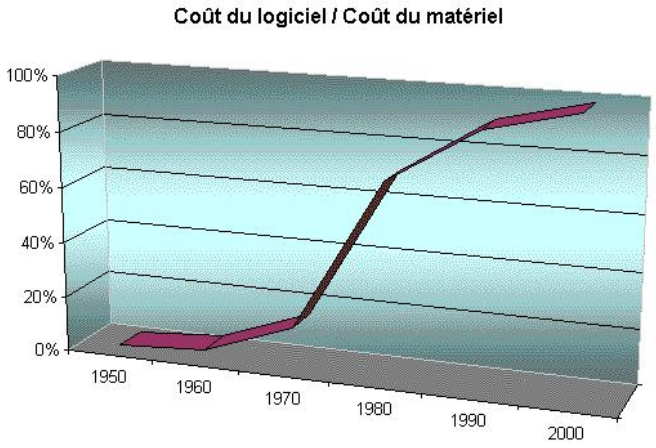


FIGURE 1.1 – Coûts logiciel vs coûts matériel

vitesse. Il se retrouve derrière les nouvelles interfaces entre l'homme et le véhicule. Il modifie les photos dans un mobile, et règle le chauffage dans les appartements... [2]

De 1965 à 1995, le volume de chaque logiciel [3] a été multiplié par 100 alors que la productivité des développeurs n'augmentait que d'un facteur 3. Le coût de développement et de maintenance du logiciel est tel que la qualité d'un logiciel est devenue une préoccupation majeure pour les chefs de projet.

Pour sécuriser les applications, il est possible d'agir à plusieurs niveaux : conception, spécification, méthode de développement, maintenance, tests unitaires jusqu'à la validation finale.

Mais qu'est-ce qu'un code de qualité ?

1.2 Qualité d'un logiciel

La certification NF Logiciels s'adresse à tous les éditeurs de logiciels. Elle valorise la qualité (basée sur la norme internationale NF EN ISO 9001) et la conformité (basée sur la norme internationale NF ISO/CEI 25051) vis-à-vis de l'utilisateur final. Il s'agit de *l'aptitude d'un produit à satisfaire les besoins des utilisateurs*. Ce critère fait donc référence à des facteurs externes, visibles de l'utilisateur. La maintenabilité du code n'en fait pas partie, or elle est essentielle pour les extensions futurs du code.

En fait, les critères de qualité d'un programme sont multiples. Si nous prenons la liste établie par B.Meyer [125] sur ces facteurs de qualité, nous avons douze critères :

1. La correction : la capacité que possède un logiciel de mener à bien sa tâche, telle qu'elle a été définie par sa spécification.
2. La robustesse : la capacité qu'offrent les systèmes logiciels à réagir de manière appropriée à la présence de conditions anormales.
3. La vérificabilité : la capacité à préparer les procédures de recette et celles permettant de détecter des erreurs et de les faire remonter, lors des phases de validation et d'opération, aux défauts dont elles proviennent.

2. Source 01.net du 1er février 2006.
3. http ://tisserant.org/cours/qualite-logiciel/qualite_logiciel.html

4. L'efficacité : la capacité d'un logiciel à utiliser le minimum de ressources matérielles, que ce soit le temps machine, l'espace occupé en mémoire externe et interne, ou la bande passante des moyens de communication.
5. L'extensibilité : la facilité d'adaptation des logiciels aux changements de spécifications.
6. La réutilisabilité : la capacité des éléments logiciels à servir à la construction de plusieurs applications différentes.
7. La compatibilité : la facilité avec laquelle les éléments logiciels peuvent être combinés à d'autres.
8. La portabilité : la facilité avec laquelle le logiciel peut être transféré d'un environnement logiciel ou matériel à un autre.
9. L'intégrité : la capacité à protéger les divers composants contre les accès et modifications non autorisés.
10. La facilité d'utilisation : la facilité que possède le logiciel de pouvoir être appris, utilisé pour résoudre des problèmes. Elle recouvre la facilité d'installation, d'opération et de contrôle.
11. La fonctionnalité : l'étendue des possibilités offertes par un système.
12. La ponctualité : la capacité d'un logiciel à être livré au moment désiré par ses utilisateurs.

Certains critères communément utilisés en sont dérivés par combinaison de ces derniers.

13. La modularité est la qualité qu'un logiciel a d'être décomposable en éléments indépendants les uns des autres et répondants à un certain nombre de critères et de principes. Elle couvre à la fois la réutilisabilité, l'extensibilité, la vérificabilité et la maintenabilité.
14. La performance dérive de l'efficacité et de la portabilité.
15. La maintenabilité est l'union des critères : facilité d'utilisation, réutilisabilité, extensibilité et vérificabilité.
16. Le respect des normes des langages et des règles d'écriture fait partie du respect des spécifications.

L'un des objectifs de mes projets de recherche a été le développement d'outils d'aide à la conception d'applications *sûres, de qualité*. J'ai participé au développement de techniques permettant de valider le code des applications par rapport à la norme du langage et de le rendre plus robuste et maintenable. Puis j'ai développé des techniques permettant de générer automatiquement des codes de calcul et de communications, corrects par construction, à partir d'une spécification mathématique des problèmes rencontrés lors de leur compilation pour des architectures parallèles. J'introduis très brièvement ce type d'architectures dans la section suivante.

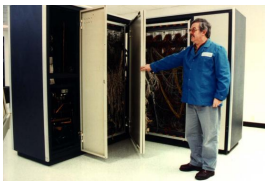
1.3 Les architectures parallèles

Deux phénomènes expliquent l'engouement pour les machines parallèle dès les années 1950. Tout d'abord, un fort besoin de puissance de calcul pour les applications de calcul intensif et de simulation, très gourmandes en temps et en mémoire, dans des domaines très variés tels que le transport, les simulations nucléaires, la visualisation 3D, les prédictions météorologiques, la santé (calcul du génome humain)... Les ordinateurs des années 1960

n'étant pas suffisamment puissants pour répondre à ce type de problème, des machines volumineuses et coûteuses se développent : *les supercomputers*.

A titre de comparaison avec nos architectures *parallèles* actuelles, voici quelques supercalculateurs connus précurseurs de ce type d'architecture :

En 1964, le supercalculateur RISC `CDC 6600` est conçu par Seymour Cray. Il possédait 10 unités de calcul arithmétique indépendantes pour une puissance de 4.58 MFlops. Il est refroidi au fréon (prix 1.000.000 $).



La partie centrale du CDC 6600.

En 1965, Burrough sort l'ILLIAC IV un supercalculateur qui possède une architecture **vectorielle**, composée de 64 processeurs **pipelinés**. Ses performances réelles atteignaient 15 MFlops.



L'Illiac IV

En 1973, Mikhail Kartsev lance le M-10 un **multiprocesseur** d'une puissance maximale de 20 à 30 MIPS[4]. Il sera suivi en 1976 du M-13 d'une puissance crête comprise entre 50 et 200 MIPS.



Le M13

En 1976, le Cray-1 est un supercalculateur de type **vectoriel**. Il dispose d'un processeur 64 bits cadencé à 83 MHertz. Ses performances crêtes sont de 166 MFlops pour un prix de 700 000$.



Le Cray 1

En 1982, le Cray X-MP est le premier **multiprocesseur** de Cray. Chacun des 4 processeurs est basé sur une architecture **vectorielle MIMD**. Sa performance crête est de 842 MFlops, pour un coût de 14 600 000 $.



Le Cray X-MP

Aujourd'hui le plus gros multiprocesseurs est surnommé ***Tianhe-2***. Cette machine peut effectuer **33,86 Pflops**. C'est un assemblage de 32 000 prises Intel Ivy Bridge et de 48 000 Intel Xeon Phi pour un total de 3 120 000 cœurs !



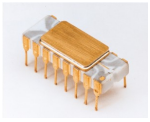
Le Tianhe-2

Au cours de la même période, les microprocesseurs évoluaient au rythme de la loi de Moore. En 1965, G. Moore [63] conjecturait que le nombre de transistors des semiconducteurs proposés en entrée de gamme doublerait tous les ans. Cette loi s'est vérifiée jusqu'à aujourd'hui. Plusieurs facteurs ont joué en sa faveur, les progrès sur la fréquence d'horloge, la finesse de la gravure, l'intégration des transistors, .. Ces évolutions ont permis

4. MIPS=millions d'instructions effectuées par le processeur par seconde

la création des premiers micro-ordinateurs en 1972. Un microprocesseur des années 1971 aurait été en 2010, 100 fois plus petit en taille pour une largeur de données équivalente.

Concernant l'évolution des puissances de calcul, le processeur Intel Core i7 (Gulftown- 147 600 MIPS - commercialisé en 2010) exécute 2 460 000 fois plus d'instructions à la seconde que le processeur Intel 4004 (1971 - 0,06 MIPS) et possède 510 000 fois plus de transistors.



L'Intel 4004



L'Intel Core i7

Toutefois en augmentant la finesse de gravure, on se rapproche des limites de la physique classique (problème connu sous le nom de *Power Wall* [92]), et on entre dans le monde de la nanotechnologie dominée par la physique quantique. Il sera difficile ou très coûteux de gagner à nouveau sur ce facteur. L'augmentation d'un facteur 2 entre deux générations de processeurs avec des applications mono-thread ne semble plus possible maintenant ; le recours aux **traitements parallèles** est devenu nécessaire. En fait, très tôt les concepts du parallélisme introduits dans les supercalculateurs, et soulignés précédemment, seront repris dans les microprocesseurs et dès 2005 les constructeurs sortent les premiers microprocesseurs **multicœurs** destinés au grand public.

Désormais, les architectures parallèles ne sont plus réservées aux industriels ayant des besoins énormes en simulation ou possédant des applications de calcul intensif, elles ont envahi notre quotidien et sont présentes dans nos téléphones, tablettes, appareils photo, robots ménagers... Ceci a un impact extrêmement important sur le logiciel qui doit être adapté pour bénéficier des ressources parallèles de ces architectures. J'ajouterai donc un critère supplémentaire de qualité d'une application, essentiel aux nouvelles architectures parallèles :

17. La capacité d'un logiciel ou d'une application à pouvoir être parallélisé.

Si ce critère n'est pas pris en compte dès la conception du logiciel, son efficacité et sa portabilité pourraient s'avérer très mauvaises.

1.4 Contexte de mes activités de recherche

Différents types de méthodes formelles [76, 70] ont été développées pour démontrer la correction de programme par rapport à un ensemble de spécifications données.

— Le *model checking* [64] analyse exhaustivement le graphe d'états du système et vérifie qu'un état indésirable donné ne figure pas parmi ces états. Cette méthode est très précise mais souffre d'explosion combinatoire lorsque les systèmes sont trop grands.

— L'analyse statique permet de trouver et valider des propriétés sur les variables d'un programme sans l'exécuter. Elle utilise des abstractions [5] et des approximations. Cependant elle peut donner d'excellents résultats sur des problèmes concrets traitant de larges applications. A titre d'exemple, Alain Deutsch a créé en 1998 la société PolySpace [80] chargée du développement industriel du prototype avec lequel il a été capable de trouver de manière automatique le bug informatique ayant amené à la destruction du premier vol d'Ariane 5. Ce prototype était un analyseur de code statique basé sur l'interprétation abstraite de code source ADA.

5. Abstraction = Représentation simplifiée d'un ensemble complexe

— La preuve automatique [5, 79], et les assistants de preuve [70, 108] simplifient l'écriture formelle de preuves et leur vérification. Les assistants nécessitent une aide manuelle pour une partie de la preuve.
— Enfin les tests permettent à tous les niveaux du développement, la validation de ces critères.

Afin de pouvoir traiter des applications de taille réelle, mes travaux comprennent 1) **des analyses** statiques et dynamiques de programmes, 2) de **l'optimisation** de programme en vue de leur exécution efficace pour des architectures parallèles et 3) de la **génération automatique de code** de contrôle et des communications à partir de spécifications, de **la synthèse de code**. Ces trois thèmes constituent des étapes classiques d'un compilateur.

Le rôle du compilateur est de transformer un code écrit dans un langage de programmation (langage source) en un autre code écrit le plus souvent dans un autre langage (langage cible). A titre d'exemple, le compilateur `GCC` transforme un programme écrit en langage `C` en langage binaire exécutable pour une machine donnée.

Les différentes étapes de la compilation sont l'analyse, l'optimisation et la génération de code :

— Les analyses syntaxiques permettent de vérifier la conformité du programme par rapport à la norme du langage choisi et aux spécifications.
— Les analyses sémantiques permettent d'extraire des informations sur les variables du programme afin de valider certaines propriétés et appliquer ensuite des transformations en vue de son optimisation, par exemple sa parallélisation. Ces analyses relèvent des critères de correction (1) et de robustesse (2).
— Les phases de transformation et d'optimisation peuvent être très variées et dépendent de différents critères : du coût d'exécution, ducompromis espace/temps et du type de code cible souhaité. Elles requièrent des informations spécifiques permettant d'établir la légalité de l'application de ces transformations et leur profitabilité.
— La génération de code dépend de l'architecture cible. Pour les machines parallèles on peut considérer deux catégories de code à générer :
  1. le code de l'application optimisée/parallélisée pouvant s'exécuter sur les différents processeurs,
  2. le code de communication et toutes les extensions nécessaires à la gestion de l'exécution en parallèle.

Quelle que soit l'architecture, l'objectif est de trouver une solution d'ordonnancement des calculs et des communications légale et efficace.

Dans le cadre de mes travaux, j'ai développé des techniques qui permettent de calculer automatiquement des informations nécessaires à la vérification des applications telles que la mise en conformité par rapport à la norme du langage source, la détection de variables non initialisées, la vérification du non-débordement des accès à des éléments de tableaux, le calcul de dépendances de données, le calcul d'invariants. Ces analyses ont été intégrées dans le compilateur `PIPS` développé au CRI (Centre de recherche en Informatique) depuis 1988. La section 3 présente ces résultats, dont l'objectif principal était la correction et la maintenance de code.

La section 4 reprend mes résultats sur la synthèse de code et notamment ceux liés à la génération automatique de codes de communication. Elle détaille les modélisations du placement des données sur les processeurs, celles des calculs pouvant être exécutés en parallèle, et celles des communications devant être générées pour conserver la cohérence des données. Enfin, les algorithmes de synthèse de code spécifiquement développés pour

`HPF`, pour une mémoire virtuelle partagée émulée sur une architecture distribuée et pour des transferts compatibles avec des DMAs sont exposés.

J'ai contribué également à la modélisation des contraintes devant être respectées pour un placement efficace d'applications embarquées temps-réel sur une architecture parallèle. Il s'agit entre autre d'une estimation :

— du nombre d'instructions devant être exécutées par un ensemble d'instructions du programme,
— de la mémoire nécessaire à l'exécution d'un ensemble d'instructions.
— du volume de communications entre deux *tâches parallèles*.
— ...

L'approche globale du problème de placement d'une application sur une architecture embarquée et les contraintes qu'il impose sont détaillés en section 5.

Une même abstraction a été choisie pour les analyses, la modélisation des problèmes d'optimisation et de génération de code : un système de contraintes linéaires où les variables sont des entiers. La restriction *affine* des contraintes et le choix des $\mathcal{Z}$*-polyèdres*, présentés en section 2, nous ont permis de rester dans un cadre algébrique disposant d'une large gamme de méthodes de résolution, aisées pour les preuves.

Enfin la section 6 conclut ce mémoire avec une présentation des travaux de recherche que j'envisage pour les cinq années à venir.

Chapitre 2

Les bons outils

Ce chapitre présente :

— Le compilateur `PIPS` (section 2.1).

— L'abstraction choisie : les *$\mathcal{Z}$-polyèdres* (section 2.2).

— Les raisons de ce choix (section 2.3).

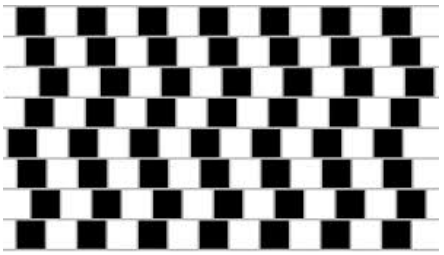
— L'abstraction *minimale* pour les dépendances de données (section 2.4).

— Les régions de tableaux (section 2.5).

L'utilisation de ces outils visent les principaux critères de qualité logicielle suivants : la vérificabilité, la fonctionnalité, la compatibilité, la robustesse et l'efficacité.

Mes publications relatives à ce chapitre sont les suivantes :[24, 78, 113, 129, 130, 140].

Vérifier que des ensembles d'instructions peuvent être exécutés en parallèle s'avère souvent être une tâche compliquée. On se retrouve comme devant cette figure, à s'interroger et à devoir utiliser les bons outils, pour prouver cette propriété : elles sont parallèles.



Les lignes de carrés blancs et noirs sont-elles bien parallèles ?

Dans le cadre des applications scientifiques, il est important de se doter d'outils mathématiques performants et d'abstractions appropriées permettant de valider les propriétés recherchées.

Deux outils importants sont à la base de nos techniques de validation : 1) la plate-forme de développement `PIPS` dans laquelle une majorité de nos travaux ont été implémentés et que je présente dans la section 2.1 et 2) la bibliothèque `LinearC3` d'algèbre linéaire. Cette dernière comprend un grand nombre d'algorithmes de manipulation de l'abstraction, le *polyèdre* [155], qui a été choisie pour représenter les informations nécessaires à la validation des applications scientifiques visées. La section 2.2 définit les *($\mathcal{Z}$-)polyèdres* et la section 2.3 détaille les raisons de ce choix. La section 2.4 montre que cette abstraction est *minimale* pour certaines analyses et transformations de programmes telle que l'analyse des dépendances du flot de données. Enfin, la section 2.5 présente les régions convexes de tableaux qui sont essentielles aux analyses de programme exposées dans les chapitres suivants.

2.1 Un environnement de développement PIPS

Le projet **P***aralléliseur* **I***nterprocédural de* **P***rogrammes* **S***cientifiques* [130, 113] [1] a démarré en 1988. L'objectif initial était la détection automatique du parallélisme à grain [2] moyen ou large dans les applications scientifiques. Depuis, `PIPS` [99] s'est développé et est devenu une formidable plate-forme modulaire permettant de tester et d'intégrer des prototypes d'analyse et d'optimisation.

Les atouts importants de `PIPS` pour nos travaux étaient les suivants :

PIPS est un compilateur source à source. Il prend en entrée un programme écrit en langage `C` ou en `Fortran` et produit un code écrit dans le même langage agrémenté soit de commentaires, de directives ou d'appels à des fonctions de bibliothèques spécifiques telles que `MPI`, `OpenCL` visant un type d'architectures spécifiques. L'avantage du source à source est qu'il est possible de vérifier les évolutions du code au cours de toutes les étapes de la transformation et de l'optimisation. Le programme modifié reste *lisible* et facilement interprétable, ce qui représente un avantage essentiel pour la maintenance du code et sa validation, notamment dans le cadre de processus industriels.

PIPS traite des applications de taille importante jusqu'à 200 000 lignes de code pour certaines applications industrielles écrites en langages C, Fortran 77 ou `HPF` [152],

PIPS est interprocédural. Les analyses sont effectuées fonction par fonction (procédure) et des résumés de ces informations sont calculés pour chacune d'elles. Lorsqu'une fonction `F` fait référence à une fonction `G`, les informations concernant `G` n'ont pas besoin d'être recalculées pour `F`, car leur résumé peut être utilisé (si il a été calculé au préalable).

PIPS est modulaire. Les analyses, les transformations et optimisations constituent un ensemble de briques de base que l'on peut combiner selon ses besoins. L'ajout de nouvelles phases est également simplifié [25].

`PIPS` comprend des phases d'analyse et de transformation. Les principales analyses et graphes de `PIPS` sont :

- Les **effets** des instructions sur les variables en lectures ou écritures.
- Les **transformers** : des prédicats entre variables scalaires représentant les changements d'état des variables.
- Les **préconditions** : des prédicats sur les variables entières scalaires du programme qui sont vérifiés avant l'exécution de l'instruction à laquelle ils sont attachés.
- Les **régions convexes de tableaux** [75] : des prédicats sur les éléments de tableaux référencés par un groupe d'instructions du programme (sections de tableaux polyédriques).
- Le **graphe des appels** qui décrit exhaustivement l'enchaînement des appels de fonction.
- Les **dépendances de données** qui décrivent les conflits mémoire potentiels empêchant la parallélisation.

Pour certaines analyses, plusieurs algorithmes ont été conçus avec des niveaux différents de qualité/précision des résultats. Par exemple, les **régions convexes de tableaux**

1. Les citations encadrées ou soulignées font partie de mes publications ou correspondent à des thèses que j'ai co-encadrées

2. La granularité du parallélisme correspond à la *taille* des tâches exécutées en parallèle, de la simple opération/instruction (grain fin) à un groupe de fonctions (large grain).

peuvent être calculées avec des algorithmes simples et rapides, pour un résultat correct mais approché des éléments de tableaux référencés par des instructions, ou avec des algorithmes plus complexes et coûteux mais fournissant un résultat plus précis.

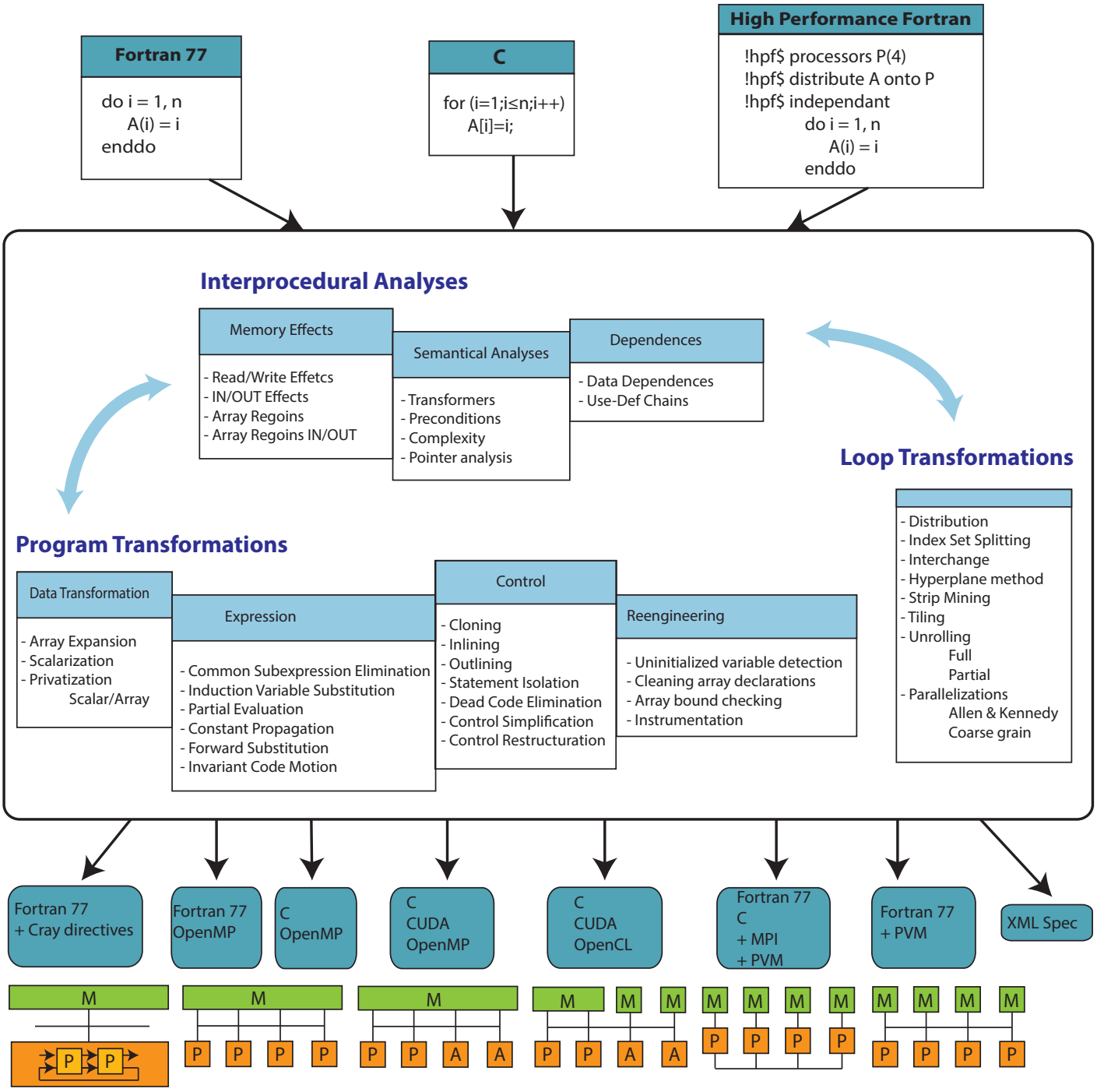


FIGURE 2.1 – Le compilateur PIPS

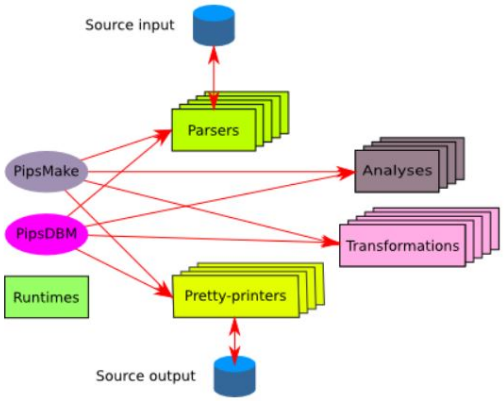
La liste des transformations de programme de `PIPS`, non exhaustive, comprend la normalisation des déclarations, les tests de non-débordement des accès aux tableaux, le nettoyage des déclarations, le *clonage*, la restructuration du contrôle, l'évaluation partielle des variables du programme, l'élimination de code mort, la privatisation de tableaux ou

de scalaires, la parallélisation et de nombreuses transformations de boucles : distribution, déroulage, strip-mining, échange, normalisation, réduction...

Enfin, plusieurs *pretty-printers* sont proposés :

- Les informations collectées peuvent être données sous forme de commentaires associés aux instructions du programme.
- Des directives de type `HPF`, `OPENMP` traduisent les informations de placement et parallélisation.
- Les codes parallèles pour architectures à mémoire distribuée utilisent des appels à des fonctions de bibiothèque telle que `PVM` ou `MPI`.
- Les codes parallèles pour machines hétérogènes avec GPU utilisent `CUDA` ou `OPENCL`.

Les analyses et les transformations sont gérées par un système à la *make*, `pipsmake`, qui garantit la cohérence entre les analyses et les modules. Les résultats sont stockés dans une base de données, pour les utilisations interprocédurales futures éventuelles, par un gestionnaire de ressources, `pipsdbm`. Les phases sont gérées par `pipsmake` et sont appelées à la demande pour effectuer les analyses ou les transformations requises par l'utilisateur. Ainsi la gestion des phases interprocédurales est très facile à formuler : seule la requête finale doit être spécifiée, `pipsmake` se charge du reste.



Le second outil est **la bibliothèque linéaire** `LinearC3` [129] qui gère les vecteurs, les matrices, les contraintes affines, les systèmes générateurs et les polyèdres. Elle fournit les opérateurs de base de manipulation de ces structures. Leur représentation est creuse. Les algorithmes utilisés sont conçus pour les entiers et/ou des coefficients rationnels. Une gestion des débordements *overflows* et *underflows* est intégrée dans tous les algorithmes par les exceptions *try-throw-catch*. Elle dispose également d'une version Gnu Multi-Précision optionnelle pour l'algorithme du simplexe. Cette bibliothèque est largement utilisée pour des analyses telles que les tests de dépendance, le calcul des préconditions et des régions convexes de tableaux, et pour les transformations, comme le *tiling*. La bibliothèque `LinearC3` fait appel à une implémentation de l'algorithme de Chernikova de la `Polylib` [107] et à une implémentation de `PIP` [83] (Parametric Integer Programmation).

La précision des analyses dépend de l'abstraction utilisée pour calculer les informations. Quel est le bon niveau d'abstraction pour une analyse donnée ? Quel est le compromis entre la précision de l'abstraction et le temps d'exécution de l'analyse ? Peut-on utiliser une seule et même abstraction pour représenter les résultats d'analyses différentes ? Voici des questions auxquelles il faut répondre très tôt dans le processus de développement de ces phases d'analyses.

Nous présentons dans le section suivante l'abstraction qui a été choisie pour représenter la majorité des analyses et modéliser certains problèmes de placement des applications sur machines parallèles : les $\mathcal{Z}$*-polyèdres*.

2.2 Définition : $\mathcal{Z}$-*polyèdre*

Quelques définitions avant de présenter les raisons du choix de cette abstraction.

Définition 1 : Un **polyèdre** convexe $P \subseteq R^n$ est un ensemble de points qui satisfont un nombre fini d'inéquations linéaires ; c'est à dire, P peut s'écrire sous la forme :

$P = \{ \mathbf{x} \in R^n \mid A \cdot \mathbf{x} \leq \mathbf{b} \}$

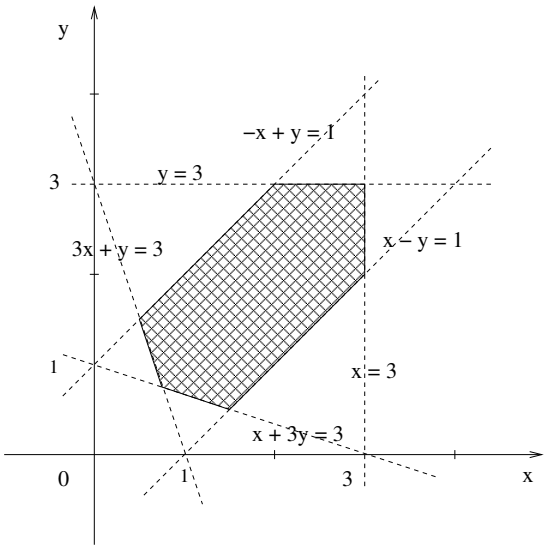
où A est une matrice de dimension mxn et $\mathbf{b}$ un vecteur constant de dimension n. P est donc une intersection finie de demi-espaces affines.

Un polyèdre convexe peut être caractérisé soit par un **système d'inéquations linéaires** soit par un **système générateur** [155]. Nous utiliserons dans ce document les systèmes d'inéquations linéaires.

L'exemple de la figure 2.2 montre un polyèdre avec sa représentation sous forme de contraintes linéaires et sa représentation graphique.

$$\begin{cases} x \leq 3 \\ y \leq 3 \\ -x + y \leq 1 \\ -3x - y \leq -3 \\ -x - 3y \leq -3 \\ -x + y \leq -1 \end{cases}$$

Représentation sous forme de système de contraintes linéaires.



Représentation graphique

FIGURE 2.2 – Exemple de polyèdre

Définition 2 : On appelle **$\mathcal{Z}$-module** [46] (notion de *lattice* ou treillis) dans $\mathcal{Z}^n$, l'ensemble des combinaisons entières d'un ensemble de vecteurs linéairement indépendants dans $\mathcal{Z}^n$. Ces vecteurs indépendants forment la base du $\mathcal{Z}$-module.

Définition 3 : On appelle **$\mathcal{Z}$-polyèdre** [46],[151] l'ensemble des points entiers résultant de l'intersection d'un polyèdre convexe et d'un $\mathcal{Z}$-module.

Tout point du $\mathcal{Z}$-polyèdre doit appartenir au système linéaire qui le définit et être une combinaison linéaire des vecteurs de base qui génèrent le $\mathcal{Z}$-module.

La figure 2.3 représente un $\mathcal{Z}$-polyèdre délimité par 5 contraintes. Le $\mathcal{Z}$-module, défini par $A = \{(0,1) + \alpha(1,0) + \beta(0,2) | (\alpha, \beta) \in \mathcal{Z}^2\}$, précise que seuls les éléments impairs sur y sont solutions. Le treillis opère comme un calque sur le polyèdre. Les éléments du $\mathcal{Z}$-polyèdre sont représentés par des étoiles sur la figure.

$$(x,y) \in A \quad et \quad \begin{cases} 1 \leq x \\ x \leq 5 \\ 2 * x - 1 \leq y \\ y \leq 2 * x + 5 \\ y \leq 11 \end{cases}$$

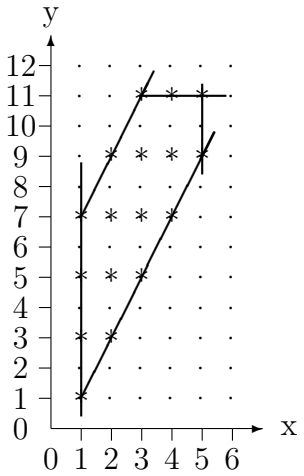


FIGURE 2.3 – Exemple de $\mathcal{Z}$-polyèdre

2.3 Choix de l'abstraction

Dans les années 80-90, de nombreuses études [47, 140],[124], ont cherché à déterminer parmi les représentations telles que les intervalles, les polyèdres, les congruences, les ensembles exhaustifs de points entiers, ... celles qui étaient les plus appropriées aux analyses statiques de programmes scientifiques dans le cadre de la parallélisation automatique.

Chacune des abstractions apporte une précision différente. Les cinq figures suivantes illustrent leur représentation du nid de boucles [3]

```
for (i=1; i<=5 ; i=i+1)
 for (j=2*i-1 ; j<= min(2*i+5,11) ; j=j+2)
      T[i,j] = B[i,j];
```

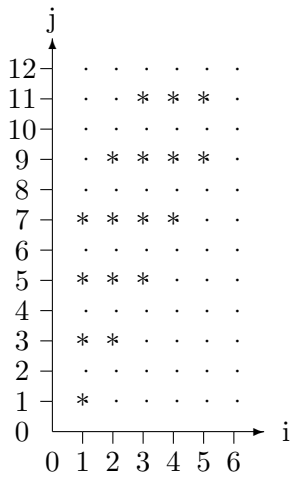
$$\begin{cases} 1 \leq i \\ i \leq 5 \\ 2 * i - 1 \leq j \\ j \leq 2 * i + 5 \\ j \leq 11 \end{cases}$$

Ensemble des itérations sous forme de contraintes

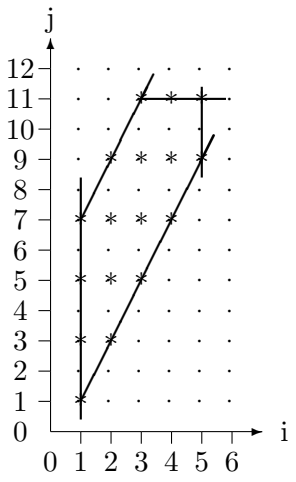
L'ensemble exact des itérations référencées par ce nid de boucles comporte 17 éléments (représentés par des étoiles). De la représentation la plus précise à la moins précise [4], nous avons la liste des itérations, la liste de polyèdres [119], l'arithmétique de Presburger [148], le $\mathcal{Z}$-polyèdre [91, 72, 155], les octogones [126, 127, 128] et les intervalles [71, 169]. Les congruences linéaires [88, 121] sont difficilement comparables aux autres abstractions car elles sont plus précises sur le treillis et moins sur le domaine de définition. L'arithmétique de Presburger, grâce à la quantification existentielle, permet de représenter des ensembles de polyèdres disjoints.

3. Dans cet exemple, la **région** qui caractérise les éléments du tableau T, référencés via la fonction d'accès (i,j) dans le nid de boucles, a la même représentation polyédrique que l'ensemble des itérations.

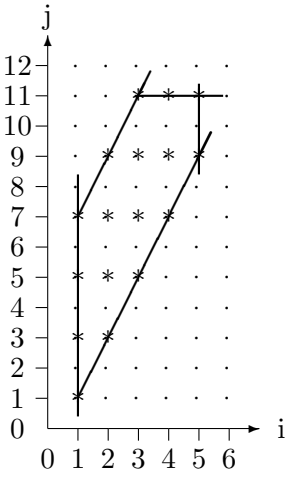
4. Le nombre d'éléments n'appartenant pas à la solution exacte, mais inclus dans l'abstraction, sont mentionnés sous chacune des représentations.



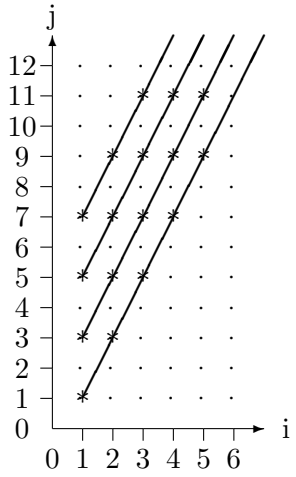
Liste de points- 0 point en sus



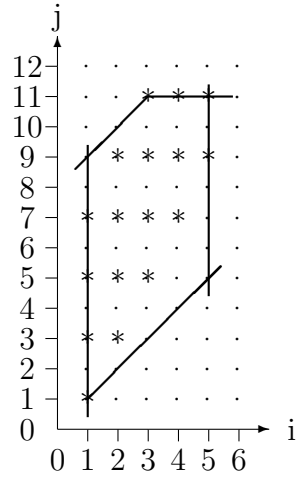
Presburger



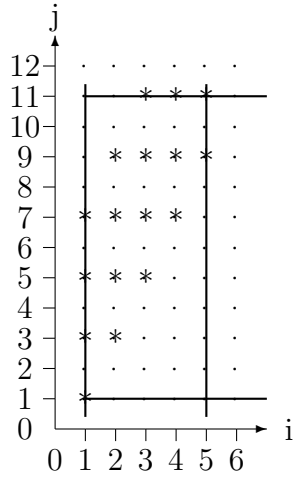
Polyèdre -
12 points en sus



Congruences



Octogone - 24 points en sus



Produit d'intervalles -
38 points en sus

En général, plus l'abstraction est simple plus faible est la complexité théorique des opérations classiquement utilisées pour la manipuler lors des analyses. Une comparaison plus précise des coûts des différents abstractions est donnée dans les mémoires [140],[124].

La thèse de Duong Nguyen [140], que j'ai dirigée sous le contrôle de F. Irigoin, avait pour objectif la définition d'une interface commune pour les bibliothèques, de manipulation de domaines abstraits, utilisées par les analyseurs statiques de programme : `PIPS` [97, 98], NBAC [163, 109], ASTRÉE [161, 54], OMEGA [164, 148] et CHINA [162, 165]. Cette recherche a été effectuée dans le cadre du projet APRON [47].

Des comparaisons des principales fonctions de manipulation de ces abstractions telles que la normalisation (simplification en fonction du type des données entiers ou rationnels), l'intersection, l'enveloppe convexe, le test d'inclusion, l'élimination des contraintes redondantes, le test de faisabilité et la projection y sont détaillées. Elles confirment le fait que certaines abstractions sont plus appropriées à certaines opérations, mais que globalement toutes ont des avantages et des inconvénients. Seul point significatif, sur les cas traités, le nombre de dimensions du polyèdre a un plus fort impact sur le risque d'explosion combinatoire que le nombre de contraintes (équations et inéquations).

Les résultats montrent également qu'il est difficile d'établir une hiérarchie entre ces différentes abstractions. À la question que nous nous posions ; "Peut-on utiliser une seule et même abstraction pour représenter les résultats d'analyses différentes ?" La réponse est qu'il ne peut pas y avoir unicité. Des compromis doivent être faits en fonction de la taille des problèmes et programmes visés, des contraintes de temps et de précision... D'où l'importance de l'adéquation entre une analyse et une abstraction.

Si une seule abstraction est choisie, dans le but d'effectuer des transformations par exemple, elle devra répondre au moins au critère de vérificabilité. Elle devra contenir les informations minimum nécessaires pour décider de la validité de l'application de ces transformations.

Pourquoi avoir choisi les polyèdres ? La représentation polyédrique s'est imposée à de nombreuses équipes de compilation [5] car :

1. Elle est **bien adaptée aux structures** de données des programmes scientifiques. Les éléments de matrices et de tableaux, les nids de boucles décrivent des ensembles que l'on approxime naturellement par des polyèdres.
2. De nombreuses transformations que l'on souhaite appliquer aux nids de boucles correspondent à des **transformations affines d'ensemble de points entiers** :
   — L'échange de boucles est une transposition.
   — La fusion de boucles.
   — Le déroulage de boucle.
   — Le tiling de boucles.

   La combinaison de transformations s'exprime alors simplement par la multiplication de ces matrices.
3. Pour les outils comme `PIPS`, qui vise des analyses fournissant des résultats d'une **grande précision**, les polyèdres sont essentiels [6] [7]. D'ailleurs, en ce qui concerne les éléments de tableaux, une abstraction par intervalles ne permettrait pas une spécification suffisamment précise de ces éléments et limiterait la vérification des propriétés les concernant. Or les éléments de tableaux/matrices avec les boucles sont les structures les plus manipulées dans les programmes scientifiques.

Toutefois, les polyèdres ont été souvent jugés inacceptables à cause de la complexité exponentielle au pire cas de certains opérateurs en temps d'exécution. Concernant les analyses, le contexte dans lequel cette représentation est utilisée est déterminant. Par exemple, le calcul des dépendances entre éléments de tableaux (développé en section 2.4) s'exprime très bien avec des polyèdres. Il bénéficie de la précision des informations intrinsèques à la représentation polyédrique des dépendances de données. Et l'algorithme manipulant cette représentation, dont le but est de détecter s'il y a dépendances ou non, résulte en un simple test de faisabilité du système de contraintes. La probabilité d'une

5. Telecom(Evry), PIPS(Fontainebleau), Verimag(Grenoble), Université de Lille, Compsys-LIP(Lyon), ENS-DI(Paris), Irisa(Rennes), ICPS-ICUBE(Strasbourg), ... mais aussi COSY(University of Colorado), ISL(Leuven University), Omega Project(University of Maryland), PLUTO(Indian Institute of Technology), Polaris(University of Illinois at Urbana-Champaign), Polly(University of Passau), SUIF(Stanford University), ...

6. L'arithmétique de Presburger apporte une précision supplémentaire mais au prix d'une complexité triplement exponentielle.

7. Les listes de polyèdres [119] sont très coûteuses pour des opérateurs tels que l'intersection ou la différence.

explosion combinatoire de ce système polyédrique est bien plus faible que dans le cadre des autres analyses [174].

Pour les algorithmes de résolution qui sont sur le plan théorique de complexité exponentielle, cette probabilité est non-nulle, même si dans la pratique, ces cas restent des exceptions. L'algorithme bien connu du simplexe, qui permet de résoudre des problèmes d'optimisation linéaire, est sur le plan théorique de complexité exponentielle mais reste très largement utilisé.

Duong Nguyen a pu observer [140] que la taille d'un polyèdre caractérisant les éléments d'un tableau lus au sein d'un nid de boucles peut atteindre parfois plus d'une dizaine de dimensions et le nombre de contraintes portant sur ces variables au cours d'une analyse peut croître exponentiellement, et donc dépasser la centaine de contraintes en quelques itérations [140].

De nouveaux algorithmes ont été développés afin de manipuler aux mieux les polyèdres : des transformations sont effectuées pour éviter de se placer dans le cas où l'explosion combinatoire des structures est probable.

J'ai conçu des heuristiques permettant d'optimiser les opérations polyédriques. En collaboration avec F. Irigoin et F. Coelho, j'ai défini les critères qui entrent en jeu dans les cas d'explosion combinatoire. Ainsi, les heuristiques suivantes ont été introduites dans la bibliothèque `LinearC3` pour l'ensemble des opérateurs polyédriques :

- les projections de variables sont effectuées dans un ordre privilégiant les variables ayant un faible coefficient ;
- les projections de variables sont effectuées dans un ordre privilégiant les variables *non liées*, dont l'élimination n'introduit pas un grand nombre de variables dans les nouvelles contraintes ;
- la normalisation permet de limiter les *overflows*. Elle est largement utilisée ;
- des approximations sont effectuées (réduction des coefficients) sans perte de précision pour la propriété recherchée ;
- enfin l'élimination des contraintes redondantes est appliquée régulièrement.

J'ai également proposé des techniques de décomposition des polyèdres [140] permettant d'accélérer le calcul de l'enveloppe convexe sur des polyèdres représentatifs des problèmes à traiter.

Néanmoins, il reste des cas, exceptionnels heureusement, où les systèmes de contraintes explosent, soit en nombre de contraintes soit en magnitude des coefficients. Lorsque les systèmes divergent, il est difficile d'inverser la tendance car même les algorithmes d'élimination des contraintes redondantes ou de normalisation, faisant appel eux-mêmes à des transformations du polyèdre, ne peuvent plus être appliqués. Dans ces cas critiques, une sur-approximation de l'ensemble des solutions est choisie : $\mathcal{Z}^n$ par exemple.

2.4 Abstraction pour les dépendances

Le choix d'une abstraction ne se limite pas à un critère de performance ou d'espace mémoire. Si on désire prouver des propriétés sur un programme, il faut que l'abstraction choisie pour l'analyse contienne les informations nécessaires à leur vérification.

La notion de dépendances de flot de données est essentielle pour les analyses statiques, les transformations et la parallélisation de programmes.

```
DO I = 1, n
  DO J = 1, n
    T(I,J) = A
    B = T(3I,J+1)
```

$$(S) \begin{cases} 1 \leq i \leq n & 1 \leq i' \leq n & 3i = i' \\ 1 \leq j \leq n & 1 \leq j' \leq n & j+1 = j' \end{cases}$$

FIGURE 2.4 – Programme P et son système de dépendances.

La figure 2.4 donne un exemple de programme P et le système de dépendances (S) associé. Deux instances différentes, i,i' de la première boucle et j,j' de la deuxième boucle, référencent le même élément du tableau T en lecture et écriture si le système (S) est satisfait.

Les dépendances précisent les contraintes d'ordre d'exécution des lectures et écritures des variables du programme. Cet ordre ne peut pas être changé sans modifications possibles de la sémantique du programme. Toute transformation/optimisation ne peut être appliquée systématiquement que si les relations de dépendance initiales sont maintenues après transformation.

Définition 4 : Une **transformation est légale** lorsque le programme possède la même sémantique après la transformation : le comportement du programme est inchangé.

On définit des abstractions *valides* et *minimales* associées à une transformation :

Définition 5 : Une **abstraction valide** associée à une transformation T est une abstraction qui contient suffisamment d'information pour décider de la légalité de T.

Définition 6 : Une **abstraction minimale** associée à une transformation T est l'abstraction qui contient le minimum d'information nécessaire pour décider quand cette transformation est légale.

La thèse de Yi Qing Yang [174], que j'ai encadrée sous le contrôle de F. Irigoin, a pour thème *Les tests de dépendances et les transformations de programmes*. Son objectif était de trouver les abstractions des dépendances *minimales* permettant de prouver la légalité de transformations de réordonnancement de boucles, primordiales en parallélisation automatique. Les articles [175, 176] apportent des réponses à cette question.

Dans les années 80-90, de nombreux chercheurs ont proposé de nouvelles abstractions ; chacune d'elles correspondait aux informations nécessaires pour appliquer une transformation particulière. A titre d'exemple, les Vecteurs de Direction des dépendances (DDV) [171] ont été introduits par Wolf pour l'échange de boucles et les Niveaux de dépendance (DL) [4] par Allen & Kennedy pour la vectorisation et la parallélisation. D'autres abstractions ont aussi été introduites telles que les Vecteurs de Dépendance (D) [133] ou encore le Polyèdre de Dépendance (DP) [100] et le Cône de dépendance (DC) [100].

Ces abstractions fournissent des précisions différentes. Leur représentation de l'ensemble des itérations du programme `P` sont illustrées sur les figures 2.5 et 2.6. Noter que DI, qui est la liste exacte des dépendances point à point, est représentée dans un repère différent. (i, j) est remplacé dans les autres représentations par $(d_i, d_j) = (i' - i,\ j' - j)$.

En fonction de leur précision, ces abstractions contiennent soit insuffisamment ou suffisamment ou trop d'informations pour décider si un transformation T particulière peut être légalement appliquée ou non.

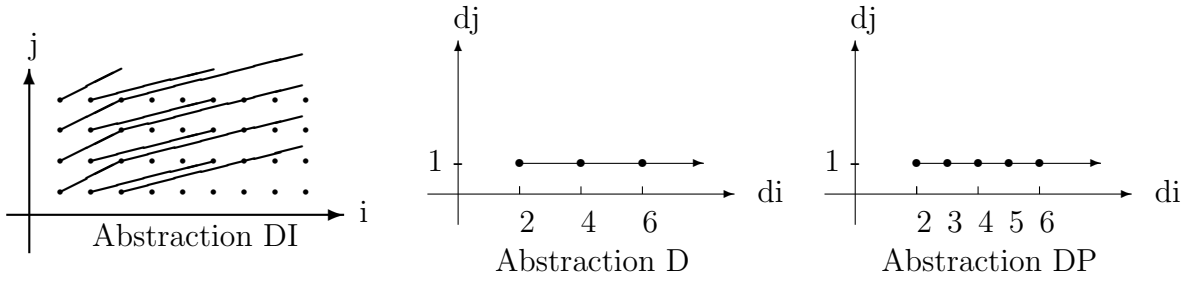


FIGURE 2.5 – Abstractions DI, D et DP du nid de boucles du programme P

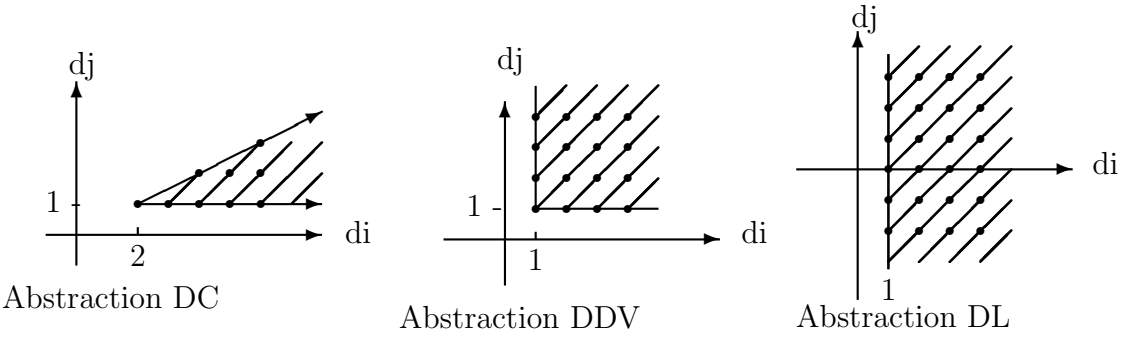


FIGURE 2.6 – Abstractions DC, DDV et DL du nid de boucles du programme P

Ces travaux ont permis de montrer qu'il existait une hiérarchie dans la précision de ces abstractions de dépendances, de la plus précise à la moins précise : DI $\supseteq$ D $\supseteq$ DP $\supseteq$ DC $\supseteq$ DDV $\supseteq$ DL [8].

Ils ont aussi permis d'identifier les abstractions minimales pour les transformations de réordonnancement : inversion de boucles, permutation, transformation unimodulaire, partitionnement et parallélisation, qui sont respectivement DL, DDV, DC, DC et DL. Toutes les abstractions plus précises que l'abstraction minimale associée à une transformation étant valides pour cette transformation, *DC* est une abstraction valide pour l'ensemble de ces transformations.

Le cône de dépendances est utilisé dans `PIPS`. Ses informations sont suffisamment précises pour tester la légalité d'un ensemble de transformations de réordonnancement de boucles et notamment la parallélisation des boucles.

8. $\subset$ est l'opérateur d'inclusion des ensembles.

2.5 Régions convexes de tableaux

Je présente, dans cette section, les régions convexes de tableaux, car elles constituent une information précieuse utilisée par de nombreuses autres analyses, transformations, optimisations de PIPS ainsi que pour la génération de code de communications. Les régions READ et WRITE ont été introduites par R.Triolet [167] en 1988 dans le but d'analyser les dépendances interprocédurales. Elles ont été ensuite reprises et complétées avec les régions IN et OUT par B. Creusillet [98, 74, 73, 75] :

— les régions READ ou WRITE représentent les effets en lecture ou en écriture des instructions du programme.

— les régions IN ou OUT représentent l'ensemble des éléments de tableaux dont la valeur est importée ou exportée par l'instruction ou la fonction considérée. Elles permettent de calculer les communications à générer avant ou après l'exécution d'une portion de code. Elles permettent également de privatiser les sections de tableaux.

Les régions représentent les *effets* opérés sur des éléments de tableaux. Dans PIPS, il y a trois types d'effets :

— Les effets *propres* qui sont locaux, relatifs à l'instruction elle même. Les effets propres d'un test n'incluent pas les effets propres de chacune de ses branches.

— Les effets *cumulés* qui représentent l'ensemble des effets relatifs à un groupe d'instructions. Ils incluent les effets des instructions qui les composent (par exemple, celles du corps de boucle pour une boucle).

— PIPS est interprocédural, et comme pour les autres analyses, pour les fonctions, un résumé des régions est calculé afin de pouvoir être utilisé au niveau des sites d'appel. Les effets *résumés* sont utilisés pour calculer les effets propres d'une fonction au site d'appel.

Si l'ensemble des éléments de tableaux effectivement référencés par une section de code n'a pas été approximé (le polyèdre ne contient pas d'éléments non référencés), un attribut `EXACT` sera associé à la région. Sinon la région sera qualifiée `MAY`.

Il est nécessaire d'utiliser d'autres analyses, notamment les préconditions, *transformers* et les conditions de continuation, pour pouvoir comparer ou combiner les régions de tableaux. La région étant exprimée dans un état mémoire donné [9], pour pouvoir appliquer un opérateur sur des régions différentes, il faut qu'elles soient exprimées dans le même état mémoire.

1. Les *transformers* et et *transformers inverses* décrivent la sémantique des instructions d'un état mémoire à un autre. Ils permettent de traduire une région dans un état mémoire donné.
2. Les préconditions apportent une information précise sur les valeurs des variables du programme relatives à un état mémoire donné.
3. Les conditions de continuation renseignent sur les conditions sous lesquelles il existe effectivement un chemin d'exécution du début du programme à l'instruction considérée.

9. Un état mémoire représente l'état courant des valeurs des variables.

Nous illustrons les résultats des régions sur l'exemple de la figure 2.7.

```
PROGRAM SAMAN
INTEGER I,J, N, WORK(100), A(200)
N = 50
DO J = 1, 100
   WORK(J) = 0
ENDDO
DO J = 1, 200
   DO I = 1, N
      WORK(I) = I+J
   ENDDO
   DO I = 1, 2*N
      A(J) = WORK(I)
   ENDDO
ENDDO
END
```

FIGURE 2.7 – Code initial

2.5.1 Le calcul des régions de tableaux READ et WRITE

Le détail des calculs des régions READ, WRITE, IN et OUT de tableaux est présenté dans les articles [98, 74, 73, 75]. Je résume, dans cette section, les grandes lignes utiles à la compréhension de l'utilisation des régions de tableaux dans les analyses, les transformations et les phases de génération de code, qui sont décrites dans mon document. Je limite cette description aux principales structures de de contrôle des langages Fortran et C : une expression, une affectation, une séquence, une instruction conditionnelle et les boucles.

Expression Les expressions sont considérées comme n'ayant pas d'effet de bord. L'effet d'une expression en lecture est l'union des effets de chacune de ses composantes.

Affectation La partie droite d'une affectation est une expression. La partie de gauche a des effets en écriture sur les éléments de tableau. Elle a aussi des effets en lecture sur les indices de la fonction d'accès aux éléments du tableau. Les régions READ d'une affectation sont définies par :

$$\begin{array}{lcl} \mathcal{R}_r[\![var = exp]\!] & = & \mathcal{R}_r[\![exp]\!] \\ \mathcal{R}_r[\![var(exp_1, \ldots, exp_k) = exp]\!] & = & \mathcal{R}_r[\![exp_1, \ldots, exp_k]\!] \cup \mathcal{R}_r[\![exp]\!] \end{array}$$

Séquence Les effets en lecture d'une séquence sont l'union des effets des instructions qui composent la séquence. Avant composition, elles doivent toutefois être ramenées à un même état mémoire. Considérons la séquence de deux instructions $S_1; S_2$, leurs régions READ sont définies par l'équation :

$$\mathcal{R}_r[\![S_1; S_2]\!] = \mathcal{R}_r[\![S_1]\!] \cup \mathcal{R}_r[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$$

où $\mathcal{T}[\![S_1]\!]$ modélise la transformation de l'état mémoire par l'exécution de de S_1.

Instruction conditionnelle Les éléments référencés sont ceux référencés par la condition et, selon l'évaluation de cette condition $\mathcal{E}[\![C]\!]$, ceux de la branche vraie ou fausse.

$$\mathcal{R}_r[\![\texttt{if } C \texttt{ then } S_1 \texttt{ else } S_2]\!] = \mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S_1]\!] \circ \mathcal{E}[\![C]\!]) \cup (\mathcal{R}_r[\![S_2]\!] \circ \mathcal{E}[\![\texttt{.not.} C]\!])$$

Boucle While(C) S Pour comprendre le calcul des régions de la boucle : `do while(`C`)` S, considérons son exécution à partir de l'état mémoire σ. Si l'évaluation de la condition $\mathcal{E}[\![C]\!]\sigma$ est fausse, la boucle stoppe. Si elle est vraie, l'instruction S est exécutée au moins une fois et les éléments $\mathcal{R}_r[\![S]\!]$ sont lus. Jusqu'à ce point, les régions READ sont définies par :

$$\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S]\!] \circ \mathcal{E}_c[\![C]\!])$$

Après cette première itération, le calcul des régions boucle mais avec un état mémoire, résultant de l'exécution de S sachant que la condition C a été évaluée à vraie, calculé grâce au transformer $\mathcal{T}[\![S]\!] \circ \mathcal{E}_c[\![C]\!]$. Nous obtenons :

$$\mathcal{R}_r[\![\texttt{do while(}C\texttt{)} \ S]\!] = \mathcal{R}_r[\![C]\!] \cup \big(\mathcal{R}_r[\![S]\!] \cup \mathcal{R}_r[\![\texttt{do while(}C\texttt{)} \ S]\!] \circ \mathcal{T}[\![S]\!]\big) \circ \mathcal{E}_c[\![C]\!]$$

Une définition, utilisant le plus petit point fixe lfp, donne la fonction sémantique pour un *while* :

$$\mathcal{R}_r[\![\texttt{do while(}C\texttt{)} \ S]\!] = \mathrm{lfp}\big(\lambda f.\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S]\!] \cup f \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\big)$$

Boucle For ou Do i=1,n S Nous introduisons maintenant le calcul des régions pour des boucles For ou Do où le nombre d'itérations des boucles est connu. La première étape consiste à calculer les régions du corps de boucle $\mathcal{R}_r[\![S]\!]$. Ces régions correspondent à une itération quelconque. Elles sont fonction de variables qui peuvent avoir été modifiées dans cette itération. Ces variables sont éliminées en utilisant le transformer de la boucle $\mathcal{T}[\![S]\!]$ qui fournit l'invariant de boucles, c'est à dire les relations entre les variables modifiées à chaque itération et les valeurs des variables avant exécution de la boucle. L'indice de boucle i est ensuite éliminé en tenant compte des contraintes de l'espace d'itérations ($0 \leq i \leq n$, dans notre exemple).

$$\begin{aligned}
&\mathcal{R}_r[\![\texttt{do i=1,n } S \texttt{ enddo}]\!] \\
&\quad = \mathrm{lfp}\big(\lambda f.\mathcal{R}_r[\![C]\!] \cup (\mathcal{R}_r[\![S]\!] \cup f \circ \mathcal{T}[\![S]\!]) \circ \mathcal{E}_c[\![C]\!]\big) \\
&\quad = \lambda\sigma.\mathcal{R}_r[\![\texttt{n}]\!]\sigma \cup \bigcup_{k=1}^{k=\mathcal{E}[\![\texttt{n}]\!]\sigma} \mathcal{R}_r[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1 \ S \texttt{ enddo}]\!]\sigma \\
&\quad = \lambda\sigma.\mathcal{R}_r[\![\texttt{n}]\!]\sigma \cup \mathrm{proj}_k\big(\mathcal{R}_r[\![S]\!] \circ \mathcal{T}[\![\texttt{do i=1,}k-1 \ S \texttt{ enddo}]\!] \\
&\qquad\qquad \circ \mathcal{E}_c[\![\texttt{(1.lt.}k\texttt{).and.(}k\texttt{.lt.n)}]\!]\big)\sigma
\end{aligned}$$

Les définitions des régions WRITE sont très similaires à celles des régions READ, exceptées pour les affectations et les entrées/sorties. Pour les affectations, les équations sont les suivantes :

$$\begin{aligned}
\mathcal{R}_w[\![var = exp]\!] &= \lambda\sigma.\{var\} \\
\mathcal{R}_w[\![var(exp_1, \ldots, exp_k) = exp]\!] &= \lambda\sigma.\{var(\mathcal{E}[\![exp_1, \ldots, exp_k]\!]\sigma)\}
\end{aligned}$$

La figure 2.8 donne les résultats des régions READ et WRITE pour le code précédent.

```
      PROGRAM SAMAN
      INTEGER I,J, N, WORK(100), A(200)
      N = 50
C  <WORK(PHI1)-W-EXACT-{1<=PHI1, PHI1<=100, N==50}>
      DO J = 1, 100
C  <WORK(PHI1)-W-EXACT-{PHI1==J, N==50, 1<=J, J<=100}>
         WORK(J) = 0
      ENDDO
C  <A(PHI1)-W-EXACT-{1<=PHI1, PHI1<=200, N==50}>
C  <WORK(PHI1)-R-EXACT-{1<=PHI1, PHI1<=100, N==50}>
C  <WORK(PHI1)-W-EXACT-{1<=PHI1, PHI1<=50, N==50}>
      DO J = 1, 200
C  <WORK(PHI1)-W-EXACT-{1<=PHI1, PHI1<=50, N==50, 1<=J, J<=200}>
         DO I = 1, N
C  <WORK(PHI1)-W-EXACT-{PHI1==I, N==50, 1<=I, I<=50, 1<=J, J<=200}>
            WORK(I) = I+J
         ENDDO
C  <A(PHI1)-W-EXACT-{PHI1==J, N==50, 1<=J, J<=200}>
C  <WORK(PHI1)-R-EXACT-{1<=PHI1, PHI1<=100, N==50, 1<=J, J<=200}>
         DO I = 1, 2*N
C  <A(PHI1)-W-EXACT-{PHI1==J, N==50, 1<=I, I<=100, 1<=J, J<=200}>
C  <WORK(PHI1)-R-EXACT-{PHI1==I, N==50, 1<=I, I<=100, 1<=J, J<=200}>
            A(J) = WORK(I)
         ENDDO
      ENDDO
      END
```

FIGURE 2.8 – Régions Read-Write

2.5.2 Le calcul des régions de tableaux IN et OUT

Les régions IN d'une instruction complexe représentent les éléments *importés*, c'est à dire les éléments qui sont lus avant d'être éventuellement écrits au sein de cette instruction. Les régions OUT d'une instruction complexe représentent les éléments de tableaux *vivants* ou *exportés*, c'est à dire ceux qui sont définis par l'instruction et utilisés par des instructions suivantes.

La propagation des régions IN sur le graphe de contrôle est ascendante tandis que celle des régions OUT est descendante.

L'objectif de cette section est de donner un aperçu des calculs effectués pour des régions IN et OUT. Je limite cette description aux affectations et aux séquences. Les calculs pour les autres structures du langage sont présentés dans les articles [98, 74, 73, 75]. Les figures 2.9 et 2.10 donnent les résultats des régions IN et OUT pour le code précédent.

```
      PROGRAM SAMAN
      INTEGER I,J, N, WORK(100), A(200)
      N = 50
      DO J = 1, 100
         WORK(J) = 0
      ENDDO
C  <WORK(PHI1)-IN-EXACT-{51<=PHI1, PHI1<=100, N==50}>
      DO J = 1, 200
         DO I = 1, N
            WORK(I) = I+J
         ENDDO
C  <WORK(PHI1)-IN-EXACT-{1<=PHI1, PHI1<=100, N==50, 1<=J, J<=200}>
         DO I = 1, 2*N
C  <WORK(PHI1)-IN-EXACT-{PHI1==I, N==50, 1<=I, I<=100, 1<=J, J<=200}>
            A(J) = WORK(I)
         ENDDO
      ENDDO
      END
```

FIGURE 2.9 – Régions IN

Affectation Les régions IN d'une affectation correspondent aux régions READ.

$$\mathcal{R}_i[\![ref = exp]\!] = \mathcal{R}_r[\![ref = exp]\!]$$

Séquence Les régions IN de la séquence $S_1; S_2$ contiennent les éléments de tableaux importés par S_1 ($\mathcal{R}_i[\![S_1]\!]$), plus ceux importés par S_2 après l'exécution de S_1 ($\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]$) mais qui ne sont pas définis par S_1 ($\mathcal{R}_w[\![S_1]\!]$) :

$$\mathcal{R}_i[\![S_1; S_2]\!] = \mathcal{R}_i[\![S_1]\!] \cup \big((\mathcal{R}_i[\![S_2]\!] \circ \mathcal{T}[\![S_1]\!]) \ominus \mathcal{R}_w[\![S_1]\!]\big)$$

Soit une séquence S comportant n instructions $S_1; S_2; ..; S_n$. Pour chaque instruction S_k, sont supposées connues les régions WRITE $\mathcal{R}_w[\![S_k]\!]$ et IN $\mathcal{R}_i[\![S_k]\!]$, exprimées dans l'état σ_k précédant S_k. Nous notons $\mathcal{R}'_i[\![S_k]\!]$ la région IN correspondant à la

séquence $S_k, \ldots, S_n$. La région est alors définie par :

$$\begin{cases} \mathcal{R}'_i[\![S_n]\!] = \mathcal{R}_i[\![S_n]\!] \\ \mathcal{R}'_i[\![S_k]\!] = \mathcal{R}_i[\![S_k]\!] \cup [\ \mathcal{T}_{\sigma_{k+1}\longrightarrow\sigma k}(\mathcal{R}'_i[\![S_{k+1}]\!]) \ \ominus\ \mathcal{R}_w[\![S_k]\!]\ ] \\ \mathcal{R}_i[\![S]\!] = \mathcal{R}'_i[\![S_1]\!] \end{cases}$$

Le transformer $\mathcal{T}_{\sigma_{k+1}\longrightarrow\sigma k}$ permet ici d'exprimer la région IN correspondant à $S_{k+1}, \ldots, S_n$ dans le même état mémoire σ_k que les régions $\mathcal{R}_w[\![S_k]\!]$ et $\mathcal{R}_i[\![S_k]\!]$. $\mathcal{T}_{\sigma_{k+1}\longrightarrow\sigma k}(\mathcal{R}'_i[\![S_{k+1}]\!]) \ominus \mathcal{R}_w[\![S_k]\!]$ représente la région importée par la sous-séquence $S_{k+1}, \ldots, S_n$, mais non définie par l'instruction S_k. La fusion avec $\mathcal{R}_i[\![S_k]\!]$ donne l'ensemble des éléments lus par la séquence $S_k, \ldots, S_n$ avant d'être éventuellement redéfinis par cette même séquence.

```
      PROGRAM SAMAN
      INTEGER I,J, N, WORK(100), A(200)
      N = 50
C  <WORK(PHI1)-OUT-EXACT-{51<=PHI1, PHI1<=100, N==50}>
      DO J = 1, 100
C  <WORK(PHI1)-OUT-EXACT-{PHI1==J, 51<=PHI1, N==50, J<=100}>
         WORK(J) = 0
      ENDDO
      DO J = 1, 200
C  <WORK(PHI1)-OUT-EXACT-{1<=PHI1, PHI1<=50, N==50, 1<=J, J<=200}>
         DO I = 1, N
C  <WORK(PHI1)-OUT-EXACT-{PHI1==I, N==50, 1<=I, I<=50, 1<=J, J<=200}>
            WORK(I) = I+J
         ENDDO
         DO I = 1, 2*N
            A(J) = WORK(I)
         ENDDO
      ENDDO
      END
```

FIGURE 2.10 – Régions OUT

La propagation des régions OUT sur le graphe de contrôle est descendante. Pour une séquence d'instructions, nous avons les fonctions sémantiques suivantes :

Séquence Nous supposons que les régions OUT de la séquence $S = S_1; S_2; ..; S_n$ sont connues, et notre but est de calculer les régions OUT de S_1,.., S_k,.., S_n respectivement $\mathcal{R}_o[\![S_1]\!]$, $\mathcal{R}_o[\![S_k]\!]$ et $\mathcal{R}_o[\![S_n]\!]$.

Nous noterons $\mathcal{R}'_o[\![S_k]\!]$ l'ensemble des éléments de tableaux exportés par les instructions $S_1, \ldots, S_k$, et dont la valeur est réutilisée *après* l'exécution de la séquence S. Les fonctions permettant de calculer les $\mathcal{R}_o[\![S_k]\!]$ sont alors :

$$\begin{cases} \mathcal{R}'_o[\![S_n]\!] = \mathcal{T}_{\sigma_S\longrightarrow\sigma n}(\mathcal{R}_o[\![S]\!]) \\ \mathcal{R}_o[\![S_n]\!] = \mathcal{R}_w[\![S_n]\!] \cap \mathcal{R}'_o[\![S_n]\!] \end{cases}$$

et, $\forall\, k \in [1..n-1]$,

$$\begin{cases} \mathcal{R}'_o[\![S_k]\!] = \mathcal{T}_{\sigma_{k+1}\longrightarrow\sigma k}(\mathcal{R}'_o[\![S_{k+1}]\!] \ \ominus\ \mathcal{R}_w[\![S_{k+1}]\!]) \\ \mathcal{R}_o[\![S_k]\!] = \mathcal{R}_w[\![S_k]\!] \cap [\ \mathcal{R}'_o[\![S_k]\!] \cup \mathcal{T}_{\sigma_{k+1}\longrightarrow\sigma k}(\mathcal{R}'_i[\![S_{k+1}]\!])\ ] \end{cases}$$

$\mathcal{R}_o[\![S]\!]$ est la région OUT associée à la séquence S, et donc à $S_1, \ldots, S_n$, et exprimée dans l'état mémoire précédant son exécution (σ_S). $\mathcal{T}_{\sigma_S \longrightarrow \sigma n}(\mathcal{R}_o[\![S]\!])$ est donc la région associée à $S_1, \ldots, S_n$, mais exprimée dans l'état mémoire σ_n précédant S_n, ce qui est la définition de $\mathcal{R}'_o[\![S_n]\!]$. Nous avons donc bien :

$$\mathcal{R}'_o[\![S_n]\!] = \mathcal{T}_{\sigma_S \longrightarrow \sigma n}(\mathcal{R}_o[\![S]\!])$$

La restriction à la sous-séquence $S1, \ldots, S_k$ de la région OUT de S, contient les éléments exportés à l'extérieur de S par la séquence $S1, \ldots, S_{k+1}$, moins les éléments déjà exportés par l'instruction S_{k+1} ; le tout exprimé dans l'état mémoire σ_k, soit :

$$\mathcal{R}'_o[\![S_k]\!] = \mathcal{T}_{\sigma_{k+1} \longrightarrow \sigma k}(\mathcal{R}'_o[\![S_{k+1}]\!] \ \ominus \ \mathcal{R}_w[\![S_{k+1}]\!])$$

Enfin, la région OUT de l'instruction S_k est la région écrite par S_k et exportée vers l'extérieur de la séquence, $\mathcal{R}_w[\![S_k]\!] \cap \mathcal{R}'_o[\![S_k]\!]$, à laquelle il faut ajouter la région écrite par S_k et exportée vers $S_{k+1}, \ldots, S_n$, soit $\mathcal{R}_w[\![S_k]\!] \cap \mathcal{T}_{\sigma_{k+1} \longrightarrow \sigma k}(\mathcal{R}'_i[\![S_{k+1}]\!])$. D'où

$$\mathcal{R}_o[\![S_k]\!] = \mathcal{R}_w[\![S_k]\!] \cap [\mathcal{R}'_o[\![S_k]\!] \cup \mathcal{T}_{\sigma_{k+1} \longrightarrow \sigma k}(\mathcal{R}'_i[\![S_{k+1}]\!])$$

Les régions convexes de tableaux constituent une information essentielle aux analyses présentées dans le chapitre 3.

2.6 Conclusion

Mes activités de recherche, visant à assurer la qualité des applications, sont guidées par les besoins des applications scientifiques exécutées sur des architectures parallèles. Elles s'appuient sur deux axes. Le premier relève de la compilation et de l'optimisation d'applications scientifiques en vue de leur exécution efficace sur des architectures parallèles. Le deuxième relève des mathématiques appliquées et de l'utilisation de l'algèbre linéaire en nombres entiers pour modéliser les problèmes rencontrés et apporter des solutions efficaces.

Concernant le premier axe, le compilateur `PIPS`, développé au CRI depuis 1988, s'est avéré une formidable plate-forme modulaire permettant de tester et d'intégrer des prototypes d'analyse et d'optimisation.

J'ai participé au développement d'algorithmes effectuant :

- les tests de dépendances déterminant les informations relatives à la parallélisation des calculs et au maintien de la cohérence des données [175, 176, 174] (section 2.4) ;
- de la vérification du non-débordement des accès aux éléments de tableaux [138, 135][136] (section 3.2) ;
- de la détection de variables non initialisées [137] (section 3.3) ;
- de la vérification des déclarations des tableaux [43, 135] (section 3.1) ;
- du calcul des invariants de boucles [26].

Je détaille ces algorithmes dans la section 3. Ils sont maintenant intégrés comme des passes dans `PIPS`.

Concernant l'aspect mathématiques appliquées, je me suis investie dans la création et les développements de la bibliothèque linéaire `LinearC3`. J'ai contribué à de nombreuses techniques de manipulation de ses composants de bases : matrices, vecteurs, systèmes de contraintes et $\mathcal{Z}$-polyèdres.

Les deux thèses co-encadrées de Yi Qing Yang[174] et Duong Nguyen [140] ont conforté le choix des polyèdres comme représentation pour 1) les analyses statiques de programme notamment pour les dépendances, les *transformers*, les préconditions et les régions convexes de tableaux, et 2) pour valider la légalité des transformations de réordonnancement des boucles.

Les algorithmes utilisés sont conçus pour les entiers et/ou des coefficients rationnels. J'ai conçu et développé de nombreux algorithmes permettant la simplification des systèmes, l'élimination de contraintes redondantes, la projection de dimension, des tests d'existence de points entiers dans les systèmes... Les outils algébriques disponibles sont utilisés au mieux pour affiner les résultats des analyses et traiter les problèmes en nombres entiers rencontrés dans le cadre de la synthèse de code de calcul et de communication pour des architectures parallèles (section 4).

L'abstraction polyédrique permet de rester dans un cadre algébrique disposant d'une large gamme de méthodes de résolution, que nous utilisons pour la validation des propriétés et la génération de codes corrects.

Chapitre 3

Sûreté par analyse, transformation et instrumentation de programmes

---

Ce chapitre présente :

— Les analyses de vérification des déclarations de tableaux (section 3.1).

— Les analyses de vérification du non-débordement des accès aux éléments de tableaux (section 3.2).

— Les analyses de détection des variables non-initialisées (section 3.3).

— Nos expériences de validation et optimisation d'applications (section 3.4).

Les principaux critères de qualité logicielle visés sont : le respect des normes, la vérificabilité et la détection des erreurs, la maintenabilité, la robustesse et l'efficacité.

Ces recherches ont été réalisées en collaboration avec F. Irigoin, F. Coelho, l'étudiante en thèse Nga Nguyen et les étudiants en Master recherche Anh Trinh Quoc et Quoc Dat Pham.

Mes publications relatives à ce chapitre sont les suivantes :[11, 13, 30, 14, 22, 23, 25, 27, 43, 45, 137, 146, 175, 176].

Ce chapitre rassemble les travaux qui sont liés à la sûreté par l'analyse et la transformation. Leur objectif est la détection des erreurs dans les applications scientifiques de taille réelle.

Parmi les erreurs classiques, nous ciblons l'utilisation de variables non initialisées et les dépassements d'accès à la zone allouée pour un buffer. Elles conduisent, selon les cas, à des résultats indéterministes ou à des exceptions (*over-flow, under-flow*, ...) qui fragilisent le logiciel et permettent d'éventuelles intrusions et attaques.

Le coût du développement logiciel étant croissant en fonction des phases de développement, il est préférable d'effectuer les analyses de détection des erreurs le plus tôt possible. L'analyse statique est l'une des méthodes formelles les plus appropriées pour traiter des applications de taille réelle. Elle permet d'extraire des propriétés qui sont toujours vérifiées indépendamment des valeurs prises par les variables de ce programme lors de l'exécution. Elle caractérise toutes les exécutions possibles. Mais comme il n'est pas toujours possible de prouver *l'atteignabilité* d'un état, elle approxime l'ensemble de ces états et peut conduire parfois à de nombreux faux négatifs. A contrario, l'analyse dynamique collecte des informations sur les valeurs du programme au cours de l'exécution et les utilise pour valider la propriété. Elle est très précise puisqu'elle utilise les informations des états courants du programme mais soulève le problème de la mise à disposition d'un ensemble de tests en entrée suffisamment exhaustif pour valider l'analyse.

La thèse de Nga Nguyen, que j'ai co-encadrée avec F. Irigoin, avait pour thème les *vérifications efficaces des applications scientifiques par analyse et instrumentation de code.* Elle propose des approches combinant à la fois les analyses statiques et dynamiques. Les trois premières sections de ce chapitre présentent ces analyses : la normalisation des déclarations de tableaux en section 3.1, la vérification du non-débordement des accès aux éléments de tableaux en section 3.2, la détection des variables utilisées avant d'avoir été initialisées en section 3.3. Ces analyses, et les corrections et/ou transformations associées, sont effectuées au niveau du programme source et peuvent être utilisées très tôt dans le processus de développement.

L'ensemble des analyses présentées dans ce chapitre ont été intégrées dans `PIPS`. Elles constituent les outils essentiels d'un *atelier de maintenance* qui a été déployé dans un cadre industriel pour traiter des codes plus anciens. Les résultats des travaux effectués sont présentés en section 3.4.

3.1 Vérification des déclarations de tableaux

Au delà du caractère purement *esthétique*, certaines règles d'écriture des programmes ont un effet sur leur lisibilité, leur robustesse et leur maintenabilité. C'est le cas pour les déclarations de tableaux. Il est de la responsabilité du programmeur d'allouer un emplacement suffisant pour les tableaux qui sont lus et écrits par l'application au cours de leur exécution. Et de nombreux langages tels que C, JAVA et Fortran autorisent les déclarations sans que la première dimension soit précisée. En C, il est possible de déclarer un tableau `A` d'entiers par `int A[][10]` ou `int *A[10]`. En Fortran, il s'agit de la dernière dimension, car l'allocation est faite par colonnes, et les deux types de déclaration `integer A(10,*)` ou `integer A(10,1)` sont fréquentes, même si le nombre d'éléments référencés sur cette dimension est grande.

Ce type de déclarations altère l'exactitude des résultats des analyses statiques qui peuvent être appliquées en donnant une information incorrecte ou insuffisante. Elles limitent ou empêchent l'application de la vérification du non-débordement des accès aux éléments de tableau, la détection de l'*aliasing* et de la parallélisation. Notons que les compilateurs/debogueurs classiques les utilisent via les options *-qcheck* de `idb`, l'option *-C* de SUN ou encore *-Wall* de `gcc`.

L'article [43] et la thèse de Thi Viet Nga Nguyen [135] présente deux méthodes d'analyse statique qui précisent et redéfinissent la taille exacte des tableaux référencés, lorsque c'est possible. La première utilise les relations entre les paramètres formels et les paramètres réels pour proposer de nouvelles déclarations dans la fonction appelée. La deuxième est basée sur l'analyse des régions de tableaux (section 2.5,p.26) qui caractérisent l'ensemble des éléments de tableaux qui sont référencés par un ensemble d'instructions.

Ces deux algorithmes sont combinés pour extraire le maximum d'information en tenant compte de styles de programmation potentiellement différents.

L'algorithme Top-Down : Cet algorithme propage de manière **descendante** sur le graphe des appels, les déclarations numériques ou symboliques rencontrées à partir du programme principal. Les dimensions des tableaux peuvent différer entre la routine appelante et la routine appelée car l'appelé peut n'utiliser qu'une partie du tableau, voire sous une autre forme (*reshaping*). Il peut aussi y avoir plusieurs appels à une même routine. Dans toutes les configurations, la taille du tableau dans la routine appelée doit être

inférieure à celle de l'appelant. Les déclarations connues sont traduites du domaine des variables de l'appelant vers celui de l'appelé. Elles sont ensuite propagées de manière interprocédurale suivant le graphe des appels.

Cette méthode s'applique bien aux programmes pour lesquels les déclarations initiales sont implicites dans le programme principal. Ce qui n'est pas toujours vrai, notamment pour les routines de bibliothèque. Lorsqu'il n'est pas possible de déterminer de manière exacte la taille de la dernière dimension du tableau, l'algorithme retourne une valeur indéterminée, représentée par `*` en Fortran.

L'algorithme Bottom-Up : Cet algorithme propage de manière **ascendante** sur le graphe des appels, les déclarations numériques ou symboliques calculées à partir des *régions* de tableaux (section 2.5,p.26). Les zones de tableaux référencées en écriture et en lecture pour une routine sont fusionnées et donnent la taille de la zone mémoire à allouer au tableau pour la routine, en supposant que le programme est correct. Cette valeur est ensuite propagée récursivement aux appelants [45] afin d'instancier les différentes déclarations intermédiaires du tableau.

La qualité de ces résultats est intrinsèquement liée à l'exactitude des résultats des analyses des régions. Les sources d'imprécision proviennent des structures de contrôle du programme où il faut regrouper les régions (union de branches) et des expressions d'accès aux éléments de tableaux non linéaires (les régions ne peuvent plus s'exprimer sous formes de polyèdres). Une approche conservatrice est utilisée. Et lorsque les *régions* de tableaux ne sont pas suffisamment précises, l'algorithme retourne une valeur indéterminée.

Cet algorithme est adapté aux programmes pour lesquels on ne dispose pas nécessairement du code du programme principal, plus particulièrement pour toutes les routines utilisées comme *bibliothèque*. Il est également intéressant pour les tableaux alloués dynamiquement ou ceux déclarés de type pointeur.

Cette phase étant ascendante et donnant l'ensemble des éléments référencés par une routine, il est aussi possible de vérifier avec une option, les déclarations *locales* numériques et symboliques de la routine.

Les résultats : Ces analyses ont été appliquées sur les *benchmarks* de Linpack, Perfect Club et SPEC CFP95. Sur la totalité des déclarations, 23% comportaient un `1` et 37% une `*`. Les analyses ont permis de préciser 79% de ces déclarations.

Mais l'intérêt important du redimensionnement automatique des tableaux est de permettre l'application d'autres analyses statiques de programmes sans qu'elles soient potentiellement *trompées* par des déclarations imprécises voire erronées de type `A(1)` en Fortran.

3.2 Vérification des accès aux éléments de tableaux

De nombreuses attaques proviennent de l'utilisation de failles du logiciel par lesquelles il est possible de produire un dépassement de buffer. L'état qui en résulte permet l'intrusion.

Les débordements de tableaux proviennent le plus souvent :

1. de déclarations imprécises des arguments *formels* d'une fonction (bornes supérieures à 1 pour les déclarations Fortran présentées dans la section précédente).

2. de déclarations imprécises de la borne inférieure d'arguments *formels* (accès en 0 en Fortran).
3. de déclarations de tableaux *locaux* numériques erronées.
4. d'astuces de programmation, comme l'initialisation des éléments d'un tableau multidimensionnel en parcourant la première dimension sur une plage de valeurs équivalente à la taille complète du tableau.
5. d'erreurs.

Ces débordements peuvent fausser les résultats de manière systématique (recouvrement avec d'autres variables) ou non déterministe. Les astuces de programmation ne sont pas nécessairement source d'erreurs lors de l'exécution (cela dépend des optimisations choisies), mais elles faussent systématiquement le résultat des analyses de programme. Les problèmes s'observent souvent lors de la phase de transformation et parallélisation.

Les publications [138, 136, 135] présentent les trois algorithmes de tests de débordements aux accès de tableaux introduits dans `PIPS`. Ces algorithmes sont plus complets que les options classiques proposées par les compilateurs. En effet, l'option *subscript check* ne détecte pas les débordements *inter*-procéduraux et reste une analyse essentiellement dynamique, donc très dépendante des cas tests fournis. Les seuls débordements statiques détectés correspondent à des références numériques entières dépassant la borne numérique de la déclaration.

D'autres options proposées par le compilateur ne sont pas toujours applicables, car elles stoppent dès le premier débordement rencontré et causent un arrêt prématuré de l'exécution. Pour tester toute l'application, il faut effectuer les modifications une à une, puis relancer le test après chacune d'elle. A ce titre, l'utilisation de l'algorithme de redimensionnement automatique des déclarations permet d'éliminer les faux-positifs pour les déclarations de type pointeur A(1) en Fortran.

Les trois algorithmes de détection de débordements de `PIPS` sont intra et/ou inter-procéduraux. Ils allient des analyses statiques et dynamiques. En intra-procédural, on vérifie que les références aux tableaux sont bien incluses dans les intervalles de déclarations locales. En inter-procédural, on vérifie les appels de fonction, la déclaration du paramètre formel doit être plus petite ou égale à la déclaration du paramètre réel associé.

Le premier algorithme génère des tests systématiques de débordement aux accès de tableaux. Les tests sont ensuite éliminés avec les autres phases de `PIPS` (élimination de code mort, élimination des tests logiques redondant ou inutiles) lorsqu'il est possible de conclure qu'ils sont inutiles. Les préconditions et les régions étant utilisées, le contexte d'exécution permet de détecter ou d'éliminer statiquement certains débordements. Lorsqu'il n'y a pas possibilité de conclure, un test dynamique est exécuté à l'exécution.

Le deuxième algorithme n'insère des tests de débordement que lorsque c'est nécessaire. Il est basé sur les résultats fournis par les régions. C'est un algorithme *top-down* qui commence par le nœud le plus haut dans le graphe de contrôle hiérarchique. Les propriétés sur les régions sont utilisées. Selon l'exactitude des résultats fournis, il est possible de conclure s'il y a réellement débordement ou pas, et s'il faut ajouter un test pour vérifier dynamiquement l'éventuel débordement. Ce second algorithme instrumente la version avec moins de tests. Il est plus précis que l'algorithme précédant, mais est aussi plus coûteux car il utilise les résultats des régions convexes de tableau.

Le troisième algorithme effectue la vérification interprocédurale des déclarations. Toute déclaration d'un paramètre formel doit être plus petite ou égale à la déclaration du paramètre réel associé.

Pour ces trois algorithmes,

1. S'il y a débordement, une instruction `STOP` (Fortran) commentée est introduite juste avant l'accès au tableau ou avant le `CALL`.
2. S'il n'y a pas débordement, aucune instruction n'est ajoutée.
3. Pour les deux derniers algorithmes, s'il n'est pas possible de conclure, un test logique caractérisant les accès débordants éventuels est ajouté avant l'instruction `STOP`.

Les résultats : Les résultats expérimentaux ont montré que :

— Le second algorithme est efficace sur des programmes de taille réelle. Il insère les vérifications au plus tôt, élimine un plus grand nombre de tests dynamiques que le premier algorithme et accélère ainsi les temps d'exécution.

— Une réduction du temps de compilation et du temps d'exécution notable en comparant `PIPS` aux compilateurs tels que f77 et celui de SUN pour des capacités de diagnostic équivalentes.

Ces analyses statiques et dynamiques du code contribuent à l'obtention d'un code en Fortran de qualité et plus sûr. Elles pourraient également être appliquées comme ADA, C, JAVA, etc, mais les algorithmes doivent être étendus pour tenir compte des pointeurs, de la récursion des fonctions et des structures de données. Certaines analyses, telles que le redimensionnement des tableaux, peuvent aider à propager la taille des tableaux lorsque les tableaux ou les pointeurs sont passés comme paramètres formels et permettre ainsi la vérification sur ces structures de données. L'outil de compilation et d'optimisation *Velociraptor* [87], ciblant les architectures CPU et GPU, utilise des techniques similaires à nos deux premiers algorithmes intraprocéduraux pour l'élimination des tests de vérification des bornes de tableaux.

3.3 Détection des variables non initialisées

L'une des erreurs les plus communes en programmation est l'utilisation de variables qui ne sont pas initialisées. Selon les compilateurs et leurs options, des valeurs indéterminées ou nulles sont retournées lors de leur référencement. Le programme peut donc produire des résultats incorrects, faire référence à des adresses mémoire indéterminées ou encore conduire à des comportements imprévisibles.

Certains langages de programmation tels que Java et C ++ ont intégré des mécanismes qui assurent dynamiquement l'initialisation des variables avec une valeur par défaut. Cela garantit une cohérence au programme, mais ne correspond pas nécessairement aux résultats escomptés par le programmeur. Certains compilateurs permettent la vérification mais pas pour tous les types de variables. Une valeur par défaut prédéfinie comme zéro n'est pas valide pour tous les types de variables, les booléens par exemple.

Pour détecter les variables non-initialisées avant d'être utilisées, l'article [137] présente une combinaison de deux algorithmes, l'un statique et l'autre dynamique. L'analyse statique seule est insuffisante en raison des flux complexes de données et de contrôle des programmes, des références non-linéaires à des éléments de tableaux ou à des indirections

qui peuvent conduire à de nombreux faux messages d'avertissement. A l'opposé, l'analyse dynamique, vérifiant la bonne initialisation de l'ensemble des variables du programme au cours de l'exécution, peut être très coûteuse car elle ne tient pas compte de l'information connue à la compilation. Le ralentissement entre les codes non-instrumentés et instrumentés mesuré dans [120] peut aller jusqu'à un facteur de 100.

Les régions de tableaux *IN* L'ordre dans lequel les références aux éléments du tableau sont effectuées est essentielle pour l'optimisation du programme. Les régions IN de tableaux ont été introduites dans [74, 73] et résument l'ensemble des éléments d'un tableau dont les valeurs sont *importées* par un fragment de code. Un élément de tableau est importé si il existe au moins une utilisation de cet élément dont la valeur n'a pas été définie précédemment au sein du fragment lui-même.

Les régions IN sont calculées en commençant par les instructions élémentaires et en remontant l'information aux instructions englobantes comme les conditionnelles, les boucles, les séquences et inter-procéduralement à travers les appels de fonctions. À chaque site d'appel, le résumé des régions IN de la fonction appelée est traduit dans l'espace de noms de l'appelant, en utilisant les relations entre paramètres réels et paramètres formels, et leurs déclarations.

L'algorithme d'analyse des variables utilisées avant d'avoir été initialisées est directement fondé sur les régions IN. Ces régions sont calculées pour les tableaux et les scalaires.

La liste des régions IN au niveau de l'entrée des modules en Fortran donne l'ensemble de toutes les variables potentiellement non définies dans le module. Ainsi, à l'entrée du module, si la liste des régions IN est vide, c'est qu'il n'y a pas de variables non-initialisées utilisées dans ce module et donc pas d'erreur dans ce module. Dans le cas contraire, chaque variable dans la liste doit être vérifiée. Les régions IN de toutes les variables globales sont propagées dans le programme principal. En fonction de la portée et la nature des variables : locales, globales ou paramètres formels, le traitement de l'information sera différent :
Cas 1 : pour les variables locales ou globales dans le programme principal. Selon la précision de la région, nous avons deux sous-cas :

- Si la région est *EXACT*, la variable est utilisée quelque part dans le module avant d'être définie et une erreur est détectée.
- Si la région est *MAY*, le code est instrumenté. Une fonction d'initialisation est ajoutée avant la première instruction du module, puis juste avant les instructions *important* cette variable de manière exacte une fonction de vérification est insérée. Pour chaque région *MAY* l'analyse continue. S'il s'agit d'un appel de procédure, une information est ajoutée pour indiquer que les paramètres formels doivent être vérifiés au niveau de l'appelé.

Cas 2 : pour les paramètres formels ou variables globales dans une procédure appelée. Si la variable est marquée comme *devant être vérifiée*, le même processus que précédemment est appliqué, mais aucune initialisation est ajoutée car elles ont déjà été effectuées dans l'un des appelants.

L'initialisation mise en œuvre attribue une valeur particulière à la variable scalaire ou à chaque élément de tableau potentiellement non initialisé. Les contrôles s'effectuent par vérification de la valeur de la variable égale ou non à cette valeur particulière. Si la variable est un tableau, le code d'initialisation et de vérification instrumenté est un nid

de boucles correspondant à la région IN, alors qu'il correspond à une instruction simple pour les variables scalaires.

L'instrumentation du code donne des informations précises des erreurs détectées : le nom des procédures et des variables, les lignes de code incriminées.

L'efficacité de cette analyse dépend de la précision de l'analyse des régions. Plus les régions de tableaux sont précises, et moins il y a de vérifications dynamiques à effectuer.

Les résultats : Ces algorithmes ont été intégrés dans `PIPS` et vérifient tous les types de variables. Analyses statiques et dynamiques se complètent mutuellement. Leur combinaison permet de réduire considérablement le nombre de variables à initialiser et à contrôler (sur les *benchmarks* étudiés, 3% des scalaires et 38% pour les éléments de tableaux). La phase statique permet de détecter des erreurs dès la compilation. Les contrôles dynamiques sont insérés uniquement lorsque les informations ne sont pas disponibles pour effectuer la vérification à la compilation. L'instrumentation de la vérification pour les tableaux est réduite aux parties des tableaux utilisées (région IN correspondante) et donc pas nécessairement et inutilement sur la totalité des tableaux.

Les expériences sur les *benchmarks* Fortran SPEC95 [137, 135] ont donné des résultats excellents autant sur le coût de l'analyse que pour les temps d'exécution. Cette méthode peut être appliquée à d'autres langages de programmation comme le langage C. Cependant, il faut étendre les algorithmes pour traiter les problèmes liés aux pointeurs, à la portée des variables et à la récursivité des fonctions pour le calcul des régions.

Les travaux présentés dans l'article [177] introduisent aussi une analyse statique pour accélérer les analyses dynamiques de détection des variables non initialisées pour des programmes C, mais les tableaux sont considérés comme des entités scalaires.

3.4 Expériences

Toutes les analyses présentées ont été intégrées dans l'environnement `PIPS`. Elles ont fait partie d'un *atelier de maintenance* que j'ai déployé dans un cadre industriel pour des applications de tailles réelles. L'objectif principal de ces vérifications était d'analyser les codes et de rechercher d'éventuelles anomalies avant l'exploitation en production de ces applications.

Un second objectif était l'optimisation des temps d'exécution. Les architectures visées étaient des machines mono-processeur vectoriel et des super-calculateurs de type MIMD à gros grain de parallélisation. Le but était d'aider à la parallélisation (vérification) de parties de code déjà identifiées comme de "bons candidats" à une exécution parallèle.

Les codes étaient écrits en Fortran, langage largement utilisé dans la communauté scientifique (mathématiciens, physiciens, chimistes, géologues, biologistes) car il possède des bibliothèques disponibles et optimisées qui se chiffrent en millions de lignes.

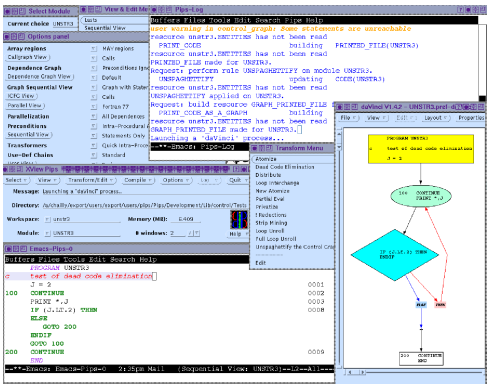
Ces expériences ont été réalisées sur plusieurs années. Je résume dans cette section quelques points clés qui ont marqué et influencé nos travaux.

3.4.1 Objectif : traiter des programmes de taille réelle.

A titre d'anecdote, la première application que nous avons étudiée, dans le cadre de ces projets, comptait 120 000 lignes de codes, 600 procédures, 18000 déclarations de tableaux, et un graphe des appels [1] des fonctions représentant 82 pages de texte ! Il n'était donc pas possible d'essayer de prouver la correction d'une propriété manuellement.

PIPS est un compilateur source à source *interprocédural*. Les résultats des analyses sont stockés en mémoire et dans des fichiers sous forme textuelle, pour chaque module du programme. Le *passage à l'échelle* pour ces gros programmes n'a pu se faire que grâce à des optimisations de l'implantation des analyses : 1) Certaines structures de données, sous forme de listes bien adaptées pour des *petits* programmes, ont été remplacées par des tables de hachage plus rapides d'accès quand il y a de nombreuses entrées ; 2) Le volume des résultats des analyses, proportionnels à la taille de programme, a été compressé. Ces transformations, réalisées en grande partie par F. Coelho, ont permis de réduire jusqu'à un facteur 100 les temps d'exécution des analyses pour les plus gros programmes.

Afin de représenter, les graphes de dépendance, les graphes d'appels des fonctions ou encore les graphes de flot de contrôle de ces gros programmes, des versions graphiques ont été développées. Elles ont été réalisées en collaboration avec un étudiant en master de recherche ; Quoc Dat Pham [146] . L'outil *daVinci* a été utilisé et on peut en voir un exemple sur la figure de droite.



3.4.2 Vérifier les codes

Afin de faciliter la maintenance des codes, et vérifier leur conformité par rapport à la norme du langage Fortran, les analyses suivantes, toutes disponibles dans PIPS, ont été effectuées systématiquement :

- Vérification du nombre d'arguments des fonctions.
- Vérification du typage des paramètres formels et réels des fonctions. La compilation des programmes s'effectuant de manière modulaire, si les types des paramètres formels et réels sont incohérents, ils peuvent engendrer des résultats erronés pour certaines options de compilation. Par exemple, un passage par registres des arguments en cas d'option d'optimisation forte conduirait à des erreurs.
- Normalisation des déclarations des programmes écrits en Fortran (section 3.1). Elle a permis d'éliminer les artefacts de mise au point des vieux codes et d'utiliser les analyses statiques classiques sur ces programmes. Dans l'une des applications comportant une dizaine de milliers de déclarations, elle a permis de préciser 733 déclarations qui n'étaient pas à la norme et de corriger 9 erreurs.
- Détection des débordements des accès aux éléments de tableaux (section 3.2).
- Détection des variables non initialisées avant leur utilisation (section 3.3) pouvant causer par la suite des divisions par zéro.

1. graphe du flot de contrôle entre les fonctions du programme.

— Détection des *alias* [135, 139, 146]. La norme Fortran 77 précise que si un appel de fonction cause un *alias* entre deux paramètres formels ou entre un paramètre formel et un élément de `COMMON`, aucun des éléments *associés* ne doit être modifié par le code de la fonction ou par n'importe qu'elle procédure faisant référence à cette fonction. Cette propriété doit être vérifiée si le compilateur veut appliquer certaines optimisations, notamment la parallélisation.

Chacune de ces analyses a permis de détecter des erreurs du code qui n'étaient pas détectées par les compilateurs classiques. Une fois que les codes sont *sûrs*, des optimisations peuvent être appliquées.

3.4.3 Optimisations

Les codes fournis pour l'optimisation étaient des codes de simulation de mécanique des structures, de mécanique des fluides et de physique. Ces applications de simulation sont très gourmandes en temps de calculs. Mon objectif a été de le réduire.

Les options d'optimisation des compilateurs ont été utilisées lorsqu'elles permettaient de vérifier le code ou proposaient des transformations pour réduire les temps d'exécution. Les options utilisées étaient :

-xtypemap permet de préciser la taille des réels, entiers et doubles. Elle peut éviter les problèmes de typage entre paramètres formels et réels.

-E stoppe après la phase de pré-processing et permet de vérifier la bonne intégration des *macros* notamment.

no-default-inlign n'*inline* pas systématiquement les fonctions.

std=standard précise les spécifications du langage d'entrée (Pour `gcc`, l'option `-ansi` équivaut à `std=c90`

-O1, -O2, -O3, -fast activent toute une série d'options dans le but d'augmenter la performance du programme. Le niveau le plus élevé *-O3* effectue des optimisations spécifiques à la machine cible. Elle peut augmenter les temps de compilation. Ses optimisations permettent également des modifications dans l'ordre des calculs (surtout en *-O3 et -fast*), la précision des résultats et la vérification de la convergence des opérateurs doivent être vérifiées.

Adaptation d'un logiciel à son utilisation réelle. L'objectif était de spécialiser un code de simulation de physique, permettant de calculer les résultats du problème en 3 dimensions ou 2 dimensions, pour son utilisation en version 2D uniquement. Le but était d'en faciliter la rétro-ingénierie et de réduire son temps d'exécution.

La phase de spécialisation 2D a permis de produire une version expurgée, spécialisée pour traiter uniquement les cas tests 2D.

En accord avec les experts de l'application, les composantes représentatives de cette version 2D ont pu être identifiées. Il s'agissait d'une liste de constantes/paramètres symboliques. L'objectif était de tenir compte de ces paramètres tout en permettant le contrôle de leur valeur. Les phases d'analyses et d'optimisation du programme ont ensuite appliquées, afin de propager les valeurs numériques des paramètres choisis et d'éliminer les parties de code mort. Ce cycle a été appliqué plusieurs fois sur le code afin d'éliminer le maximum de code inutile et obtenir un code plus compact. Après spécialisation, nous avons obtenu sur cette application :

— L'élimination d'instructions qui n'étaient pas exécutées en version 2D : 4908 instructions supprimées sur 55491 , soit 9%

— Un gain en temps de 10%

Parallélisation à grain fin ou moyen Pour les applications étudiées, seules les routines représentant un fort pourcentage du temps d'exécution globale de l'application ont été optimisées. Leurs nids de boucles étaient suffisamment bien écrits pour que les outils de parallélisation/vectorisation automatiques puissent vérifier les choix proposés. Nous avons juste suggéré 1) quelques changements dans l'ordre des boucles pour améliorer l'utilisation de la mémoire, 2) des distributions ou fusion de boucles pour améliorer l'utilisation du cache.

Parallélisation à gros grain. L'objectif était d'aider à la parallélisation de trois applications.

L'architecture visée était de type multiprocesseurs MIMD à mémoire partagée. Le parallélisme était à gros grain et était exprimé à un niveau élevé dans le graphe d'appel de l'application.

Pour ce type d'applications et ce type de parallélisme, nous n'avions pas de recette générique. Nous avons utilisé les résultats des tests de dépendances, des *use-def chains* interprocédurales, développé certaines transformations de programme telles que le *clonage de fonction*, et utilisé des techniques pour spécialiser le code qui ont permis de prouver que les branches parallèles des applications n'opéraient pas (ou opéraient encore) sur les mêmes variables du programme.

Le premier programme comportait, dans deux sections distinctes, des lectures et mises à jour d'une même variable. La détection de ces dépendances de flots de données empêchait l'exécution en parallèle des deux sections. Les lectures et mises à jour de la variable ont pu être déplacées dans la partie séquentielle du programme exécutée en amont des deux sections. Les deux sections ont pu être parallélisées.

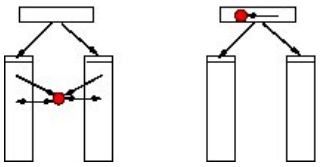


FIGURE 3.1 – Dépendances de données

Dans le deuxième programme, une même fonction était chargée à la fois : de l'initialisation, des lectures et des mises à jour de tableaux en fonction des valeurs d'un de ses paramètres formels. Cette fonction était appelée de nombreuses fois par le programme et concentrait toutes les dépendances de données en empêchant une parallélisation à gros grain. La fonction a été *clonée* en fonction des valeurs de ce paramètre formel. Trois nouvelles fonctions ont été ainsi créées, et le programme complet a été spécialisé pour n'appeler que ces trois nouvelles fonctions. Après transformation automatique, les nouvelles dépendances des parties du code, pressenties pour une exécution parallèle, ne portaient plus que sur des lectures de tableaux. Une parallélisation à gros grain a pu être effectuée.

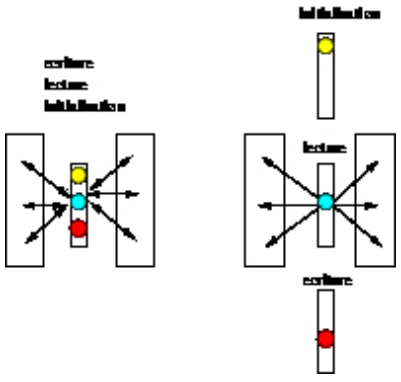


FIGURE 3.2 – Clones

Le troisième programme calculait deux parties distinctes d'un même tableau. Une version parallèle du programme avait déjà été proposée. Cependant, les exécutions parallèles de ces calculs donnaient des résultats non-déterministes. Les calculs étaient inclus dans 3 niveaux de boucles imbriquées et semblaient être de bons candidats pour la parallélisation. Mais, au sein des calculs, une même variable correspondant à un seuil était modifiée et utilisée. La détection de cette dépendance a permis d'expliquer les raisons du non-déterminisme.

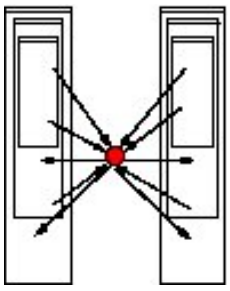


FIGURE 3.3 – Contrôle

Ces optimisations, moins systématiques que la parallélisation vectorielle, ont pu être effectuées en combinant les analyses automatiques déjà développées et en y associant quelques nouvelles transformations telles que la spécialisation de code et le *clonage*. Afin de poursuivre sur ce thème de l'analyse et de la vérification des applications, l'un de mes axes de recherche pour les prochaines années est l'extension de ce type d'analyses pour les langages parallèles et codes déjà parallélisés.

3.5 Conclusion

Les analyses développées ont été testées sur des codes industriels de taille réelle. A-t-on répondu aux attentes en terme de sécurisation et vérification des applications ? Oui, grâce aux analyses et optimisations automatiques de programmes développées et intégrées dans `PIPS`. Les points forts que nous avons retenus de l'ensemble de ces travaux sont les suivants :

Source-à-source Pour les programmes industriels, il est essentiel de pouvoir proposer des optimisations qui puissent être intégrées au code *source* initial. Ces corrections et transformations doivent être effectuées et validées très tôt dans le processus de développement industriel.

Les analyses de PIPS sont intraprocédurales et interprocédurales et des résumés de leurs résultats sont stockés pour chaque fonction. Ces informations n'ont pas besoin d'être recalculées à chaque appel de fonctions. Parmi les techniques classiques d'analyse qui ne traitent pas de l'interprocédural, les techniques d'*inlining* ne sont pas applicables sur des gros codes. Les autres n'effectuent pas de traduction des informations lors des appels de fonctions et fournissent des résultats moins précis. Avec les techniques interprocédurales, les temps de propagation des résultats des analyses sont réduits.

Des règles d'écriture pour les programmes [38]. Les techniques d'analyse et d'optimisation ne peuvent pas être totalement automatisées. Atteindre les meilleures performances pour une application quelle que soit la machine cible est impossible. Toutefois il est possible d'aider le compilateur à générer un code efficace : en respectant les spécifications du langage, en rendant explicite les informations que le développeur connaît concernant les structures de données (taille et dimension des tableaux) et les structures du contrôle, en évitant des astuces de programmation et les optimisations spécifiques à une architecture. Ces règles permettent un meilleure vérification du code, une recherche plus précise des informations sémantiques et

donc d'en dériver des optimisations et transformations appropriées. Nous avons défini une liste de ces règles dans le cadre du projet OpenGPU [8, 38].

Combinaison d'analyses statiques et instrumentation Les analyses dynamiques seules vérifient le code en fonction des cas tests disponibles qui sont exécutés, mais ne garantissent pas que le programme source est *sûr*.

L'association des analyses statiques et de l'instrumentation de code permet de générer des programmes sûrs. Le nombre de tests générés et exécutés lors de l'exécution est limité grâce aux résultats fournis à la compilation par les analyses statiques. L'instrumentation du code le protège contre toutes références non valides aux variables du programme, non seulement pour les cas tests disponibles mais aussi pour tous les cas tests futurs.

De nombreuses expériences ont été menées sur des *benchmarks* classiques : Linpack, Perfect Club, SPEC CFP95, et sur des applications industrielles de taille importante. Les analyses présentées ont donné des résultats originaux et précis, en raison de la combinaison d'analyses statiques et dynamiques et de l'utilisation des *régions convexes de tableaux* qui est une analyse unique au compilateur `PIPS`. Le déploiement de ces techniques a permis de vérifier les applications et de les sécuriser après détection d'erreurs potentielles. Toutes les suggestions d'amélioration des applications industrielles ont été intégrées dans leurs versions de développement ultérieures.

Suite à ces travaux, j'aimerais étendre les analyses de code dans le cadre de programmes ayant déjà une sémantique *parallèle.* Une transformation peut être appliquée sur une partie de programme si ce dernier vérifie certaines propriétés. Pour exemple, la parallélisation de boucle ne s'applique que si les dépendances de données sont respectées. De nombreuses analyses ont déjà été développées dans le compilateur `PIPS` afin de fournir les informations nécessaires à l'application d'optimisations. Ces analyses ont été réalisées pour des programmes séquentiels. Les analyses de vérification du code sont toujours correctes pour des programmes comportant des annotations ou *pragmas* de parallélisation. Mais des analyses telles que l'estimation du nombre d'opérations dans un bloc d'instructions et la caractérisation des éléments de tableaux référencés dans une section de code ont besoin d'être revisitées pour être plus précises (voire correctes) dans le cadre de sections parallèles.

Chapitre 4

Synthèse de code

Ce chapitre présente :

— La synthèse de code de contrôle à partir d'un polyèdre (section 4.1).

— La synthèse de code de communication pour une architecture à mémoire partagée émulée (section 4.2).

— Un cadre algébrique pour spécifier les distributions de type `HPF` et synthétiser les codes distribués de calculs et des communications (section 4.3).

— L'optimisation de communications pour des distributions complexes et des DMAs (section 4.4).

L'objectif est la synthèse de code à partir de spécifications algébriques pour générer des programmes sûrs. Les principaux critères de qualité logicielle visés sont : la correction, la robustesse, l'efficacité, la réutilisabilité, la portabilité et la compatibilité.

Ces recherches ont été réalisées principalement avec F. Irigoin, F. Coelho, R. Keryell, T. Petrisor et E. Lenormand.

Mes publications relatives à ce chapitre sont les suivantes : [7, 9, 10, 12, 28, 29, 32, 33, 41, 42, 44, 46, 66, 95, 101, 102, 104, 105]

La conception des programmes est coûteuse, leur vérification et leur validation sont souvent difficiles surtout lorsqu'un haut niveau de sûreté est recherché. En moyenne une dizaine d'heures sont nécessaires pour *revoir* 200 lignes de code d'après l'article de Wikipédia [1]. Même si un programme est reconnu *correct* à un instant donné, il est destiné à évoluer. La maintenance du code peut nécessiter des mises à jour de conformité par rapport aux nouvelles normes. L'évolution des besoins fonctionnels est bien entendu le plus important facteur de changements, mais l'impact des nouvelles architectures est également non négligeable. De plus en plus d'architectures sont parallèles et/ou hétérogènes et imposent des changements profonds du code.

Générer du code parallèle est une optimisation qui vise à réduire le temps d'exécution du programme. Mais générer du code parallèle reste un problème très complexe, même pour les programmeurs avertis. Après avoir extrait le parallélisme potentiel et avoir choisi *la* transformation qui sera appliquée au code, en vue de son optimisation ou placement sur l'architecture cible, l'étape critique est celle de la génération du code.

L'une des options pour générer du code correct est de le générer automatiquement (le synthétiser) à partir de spécifications claires. Si ces spécifications sont mathématiques

1. Revue de code - Wikipédia

et ont été correctement définies, il est possible de valider les étapes de la génération automatique et de garantir ainsi la qualité du code.

L'un de mes axes de recherche est la synthèse de code pour des architectures parallèles. Depuis les années 70, la diversité des architectures parallèles n'a fait qu'augmenter et je citerai à titre d'exemples : les multiprocesseurs (CRAY T3D) et grappes de processeurs, les multi-cœurs (CELL - PS3), les GPU largement utilisés pour les jeux vidéos (nVidia G80) et les FPGA circuits logiques programmables, les multi-cœurs et manycore (MPPA)... Les codes qui ciblent ce type d'architectures doivent intégrer des directives de parallélisation et les communications, qui sont nécessaires aux transferts de données lors de l'exécution des calculs répartis sur les différents processeurs ou coprocesseurs.

Les optimisations utilisées par les paralléliseurs pour générer du code parallèle efficace s'effectuent en général en trois étapes : 1) l'analyse des dépendances donne les conditions pour préserver la sémantique du programme initial ; 2) une transformation est choisie afin d'optimiser (en temps ou en espace) l'application ; 3) la transformation est appliquée au programme.

Le potentiel d'optimisation pour les programmes scientifiques se situe essentiellement dans les nids de boucles. Et dès les années 70 [2, 173], de nombreuses techniques ont été développées pour optimiser l'ordre des calculs au sein des nids de boucles : échanges[171, 3], distribution, fusion, *tiling*[2] [103, 106], *strip-mining*[3] [170], *skewing*[4] [172, 154], ... qui permettent non seulement d'augmenter le parallélisme potentiel mais aussi de réduire 1) la taille de mémoire nécessaire à l'exécution et 2) les temps d'accès aux mémoires (locales, caches, globale) en améliorant la localité spatiale ou temporelle.

Dans de nombreux cas, ces transformations de boucles peuvent être exprimées comme des transformations affines d'ensembles de points entiers représentés par des polyèdres. A partir du polyèdre transformé, il faut générer le nouveau code correspondant. La section 4.1 présente les techniques que nous avons développées pour générer automatiquement le code référençant tous les éléments d'un $\mathcal{Z}$-polyèdre. Ces techniques ont ensuite été complétées pour tenir compte de nouvelles caractéristiques architecturales et proposer des codes générés appropriés et efficaces.

En effet, une grande partie de mes travaux a été consacrée à la génération de code pour des multiprocesseurs à mémoire distribuée. Les communications inter-processeurs peuvent aussi être représentées par des polyèdres. Les trois sections suivantes résument mes travaux sur la synthèse de code pour ce type d'architectures. La section 4.2 présente la synthèse de codes de calculs et de communications pour une architecture avec une mémoire partagée émulée sur des composantes distribuées. La section 4.3 introduit les spécifications qui ont permis de synthétiser du code pour des applications `HPF`[5]. Enfin, la section 4.4 propose une méthode pour synthétiser les communications dans le cas de redistributions quelconques des données, éventuellement redondantes[6] et optimisées pour les transferts avec DMA[7].

2. Le *tiling* réorganise un espace d'itérations en tuiles dont la taille est optimisée, pour le cache en général.

3. Le *strip-mining* décompose une boucle en deux boucles.

4. Le *loop skewing* réarrange l'espace d'itérations pour que les dépendances de données portent sur une boucle particulière.

5. HPF (*High Performance Fortran*) est un langage de programmation pour les machines parallèles. Il est basé sur Fortran 90

6. Une donnée redondante est une donnée présente sur plusieurs mémoires distribuées.

7. Un DMA (*Direct Memory Access*) est un composant informatique permettant de transférer des données, directement par un contrôleur adapté, en même temps que les calculs.

4.1 Synthèse de code à partir d'un *Z-polyèdre*

Pour les programmes scientifiques, le plus fort taux de parallélisme se situe dans les nids de boucles. De nombreuses optimisations de ces nids de boucles peuvent être exprimées comme des transformations affines sur des ensembles d'entiers définis par des polyèdres. L'objectif de l'article [41] était de présenter un algorithme permettant de générer automatiquement le nouveau nid de boucles après transformation.

L'algorithme est basé sur la méthode de projection de Fourier-Motzkin. A partir d'un polyèdre défini par un système d'inéquations et d'équations linéaires, il calcule les nouvelles bornes de boucles qui permettent d'énumérer tous les points ce polyèdre.

Les sections suivantes illustrent les différentes étapes de l'application d'une transformation affine sur un nid de boucles, de la spécification mathématique des ensembles de points à énumérer, aux transformations jusqu'à la génération des nid de boucles.

4.1.1 Exemple de transformation d'un nid de boucles : le *tiling*

La figure 4.2 représente le stencil de points utilisés par chaque itération du programme de gauche. Pour réduire le nombre de communications par rapport aux nombre de calculs sur une machine à mémoire distribuée, le *tiling* optimal est hexagonal [166].

```
DO I = 1, N
  DO J = 1, N + 1 - I
    A(I,J) = PHI(A(I-2,J),A(I-1,J),A(I,J),
                 A(I+1,J),A(I+2,J),A(I,J-2)
                 A(I,J-1),A(I,J+1),A(I,J+2))
  ENDDO
ENDDO
```

FIGURE 4.1 – Programme

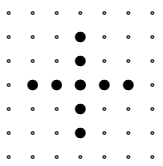


FIGURE 4.2 – Pattern des 9 points référencés

Ce *tiling* est illustré sur la figure 4.3. L'espace d'itérations du programme est triangulaire et nous prenons pour la figure `N=25`. Chaque point représente une itération du nid de la boucle. Chaque tuile est de couleur rouge et contient des itérations. Ces itérations sont définies par le système d'inégalités S suivant :

$$S = \begin{cases} -3 \leq j < 3 \\ 0 \leq i + j < 6 \\ 0 \leq i < 6 \end{cases}$$

Le *tiling* définit des tuiles qui sont toutes identiques. Dans la figure 4.3, les origines sont mises en évidence par les petits cercles bleus. Notons la diversité des types de tuiles. La tuile `C` contient 4 éléments en haut à gauche, la tuile `A` contient 27 éléments et la tuile `B` a 15 éléments situés au milieu. La tuile `D` est vide, elle ne référence aucun élément de l'espace d'itérations. Elle ne devra pas être énumérée par le nouveau nid de boucles.

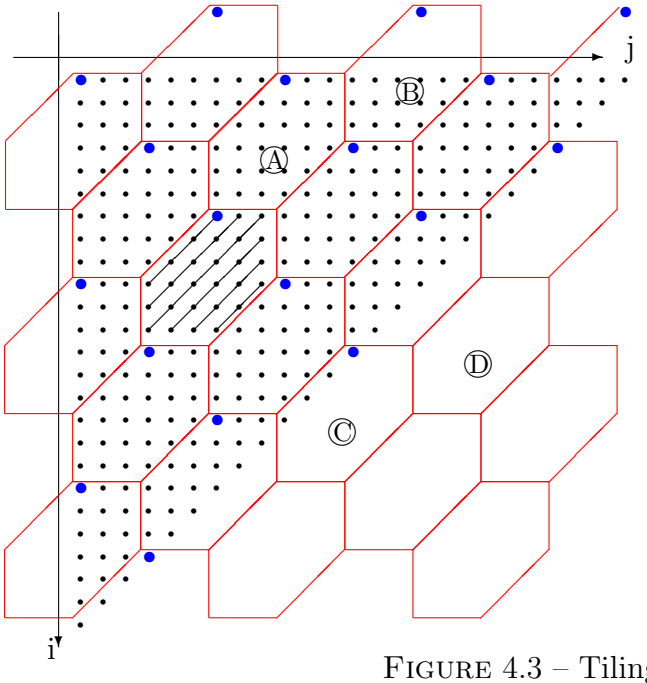


FIGURE 4.3 – Tiling hexagonal

Spécification des tuiles

Les origines des tuiles (de couleur bleue sur la figure 4.3) appartiennent à un treillis défini par deux vecteurs générateurs. Plusieurs couples de vecteurs équivalents sont possibles et la paire suivante est choisie pour expliquer d'où les tuiles non vides proviennent :

$$L = \begin{pmatrix} -3 & 9 \\ 6 & -9 \end{pmatrix}$$

L'origine du treillis dépend de l'ensemble des itérations en entrée. Pour simplifier les calculs, elle est fixée au point $(1, 1)$ (voir figure 4.3). Appelons t_1 et t_2 les coordonnées des origines des tuiles dans leur propre espace. Les coordonnées de ces tuiles sont liées aux coordonnées des itérations (i_0, j_0) par les équations suivantes :

$$\begin{cases} i_0 &= 1 - 3t_1 + 9t_2 \\ j_0 &= 1 + 6t_1 - 9t_2 \end{cases}$$

Ces équations peuvent être combinées avec un système simple B qui est dérivé des bornes de boucles du programme initial et qui définit l'espace d'itérations :

$$B = \begin{cases} 1 \leq i \leq 25 \\ 1 \leq j \leq 26 - i \end{cases}$$

Afin de caractériser l'ensemble des tuiles non-vides, nous définissons le système qui caractérise l'ensemble des points entiers appartenant à une tuile d'origine (i_0, j_0). Les tuiles (t_1, t_2) qui contiennent au moins une d'itération (i, j) vérifient :

$$\begin{cases} -3 \leq j - j_0 \leq 2 \\ 0 \leq i - i_0 + j - j_0 \leq 5 \\ 0 \leq i - i_0 \leq 5 \end{cases}$$

Les origines de la tuile (i_0, j_0) sont éliminées du système pour générer les bornes de boucles correctes pour t_1 et t_2 :

$$\begin{cases} 1 \leq i \leq 25 \\ 1 \leq j \leq 26 - i \\ -3 \leq j - 1 - 6t_1 + 9t_2 \leq 2 \\ 0 \leq i + j + 2 - 3t_1 \leq 5 \\ 0 \leq i - 1 + 3t_1 - 9t_2 \leq 5 \end{cases}$$

Pour énumérer les tuiles non-vides, les variables i et j doivent aussi être éliminées. En d'autres termes, un polyèdre à quatre dimensions sur (i, j, t_1, t_2) doit être projeté sur le sous-espace (t_1, t_2).

L'ensemble des tuiles résultantes est représenté sur la figure 4.4 ce qui se traduit par les bornes de boucles suivantes :

```
DO T1 = 0, 8
  DO T2 = (T1+1)/3, 2*T1/3
```

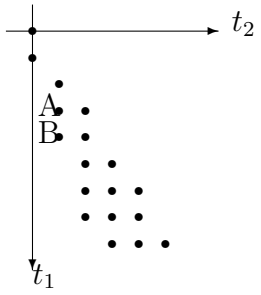


FIGURE 4.4 – Ensemble des tuiles non-vides

Dans cet exemple de *tiling* nous avions besoin d'un algorithme prenant en entrée un espace d'itérations défini par un système d'inégalités linéaires B et un *tiling* uniforme défini par un treillis L et renvoyant comme sortie les bornes de boucles B_T permettant d'énumérer les tuiles non-vides.

Soit T l'espace des tuiles, S le système définissant la forme d'une tuile et $\vec{t}$ les coordonnées d'une tuile. Soit I l'espace d'itérations, B le système dérivé des bornes de boucles, et $\vec{\imath}$ les coordonnées d'une itération. Les nouvelles bornes de boucles B_T sont définies par :

$$\{\vec{t} \in T | \exists \vec{\imath} \in I \quad s.t. \quad B\vec{\imath} \leq \vec{b} \wedge S(\vec{\imath} - L\vec{t}) \leq \vec{s}\} = \{\vec{t} \in T | B_T \vec{t} \leq \vec{b}_T\} \tag{4.1}$$

Spécification des itérations au sein d'une tuile

Au sein de la tuile, plusieurs ordonnancements des itérations peuvent être compatibles avec la sémantique du programme initial. Plusieurs transformations peuvent être appliquées sur l'espace d'itérations restreint à la tuile telles que l'échange de boucles[171, 3], la méthode hyperplane [111, 118] ou la permutation[48]. Ces transformations doivent être unimodulaires [48] de façon à assurer une bijection entre les ensembles de points entiers et conserver la sémantique du programme. Donc le changement de matrice de base, correspondant à la transformation affine, doit être unimodulaire.

Pour notre exemple, le changement de repère choisi U_1 et le changement de variables résultant permettent d'exécuter les itérations selon les 6 fronts représentés sur la figure 4.3.

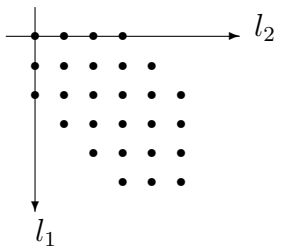
Ils remplissent ces conditions :

$$U_1 = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \qquad \begin{cases} i = l_2 \\ j = l_1 - l_2 \end{cases}$$

et définissent un nouvel espace d'itérations locales pour une tuile, représenté sur la figure 4.5. Il respecte les contraintes suivantes :

$$SU_1 = \begin{cases} 0 \leq l_2 \leq 5 \\ -3 \leq l1 - l_2 \leq 2 \\ 0 \leq l_1 \leq 5 \end{cases}$$

Comme l_1 est bornée il est facile de dériver de nouvelles bornes de boucles pour ce pattern d'exécution.




```
DO L1 = 0, 5
  DO L2 = MAX(0, L1-2), MIN(5, L1+3)
    ...
```

FIGURE 4.5 – Itérations dans une tuile

Mais ce n'est pas toujours le cas. Il y a donc besoin d'un algorithme prenant en entrée :

1. un polyèdre défini par un système d'inéquations,
2. et un ensemble ordonné de variables (une base),

et renvoyant en sortie le même polyèdre défini par un nouveau système d'inégalités tel que : chaque variable est bornée par une expression `MIN` ou `MAX` qui ne contient que des variables de rang plus élevé dans la base. Il n'y a pas de contraintes sur les variables qui ne font pas partie de la base car elles correspondent en général à des constantes symboliques. Un tel système peut être utilisé pour dériver directement les nouvelles bornes de boucles.

Spécification des communications de la tuile

Si le programme a été tuilé pour une exécution sur une architecture parallèle et s'il s'agit d'une machine à mémoire distribuée, il faut spécifier aussi l'ensemble des éléments à communiquer. Il s'agit des éléments lus ou écrits lors de l'exécution d'une tuile.

La figure 4.6 présente les éléments du tableau `A` qui doivent être transférés dans la mémoire locale d'un processeur avant de pouvoir exécuter une tuile hexagonale. Notez que cet ensemble est non-convexe. Deux éléments sont manquants dans les coins inférieur gauche et supérieur droit.

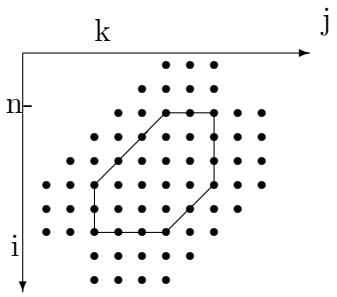


FIGURE 4.6 – Pattern non-convexe

Comme il peut être nécessaire de transférer ces ensembles de manière exacte afin de préserver la cohérence de la mémoire, il faut disposer d'un algorithme prenant en entrée :

1. un espace d'itérations défini par un système d'inégalités linéaires B,
2. et un ensemble de références R_1, R_2, ... à un tableau `A`,

et renvoyant en sortie :

1. un ensemble de boucles bornées C, éventuellement non linéaires,
2. et une référence à un tableau R de telle sorte que chaque élément de tableau accédé en entrée est communiqué une fois et une seule fois par le nid de boucles des communications.

Ce nouvel ensemble est défini pour deux références R_1, R_2 par :

$$\{\vec{a} \in A | \exists \vec{\imath}\,' \in I' \;\; s.t. \;\; C(\vec{\imath}\,') \;\wedge\; \vec{a} = R\vec{\imath}\,' + \vec{r}\} =$$
$$\{\vec{a} \in A | \exists \vec{\imath} \in I \;\; s.t. \;\; B\vec{\imath} \leq \vec{b} \;\wedge\; (\vec{a} = R_1(\vec{\imath}) + \vec{r}_1 \vee \vec{a} = R_2(\vec{\imath}) + \vec{r}_2)\}$$

4.1.2 Énumérer les points entiers d'un polyèdre

Avant de présenter l'algorithme de génération de code de parcours d'un polyèdre, quelques points importants à rappeler :

— les bornes d'un nid de boucles définissent des contraintes particulières : chaque indice de boucle ne peut apparaître que dans les bornes de boucles plus internes. Ainsi, la plupart des systèmes d'inégalités affines ne peuvent pas être directement utilisés pour générer des bornes de boucles.

— la projection, ou plus généralement l'image affine, d'un polyèdre n'est pas nécessairement un polyèdre convexe. Étant donné que des bornes de boucles linéaires génèrent uniquement des ensembles convexes, ils ne peuvent pas être utilisés pour énumérer de manière exacte ces ensembles, mais ils peuvent être utilisés pour énumérer un sur-ensemble.

— l'élimination exacte de variables par projection n'est pas toujours possible. Il peut être nécessaire d'utiliser des contraintes pseudo-linéaires, des conditions avec modulos pour ajouter un test permettant de préserver l'exactitude des points contenus dans le polyèdre et, dans le pire cas, de générer une boucle supplémentaire.

L'algorithme de *row_echelon*, que j'ai développé [41], est présenté sur la figure 4.7 :

— Le système S' définit toujours le même ensemble de points entiers. Il est initialisé avec le système initial S et est ensuite alimenté par les projections de S sur les sous-espaces de plus en plus petits. La dernière projection fournit des bornes constantes pour la boucle la plus externe, et l'avant-dernière fournit des bornes qui n'utilisent que l'indice de la boucle la plus externe, et ainsi de suite.

```
row_echelon(S)
/* S, S' and SP are lists of parametric linear
   constraints over a n-dimensional space*/
S' := SP := S;
/* compute projections in reverse order to avoid
   redundant computations */
for i := n-1 to 1 by -1 do
    SP := fourier(SP, i+1);
    S' := union(S',SP);
done
/* sort constraints in S' by increasing constraint rank */
S' := sort(S');
/* try to eliminate redundant constraints innermost
   first because they would be executed more often */
for i := number_of_constraints(S') to 1 by -1 do
    /* let S'(i) be the i-th constraints in S' */
    if not last(S',S'(i)) then
        S'(i) := complement(S'(i));
        if fourier_empty_p(S') then
            /* 0 is the trivial constraint 0 <= 0 */
            S'(i) := 0;
        else
            /* restore its original value */
            S'(i) := complement(S'(i));
        endif
    else
        /* preserve S'(i) as a unique lower or upper
           loop bound */
    endif
    S'(i) := normalize(S'(i),rank(S'(i)));
done
return S';
end
```

FIGURE 4.7 – Algorithme *row_echelon*

— Comme la projection de Fourier-Motzkin est utilisée, des points inutiles peuvent être ajoutés à tout moment dans l'espace projeté. Toutefois, le système initial S est inclus dans S' et des itérations inutilement générées par des boucles externes seront automatiquement éliminées par les contraintes sur les boucles les plus internes.
— En général, le nombre total de contraintes S' après cette première phase est important. Mais beaucoup d'entre elles sont redondantes. Cependant, elles ne peuvent pas toutes être éliminées car au moins une contrainte est nécessaire pour générer les bornes inférieure et supérieure des indices de boucles. D'où le test avec `last`.
— Le test de redondance n'a pas besoin d'être exact. Plus il est précis et plus de contraintes redondantes seront éliminées. Toutefois, les boucles générées sont toujours correctes même lorsque trop de contraintes sont utilisées. Un test de redondance très simple, appelé `fourier_empty_p`, a été développé sur la base d'un test de faisabilité utilisant Fourier-Motzkin. D'autres tests plus rapides peuvent être utilisés.
— Enfin, les contraintes doivent être normalisées pour avoir un $+1$ ou -1 en tant que coefficient pour l'indice correspondant, lorsque c'est possible.

Élimination légale par Fourier-Motzkin

Une attention particulière doit être portée lors de l'élimination des variables pour ne pas modifier l'ensemble des points entiers de l'image affine d'un polyèdre.

J'ai introduit des conditions suffisantes pour éliminer les variables de manière exacte [46]. Elles seront utilisées pour éliminer autant de variables que possible tout en conservant la linéarité de la projection et du système. Ces conditions sont données dans le théorème suivant :

Theorem 1.

$$Soit \quad S \;=\; \left\{ \begin{array}{rcl} E_p \;+\; a_{pk}\; i_k & \leq & 0 \\ E_q \;-\; a_{qk}\; i_k & \leq & 0 \\ \multicolumn{3}{c}{R} \end{array} \right.$$

$$et \quad S' \;=\; \left\{ \begin{array}{rcl} a_{pk}\; E_q & \leq & -\; a_{qk}\; E_p \\ \multicolumn{3}{c}{R} \end{array} \right.$$

où S' est obtenu à partir de S par l'élimination par Fourier-Motkin et R est un système quelconque.

$$a_{pk} = 1 \vee a_{qk} = 1 \Longrightarrow$$

$$proj(\lfloor S \rfloor, i_k) = \lfloor proj(S, i_k) \rfloor = proj(\lfloor S' \rfloor, i_k)$$

Quand aucune des conditions précédentes n'est vérifiée, il est possible d'éliminer la variable i_k en combinant les 2 inéquations et en introduisant une division entière. Soit E_p, E_q deux expressions linéaires, R un ensemble de contraintes, a_{pk} et a_{qk} des entiers positifs et i_k une variable.

Theorem 2.

$$Soit \quad S \;=\; \left\{ \begin{array}{rcl} E_p \;+\; a_{pk}\; i_k & \leq & 0 \quad (C1) \\ E_q \;-\; a_{qk}\; i_k & \leq & 0 \quad (C2) \\ & R & \end{array} \right.$$

$$et \quad S_{/k} \;=\; \left\{ \begin{array}{rcl} \frac{E_q \;+\; a_{qk} \;-\; 1}{a_{qk}} & \leq & \frac{-\; E_p}{a_{pk}} \quad (C12) \\ & R & \end{array} \right.$$

où $S_{/k}$ est dérivé de S après élimination de i_k en utilisant des divisions entières. Alors :

$$proj(\lfloor S \rfloor^2, i_k) \;=\; proj(\lfloor S_{/k} \rfloor, i_k)$$

.

Le système pseudo-linéaire $S_{/k}$ ne définit pas nécessairement un polyèdre puisque les divisions entières peuvent introduire des *trous* dans le polyèdre convexe. Par conséquent, cette opération d'élimination n'est pas une opération interne.

Exemple 1 :

$$S1 \;=\; \left\{ \begin{array}{rcl} -i_3 + 3\; i_1 & \leq & 3 \\ i_2 \;-\; 3\; i_1 & \leq & -2 \\ -i_2 & \leq & 0 \end{array} \right.$$

$$\Longleftrightarrow \qquad S1 \;=\; \left\{ \begin{array}{rcl} i_2 + 2 \leq\; 3\; i_1 & \leq & 3 + i_3 \quad (I) \\ -i_2 & \leq & 0 \end{array} \right.$$

(I) est équivalente à $\frac{i_2+4}{3} \leq \frac{3+i_3}{3}$ qui peut être réécrite $i_2+4 \leq\; 3(\frac{3+i_3}{3})+2$, ou $i_2 \leq\; 3(\frac{i_3}{3})+1$

Exemple 2 :

$$S2 \;=\; \left\{ \begin{array}{rcl} -i_2 \;+\; 3\; i_1 & \leq & 1 \\ i_2 \;-\; 3\; i_1 & \leq & 0 \\ -i_2 & \leq & 0 \end{array} \right.$$

$$\Longleftrightarrow \qquad S2 \;=\; \left\{ \begin{array}{rcl} i_2 \leq\; 3\; i_1 & \leq & i_2 + 1 \quad (I) \\ -i_2 & \leq & 0 \end{array} \right.$$

(I) est équivalente à $\frac{i_2+2}{3} \leq \frac{i_2+1}{3}$ qui ne peut pas être transformée facilement en contrainte linéaire sur i_2 ou en contrainte contenant des divisions entières. Une contrainte avec modulo peut être utilisée : $(I) \iff Mod(i_2, 3) = 0 \; or \; 2$

Conclusion

L'un de mes axes de recherche est la synthèse de code pour machines parallèles. L'algorithme de génération automatique de nids de boucles à partir d'une représentation polyédrique d'un ensemble de points entiers, présenté dans cette section, répondait à un besoin réel. Il a été repris pour la génération de code et l'optimisation des communications, à partir des méthodes décrites dans ma thèse [46], dans le compilateur de Stanford `SUIF` [6] et dans le compilateur `PARADIGM` [160]. D'autres techniques ont ensuite été développées pour générer du code à partir de *plusieurs* polyèdres : J.F. Collard & *al.* [67], W. Kelly & *al.* [112] et F. Quillere & *al.* [150]. C. Bastoul a étendu les techniques de Quilleré & *al.* dans `CLooG` [52]. J.F. Collard & *al.* ont proposé une technique différente de Fourier-Motzkin basée sur la méthode duale du simplexe [69]. P. Boulet & *al.* utilisent

2. $\lfloor S \rfloor$ est l'ensemble de points entiers inclus dans S

un automate fini pour générer le parcours sans nids de boucles [57]. Des méthodes mixtes sont parfois utilisées dans la synthèse haut niveau (HLS) [132].

S'assurer que le code généré est correct, c'est déterminer les hypothèses sous lesquelles les algorithmes utilisés respectent les conditions d'exactitude souhaitées. L'algorithme de Fourier-Motzkin, classiquement employé pour projeter les dimensions d'un polyèdre, ne peut pas être utilisé tel quel dans le cadre d'ensemble de points *entiers*, il n'est pas exact. J'ai défini les conditions permettant d'assurer l'exactitude des calculs lors des projections [46].

La correction des algorithmes présentés dans cette section est facile à prouver parce qu'ils sont basés sur des concepts d'algèbre linéaire bien connus.

4.2 Code distribué pour une mémoire partagée émulée

Par rapport aux architectures à mémoire partagée, écrire des programmes scientifiques parallèles pour des multiprocesseurs à mémoire répartie ajoute une difficulté. Le code doit être divisé en tâches placées sur les mémoires locales. Des instructions *receive* doivent être ajoutées dans chaque tâche pour récupérer les données résidant dans d'autres processeurs et, de même, des instructions *send* doivent être introduites dans celles des tâches qui peuvent accéder localement ces mêmes éléments. Toutes les tâches doivent être soigneusement synchronisées pour éviter les erreurs. Écrire et mettre au point des tâches parallèles est beaucoup plus difficile que de déclarer une boucle parallèle.

Dans le cadre du projet PUMA (ESPRIT 2701), nous avons développé un prototype de compilateur, prenant en entrée du code Fortran séquentiel et générant en sortie des tâches parallèles distribuées. Les articles [33, 9] présentent l'approche étudiée pour un réseau de transputers T9000 (Figure 4.8). Nous utilisons le programme d'une transposition de matrice de la figure 4.9 pour illustrer cette approche.

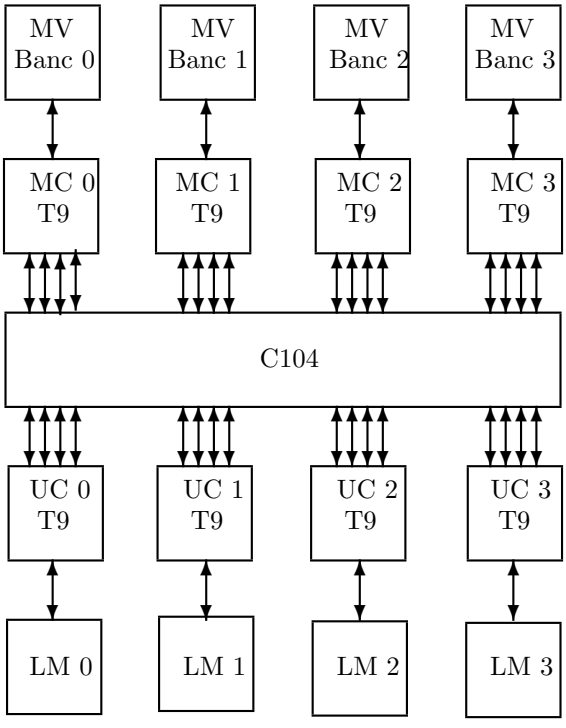


FIGURE 4.8 – Implémentation à base de T9

```
DO I = 1, SIZE-1
   DO J = I+1, SIZE
      T = M(I,J)
      M(I,J) = M(J,I)
      M(J,I) = T
   ENDDO
ENDDO
```

FIGURE 4.9 – Transposition de matrice

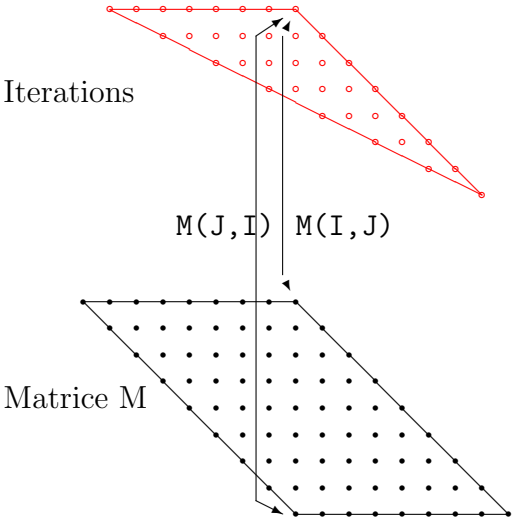


FIGURE 4.10 – Accès mémoire pour une transposition

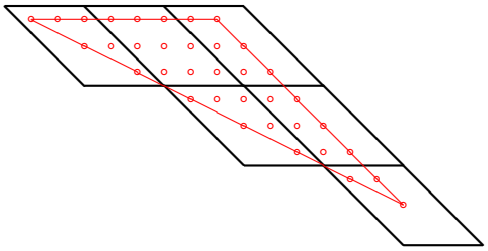


FIGURE 4.11 – Distribution des calculs

L'idée consistait à émuler les bancs d'une mémoire virtuelle partagée (MV) sur une partie des processeurs d'une architecture distribuée. Les autres processeurs constituent les unités de calcul (UC). Les calculs ne dépendent que du contrôle du programme. La distribution des données est implicite ; elles sont uniformément et cycliquement distribuées sur les bancs mémoire.

Des transformations de boucles telles que le tiling peuvent être appliquées pour définir les blocs d'instructions pouvant être exécutées en parallèle et ajuster leur taille de manière à ce que les temps de communication ne soient pas supérieurs aux temps de calcul. Le domaine rouge de la figure 4.10 représente le domaine des itérations à exécuter pour l'exemple de la transposition. La figure 4.11 représente le découpage des itérations en tuiles proposé.

La granularité du parallélisme peut être aussi ajustée car elle n'est pas directement liée à la distribution des données mais uniquement au graphe de contrôle du programme. Chaque bloc est ainsi vu comme une tâche indépendante. Un bon équilibre de charge est plus facile à obtenir que dans le cas de la règle *owner rule*[8] car n'importe quelle tâche peut être exécutée sur n'importe quel processeur.

Lorsque le partitionnement des itérations en tâches est déterminé, les transferts de données entre la mémoire partagée émulée et les mémoires locales des processeurs de calcul peuvent être calculés. La figure 4.12 détaille, en vert, les ensembles accédés au cours du calcul de la tuile rouge.

Chaque tâche est constituée de trois parties : la première correspond aux transferts des données utiles à l'exécution de la tâche de la mémoire globale vers la mémoire locale du processeur, la seconde correspond aux calculs et la dernière aux transferts des données, modifiées au cours de l'exécution, dans la mémoire partagée.

Chaque processeur de calcul exécute ses instructions en ne référençant que les données transférées dans sa mémoire locale. L'ensemble des éléments de tableau lus lors de l'exécution doit donc être copié en mémoire locale. Cet ensemble peut être approximé et surestimé sans que cela n'introduise de problème de cohérence mémoire puisqu'il s'agit d'éléments lus.

8. Cette règle stipule qu'un calcul est effectué sur la machine où est stocké le résultat.

Une fois les calculs terminés, les données modifiées doivent être recopiées dans la mémoire partagée. Ces ensembles d'éléments ne sont pas nécessairement convexes, et ne correspondent pas systématiquement à des éléments contigus sur les bancs mémoire. Pour ne pas introduire de conflits et respecter les dépendances, seuls les éléments modifiés doivent être transférés.

L'ensemble des éléments $\vec{a}$ du tableau A est défini par :

$$\{\vec{a} \in A \mid \exists \vec{\imath} \in I \ t.q. \ B\vec{\imath} \leq \vec{b} \\ \wedge \ (\vec{a} = R_1(\vec{\imath}) + \vec{r}_1 \vee \vec{a} = R_2(\vec{\imath}) + \vec{r}_2) \vee .. \vee \vec{a} = R_k(\vec{\imath}) + \vec{r}_k)\}$$

qui se réécrit :

$$\{\vec{a} \in A \mid \exists \vec{\imath}\,' \in I' \ t.q. \ C(\vec{\imath}\,') \ \wedge \ \vec{a} = R\vec{\imath}\,' + \vec{r}\}$$

où R_1, R_2..., R_k représentent des fonctions d'accès aux éléments de tableau dans le noyau original, les itérations $\vec{\imath}$ du nid de boucles de calculs sont définies par les bornes de boucles $B\vec{\imath} \leq \vec{b}$. Le calcul du meilleur $\mathcal{Z}$-module R est discuté dans [46]. Les itérations $\vec{\imath}\,'$ correspondent aux itérations exprimées dans la nouvelle base de parcours et $C(\vec{\imath}\,')$ aux contraintes après changement de base.

Cette nouvelle base de parcours dépend de l'émetteur ou du récepteur. Pour les processeurs émulant les bancs mémoire, les transferts s'effectuent selon l'ordre suivant : pour chaque banc mémoire, pour chaque processeur, pour chaque ligne du banc contenant un élément et enfin à partir de l'offset du premier élément de tableau de la ligne pour un nombre d'éléments donné.

En réception, on parcourt les éléments en fonction du banc dont ils proviennent, dans l'ordre des dimensions du tableau en tenant compte de son *layout*. En Fortran, les éléments de tableau sont stockés par colonne. Les parcours s'effectuent alors des dimensions les plus grandes vers la dimension la plus petite pour laquelle les éléments sont contigus. Il existe une relation de linéarisation entre le *layout* des éléments du tableau et leur placement sur les bancs mémoire que l'on peut exprimer par l'équation [9] :

$$\sum_{i=1}^{n} a_i \cdot \prod_{k=1}^{n-1} dim(a_k) = l \cdot NB \cdot LS + b \cdot LS + o$$

où $dim(a_k)$ est le nombre d'éléments sur la dimension k, NB le nombre de bancs, LS la taille d'une ligne de banc, et o l'offset dans la ligne.

Les principales étapes de la génération automatique de ce code sont les suivantes : (1) analyse du programme et déclarations de variables, (2) génération des blocs de tâches parallèles, (3) génération des codes de transfert des données.

Déclarations des variables Il est important de pouvoir détecter les variables locales à un corps de boucles parce qu'elles peuvent être allouées directement dans les mémoires locales des processeurs de calcul. Elles n'ont pas besoin d'être transférées entre la mémoire

9. En C, nous aurions la relation suivante où les indices k sont différents :

$$\sum_{i=1}^{n} a_i \cdot \prod_{k=i+1}^{n} dim(a_k) = l \cdot NB \cdot LS + b \cdot LS + o$$

partagée et les mémoires locales durant les exécutions. Les variables scalaires *privées* sont déclarées dans les mémoires locales mais pas dans la mémoire partagée émulée.

Il est aussi indispensable de prendre en compte les *input dependences* qui existent entre des lectures de la même variable pour savoir combien de copies locales il faut associer à chaque variable initiale. Dans le cas de la transposition de matrice, les deux références `M(I,J)` et `M(J,I)` sont totalement indépendantes. Il faut donc prévoir la déclaration de *deux* zones de copies locales pour `M`, qui sont appelées `L_M_0` et `L_M_1`(`L` comme local). Il faut aussi déclarer des zones sur chacun des bancs mémoire pour contenir `M`. Ces zones sont appelées `MV_M` (`MV` comme Mémoire Virtuelle).

Par contre, dans le cas d'un calcul de convolution, on trouverait que toutes les références à un voisinage sont en dépendance et donc qu'il ne faut allouer qu'une seule copie locale.

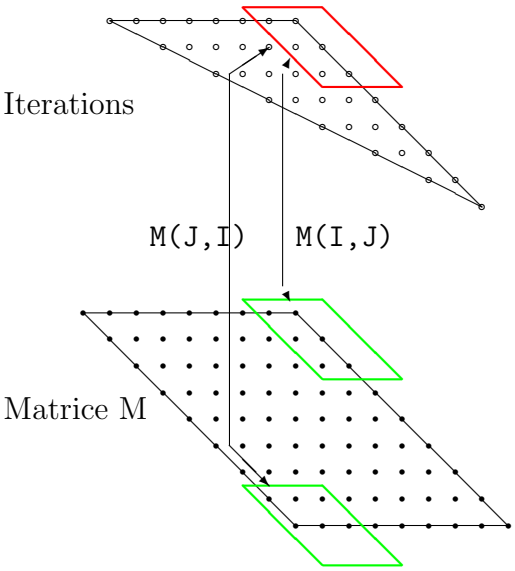


FIGURE 4.12 – Éléments lus et écrits pour une tuile

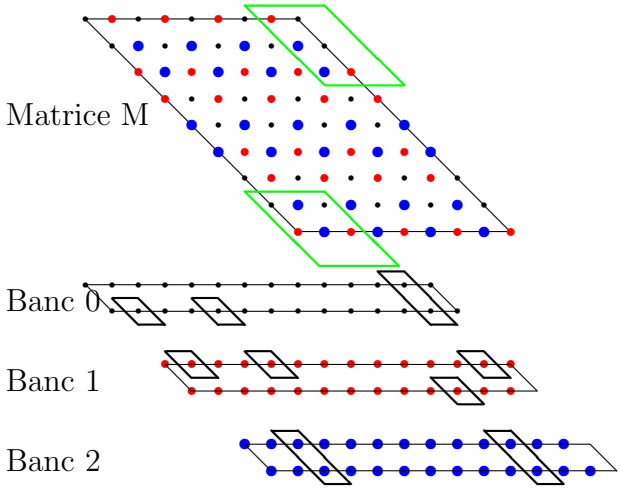


FIGURE 4.13 – Pattern des accès pour une tuile

Génération des codes de transferts La figure 4.12 représente les deux ensembles d'éléments référencés en lecture et en écriture au cours du calcul d'une tuile. La figure 4.13 représente les adresses sur les bancs mémoire de ces deux ensembles de régions. Pour obtenir un dessin lisible, le nombre de bancs est 3, le nombre d'éléments par ligne de bancs est 2 et la taille de la matrice est 9.

Le code généré est constitué de deux sous-programme : l'un, appelé `COMPUTE`, contient le code destiné aux processeurs de calcul, l'autre, appelé `BANK`, celui destinée aux processeurs gérant la mémoire partagée. Des extraits de deux sous-programmes sont présentés ci-dessous. Le sous-programme `COMPUTE` contient les phases de calcul à effectuer, précédées des codes de transferts des données utilisées au cours de ces calculs et suivis par les transferts des données qui sont modifiées au cours de l'exécution. Le sous-programme `BANK` contient les codes de transfert duaux des précédents.
Leurs codes sont constitués de deux boucles externes leur permettant d'exécuter toutes les partitions en fonction du numéro du processeur sur lequel ils s'exécutent. Chaque nid de boucles correspond alors un transfert de données ou à l'exécution du code initial.
Les transferts sont déterminés pour accéder des mots continus. Ceci permet d'implémenter

très efficacement les deux routines de bas-niveau que sont `MY_SEND` et `MY_RECEIVE` puisqu'elles n'effectuent que des transferts DMAs.

Extrait du code reparti produit pour la transposition de matrice

```
                SUBROUTINE COMPUTE(PROC_ID)
                INTEGER PROC_ID,BANK_ID,L,I_0,L_I,J_0,L_J,L_I_1,L_I_2
                REAL*8 L_M_0_0(0 :24,0 :24),L_M_1_0(0 :24,0 :24)

C               DISTRIBUTED CODE FOR TRANSP

C               TO SCAN THE TILE SET
                DO 99975 I_0 = PROC_ID, 3, 4
                     DO 99976 J_0 = I_0, 3
                          DOALL 99994 BANK_ID = 0, 3
                               DO 99995 L_J = MAX(0, 25*I_0-25*J_0), MIN(24, -25*J_0+
     &                              98)
                                    DO 99996 L = (-1*BANK_ID+I_0+100*J_0+4)/4, MIN((396
     &                                   -BANK_ID+I_0)/4, (100-BANK_ID+I_0+100*J_0)/4)



                                         L_I_1 = MAX(0, -2500*J_0-100*L_J-25*I_0-100+25*
     &                                   BANK_ID+100*L)
                                         L_I_2 = MIN(24, 25*J_0+L_J-25*I_0, -2500*J_0-100
     &                                   *L_J-25*I_0-76+25*BANK_ID+100*L)
                                         CALL MY_RECEIVE(BANK_ID, L_M_0_0(L_I_1,L_J),
     &                                   L_I_2-L_I_1+1)


99996                                 CONTINUE
99995                            CONTINUE
99994                       CONTINUE
                                 ...



C               TO SCAN EACH ITERATION OF THE CURRENT TILE
                          DO 99987 L_I = 0, 24
                               DO 99988 L_J = MAX(0, -25*J_0+L_I+25*I_0), MIN(24, -25
     &                         *J_0+98)
                                    T = L_M_0_0(L_I,L_J)
                                    L_M_0_0(L_I,L_J) = L_M_1_0(L_J,L_I)
                                    L_M_1_0(L_J,L_I) = T
200                                 CONTINUE
99988                            CONTINUE
99987                       CONTINUE
                                 ...
99976                    CONTINUE
99975               CONTINUE
                    RETURN
                    END
```

Émulation de la mémoire globale pour la transposition de matrice

```
                  SUBROUTINE BANK(BANK_ID)
                  INTEGER PROC_ID,BANK_ID,L,O,I_0,L_I,J_0,L_J,O_1,O_2
                  REAL*8 MV_M(0 :24,0 :99)

C                 BANK DISTRIBUTED CODE FOR TRANSP

C                 TO SCAN THE TILE SET FOR BANK
                    DO 99973 I_0 = 0, 3
                            PROC_ID = MOD(I_0, 4)
                            DO 99974 J_0 = I_0, 3
                                     DO 99997 L_J = MAX(0, -25*J_0+25*I_0), MIN(24, 98-25*J_0)
                                              DO 99998 L = (4+I_0-BANK_ID+100*J_0)/4, MIN((396+I_0-
     &                                          BANK_ID)/4, (100+I_0-BANK_ID+100*J_0)/4)
                                                       O_1 = MAX(100*L_J+2500*J_0+25*I_0-100*L-25*BANK_ID+
     &                                                   100, 0)
                                                       O_2 = MIN(100*L_J+2500*J_0+25*I_0-100*L-25*BANK_ID+
     &                                                   124, 101*L_J+2525*J_0-100*L-25*BANK_ID+100, 24)
                                                       CALL MY_SEND(PROC_ID, MV_M(O_1,L), O_2-O_1+1)
99998                                            CONTINUE
99997                                       CONTINUE
                                          ...
99974                             CONTINUE
99973                    CONTINUE
                    RETURN
                    END
```

Les premiers résultats de cette étude ont été satisfaisants puisque le code produit par notre prototype a été exécuté sur un réseau de Transputer et sur un réseau de stations de travail. Ils ont validé l'approche sur un noyau de calculs. Les résultats étaient corrects.

Conclusion

Les avantages de la distribution automatique présentée sont :
— de pouvoir s'abstraire du problème de la distribution des données sur les mémoires distribuées,
— de pouvoir considérer des noyaux de calculs plus complexes que ne le permet la règle *owner compute rule*,
— de pouvoir réduire les coûts de communication en augmentant la localité des données accédées et en optimisant les transferts de données,
— d'avoir la possibilité d'utiliser des processeurs dédiés aux requêtes mémoire et n'effectuant pas de calculs,
— de pouvoir obtenir un bon équilibre de charge car n'importe quelle tâche peut être exécutée sur n'importe quel processeur,
— d'avoir un bon taux de parallélisme dû à l'absence de synchronisation interne aux tâches parallèles.

Le problème de la génération automatique de code distribué est plus complexe que le problème de la parallélisation automatique de programme.

L'approche a permis de valider le fait qu'il est possible de générer automatiquement les codes de calculs et de communications pour une machine à mémoire répartie en émulant une mémoire partagée virtuelle. En émulant cette mémoire virtuelle, répartie sur un sous-ensemble de processeurs distribués, on élimine une partie des synchronisations nécessaires entre les tâches de calcul et le gestionnaire de la mémoire. On se replace ainsi dans le cadre *moins complexe* d'une mémoire partagée, où il est plus facile de prouver que le code généré est correct.

Ce prototype a également permis de valider nos techniques de génération des codes de calculs et des communications à partir d'une formalisation affine des espaces d'itérations et du placement des données sur les bancs mémoire.

4.3 Code distribué pour HPF

D'autres approches, que celle exposée dans la section précédente, ont été proposées pour utiliser un environnement de mémoire partagée sur une architecture à mémoire distribuée. Par exemple, le langage `HPF` a été développé pour exécuter des programmes à parallélisme de données ou *data-parallèle* sur des machines à mémoire répartie. Le programmeur précise la distribution des données sur les processeurs par le biais de directives. Le compilateur exploite ces directives pour allouer les tableaux en mémoire locale, affecter les calculs propres à chaque processeur élémentaire et générer les communications entre les processeurs.

Dans le cadre d'`HPF`, la règle *owner compute rule*[10] est utilisée et les processeurs exécutent les instructions si les mises à jour des données correspondent à des données locales. Les communications sont déduites de la distribution des données.

4.3.1 Formalisation des distributions HPF

Dans l'article [28], j'ai proposé, en collaboration avec Fabien Coelho, François Irigoin et Ronan Keryell, un cadre algébrique affine pour modéliser et compiler `HPF`. La distribution des données est précisée en deux temps. Les éléments de tableau sont *alignés* sur des

10. Cette règle stipule qu'un calcul est effectué sur la machine où est stocké le résultat.

processeurs virtuels appelés des *templates*. Ensuite les *templates* sont *distribués* sur les processeurs par blocs ou de manière cyclique. Les contraintes modélisant la distribution et l'*alignement* des données sur les processeurs sont résumées dans le système suivant :

$$\begin{cases} 0 \leq D^{-1}\vec{a} < 1 \ , \\ 0 \leq T^{-1}\vec{t} < 1 \ , \\ 0 \leq P^{-1}\vec{p} < 1 \ , \\ 0 \leq C^{-1}\vec{l} < 1 \ , \\ \wp\vec{t} = A\vec{a} + \vec{t_0} \ , \\ \Pi\vec{t} = CP\vec{c} + C\vec{p} + \vec{l} \end{cases} \tag{4.2}$$

Les quatre premières contraintes reprennent respectivement les déclarations des tableaux D, des *templates* T, des processeurs P et des tailles de blocs C précisées par la directive de distribution. La cinquième contrainte donne l'alignement des éléments $\vec{a}$ du tableau sur un *template* $\vec{t}$. Il peut s'exprimer dimension par dimension comme combinaison linéaire d'éléments des *templates*. Afin de pouvoir traduire la duplication éventuelle d'éléments du tableau sur des dimensions différentes du *template*, une matrice de projection $\wp$ supplémentaire est nécessaire.

La sixième et dernière contrainte du système traduit la distribution des *templates* par bloc sur les processeurs. Elle relie les coordonnées des processeurs $\vec{p}$, celles du *template* $\vec{t}$ et deux autres variables : $\vec{l}$ et $\vec{c}$. Le vecteur $\vec{l}$ représente l'offset dans un bloc sur un processeur et le vecteur $\vec{c}$ le nombre de cycles nécessaires pour allouer les blocs sur les processeurs. La matrice Π est utile lorsque plusieurs dimensions du *template* sont rassemblées sur un même processeur. P est une matrice diagonale et décrit la géométrie de la machine. Chaque élément de sa diagonale est égale au nombre de processeurs pour chaque dimension.

Afin de générer le code correspondant aux éléments référencés par le noyau de calcul, il faut ajouter l'ensemble des contraintes reliant le domaine d'itération B d'un nid de boucles de calcul et une référence $(R, \vec{r})$ au tableau :

$$B\vec{\imath} \leq \vec{b}, \vec{a} = R\vec{\imath} + \vec{r}$$

L'ensemble $\mathrm{Own}_X(p)$ des éléments d'un tableau X placés sur un processeur $\vec{p}$ est défini par :

$$\begin{aligned} \mathrm{Own}_X(\vec{p}) = \{ \ \vec{a} \mid \quad & \exists\vec{t} \ , \ \exists\vec{c} \ , \exists\vec{l} \ t.q. \\ & 0 \leq D_X^{-1}\vec{a} < 1 \ , 0 \leq T_X^{-1}\vec{t} < 1 \ , 0 \leq P^{-1}\vec{p} < 1 \ , 0 \leq C_X^{-1}\vec{l} < 1 \ , \\ & \wp\vec{t} = A\vec{a} + \vec{t_0} \ , \Pi\vec{t} = C_X P\vec{c} + C_X\vec{p} + \vec{l}\} \end{aligned}$$

Notons que cet ensemble $\mathrm{Own}_X(p)$ d'éléments $\vec{a}$ n'est pas un dense, p ne possède que les éléments $\vec{l}$ de ses blocs et de certains cycles $\vec{c}$. Il peut être vu comme dense dans l'espace $(\vec{a}, \vec{c}, \vec{l})$, que nous exploiterons pour optimiser l'énumération des éléments référencés.

En utilisant la règle *owner compute*, pour un noyau d'instructions calculant les valeurs d'un tableau X, nous dérivons l'ensemble des itérations devant être exécutées sur le processeur $\vec{p}$:

$$\mathrm{Compute}(\vec{p}) = \left\{ \vec{\imath} \mid R_X\vec{\imath} + \vec{r}_X \in \mathrm{Own}_X(\vec{p}) \ \wedge \ B\vec{\imath} \leq \vec{b} \right\}$$

L'ensemble des éléments d'un tableau Y référencés par une référence $(R_Y, \vec{r}_Y)$ de ce même noyau est défini par :

$$\text{View}_Y(\vec{p}) = \{\vec{a} \mid \exists \vec{\imath} \in \text{Compute}(\vec{p}) \ t.q. \ \vec{a} = R_Y\vec{\imath} + \vec{r}_Y\}$$

Lorsque les tableaux en lecture sont référencés plusieurs fois et que ces références sont indépendantes, il est préférable d'allouer des copies locales différentes pour chacune de ces références. Si elles sont dépendantes, une seule copie locale est allouée et la région convexe des éléments référencés du tableau Y sera considérée comme nouvelle définition de $\text{View}_Y(\vec{p})$.

Les ensembles d'éléments qui appartiennent à un processeur $\vec{p}$ et qui sont utilisés par un autre processeur $\vec{p}\,'$ doivent être transférés. Ils sont définis par les ensembles suivants :

$$\text{Send}_Y(\vec{p}, \vec{p}\,') = \text{Own}_Y(\vec{p}) \bigcap \text{View}_Y(\vec{p}\,')$$
$$\text{Receive}_Y(\vec{p}, \vec{p}\,') = \text{View}_Y(\vec{p}) \bigcap \text{Own}_Y(\vec{p}\,')$$

Les ensembles $\text{Own}(\vec{p}), \text{View}(\vec{p}), \text{Send}(\vec{p}, \vec{p}\,')$ et $\text{Receive}(\vec{p}, \vec{p}\,')$ représentent des ensembles non nécessairement convexes. Il faut rester vigilant lors de la génération de code afin d'obtenir un code efficace. Il faut également être attentif aux points suivants :

1. un processeur ne doit pas faire de communication avec lui-même,
2. les tableaux répliqués sur plusieurs processeurs peuvent engendrer des communications redondantes qu'il faut éviter,
3. l'ordre des transferts des données doit être établi afin de tenir compte : de l'allocation des tableaux, de leurs *layouts*, de l'introduction de tests qui pourraient être superflus,...

Le point (1) ne peut pas s'exprimer par des contraintes affines et un test doit être ajouté dans le code des communications.

Le point (2) peut se traiter en ajoutant des contraintes affines supplémentaires aux ensembles $\text{Send}_Y(\vec{p}, \vec{p}\,')$ et $\text{Receive}_Y(\vec{p}, \vec{p}\,')$. En effet, les seules réplications possibles en `HPF` sont les réplications de toute une dimension du tableau sur plusieurs processeurs. Lorsqu'une dimension de tableau est répliquée sur les processeurs source, il serait possible de choisir arbitrairement un processeur parmi ces derniers pour diffuser l'ensemble de ses données aux processeurs cibles. Mais nous avons opté pour un meilleur équilibrage de charge des communications : en attribuant de manière cyclique aux différents processeurs source la responsabilité du transfert des éléments répliqués aux processeurs cibles. Cette condition s'exprime par une relation linéaire de distribution cyclique des processeurs cibles sur les processeurs source possédant des dimensions répliquées. L'article [66] détaille ces solutions permettant d'optimiser les communications liées aux redistributions d'`HPF`.

Concernant le point (3), la méthode utilisée pour énumérer les itérations locales au processeur et les transferts de données entre processeurs est basée sur la résolution des équations de distributions `HPF`. Elle recherche des bases appropriées pour les treillis [11] décrivant les éléments à énumérer. Cette recherche utilise deux formes de Hermite.

11. L'ensemble des combinaisons linéaires entières des vecteurs de base

Les formes normales de Hermite (ou de Smith) permettent de trouver les solutions entières ou rationnelles d'un système d'équations $A \cdot \vec{x} = \vec{c}$.

Soit une matrice $A(n,m)$ entière de rang r ($r \leq n, r \leq m$), il existe des matrices $P(n,n)$ et $Q(m,m)$ unimodulaires telles que $H = P \cdot A \cdot Q$ où $H(n,m)$ est la forme normale de Hermite. On a :

$$P \cdot A \cdot Q = \begin{pmatrix} H_L & 0 \\ H_S & 0 \end{pmatrix} = H$$

avec $H_L(r,r)$ une matrice triangulaire inférieure et $H_S(n-r,r)$ une matrice. P est une matrice de permutation et Q est unimodulaire quelconque .

A partir de l'équation $A\vec{x} = \vec{c}$, on peut écrire $A \cdot Q \cdot Q^{-1}\vec{x} = \vec{c}$. Après le changement de variable $\vec{y} = Q^{-1}\vec{x}$, nous obtenons assez facilement une solution pour les r premières composantes de $\vec{y}$ qui vérifient l'équation $H\vec{y} = P \cdot \vec{c}$, car H est échelonnée. Les $(m-r)$ dernières composantes de $\vec{y}$ sont libres puisque les $(m-r)$ dernières composantes de H sont nulles. A partir de cette solution particulière $\vec{y_0} = \begin{pmatrix} y_{0,0} \\ y_{0,1} \\ .. \\ y_{0,r} \\ k_1 \\ .. \\ k_{m-r} \end{pmatrix}$ où $y_{o,i\ (1 \leq i \leq r)}$ est un entier et $k_{j\ (1 \leq j \leq m-r)}$ est une variable libre, nous obtenons la solution générale de l'équation $\vec{x} = Q\vec{y_0}$.

Notons que H donne le treillis et les colonnes de Q donne les vecteurs de base des solutions de l'équation générale. Q est une matrice unimodulaire quelconque et la formulation de $\vec{x} = Q\vec{y_0}$ n'est pas nécessairement échelonnée pour la base des x_i.

Dans le cadre de la compilation des programmes `HPF`, l'ordre d'énumération des éléments à transférer et leur allocation en mémoire ne sont pas fixés au départ. Le calcul de nouvelles bases de parcours permet de générer des codes plus efficaces. Les formes de Hermite ont été utilisées 1) pour préserver l'ordre des variables $(\vec{l}, \vec{c})$ relatives à l'allocation des tableaux et 2) pour favoriser des parcours contigus, efficaces en mémoire locale.

L'article [28] détaille le schéma de compilation complet proposé pour les programmes `HPF`, ainsi que les techniques permettant d'optimiser l'allocation des tableaux locaux et la génération des codes de communications.

Conclusion

Même si `HPF` n'a pas eu le succès attendu, il a fait partie de toute une lignée de langages comme Fortran-D [86], Fortran-S [55], `dHPF` [123], `Co-Array Fortran` [141], `dSTEP` [90] destinés à faciliter la programmation de machines parallèles à mémoire distribuée. Ces langages proposent des extensions au langage `Fortran` avec des directives précisant le parallélisme et le placement des données. Ils ont été suivis par la génération des langages PGAS (*Partitioned Global Address Space*) comme `Chapel` [94], `X10` [153], `UPC` [168, 65], `Habanero-java` [61] conçus pour la programmation parallèle avec des accès mémoire distants. D'autres langages comme `OpenHMPP` [81], `OpenCL` [159] ou `OpenACC` [142] proposent des extensions pour `Fortran` et `C`, et ciblent les machines parallèles hétérogènes. `OpenMP` [143, 144] vise les machines parallèles à mémoire partagée.

Nous avons proposé un cadre affine unifié permettant de spécifier les directives du langage HPF et fournissant les informations nécessaires à la compilation du langage. De nouveaux algorithmes ont été développés pour énumérer les itérations locales aux processeurs (respectant la règle *owner compute*), pour allouer les tableaux locaux distribués, générer les communications et allouer les variables temporaires utilisées au cours des calculs.

Un effort particulier a été fait pour tenir compte des caches et générer des accès efficaces contigus aux éléments de tableau. Des outils mathématiques, tels que le calcul des formes de Hermite, ont facilité l'obtention d'une allocation optimale pour les tableaux locaux.

Nous avons tiré avantage des restrictions du langage HPF, notamment concernant la réplication des éléments de tableaux sur les processeurs. L'optimisation des communications dans le cas de réplications moins contraintes est reprise dans la section 4.4.

4.4 Optimisations des communications pour des transferts par accès direct aux mémoires (DMA)

Les applications de traitement d'images et du signal sont de bons candidats pour une parallélisation à grain fin car elles traitent de larges flots de données parallèles. L'outil SPEAR-DE [42] est un environnement graphique de programmation qui aide au placement de ce type d'applications sur des architectures parallèles variées, hétérogènes et/ou hiérarchiques.

Les applications sont modélisées par des graphes de flots de données acycliques où les dépendances entre tâches sont explicites (cf. figure 4.14). Les tâches de calculs sont des nids de boucles parallèles exécutant une fonction élémentaire. Elles itèrent sur des signaux (tableaux) souvent multidimensionnels.

L'outil SPEAR-DE propose un placement de l'application sur l'architecture cible en attribuant aux parties de l'application (segments) les composants architecturaux responsables de leur exécution.

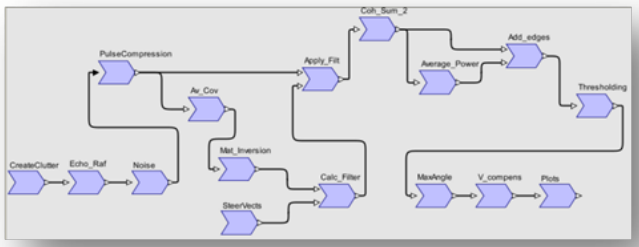


FIGURE 4.14 – Exemple d'application SPEAR-DE

L'article [44] présente le contexte et les techniques que nous avons développées pour générer automatiquement les codes de communication nécessaires aux transferts des données accédées par deux segments de l'application et placés sur des parties différentes de l'architecture.

Ces travaux sont originaux car 1) les communications doivent s'effectuer par des contrôleurs DMAs multidimensionnels et 2) le placement des calculs sur les processeurs peut conduire à une réplication partielle des données sur différentes mémoires locales. L'objectif était de générer des codes de communication corrects, compatibles avec des contrôleurs DMAs multidimensionnels et efficaces, au moins en nombre de communications.

4.4.1 Formalisation des redistributions SPEAR-DE

Les dimensions des tableaux référencées par les applications de traitement du signal ou d'image sont souvent indépendantes. Les boucles qui itèrent sur ces dimensions peuvent être exécutées en parallèle et placées sur des processeurs différents. Toutefois, afin de conserver une granularité suffisante permettant de recouvrir les communications et les calculs, certaines itérations ne sont pas parallélisées sur les processeurs et s'exécutent séquentiellement sur tous les processeurs. D'autres itérations appartiennent au *motif* indivisible d'une unité de calcul. Ces dernières ne sont pas parallélisées et constituent avec les précédentes les itérations *internes* $\vec{\imath}_t$ au nid de boucles.

Les contraintes du placement d'un nid de boucles par `SPEAR-DE` peuvent se formuler par le système suivant :

$$\begin{cases} 0 \leq D_A^{-1}\vec{a} < 1 \ , B_e\vec{\imath}_e \leq \vec{b}_e, B_t\vec{\imath}_t \leq \vec{b}_t, \\ 0 \leq P^{-1}\vec{p} < 1 \ , \ 0 \leq L^{-1}\vec{l} < 1, \quad 0 \leq C^{-1}\vec{c} < 1, \\ \vec{a} = R\vec{\imath} + \vec{r}_0 \ , \\ \vec{\imath} = E\vec{\imath}_e + T\vec{\imath}_t \\ \vec{\imath}_e = LP\vec{c} + L\vec{p} + \vec{l} \end{cases} \tag{4.3}$$

Les premières contraintes reprennent respectivement les déclarations des tableaux D_A, des bornes de boucles des itérations externes $(B_e, \vec{b}_e)$ et internes $(B_t, \vec{b}_t)$, des processeurs P et des tailles L de blocs d'itérations allouées à chaque processeur. Les équations donnent 1) la fonction d'accès des éléments $\vec{a}$ de tableaux, 2) le partitionnement entre les itérations externes $\vec{\imath}_e$ du nid de boucles qui sont distribuées aux processeurs et les itérations internes $\vec{\imath}_t$, et 3) la distribution (cyclique) des itérations externes par blocs sur les processeurs $\vec{p}$.

Cette dernière équation relie les coordonnées des processeurs $\vec{p}$, celles des itérations externes $\vec{\imath}_e$ et deux autres variables : $\vec{l}$ et $\vec{c}$. Le vecteur $\vec{l}$ représente l'offset dans un bloc sur un processeur et le vecteur $\vec{c}$ le nombre de cycles nécessaires pour allouer tous les blocs sur les processeurs. Les matrices E, T, L et P sont données en entrée comme des résultats du placement de l'application sur l'architecture parallèle.

De manière plus générale, les itérations i_e peuvent être distribuées par blocs et/ou cycliquement sur des processeurs qui sont hiérarchiques. La variable *processeur* peut être vue comme une composition linéaire d'autres composantes architecturales. Nous avons donc :

$$i_e = \sum_{i=1}^{n} (\prod_{k=1}^{i-1} dim(C_k)) \cdot l_i$$

où les termes l_i représentent soit des processeurs, soit des itérations dans un bloc alloué à un processeur, soit une composante d'une dimension cyclique.

La différence majeure entre les contraintes de placement de `SPEAR-DE` et les distributions `HPF` (système (4.2)) est que ce sont les itérations qui sont distribuées sur les proces-

seurs et non les données. La distribution des données après placement par SPEAR-DE sur les processeurs est donc plus complexe.

Un élément de tableau $\vec{a}$ sera référencé au cours des calculs par un processeur $\vec{p}$ s'il vérifie la contrainte suivante :

$$\begin{array}{c} \exists\ \vec{\imath} \in I,\ \exists\ \vec{c}\ \in \mathcal{C},\ \exists\ \vec{l} \in \mathcal{L}, B\vec{\imath} \leq \vec{b},\ 0 \leq L^{-1}\vec{l} < 1,\ B_t\vec{\imath} \leq \vec{b}_t,\ \leq C^{-1}\vec{c} < 1 \\ \vec{a} = R_e ELP\vec{c} + EL\vec{p} + E\vec{l} + R_t \cdot T\vec{\imath}_t + \vec{r}_0 \end{array} \tag{4.4}$$

qui est directement déduite des équations du système (4.3).

La *distribution* des éléments de tableaux est donc liée aux calculs et itérations attribués au processeur.

Nous définissons la distribution comme une relation entre une adresse logique $\vec{\mu} = (\vec{p}, \vec{l}, \vec{c}, \vec{i})$ de $\mathcal{P}\mathtt{x}\mathcal{L}\mathtt{x}\mathcal{C}\mathtt{x}\mathcal{I}$ où $\mathcal{P}$ est l'ensemble des processeurs, $\mathcal{L}$ représente les itérations locales à un processeur, $\mathcal{C}$ est une dimension cyclique et $\mathcal{I}$ est l'ensemble des itérations (non distribuées) et l'élément $\vec{a}$ du tableau A placé à cette adresse.

On note l'équation (4.4)

$$\mathcal{D}_A(\vec{\mu}) = \vec{a} \tag{4.5}$$

avec $\vec{\mu} = (\vec{p}, \vec{l}, \vec{c}, \vec{i})$.

Nous définissons également $\mathcal{D}om(\vec{\mu})$ le système de contraintes caractérisant les coordonnées de $\vec{\mu}$ pour lesquelles $\vec{a}$ est défini :

$$\mathcal{D}om(\vec{\mu}) = \left\{ \begin{array}{c} B\vec{\imath} \leq \vec{b} \\ 0 \leq P^{-1}\vec{p} < 1 \\ 0 \leq L^{-1}\vec{l} < 1, \\ 0 \leq C^{-1}\vec{c} < 1, \end{array} \right. \tag{4.6}$$

Nous définissons l'ensemble des éléments d'un tableau A distribués sur un processeur $\vec{p}$ par l'ensemble $\mathcal{D}ist(A, \vec{p})$:

$$\mathcal{D}ist(A, \vec{p}) = \left\{ \vec{a} \in D_A \mid \exists\ \vec{\imath} \in I,\ \exists\ \vec{c}\ \in \mathcal{C},\ \exists\ \vec{l} \in \mathcal{L}\ t.q.\ \mathcal{D}_A((\vec{p}, \vec{l}, \vec{c}, \vec{i})) = \vec{a} \right\} \tag{4.7}$$

Lorsque deux segments d'une application sont placés sur des parties différentes de l'architecture, et que ces deux segments référencent un même tableau, il faut transférer les éléments calculés par le premier segment au second. Soit $\mathcal{D}ist_S(A, \vec{p}_s)$ la distribution des éléments d'un tableau A sur les processeurs $\vec{p}_s$ à la source et $\mathcal{D}ist_D(A, \vec{p}_d)$ leurs distributions sur des processeurs $\vec{p}_d$ à la destination. La redistribution est possible si tous les éléments qui doivent être communiqués à la destination sont présents à la source. Le prédicat suivant doit être vérifié :

$$\forall \vec{a}_d \in \mathcal{D}ist_D(A, \vec{p}_d), \exists\ \vec{a}_s \in \mathcal{D}ist_S(A, \vec{p}_s)\ t.q.\ \vec{a}_s = \vec{a}_d \tag{4.8}$$

Nous définissons l'ensemble des éléments d'un tableau A, devant être redistribués, à partir de sa distribution sur des processeurs à la source $\mathcal{D}ist_S(A, \vec{p}_s)$ et de sa distribution sur des processeurs à la destination $\mathcal{D}ist_D(A, \vec{p}_d)$ par l'ensemble :

$$\mathcal{R}dist(A, \vec{p}_s, \vec{p}_d) = \{\vec{a}_d \in \mathcal{D}ist_D(A, \vec{p}_d) \mid \exists\ \vec{a}_s \in \mathcal{D}ist_S(A, \vec{p}_s) \wedge \vec{a}_s = \vec{a}_d\} \tag{4.9}$$

Le problème de la redistribution revient à trouver l'ensemble des éléments du tableau `A` qui appartiennent aux deux distributions et sont des solutions de l'intersection des deux systèmes de contraintes, source et destination :

$$\begin{cases} \mathcal{D}_A(\mu_s^1) = \mathcal{D}(\mu_d^1) \\ .... \\ \mathcal{D}_A(\mu_s^h) = \mathcal{D}(\mu_d^h) \end{cases} \quad \begin{array}{ll} 0 \leq D^{-1}\vec{a}_s < 1 & 0 \leq D^{-1}\vec{a}_d < 1 \\ 0 \leq P^{-1}\vec{s} < 1 & 0 \leq P^{-1}\vec{d} < 1 \\ 0 \leq L_s^{-1}\vec{l}_s < 1, & 0 \leq L_d^{-1}\vec{l}_d < 1, \\ 0 \leq C_s^{-1}\vec{c}_s < 1, & 0 \leq C_d^{-1}\vec{c}_d < 1, \\ B_{es}\vec{i}_{es} \leq \vec{b}_{es}, & B_{ed}\vec{i}_{ed} \leq \vec{b}_{ed}, \\ B_{ts}\vec{i}_{ts} \leq \vec{b}_{ts} & B_{td}\vec{i}_{td} \leq \vec{b}_{td} \end{array} \tag{4.10}$$

où les termes $(\mu_s^k;_{1\leq k\leq h})$ et $(\mu_d^k;_{1\leq k\leq h})$ sont les variables, respectivement de la source et de la destination, sur lesquelles les éléments de la dimension k du tableau sont distribués.

Afin de clarifier la présentation, nous utilisons dans les sections suivantes une forme plus générique pour le système d'équations. L'équation $\mathcal{D}_A(\vec{\mu_s^k}) - \mathcal{D}_A(\vec{\mu_d^k}) = 0$ est remplacée par $\sum_{i=1}^{m^k} \delta_i^k \cdot x_i^k - \delta_0^k = 0$ où x_i^k représente l'une des variables de μ_s^k ou μ_d^k.

4.4.2 Les contraintes liées aux DMAs

Les mécanismes d'accès direct à la mémoire (DMA ou *Direct Memory Access*) sont des composants matériels qui permettent d'accéder directement à la mémoire sans faire intervenir les processeurs. Les transferts DMAs ne bloquent pas le processeur et les calculs peuvent se poursuivre en parallèle avec les communications. Les DMAs sont très efficaces pour de larges transferts agrégés par blocs. Les blocs de données doivent être soient contigus, soient espacés d'un *pas* fixe. Pour initier un transfert, le DMA a besoin de l'adresse des données, du décalage en mémoire du premier élément, du nombre d'éléments à transférer et du *pas* entre deux éléments consécutifs dans le bloc. Tout code de communication d'un tableau multidimensionnel ciblant les DMAs doit fournir ces quatre informations pour chacune des dimension du tableau.

L'article [44] présente une méthode pour synthétiser les codes de transferts des données devant être communiquées entre deux segments d'une application placés sur une architecture parallèle et possédant des dispositifs DMAs multidimensionnels. L'objectif était de trouver un algorithme permettant de transférer automatiquement une seule fois (sans redondance) les points respectant les contraintes de la redistribution. Ce problème revient à nouveau à énumérer l'ensemble des points d'un polyèdre.

Il semblait donc possible d'utiliser des algorithmes classiques de parcours des points entiers d'un polyèdre. Mais ces algorithmes conduisent à des codes de transferts qui ne répondent pas totalement aux besoins des DMAs multidimensionnels. En effet, de nombreuses opérations telles que les divisions sont nécessaires pour exprimer les contraintes sur le treillis auquel appartiennent les variables du système. Le fait, par exemple, de ne prendre qu'un point sur 5 peut se traduire par un test comportant des divisions entières (par exemple $\frac{x+4}{5} <= \frac{x}{5}$) ou des opérateurs modulo ($mod(x, 5) = 0$). Or pour des transferts de type DMA il n'est pas possible d'intégrer des tests dans les boucles de parcours puisqu'il faut préciser le premier élément à transférer puis le saut entre deux éléments et enfin le nombre d'éléments à transférer. Le transfert d'un élément sur 5 doit se traduire impérativement par un saut de 5 dans la génération de code des nids de boucles. Pour cette raison il était nécessaire de déterminer précisément le treillis auquel les variables du système appartiennent, tout en respectant l'ordre donné pour ces variables.

Il y avait besoin d'un algorithme prenant en entrée un polyèdre défini par

1. un système d'inéquations,
2. un ensemble ordonné de variables (une base),
3. et un système d'équations décrivant les relations entre les éléments de la base

et renvoyant en sortie le même polyèdre défini par un nouveau système d'égalités et d'inégalités tel que :

1. chaque variable est définie par une équation comme une combinaison linéaire des variables de rang plus élevé dans la base et d'une seule variable libre,
2. chaque variable libre est bornée par une expression `MIN` ou `MAX` qui ne contient que des variables de rang plus élevé dans la base.

Le polyèdre en entrée correspond au système formalisant la redistribution. Les équations du polyèdre de sortie donnent le treillis auquel appartiennent les variables de la base ; les coefficients des variables libres donnent les *pas* pour les DMAs.

Il n'y a pas de contraintes sur les variables qui ne font pas partie de la base car elles correspondent en général à des constantes symboliques. Un tel système peut être utilisé pour dériver directement les nouvelles bornes de boucles pour des DMAs.

4.4.3 Utilisation d'outils mathématiques

Afin de présenter les nouvelles techniques que nous avons introduites, je rappelle dans cette section quelques théorèmes et résultats concernant la résolution d'équations linéaires sur nombres entiers relatifs ou naturels.

Nous avons besoin de trouver le treillis auquel appartient l'ensemble des variables définies par le système d'équations de (4.10) sans effectuer de changement de base qui modifierait l'ordre de parcours de ces éléments. La forme normale de Hermite (p.65) est unique, mais nécessite des changements de base unimodulaires quelconques. Nous avons donc utilisé un autre algorithme.

Recherche de solutions de l'équation $\sum_{i=1}^{m} \delta_i \cdot x_i = \delta_0$

En partant de la solution générale d'une équation diophantienne à deux dimensions, nous avons généralisé le résultat d'une équation diophantienne à un nombre quelconque de dimensions. Cela nous permet d'obtenir les informations dont nous avions besoin pour les DMAs.

Si δ_0 est un multiple du plus grand commun diviseur des coefficients δ_i, l'équation a une infinité de solutions. Nous utilisons l'algorithme étendu d'Euclide pour calculer la solution pour une dimension arbitraire m. La solution générale [56] de l'équation linéaire diophantienne $\sum_{i=1}^{m} \delta_i \cdot x_i = \delta_0$ peut s'exprimer sous la forme matricielle suivante.

$$\vec{x} = \vec{x_0} + \begin{pmatrix} d_1 & 0 & 0 & 0 & 0 \\ -\gamma_1 x_{2,1} & 0 & 0 & 0 & 0 \\ . & . & 0 & 0 & 0 \\ -\gamma_1 x_{i,1} & . & d_i & 0 & 0 \\ -\gamma_1 x_{j,1} & . & -\gamma_i x_{j,i} & 0 & 0 \\ . & . & . & . & 0 \\ -\gamma_1 x_{m-1,1} & . & -\gamma_i x_{m-1,i} & . & \gamma_m \\ -\gamma_1 x_{m,1} & . & -\gamma_i x_{m,i} & . & -\gamma_{m-1} \end{pmatrix} \cdot \vec{k} \tag{4.11}$$

où
— $d_j = pgcd_{j+1 \leq i \leq m} \left(\frac{\delta_i}{\Delta_{j-1}} \right) = \frac{pgcd_{j+1 \leq i \leq m}(\delta_i)}{\Delta_{j-1}}$
— avec $\Delta_0 = 1$.
— $\Delta_n = \prod_{i=1}^{n} d_i$
— $\gamma_i = \frac{\delta_i}{\Delta_{i-1}}$

Les $\vec{k}$ sont des variables libres. Si la dimension de $\vec{x}$ est m, seulement $m-1$ variables libres seront nécessaires pour exprimer la solution générale.

Les $x_{*,k}$ sont les composantes de la solution particulière de l'équation $\sum_{i=k+1}^{m} \delta_i \cdot x_{i,k} = 1$ et $\vec{x_0}$ est une solution particulière de l'équation $\sum_{i=1}^{m} \delta_i \cdot x_i = \delta_0$. Ces solutions particulières peuvent être calculées à partir des techniques dérivées de l'algorithme de Bezout.

Cette expression de la solution générale de l'équation présente des propriétés très intéressantes car la matrice est triangulaire inférieure. Pour chaque dimension i, une seule variable libre k_i est ajoutée. L'expression de la solution générale pour x_i est donnée par les équations :

$$x_i = x_{0_i} + \sum_{j=1}^{i-1} -\gamma_j \cdot x_{i,j} \cdot k_j + d_i \cdot k_i \tag{4.12}$$

x_{0_i}, γ_j, $x_{i,j}$ et d_i sont des constantes entières.

Cette expression de la solution peut être vue comme l'addition d'un *offset*=$x_{0_i} + \sum_{j=1}^{i-1} \gamma_j \cdot x_{i,j} \cdot k_j$ et de $d_i \cdot k_i$.

Dans notre contexte d_i est le *pas* entre deux éléments x_i consécutifs sur la i-ième dimension.

Les solutions pour les deux dernières dimensions s'expriment sous la forme suivante :

$$x_{m-1} = x_{0_{m-1}} + \sum_{j=1}^{m-2} -\gamma_j \cdot x_{m-1,j} \cdot k_j + \gamma_m \cdot k_{m-1}$$

$$x_m = x_{0_m} + \sum_{j=1}^{m-1} -\gamma_j \cdot x_{m,j} \cdot k_j - \gamma_{m-1} \cdot k_{m-1}$$

Elles partagent une même variable libre k_{m-1}. Ce qui implique un lien sur l'énumération de ces deux dimensions. C'est l'une des raisons pour laquelle il est approprié de choisir

respectivement les dimensions mémoire à la source et à la destination pour ces deux dimensions. Dans ce cas, les contrôleurs mémoires à la source et à la destination effectueront des accès réguliers et synchronisés.

Nous définissons l'opérateur `iSolve` qui prend en entrée une équation eq et une liste variable $\mathcal{L}$ définissant un ordre d'énumération pour les m variables x_i de l'équation. `iSolve` donne en sortie : m nouvelles équations définissant les m variables x_i comme des combinaisons linéaires des variables de rang plus élevé. Chaque variable x_i s'écrit :

$$x_i = x_{0_i} + \sum_{j=1}^{i-1} -\gamma_j \cdot x_{i,j} \cdot k_j + d_i \cdot k_i \tag{4.13}$$

Recherche de solutions parmi les entiers naturels

Les solutions auxquelles nous nous intéressons dans le cadre de la génération de code et de communications sont souvent des entiers naturels (par exemple, les itérateurs des composantes architecturales sont positifs). Les résultats qui suivent permettent de se restreindre à ce domaine lorsque c'est nécessaire et possible. Ils renseignent aussi sur l'existence d'une solution et donnent leur nombre.

Théorème de Cesàro Le théorème de **Cesàro** [62] nous renseigne sur le nombre de solutions de l'équation $a \cdot x + b \cdot y = c$:

> *Il existe exactement* $ab - \frac{1}{2}(a-1)(b-1)$ *entiers naturels* r *compris entre* 0 *et* $ab-1$ *pour lesquels l'équation* $ax + by = r$ *admet une solution.*

Lorsque un entier r compris entre 1 et $ab-1$ est donné, pour déterminer si l'équation $ax + by = r$ admet une solution il faut regarder le système *minimal*. Le point de la droite $ax + by = r$ le plus proche, au sens de la distance euclidienne, de l'origine est de coordonnées rationnelles positives. L'unique solution entière positive (si elle existe) correspond à un des points de la droite à coordonnées entières les plus proches de ce point. Si (x_1, y_1) est une solution dans l'ensemble des entiers relatifs de l'équation $ax + by = r$, l'unique solution positive (si elle existe) est à chercher parmi les couples

$$(x_1 + bk_1, y_1 - ak_1)$$

ou

$$(x_1 + bk_2, y_1 - ak_2)$$

où k_1 et k_2 sont les deux entiers les plus proches de $\frac{ay_1 - bx_1}{a^2+b^2}$.

Nous utilisons ces résultats pour calculer les solutions particulières des équations $\sum_{i=k+1}^{m} \delta_i \cdot x_{i,k} = 1$ et $\sum_{i=1}^{m} \delta_i \cdot x_i = \delta_0$ utilisées pour le système (4.11).

Expression d'une variable x_j en fonction des x_h, $h < j$

Afin de générer dans le code des communications des références aux variables en fonction uniquement de variables de plus haut rang dans la base, nous avions besoin d'exprimer la variable x_j de l'équation générale $\sum_{i=1}^{m} \delta_i \cdot x_i = c$ en fonction des variables x_h, $h < j$.

On définit récursivement la fonction $\mathcal{B}(l, j)$:

$$\mathcal{B}(l, j) = \sum_{i=l}^{j-1} \frac{\gamma_i}{d_i} \cdot \mathcal{B}(l, i) \cdot x_{j,i}$$

où $x_{j,i}$ est la j-ième composante de la solution particulière de l'équation $\sum_{l=i+1}^{m} \delta_l \cdot x_l = 1$.

La solution x_j s'exprime en fonction des autres variables du système x_h, $h < j$ de la façon suivante :

$$x_j = x_{j,0} + \sum_{h=1}^{j} \mathcal{B}(h,j) \cdot x_{h,0} - \sum_{h=1}^{j-1} \mathcal{B}(h,j) \cdot x_h + d_j \cdot k_j$$

$$x_j = x_{j,0} - \sum_{h=1}^{j-1} \mathcal{B}(h,j) \cdot (x_h - x_{h,0}) + d_j \cdot k_j$$

et

$$x_{m-1} = x_{m-1,0} - \sum_{h=1}^{m-2} \mathcal{B}(h,m) \cdot (x_h - x_{h,0}) + \gamma_m \cdot k_{m-1}$$

$$x_m = x_{m,0} - \sum_{h=1}^{m-2} \mathcal{B}(h,m) \cdot (x_h - x_{h,0}) - \gamma_{m-1} \cdot k_{m-1}$$

4.4.4 Algorithme de génération de code pour DMA

Mon algorithme de génération de code pour les *DMAs* utilise les techniques précédentes. Il est la base des techniques de génération de code des redistributions présentées dans l'article.

Pour simplifier la présentation, nous réécrivons le système spécifiant la *redistribution* (4.10) sous la forme :

$$\begin{array}{ll} \mathcal{E}q_{s,d} & \left\{ \begin{array}{l} \sum_{i=1}^{m_1} \delta_i^1 \cdot x_i^1 = \delta_0^1 \\ .... \\ \sum_{i=1}^{m_k} \delta_i^k \cdot x_i^k = \delta_0^k \\ .... \\ \sum_{i=1}^{m_h} \delta_i^h \cdot x_i^h = \delta_0^h \end{array} \right. \\ \mathcal{I}neq_{s,d} & \left\{ \begin{array}{l} L_{x_*^1} \leq x_*^1 \leq U_{x_*^1} \\ \cdots \\ L_{x_*^k} \leq x_*^k \leq U_{x_*^k} \\ \cdots \\ L_{x_*^h} \leq x_*^h \leq U_{x_*^h} \end{array} \right. \end{array} \tag{4.14}$$

Les termes $(L_{x_i^k};_{1 \leq i \leq m_k};_{1 \leq k \leq m_h})$ et $(U_{x_i^k};_{1 \leq i \leq m_k};_{1 \leq k \leq m_h})$ sont des expressions linéaires représentant les bornes inférieures et supérieures paramétriques de x_i^k.

L'algorithme prend en entrée une redistribution des données à transférer, résultant de l'union d'une **distribution source** et d'une **distribution cible**, caractérisée par un ensemble (4.14) d'équations $\mathcal{E}q_{s,d}$ et d'inéquations $\mathcal{I}neq_{s,d}$, une **liste de variables $\mathcal{L}$** définissant l'ordre d'énumération des composantes x_i pour la génération de code.

Le code de sortie fournit toutes les informations nécessaires aux *DMAs*. Il transfère toutes les données qui satisfont aux contraintes de la redistribution.

L'algorithme se déroule comme suit :

1. Chaque équation eq de $\mathcal{E}_{s,d}$ est remplacée par la liste des équations leq, définissant les m_l variables de l'équation eq et résultant de l'opérateur $Isolve(eq, \mathcal{L})$. Cette

étape introduit $m_l - 1$ nouvelles variables libres k dans le système. Chaque nouvelle équation est de la forme (eq.4.13) :

$$x_l = x_{0_l} + \sum_{j=1}^{l-1} -\gamma_j \cdot x_{l,j} \cdot k_j + d_l \cdot k_l, \quad 1 \leq l \leq m_l$$

Nous savons que $x_{0_l}, \gamma_j, x_{l,j}$ et d_l sont des nombres entiers constants. Seules x_l, k_j et k_l sont des variables.

2. L'algorithme (présenté en section 4.1.2) d'énumération des points entiers d'un polyèdre est utilisé sur ce $\mathbb{Z}$-polyèdre, défini par $\mathcal{E}q_{s,r}$ et $\mathcal{I}neq_{s,r}$. Il permet de trouver les nouvelles bornes de boucles pour les $m_l - 1$ variables libres k_l introduites par l'étape précédente. Toutes les autres variables sont éliminées.
3. Les équations de *leq* sont utilisées pour générer les informations nécessaires aux DMAs. Pour la variable x_l, l'offset est $x_{0_l} + \sum_{j=1}^{l-1} -\gamma_j \cdot x_{l,j} \cdot k_j$, le *pas* entre deux éléments successifs est d_l et le nombre d'éléments à transférer est égal à $U_{k_l} - L_{k_l} + 1$ où L_{k_l} et U_{k_l} sont les bornes inférieure et supérieure de k_l calculées à l'étape précédente.

Cet algorithme est la base des méthodes de génération de code des redistributions présentées dans l'article. Des optimisations permettant de minimiser le nombre de communications, en cas de redistributions avec *recouvrements* par exemple, ajoutent des contraintes au système d'inéquations de la distribution à la source $\mathcal{D}ist_S(A, \vec{p}_s)$ avant de l'utiliser. La section suivante présente rapidement ces optimisations.

Notons également que, pour certaines distributions spécifiques, où une solution particulière simple peut être trouvée à chaque étape du processus du calcul des solutions (eq.4.13), il est possible de générer un code de communication paramétrique et générique (où certains coefficients δ_i^k sont symboliques) avec cet algorithme.

4.4.5 Cas des distributions avec éléments redondants

Les transferts redondants correspondent à des communications multiples et inutiles d'un même élément. Ils peuvent apparaître lorsqu'un même élément est placé sur la source à des emplacements mémoires différentes. Il n'est pas nécessaire de transférer toutes les instances de l'élément présentes sur les différentes mémoire source vers les destinations. Une seule copie suffit. A l'opposé, si l'élément doit être présent dans plusieurs mémoires à la destination, plusieurs communications sont nécessaires.

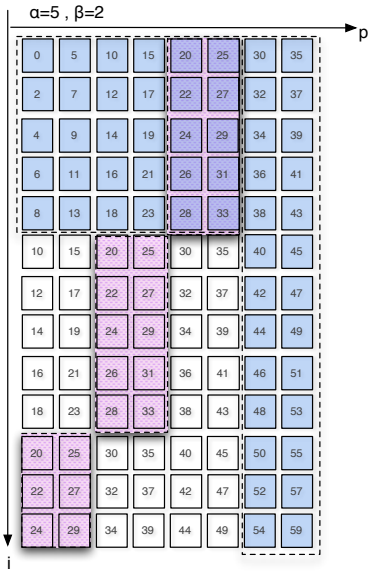
Dans le cadre des travaux menés pour le placement d'applications de traitement du signal, tel que nous venons de le définir en section 4.4.1, des éléments peuvent être présents plusieurs fois dans les distributions car les domaines de définition des itérations internes et des *motifs* sont quelconques. Les réplications d'éléments ne sont pas limitées comme en `HPF` à des dimensions entières de tableau, elles peuvent être multiples.

L'article [44] donne les conditions permettant de détecter la présence d'éléments redondants dans les distributions *source* bidimensionnelles. Puis deux stratégies sont proposées pour extraire des sous-ensembles équivalents de ces distributions ne comportant plus aucun élément redondant.

La première procède avec des coupes des domaines de la distribution à la source, et est généralisable à un nombre quelconque de dimensions. La deuxième technique décompose la distribution en sous-ensembles plus réguliers et comportant une nombre équivalent

d'éléments. Elle permet de générer des codes avec un meilleur équilibrage de charge des communications pour les processeurs. Mais elle est plus difficilement généralisable à des dimensions supérieures à 2.

Au cours de la généralisation, une attention particulière doit être portée aux cas singuliers. Des théorèmes classiques de théorie des nombres, les théorèmes de Paoli [1] et de Cesàro (p.72), ont aidé aux développements de solutions exactes pour la génération automatique des codes de communications *non-redondantes* multidimensionnelles.



Distribution

$$a = \alpha p + \beta i$$

$$\begin{cases} L_p & \leq p \leq & U_p \\ L_i & \leq i \leq & U_i \end{cases}$$

Deux distributions sans redondante

$$\begin{cases} L_p & \leq p \leq & \beta - 1 \\ L_i & \leq i \leq & U_i - \alpha \end{cases}$$

$$et \quad \begin{cases} L_p & \leq p \leq & U_p \\ U_i - \alpha + 1 & \leq i \leq & U_i \end{cases}$$

FIGURE 4.15 – Exemple de distribution 2D des éléments $a \in [0..59]$ sans redondance.

La figure 4.15 présente un exemple de distribution bidimensionnelle de données $a \in [0..59]$ sur une dimension architecturale p. L'équation de distribution est simple $a = 5p + 2i_t$. En reprenant la définition de (4.3), nous avons :

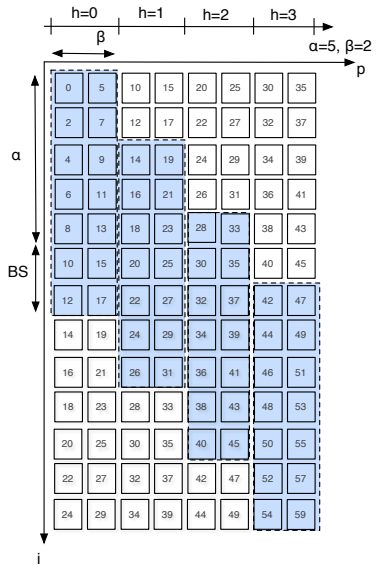
$$0 \leq a \leq 59, 0 \leq i_e < 8, 0 \leq i_t < 13, T = 2, R_e = R_t = 1, L = 1, C = 1, P = 8, E = 5$$

Certains éléments de tableaux sont présents sur plusieurs processeurs p. Notamment les ensembles roses sont périodiquement répétés si les domaines sont suffisamment grands. Les tests de dépendance, classiquement utilisés en parallélisation automatique des boucles, permettent de trouver la base des vecteurs caractérisant une redondance. La première stratégie découpe l'ensemble des éléments distribués en deux sous-ensembles vérifiant les contraintes de droite de la figure 4.15. Nous obtenons alors deux ensembles convexes (en bleu) dont l'union ne comporte pas de doublons.

La seconde stratégie, illustrée sur la figure 4.16, attribue à chaque processeur le même nombre $(BS+\alpha)$ de données (en bleu) à communiquer. BS est le nombre de données qu'il faut ré-attribuer à chaque processeur pour équilibrer les communications. Des contraintes supplémentaires, données à droite de la figure, sont ajoutées pour décaler d'autant les domaines distribués aux processeurs source et ainsi éviter la redondance.

4.5 Conclusion

Écrire manuellement un code en vue de son exécution sur une architecture parallèle est source d'erreurs. Les outils mathématiques et leur mise en œuvre dans des techniques de génération de code automatique permettent d'automatiser cette tâche.



$$a = \alpha p + \beta i$$

$$BS = \frac{U_i + 1 - \alpha}{\frac{U_p - L_p + 1}{\beta}}$$

$$\begin{cases} L_p & \leq p \leq & U_p \\ L_i & \leq i \leq & U_i \\ h & = & p/\beta \\ BS \cdot h & \leq i \leq & BS \cdot +\alpha + BS - 1 \end{cases}$$

FIGURE 4.16 – Exemple de distribution 2D des éléments $a \in [0..59]$ sans redondance avec équilibrage de charge pour les processeurs.

Nous avons montré qu'à partir d'un nombre limité d'algorithmes classiques d'algèbre linéaire, faciles à prouver, il était possible de fournir les outils de base pour :

- déterminer les itérations locales exécutées par les processeurs,
- déterminer des bases de parcours appropriées pour l'énumération des éléments de tableaux accédés au cours des calculs et ceux devant être transférés,
- déterminer l'allocation optimale des tableaux locaux et temporaires en mémoire locale,
- synthétiser des codes de contrôle corrects et efficaces, minimisant les tests et les expressions complexes dans les bornes de boucles,
- synthétiser des codes de communication efficaces (sans redondance) et compatibles avec des accès directs à la mémoires (DMAs).

Les algorithmes développés dans le cadre de ces travaux répondaient à un besoin réel. Par exemple, l'algorithme *row-echelon* a été largement repris dans la littérature (415 citations) pour traiter les problèmes de génération de code à partir de leur formulation polyédrique.

La bibliothèque d'outils mathématiques que nous avons développée s'est enrichie au fur et à mesure des travaux de recherche réalisés. Les algorithmes de base ont été progressivement améliorés, pour prendre en compte précisément les ensembles de points entiers et intégrer les besoins liés aux caractéristiques des nouvelles architectures. Ces outils ont été intégrés dans la bibliothèque linéaire `LinearC3` du compilateur `PIPS`. Ils ont été très largement utilisés et sont robustes.

L'arrivée des architectures telles que les GPU et les *many-cores*, soulève les mêmes difficultés que celles rencontrées pour gérer efficacement et de manière cohérente des mémoires réparties, entre processeurs et coprocesseurs associés. Le placement des données sur des mémoires réparties, ainsi que la génération automatique des transferts induits par leur répartition sont donc plus que jamais d'actualité.

Chapitre 5

Placement d'applications embarquées à l'aide de la programmation concurrente par contraintes

---

Ce chapitre présente mes travaux effectués dans le cadre du placement d'applications de traitement du signal sur des architectures embarquées. Il introduit la modélisation des contraintes devant être respectées pour que la synthèse du code placé sur l'architecture cible soit *correcte*.

Les principaux critères de qualité logicielle visés sont la correction, la robustesse et l'efficacité en temps.

Ces recherches ont été réalisées en collaboration avec François Irigoin, Denis Barthou ; Michel Barreteau et Juliette Mattioli (Thales TRT) ; Jean Jourdan (Thomson-CSF-LCR-ATS Lab.) ; Thierry Grandpierre, Christophe Lavarenne, Yves Sorel (INRIA) ; Frédéric Paquier et Philippe Kajfasz (Thomson-CSF Communications) ; Bernard Dion (SIMULOG) et avec les étudiants en thèse : Christophe Guettier, Nicolas Museux et Isabelle Hurbain.

Mes publications relatives à ce chapitre sont les suivantes : [12, 15, 16, 17, 18, 19, 20, 21, 31, 34, 35, 36, 37, 38, 39, 40, 42, 49, 50, 93, 96]

Les systèmes embarqués sont présents dans des domaines applicatifs très variés : l'automobile (systèmes anti-blocages de freins, informatique de confort), l'avion, le spatial mais aussi dans les produits de grande consommation (cafetières, machines à laver) et dans l'électronique grand public (caméras numériques, appareils photos, multimédia, téléphones portables). Les applications de ces systèmes doivent être développées, validées et opérationnelles dans des délais courts car le matériel évolue très rapidement et la concurrence impose une mise sur le marché au plus tôt. Mais elles doivent surtout tenir compte des ressources restreintes du système telles que le nombre de processeurs, la taille des mémoires locales, la taille de la mémoire globale, la consommation énergétique... Ces contraintes peuvent aussi être temps-réel, comme c'est le cas pour les systèmes antiblocage (ABS) de freinage des voitures qui doivent réagir avec une latence minimale. Pour les systèmes embarqués complexes, la sûreté de fonctionnement reste une priorité.

Les méthodes d'analyses statiques et dynamiques présentées dans les sections précédentes

sont toujours applicables et permettent de vérifier la qualité du code. Mais elles ne garantissent pas que l'exécution d'une application sur la machine cible respecte toutes les contraintes architecturales du système. Par exemple, il n'est pas certain que les données utilisées au cours de l'exécution tiennent dans la mémoire locale d'un processeur.

Dans les travaux présentés dans ce chapitre, il s'agit de trouver les paramètres de placement d'une application sur une architecture de telle sorte que son exécution respecte l'ensemble des contraintes du système. La correction ne s'applique qu'aux éléments de placement.

Les applications *embarquées* comportent des spécificités qui doivent être prises en compte lors de l'analyse et du placement. Ces spécificités peuvent être :

Matérielles Le nombre de processeurs, la taille des mémoires, la latence, le débit du réseau de communications et celles des entrées-sorties sont des facteurs importants pour l'ordonnancement.

Applicatives La dimension *infinie* du temps, les traitements systématiques sont des caractéristiques importantes pour la mémoire.

Utilisation de bibliothèques L'utilisation de fonctions de bibliothèque optimisées est fréquente. Le placement des parties de l'application qui y font appel nécessite des informations sur 1) les temps d'exécution des fonctions de bibliothèque sur l'architecture cible, 2) leur comportement en mémoire (taille des paramètres en entrée et sortie, taille des données privées à allouer, taille mémoire du code des fonctions) et 3) les effets de bord éventuels sur l'environnement d'exécution.

Ce sont autant de caractéristiques qui jouent un rôle important dans la mise en œuvre des solutions. Elles constituent non seulement un ensemble de contraintes devant être respectées mais ont aussi des spécificités permettant l'optimisation.

Les articles [42, 49, 50, 93] présentent un aperçu des travaux auxquels j'ai participé sur ce thème. Les résultats ont été le fruit d'une large collaboration qui s'est poursuivie durant une dizaine d'année avec deux thèses [89, 134] et quatre projets `ARRAY-OL`, `PSP-RBE2`,`PROMPT` et `DREAM-UP` (pages ??-??). J'ai principalement contribué à la modélisation du code, des transformations possibles et des machines cibles. Cette étape est le préalable essentiel à une synthèse de code placé *correct*.

Ces travaux reposent sur une approche multi-modèles et utilise la programmation par contraintes comme méthode de résolution du problème global du placement. Ils visaient initialement des machines parallèles homogènes `SPMD`, puis ils ont été étendus à du multi-`SPMD`. Les applications embarquées temps-réel que nous avons traitées sont des applications de traitements systématiques du signal et d'images. Elles possèdent un fort potentiel de parallélisme lié aux données et les temps d'exécution des fonctions sont statiquement prédictibles. La majeure partie du temps d'exécution est passé dans les nids de boucles qui traitent des données sous forme de tableaux. Placer efficacement ces nids de boucles de calcul, tout en garantissant le respect de toutes les contraintes architecturales, représentait l'enjeu essentiel de ces recherches.

5.1 Une approche globale

Le problème de l'optimisation globale du placement se décompose en plusieurs sous-problèmes : 1) le respect des dépendances de données, 2) le partitionnement en tâches parallèles, 3) l'ordonnancement, 4) la minimisation de la mémoire, 5) le recouvrement des

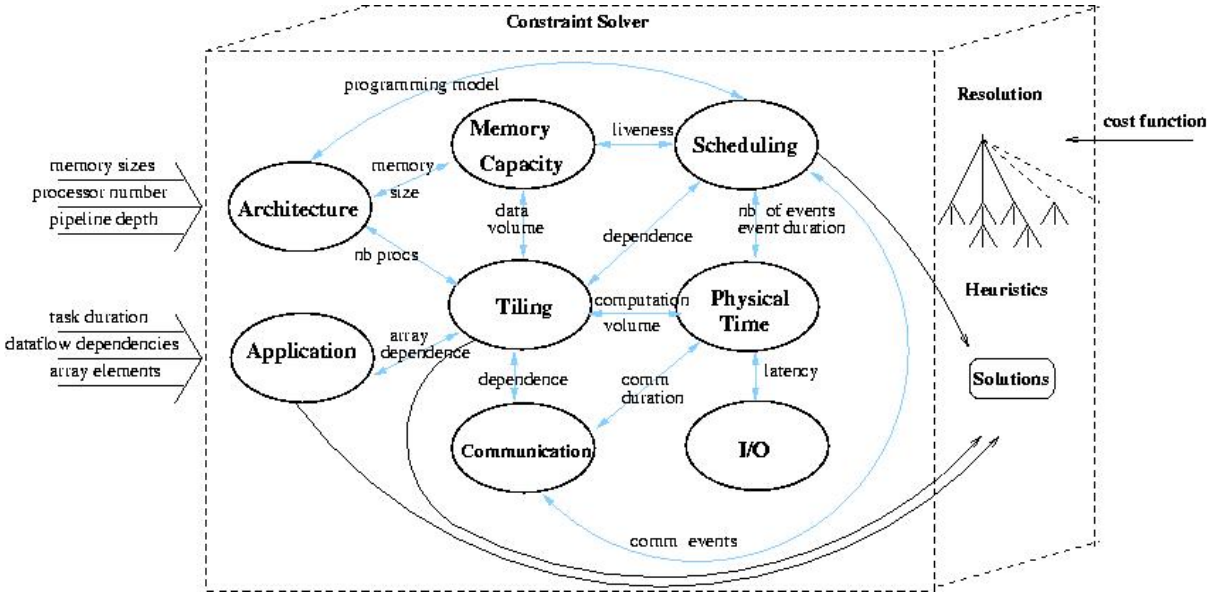


FIGURE 5.1 – L'approche de programmation concurrente par contraintes

calculs et des communications, 6) la minimisation de latence, ... Chaque sous-problème est un problème d'optimisation en soi. Seule une approche concurrente par modèles pouvait répondre à nos besoins [84, 77].

La figure 5.1 présente l'approche globale de résolution choisie : la programmation concurrente par contraintes. L'application, l'architecture, la mémoire, le partitionnement, l'ordonnancement des tâches, les communications, les dépendances de données et les temps de latence sont tous modélisés avec des contraintes affines et non-affines. Ces modèles sont introduits dans la section 5.2.

L'outil `APOTRES`, qui a été développé dans le cadre de ce projet, prend en entrée les paramètres applicatifs et architecturaux. Un fichier de spécification précise le nombre de processeurs, les tailles des mémoires globale et locales, la profondeur du pipeline. Les informations applicatives sont automatiquement extraites par le compilateur `PIPS`. Les durées des temps d'exécution des tâches (séquences d'instructions) sont estimées en utilisant les travaux de L. Zhou [178] et les éléments des tableaux accédés par chaque tâche sont évalués en utilisant les régions convexes de tableaux (section 2.5, p.26).

Ces informations sont prises en compte et propagées, avec les informations déduites ou calculées par les modèles. Au cours de la résolution les modèles échangent des informations sur les intervalles de valeurs de leurs variables. La résolution utilise des heuristiques spécifiques à la fonction de coût choisie : le temps d'exécution, la taille de la mémoire, le coût de l'architecture ou la latence. Toutes les solutions satisfont à toutes les contraintes des modèles et le processus de recherche est totalement automatique. Il n'y a jamais de *mauvaise* solution, car toute solution produite respecte toutes les contraintes.

5.2 Les modèles

Le placement d'une application sur une architecture utilise différents modèles qui interagissent (cf. figure 5.1). J'introduis dans cette section quelques contraintes utilisées dans les modèles principaux pour expliquer le principe de notre approche.

Partitionnement

Nous reprenons, pour le partitionnement des boucles, la même formulation que celle présentée en section 4.4.1 dans l'équation 4.3 (page 67).

$$\begin{cases} B\vec{\imath} \leq \vec{b} \\ 0 \leq P^{-1}\vec{p} < 1 \ , \\ 0 \leq L^{-1}\vec{l} < 1, \\ 0 \leq C^{-1}\vec{c} < 1, \\ \vec{\imath} = LP\vec{c} + L\vec{p} + \vec{l} \end{cases} \tag{5.1}$$

Le partitionnement distribue les itérations selon 3 dimensions : (1) une dimension cyclique temporelle c, (2) une dimension processeur p, (3) une dimension locale l qui exploite la mémoire locale. A chaque temps c, p processeurs exécutent un bloc de l itérations. Une éventuelle dimension *infinie* du temps est associée à la dimension cyclique c la plus externe dans le nid de boucles et un traitement particulier est fait pour le calcul de ses bornes.

Temps d'exécution

Les temps d'exécution sont calculés au pire cas, en utilisant les techniques développées dans la thèse de Lei Zhou [178] qui ont ensuite été complétées par les étudiantes Prachi Kalra [110] et Molka Becher [53].

Ordonnancement

Le modèle d'ordonnancement associe à chaque bloc de calculs c une *date logique* de son exécution sur un processeur. C'est un ordonnancement événementiel qui permet de décorréler le *temps réel* des choix effectués pour le partitionnement et de définir un ordre total entre les blocs de calculs.

Les applications de traitement du signal systématique sont adaptées à la classe des ordonnancements périodiques. Ces derniers ont l'avantage de permettre une prédiction plus précise des temps d'exécution et des consommations mémoire nécessaires.

Un ordonnancement monodimensionnel affine a été choisi. Les ordres d'exécution sont calculés en fonction des partitionnements. Un ordonnancement est légal s'il respecte les contraintes de flots de données. Un bloc de calculs c^k du k-iéme nid de boucles, s'exécutera à la date logique $d^k(c^k)$ définie par l'équation suivante :

$$d^k(c^k) = \alpha^k.c^k + \beta^k \tag{5.2}$$

où c^k est une itération d'un bloc de calculs, α^k est un vecteur de la même dimension que c^k, "." est le produit scalaire standard et β^k est un entier. Le solveur sélectionne des valeurs pour chaque α^k et β^k.

De même nous pouvons exprimer les contraintes d'ordonnancement entre deux blocs c_i^k et c_j^k d'un même nid de boucles B^k. Pour assurer l'ordre total local d'exécution des blocs de calculs, les $\alpha^k = \begin{pmatrix} \alpha_1^k \\ . \\ . \\ \alpha_n^k \end{pmatrix}$ sont contraints par :

$$\forall 1 \leq i \leq n-1, \quad \alpha_i^k \geq \alpha_{i+1}^k \cdot max(c_{i+1}^k), \quad \alpha_n^k \geq 1 \tag{5.3}$$

ainsi

$$\forall(c_i^k, c_j^k), (c_i^k \prec c_j^k) \Leftrightarrow (d^k(c_i^k) < d^k(c_j^k))$$

Dépendances de flots de données

Les contraintes de dépendances de données relie le partitionnement au modèle d'ordonnancement. Si un bloc c^r d'un nid de boucles B^r utilise une valeur définit par un bloc c^w d'un nid de boucles N^w, alors c^w doit être calculé en premier. S'il y a des dépendances entre deux blocs de calcul c^r et c^w, alors un ordonnancement légal doit respecter la contrainte

$$d^w(c^w) + 1 \leq d^r(c^r) \tag{5.4}$$

où d^r (respectivement d^w) est le temps logique associé au nid de boucles B^r (respectivement B^w).

Architecture SPMD

Dans le modèle de programmation SPMD, tous les processeurs actifs effectuent au même instant le même programme. De manière à éviter que l'exécution de deux blocs de calculs de deux nids de boucles différents soient planifiés à la même date logique et dans le but de simplifier le calcul et la propagation des fonctions d'ordonnancement des différents nids de boucles, nous utilisons l'ordonnancement suivant :

$$d^k(c^k) = N(\alpha^k.c^k + \beta^k) + k \tag{5.5}$$

où N est le nombre de nids de boucles total et d^k est l'ordonnancement associé au k-ième nid de boucles.

Cette méthode introduit des dates de non-événements qui ne sont heureusement pas prises en compte lors de calculs de latence en temps réel. Cependant, la génération de code reste délicate. N. Museux propose de combiner les travaux de Jean-François Collard & al. [68], basés sur un contrôle dynamique de l'ordre d'exécution des blocs de calculs, et ceux de Denis Barthou & al. [51], basés sur la recherche d'une expression régulière à partir des fonctions d'ordonnancement. Des contraintes supplémentaires sur les vecteurs α, des blocs de calculs dépendants, peuvent aussi faciliter la génération de code, mais conduire à des solutions qui s'éloignent de l'optimum de la fonction de coût choisie. Il s'agit donc de trouver le meilleur compromis entre la facilité d'implémentation des solutions proposées par `APOTRES` et l'optimum recherché.

Capacité mémoire

Un modèle capacitif a été défini. Comme tous les processeurs actifs exécutent le même code, la taille de la mémoire requise est la même pour chaque processeur. Chaque tâche (bloc de calculs) alloue ses buffers privés en entrée, mais toutes les tâches partagent un même buffer en sortie. Aussitôt les calculs effectués, les résultats sont envoyés aux buffers d'entrée des tâches qui vont les consommer. Le buffer en sortie a une taille permettant de recevoir les sorties de n'importe quelle tâche et peut supporter le mécanisme de flip-flop utilisé pour recouvrir les calculs et les communications.

La taille d'un buffer d'entrée est la somme des espaces requis pour chaque argument. Sa capacité minimum est le minimum obtenu pour les 4 schémas d'allocation suivants :

1. la taille maximale du tableau ;
2. le volume de l'hypercube accédé par tous les cycles de calcul ;
3. le volume des données accédées par cycle multiplié par le nombre maximal d'itérations vivantes, pour les cas de recouvrements des données. L'information de *liveness* est calculée à partir des dépendances et de l'ordonnancement ;
4. le nombre de données accédées par cycle multiplié par le nombre maximal d'itérations vivantes, utiles en cas d'accès non-contigus.

Les contraintes mémoire sont liées au partitionnement, aux dépendances et aux paramètres d'ordonnancement (cf. figure 5.1).

La thèse de N. Museux [134] détaille les contraintes portant sur le volume mémoire des données à allouer 1) en émission et en réception pour les éléments calculés par une tâche et communiqués à une autre, 2) en émission et réception pour les entrées-sorties (I/O), 3) en interne pour les tableaux locaux à la tâche et 4) globalement pour l'exécution l'application.

5.3 Un modèle de programmation logique concurrent par contraintes (PLCC)

Certaines contraintes peuvent être traduites sous la forme d'équations et d'inéquations linéaires et résolues par des algorithmes classiques de programmation linéaire, d'autres, comme les contraintes de ressources, exigent des traitements **non linéaires**. Les techniques de résolution pour des systèmes qui comportent à la fois des contraintes linéaires et des contraintes non linéaires sont très complexes. Elles nécessitent des méthodes combinant de la programmation en nombres entiers et des techniques de recherche telles que la programmation dynamique, la théorie des graphes, les méthodes arborescentes, les algorithmes de *Branch-and-Bound* ou encore la programmation par contraintes.

Un problème de programmation par contraintes se caractérise par un ensemble de relations (les contraintes) entre des variables ayant chacune son domaine (représenté par un ensemble de valeurs ou des intervalles). Une solution est une instanciation cohérente de ces variables. L'espace des instances possibles est représenté par un arbre de nœuds (les variables) et d'arcs (les valeurs). Les branches, de la racine aux feuilles, forment les solutions du problème.

La recherche de solutions repose sur deux mécanismes : la propagation des contraintes et l'énumération des valeurs. Le premier est chargé du maintien de la cohérence partielle du système, en propageant les contraintes, et en réduisant les domaines des variables aux valeurs pour lesquelles il peut exister des solutions. Le second construit l'arbre de recherche, à l'aide d'heuristiques pour ordonner les variables et leurs valeurs. Après chaque affectation, il valide les décisions en faisant appel à nouveau à la propagation des contraintes. Les deux mécanismes sont liés et s'exécutent jusqu'à ce que toutes les variables soient instanciées à des valeurs respectant les contraintes.

La programmation logique concurrent par contraintes (`PLCC`) gère des contraintes linéaires et non linéaires et fournit, à travers la propagation concurrente des contraintes entre tous les modèles, des solutions répondant au problème global. Les modèles, présentés en section 5.2, sont liés par les variables qui servent simultanément à leurs spécifications.

Par exemple, la contrainte 5.6 relie le modèle de partitionnement et le modèle de l'architecture. Sous l'hypothèse `SIMD` ou `SPMD`, pour chaque nid de boucles k, le nombre de processeurs requis par le partitionnement doit être plus petit que le nombre de processeurs disponibles `ProcessorNumber`.

$$\forall k \quad ProcessorNumber \geq max_k(det(P^k)) \tag{5.6}$$

où P^k est la matrice de partitionnement des processeurs pour le nid de boucles k. Dans notre contexte, P^k est diagonale, nous avons $det(P^k) = \prod_{i=1}^{n_k} P^k_{i,i}$.

Un autre exemple, l'indice d'un bloc de calcul c apparaît :

— dans le modèle de partitionnement comme l'indice de bloc du nid de boucles partitionné (eq. 5.1)

— dans le modèle des dépendances de données dans les relations de précédence (eq. 5.4) ;

— dans le modèle d'ordonnancement (eq. 5.2).

Au cours du processus de résolution, les modèles communiquent leurs informations partielles : les intervalles de valeurs de leurs variables respectives.

Le système `PLCC` construit un espace de solutions modèle par modèle. La recherche globale cherche des solutions partielles parmi les différents modèles concurrents. Seules les informations pertinentes se propagent entre les modèles. Tout le raisonnement mis en œuvre par les contraintes est basé sur le paradigme de la manipulation d'informations partielles. L'avantage est que le système peut prendre des décisions sans attendre que l'ensemble soit déterminé. À chaque prise de décision l'ensemble des variables et contraintes reste cohérent.

Plusieurs heuristiques globales sont utilisées pour améliorer la résolution, par exemple, pour l'ordonnancement, les choix sont guidés par le calcul du plus court chemin dans le graphe de flot de données. L'outil fournit un ordonnancement au plus tôt.

L'article [93] présente les résultats obtenus pour trois critères d'optimisation :

— le premier donne la configuration du circuit la moins onéreuse capable d'exécuter l'application et respectant ou non la contrainte temps-réel ;

— le deuxième minimise la mémoire et donne la taille minimale nécessaire à l'exécution par processeur ;

— et le troisième optimise le temps d'exécution et fournit des solutions qui satisfont à la fois les contraintes architecturales et les contraintes temps-réel.

Minimisation du coût de l'architecture La réduction des coûts de l'architecture est clé pour les systèmes embarqués. Elle dépend du nombre de processeurs et de la taille des mémoires. On suppose que le coût du processeur est prépondérant par rapport à la mémoire. Sans contrainte mémoire, l'outil sélectionne des solutions avec le nombre minimal de processeurs. Ils peuvent exécuter un grand nombre d'itérations par bloc car il n'y a pas de contrainte mémoire. Si une contrainte sur la mémoire disponible sur chaque processeur est ajoutée, les solutions comportent plus de processeurs que précédemment. Le nombre d'itérations par bloc est réduit afin de diminuer le nombre de variables vivantes et le volume de mémoire locale nécessaire à leurs stockages.

Minimisation de la mémoire Ce critère permet de connaître la taille minimale de mémoire requise pour exécuter l'application. Les solutions comportent un grand nombre de processeurs et un nombre d'itérations par bloc minimal.

Minimisation des temps d'exécution sous contrainte mémoire Les résultats des optimisations précédentes permettent d'obtenir des intervalles de valeurs concernant les temps d'exécution en fonction de la taille de la mémoire par processeur.

En s'inspirant de ces résultats et en imposant des contraintes acceptables pour la mémoire, la minimisation des temps d'exécution favorise les solutions avec un plus grand nombre de processeurs et très peu de blocs. L'ordonnancement n'est presque plus entrelacé. Si on limite à nouveau davantage la mémoire, un partitionnement supplémentaire des itérations est appliqué.

Expériences

Un outil d'aide au placement d'applications de traitement du signal, appelé `APOTRES`, à base de `PLCC`, a été développé par Thales [122] et le centre de recherche en informatique de `MINES ParisTech`. Il utilise le solveur `ECLAIR` [59], une bibliothèque de contraintes sur les domaines finis, également développée par Thales et construite au dessus de `CLAIRE` [58, 60], qui est un langage de programmation fonctionnel. La procédure d'optimisation utilise un algorithme classique de *Branch-and-Bound*. `APOTRES` a été couplé avec `PIPS`. Après analyse des applications, `PIPS` génère automatiquement les contraintes en langage `CLAIRE`. Elles sont alors directement exploitées par le solveur `ECLAIR`.

Les modèles développés sont dédiés au traitement du signal systématique et sont obtenus à partir de techniques de parallélisation présentées dans [28]. Les expériences ont été menées sur un ensemble de *benchmarks* fournis par les partenaires industriels des projets. Les fonctions de coûts différentes ont permis aux utilisateurs d'obtenir automatiquement une variété de solutions, très proches de celles obtenues par des experts.

Toutefois, l'aspect concurrentiel de la résolution du solveur et des techniques de propagation des informations inter-modèles mériteraient une analyse et des recherches approfondies supplémentaires. Car la bonne propagation des informations nécessitent parfois des approximations, ou des contraintes redondantes, dans les modèles.

5.4 Conclusion

L'utilisation de la programmation par contraintes pour le placement d'applications de traitements du signal systématiques temps-réel sur une architecture embarquée permet une exploration efficace de l'espace combinatoire des partitionnements et ordonnancements affines possibles.

Cette approche est automatique. Elle nécessite une modélisation fine de l'architecture et de l'application ainsi que des heuristiques explicites permettant de guider l'exploration de l'espace des solutions. Les contraintes peuvent être **linéaires et non linéaires**, ce qui donne le pouvoir d'expression nécessaire à la modélisation du problème *global* du placement des applications de traitement du signal systématique sur une architecture embarquée.

L'approche de modélisation par contraintes rend faisable et automatise la recherche d'un placement *correct*[1] de ce type d'applications. Il reste à optimiser en temps d'exécution la recherche de solution et à étendre l'ensemble des applications traitées.

Les détails des modèles présentés dans ce chapitre et la présentation des autres modèles implantés dans le cadre de ces travaux tels que les communications, la latence, les entrées-sorties, les mémoires hiérarchiques... sont détaillés dans les articles [122][122, 42, 49, 50, 93], de nombreux rapports [12, 15, 16, 17, 18, 19, 20, 21, 96] et deux thèses [89, 134] de Christophe Guettier et Nicolas Museux.

1. Un placement est correct s'il respecte les contraintes de ressources du système

Chapitre 6

Contributions et perspectives

It seems to me now that mathematics is capable of an artistic excellence as great as that of any music, perhaps greater ; not because the pleasure it gives is comparable, either in intensity or in the number of people who feel it, to that of music, but because it gives in absolute perfection that combination, characteristic of great art, of godlike freedom, with the sense of inevitable destiny ; because, in fact, it constructs an ideal world where everything is perfect and yet true.

Bertand Russel, *Letter to Gilbert Murray, April 3, 1902*

Les machines multiprocesseurs et multi-cœurs se généralisent et pourtant il reste difficile pour les programmeurs de tirer profit de leurs capacités. Les langages *parallèles* (`HPF`, OpenMP, OpenCL, CUDA, Cilk, X10, Chapel, ...) [116, 117] au parallélisme explicite permettent l'écriture de versions optimisées des applications, mais dédiées à un type d'architectures et donc non portables. La compilation des applications écrites dans des langages plus classiques (C, C++, MATLAB...) est moins efficace sur les architectures parallèles, si on ne les restreint pas avec des règles de codage. Une étude récente montrait que seul 3% des développeurs introduisent des directives de parallélisation dans leurs applications. La parallélisation des applications reste encore réservée aux spécialistes. Des outils d'aide à la parallélisation, permettant de réduire les coûts de développement, sont nécessaires.

Mes travaux de recherche passés et mes projets à venir se répartissent selon trois axes principaux. Les deux premiers, la vérification des applications et la synthèse de code, relèvent de la compilation et de l'optimisation d'applications scientifiques en vue de leur exécution efficace sur des architectures parallèles. Le troisième concerne les mathématiques appliquées et l'utilisation de l'algèbre linéaire en nombres entiers pour modéliser les problèmes rencontrés.

6.1 Axe 1 : Vérifications des applications

Parmi les méthodes de vérification de programmes, nous avons montré que l'analyse statique de programmes pouvait fournir de très bons résultats pour des codes industriels de taille importante.

Mes contributions dans ce domaine concernent :

— la mise en conformité par rapport à la norme du langage source,
— la détection de variables non initialisées avant leur utilisation,
— la vérification du non-débordement des accès à des éléments de tableaux,
— le calcul automatique des informations nécessaires à la validation de certaines transformations de programme (calcul des dépendances),
— le calcul d'invariants.

Ces contributions ont été concrétisées sous forme de passes, présentées en section 3, et intégrées dans l'atelier logiciel `PIPS` que j'ai déployé en milieu industriel sur des codes de simulation de mécanique des structures, de mécanique des fluides et de physique.

L'aspect original de ces recherches était la combinaison des analyses statiques et dynamiques. Les analyses dynamiques seules permettent de vérifier le code en fonction de cas tests que l'on exécute. Mais ils ne garantissent pas que le programme source est *sûr* pour d'autres données en entrée. L'association des analyses statiques et de l'instrumentation de code permet de garantir que les programmes sont de qualité pour toutes les exécutions possibles. Le nombre de tests dynamiques générés et effectués lors de l'exécution est très fortement réduit grâce aux résultats fournis à la compilation par les analyses statiques. L'instrumentation du code le protège contre toutes références non valides aux variables du programme, non seulement pour les cas tests disponibles mais aussi pour tous les cas tests futurs.

Le déploiement de ces techniques a permis de vérifier des applications industrielles de taille importante (120 KLignes) et de les sécuriser après détection d'erreurs. Le caractère source-à-source de nos analyses était essentiel. Les résultats fournis ont pu, en effet, être validés rapidement par les équipes de développement du partenaire industriel et intégrés aux applications.

Projet : Les analyses statiques pour des programmes parallèles

J'aimerais désormais approfondir les techniques d'analyse de code dans le cadre de programmes *parallèles*. Une transformation peut être appliquée sur une partie de programme si ce dernier vérifie certaines propriétés. Par exemple, la parallélisation de boucle ne s'applique que si les dépendances de données sont respectées. De nombreuses analyses ont déjà été développées dans le compilateur `PIPS` afin de fournir les informations nécessaires à l'application d'optimisations. Cependant, ces analyses ont été réalisées pour des programmes séquentiels. Des analyses telles que l'estimation du nombre d'opérations dans un bloc d'instructions et la caractérisation des éléments de tableaux référencés dans une section de code ont besoin d'être améliorées pour être plus précises, voire correctes, dans le cadre de sections parallèles.

6.2 Axe 2 : Synthèse de code

Écrire manuellement un code est source d'erreurs. Les outils mathématiques et leur mise en œuvre dans des techniques de génération de code automatique permettent d'automatiser partiellement cette tâche.

Mes contributions sur ce thème concernent la modélisation affine des problèmes de placement d'applications sur des machines parallèles et la synthèse de code, corrects par construction, à partir de ces modélisations. Nos travaux ont permis de proposer :

- une modélisation précise du placement des données sur les processeurs, pour des programmes distribués sur une architecture émulant une mémoire virtuelle partagée et pour des programmes `HPF`,
- une modélisation des calculs pouvant être exécutés en parallèle pour ces mêmes programmes,
- une modélisation des communications devant être générées pour conserver la cohérence des données sur différents types d'architectures distribuées,
- des algorithmes de synthèse de code de contrôle et de communications.

Dans le cadre de la programmation pour machines à mémoire répartie, nous avons montré qu'à partir d'un nombre limité d'algorithmes classiques d'algèbre linéaire, il était possible de fournir les outils de base pour :

- déterminer les itérations locales exécutées par les processeurs,
- déterminer des bases de parcours appropriées pour l'énumération des éléments de tableaux accédés au cours des calculs et ceux devant être transférés,
- déterminer l'allocation optimale des tableaux locaux et temporaires en mémoire locale,
- synthétiser des codes de contrôle corrects et efficaces, minimisant les tests et les expressions complexes dans les bornes de boucles,
- synthétiser des codes de communications efficaces (sans redondance) et compatibles avec des accès directs à la mémoires (DMAs).

L'algorithme de génération automatique d'un code qui parcourt l'ensemble des points entiers d'un polyèdre *row-echelon* a évolué en fonction des besoins. La première version a été présentée dans l'article [41]. Une deuxième version existe également dans ma thèse [46]. Elle traite de la génération de code de parcours pour des ensembles non-convexes, résultant de l'image par une transformation affine d'un polyèdre convexe. Elle est appliquée aux codes de transfert pour des multiprocesseurs à mémoires locales. Cette version intègre les changements de bases à partir de Hermite et le partitionnement des ensembles non convexes en polyèdres pour optimiser le code généré. Un nouvel algorithme a été proposé dans [44] pour cibler les contrôleurs DMAs et générer des codes de communication appropriés, pour des ensembles de points appartenant à des treillis non unitaires.

La modélisation des contraintes devant être respectées pour un placement correct des applications temps-réel sur une architecture parallèle embarquée représente une étape préalable et essentielle à la synthèse d'un code. J'ai contribué à la définition de nombreux modèles dans le cadre des projets `ARRAY-OL`, `PSP-RBE2`, `PROMPT` et `DREAM-UP` (pages **??**-**??**).

Concernant la synthèse de code, mes projets s'inscrivent autour des problématiques suivantes :

1. définir des méthodologies de génération de code distribué dont on puisse prouver la correction par étapes,
2. aller au delà du *linéaire* dans la modélisation du placement des applications pa-

rallèles.

Projet 1 : Génération de code parallèle *vérifiable*

L'étape de génération de code parallèle est complexe, particulièrement pour les architectures à mémoire distribuée, à hiérarchie mémoire, voire mixte partagée et distribuée. La méthode traditionnellement utilisée vise à repérer les parties de l'application qui peuvent profiter d'une exécution parallèle [114], puis à proposer un *mapping* de ces parties sur les processeurs (virtuels) de la machine cible [115]. L'étape de génération de code introduit ensuite les communications, les synchronisations et tous les appels au *run-time*, dépendant du langage parallèle choisi, nécessaire à son exécution. Cette étape est généralement réalisée en une seule fois et prouver sa correction est difficile.

L'objectif de ce thème de recherche consiste, à partir d'une application écrite en C et d'un mapping donné des instructions sur les processeurs, à définir l'ensemble des transformations du code nécessaires jusqu'à sa spécialisation pour un processeur. Chacune de ces transformations doit être suffisamment *simple* pour que l'on puisse garantir la correction des transformations au fur et à mesure. Dans un premier temps, il faut définir les étapes introduisant les futures composantes parallèles, par exemple des copies de variables, puis trouver les optimisations applicables au code pour chaque processeur avant d'effectuer l'étape de spécialisation finale qui introduit les communications et synchronisations.

Projet 2 : Modélisation pour le problème du placement "au delà du linéaire"

Une deuxième problématique que j'aimerais étudier de manière plus approfondie concerne la modélisation du problème du placement. La modélisation du placement des applications sur des machines parallèles nécessite l'utilisation d'un grand nombre de paramètres afin d'exprimer la généricité. Dans le cadre des thèses de Christophe Guettier et de Nicolas Museux encadrées au CRI, nous avions pu constater que la programmation logique concurrente par contraintes rendait possible l'exploration de solutions d'ordonnancement parallèle des calculs et des communications avec une modélisation dont l'expression va au delà du *linéaire*. Je souhaite approfondir ces recherches. Les ordinateurs actuels fournissent la puissance nécessaire aux méthodes d'exploration et les moteurs de recherche de solutions des systèmes sont eux-mêmes concurrents. Cette approche est dynamique, car certains intervalles de valeurs des paramètres doivent être connus numériquement, et complémentaire des solutions affines que nous cherchons à trouver dans la cadre de nos analyses statiques.

Cette problématique rejoint celle exprimée par P. Feautier [85] d'extension des techniques de compilation pour intégrer des contraintes polynomiales et le développement de nouveaux algorithmes de calcul des tests de dépendances, d'ordonnancement, de partitionnement et de génération de codes à partir d'informations polynomiales.

6.3 Axe 3 : Cadre algébrique - le modèle polyédrique

Concernant l'aspect mathématiques appliquées, j'ai contribué à la création et aux développements de la bibliothèque linéaire `LinearC3` depuis ses débuts en 1987. Cette bibliothèque s'est enrichie au fur et à mesure de nos projets. Les algorithmes de base ont été progressivement améliorés pour :

- prendre en compte précisément les ensembles de points entiers,
- intégrer des mécanismes de gestion des débordements pour les cas limites (*overflows* et *underflows*),
- intégrer des algorithmes de manipulation efficaces des structures (vecteurs, systèmes de contraintes, matrices) couramment utilisées par les analyses de programmes et la synthèse de code.

Les fonctions de la bibliothèque sont utilisées de manière intensive par le compilateur `PIPS`. Elles sont maintenant très robustes. Ces techniques ont montré en pratique des temps d'exécution très raisonnables, et commencent à être intégrées dans des compilateurs plus généraux tels que `Graphite/GCC`, `Polly/LLVM`.

Pourquoi avoir choisi les polyèdres plutôt que l'arithmétique de Presburger ?

Les formules de Presburger permettent l'utilisation des opérateurs logiques $\neg, \wedge, \vee$, des contraintes affines et des quantificateurs $\exists$ et $\forall$. Elles sont souvent employées pour la vérification des systèmes parce que l'arithmétique de Presburger est décidable [147]. Deux représentations servent à sa représentation et sa manipulation. Elles sont basées sur des automates, déterministes ou non-déterministes [82], ou sur des polyèdres [149]. Le principal challenge dans l'utilisation de l'arithmétique de Presburger est l'efficacité. La procédure la plus connue pour décider de l'arithmétique de Presburger peut conduire, dans le pire cas, à un automate de taille triplement exponentielle de la taille de la formule [145]. Avec une représentation polyédrique, la première étape de la vérification de la satisfiabilité est l'élimination des quantificateurs. Plusieurs algorithmes ont été développés [131]. L'ensemble des solutions s'exprime alors comme une liste de systèmes d'égalités et d'inégalités affines et de congruences [156].

L'arithmétique de Presburger offre une meilleure précision qu'un polyèdre (approximation compacte par un seul système de contraintes affines). Mais l'utilisation de listes de polyèdres comme représentation pour des analyses qui manipulent et combinent cette abstraction, telles que les régions convexes de tableaux, s'avère très complexe. Cependant, l'arithmétique de Presburger est précise et efficace pour tester la faisabilité de dépendances dans le cadre de la parallélisation de programmes [149].

L'intérêt principal du modèle polyédrique est de pouvoir exprimer, à partir d'une même abstraction, à la fois :

- les domaines des variables du programme,
- les résultats des analyses statiques : les prédicats sur les variables peuvent être représentés, ou approximés, par des polyèdres,
- la spécification des problèmes à résoudre : un grand nombre de transformations de programmes peuvent s'exprimer comme une succession de combinaison de changements de base, de transformations affines et de projections sur des domaines polyédriques,
- des informations que l'on peut traduire quasiment directement sous une forme textuelle et lisible : les contraintes expriment des intervalles de valeurs pour les variables ou des bornes de boucles, ...

Cette unicité évite une perte de précision sur les résultats, potentiellement engendrée par

des conversions entre abstractions différentes.

Le polyèdre peut, lui-même, être représenté soit par un système de contraintes, soit par un système générateur. Les conversions d'une représentation à l'autre sont exactes mais coûteuses en temps. Elles sont limitées dans nos implémentations à des opérateurs particuliers pour lesquels le gain est avéré : l'union convexe de deux systèmes générateurs est plus rapide que pour des systèmes de contraintes, pour leur intersection c'est le contraire. La représentation la plus communément utilisée dans la bibliothèque `LinearC3` est le système de contraintes.

Cependant, indépendamment de sa représentation, il n'existe pas de forme normale pour un polyèdre. Même si un polyèdre particulier représente l'ensemble exact des éléments recherchés, il n'y a pas unicité sur sa forme. Cette propriété m'a conduite à développer de nombreux algorithmes et heuristiques pour tenter de 1) *normaliser* les systèmes en réduisant, quand cela est possible, les coefficients des variables dans les contraintes, 2) *éliminer les contraintes redondantes* dans les systèmes et 3) formaliser finement le problème en amont pour *forcer* une formulation adéquate et précise des solutions recherchées.

Plusieurs stratégies ont été implantées pour prendre en considération

— le nombre de variables dans les contraintes,
— les coefficients des variables (la projection d'une variable peut conduite à des coefficients plus grands),
— le nombre de contraintes sur une variable en fonction de son rang dans la base du système,
— le typage des variables (entiers ou rationnels),
— l'impact de la complexité de ces contraintes sur le code généré automatiquement à partir de ces dernières.

Le modèle polyédrique fournit des outils puissants. Dans le cadre de l'optimisation d'applications pour des architectures parallèles, j'ai cherché à assurer leur meilleure mise en œuvre dans les méthodes présentées, afin de proposer des solutions correctes et efficaces.

Projet : Modélisation et pré-conditionnement des polyèdres

Les outils d'algèbre linéaire paraissent simples, mais trouver une solution efficace à un problème reste parfois difficile. L'algorithme du simplexe [1] en est un bon exemple. Il est simple dans son principe, mais délicat dans sa mise en œuvre. De nombreuses implantations existent car leurs performances dépendent très fortement des systèmes en entrée. D'où l'intérêt porté par certains chercheurs [157, 158], pendant plusieurs années, au préconditionnement des systèmes en entrée. Pour les méthodes polyédriques, développées dans le cadre de la compilation et de synthèse de code, je souhaite approfondir l'impact du *pré-conditionnment* de nos systèmes en entrée et celui de la modélisation sur les algorithmes utilisés. Répondre, par exemple, à la question : faut-il toujours rechercher la présence potentielle d'équations parmi les inéquations dans un système ?

1. L'algorithme du simplexe est un algorithme de résolution des problèmes d'optimisation linéaire.

6.4 Bilan

L'informatique haute performante nécessite des investissements importants. Les architectures sont de plus en plus complexes et les programmeurs ne peuvent pas être experts à la fois sur les nouvelles technologies, les compilateurs et les nouveaux langages. Or le succès d'une nouvelle architecture est souvent liée à sa facilité de prise en main, sa programmation et son compilateur. Pour exemple, le processeur CELL a été rapidement *boudé*, et abandonné par certains développeurs, car jugé trop complexe à programmer, le *Software Development kit* d'IBM ne facilitant pas suffisamment la programmation de cette architecture hétérogène. Des outils d'aide au développement des applications pour les nouvelles architectures sont donc nécessaires. Les techniques développées dans le cadre de mes travaux de recherche et ceux que j'envisage pour les années à venir restent dans cet objectif.

La plupart des *gamers* ne se doutent pas des efforts que certains chercheurs ont dû déployer pour qu'ils puissent s'installer tranquillement devant leur canapé et utiliser leur dernière console de jeux : des many-cœurs, des GPU ... L'arrivée de ces architectures hétérogènes, soulève les mêmes difficultés que celles rencontrées pour générer efficacement des codes pour des architectures à mémoires réparties. Toutefois chaque innovation soulève de nouveaux défis pour les compilateurs.

Chapitre 7

Liste de mes publications scientifiques

L'objectif de cette thèse d'habilitation à diriger des recherches était de donner un aperçu de mes travaux. A titre indicatif, la liste complète de mes publications est récapitulée dans le tableau ci-dessous et détaillée dans le même ordre pour chacune des catégories dans la bibliographie qui suit.

Type de publication	*Nombre*
Ouvrage et chapitre d'ouvrage	2
Articles de journaux et revues internationales avec comité de lecture	7
Conférences et workshops internationaux avec comité de lecture	19
Conférences internationales sans actes	3
Conférences nationales	3
Rapports de recherche (essentiellement liés aux contrats)	29
Tutoriel	1

Mon *h-index* est 12 et le nombre de citations de mes publications est 940 (référence *Google scholar*).

Bibliographie de Corinne Ancourt

[1] *Études et Recherches en Informatique, Algorithmique parallèle*, pages 261–269. Chapitre : Nids de boucles et machines à mémoire repartie. 1992.

[2] Mehdi Amini, Corinne Ancourt, Béatrice Creusillet, François Irigoin, and Ronan Keryell. *SAX-S57 : Patterns for parallel programming on GPU's*. Chapitre 6 : Program Sequentially, Carefully, and Benefit From Compiler Advances For Parallel Heterogeneous Computing. Saxe Coburg Publications, frederic magoules edition, 2014. ISSN 1759-3158, ISBN 978-1-874672-57-9.

[3] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. *SIGPLAN Notice*, 26(7) :39–50, April 1991.

[4] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations : Extended version. *International Journal of Parallel Programming*, 23(4) :359–388, August 1995.

[5] Fabien Coelho and Corinne Ancourt. Optimal compilation of HPF remappings. *Journal of Parallel and Distributed Computing*, 38(2) :229–236, November 1996.

[6] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A linear algebra framework for static High-Performance Fortran code distribution. *Scientific Programming*, 6(1) :3–27, January 1997.

[7] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electron. Notes Theor. Comput. Sci.*, 267(1) :3–16, October 2010.

[8] François Irigoin, Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, and Ronan Keryell. Polyèdres et compilation (version journal). *Technique et Science Informatiques (TSI)*, 31(8-9-10), 2012.

[9] Dounia Khaldi, Pierre Jouvelot, and Corinne Ancourt. Parallelizing with bdsc, a resource-constrained scheduling algorithm for shared and distributed memory systems. Technical report, Technical Report CRI/A-499 (Submitted to Parallel Computing), MINES ParisTech, 2012.

[10] Corinne Ancourt. Code generation for data movements in hierarchical memory machines. In *Int. Workshop on Compilers for Parallel Computers (CPC'90)*, pages 91–102, Paris, France, 1990.

[11] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'91, pages 39–50, Williamsburg, Virginia, USA, 1991. ACM.

[12] Corinne Ancourt and François Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers (CPC'92)*, Vienna, Austria, 1992. Citeseer.

[13] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A linear algebra framework for static HPF code distribution. In *International Workshop on Compilers for Parallel Computers (CPC'93)*, Delft, Pays-Bas, 1993.

[14] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC'94, pages 201–216, Ithaca, NY, USA, 1994. Springer-Verlag.

[15] Corinne Ancourt, Fabien Coelho, and Ronan Keryell. How to add a new phase in PIPS : the case of dead code elimination. In *Sixth International Workshop on Compilers for Parallel Computers (CPC'96)*, pages 19–30, Aachen, Germany, 1996.

[16] Corinne Ancourt, Denis Barthou, Christope. Guettier, François Irigoin, Bertrand Jeannet, Jean Jourdan, and Juliette Mattioli. Automatic data mapping of signal processing applications. In *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, ASAP'97, pages 350–, Swiss Federal Institute of Technology, Zurich, Switzerland, 1997. IEEE Computer Society.

[17] Thi Viet Nga Nguyen, François Irigoin, Corinne Ancourt, and Ronan Keryell. Efficient intraprocedural array bound checking. In *Second International Workshop on Automated Program Analysis, Testing and Verification, WAPATV*, volume 1, Toronto, Canada, 2000. Citeseer.

[18] Michel Barreteau, Juliette Mattioli, François Irigoin, Corinne Ancourt, Thierry Grandpierre, Christophe Lavarenne, Yves Sorel, Philippe Bonnot, Philippe Kajfasz, and Bernard Dion. PROMPT placement rapide optimisé sur machines parallèles pour applications Telecoms. In *Workshop AAA sur l'adéquation algorithme architecture*, pages 59–64, 2000.

[19] Michel Barreteau, Juliette Mattioli, François Irigoin, Corinne Ancourt, Thierry Grandpierre, Christophe Lavarenne, Yves Sorel, Philippe Bonnot, and Philippe Kajfasz. PROMPT : A Mapping Environment for Telecom Applications on System-On-a-Chip. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES'00, pages 41–47, San Jose, California, USA, 2000. ACM.

[20] Corinne Ancourt and Thi Viet Nga Nguyen. Array resizing for scientific code debugging, maintenance and reuse. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE'01, pages 32–37, Snowbird, Utah, USA, 2001. ACM.

[21] Thi Viet Nga Nguyen, François Irigoin, Corinne Ancourt, and Fabien Coelho. Automatic detection of uninitialized variables. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 217–231, Warsaw, Poland, 2003. Springer-Verlag.

[22] Isabelle Hurbain, Corinne Ancourt, François Irigoin, Michel Barreteau, Nicolas Museux, and Frederic Pasquier. A case study of design space exploration for embedded multimedia applications on SoCs. In *Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping*, RSP'06, pages 133–139, Chania, Crête, 2006. IEEE Computer Society.

[23] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. In *Numerical and Symbolic Abstract*

Domains (NSAD 2011), volume 267 of *NSAD 2011*, pages 3–16, Perpignan, France, 2010. Elsevier.

[24] Fabien Coelho, Pierre Jouvelot, Corinne Ancourt, and François Irigoin. Data and process abstraction in PIPS internal representation. In *First Workshop on Intermediate Representations (WIR-1)*, Chamonix, France, 2011.

[25] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. PIPS Is not (just) Polyhedral Software, Adding GPU Code Generation in PIPS. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO*, Chamonix, France, 2011.

[26] Dounia Khaldi, Pierre Jouvelot, Corinne Ancourt, and François Irigoin. Task Parallelism and Data Distribution : An Overview of Explicit Parallel Programming Languages. In *Languages and Compilers for Parallel Computing*, volume 7760 of *LCPC 2012*, pages 174–189. Springer Berlin Heidelberg, Waseda University, Tokyo, Japan, 2012.

[27] Dounia Khaldi, Pierre Jouvelot, François Irigoin, and Corinne Ancourt. SPIRE : A methodology for sequential to parallel intermediate representation extension. In *17th Workshop on Compilers for Parallel Computing (CPC 2013)*, Lyon, France, 2013.

[28] Corinne Ancourt, Teodora Petrisor, François Irigoin, and Eric Lenormand. Automatic generation of communications for redundant multi-dimensional data parallel redistributions. In *IEEE International Conference on High-Performance Computing and Communications*, HPCC 2013, pages 800–811, Zhangjiajie, Chine, 2013.

[29] Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, and Ronan Keryell. PIPS : a workbench for program parallelization and optimization. *European Parallel Tool Meeting'96*, 1996.

[30] Corinne Ancourt, François Irigoin, Teodora Petrisor, and Eric Lenormand. A complete industrial multi-target graphical tool-chain for parallel implementations of signal/image applications. In *Workshop on Mapping of Applications to MPSoCs*, Rheinfels Castle, St. Goar, Germany, 2011.

[31] Dounia Khaldi, Pierre Jouvelot, François Irigoin, and Corinne Ancourt. SPIRE : A methodology for sequential to parallel intermediate representation extension. In *HiPEAC Computing Systems Week*, Paris, France, 2013.

[32] Corinne Ancourt. Génération de code pour multiprocesseur à mémoires locales. In *Réunions C^3*, pages 91–102, Rennes, July 1990.

[33] Fabien Coelho, Pierre Jouvelot, Corinne Ancourt, and François Irigoin. Data and process abstraction in PIPS internal representation. Troisièmes Rencontres de la Communauté Française de Compilation, Manoir de la Vicomte, Dinard, France, 2011.

[34] François Irigoin, Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, and Ronan Keryell. Polyèdres et compilation. In *Rencontres francophones du Parallélisme (RenPar'20)*, Saint-Malo, France, 2011.

[35] Corinne Ancourt. Utilisation de systèmes linéaires sur $\mathcal{Z}$ pour la parallélisation de programmes. Master's thesis, École Nationale Supérieure des Mines de Paris, Fontainebleau, July 1987.

[36] Martine-Corinne Ancourt. *Génération Automatique de Code de Transfert pour Multiprocesseurs à Mémoires Locales.* PhD thesis, Université Paris VI, March 1991.

[37] François Irigoin and Corinne Ancourt. Software caching for simulated global memory. Final Project Report -WP65 No 155, MINES ParisTech, 1991.

[38] Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, and Ronan Keryell. PIPS development environment. Technical Report No 291, MINES ParisTech, 1996.

[39] Corinne Ancourt, Denis Barthou, Christophe Guettier, and François Irigoin. Compilation of VBL specifications. Contrat THOMSON MARCONI SONAR No 194, MINES ParisTech, 1996.

[40] François Irigoin, Denis Barthou, and Corinne Ancourt. ARRAY-OL et la parallélisation automatique. Contrat THOMSON MARCONI SONAR No 204, MINES ParisTech, 1996.

[41] Corinne Ancourt, Denis Barthou, François Irigoin, Juliette Mattioli, Bertrand Jeannet, Jean Jourdan, and Christophe Guettier. Formalisation et modélisation exécutable du placement automatique. Contrat THOMSON MARCONI SONAR No 205, MINES ParisTech, 1996.

[42] Corinne Ancourt, Denis Barthou, and François Irigoin. D'Astéride à l'accélérateur synchrone. Contrat THOMSON MARCONI SONAR No 206, MINES ParisTech, 1996.

[43] Corinne Ancourt, Denis Barthou, Christophe Guettier, and François Irigoin. Modèles pour le placement ARRAY-OL. Contrat THOMSON MARCONI SONAR No 207, MINES ParisTech, 1996.

[44] Corinne Ancourt. Étude de la génération des communications pour le PSP/RBE2. Contrat THOMSON-CSF No 217, MINES ParisTech, 1998.

[45] Corinne Ancourt and François Irigoin. Applications temps réel, architecture cible et placement générique : d'APOTRES à EPHORAT. Contrat PROMPT No 225, MINES ParisTech, 2000.

[46] Corinne Ancourt and François Irigoin. Entrées/sorties. Contrat PROMPT No 234, MINES ParisTech, 2000.

[47] Corinne Ancourt and François Irigoin. Mémoires multiples et hiérarchie mémoire. Contrat PROMPT No 233, MINES ParisTech, 2000.

[48] Corinne Ancourt, Michel Barreteau, Bernard Dion, François Irigoin, Philippe Kajfasz, Christophe Lavarenne, Juliette Mattioli, Nicolas Museux, and Yves Sorel. Étude de la fonction placement sur les parties homogènes des SOC. Contrat PROMPT, MINES ParisTech, 2000.

[49] Corinne Ancourt, Michel Barreteau, Bernard Dion, François Irigoin, Philippe Kajfasz, Christophe Lavarenne, Juliette Mattioli, and Yves Sorel. Modélisation de la partie homogène de l'architecture des SOC. Contrat PROMPT, MINES ParisTech, 2000.

[50] Corinne Ancourt, Michel Barreteau, Bernard Dion, François Irigoin, Philippe Kajfasz, Christophe Lavarenne, Juliette Mattioli, and Yves Sorel. Modélisation des applications de traitement du signal systématique Télécom. Contrat PROMPT, MINES ParisTech, 2000.

[51] François Irigoin, Corinne Ancourt, Béatrice Creusillet, and Nga Nguyen. Interprocedural analyses and compilers. Technical Report No 245, MINES ParisTech, 2001.

[52] Corinne Ancourt. Analyse de l'application A1. Contrat INDUSTRIEL No 219, MINES ParisTech, 1998.

[53] Corinne Ancourt. Analyse de l'application A2. Contrat INDUSTRIEL No 248, MINES ParisTech, 2001.

[54] Corinne Ancourt, Coelho Fabien, and François Irigoin. Vérification de la parallélisation de l'application A3. Contrat INDUSTRIEL No 251, MINES ParisTech, 2001.

[55] Corinne Ancourt. Analyse de l'application A3. Contrat INDUSTRIEL No 254, MINES ParisTech, 2002.

[56] Corinne Ancourt and Fabien Coelho. Étude de l'application A4. Optimisation de l'application. Contrat INDUSTRIEL, MINES ParisTech, 2002.

[57] Corinne Ancourt, Isabelle Hurbain, François Irigoin, and Nicolas Museux. Ensemble et polytope de dépendance, système générateur et ordonnancement. Contrat DREAM-UP No 258, MINES ParisTech, 2004.

[58] Corinne Ancourt, François Irigoin, and Nicolas Museux. Modèles de mémoire. Contrat DREAM-UP No 265, MINES ParisTech, 2005.

[59] Corinne Ancourt, François Irigoin, and Nicolas Museux. Modèles de partitionnement. Contrat DREAM-UP No 268, MINES ParisTech, 2005.

[60] Corinne Ancourt and François Irigoin. Modélisation des fonctions de bibliothèques, des applications et de leurs interfaces. Contrat TERAOPS No 301, MINES ParisTech, 2008.

[61] Dounia Khaldi, Corinne Ancourt, and François Irigoin. Towards automatic C programs optimization and parallelization using the PIPS-POCC integration. Contrat OPENGPU No A/448, MINES ParisTech, 2011.

[62] Corinne Ancourt, François Irigoin, Ronan Keryell, and Béatrice Creusillet. PAR4ALL & PIPS programming rules. Contrat OPENGPU No 315, MINES ParisTech, 2012.

[63] Mehdi Amini, Ronan Keryell, Beatrice Creusillet, Corinne Ancourt, and François Irigoin. Program sequentially, carefully, and benefit from compiler advances for parallel heterogeneous computing. Confidentiel No 320, MINES ParisTech, 2012.

[64] Laurent Daverio, Corinne Ancourt, Fabien Coelho, Stéphanie Even, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Frédérique Silber-Chaussumier. PIPS – An Interprocedural, Extensible, Source-to-Source Compiler Infrastructure for Code Transformations and Instrumentations. Tutorial at PPoPP, Bengalore, India, January 2010 ; Tutorial at CGO, Chamonix, France, April 2011. `http://pips4u.org/doc/tutorial/tutorial-no-animations.pdf` presented by Corinne Ancourt, François Irigoin, Serge Guelton, Ronan Keryell et Frédérique Silber-Chaussumier.

La majorité des rapports de recherche sont confidentiels, car établis dans le cadre de contrats avec des industriels et soumis à NDA.

Bibliographie Générale

[1] *History of the theory of numbers*. Washington, Canergie institution of Washington, 1874.

[2] J. Allen, F. E. anf Cocke. A catalogue of optimizing transformations. *Design and Optimization of Compilers*, pages 1–30, 1971. R. Rustin, Prentice-Hall.

[3] J. R. Allen and K. Kennedy. Automatic loop interchange. In *SIGPLAN'84 Symposium on Compiler Construction*, volume 19 of *SIGPLAN Notices*, 1984.

[4] R Allen and K Kennedy. Automatic translation of fortran programs to vector form. In *ACM Transactions on Programming Languages and Systems*, Oct. 1987.

[5] Alt-Ergo. The alt-ergo, 2013.

[6] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. *SIGPLAN Not.*, 28(6) :126–138, June 1993.

[7] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. PIPS Is not (just) Polyhedral Software, Adding GPU Code Generation in PIPS. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011) in conjonction with CGO*, Chamonix, France, 2011.

[8] Mehdi Amini, Corinne Ancourt, Béatrice Creusillet, François Irigoin, and Ronan Keryell. *SAX-S57 : Patterns for parallel programming on GPU's*. Chapitre 6 : Program Sequentially, Carefully, and Benefit From Compiler Advances For Parallel Heterogeneous Computing. Saxe Coburg Publications, frederic magoules edition, 2014. ISSN 1759-3158, ISBN 978-1-874672-57-9.

[9] Corinne Ancourt. Code generation for data movements in hierarchical memory machines. In *Int. Workshop on Compilers for Parallel Computers (CPC'90)*, pages 91–102, Paris, France, 1990.

[10] Corinne Ancourt. Génération de code pour multiprocesseur à mémoires locales. In *Réunions C^3*, pages 91–102, Rennes, July 1990.

[11] Corinne Ancourt. Analyse de l'application A1. Contrat INDUSTRIEL No 219, MINES ParisTech, 1998.

[12] Corinne Ancourt. Étude de la génération des communications pour le PSP/RBE2. Contrat THOMSON-CSF No 217, MINES ParisTech, 1998.

[13] Corinne Ancourt. Analyse de l'application A2. Contrat INDUSTRIEL No 248, MINES ParisTech, 2001.

[14] Corinne Ancourt. Analyse de l'application A3. Contrat INDUSTRIEL No 254, MINES ParisTech, 2002.

[15] Corinne Ancourt, Michel Barreteau, Bernard Dion, François Irigoin, Philippe Kajfasz, Christophe Lavarenne, Juliette Mattioli, Nicolas Museux, and Yves Sorel. Étude de la fonction placement sur les parties homogènes des SOC. Contrat PROMPT, MINES ParisTech, 2000.

[16] Corinne Ancourt, Michel Barreteau, Bernard Dion, François Irigoin, Philippe Kajfasz, Christophe Lavarenne, Juliette Mattioli, and Yves Sorel. Modélisation de la partie homogène de l'architecture des SOC. Contrat PROMPT, MINES ParisTech, 2000.

[17] Corinne Ancourt, Michel Barreteau, Bernard Dion, François Irigoin, Philippe Kajfasz, Christophe Lavarenne, Juliette Mattioli, and Yves Sorel. Modélisation des applications de traitement du signal systématique Télécom. Contrat PROMPT, MINES ParisTech, 2000.

[18] Corinne Ancourt, Denis Barthou, Christophe Guettier, and François Irigoin. Compilation of VBL specifications. Contrat THOMSON MARCONI SONAR No 194, MINES ParisTech, 1996.

[19] Corinne Ancourt, Denis Barthou, Christophe Guettier, and François Irigoin. Modèles pour le placement ARRAY-OL. Contrat THOMSON MARCONI SONAR No 207, MINES ParisTech, 1996.

[20] Corinne Ancourt, Denis Barthou, and François Irigoin. D'Astéride à l'accélérateur synchrone. Contrat THOMSON MARCONI SONAR No 206, MINES ParisTech, 1996.

[21] Corinne Ancourt, Denis Barthou, François Irigoin, Juliette Mattioli, Bertrand Jeannet, Jean Jourdan, and Christophe Guettier. Formalisation et modélisation exécutable du placement automatique. Contrat THOMSON MARCONI SONAR No 205, MINES ParisTech, 1996.

[22] Corinne Ancourt and Fabien Coelho. Étude de l'application A4. Optimisation de l'application. Contrat INDUSTRIEL, MINES ParisTech, 2002.

[23] Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, and Ronan Keryell. PIPS : a workbench for program parallelization and optimization. In *European Parallel Tool Meeting 1996 and EPTM'96*. Onera, France, October 1996.

[24] Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, and Ronan Keryell. PIPS development environment. Technical Report No 291, MINES ParisTech, 1996.

[25] Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, and Ronan Keryell. How to add a new phase in PIPS : the case of dead code elimination. In *Sixth Workshop on Compilers for Parallel Computers, CPC'96*, pages pp 19–30, Aachen, Germany, Décembre 1996.

[26] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. *Electron. Notes Theor. Comput. Sci.*, 267(1) :3–16, October 2010.

[27] Corinne Ancourt, Fabien Coelho, and François Irigoin. A modular static analysis approach to affine loop invariants detection. In *Numerical and Symbolic Abstract Domains (NSAD 2011)*, volume 267 of *NSAD 2011*, pages 3–16, Perpignan, France, 2010. Elsevier.

[28] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A linear algebra framework for static High-Performance Fortran code distribution. *Scientific Programming*, 6(1) :3–27, January 1997.

[29] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A linear algebra framework for static HPF code distribution. In *Fourth Workshop on Compilers for Parallel Computers, CPC'93*, Delft, Pays-Bas, Novembre 1993.

[30] Corinne Ancourt, Coelho Fabien, and François Irigoin. Vérification de la parallélisation de l'application A3. Contrat INDUSTRIEL No 251, MINES ParisTech, 2001.

[31] Corinne Ancourt, Isabelle Hurbain, François Irigoin, and Nicolas Museux. Ensemble et polytope de dépendance, système générateur et ordonnancement. Contrat DREAM-UP No 258, MINES ParisTech, 2004.

[32] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. *SIGPLAN Notice*, 26(7) :39–50, April 1991.

[33] Corinne Ancourt and François Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers (CPC'92)*, Vienna, Austria, 1992. Citeseer.

[34] Corinne Ancourt and François Irigoin. Applications temps réel, architecture cible et placement générique : d'APOTRES à EPHORAT. Contrat PROMPT No 225, MINES ParisTech, 2000.

[35] Corinne Ancourt and François Irigoin. Entrées/sorties. Contrat PROMPT No 234, MINES ParisTech, 2000.

[36] Corinne Ancourt and François Irigoin. Mémoires multiples et hiérarchie mémoire. Contrat PROMPT No 233, MINES ParisTech, 2000.

[37] Corinne Ancourt and François Irigoin. Modélisation des fonctions de bibliothèques, des applications et de leurs interfaces. Contrat TERAOPS No 301, MINES ParisTech, 2008.

[38] Corinne Ancourt, François Irigoin, Ronan Keryell, and Béatrice Creusillet. PAR4ALL & PIPS programming rules. Contrat OPENGPU No 315, MINES ParisTech, 2012.

[39] Corinne Ancourt, François Irigoin, and Nicolas Museux. Modèles de mémoire. Contrat DREAM-UP No 265, MINES ParisTech, 2005.

[40] Corinne Ancourt, François Irigoin, and Nicolas Museux. Modèles de partitionnement. Contrat DREAM-UP No 268, MINES ParisTech, 2005.

[41] Corinne Ancourt and François Irigoin. Scanning polyhedra with DO loops. *ACM Sigplan Notices*, 26(7) :39–50, 1991.

[42] Corinne Ancourt, François Irigoin, Teodora Petrisor, and Eric Lenormand. A complete industrial multi-target graphical tool-chain for parallel implementations of signal/image applications. In *Workshop on Mapping of Applications to MPSoCs*, Rheinfels Castle, St. Goar, Germany, 2011.

[43] Corinne Ancourt and Thi Viet Nga Nguyen. Array resizing for scientific code debugging, maintenance and reuse. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE'01, pages 32–37, Snowbird, Utah, USA, 2001. ACM.

[44] Corinne Ancourt, Teodora Petrisor, François Irigoin, and Eric Lenormand. Automatic generation of communications for redundant multi-dimensional data parallel redistributions. In *IEEE International Conference on High-Performance Computing and Communications*, HPCC 2013, pages 800–811, Zhangjiajie, Chine, 2013.

[45] Corinne Ancourt and Anh Trinh Quoc. Restructuration et normalisation de programmes fortran. Confidentiel No 240, MINES ParisTech, 2000.

[46] Martine-Corinne Ancourt. *Génération Automatique de Code de Transfert pour Multiprocesseurs à Mémoires Locales.* PhD thesis, Université Paris VI, March 1991.

[47] APRON. Numerical program analysis. http ://www.cri.ensmp.fr/apron, 2005. Project APRON website.

[48] U. Banerjee. A theory of loop permutations. In *2nd Workshop on Languages and compilers for parallel computing*, 1989.

[49] Michel Barreteau, Juliette Mattioli, François Irigoin, Corinne Ancourt, Thierry Grandpierre, Christophe Lavarenne, Yves Sorel, Philippe Bonnot, and Philippe Kajfasz. PROMPT : A Mapping Environment for Telecom Applications on System-On-a-Chip. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES'00, pages 41–47, San Jose, California, USA, 2000. ACM.

[50] Michel Barreteau, Juliette Mattioli, François Irigoin, Corinne Ancourt, Thierry Grandpierre, Christophe Lavarenne, Yves Sorel, Philippe Bonnot, Philippe Kajfasz, and Bernard Dion. PROMPT placement rapide optimisé sur machines parallèles pour applications Telecoms. In *Workshop AAA sur l'adéquation algorithme architecture*, pages 59–64, 2000.

[51] Denis Barthou and François Irigoin. Méthodes de génération de code à partir d'un ordonnancement. Technical Report No 208, MINES ParisTech, 1996.

[52] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *In IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'04*, pages 7–16, 2004.

[53] Molka Becher. Modélisation et estimation des temps d'exécution des instructions d'un programme c. Technical report, MINES ParisTech, 2011.

[54] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*. ACM, 2003.

[55] F. Bodin, L. Kervella, and T. Priol. Fortran-s : A fortran interface for shared virtual memory architectures. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 274–283, New York, NY, USA, 1993. ACM.

[56] James Bond. Calculating the general solution of a linear diophantine equation. In *American Math Monthly*, volume 74. 1967.

[57] Pierre Boulet and Paul Feautrier. Scanning polyhedra without do-loops. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 4–, Washington, DC, USA, 1998. IEEE Computer Society.

[58] Yves Caseau, François-Xavier Josset, and François Laburthe. Claire : Combining sets, search and rules to better express algorithms. In *Theory and Practice of Logic Programming*, volume 2, 2002.

[59] Yves Caseau, François-Xavier Josset, François Laburthe, B. Rottembourg, S. de Givry, Jean Jourdan, Juliette Mattioli, and P. Savéat. Eclair at a glance. Technical report, Tsi/99-876, Thomson-CSF/LCR, 1999.

[60] Yves Caseau and François Laburthe. Introduction to the claire programming language. In *LIENS, report Paris*, 1996.

[61] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java : The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ11, pages 51–61, New York, NY, USA, 2011. ACM.

[62] Ernesto Cesaro. Sur diverses questions d'arithmetique, 1883.

[63] Andrew Chien and Vijak Karamcheti. Moorés law : the first ending and a new beginning. In *Computer*, volume 46. december 2013.

[64] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[65] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages : Co-array fortran and unified parallel c. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[66] Fabien Coelho and Corinne Ancourt. Optimal compilation of HPF remappings. *Journal of Parallel and Distributed Computing*, 38(2) :229–236, November 1996.

[67] Jean-Francois Collard. Code generation in automatic parallelizers. In Claude Girault, editor, *Applications in Parallel and Distributed Computing*, volume A-44 of *IFIP Transactions*, pages 185–194. North-Holland, 1994.

[68] Jean-Francois Collard. Code generation in automatic parallelizers. In *Proceedings of the IFIP WG10.3 Working Conference on Applications in Parallel and Distributed Computing*, pages 185–194, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[69] Jean-Francois Collard, Tanguy Risset, and Paul Feautrier. Construction of do loops from systems of affine constraints. *Parallel Processing Letters*, 5 :421–436, 1995.

[70] Coq. The coq proof assistant reference manual, 2009.

[71] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, pages 106–130, April 1976.

[72] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, January 1978.

[73] Béatrice Creusillet. IN and OUT array region analyses. In *CPC*, pages 233–246, June 1995.

[74] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. In *LCPC*, volume 1033 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 1995.

[75] Béatrice Creusillet-Apvrille. *Analyses de régions de tableaux et applications*. PhD thesis, École des mines de Paris, December 1996. (in English).

[76] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c : A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[77] Alain Darte. *Techniques de parallélisation automatique de nids de boucles*. PhD thesis, Ecole Normale de Lyon, 1993.

[78] Laurent Daverio, Corinne Ancourt, Fabien Coelho, Stéphanie Even, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Frédérique Silber-Chaussumier. PIPS an interprodedural, extensible, source-to-source compiler infrastructure for code transformations and instrumentations. Chamonix, France, 2011.

[79] David Delahaye, Claude Marché, and David Mentré. Le projet BWare : une plateforme pour la vérification automatique d'obligations de preuve B. In *Approches Formelles dans l'Assistance au Développement de Logiciels*, Paris, France, June 2014. EasyChair.

[80] Alain Deutsch. Polyspace. http ://fr.mathworks.com/products/polyspace/.

[81] Romain Dolbeau, Stéphane Bihan, François Bodin, and Caps Entreprise. 1 hmpp^{TM} : A hybrid multi-core parallel programming environment.

[82] Antoine Durand-Gasselin. *Automata Based Logics for Program Verification*. PhD thesis, Paris 7, 2013.

[83] Paul Feautrier. PIPlib. http ://www.PipLib.org/. Open source, under GPLv2.1.

[84] Paul Feautrier. Fine-grain scheduling under resource constraints. In *In Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, pages 1–15. Springer-Verlag, 1994.

[85] Paul Feautrier. The Power of Polynomials . In *5th International Workshop on Polyhedral Comilation Techniques*, Amsterdam, Netherlands, January 2015. Alexandra Jimborean, Alain Darte.

[86] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran d language specification. Technical report, 1990.

[87] Rahul Garg and Laurie Hendren. Velociraptor : An embedded compiler toolkit for numerical programs targeting cpus and gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 317–330, New York, NY, USA, 2014. ACM.

[88] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT'91 : Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 1 : Colloquium on Trees in Algebra and Programming (CAAP'91)*, pages 169–192, 1991.

[89] Christophe Guettier. *Optimisation globale du placement d'application de traitement du signal sur architectures parallèles utilisant la programmation logique avec contraintes*. PhD thesis, Paris, École Nationale Supérieure des Mines de Paris, 1997. Th. : informatique, temps réel robotique, automatique.

[90] Rachid Habel. *Programmation haute performance pour architectures hybrides*. PhD thesis, MINES ParisTech, november 2014.

[91] Nicolas Halbwachs. *Détermination automatique de relations linéares verifiées par les variables d'un programe*. PhD thesis, Universite Scientifique et Medicale de Grenoble, France, Mars 1979. Thesis.

[92] John L. Hennessy and David A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[93] Isabelle Hurbain, Corinne Ancourt, François Irigoin, Michel Barreteau, Nicolas Museux, and Frederic Pasquier. A case study of design space exploration for embedded multimedia applications on SoCs. In *Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping*, RSP'06, pages 133–139, Chania, Crête, 2006. IEEE Computer Society.

[94] Cray Inc. Chapel language specification 0.796. http ://chapel.cray.com/spec/spec-0.796.pdf, 2010.

[95] François Irigoin and Corinne Ancourt. Software caching for simulated global memory. Final Project Report -WP65 No 155, MINES ParisTech, 1991.

[96] François Irigoin, Denis Barthou, and Corinne Ancourt. ARRAY-OL et la parallélisation automatique. Contrat THOMSON MARCONI SONAR No 204, MINES ParisTech, 1996.

[97] François Irigoin, Piere Jouvelot, and Rémi Triolet. Overview of the PIPS project. International Workshop on Compilers for Parallel Computers, 1990.

[98] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization : an overview of the PIPS project. In *ICS*, pages 144–151, June 1991.

[99] François Irigoin, Pierre Jouvelot, and Remi Triolet. Author retrospective for semantical interprocedural parallelization : An overview of the PIPS project. In *25th Anniversary International Conference on Supercomputing Anniversary Volume*, pages 12–14, New York, NY, USA, 2014. ACM.

[100] François Irigoin and Rémi Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical report, Ecole des Mines de Paris, 1987.

[101] François Irigoin, Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, and Ronan Keryell. Polyèdres et compilation. In *Rencontres francophones du Parallélisme (RenPar'20)*, Saint-Malo, France, 2011.

[102] François Irigoin, Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, and Ronan Keryell. Polyèdres et compilation (version journal). *Technique et Science Informatiques (TSI)*, 31(8-9-10), 2012.

[103] François Irigoin. *Partitionnement des boucles imbriquees : une technique d'optimisation pour les programmes scientifiques*. 1987.

[104] François Irigoin and Corinne Ancourt. *Études et Recherches en Informatique, Algorithmique parallèle*, chapter Compilation pour Machines à Mémoire Répartie, pages pp.261–269. Masson, May 1992.

[105] François Irigoin and Corinne Ancourt. Nids de boucles et machines à mémoire répartie. In *20ème École de Printemps du LITP*, May 1992.

[106] François Irigoin and Rémi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[107] IRISA. Polylib. http ://www.irisa.fr/polylib/. Open source, under GPLv3.

[108] Isabelle. Isabelle.

[109] Bertrand Jeannet. *Partitionnement Dynamique dans l'Analyse de Relations Linéaires et Application à la Vérification de Programmes Synchrones*. PhD thesis, Institut National Polytechnique de Grenoble, September 2000.

[110] Prachi Kalra. Rapport de stage : Static estimation of execution times. Technical report, Mines ParisTech, 2007.

[111] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurence equations. In *Journal of the ACM*, volume 14, pages 563–590, 1967.

[112] W. Kelly, William Pugh, and E. Rosser. Code generation for multiple mappings. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers '95., Fifth Symposium on the*, pages 332 – 341. IEEE, IEEE, 1995/02/06/9 1995.

[113] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. PIPS : A workbench for building interprocedural parallelizers, compilers and optimizers. Technical Report No 289, MINES ParisTech, 1996.

[114] Dounia Khaldi, Corinne Ancourt, and François Irigoin. Towards automatic C programs optimization and parallelization using the PIPS-POCC integration. Contrat OPENGPU No A/448, MINES ParisTech, 2011.

[115] Dounia Khaldi, Pierre Jouvelot, and Corinne Ancourt. Parallelizing with BDSC, a resource-constrained scheduling algorithm for shared and distributed memory systems. *Parallel Computing*, 41 :66 – 89, January 2015.

[116] Dounia Khaldi, Pierre Jouvelot, Corinne Ancourt, and François Irigoin. Task Parallelism and Data Distribution : An Overview of Explicit Parallel Programming Languages. In *Languages and Compilers for Parallel Computing*, volume 7760 of *LCPC 2012*, pages 174–189. Springer Berlin Heidelberg, Waseda University, Tokyo, Japan, 2012.

[117] Dounia Khaldi, Pierre Jouvelot, François Irigoin, and Corinne Ancourt. SPIRE : A methodology for sequential to parallel intermediate representation extension. In *HiPEAC Computing Systems Week*, Paris, France, 2013.

[118] Leslie Lamport. The parallel execution of do loops. In *Communications of the ACM*, volume 17, pages 83–93, 1974.

[119] Arnault Leservot. *Analyse interprocédurale du flot des données*. PhD thesis, Université PARIS VI, 1996.

[120] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas W. Reps. Debugging via run-time type checking. In *FASE*, pages 217–232, April 2001.

[121] François Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *ICS*, pages 226–235, 1992.

[122] J. Mattioli, N. Museux, J. Jourdan, and S. De Givry. Givry. a constraint optimization framework for mapping a digital signal processing application onto a parallel architecture. In *In Principles and Practice of Constraint Programming*, 2000.

[123] John Mellor-Crummey, Vikram Adve, V. Adve, Bradley Broom, Daniel Chavarrla-Miranda, Guohua Jin, Robert Fowler, Qing Yi, G. Jin, K. Kennedy, and Q. Yi. Advanced optimization strategies in the rice dhpf compiler, 2001.

[124] Hugo Metivier. Représentation d'ensembles de valeurs numériques en vérification de programmes. Master's thesis, IRISA,, 2005.

[125] Bertrand Meyer. *Conception et programmation orientées objet*. Eyrolles, 2000.

[126] Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. http ://www.di.ens.fr/~mine/publi/article-mine-ast01.pdf.

[127] Antoine Miné. A few graph-based relational numerical abstract domains. In *SAS'02*, volume 2477 of *LNCS*, pages 117–132. Springer-Verlag, 2002. http ://www.di.ens.fr/~mine/publi/article-mine-sas02.pdf.

[128] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004. http ://www.di.ens.fr/~mine/publi/article-mine-esop04.pdf.

[129] MINES-ParisTech. LinearC3 - PIPS project. http ://pips4u.org, 1988. Open source, under LGPL.

[130] MINES-ParisTech. PIPS. http ://pips4u.org, 1989–2009. Open source, under GPLv3.

[131] David Monniaux. A quantifier elimination algorithm for linear real arithmetic. In *In LPAR (Logic for Programming, Artificial Intelligence, and Reasoning), LNCS*, page 2008. Springer.

[132] Antoine Morvan, Steven Derrien, and Patrice Quinton. Polyhedral bubble insertion : A method to improve nested loop pipelining for high-level synthesis. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 32(3) :339–352, 2013.

[133] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-champaign, Febr. 1971.

[134] Nicolas Museux. *Aide au placement d'applications de traitement du signal sur machines parallèles multi-SPMD : rencontre de la parallélisation automatique et de la programmation par contraintes*. PhD thesis, Paris, École Nationale Supérieure des Mines de Paris, 2001. Th. : informatique temps réel, robotique, automatique.

[135] Thi Viet Nga Nguyen. *Vérifications efficaces des applications scientifiques par analyse statique et instrumentation de code. Efficient and effective software verifications for scientific applications using static analysis and code instrumentation.* PhD thesis, MINES ParisTech, november 2002.

[136] Thi Viet Nga Nguyen and François Irigoin. Efficient and effective array bound checking. *ACM Transactions On Programming Languages and Systems*, 27(3) :527–570, May 2005.

[137] Thi Viet Nga Nguyen, François Irigoin, Corinne Ancourt, and Fabien Coelho. Automatic detection of uninitialized variables. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 217–231, Warsaw, Poland, 2003. Springer-Verlag.

[138] Thi Viet Nga Nguyen, François Irigoin, Corinne Ancourt, and Ronan Keryell. Efficient intraprocedural array bound checking. In *Second International Workshop on Automated Program Analysis, Testing and Verification, WAPATV*, volume 1, Toronto, Canada, 2000. Citeseer.

[139] Thi Viet Nga Nguyen and François Irigoin. Alias verification for fortran code optimization, 2002.

[140] Duong Nguyen Que. *Robust and Generic Abstract Domain for Static Program Analyses : The Polyhedral Case.* PhD thesis, École des Mines de Paris, July 2013.

[141] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2) :1–31, August 1998.

[142] OpenACC Consortium. OpenACC API. http ://www.nvidia.com/docs/IO/116711/OpenACC-API.pdf.

[143] OpenMP Consortium. OpenMP. http ://openmp.org/wp/2013/07/openmp-40/.

[144] OpenMP Consortium. OpenMP Compilers. http ://openmp.org/wp/openmp-compilers/.

[145] Derek C Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of presburger arithmetic. *Journal of Computer and System Sciences*, 16(3) :323–332, 1978.

[146] Quoc Dat Pham. Maintenance et mise au point de programmes : utilisation du graphe de dépendance et de l'information sur les alias. Confidentiel No 253, MINES ParisTech, 2003.

[147] M. Presburger. Uber die vollstandigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes rendus du premier congrès des Mathématiques des Pays Slaves*, pages p 92–101, 1929.

[148] William Pugh. The omega test : a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[149] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8) :102–114, 1992.

[150] Fabien Quillere, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28 :2000, 2000.

[151] Patrice Quinton, Sanjay Rajopadhye, and Tanguy Risset. On manipulating z-polyhedra. Technical report, 1996.

[152] Texas Rice University, Houston. High Performance Fortran specification, May 1993.

[153] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. http ://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf, 2013.

[154] Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical report, 1990.

[155] Alexander Schrijver. *Theory of linear and integer Programming.* WI, New York, 1986. Book.

[156] T. R. Shiple, J. H. Kukula, and R. K. Ranjan. A comparison of presburger engines for efsm reachability. volume 1427, pages p 280–292., 1998.

[157] Jean-Claude Sogno. http ://raweb.inria.fr/rapportsactivite/RA2002/contraintes/uid18.html.

[158] Jean-Claude Sogno. The janus test : a hierarchical algorithm for computing direction and distance vectors (extended version)*. *Parallel Algorithms Appl.*, 12(1-3) :57–82, 1997.

[159] John E. Stone, David Gohara, and Guochun Shi. Opencl : A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3) :66–73, May 2010.

[160] Ernesto Su, Antonio Lain, Shankar Ramaswamy, Daniel J. Palermo, Eugene W. Hodges, IV, and Prithviraj Banerjee. Advanced compilation techniques in the paradigm compiler for distributed-memory multicomputers. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 424–433, New York, NY, USA, 1995. ACM.

[161] ASTRÉE team. Analyseur statique de logiciels temps-reel embarques. http ://www.astree.ens.fr, 2002. Project ASTRÉE website.

[162] CHINA team. A data-flow analyzer for clp languages. http ://www.cs.unipr.it/China, 2002. Project website.

[163] NBAC team. Nbac. http ://www.irisa.fr/prive/bjeannet/nbac/nbac.html, 2002. Project NBAC website.

[164] Omega team. Omega. http ://www.cs.umd.edu/projects/omega, 2002. Project website.

[165] PPL team. Parma polyhedral library. http ://www.cs.unipr.it/ppl, 2002. Project website.

[166] A. E. Terrano. Optimal tiling for iterative pde solvers. In *Frontiers of Massively Parallel Computation*, 1988.

[167] Rémi Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de Procédures*. PhD thesis, Université Paris VI, 1984.

[168] Consortium UPC. UPC. http ://upc.gwu.edu.

[169] Clark Verbrugge, Phong Co, and Laurie Hendren. Generalized constant propagation a study in c. In *In 6th Int. Conf. on Compiler Construction, volume 1060 of Lec. Notes in Comp. Sci*, pages 74–90. Springer, 1996. http ://citeseer.ist.psu.edu/verbrugge96generalized.html.

[170] Michael Weiss. Strip mining on simd architectures. In *Proceedings of the 5th International Conference on Supercomputing*, ICS '91, pages 234–243, New York, NY, USA, 1991. ACM.

[171] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Oct. 1982.

[172] Michael Wolfe. Loop skewing : The wavefront method revisited. *Int. J. Parallel Program.*, 15(4) :279–293, October 1986.

[173] M.J. Wolfe. *Techniques for Improving the Inherent Parallelism in Programs*. UILU-ENG / 17 : UILU-ENG. Department of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[174] Yi Qing Yang. *Tests de Dependance et Transformations de programme*. PhD thesis, Université Pierre et Marie Curie, Nov. 1993.

[175] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC'94, pages 201–216, Ithaca, NY, USA, 1994. Springer-Verlag.

[176] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations : Extended version. *International Journal of Parallel Programming*, 23(4) :359–388, August 1995.

[177] Ding Ye, Yulei Sui, and Jingling Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 154 :154–154 :164, New York, NY, USA, 2014. ACM.

[178] Lei Zhou. *Statical and dynamical analysis of program complexity.* PhD thesis, Paris, École Nationale Supérieure des Mines de Paris, 1994.

Table des figures

1.1 Coûts logiciel vs coûts matériel 8

2.1 Le compilateur PIPS 17
2.2 Exemple de polyèdre 19
2.3 Exemple de $\mathcal{Z}$-polyèdre 20
2.4 Programme P et son système de dépendances. 24
2.5 Abstractions DI, D et DP du nid de boucles du programme P 25
2.6 Abstractions DC, DDV et DL du nid de boucles du programme P 25
2.7 Code initial 27
2.8 Régions Read-Write 29
2.9 Régions IN 30
2.10 Régions OUT 31

3.1 Dépendances de données 44
3.2 Clones 44
3.3 Contrôle 45

4.1 Programme 49
4.2 Pattern des 9 points référencés 49
4.3 Tiling hexagonal 50
4.4 Ensemble des tuiles non-vides 51
4.5 Itérations dans une tuile 52
4.6 Pattern non-convexe 53
4.7 Algorithme *row_echelon* 54
4.8 Implémentation à base de T9 57
4.9 Transposition de matrice 57
4.10 Accès mémoire pour une transposition 58
4.11 Distribution des calculs 58
4.12 Éléments lus et écrits pour une tuile 60
4.13 Pattern des accès pour une tuile 60
4.14 Exemple d'application SPEAR-DE 66
4.15 Exemple de distribution 2D des éléments $a \in [0..59]$ sans redondance. . . . 75
4.16 Exemple de distribution 2D des éléments $a \in [0..59]$ sans redondance avec équilibrage de charge pour les processeurs. 76

5.1 L'approche de programmation concurrente par contraintes 79

RÉSUMÉ

Les machines multiprocesseurs, multi-coeurs et les accélérateurs de type GPU se généralisent et pourtant il devient de plus en plus difficile pour les programmeurs de tirer profit de leurs capacités. La compilation source-à-source des applications permet de faciliter le développement d'implémentations efficaces pour ces architectures complexes.

Mes travaux de recherche s'inscrivent selon deux axes principaux. Le premier relève de la compilation et de l'optimisation d'applications en vue de leur exécution efficace sur des architectures parallèles. Le deuxième axe relève de l'utilisation de l'algèbre linéaire en nombres entiers pour modéliser les problèmes rencontrés.

Cette thèse présente, tout d'abord, les analyses statiques et dynamiques de programmes développées pour vérifier la correction d'un code et faciliter sa maintenance, telles que la mise en conformité par rapport à la norme du langage source, la détection de variables non initialisées, la vérification du non-débordement des accès à des éléments de tableaux et le calcul de dépendances de données. Ces analyses ont été intégrées dans le compilateur `PIPS`.

Puis, des méthodes de génération automatique de code de contrôle et des communications à partir de spécifications sont exposées. Des modélisations du placement des données sur les processeurs, des calculs pouvant être exécutés en parallèle et des communications devant être générées pour conserver la cohérence des données sont proposées. Les algorithmes de synthèse de code spécifiquement développés pour `HPF`, pour une mémoire virtuelle partagée émulée sur une architecture distribuée et pour des transferts compatibles avec des `DMAs` sont ensuite détaillés.

Finalement, une approche globale du problème de placement d'une application pour une architecture embarquée parallèle, qui utilise la programmation logique concurrente par contraintes comme moteur, est présentée.

Une même abstraction a été choisie pour les analyses et la modélisation des problèmes d'optimisation et de génération de code : un système de contraintes linéaires où les variables sont des entiers. Le choix des *$\mathcal{Z}$-polyèdres* a permis de rester dans un cadre algébrique disposant d'une large gamme de méthodes de résolution, aisées pour les preuves.

ABSTRACT

Multiprocessor, multicore and accelerator machines are widespread but benefiting from their capabilities becomes more and more difficult. The source-to-source compilation of applications offers help for the development of efficient implementations targeting these complex architectures.

My research fits into two main areas. The first one is the compilation and optimization of applications for parallel architectures. The second one focuses on the use of diophantine linear algebra to model the encountered problems.

First, this thesis presents static and dynamic program analyses that address the issues of code correctness and maintenance, in particular the compliance with a source language standard, the detection of uninitialized variables, the detection of out-of-bound array accesses and the computation of data dependencies. These analyses have been developed in the `PIPS` compiler.

Then, new methods to automatically generate control and communication codes from specifications are detailed. Models for the mapping of data onto processors, for the computations that can be executed in parallel, and for the communications to be generated to maintain data consistency are proposed. Specific code synthesis algorithms developed for `HPF`, for a shared virtual memory emulated onto a distributed architecture and for architectures having `DMAs` transfer capabilities are detailed.

Finally, a global approach to the mapping problem of an application onto an embedded parallel architecture using the Concurrent Constraint Logic Programming paradigm is presented.

A unifying abstraction has been chosen to specify all these analyses, optimization models and code generation problems : a system of affine constraints where the variables are integers. These so-called *$\mathcal{Z}$-polyhedra* allow us to stay within a single algebraic framework, offering a wide range of resolution methods, thus facilitating proofs.