École nationale supérieure des mines de Paris

Contributions à l'optimisation de programmes scientifiques

THÈSE

présentée et soutenue publiquement le 17 decembre 1999

pour l'obtention du

Doctorat de l'École des Mines de Paris
(spécialité Informatique temps réel, Robotique, Automatique)

par

Julien Zory

Composition du jury

Président :	Paul Feautrier
Rapporteurs :	François Bodin Bastien Chopard
Examinateurs :	Patrick Baudin Michel Cosnard Frédéric Desprez Benoit Dupont de Dinechin Catherine Mongenet
Directeurs :	Fabien Coelho François Irigoin

Centre de Recherche en Informatique

Résumé

L'optimisation des applications de calcul scientifique permet de profiter pleinement de la puissance toujours croissante offerte par la technologie informatique. Les travaux d'optimisation présentés dans cette thèse s'appuient sur différentes techniques de transformations source-à-source des programmes. L'approche proposée repose, d'une part, sur l'exploitation des propriétés algébriques des opérateurs manipulés dans le cadre de l'évaluation des expressions et, d'autre part, sur la prise en compte des capacités de travail concurrent des machines actuelles (processeurs superscalaires et architectures parallèles). Les contributions de cette thèse s'articulent autour de trois principaux axes :

• Différentes techniques pour l'amélioration de la compilation du langage High Performance Fortran sont proposées ; la réduction du surcoût introduit par les communications ainsi que l'amélioration du code généré automatiquement sont notamment étudiées.

• Les propriétés algébriques des opérateurs sont exploitées pour réécrire les expressions sous des formes mathématiquement équivalentes et mieux adaptées à leur évaluation par des processeurs disposant de parallélisme d'instructions. L'extraction des opérations fusionnées de type multiply-add est aussi étudiée.

• Les propriétés algébriques des opérateurs sont également utilisées afin d'exploiter de nouvelles opportunités d'optimisations de boucles. Les techniques de déplacement de code invariant et d'élimination des sous-expressions communes sont analysées dans un contexte associatif-commutatif.

Ce travail concerne donc l'étude, la mise en œuvre puis l'expérimentation de nouvelles techniques de compilation qui permettent de diminuer le temps d'exécution des applications de calcul scientifique. Il intéresse aussi les développeurs d'applications parallèles soucieux de performance. Des gains significatifs ont été obtenus pour des applications réelles et un prototype, EOLE, a été implémenté afin de valider l'approche proposée.

Abstract

Optimization of scientific computations is the common goal of my contributions. They are based on a set of source-to-source program transformations. These transformations exploit the algebraic properties of operators used in computation and control expressions on one hand, and the different kinds of parallelism provided by recent computers, e.g. superscalar and multiprocessors, on the other hand. These techniques are grouped in three different areas :

• Several techniques are presented to improve the compilation of Fortran HPF; the communication overhead is reduced, IO's and control code are optimized in a prototype compiler, HPFC.

• Source arithmetic expressions are rewritten using algebraic properties of operators into equivalent ones which require fewer operations and are better suited for a parallel pipelined evaluation with limited resources. Merged operators such as multiply-add are also targeted by the expression restructurations.

• Algebraic properties of operators are also used to generate new opportunities of loop optimizations. Invariant code motion and subexpression elimination are revisited using associativity and commutativity.

These new techniques are presented as well as their implementation and experimental results which show their efficiency at reducing execution times for scientific application. These source-to-source techniques are relevant for parallel application programmers as well as for compiler writers. Significant speedups were measured on real applications processed by a prototype compiler called EOLE and developped to assess the potential of these techniques.

Remerciements

Les remerciements ont droit à une place d'honneur dans tout manuscrit de thèse puisqu'ils se trouvent en général dans les toutes premières pages de celui-ci. Je pense que cette place est méritée et je n'aurais pu envisager de réaliser ce travail seul, sans le soutien que m'ont apporté tant de personnes au cours de ces trois années.

Mes premiers mots sont à l'intention de deux personnes qui ont accepté de mener un travail de longue haleine (qui, je l'espère, était tout de même intéressant) ; j'adresse donc un grand merci à François Bodin et Bastien Chopard, tous deux rapporteurs de ma thèse.

Je tiens également à réserver une place toute particulière à Michel Cosnard qui, après m'avoir beaucoup aidé lors de mon passage au LIP il y a quelques années, m'a fait le plaisir de présider mon jury de thèse.

Je remercie également toutes les personnes qui ont accepté de participer à mon jury de thèse. Merci donc à Paul Feautrier et Catherine Mongenet qui par leur présence dans mon jury honorent mon travail. Merci également à Benoit Dupont de Dinechin dont la participation en tant qu'ancien du CRI et digne représentant de la recherche industrielle est précieuse dans ce jury.

J'adresse tous mes remerciements à Patrick Baudin qui a grandement contribué à mon travail de thèse en me permettant d'exploiter les capacités du simplificateur de CAVEAT. Merci également pour l'accueil qu'il m'a réservé au CEA en compagnie de Jacques Raguideau.

Un grand merci également à Frédéric Desprez qui après m'avoir guidé pendant mon DEA, participe maintenant à mon jury de thèse. Plus qu'à un chercheur, c'est à un ami que j'adresse ici mes remerciements.

Que dire de Fabien Coelho et François Irigoin qui m'ont accompagné pendant ces trois années et ont mis en place toutes les conditions nécessaires à l'évolution de mon travail de thèse. Merci donc à Fabien pour qui je suis et je resterai le premier thésard. Parfois têtu mais toujours de bon conseil et excellent dans son rôle de chercheur, je lui dois beaucoup quant à la progression de mon travail. La présence de François à mes cotés a quant à elle apporté plus de rigueur dans mes recherches. Côtoyer tous les jours une personne aussi pointue dans son domaine et possédant autant de connaissances dans tous les autres aura été pour moi une belle expérience. Merci enfin à tous les deux pour le travail de relecture qu'ils ont réalisé, tant pour les publications que pour le manuscrit de thèse.

Ma présence au Centre de Recherche en Informatique de l'Ecole des Mines de Paris pendant ces quelques années restera ancrée dans ma mémoire pour longtemps. Merci donc à Robert Mahl pour m'avoir acceuilli à ses cotés et partagé quelques séances de ping-pong bien agréables.

Je tiens également à remercier tout particulièrement Jacqueline Altimira qui, tout au long de ces trois années, a toujours fait le maximum pour faciliter mon travail et illuminer le CRI par sa seule présence.

Pierre Jouvelot, personnage atypique et au combien attachant, a lui aussi participé activement à mon travail de recherche par ses relectures nombreuses et ses commentaires toujours constructifs. Mais, ce que je retiendrai le plus, ce sont sans aucun doute nos nombreuses discussions dans la navette, nos parties de ping-pong, un tour d'avion ou encore ce somptueux repas à l'Aigle Noir.

Je tiens maintenant à féliciter Corinne Ancourt qui a réussi à supporter les discussions parfois épineuses que Fabien et moi avions dans son bureau, sans jamais nous jetter dehors. Bravo pour cette belle preuve de patience ! Je la remercie également pour son aide régulière et pour la touche féminine qu'elle apporte au rez-de-chaussée du CRI qui, sans elle, ressemblerait un peu à un pensionnat pour garçons.

Quoi de plus normal que de rendre hommage à Katarzyna Wegrzyn et Olivier Albiez pour leur présence à mes cotés tous les midis dans des discussions philosophiques autour d'une tasse de café.

Je remercie également Nadine Olivier, Ronan Keryell, Laurent Daverio et Jean-Marie Kessis pour leur gentillesse et les nombreux coups de mains qu'ils m'ont apportés ça et là au cours de ma thèse.

J'adresse enfin un grand merci à Etienne Decencière, Nicolas Jeannee, Christian Lajaunie, Estelle Schulher, Benjamin Schwartz et d'autres encore qui en grimpant, courant, pédalant à mes cotés ont apporté à ma thèse la touche sportive et conviviale dont j'ai toujours eu besoin.

Je terminerai ici par le plus important à mes yeux, en remerciant ma femme, mon fils et toute ma famille qui m'ont accompagné et soutenu pendant trois ans, me donnant la volonté de mener ce projet du mieux possible.

A Rémi et Sophie.

Table des matières

Liste des tableaux

Table des figures

Première partie

Introduction

Chapitre 1

La course aux performances

La puissance de calcul des ordinateurs a considérablement augmenté au cours de ces dernières décennies et le phénomène devrait continuer encore quelque temps. Les progrès reposent à la fois sur des avancées technologiques (circuits intégrés, semi-conducteurs, etc.) et architecturales (intégration des différents composants, mémoire cache, unités de calcul, etc.). L'objectif des constructeurs est d'atteindre sans cesse de meilleures performances à un coût raisonnable.

1.1 Des besoins toujours plus importants

Si l'on cherche à faire progresser la technologie pour augmenter la puissance de calcul, c'est avant tout parce que l'on en éprouve le besoin. Que ce soit pour surfer sur le désormais célèbre WWW ou pour modéliser la naissance d'une étoile, l'utilisateur est toujours plus gourmand en ressources de calcul. Les travaux présentés tout au long de cette thèse concernent plus particulièrement une catégorie d'applications nécessitant de grosses capacités de calcul : les programmes scientifiques. La puissance de calcul des ordinateurs est en effet très utile à la fois en recherche théorique (pour construire et tester de nouveaux modèles) et dans la recherche appliquée et/ou industrielle (pour effectuer des simulations). Avant d'atteindre la solution du problème, il n'est pas rare que l'ordinateur ait à effectuer quelques trillions d'opérations ! L'augmentation de la puissance des ordinateurs permet donc aux scientifiques d'ouvrir de nouvelles portes d'exploration dans des espaces de grande dimension (l'univers, la physique des particules, la biologie moléculaire, etc.).

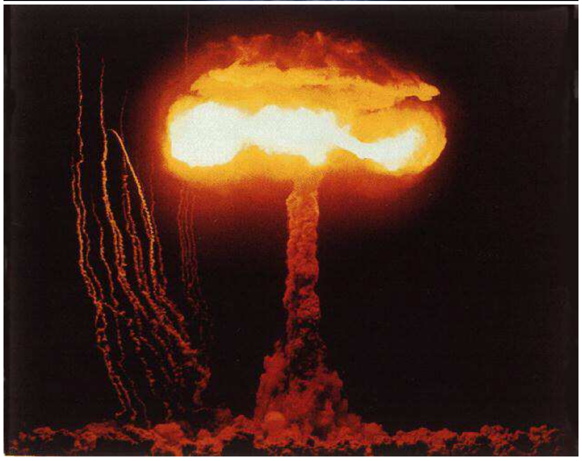
Remplacer les prototypes par des simulations, explorer des domaines toujours plus grands, modéliser un phénomène avec toujours plus de précision... Les exemples ne manquent pas pour décrire les besoins en puissance de calcul des ordinateurs ; en voici quelques uns :

• **Prévisions météorologiques**

Les tornades causent chaque année des pertes d'environ 14 milliards de dollars. Or, ce coût pourrait considérablement diminuer si les tornades étaient prévues suffisamment à l'avance ; cela permettrait par exemple de détourner les avions vers des aéroports situés dans une zone plus tranquille. L'utilisation d'un supercalculateur T3D permet des prévisions à 5h30 en 1h20 de calculs.
L'acquisition d'un supercalculateur `Fujitsu VPP700E` par les services de Météo-France en 1998 est également significative de la puissance de calcul que demande la prévision météorologique.

• **Prototypage**
Le développement d'un moteur (avion, voiture, etc.) nécessite la mise en place de prototypes permettant de valider les évolutions technologiques proposées. Grâce à la mise en place de modèles simulés sur ordinateurs, le constructeur Renault a pu diminuer les coûts de fabrication d'un facteur 2 (économie de 2,5 millions de francs par développement).

• **Simulation nucléaires**
Suite à l'application du moratoire sur les essais nucléaires, les grandes nations ont investi lourdement dans les simulations numériques. Ainsi, le gouvernement américain a investi un milliard de dollars (sur 10 ans) dans le projet ASCI (Accelerated Strategic Computing Initiative), avec l'objectif de construire des machines capables d'atteindre les performances nécessaires aux simulations nucléaires envisagées.



1.2 Des évolutions architecturales

Parallèlement aux avancées technologiques qui permettent de mettre en place des processeurs travaillant à des fréquences de plus en plus élevées, les concepteurs de machines font également évoluer l'architecture des ordinateurs. Les deux types d'évolution qui seront introduits ici concernent tous deux l'utilisation de la notion de parallélisme. Seules les caractéristiques que nous souhaitons prendre en compte pour l'élaboration de nouvelles techniques de compilation seront introduites. Les notions de hiérarchie mémoire, jeux d'instructions, entrées-sorties, etc. ne seront quant à elles pas du tout abordées. Le lecteur est invité à consulter les ouvrages [SV94, SM95, PH96, SL96, GUD96, Bay98] pour obtenir plus d'informations.

L'introduction de plus en plus massive de la notion de *concurrence* est au cœur de l'évolution des architectures modernes ; en voici quelques exemples :

– La mémoire a été organisée en *bancs* pour permettre un accès concurrent à différentes adresses.
– Les modules d'entrées-sorties sont de plus en plus souvent répliqués afin de permettre des accès simultanés.
– Plusieurs instructions peuvent être en cours d'exécution simultanément grâce au pipeline (voir ci-dessous).
– Les supercalculateurs sont composés de plusieurs processeurs capables de traiter indépendamment leur propre flot d'instructions.
– Chaque processeur contient plusieurs unités de calcul lui permettant d'exécuter plusieurs instructions en parallèle.

L'émergence de cette capacité de travail concurrent (ou parallèle) est sans doute l'une des évolutions architecturales majeures ayant permis d'atteindre la puissance de calcul actuelle.

Pipeline Le *pipeline* est une technique par laquelle plusieurs instructions peuvent être recouvertes entre elles à l'exécution. Le parallèle avec une ligne d'assemblage automobile permet de mieux comprendre le procédé : la construction d'un véhicule est découpée en plusieurs étapes, chacune effectuant une tâche bien définie. Bien que chacune des voitures soit construite de manière séquentielle, la ligne de fabrication opère simultanément sur plusieurs véhicules (chacune à une étape différente). Dans un pipeline, chacune des étapes (appelée étage) gère une partie d'une des multiples instructions en parallèle. Si l'on suppose que le pipeline contient k étages menant des tâches de durées identiques d, le temps de traitement d'une instruction est $k \times d$ unités de temps ; cependant, une instruction nouvelle est terminée toutes les d unités de temps. Le pipeline permet donc d'augmenter le débit d'exécution des instructions sans toutefois diminuer le temps de traitement individuel d'une instruction. La figure 1.1 illustre le mécanisme de pipeline ; les étages représentés sont ceux que l'on retrouve le plus couramment lors de l'exécution d'une instruction (IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access et WB = write back [PH96]).

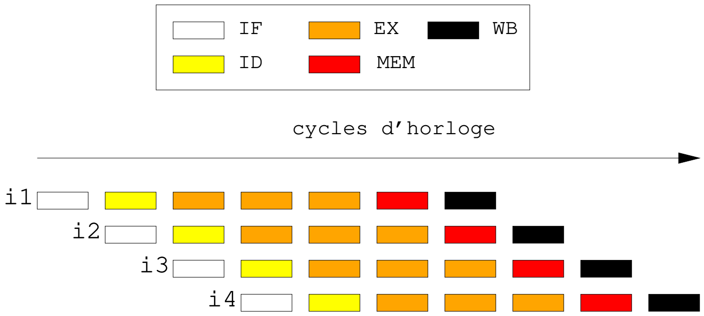


FIG. 1.1 – *Instructions pipelinées.*

Le pipeline constitue donc un concept majeur permettant d'accélérer l'exécution des programmes. Il est caractérisé par sa profondeur (le nombre d'étages) et son temps de cycle (durée de l'étage le plus lent). Les implémentations de cette technique varient d'un processeur à un autre (pipeline de 7 étages sur le `MIPS R10000`, 8 étages sur le `Power2`, 9 étages sur le `SUN UltraSparc IIi` et 12 sur l'`INTEL Pentium Pro`).

Parallélisme d'instructions Nous avons vu que le pipeline permet de recouvrir les calculs entre eux et donc, lorsque les instructions sont indépendantes, de les exécuter en parallèle. Cette technique constitue donc un des composants de ce que l'on appelle le *parallélisme d'instruction* (ILP [1]). Le pipeline permet de réduire le nombre de cycles séparant l'exécution de deux instructions successives. Cependant, cette technique ne permet pas de passer en dessous de la limite de 1 cycle par instruction (car une seule instruction est chargée à chaque cycle d'horloge). Pour cela, il faut donc pouvoir lancer plusieurs instructions par cycle d'horloge. L'évolution de l'architecture dans ce sens a donné naissance aux processeurs **VLIW** (Very Long Instruction Word) et **SUPERSCALAIRES**.

- Les processeurs *VLIW* [Fis84] lancent un nombre fixé d'instructions à chaque cycle. La technique consiste à compacter plusieurs opérations (par exemple deux instructions entières, deux instructions flottantes, deux accès mémoire et un branchement) dans une « grande » instruction. Le travail consistant à regrouper des opérations indépendantes pour former ces longues instructions est laissé au compilateur.
- Un processeur *superscalaire* [SS95b] peut typiquement aujourd'hui lancer entre 2 et 8 instructions par cycle d'horloge (4 sur `ultrasparc`, 4 sur `R10000` et 6 sur `Power2`). Ces instructions doivent être indépendantes et le nombre d'accès à la mémoire est limité. Les instructions peuvent être ordonnancées de manière statique par le compilateur ou de manière dynamique géré par le matériel.

1. Intruction Level Parallelism

Le modèle **VLIW** a été partiellement abandonné au profit des processeurs **superscalaires**. En effet, le VLIW ne comporte pas de compatibilité binaire entre les générations successives d'une architecture VLIW. L'augmentation de la taille du code, le nombre de trous dans le flot d'instructions ainsi que la longueur du chemin d'accès aux instructions ont également été dénoncés. De plus, la gestion du parallélisme en VLIW ne repose que sur le compilateur ; or les techniques efficaces disponibles aujourd'hui pour les processeurs superscalaires sont arrivées un peu tard pour le VLIW.

Latence des opérations Une partie du travail présenté dans cette thèse a pour objectif de diminuer le *nombre* d'opérations qu'il est nécessaire d'évaluer au cours de l'exécution d'un programme. Or, toutes les opérations ne sont pas égales face au temps d'évaluation. Diminuer le nombre d'occurrences de calcul d'une racine carré sera par exemple beaucoup plus rentable que de réduire simplement le nombre d'additions. Il faut tout d'abord distinguer les opérations qui sont effectuées par le logiciel (par exemple les appels de fonctions mathématiques comme le logarithme ou les fonctions trigonométriques) de celles qui sont évaluées par le matériel (addition, multiplication, etc.). Le surcoût introduit par le changement de contexte d'exécution pour une évaluation logicielle est en effet très important. Certaines opérations passent d'une catégorie à l'autre selon le processeur utilisé, par exemple la racine carré.

La latence d'une opération est définie par le nombre de cycles d'horloge écoulés entre l'instant où tous les opérandes sont disponibles et le moment où le résultat est disponible pour une autre instruction. Cela correspond donc en général à la durée de tous les étages d'exécution du pipeline. L'information portée par la latence est évidemment insuffisante puisqu'elle est étroitement liée à la fréquence du processeur, c'est-à-dire la durée d'un cycle d'horloge. Quelques exemples de latence pour les opérations flottantes sont donnés dans le tableau 1.1.

Opération	MIPS R10000 cycle - 14 ns	IBM Power2 cycle - 3,65 ns	INTEL PentiumPro cycle - 5 ns	SUN UltraSparcIIi cycle - 3 ns
Addition	5	2	3	3
Multiplication	5	2	5	3
Division	18	17	38	22
Racine carrée	32	27	69	22

TAB. 1.1 – *Latences des opérations flottantes (double précision).*

Machines parallèles L'idée du calcul parallèle est de construire un ensemble d'éléments de calcul capables de communiquer et de coopérer pour résoudre un problème donné de grande taille. On utilise pour cela des supercalculateurs composés d'un nombre plus ou moins grand de processeurs. L'histoire du calcul parallèle a déjà connu l'utilisation des machines *vectorielles multi-processeurs* à mémoire partagée, des machines *synchrones* (très grand nombre d'éléments de calcul exécutant de manière synchrone la même instruction sur des données locales - ex : Connection Machine), des machines à *mémoire distribuée* (processeurs « ordinaires » communiquant entre eux par un réseau d'interconnexion - ex : T3D, SP2) et des machines à *mémoire partagée* (SMP [2] - ex : E10000 ou NUMA [3] - ex : Origin2000). Les travaux de cette thèse qui concernent le calcul parallèle (partie II) sont fondés sur l'utilisation de *machines parallèles à mémoire répartie*. Le schéma général de ce modèle de machines est présenté figure 1.2. Chaque processeur dispose d'une mémoire locale et d'un séquenceur propre (il travaille à son propre rythme). Les communications et les points de synchronisation entre les processeurs se font grâce au réseau d'interconnexion.

Les processeurs communiquent entre eux par le biais de messages qui circulent sur le réseau d'interconnexion. Il existe de nombreuses topologies (grille, tore, hypercube, anneau, etc.) et les caractéristiques telles que le diamètre (distance maximale entre deux processeurs) ou la connectivité (nombre de chemins permettant de relier deux processeurs) déterminent la qualité du réseau.

2. Shared Memory Processors
3. Non Uniform Memory Access

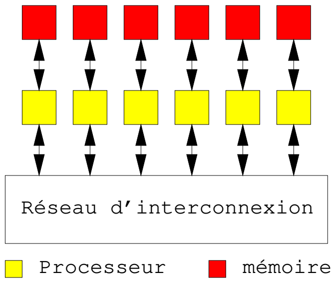


FIG. 1.2 – *Schéma général d'une machine parallèle à mémoire distribuée.*

1.3 L'impact sur le logiciel

Face à cette incroyable augmentation de la puissance des ordinateurs, un des défis du calcul scientifique est de réussir à développer les techniques logicielles permettant de s'en servir. En effet, le constat est amer lorsque l'on compare les performances théoriques promises par les constructeurs avec celles qui sont généralement obtenues pour des applications réelles.

De nombreux travaux ont été menés depuis des années dans le domaine de la compilation ; la littérature est très abondante. Force est de constater cependant que, malgré toutes les techniques déjà disponibles, le passage de l'application scientifique à son exécution sur un ordinateur n'est toujours pas satisfaisant. En effet, les caractéristiques nouvelles des machines d'aujourd'hui (parallèles, superscalaires, etc.) ainsi que l'évolution du type de programmes traités (langages de plus en plus haut niveau, génération automatique de code, etc.) nécessitent la mise en place de nouvelles techniques. Avant d'entrer dans l'exposé des travaux réalisés au cours de cette thèse, nous rappelons l'éventail des techniques utilisées en compilation pour améliorer les temps d'exécution d'un programme. Seuls les points utiles à la compréhension de l'exposé seront véritablement abordés. Le lecteur est invité à consulter les nombreux ouvrages de référence [ASU86, ZC91, Wol96].

1.3.1 Optimisation de code

Il existe une multitude de manipulations permettant de diminuer le temps d'exécution d'un programme. La première, que nous n'aborderons pas dans cette thèse, consiste bien évidemment à rechercher l'algorithme ou la méthode de calcul qui sont le mieux adaptés au problème à résoudre. Le résultat de ce travail, le programme, peut ensuite être amélioré en différents points, soit au niveau source, soit au cours de la compilation. On peut distinguer deux catégories de transformations : celles dont l'objectif est d'améliorer la « qualité » générale du code et celles qui tentent d'adapter le programme à un contexte d'exécution précis. L'amélioration maximale pour un effort minimal sera obtenue si l'on peut identifier les parties du programme les plus fréquemment utilisées. En pratique, ce sont les nids de boucles qui constituent les points clés à optimiser.

Les optimisations *indépendantes de la machine* sont des transformations de programme qui améliorent le code sans prendre en considération les propriétés de la machine. On peut mettre dans le terme *amélioration* différentes modifications du programme telles que la taille du code (machines embarquées [BBB+98]) ou l'espace mémoire occupé lors de l'exécution du code. Mais, nous nous intéresserons uniquement ici aux techniques permettant de réduire le temps d'exécution du programme ; les transformations diminuant le nombre d'opérations (ou d'instructions) du programme seront plus particulièrement étudiées. Les optimisations les plus couramment utilisées sont le déplacement de code invariant, l'élimination des sous-expresions communes, l'élimination de code mort, l'évaluation partielle, la propagation des copies, la réduction de force, etc. (cf. ouvrage de référence [ASU86]).

Parmi les transformations permettant d'adapter le programme pour une exécution dans un contexte donné, on peut distinguer les transformations de haut niveau (au niveau source) et les transformations de bas niveau (au niveau assembleur).

- Selon le langage et l'architecture utilisés, les *trucs et astuces* d'une programmation efficace sont nombreux ([IBM95b, ABH97, Chi97]). Lorsque ces techniques sont inconnues du programmeur, il est possible d'en appliquer certaines automatiquement. Parmi les plus courantes, on trouve les transformations de boucles (échange de boucles, fusion de boucles, etc. [BDFH96]) utilisées notamment pour améliorer les accès à la mémoire cache [MCT96].
- Il existe également de nombreuses optimisations de bas niveau, permettant de tirer profit des caractéristiques de la machine utilisée. Parmi les plus importantes se trouvent les techniques d'ordonnancement, le *software pipelining* [AJLA95, LAV98] (avec loop unrolling), l'allocation de registres [Set75, BS76, LM97], le *data prefetching* [VWL97, SGH97], etc. Le lecteur pourra se reporter à des ouvrages de référence tels que [DFSLOJ94, PH96].

1.3.2 Calcul parallèle

La complexité introduite par l'exécution parallèle de plusieurs tâches a demandé rapidement la mise en place de supports de programmation. Les sources de parallélisme dans une application peuvent être très variées. On distingue en général trois types de parallélisme :

- le **parallélisme de données** (*data parallelism*), pour lequel on cherche à appliquer en parallèle un même calcul à des données distinctes,
- le **parallélisme de contrôle** (*control parallelism*), qui effectue plusieurs tâches différentes en parallèle,
- le **parallélisme de flux** (*flow parallelism*), consistant à pipeliner les travaux (cf. plus haut).

Le calcul parallèle dont nous parlerons ici sera uniquement fondé sur le *parallélisme de données*. Les programmes traités manipulent en général un grand nombre de données sous forme de vecteurs ou matrices de grande taille. Ces données seront découpées (distribuées) et affectées aux différents éléments de calcul (les processeurs). Il existe plusieurs modèles de programmation data-parallèle avec lesquels le programmeur aura plus ou moins d'effort à fournir, pour de plus ou moins bonnes performances. Deux écoles s'opposent aujourd'hui à ce sujet ; certains désirent conserver un parallélisme explicite et demandent au programmeur de générer lui même toutes les routines de « bas niveau », alors que d'autres tentent au contraire de soulager un maximum le programmeur, en générant « automatiquement » du code parallèle à partir de renseignements apportés sur la sémantique de l'algorithme. Chacune des deux approches présente des avantages et des inconvénients ([VNL97]).

Parallélisation automatique La programmation parallèle est plus complexe que la programmation séquentielle. De plus, une grande partie des applications scientifiques existe déjà sous la forme d'un programme séquentiel. Partant de ces deux constats, la mise en place de techniques permettant de construire automatiquement une version parallèle à partir de la version séquentielle est un des points clés du calcul parallèle. De nombreux travaux ont été menés dans ce domaine depuis 1990 et plusieurs ouvrages présentent les résultats obtenus [ZC91, Wol96, GUD96]. Il existe aujourd'hui plusieurs outils de parallélisation automatique (PIPS [IJT$^+$96], PARTITA [HAS95], Polaris [BEF$^+$96], SUIF [WFW$^+$94], NESTOR [SD99], PAF [FDT87], etc.).

La parallélisation automatique de programmes est une tâche complexe ; des langages data-parallèles contenant des directives destinées à aider le compilateur dans le processus de parallélisation ont donc été mis au point. Parmi les langages construits à partir de Fortran, les plus utilisés sont **HPF** [For93] (voir chapitre 3) et **OPENMP** [Boa97]. Il existe également des propositions pour des langages adaptés au calcul parallèle à partir de C (`HyperC` [CP94]), C++ (`C++//` [sys96]) ou Java (`Do!` [LP97] - `Java//` [DC98]).

Programmation par passage de messages Le programmeur a également la possibilité de gérer lui même la collaboration entre les processeurs en utilisant des primitives de communications. Afin de concilier performance, confort d'utilisation et portabilité, des bibliothèques de communications ont été développées et progressivement acceptées par la communauté. La bibliothèque **PVM** (*Parallel Virtual Machine* [GBD$^+$93]) a été pendant longtemps le standard du domaine public. Un forum constitué de constructeurs, utilisateurs et universitaires a ensuite donné naissance à **MPI**[4] (*Message Passing Interface*[SOHL+96]), qui s'est imposé comme le nouveau standard (voir l'article [GL97] pour une comparaison entre les deux librairies). Il existe également des librairies *constructeurs* livrées avec les machines parallèles (par exemple MPL sur SP2 ou Intel-NX/2 sur Paragon) ; elles sont cependant de moins en moins utilisées en raison de leur non portabilité.

1.4 Quelle mesure de performance?

Tout au long de cette thèse, nous chercherons à évaluer l'amélioration en terme de *performance*, obtenue par l'application d'une technique donnée. Il n'existe pas réellement de mesure parfaite permettant d'évaluer cette performance. La méthode la plus directe consiste à comparer les temps d'exécution respectifs des deux versions (initiale et transformée) d'une même application. Nous emploierons également une mesure donnée en MFLOP/s, indiquant le nombre (en millions) d'opérations flottantes exécutées par secondes. Cette mesure permet de connaître l'efficacité de l'exécution d'un code et donc d'avoir une idée du potentiel d'optimisation restant. Cependant, cette mesure ne peut être utilisée que pour des transformations à nombre d'opérations constant. Nous utiliserons le terme de *performance crête* pour désigner la performance maximale théorique en MFLOP/s pouvant être atteinte par une machine donnée.

4. Les transparents de mon cours sur MPI sont disponibles depuis `http://www.cri.ensmp.fr/~zory`

Chapitre 2

Organisation de la thèse

Les travaux présentés tout au long de cette thèse sont guidés par un seul et même objectif : diminuer le temps d'exécution des applications de calcul scientifique. Afin de mieux illustrer la variété des techniques permettant d'améliorer les performances d'un code, nous avons choisi de nous appuyer, tout au long de ce document, sur une application réelle appelée **ONDE24**.

2.1 Onde24 : le fils rouge

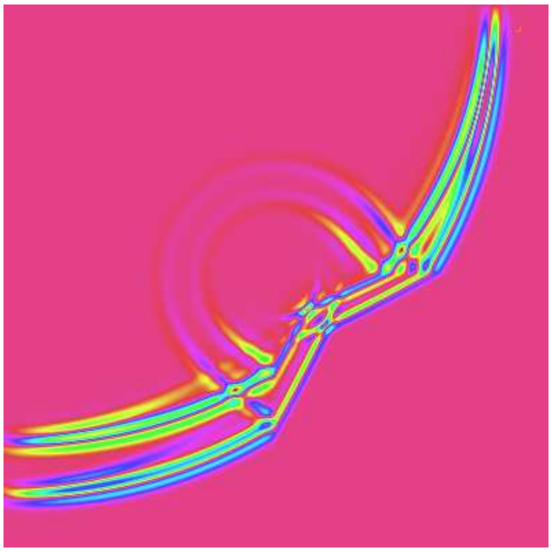


FIG. 2.1 – *Propagation d'onde en 2 dimension.*

Onde24 est une benchmark développée et utilisée à l'Institut Français du pétrole (IFP). Elle réalise une modélisation par équation d'ondes d'un phénomène de propagation en dimension 2. Le domaine étudié est supposé carré, avec une surface libre et des conditions absorbantes du second ordre sur les bords. L'application modélise donc la propagation d'une onde (du type champ de pression) générée par une source donnée, dans un milieu hétérogène (la vitesse de propagation de l'onde n'est pas constante en tout point du domaine). La figure 2.1 illustre ce phénomène avec une onde se propageant suite à une explosion générée en haut à gauche de l'image ; la forme de l'onde montre une discontinuité dans le champ de vitesse de propagation (ici de forme carrée).

L'extrait de code en partie gauche de la figure 2.2 représente le noyau de calcul de l'application (75% du temps d'exécution de l'application d'origine est passé dans cette boucle). La méthode de résolution est fondée sur l'utilisation d'un schéma aux différences finies d'ordre 2 en temps et 4 en espace. Comme le montre la partie droite de la figure 2.2, la solution U en un point (I,J) du domaine à l'instant $t+1$ (KP) est calculée en fonction de :

- la solution en ce point aux instants $t-1$ (KP) et t (KM),
- la solution pour ses quatre voisins à distance 1 à l'instant t,
- la solution pour ses quatre voisins à distance 2 (sur les axes) à l'instant t,
- la vitesse de propagation V en ce point.

Le tableau utilisé pour stocker la solution (pour un domaine en 2 dimensions) comporte une troisième dimension permettant de mémoriser la solution pour deux pas de temps (KM et KP). Les valeurs de KM et KP sont simplement permutées à chaque iteration.

```
[...]
 DO J = 3,NP-2
   DO I = 3, NP-2
     U(I,J,KP) =
$      2 * U(I,J,KM) - U(I,J,KP)
$      - V(I,J) * ( 60*U(I,J,KM)
$      -16*(U(I+1,J,KM) + U(I-1,J,KM)
$      + U(I,J-1,KM) + U(I,J+1,KM))
$      + U(I+2,J,KM) + U(I-2,J,KM)
$      + U(I,J-2,KM) + U(I,J+2,KM))
   ENDDO
 ENDDO
[...]
```

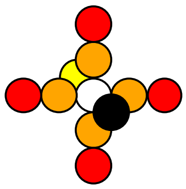


FIG. 2.2 – *Différences finies d'ordre 2 en temps et 4 en espace.*

Des sismographes sont placés en différents points du domaine pour enregistrer l'évolution du champ de pression au cours du temps. Notons enfin que la résolution aux bords du domaine est faite avec les différents schémas présentés figure 2.3.

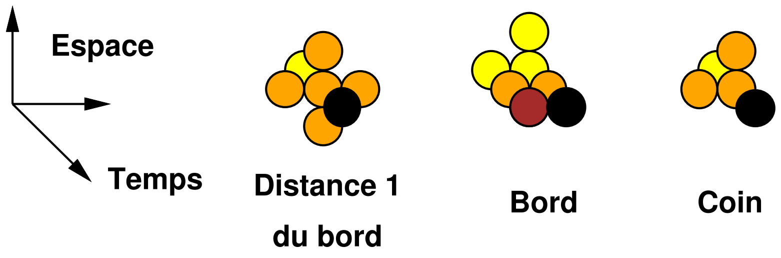


FIG. 2.3 – *Schémas utilisés sur les bords.*

L'application, écrite en Fortran 77, comprend environ 500 lignes de code (sans compter les commentaires). L'application se compose globalement de quatre phases : lecture du champ de vitesse dans un fichier, initialisation des valeurs, boucle en temps pour le calcul de la solution, puis écriture des résultats dans un fichier (plus quelques statistiques). Le champ de vitesse et la solution sont stockés sous la forme de REAL*8 (réels codés sur 8 octets).

2.2 Approche proposée

Le leitmotiv de ce travail est de mettre en place de nouvelles techniques permettant de réduire le temps d'exécution des applications de calcul scientifique. L'application Onde24 présentée dans ce chapitre constituera le fils rouge de notre discours ; nous essaierons, chaque fois que possible, d'illustrer les techniques proposées à partir d'extraits de ce programme.

L'exploitation des possibilités de travail concurrent qu'elles soient au niveau des instructions (ILP) ou au niveau des données (data-parallélisme) sera au cœur de nos préoccupations. Les techniques présentées tout au long de ce document reposent sur la prise en compte des évolutions architecturales (machines parallèles et processeurs superscalaires). En outre, des techniques permettant d'améliorer la qualité du code généré automatiquement dans le cadre de la compilation de programmes data-parallèles seront proposées à deux niveaux ; le travail sera effectué d'une part sur les communications et d'autre part sur le code séquentiel généré. L'utilisation de transformations algébriques afin de créer de plus grandes opportunités d'optimisation de ces codes sera notamment importante. Enfin, nous profiterons également des propriétés arithmétiques des opérateurs manipulés pour favoriser l'extraction du parallélisme d'instruction.

2.2.1 Quelques résultats expérimentaux

Le tableau 2.1 présente les temps d'exécution que nous avons obtenus pour différentes versions de cette application sur des processeurs `IBM P2SC` cadencés à 160MHz. Les résultats sont donnés pour l'application complète (entrées-sorties comprises) avec un jeu de données tel que $NP = 800$ (taille du domaine) et $NT = 2400$ (nombre de pas de temps). Les compilateurs natifs IBM `xlf v4.1` et `xlhpf v.1.2` (pour la version HPF) ont été utilisés.

Version	Temps d'exécution (secondes)	Gain
Initiale	229	-
EOLE	171	1,3
Software pipelining	121	1,9
HPF (4 PE)	87	2,6

TAB. 2.1 – *Résultats expérimentaux obtenus pour Onde24.*

- La version *initiale* de l'application est celle qui nous a été fournie par l'Institut Français du Pétrole.
- La version *EOLE* correspond au code obtenu suite aux transformations automatiques de programmes qui sont présentées dans la partie III. Ces transformations visent à réduire le nombre d'opérations et améliorer la structure des expressions afin de profiter du parallélisme d'instructions des processeurs superscalaires et des instructions de type *multiply-add*. Le gain obtenu est de 1,3.
- Nous avons également souhaité connaître le gain pouvant être obtenu par l'application (manuelle) d'optimisations plus poussées. Seul le noyau de calcul de l'application a été optimisé. La boucle interne (indice I) a été déroulée d'un facteur 4 ; les calculs des quatre itérations ont été réordonnancés de manière à mieux exploiter le parallélisme d'instruction (software pipelining). Nous avons également guidé le compilateur pour le chargement de mots de 128 bits en mémoire (à l'aide des instructions `quad load` et `store`). Cette optimisation manuelle a généré un gain d'un facteur 1,9.
- L'application a été portée en HPF (langage data-parallèle) et exécutée en parallèle sur 4 processeurs (P2SC à 120MHz). Le gain obtenu par ce processus de parallélisation est de 2,6.

2.2.2 À l'écoute des expériences ...

Avant d'entrer dans l'exposé des techniques utilisées pour accélérer l'exécution de Onde24, ce paragraphe décrit les constats que nous avons pu dresser lors des différentes tentatives d'optimisation de ce code. Ce sont autant de problèmes auxquels nous tenterons d'apporter des réponses au fur et à mesure de l'avancement de ce document.

• Les expériences menées sur cette application réelle nous ont rappelé à quel point l'évaluation des expressions au sein des nids de boucles est prépondérante pour les performances de l'application dans sa globalité. Le noyau de calcul de Onde24 (une seule expression !) monopolise a lui seul 75% du temps d'exécution. Dès lors que la solution du problème nécessite l'évaluation de calculs mathématiques sur un domaine souvent très grand et pour un nombre de pas de temps important, l'optimisation de l'évaluation des expressions doit être placée au cœur des efforts de compilation.

• L'augmentation de la puissance théorique de calcul enregistrée en cette fin de siècle est considérable. Pourtant, la répercution de ce phénomène dans la pratique sur les temps d'exécution des applications scientifiques est encore trop faible. En effet, force est de constater que plus les capacités des processeurs augmentent, plus il est difficile d'atteindre les performances maximales. Par exemple, pour la version initiale de l'application Onde24, nous avons observé des performances [5] de 44 MFLOP/s (resp. 91 MFLOP/s) sur un processeur Power2 (resp. P2SC) cadencé à 66 Mhz (resp. 160Mhz). Les performances crêtes étant de 266 (resp. 640) MFLOP/s, l'efficacité de l'exécution de Onde24 est passée de 16,5% à 14,2%. Dans les deux cas, les performances mesurées sont bien faibles comparées aux performances crêtes du processeur. De plus grands efforts doivent donc être réalisés pour tenter de rentabiliser le potentiel du matériel et notamment les opportunités importantes offertes par l'augmentation du parallélisme d'instructions et la mise en place d'instructions nouvelles telles que les *fma*.

• L'application de transformations algébriques sur le noyau de calcul de l'application (manuelle) puis sur toutes les expressions contenues dans l'application (automatique) a prouvé l'impact de ces techniques sur les temps d'évaluation des expressions exécutées sur des processeurs superscalaires. Cependant, cette opportunité n'est aujourd'hui pas prise en compte par les compilateurs et reste inaccessible pour le programmeur en raison des connaissances requises et de la difficulté de son application manuelle. Notons également que la manipulation des expressions à l'aide de l'associativité et de la commutativité des opérateurs est rendue difficile par une manipulation sous forme de code 3-adresses. Enfin, les tentatives d'optimisation plus poussées montrent que le potentiel de progression est encore grand.

• Le portage de Onde24 vers HPF s'est avéré plus long et plus délicat que nous ne l'avions supposé. Or, c'est une application réelle qui ne présente structurellement pas de difficulté pour une programmation avec parallélisme de données. De nombreuses difficultés ont été rencontrées au cours de ce travail et les performances, compte tenu de l'effort de portage réalisé, ne sont pas vraiment satisfaisantes. Les expériences réalisées sur plusieurs compilateurs HPF ont montré le manque de maturité de ces outils :

- Les sous-ensembles du langage acceptés étaient insuffisants.
- Les performances obtenues montrent le surcoût encore trop important introduit par les phases de communications entre les processeurs. Il est donc nécessaire d'une part de réduire le nombre de communications générées par les compilateurs et, d'autre part, de mettre en place des techniques permettant de limiter le coût des communications « obligatoires » restantes.
- Le programme parallèle généré par chacun des compilateurs a prouvé une fois encore que la génération de code automatique est une tâche difficile. Le code écrit pour le parcours des structures de données est en effet fortement optimisable. Cependant, la spécificité de cette nouvelle catégorie de codes générés *automatiquement* fait émerger de nouveaux besoins en termes de transformations de programmes.

2.2.3 Contenu de la thèse

Ce travail concerne la conception, la mise en œuvre et l'expérimentation de techniques permettant de diminuer le temps d'exécution des applications de calcul scientifique. Des gains significatifs ont pu être obtenus aussi bien en calcul parallèle distribué que pour l'exécution séquentielle mono-processeur. Trois axes de travail ont été dégagés des problèmes expérimentaux rencontrés :

5. Mesures données pour l'application complète (i.e. pas uniquement pour le noyau de calcul)

• Différentes techniques pour l'amélioration de la compilation du langage High Performance Fortran sont présentées :

- Nous proposons au chapitre 4 une analyse expérimentale du portage de l'application Onde24 de Fortran 77 vers HPF. Ce travail repose sur l'analyse comparative des résultats obtenus avec trois compilateurs différents.
- Le chapitre 5 présente une amélioration de la compilation des entrées-sorties en HPF visant à réduire considérablement le nombre de communications générées. La technique repose sur l'analyse des accès mémoire du programme et permet de résoudre un problème de performance constaté en pratique lors de la compilation d'applications de calcul scientifique réelles avec les compilateurs actuellement disponibles.
- Une analyse est présentée au chapitre 6 concernant le gain en performance pouvant être obtenu par l'utilisation du pipeline, dans le cadre de la réduction du surcoût des communications en calcul parallèle. Cette technique, utilisable par un compilateur HPF, permet de réduire le temps d'exécution des applications scientifiques exécutées sur machines à mémoire distribuée.

• Les propriétés algébriques telles que l'associativité ou la distributivité des opérateurs permettent de manipuler des expressions équivalentes au sens mathématique. Nous montrons que les temps d'évaluation de deux expressions équivalentes sur des processeurs superscalaires peuvent être très différents. Nous proposons des techniques permettant de réécrire les expressions afin d'améliorer les temps d'évaluations sur des processeurs disposant de parallélisme d'instructions. Un mécanisme de réécriture associative-commutative, deux heuristiques (factorisation et d'extraction d'opérations de type *multiply-add*) ainsi qu'un algorithme d'équilibrage des expressions sont décrits aux chapitres 8 et 9.

• Nous montrons également que l'utilisation des propriétés algébriques des opérateurs permet de créer de nouvelles opportunités pour les optimisations de boucles. Un algorithme de déplacement de code invariant est présenté au chapitre 13 puis une technique de réassociation permettant de construire des sous-expressions invariantes est proposée au chapitre 14. Enfin, une heuristique d'élimination des sous-expressions communes dans un contexte associatif-commutatif est présentée au chapitre 15.

2.3 Structure du document

Ce document a été découpé en trois grandes parties, apportant chacune des éléments de réponses aux axes de travail que les expériences sur Onde24 ont suggérés. Cette organisation tranchée ne doit pas faire oublier que chacune des parties interagit avec les deux autres. Malheureusement l'écriture séquentielle de cet ouvrage ne permet pas d'exprimer facilement les véritables interactions. Cette première partie d'*introduction* s'achève ici et sera immédiatement suivie par l'exposé des travaux concernant la *compilation du High Performance Fortran*. La seconde partie regroupe les techniques proposées dans le cadre de l'*évaluation d'expressions sur processeurs superscalaires*. Les contributions concernant les *transformations algébriques et optimisations de boucles* seront décrites par la troisième partie, puis nous tenterons de faire une synthèse de ce travail au cours de la *conclusion* qui sera agrémentée de quelques discussions.

Deuxième partie

Compilation du High Performance Fortran

Chapitre 3

Introduction au High Performance Fortran

La percée du parallélisme dans le monde industriel est très nettement freinée par la difficulté rencontrée à écrire des programmes portables et efficaces avec un effort de programmation raisonnable. C'est pour tenter d'apporter de nouvelles solutions qu'un forum réunissant chercheurs et constructeurs de machines s'est créé au début des années 90 ([For93, KLS+94, Ric95]). L'objectif était la définition d'extensions du langage FORTRAN 90 pour permettre le développement de compilateurs paralléliseurs. Le résultat de ce travail est un langage de type Fortran appelé **High Performance Fortran** (HPF).

3.1 Principes de base du langage HPF

HPF possède des extensions permettant de définir l'alignement des données entre elles, leur distribution sur les processeurs, ainsi qu'un certain nombre de constructions spécifiques à la parallélisation de boucles. Ce nouveau langage fondé sur le modèle du parallélisme de données est un pas en avant vers la parallélisation automatique de programmes. La parallélisation est en effet semi-automatique dans la mesure où l'utilisateur assiste le compilateur en insérant dans son programme des informations destinées à faciliter la génération du code parallèle. Le travail du compilateur consiste à produire, à partir du programme écrit en HPF, un unique code qui sera exécuté sur tous les processeurs (modèle SPMD [6]).

Le langage HPF peut être considéré comme une extension du langage Fortran 90 :

- HPF intègre un certain nombre de directives de compilation permettant d'apporter des indications au compilateur. Elles sont insérées dans une ligne de commentaires afin de pouvoir être ignorées par un compilateur Fortran standard. Ces lignes commencent par la suite de caractères **!HPF$**. Leur rôle est uniquement de fournir des stratégies et elles ne modifient en rien la sémantique du programme.
- HPF intègre une nouvelle instruction appelée **FORALL** qui est une généralisation de l'instruction d'affectation de tableau Fortran 90 associée à la condition `WHERE` (voir section 3.3).
- De nouvelles fonctions intrinsèques regroupées dans une librairie spécifique ont également été ajoutées à la spécification HPF. Ces fonctions permettent l'accès à des informations, par exemple le nombre de processeurs, ou l'exécution de certains calculs sur des tableaux, par exemple des réductions.

Remarque 3.1
Le langage HPF, dans sa version originale, n'est pas adapté aux structures de données autres que matricielles, ce qui interdit la prise en compte des problèmes irréguliers. De nombreuses recherches sont en cours sur ce sujet à travers HPF-2 [HPF96, Sch97].

6. Single Program Multiple Data.

3.2 Alignement et distribution des données

La première étape dans le processus de parallélisation d'une application avec HPF consiste à étudier la répartition des données (de type tableau) sur les processeurs (virtuels). Cette information est communiquée au compilateur par le biais de directives. Dans un premier temps, les tableaux sont alignés sur des templates (paragraphe 3.2.2), puis les différents templates sont réparties (paragraphe 3.2.3) sur une grille virtuelle de processeurs (paragraphe 3.2.1). Ce principe général est illustré par la figure 3.1.

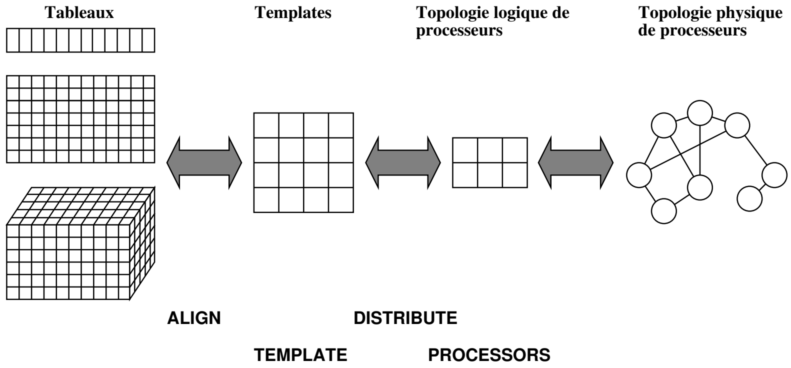


FIG. 3.1 – *Principe général de répartition des données en HPF.*

3.2.1 Topologie logique des processeurs

Afin de conserver une portabilité maximale, la topologie physique (réelle) de la machine parallèle sur laquelle va être exécuté le programme ne doit pas intervenir dans la conception du programme. Cela permettra à un même programme de fonctionner, sans restriction, sur toutes sortes de topologie (anneau, hypercube, tore, etc.). Le compilateur et/ou le support d'exécution se chargeront ensuite d'optimiser les communications pour la topologie physique de la machine.

Les processeurs virtuels manipulés au niveau du programme HPF sont organisés en grilles. Les tailles et dimensions de ces tableaux de processeurs seront spécifiées par l'utilisateur à l'aide de la directive **PROCESSORS**. Plusieurs topologies virtuelles différentes peuvent cohabiter dans un même programme.

```
!HPF$ PROCESSORS, DIMENSION(8)     :: P1
!HPF$ PROCESSORS, DIMENSION(4,2)   :: P2
!HPF$ PROCESSORS, DIMENSION(2,2,2) :: P3
```

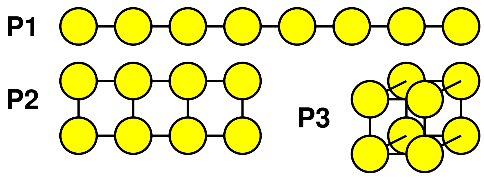


FIG. 3.2 – *Utilisation de la directive PROCESSORS.*

3.2.2 Alignement des données

Le langage HPF offre la possibilité d'aligner les tableaux de données entre eux et ainsi potentiellement de diminuer le nombre de communications entre les processeurs. Pour cela, le programmeur peut soit aligner les tableaux entre eux directement, soit déclarer dans un premier temps un *template*[7], puis aligner chacun des tableaux sur ces templates. Deux directives sont disponibles à cet effet :

- La directive **TEMPLATE** permet de définir un template de dimension quelconque.
  Exemple : **!HPF$ TEMPLATE** T(100,100)

7. Sorte de matrice virtuelle.

(définition d'une matrice virtuelle de dimension 2)

– La directive **ALIGN** permet de spécifier l'alignement d'un tableau sur un template ou sur un autre tableau.

  Exemple : **!HPF$ ALIGN** A(I,*) with B(I)

  (la première dimension du tableau A est alignée sur le vecteur B)

  Exemple : **!HPF$ ALIGN** C(J,I) with T(I,J)

  (Le tableau C est aligné sur le template T - avec inversion des coordonnées)

La sémantique très riche de cette directive offre de nombreuses possibilités ; les tableaux peuvent être rèpliqués, transposés, écrasés, etc. les uns par rapport aux autres.

3.2.3 Distribution des données

La répartition des données sur les processeurs virtuels est spécifiée à l'aide de la directive **DISTRIBUTE**. L'utilisateur peut distribuer soit directement les tableaux de données, soit les templates qu'il a créés. Il existe deux façons de distribuer les données sur les processeurs : la distribution par blocs (**BLOCK**) et la distribution cyclique (**CYCLIC**).

```
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO P2
!HPF$ DISTRIBUTE B(CYCLIC(5)) ONTO P1
```

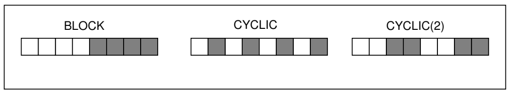


FIG. 3.3 – *Distribution des données en HPF.*

3.3 Parallélisme de données en HPF

Il existe deux façons d'exprimer le parallélisme en HPF. Ce sont la directive **INDEPENDENT** et l'instruction **FORALL**. Chacune d'elles permet d'indiquer au compilateur la possibilité d'exécuter en parallèle certaines opérations.

Remarque 3.2
*Il existe également d'autres directives telles que **NEW** ou **REDUCTION**, qui seront utilisées au chapitre 4. Pour un éventail de toutes ces fonctionnalités du langage HPF, le lecteur est invité à consulter les documents [For93, KLS+94, HPF96].*

3.3.1 La directive INDEPENDENT

La directive **INDEPENDENT** est utilisée par le programmeur pour indiquer au compilateur qu'une boucle peut être exécutée en parallèle. Toutes les itérations de la boucle sont indépendantes les unes des autres et peuvent donc être exécutées dans un ordre arbitraire choisi par le compilateur. Ce dernier n'est pas censé vérifier la validité de cette affirmation et c'est donc au programmeur de contrôler que l'exécution parallèle du programme ne modifie pas la sémantique du programme. En particulier, les dépendances doivent être respectées (voir chapitre 12). La figure 3.4 propose un exemple d'utilisation de cette directive.

3.3.2 L'instruction FORALL

L'instruction **FORALL** est une extension du langage Fortran 90 qui permet d'effectuer des *affectations* de manière parallèle. C'est une généralisation des instructions d'affectation de tableau à laquelle on a ajouté un masque d'exécution. Par exemple, l'instruction de la figure 3.5 définit un ensemble de points actifs dans le domaine d'itération (ici la partie triangulaire supérieure de la matrice) pour lequel l'instruction d'affectation sera exécutée. Tous les membres droits de l'affectation sont calculés, puis on

```
!HPF$ INDEPENDENT, NEW(j)
      do i = 1, n
!HPF$ INDEPENDENT
        do j = 1, m
          A(i,j) = B(i,j)
        enddo
      enddo
```

FIG. 3.4 – *Exemple d'utilisation de la directive* ***INDEPENDENT***.

```
FORALL (I=1:N, J=1:M, I>J) A(I,J) = A(J,I)
```

FIG. 3.5 – *Exemple d'utilisation de l'instruction* ***FORALL***.

effectue toutes les affectations aux membres gauches. Évaluations et affectations peuvent être effectuées dans un ordre quelconque.

La distinction entre l'instruction **FORALL** et la directive **INDEPENDENT** se caractérise par le type d'opérations qu'il est possible d'exécuter en parallèle. L'instruction **FORALL** permet d'effectuer des affectations en parallèle, alors que la directive **INDEPENDENT** autorise l'exécution d'*instructions* complètes en parallèle. Cette distinction est schématisée figure 3.6.

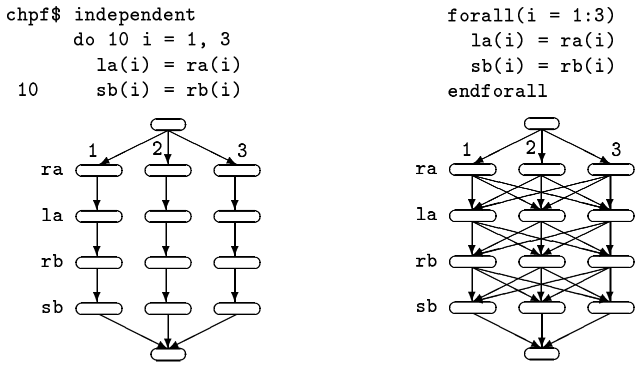


FIG. 3.6 – *Schéma d'exécution de l'intruction* ***FORALL*** *et de la directive* ***INDEPENDENT***.

Chapitre 4

Une expérience avec trois compilateurs HPF

Ce chapitre présente le portage de l'application Onde24 vers le langage High Performance Fortran. Ce travail a été effectué avec divers objectifs :

• En choisissant une application relativement simple et bien adaptée à ce type de parallélisation, mais néanmoins sur une véritable application industrielle, nous cherchons à mesurer le gain en performance pouvant être obtenu. D'autre part, nous souhaitons évaluer la complexité du portage vers HPF pour une application ne posant pas de difficulté majeure pour ce type de parallélisation.

• Connaître l'état d'avancement des travaux réalisés dans le domaine de la compilation de ce langage après quelques années de « lancement » est également important. Ce travail a été diffusé auprès des différents utilisateurs potentiels du parallélisme à l'École des mines de Paris. Nous avons également souhaité partager ces informations avec la communauté scientifique du domaine HPF en présentant notre travail à HUG'98 [Zor98].

• Dans le cadre des travaux réalisés par Fabien Coelho ([Coe96]) pour la mise en place d'un prototype de compilation du HPF, l'obtention de points de comparaison avec d'autres compilateurs est un point clé de notre étude.

• Enfin, ce travail a également pour but de déterminer de nouvelles pistes de travail, tant dans le domaine précis de la compilation du HPF, que concernant des domaines qui lui sont connexes (tels que la génération automatique de code, l'optimisation, etc.).

Notre approche a consisté à mesurer l'écart existant entre les capacités théoriques du langage et la réalité rencontrée lors du portage d'une application industrielle avec différents compilateurs. En effet, un des principaux atouts d'HPF, à l'origine de la définition de ce langage, est la simplicité de son utilisation. Quelques déclarations (topologie, templates, etc.), un alignement des données puis quelques directives devant les boucles devraient théoriquement amener l'utilisateur vers une application parallèle performante. Or, plusieurs problèmes existent en pratique. Les résultats apportés dans ce chapitre montreront en effet qu'il est important d'avoir une bonne connaissance des mouvements de données, donc une bonne connaissance du parallélisme de données en général, et que l'utilisation des outils de debug et de profiling est nécessaire. Par ailleurs, le manque d'homogénéité entre les compilateurs rend encore plus difficile le travail de portage de l'application.

4.1 Portage de ONDE24 vers HPF

L'application ayant été décrite en détail au chapitre 2, nous allons ici nous limiter à la présentation des caractéristiques de l'application utiles à son portage vers HPF. La première information importante à rappeler est que Onde24 est à l'origine une application vectorielle écrite en Fortran 77 ; l'effort de portage du code vers HPF doit, en théorie, être limité. Considérons le noyau de calcul de l'application présenté figure 4.1. Aucune valeur définie par une itération de la boucle n'est utilisée par une autre itération parce que les valeurs de KP et KM sont toujours différentes. Cette boucle peut donc être exécutée en parallèle, ce qui se traduira ici par l'ajoût de la directive **INDEPENDENT**.

```
!HPF$ independent
   [...]
   DO J = 3,NP-2
!HPF$ independent
     DO I = 3, NP-2
       U(I,J,KP) =
  $      2 * U(I,J,KM) - U(I,J,KP)
  $      - V(I,J) * ( 60*U(I,J,KM)
  $      -16*(U(I+1,J,KM) + U(I-1,J,KM)
  $      + U(I,J-1,KM) + U(I,J+1,KM))
  $      + U(I+2,J,KM) + U(I-2,J,KM)
  $      + U(I,J-2,KM) + U(I,J+2,KM))
     ENDDO
   ENDDO
   [...]
```

FIG. 4.1 – *Noyau de calcul de Onde24.*

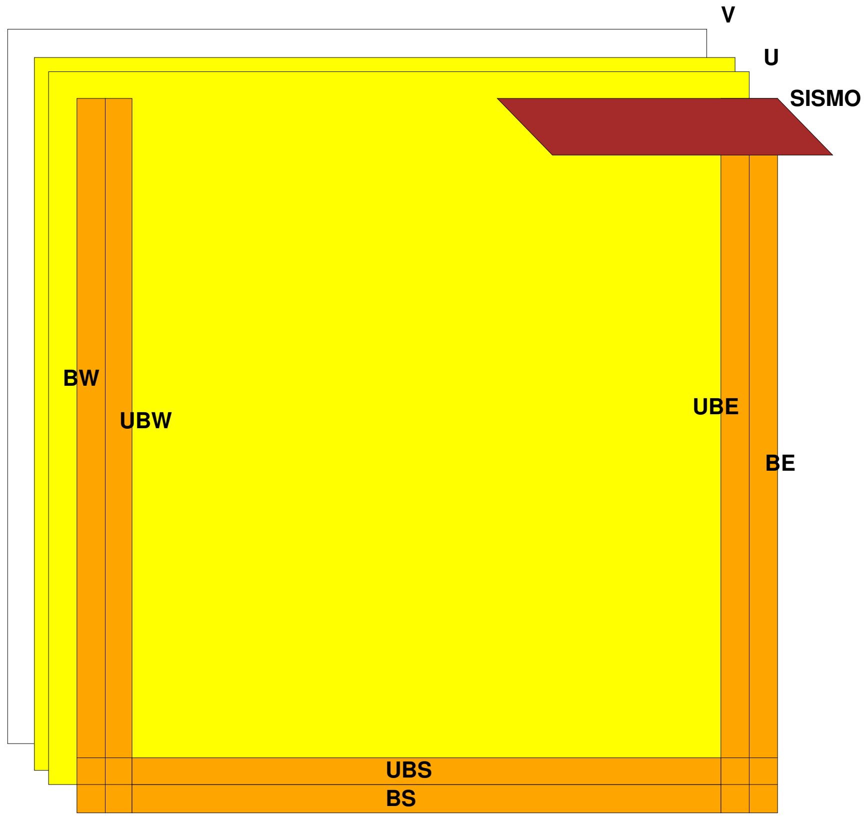
L'alignement et la distribution des données sur les processeurs doivent également être étudiés. Nous devons considérer les tableaux U (la solution) et V (le champ de vitesse), ainsi qu'un certain nombre de tableaux utilisés pour stocker des résultats intermédiaires ou pour calculer la solution dans les coins. La figure 4.2 schématise l'alignement, induit par le problème, entre les principaux tableaux manipulés. La valeur V(I,J) est par exemple lue par le processeur calculant la solution au point U(I,J,?). Ces deux tableaux seront donc alignés entre eux. Notons également que les solutions aux temps t et $t+1$ seront détenues par le même processeur.

Remarque 4.1
Le programme initial utilise des vecteurs temporaires (UINT, B et UB) pour le calcul des solutions aux bords du domaine. Nous les avons découpés et/ou dupliqués afin de permettre une distribution et un alignement de tableaux performants (voir figure 4.2). Le vecteur U est par exemple découpé en 3 vecteurs UW, US et UE pour respectivement les bords ouest, sud et est.

La figure 4.3 présente les principales directives de placement et de distribution qui ont été utilisées pour le portage de Onde24 en HPF. Un template de la taille du domaine considéré est créé, sur lequel seront alignés les différents tableaux. Ce template est distribué par blocs sur une grille de processeurs de dimension 2. Les directives **INDEPENDENT**, **NEW** et **REDUCTION** ont également été utilisées.

4.2 Description des expérimentations

Afin de pouvoir comparer les résultats obtenus, nous avons réalisé les expériences de portage de l'application Onde24 sur une unique plateforme, mais avec plusieurs compilateurs différents.



V: vitesse de propagation de l'onde

U: solution recherchée (pression)

B/UB{W,S,E}: tableaux temporaires

SISMO: enregistrement par les sismographes

FIG. 4.2 – *Onde24 - Le domaine de travail.*

!HPF$ TEMPLATE DOMAIN(NP,NP)
!HPF$ PROCESSORS PE(2,2)
!HPF$ DISTRIBUTE DOMAIN(BLOCK,BLOCK) ONTO PE

!HPF$ ALIGN V(I,J) WITH DOMAIN(I,J)
!HPF$ ALIGN U(I,J,*) WITH DOMAIN(I,J)

!HPF$ ALIGN UBE(I) WITH DOMAIN(I,NP-1)
!HPF$ ALIGN BE(I,*) WITH DOMAIN(I,NP)

FIG. 4.3 – *Onde24 - Alignement et distribution des données.*

4.2.1 Plateforme utilisée

Les expériences ont été réalisées sur le super-calculateur **IBM SP2** de l'École des mines de Paris. Cette machine est destinée à la mise à disposition d'importantes ressources de calcul pour les scientifiques (non-informaticiens) travaillant dans les différents centres de recherche de l'École des Mines de Paris. Elle est composée de 9 nœuds dédiés au calcul parallèle, d'un nœud dédié au travaux interactifs (compilation, édition, debug, etc.) et de deux nœuds utilisés pour lancer des calculs séquentiels demandant des ressources mémoire importantes.

Les processeurs sur les nœuds dédiés au calcul parallèle, issus de la famille IBM RS6000, sont des **P2SC** cadencés à 120MHz [8] ; l'architecture sera décrite au chapitre 7. Un commutateur à haute vitesse, le *High Performance Switch*, à chemins multiples et redondants, assure les communications entre les processeurs. La latence et le débit des communications restent ainsi constants quels que soient le nombre et la position des processeurs, ce qui permet de ne pas se soucier de la topologie du réseau. Le commutateur supporte les protocoles de communication `IP` et `User space`. Les performances du réseau données pour le SP2 de l'École des Mines indiquent pour le mode User space (resp. IP) une latence de 30 (resp. 180) micro-secondes et un débit point-à-point de 100 (resp. 87) méga-octets par seconde. Les deux librairies natives de communications disponibles sur le SP2 sont MPI [SOHL+96] et MPL [9] [IBM95a].

4.2.2 Présentation des 3 compilateurs

Avant de rentrer dans le descriptif de nos expériences, nous allons décrire brièvement les trois compilateurs HPF que nous avons utilisés. Pour plus de détails sur ces compilateurs, le lecteur se reportera aux références bibliographiques proposées.

IBM xlhpf

XLHPF est un compilateur natif proposé par IBM [IBM96a, IBM96b] en complément de sa gamme de compilateurs Fortran (`xlf77` et `xlf90`). Ce compilateur **commercial** se distingue des deux autres compilateurs utilisés (voir ci-dessous) par la génération directe d'un code assembleur pour les processeurs de la gamme Power [10]. La version de XLHPF que nous avons utilisée est la 1.2. Les librairies de communication MPI et MPL peuvent être utilisées indifféremment.

Remarque 4.2
La version 1.2 *n'est pas la dernière version disponible sur le SP2. Cependant, la version* 1.3 *n'inclut aucune nouvelle fonctionnalité HPF et se contente d'offrir une compatibilité complète avec le compilateur Fortran* `xlf77 version 5.1`.

Adaptor

ADAPTOR (Automatic DAta Parallelism translaTOR) est un outil de compilation du domaine public développé par Thomas Brandes en Allemagne au GMD/SCAI [BZ94, Bra94]. Il effectue une transformation source-à-source (Figure 4.4) avec en entrée un programme HPF et en sortie un programme SPMD Fortran 90 qui sera exécuté par chaque nœud. Un support d'exécution appelé **DALIB** (Distributed Array LIBrary) a été développé dans le but d'y placer toutes les fonctions de communication, transposition, redistribution, réduction, etc. Cette bibliothèque ayant été développée pour pratiquement toutes les machines parallèles classiques (MIMD), le programme généré peut être exécuté sur toutes ces machines sans devoir reconstruire le source SPMD ; seule la library DALIB est changée.

Nous avons utilisé la version 6.1 de ce compilateur, disponible sur 16 machines parallèles (UNIX) différentes. Les supports de communication proposés pour le SP2 sont MPL, MPI, et PVM.

8. Les nœuds utilisés pour le calcul séquentiel ont eux une fréquence de 160MHz.
9. Abandonnée par IBM au profit de MPI.
10. Gamme qui comprend les processeurs Power1, Power2, et P2SC

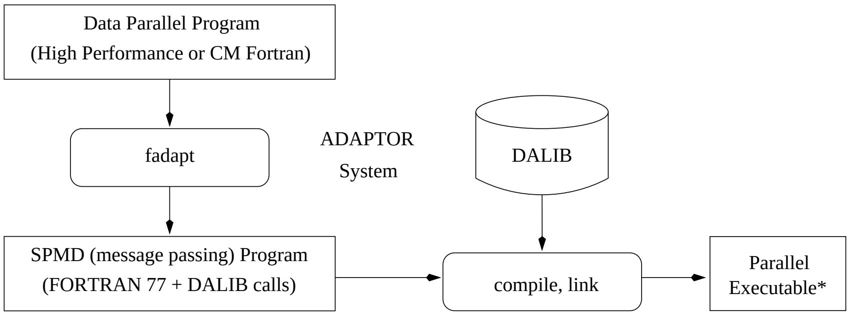


FIG. 4.4 – *Une vue d'ensemble d'*ADAPTOR.

HPFC

HPFC est un prototype de compilateur développé par Fabien Coelho [Coe96] de l'École des Mines de Paris. Cet outil a été développé dans le cadre du projet PIPS (présenté dans l'annexe B) et s'appuie donc sur les analyses interprocédurales de ce dernier. Le code généré par HPFC est fondé sur un modèle *host-node*[11]. La structure générale de ce compilateur est schématisée figure 4.5. Les supports de communication PVM et MPI sont supportés.

Remarque 4.3
Il existe bien évidemment beaucoup d'autres compilateurs commerciaux ou non qu'il aurait été également intéressant d'évaluer. Le tableau 4.1 dresse une liste non exhaustive de compilateurs (les commerciaux sont notés en gras et ceux du domaine public en italique). Cependant, pour des raisons de temps et de moyens, tous les compilateurs commerciaux n'étant pas fournis sur notre SP2, nous nous sommes limités à ceux mentionnés ci-dessus. Le lecteur trouvera dans la littérature d'autres travaux proposant une comparaison de plusieurs compilateurs [GBC+93, YFA97].

Compilateur	Conception
EPPP	CRIM Montreal
Fortran D	Rice University
Pandore	IRISA
PARADIGM	University of Illinois
DEC Fortran 90 Compiler	DEC
HPF Plus	N.A.Software
pghpf	Portland Group Inc.
VAST-HPF	Pacific-Sierra Research Corp.
xHPF	Applied Parallel Research Inc.

TAB. 4.1 – *Quelques compilateurs HPF commerciaux ou domaine public.*

4.3 Problèmes rencontrés

Plusieurs problèmes ont été rencontrés au cours du portage de Onde24 vers HPF. Ils concernent la spécification de HPF, les techniques de compilation utilisés et les sous-ensembles de la spécification HPF supportés par les compilateurs.

11. Le hôte (host) est chargé des entrées-sorties et de la gestion des processus. Les nœuds sont en charge des calculs.

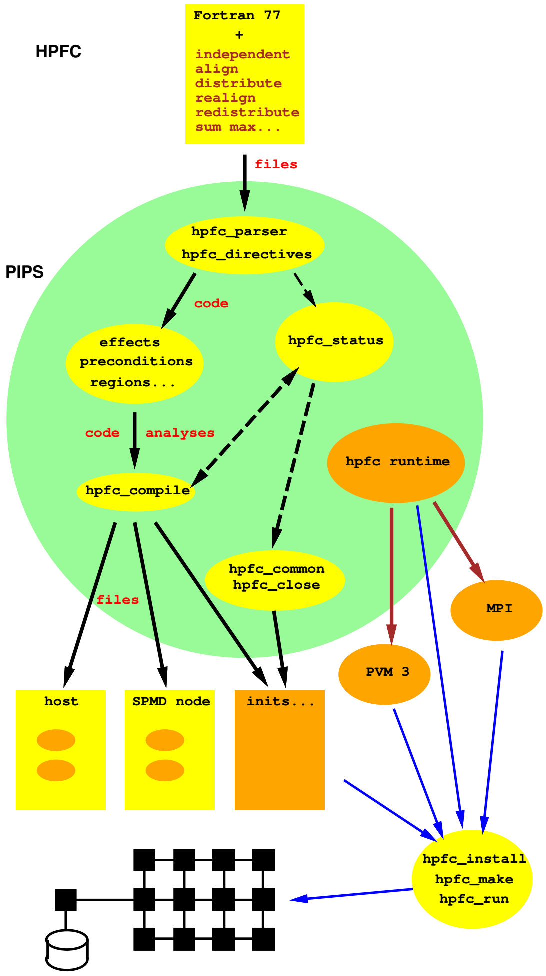


FIG. 4.5 – *Une vue d'ensemble de* HPFC.

4.3.1 Compilation des entrées-sorties

Onde24, comme la majorité des applications scientifiques, effectue des calculs sur une quantité importante de données. L'application commence donc par une phase de lecture de données sur le disque et termine par une phase d'écriture des résultats. Le nid de boucle présenté figure 4.6 est extrait du programme original ; la vitesse de propagation de l'onde pour tous les points du domaine est lue dans un fichier (notons que les accès aux données ne sont pas bien adaptés au cache).

```
Do I = 1,NP
   Do J = 1,NP
      READ (3,*) V(I,J)
   Enddo
Enddo
```

FIG. 4.6 – *Entrées-sorties dans Onde24.*

Pour ce nid de boucle, les compilateurs XLHPF et ADAPTOR génèrent un programme parallèle effectuant des communications à l'**intérieur** de la boucle (voir figure 4.7). Une communication globale (de type *broadcast*) entre tous les processeurs est donc requise pour tous les points du domaine. Les performances du programme sont considérablement réduites, anéantissant tout effet positif attendu de la parallélisation des calculs.

```
do I=1,NP
   do J=1,NP
      if (DALIB_pid() .eq. 1) then
         read (3,*) TMP7
      end if
      call DALIB_broadcast (TMP7,8,1)
      if (DALIB_is_local(V_DSP,2,J)) then
         if (DALIB_is_local(V_DSP,1,I)) then
            V(V_ZERO+J*V_DIM1+I) = TMP7
         end if
      end if
   end do
end do
```

FIG. 4.7 – *Extrait du code généré par* ADAPTOR.

Le compilateur HPFC est quant à lui capable de reconnaître une communication de type entrées-sorties. Or, grâce à une analyse des régions de tableaux, dont les détails sont donnés au chapitre 5, il peut prouver que les communications à l'intérieur de la boucle sont inutiles. HPFC fonctionne sur un modèle *host-node* ; le *host* lit les données sur le disque puis les envoie d'un seul bloc à tous les *nodes* (une seule communication par nœud).

Outre la solution proposée par HPFC qui nécessite des analyses très fines du programme, le programmeur dispose de plusieurs solutions à ce problème de communications inutiles.

Implied Do

Cette première solution consiste à utiliser la construction **Implied Do** du langage Fortran à la place du nid de boucles employé dans le programme original. Le code devient alors celui présenté figure 4.8. Cette modification permet aux compilateurs ADAPTOR et XLHPF de ne générer qu'une communication globale avant et après la lecture sur le disque de la vitesse de propagation pour tout le domaine.

```
READ (3,*) ((V(I,J), J=1, NP), I=1, NP)
```

FIG. 4.8 – *Utilisation de la construction Implied-Do.*

Accès de tableaux à la Fortran 90

La seconde solution est fondée sur l'écriture des accès de tableaux au format Fortran 90. Là encore, la transformation du programme permet d'éliminer les communications inutiles. La syntaxe utilisée est présentée figure 4.9. Notons que la disposition des tableaux en mémoire sous la forme « colonne d'abord » [12] impose la transposition de la matrice contenant les valeurs lues. Cette manipulation peut être appliquée soit au niveau du fichier de données, soit durant l'exécution du programme.

```
READ (3,*) V(1:NP,1:NP)
V =  TRANSPOSE(V)
```

FIG. 4.9 – *Utilisation des accès de tableaux à la Fortran 90.*

Nous avons rencontré un autre problème concernant la gestion des entrées-sorties dans un code HPF. En effet, étant donné le grand volume de données généré par un programme, il est souvent intéressant pour l'interprétation des résultats de pouvoir utiliser un affichage graphique approprié. L'usage des couleurs et des courbes superposées permet en effet de visualiser et donc interpréter les résultats plus facilement. Pour ce faire, l'approche la plus courante consiste à faire appel à une librairie dédiée à ce type d'affichage depuis le programme considéré. Or, si ce procédé ne pose aucun problème en calcul séquentiel, son introduction dans un programme écrit en HPF pose quelques difficultés puisqu'**un seul** des processeurs doit appeler la procédure d'entrées-sorties.

L'objectif d'Onde24 étant de simuler la propagation d'une onde dans un milieu hétérogène (du point de vue de la vitesse de propagation), il est intéressant de pouvoir visualiser l'évolution au cours du temps de la solution en certains points du domaine (voir figure 4.10). Les trois procédures qui sont utilisées dans Onde24 à cet effet sont les suivantes :

- call **open display**(X_size,Y_size,window)
  ouverture d'une fenêtre dédiée à l'affichage du sismographe.
- call **build display**(SISMO(1,1),record,NT,window)
  affichage des données dans la fenêtre.
- call **close display**(window)
  fermeture de la fenêtre.

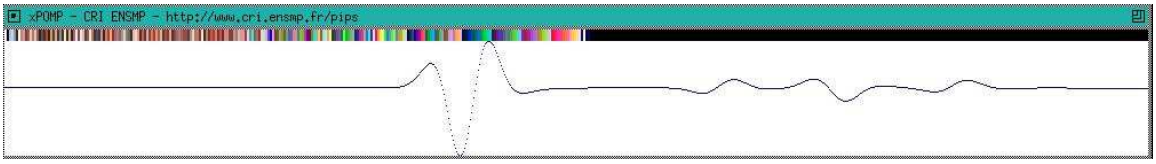


FIG. 4.10 – *Évolution de la solution enregistrée par un capteur.*

La spécification du langage HPF fournit une solution bien adaptée à ce problème. Le programmeur peut en effet spécifier que la procédure est écrite en Fortran 77, et qu'elle ne doit être exécutée que sur un unique processeur. La syntaxe utilisée est présentée figure 4.11.

Cette fonctionnalité du langage HPF est bien supportée par ADAPTOR. En revanche, ni XLHPF ni HPFC ne permettent l'utilisation de cette syntaxe particulière. Notons toutefois que HPFC fournit un moyen d'intégrer un appel à une fonction externe d'entrées-sorties en proposant une directive spécifique appelée !FCD$ IO [Coe96].

12. column major order

```
INTERFACE
   EXTRINSIC(F77_SERIAL) SUBROUTINE
&     open display(X, Y, window)
      INTEGER X, Y, window
      END SUBROUTINE open display
END INTERFACE
```

FIG. 4.11 – *Syntaxe à utiliser pour l'appel d'une fonction externe.*

4.3.2 Fonctionnalités non supportées

Nous avons également rencontré des problèmes suite à l'utilisation de fonctionnalités du langage HPF ou de ses extensions qui ne sont pas supportées par les trois compilateurs.

Réductions

L'application Onde24 calcule à plusieurs reprises dans le code des valeurs minimales ou maximales sur une matrice (voir l'extrait de code de la figure 4.12). Sans aucune information complémentaire sur la nature du calcul effectué, le compilateur génère des communications entre les processeurs dans la boucle la plus interne. C'est pourquoi la norme HPF contient une directive, appelée **REDUCTION**, qui permet de signaler au compilateur la nature particulière de la boucle, et ainsi générer une boucle parallèle suivie d'une phase de « collecte » des résultats locaux.

```
!HPF$ INDEPENDENT, REDUCTION(VMIN)
      Do J=1,NP
!HPF$ INDEPENDENT
         Do I=1,NP
            VMIN = MIN(VMIN,V(I,J))
         Enddo
      Enddo
```

FIG. 4.12 – *Opération de réduction.*

Cette directive n'est pas disponible dans la version 1.2 du compilateur XLHPF. Nous avons donc eu une nouvelle fois recours à l'usage du Fortran 90 avec l'utilisation de la fonction intrinsèque[13] `minval`. Le nid de boucle de la figure 4.12 est remplacé par l'appel de fonction VMIN=`minval`(V).

ON HOME directive

L'application comporte un grand nombre de calculs de petite taille, placés en dehors des boucles, et destinés au traitement des initialisations et conditions aux bords du domaine. La figure 4.13 présente un extrait de code qui calcule l'initialisation du coin sud ouest. Cette fois encore, le compilateur a besoin d'informations pour éviter la génération de communications inutiles (et coûteuses). Il faut donc indiquer au compilateur que les variables C2 et W sont privées (utilisées uniquement pour stocker le résultat de calculs locaux).

Les directives **ON HOME** et **NEW** ont été créées à cet effet. **ON HOME** permet, comme le montre la figure 4.14, d'indiquer au compilateur que le calcul doit être effectué sur le processeur détenant la valeur V(NP,1) (qui contient également BS(1,1)). La directive **NEW** permet de signaler que les deux variables C2 et W sont privées.

La directive **ON HOME** n'est pas intégrée dans la spécification standard de HPF. Ce n'est en effet qu'une extension approuvée par le comité de mise en place de HPF2. Si elle est supportée par ADAPTOR, elle ne l'est en revanche ni par XLHPF ni par HPFC. Cependant, ce dernier permet d'exploiter

13. intrinsic

```
          C2      = 4/((1/V(NP,1))+(1/V(NP,2)) +
&                   (1/V(NP-1,1))+(1/V(NP-1,2)) )
          W       = C2*DELTAT
          BS(1,1) = 2*W/(4*W+3*H)
          V(NP,1) = C2
```

FIG. 4.13 – *Des calculs locaux dans Onde24.*

```
!HPF$ ON HOME (V(NP,1)), NEW(C2,W) BEGIN
      C2      = 4/((1/V(NP,1))+(1/V(NP,2)) +
&                 (1/V(NP-1,1))+(1/V(NP-1,2)) )
      W       = C2*DELTAT
      BS(1,1) = 2*W/(4*W+3*H)
      V(NP,1) = C2
!HPF$ END ON
```

FIG. 4.14 – *Utilisation de la directive ON HOME.*

cette fonctionnalité par l'utilisation d'une directive spécifique !FCD$ LOCAL, NEW(), antérieure à la définition de **ON HOME** dans HPF2.

4.3.3 Portage de boucles optimisées

Dans le cadre des expériences menées pour améliorer l'évaluation des expressions sur les processeurs modernes (cf. partie III), une version fortement optimisée à la main du noyau de Onde24 a été écrite. Un grand nombre d'optimisations ont été appliquées avec notamment un déroulage de boucle, du software pipelining[14] et une phase d'atomisation (créant un grand nombre de variables privées). Partant du nid de boucle initial ne contenant qu'une instruction, nous avons écrit une nouvelle version contenant environ 100 lignes de code.

Ce code fortement optimisé est donc beaucoup plus long mais également plus performant que la version originale. Un gain d'un facteur 2 a été mesuré pour cette version séquentielle optimisée par rapport à la version séquentielle initiale. Or, en raison de la complexité importante de ce nouveau code, aucun des trois compilateurs HPF n'a réussi à générer un code contenant une boucle parallèle autour du noyau de calcul. Les performances du programme HPF sont donc véritablement restées médiocres ! Les deux principales sources de difficultés rencontrées sont les suivantes :

- Les variables introduites par la phase d'atomisation doivent être déclarées en tant que variables privées (par la directive **NEW**).
- Le déroulage de boucle (ici d'un facteur 4) pose des problèmes pour la génération des indices locaux des boucles. Le schéma *owner computes rule*[15] au niveau des itérations ne peut plus être respecté, car une itération de la boucle écrit des données portées par différents processeurs.

4.3.4 Mesures du temps

Le premier objectif de la parallélisation est la diminution du temps d'exécution du programme. Pour pouvoir évaluer le gain obtenu par cette transformation, il est donc nécessaire de pouvoir mesurer correctement les temps d'exécution. De même, pour ce travail qui vise, en autres choses, à comparer les résultats obtenus pour les trois compilateurs étudiés, nous avons cherché à obtenir des informations précises sur les temps d'exécution. Or, mesurer le temps dans un programme parallèle est problématique, puisqu'il existe plusieurs types de mesure. Supposons que nous cherchions à mesurer le temps d'exécution pour

14. Technique consistant à intercaler les calculs de plusieurs itérations de boucles pour tenter d'éviter les « trous » dans le flot d'instructions.

15. Le processeur qui détient la donnée à écrire effectue les calculs.

une itération (noté t) et pour n itérations (noté T). Aucun des deux types de mesure utilisés en général ne permet d'obtenir des résultats « exacts » :

- Mesure **sans** synchronisation (figure 4.15 à gauche)

  Le temps d'exécution pour une itération (t) est *sous-estimé*. En effet, comme le montre la figure 4.15, seule la durée d'exécution pour un processeur (celui qui mesure le temps) est prise en compte.

  Le temps d'exécution des n itérations s'approche de la valeur exacte si le nombre d'itérations est important et si tous les processeurs effectuent un travail de durée équivalente.
- Mesure **avec** synchronisation (figure 4.15 à droite)

  Le temps d'exécution nécessaire pour effectuer toutes les opérations d'une itération est mesuré correctement grâce à une synchronisation entre les processeurs.

  Par contre, le temps utile pour exécuter n itérations (T) est *sur-estimé*. En effet, l'introduction de points de synchronisation entre les processeurs peut dans certains cas ralentir considérablement le programme.

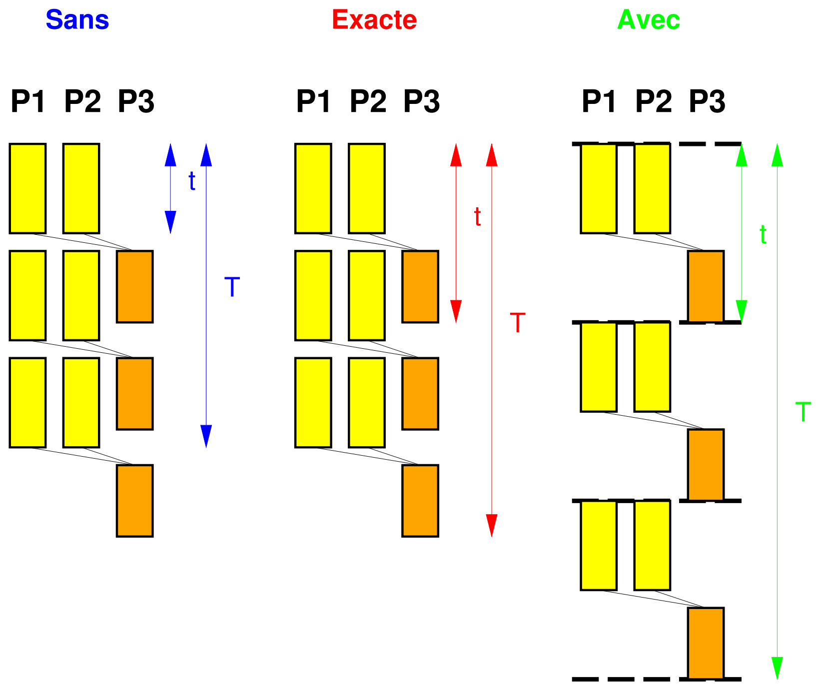


FIG. 4.15 – *Mesure du temps dans un programme parallèle.*

Les trois compilateurs que nous avons utilisés pour ces expériences ont des comportements bien différents en ce qui concerne cette mesure du temps.

- ADAPTOR et XLHPF utilisent la fonction intrinsèque `system_clock`. Les mesures du temps sont donc données sans synchronisation.
- HPFC fait appel à deux directives spécifiques (timeon - timeoff) qui implémentent une mesure du temps avec synchronisation entre les processeurs.

Aucune des deux approches proposées n'est satisfaisante puiqu'elles donnent des résultats potentiellement faux (sans synchronisation) ou inutile (avec synchronisation). Il serait par ailleurs bénéfique d'ajouter à la spécification du langage le type de mesure que les compilateurs doivent implémenter afin de faciliter les comparaisons.

Une mesure exacte sans perturbation du flot d'exécution (donc sans ajouter de synchronisation) est assez difficile à mettre en place. Une solution assez satisfaisante consiste à dédier un processeur à la mesure du temps (le nœud hôte). Ce dernier est chargé de collecter les mesures « locales » prises par chacun des processeurs, puis d'effectuer la synthèse de mesures pour calculer le temps « exact ». Tous les processeurs doivent évidemment disposer d'une horloge commune ! L'utilisation de communications non-bloquantes (par exemple avec MPI_ISEND, MPI_IRECV et MPI_WAIT) permet de limiter les risques d'ajouter une synchronisation entre les processeurs.

La figure 4.16 donne une idée du code à générer pour l'hôte et pour les nœuds. Chaque nœud envoie la valeur courante de l'horloge avant et après l'exécution de l'extrait de code dont on cherche à mesurer le temps d'exécution. L'hôte reçoit toutes les données et calcule le temps écoulé. La figure 4.16 illustre cette technique. On cherche dans cet exemple à mesurer le temps d'exécution d'un calcul pipeliné (donc avec démarrage décalé des différents processeurs). Ce calcul est effectué plusieurs fois dans le programme (avec des valeurs différentes par exemple). L'introduction d'une synchronisation avant et après ce calcul pipeliné augmenterait donc le temps d'exécution total du programme (pouvant être lui aussi mesuré par ailleurs). L'utilisation de communications non-bloquantes entre les processeurs permet de ne pas introduire des moments d'attente mutuelle entre les nœuds et l'hôte.

```
CODE EXÉCUTÉ PAR CHAQUE NOEUD.
==============================
  Initier envoi non-bloquant Tstart_i  (MPI_ISEND)

  // Exécution du code

  Initier envoi non-bloquant Tstop_i (MPI_ISEND)
  Terminer envoi non-bloquant Tstart_i (MPI_WAIT)
  Terminer envoi non-bloquant Tstop_i (MPI_WAIT)

CODE EXÉCUTÉ PAR L'HÔTE.
=========================
  Pour tous les noeuds
    Initier réception non-bloquante Tstart_i (MPI_IRECV)
    Initier réception non-bloquante Tstop_i (MPI_IRECV)
  FinPour
  Pour tous les noeuds
    Terminer réception non-bloquante Tstart_i (MPI_WAIT)
  FinPour
  Calculer Tstart=min(Tstart_i)
  Pour tous les noeuds
    Terminer réception non-bloquante Tstop_i (MPI_WAIT)
  FinPour
  Calculer Tstop=max(Tstop_i)
  Calculer T = (Tstop - Tstart)
```

FIG. 4.16 – *Mesure du temps - modèle host-node avec communications non-bloquantes.*

Remarque 4.4
Cette solution présente l'inconvénient (comme les synchronisations) d'introduire des communications supplémentaires (au démarrage et à l'arrêt du « chrono ») qui augmentent le temps d'exécution. De plus, cette technique impose l'utilisation d'un processeur en tant que hôte, ce qui n'est pas très rentable puisqu'il ne participe pas aux autres calculs.

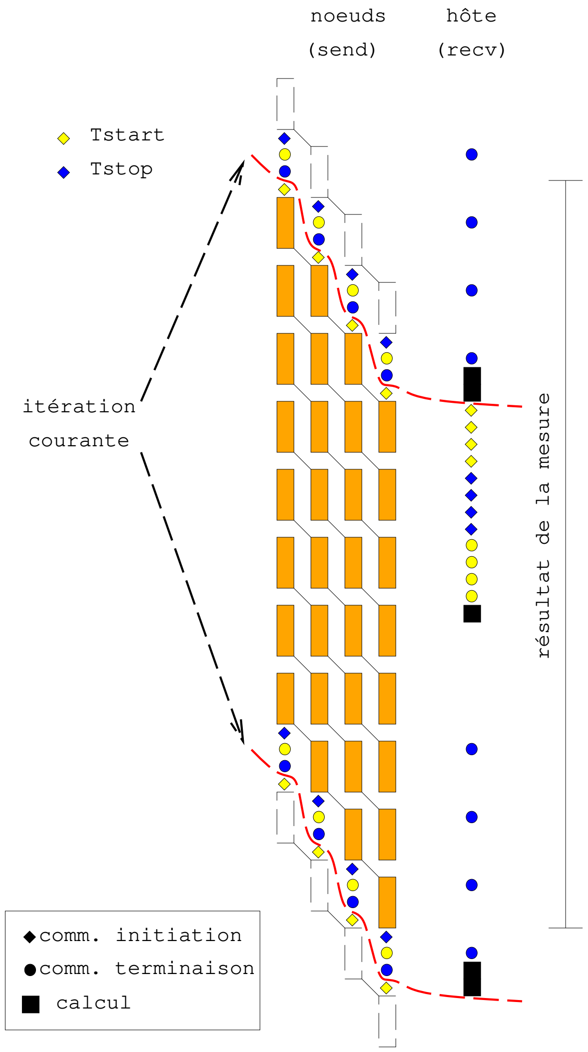


FIG. 4.17 – *Mesure du temps à l'aide de communications non-bloquantes.*

4.4 Résultats - Performances

Les temps d'exécution des versions parallèles de Onde24 issues des trois compilateurs évalués sont données figure 4.18 pour un SP2 avec 4 processeurs. Notons que ce sont les résultats obtenus après la phase de correction des problèmes mentionnés ci-dessus (les premiers résultats obtenus étant très mauvais, avec une hiérarchie différente entre les 3 compilateurs). Le meilleur résultat a été obtenu avec le compilateur XLHPF qui nous a permis d'atteindre 206 MFLOP/s, soit un gain de 2,5 sur 4 processeurs[16]. Le compilateur HPFC avec un gain de 2 pour 165 MFLOP/s est suivi par le compilateur ADAPTOR qui génère une version parallèle donnant 112 MFLOP/s pour un gain de 1,4.

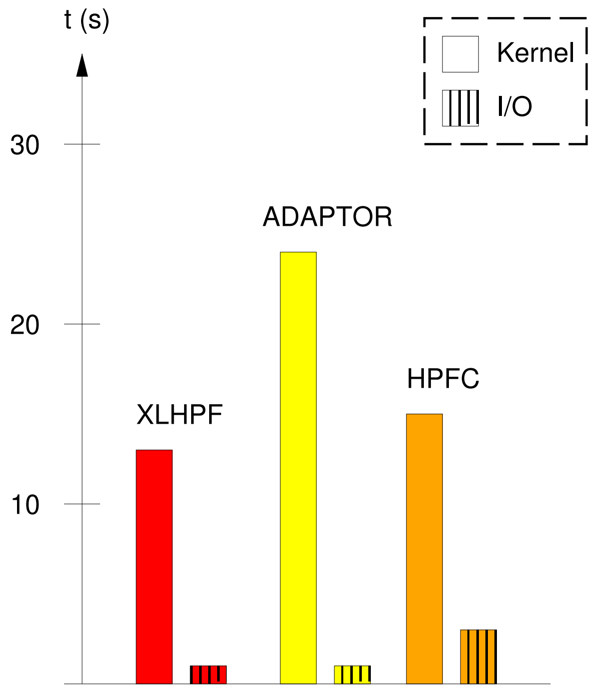


FIG. 4.18 – *Étude de performances.*

Les entrées-sorties sont effectuées plus efficacement avec les compilateurs XLHPF et ADAPTOR qu'avec le compilateur HPFC (voir résultats figure 4.18). Ceci est dû au choix de conserver, dans la version du programme compilée avec HPFC, le code initial ne contenant pas les instructions IMPLIED-DO. Cela pourrait être fait et réduirait ce défaut de performances.

Les gains obtenus pour l'exécution en parallèle de Onde24 sur 4 processeurs ne sont pas très élevés (2,5 maximum) pour une application parfaitement bien adaptée à ce type de parallélisation. La part importante prise par les entrées-sorties dans cette application explique en partie ce phénomène. Comparé à celui obtenu par optimisation manuelle du code (facteur 1,9), le gain d'un facteur 2,5 mesuré pour l'exécution parallèle de ce programme sur 4 processeurs est donc relativement faible.

Remarque 4.5
Les mesures de performances donnies ci-dessous ont iti effectuies ***avec*** *synchronisation, sans boucle englobante. L'estimation du temps peut donc jtre considirie comme trhs proche de la rialiti.*

L'effort de portage en HPF est également un paramètre important que nous avons souhaité mesurer. En effet, si le compilateur XLHPF est celui qui nous a permis d'atteindre les meilleures performances, il nous a également demandé le plus gros effort de portage. La performance est donc meilleure, mais pour

16. Les résultats indiqués ont été obtenus pour des jeux de données avec NP=400 et NT=1200

un coût de portage plus important. De même, le compilateur ADAPTOR est celui qui s'est avéré le plus facile d'utilisation, malgré des performances moindres au bout du compte. La figure 4.19 schématise l'effort de portage réalisé pour les trois compilateurs. Les valeurs indiquées schématisent la quantité de problèmes « majeurs » que nous avons dû résoudre (résumés par le tableau 4.2).

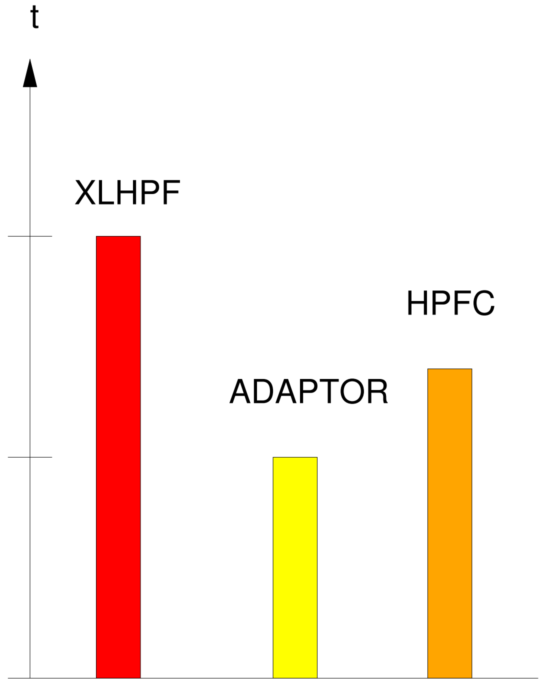


FIG. 4.19 – *Effort de portage.*

	XLHPF	ADAPTOR	HPFC
Entrées-sorties	+	+	
F77-Serial SUBROUTINE	+		+
Réductions	+		
ON HOME	+		+

TAB. 4.2 – *Problèmes majeurs à résoudre lors du portage.*

4.5 Conclusion

Onde24 est une application dont le portage vers HPF devrait être « trivial » ; c'est en effet un programme intrinsèquement parallèle (peu de phases de communications). Nous avons effectivement réussi ce portage de Fortran 77 vers HPF avec des performances raisonnables, et pour un effort de portage réduit compte tenu de celui qui aurait été nécessaire avec une programmation par passage de messages.

Cependant, le portage d'une application vers HPF n'est pas aussi facile que l'on pourrait l'espérer. Tout d'abord, nous avons dû à plusieurs reprises transformer le code initial pour adapter l'application aux spécificités des compilateurs ; l'effort de portage est devenu ainsi très dépendant du compilateur utilisé. Cela nous a donc contraints à maintenir différentes versions du programme HPF, chacune adaptée à un compilateur. Le programme devient donc difficile à maintenir et peu portable (ce qui va à l'encontre des objectifs du langage HPF). De même, le passage par Fortran 90 pour résoudre certains problèmes est problématique si l'utilisateur souhaite conserver la version séquentielle en Fortran 77.

D'autre part, pour obtenir un programme parallèle qui, dans un premier temps, s'exécute puis atteint des performances acceptables, nous avons eu recours, à de nombreuses reprises, aux outils de debug et de profiling disponibles sur le SP2, notamment pour connaître l'origine des mauvaises performances initiales.

Remarque 4.6
Il est important de noter également que l'utilisation de la syntaxe Fortran 90 pour XLHPF *et* ADAPTOR *a participé à la dégradation des performances obtenues au final. En effet, le compilateur Fortran 90 s'est montré moins performant que le compilateur Fortran 77, en raison de la plus grande complexité de Fortran 90 et du manque de maturité des compilateurs. Cette constatation a fait l'objet de plusieurs témoignages allant dans le même sens lors de la conférence HUG'98.*

Nos expériences nous amènent à deux principales constatations :

- Certaines extensions *approuvées* mais *non intégrées* dans la spécification de la norme HPF2 [HPF96] se sont avérées véritablement utiles en pratique. C'est le cas notamment de la directive ON HOME.
- Les travaux menés dans le domaine de la compilation du HPF suivent actuellement deux stratégies distinctes. Certains compilateurs mettent en place de nouvelles techniques pour tenter d'augmenter au maximum les performances pour un ensemble « réduit » d'applications. D'autres s'attachent plus particulièrement à suivre avec attention les évolutions de la spécification du langage pour fournir un compilateur complet mais peu performant. Or, il semble que mener de front ces deux stratégies est une tâche difficile que peu d'équipes réussissent à réaliser. Les performances sont donc souvent atteintes au détriment de la facilité de portage (et vice-versa).

Chapitre 5

Compilation des Entrées-Sorties à l'aide des effets et régions de tableaux

Les applications de calcul scientifique manipulent pour la plupart de grands volumes de données. Ces dernières sont lues en début d'application puis utilisées par le noyau de calcul ; les résultats sont en général ensuite enregistrés sur un support magnétique en fin d'exécution. Outre la parallélisation des calculs, la compilation des langages data-parallèles tels que High Performance Fortran requiert donc une bonne gestion des entrées-sorties. Le travail présenté dans ce chapitre concerne la compilation des *communications* liées aux entrées-sorties pour HPF.

5.1 Présentation du problème

Pour clarifier l'exposé des techniques utilisées, nous appellons *hôte* le nœud chargé des entrées-sorties ; tous les autres sont appelés *nœuds*. Cependant, le travail présenté ne se limite pas au modèle « hôte-nœuds » et pourrait être utilisé dans le cadre d'entrées-sorties parallèles.

Les données écrites par une application (affichage sur l'entrée standard, écriture sur disque) doivent être dans un premier temps collectées sur le processeur chargé d'effectuer les entrées-sorties. De la même manière, ce dernier doit permettre la mise à jour, sur les nœuds, des données qui sont lues. Une solution couramment proposée dans la gestion des entrées-sorties consiste à déterminer l'ensemble de données utiles à chaque nœud, puis à générer les communications adéquates. Considérons l'exemple de la figure 5.1. Une matrice A de dimension 2 est distribuée sur une grille de 4×2 processeurs. L'instruction d'entrées-sorties qui se trouve dans le nid de boucles permet la lecture de la partie supérieure gauche de la matrice A (voir figure 5.2). Les travaux réalisés par Fabien Coelho [Coe95], fondés sur une formulation mathématique à base de polyèdres, permettent de décrire les zones de données à communiquer. Les accès aux éléments de tableaux sont ici caractérisés par le type d'accès (écriture ou lecture) et l'information sur la zone de donnée accédée (toute la matrice ou seulement un sous-ensemble). Pour la matrice A, l'analyse indique par exemple un accès en écriture sur un sous-ensemble des données.

Les expériences décrites au chapitre 4 ont montré qu'un grand nombre de communications et de synchronisations inutiles sont générées par les compilateurs suite à une analyse insuffisante des entrées-sorties. La définition plus précise des zones de données à communiquer permet de réduire le *volume* de communications [Coe95]. Or, en analysant également l'utilisation qui est faite de ces données, il est également possible de réduire le *nombre* de communications. Par exemple, si les données écrites par l'hôte ne sont pas utilisées par les nœuds, la mise à jour n'est pas nécessaire. Pour l'exemple de la figure 5.1, les analyses du prototype de compilateur HPFC (cf. chapitre 4) permettent de ne générer qu'une seule communication pour la mise à jour de la partie supérieure gauche de la matrice sur les nœuds. Cependant, l'hôte écrit également les variables i et j ; des communications seront donc générées pour mettre à jour les « nou-

```
      program triangle
      real A(30,30)
!hpf$ template T(68,30)
!hpf$ align A(i,j) with T(2*i,j)
!hpf$ processors P(4,2)
!hpf$ distribute T(block,cyclic(5)) onto P
      read *, m
      do 1 j=3, m
         do 1 i=3, m-j+1
 1          read *, A(i,j)
      end
```

FIG. 5.1 – *Exemple d'entrées-sorties dans un programme HPF.*

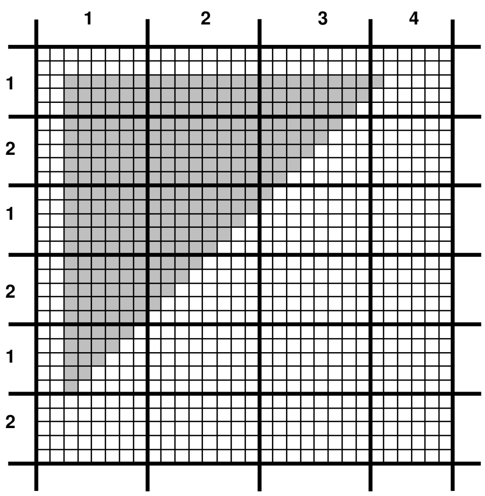


FIG. 5.2 – *Sous-ensemble des données écrites par l'instruction d'entrées-sorties pour* $m = 27$

velles » valeurs de i et j sur les nœuds. Or, ces communications ne sont nécessaires que dans les cas (rares en pratique) où ces variables sont réellement utilisées dans la suite du programme.

Considérons maintenant les entrées-sorties effectuées par le petit programme de test présenté figure 5.3. Ce morceau de code effectue simplement une lecture sur disque suivie d'un affichage sur la sortie standard de tous les éléments d'une matrice.

Le code SPMD généré par le compilateur HPF ADAPTOR (cf. chapitre 4) est présenté figure 5.4. Une communication globale de type broadcast est effectuée entre le nœud chargé des entrées-sorties et les autres nœuds pour diffuser les éléments lus par l'instruction `read`. Une deuxième communication globale (entre le nœud qui possède la donnée A(i,j) et tous les autres nœuds) a lieu pour mettre à jour les données avant l'affichage par l'hôte.

Grâce à ses analyses, le compilateur HPFC ne génère aucune communication à l'intérieur de la boucle (voir figure 5.5) ; le nid de boucles dans sa globalité (l'entête et les deux instructions) est considéré comme une instruction d'entrées-sorties. En revanche, du fait des accès en écriture aux éléments de tableau, le compilateur génère des communications pour mettre à jour les données sur les nœuds. Or, comme les données ne sont pas utilisées par la suite, ces communications sont inutiles.

```
      PROGRAM io
      integer i,j
      parameter (n=10)
      real*8 A(n,n)

!hpf$ processors P(number_of_processors())
!hpf$ distribute A(BLOCK,*) onto P

      open(unit=1,file='data')
      do j=1,n
         do i=1,n
            read (1,*) A(i,j)
            print *, A(i,j)
         enddo
      enddo
      close(unit=1)

      end
```

FIG. 5.3 – *Un petit programme test pour les entrées-sorties.*

```
         do J=1,N
            do I=1,N
               if (DALIB_pid() .eq. 1) then
                  read (1,*) TMP1
               end if
! First node broadcasts replicated data
               call DALIB_broadcast (TMP1,8,1)
               if (DALIB_is_local(A_DSP,1,I)) then
                  A(A_ZERO+J*A_DIM1+I) = TMP1
                  TMP2 = A(A_ZERO+J*A_DIM1+I)
               end if
! Owner broadcasts replicated data
               call DALIB_find_owner (A_DSP,1,1,I)
               call DALIB_node_get (TMP2,8)
               if (DALIB_pid() .eq. 1) then
                  print *,TMP2
               end if
            end do
         end do
```

FIG. 5.4 – *Communications générées par* ADAPTOR.

```
! io instruction
      DO J = 1, N
         DO I = 1, N
            READ (1, *) A(I,J)
            PRINT *, A(I,J)
         ENDDO
      ENDDO
! updating shared variable J
      BUF INDEX = 0
      SEND NOT INITIALIZED = .TRUE.
      BUF INDEX = BUF INDEX+1
      HPFC INTEGER4 BUFF(BUF INDEX) = J
      CALL HPFC INTEGER4 BUFPCK
      CALL HPFC HCAST
! end of update
! updating shared variable I
      BUF INDEX = 0
      SEND NOT INITIALIZED = .TRUE.
      BUF INDEX = BUF INDEX+1
      HPFC INTEGER4 BUFF(BUF INDEX) = I
      CALL HPFC INTEGER4 BUFPCK
      CALL HPFC HCAST
! end of update
! updating distributed variable A
      DO PSI1 = 1, 5
         BUF INDEX = 0
         SEND NOT INITIALIZED = .TRUE.
         DO ALPHA2 = 1, 10
            DO ALPHA1 = -1+2*PSI1, MIN(10, 2*PSI1)
               LALPHA1 = 2+ALPHA1-2*PSI1
               LALPHA2 = ALPHA2
               BUF INDEX = BUF INDEX+1
               HPFC REAL8 BUFF(BUF INDEX) = A(ALPHA1,ALPHA2)
            ENDDO
         ENDDO
         T LID = PSI1
         CALL HPFC REAL8 BUFPCK
         CALL HPFC SND TO NODE(T LID)
      ENDDO
! end of update
```

FIG. 5.5 – *Communications générées par* HPFC *(code de l'hôte).*

Le code généré par le compilateur XLHPF a été également étudié. Le résultat obtenu est sensiblement le même qu'avec le compilateur ADAPTOR, à savoir deux communications globales pour chaque itération de la boucle.

5.2 Solution proposée

Pour éviter la génération de communications inutiles concernant le traitement des entrées-sorties, le compilateur doit analyser les accès mémoire requis par l'exécution du programme.

5.2.1 Analyse des accès mémoire

L'analyse des accès mémoire dans un programme existe sous deux formes dans la littérature : les *effets* et les *régions* de tableaux.

Les effets

Les effets peuvent être vus comme une abstraction de chaque instruction du programme au niveau opérationnel (par opposition à fonctionnel). Ils peuvent décrire aussi bien les accès mémoire que les entrées-sorties ou la complexité de l'instruction.

Nous nous intéresserons, dans le cadre de ce travail, à l'analyse des accès mémoire ; les informations recherchées sont la référence concernée et le type d'accès (écriture ou lecture). Les effets qui ont *toujours* lieu seront nommés `EXACTS` ; ceux qui *peuvent* avoir lieu seront appelés `MAY`. Deux types d'effets seront évoqués à plusieurs reprises dans cette thèse. Les effets `READ` (resp. `WRITE`) concernent les valeurs lues (resp. écrites) par une instruction. Les effets `IN` (resp. `OUT`) concernent les valeurs importées (resp. exportées) par une instruction.

Les régions de tableaux

Les tableaux constituent la principale structure de données utilisée dans les programmes scientifiques, en dehors des variables scalaires. Ils sont la plupart du temps initialisés et/ou référencés dans des boucles. L'analyse des accès aux éléments de tableaux s'est donc avérée primordiale dans différents domaines dont l'analyse de dépendances. Le calcul des régions consiste à collecter des informations sur la manière dont les éléments de tableaux sont lus et écrits par le programme. Ces analyses ont été introduites par Rémi Triolet [Tri84] pour étudier les dépendances interprocédurales. Béatrice Creusillet [Cre96] a ensuite proposé une étude théorique de la sémantique des analyses de régions ; nos travaux s'appuient sur ses résultats.

Les méthodes d'analyse existantes ne permettent dans tous les cas un calcul `EXACT` ; lorsque l'exactitude ne peut être conservée, l'analyse devient approximée (région `MAY`). Il existe quatre types de régions : les régions `READ`, `WRITE`, `IN` et `OUT`.

Régions `READ-WRITE` Les régions `READ` et `WRITE` représentent les éléments de tableau lus ou écrits par les instructions (simples ou composées) lors de leur exécution. Le résultat de l'analyse des régions de tableaux `READ-WRITE` pour l'exemple de la figure 5.3 est présenté figure 5.6. La première ligne décrit par exemple de manière *exacte* la region `READ` de la matrice A ; les informations obtenues pour les coordonnées des éléments de tableaux, ici PHI1 et PHI2, sont issues de la définition du paramètre n (cf. figure 5.3).

Régions `IN-OUT` Les régions `IN` d'une instruction représentent les éléments de tableaux qu'elle importe ; seuls les éléments définis à l'extérieur de l'instruction sont pris en compte, ce qui distingue les régions `IN` des régions `READ`. De la même façon, les régions `OUT` d'une instruction représentent les éléments de tableaux qu'elle exporte ; seuls les éléments de tableaux utilisés dans la suite de l'exécution du programme sont pris en compte, ce qui les différencie des régions `WRITE`.

```
C  <A(PHI1,PHI2)-R-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10>
C  <A(PHI1,PHI2)-W-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10>
      DO J = 1, N                                              0002

C  <A(PHI1,PHI2)-R-EXACT-PHI2==J, 1<=PHI1, PHI1<=10, 1<=PHI2,
C    PHI2<=10>
C  <A(PHI1,PHI2)-W-EXACT-PHI2==J, 1<=PHI1, PHI1<=10, 1<=PHI2,
C    PHI2<=10>
         DO I = 1, N                                           0003

C  <A(PHI1,PHI2)-W-EXACT-PHI1==I, PHI2==J, 1<=PHI1, PHI1<=10,
C    1<=PHI2, PHI2<=10>
            READ (1, *) A(I,J)                                 0004

C  <A(PHI1,PHI2)-R-EXACT-PHI1==I, PHI2==J, 1<=PHI1, PHI1<=10,
C    1<=PHI2, PHI2<=10>
            PRINT *, A(I,J)                                    0005
         ENDDO
      ENDDO
```

FIG. 5.6 – *Résultats obtenus pour l'analyse des régions de tableaux.*

5.2.2 Utilisation de ces analyses pour gérer les communications

L'utilisation des effets et/ou des régions permet au compilateur de vérifier s'il est nécessaire de générer les communications dues aux entrées-sorties. Si les effets ou les régions `IN` d'une instruction effectuant des entrées-sorties sont vides, il n'est pas nécessaire de générer la communication mettant à jour les données sur l'hôte (avant les entrées-sorties). De même, si les effets ou les régions `OUT` sont vides, la mise à jour des données sur les nœuds est inutile.

L'analyse des régions `READ-WRITE` est souvent insuffisante. En effet, ces dernières se contentent de déterminer l'ensemble des données lues ou écrites par une instruction. Or, nous avons également besoin de savoir si les données sont accédées par d'autres instructions du programme. Nous utilisons donc les effets et les régions `IN-OUT`.

L'analyse des régions permet d'aller un peu plus loin dans le processus d'optimisation des communications. En effet, grâce à la connaissance des accès aux éléments de tableaux, il est possible de ne générer des communications que pour la portion des données qui est effectivement utilisée par ailleurs. Considérons l'exemple de la figure 5.7. Seuls les éléments situés dans la partie supérieure gauche de la matrice sont modifiés avant l'affichage des valeurs. La communication ne concernera donc que cette portion des données, bien que l'affichage soit total. Les régions `IN` et `OUT` obtenues sont indiquées sur la figure 5.7 sous forme de commentaires. La première décrit de manière exacte la région `OUT` de la matrice A, c'est-à-dire la partie supérieure gauche.

Considérons maintenant le code de la figure 5.8. Seule une partie de la matrice est modifiée entre les deux affichages par l'hôte. Ce dernier peut donc ne faire qu'une mise à jour partielle des données, avant le second affichage. Or, les régions `IN` pour cette instruction indiquent que toute la matrice à été écrite. Une analyse des variables qui sont *vivantes* serait donc nécessaire pour réduire le volume de communications dans ce contexte.

5.3 Implémentation dans le compilateur HPF - `hpfc`

J'ai implémenté cette technique dans le prototype de compilateur HPFC développé par Fabien Coelho. Le travail réalisé par Béatrice Creusillet [Cre96] pour le calcul des régions de tableaux dans l'outil d'ana-

```
C  <A(PHI1,PHI2)-OUT-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10,
C    PHI1+PHI2<=11>
C  <F(PHI1,PHI2)-IN-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI1+PHI2<=11>
      do j=1, n
         do i=1, n-j+1
            A(i,j) = f(i,j)
         enddo
      enddo

C  <A(PHI1,PHI2)-IN-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10>
      do j=1,n
         do i=1,n
            print *, A(i,j)
         enddo
      enddo
```

FIG. 5.7 – *Réduction du volume de communication grâce aux analyses.*

```
C  <A(PHI1,PHI2)-OUT-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10>
      do j=1,n
         do i=1,n
            read (1,*) A(i,j)
         enddo
      enddo

C  <A(PHI1,PHI2)-IN-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10>
      do j=1,n
         do i=1,n
            print *, A(i,j)
         enddo
      enddo

C  <A(PHI1,PHI2)-OUT-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10,
C    PHI1+PHI2<=11>
C  <F(PHI1,PHI2)-IN-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI1+PHI2<=11>
      do j=1, n
         do i=1, n-j+1
            A(i,j) = f(i,j)
         enddo
      enddo

C  <A(PHI1,PHI2)-IN-EXACT-1<=PHI1, PHI1<=10, 1<=PHI2, PHI2<=10>
      do j=1,n
         do i=1,n
            print *, A(i,j)
         enddo
      enddo
```

FIG. 5.8 – *Cas où la mise à jour partielle des données serait suffisante.*

lyses interprocédurales PIPS (cf. annexe B) a permis d'obtenir rapidement des résultats intéressants sur Onde24. Toutes les communications générées dans le cas d'école présenté figure 5.5 sont également supprimées.

Nous avons également utilisé ce travail dans le cadre des travaux de portage de l'application Onde24. Afin de mieux visualiser l'évolution de la solution au cours du temps, des fonctions d'affichage graphique ont été utilisées (voir chapitre 4 section 4.3). Les données de la matrice décrivant le domaine sont communiquées à l'hôte avant l'affichage (car elles ont été modifiées). Cependant, comme les fonctions d'affichage ne modifient pas les données, il n'est pas nécessaire d'effectuer une mise à jour des données sur les nœuds après l'affichage. La suppression de cette phase de synchronisation entre les processeurs permet d'effectuer l'affichage sans trop ralentir le calcul de la solution ; les nœuds peuvent continuer les calculs pour les pas de temps suivants pendant que l'hôte se charge de l'affichage (qui n'a lieu que tous les N pas de temps).

5.4 Conclusion

Les mauvaises performances constatées lors de l'exécution de programmes data-parallèles sur machines parallèles à mémoire répartie sont en partie dues à une mauvaise gestion des entrées-sorties. Nous avons notamment pu constater que les compilateurs HPF disponibles actuellement génèrent un nombre trop important de communications, ce qui introduit un surcoût non négligeable. Un effort important doit donc être réalisé dans la gestion des entrées-sorties pour les programmes data-parallèles, notamment en HPF, afin que le gain obtenu grâce à la parallélisation des calculs ne soit pas réduit. Nous proposons une analyse des accès mémoire fondée sur le calcul des effets et des régions de tableaux, permettant de réduire le volume et le nombre de communications nécessaires. Cette technique a été implémentée dans le prototype de compilateur HPFC.

Chapitre 6

De l'utilité du pipeline dans un compilateur HPF

Les machines parallèles à mémoire répartie ont apporté une augmentation des performances et des capacités de la mémoire. Malheureusement, leur utilisation introduit également un surcoût dû aux communications. En conséquence, pour obtenir des programmes performants pour des tailles de problèmes importantes, ce surcoût doit être minimisé. Une bonne distribution des données est dans un premier temps indispensable pour réduire le nombre et le volume des communications. Les couches de communications, matérielle et logicielle, doivent être également efficaces en fournissant une latence faible et un débit important.

Certaines communications ne peuvent être supprimées ; elles peuvent cependant être masquées par recouvrement avec les calculs. Quinn et Hatcher ont étudié dans [QH96] le gain pouvant être obtenu par l'utilisation de diverses méthodes de recouvrements entre calculs et communications. Le recouvrement pipeliné, utilisé dans le cas où des dépendances de données ne permettent pas de se contenter d'un mécanisme de communications non-bloquantes, y est également étudié. Le pipeline fonctionne par modification du grain de calcul. Quinn et Hatcher affirment que le gain du recouvrement pipeliné est négligeable pour les programmes parallèles classiques tels que le volume de calcul est grand par rapport au volume de communications. Cependant, leur étude se limite à l'étude du gain sur un processeur. L'objectif de notre travail est donc d'évaluer le gain obtenu par utilisation d'un mécanisme de pipeline en considérant le temps d'exécution de l'ensemble des processeurs pour un programme parallèle donné. Le recouvrement des communications par les calculs existe toujours, mais des recouvrement entre calculs s'ajoutent. Nous montrons que le gain varie selon le nombre de dimensions distribuées et pipelinées et qu'il peut devenir vraiment significatif.

L'organisation de ce chapitre est la suivante : la section 6.1 propose un rappel des techniques de recouvrement et présente brièvement les résultats de [QH96]. L'étude de gain du pipeline multi-dimensionnel sera proposée pour différentes configurations à travers la section 6.2. Nous terminerons à la section 6.3 par un rapide survol de l'implémentation du pipeline dans le compilateur HPF ADAPTOR (décrite dans [Zor96] et [BD96]).

6.1 Recouvrement des calculs et des communications

Il existe différentes façons de recouvrir les calculs et les communications selon le modèle d'architecture disponible et selon le cas de figure étudié.

Communiquer un ensemble de données pendant que l'on effectue des calculs sur un autre ensemble, permet de recouvrir les calculs et les communications. De nombreux algorithmes peuvent bénéficier de telles optimisations [DP94, DG95, DDT95, DRR96]. Cela suppose que la machine cible soit capable de recouvrir les communications par les calculs en utilisant un mécanisme matériel et/ou logiciel autonome ; le matériel doit supporter le recouvrement grâce par exemple à l'utilisation de DMA[17] et le logiciel doit permettre l'utilisation d'envois et réceptions de messages non-bloquants.

D'autres techniques, que nous n'étudierons pas ici, telles que les processus légers [FN92, FHM95, And96] (*threads*) ou les messages actifs [ECGS92] (*active messages*) permettent également d'effectuer des recouvrements.

L'existence de dépendances entre l'ensemble des données *communiquées* et celui des données *calculées* interdit le recouvrement par utilisation de communications non-bloquantes. Dans ce cas, le recouvrement pipeliné, qui consiste à réduire le grain de calcul, peut être utilisé (cette technique est appelée *Pipelined Communication-Computation Overlap* dans [QH96]). La modification du grain de calcul permet deux types de recouvrements :

Considérons dans un premier temps l'intérêt apporté par le recouvrement pour chaque processeur pris séparément. Grâce à un découpage des calculs en blocs de plus petites tailles, la communication d'un bloc i peut être recouverte par le calcul d'un bloc $i+1$. Ce mécanisme est schématisé sur la figure 6.1.

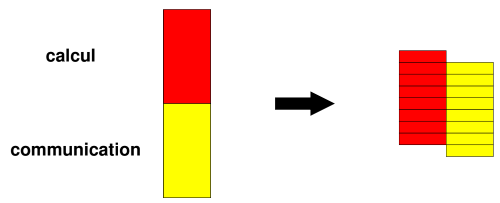


FIG. 6.1 – *Pipeline : recouvrement calculs-communications.*

Si l'on considère maintenant le problème dans sa globalité, c'est-à-dire pour tous les processeurs concernés par le calcul, le pipeline permet également de recouvrir les calculs entre eux. L'existence de dépendances entre les calculs entraînent fréquemment un schéma tel que celui présenté figure 6.2(A). Pour commencer ses calculs, le processeur P1 doit attendre que le processeur P0 ait terminé les siens et lui envoie les résultats. À son tour, dès que ses calculs sont terminés, P1 envoie ses résultats à P2 qui peut alors commencer son travail à partir des résultats reçus, et ainsi de suite. L'exécution est donc séquentialisée. Le temps d'exécution total pour l'ensemble des processeurs est plus grand que le temps d'exécution séquentiel sur un processeur, à cause des communications. Une première solution consisterait à initier les communications le plus tôt possible, c'est-à-dire dès qu'un élément de donnée à été calculé. Ce pipeline à grain fin ((B) sur la figure 6.2) entraîne un surcoût important à cause des latences de communications. Un compromis entre le nombre de communications et le temps de démarrage des processeurs doit donc être trouvé afin de constituer un pipeline à grain « moyen » ((C) sur la figure 6.2). La figure 6.3 donne un exemple de code utilisant la syntaxe HPF et pouvant bénéficier de l'utilisation du pipeline en raison de dépendances de données portées par la boucle.

6.1.1 État de l'art

De nombreux travaux ont concerné les problèmes de recouvrement depuis l'apparition de la première machine parallèle à mémoire répartie. L'intérêt pour ce type d'optimisation a augmenté ces dernières années en raison des faibles performances mesurées sur ces machines et dues au surcoût des communications.

Les recouvrements calculs-communications, dont le pipeline, ont été utilisés à différentes reprises pour améliorer les performances des programmes data-parallèles [KCN88, CCM95, SKK+96, AGGL97]. La taille de bloc optimale pour le pipeline a été étudiée par les auteurs de [SS94, OSKO95, SS95a, CLS98] et

17. Direct Memory Access

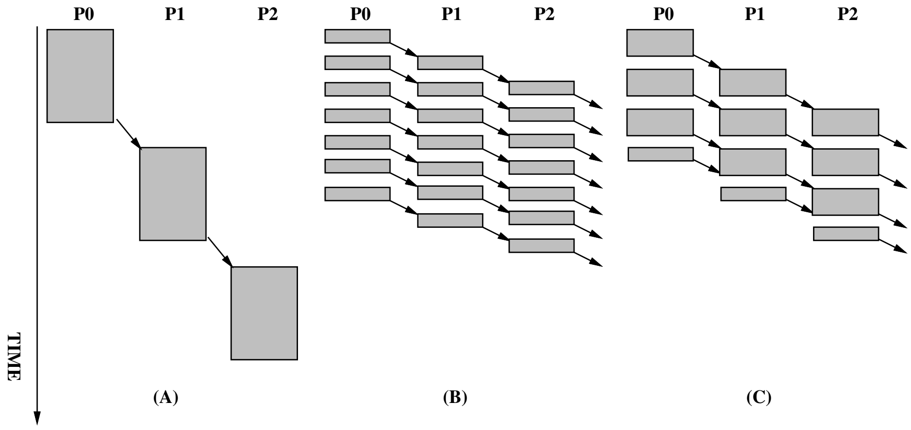


FIG. 6.2 – *Pipeline : recouvrement calculs-calculs.*

```
      REAL A(N,N)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK)
      ...
      DO I=2,N
        DO J=2,N
  (S)       A(I,J) = f(A(I-1,J),A(I,J-1))
        END DO
      END DO
```

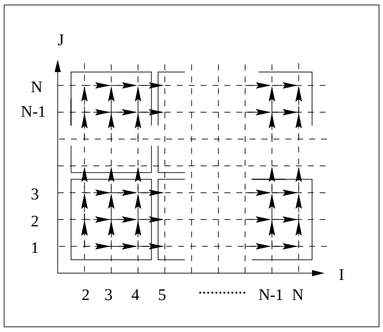


FIG. 6.3 – *Un exemple de code pouvant bénéficier du pipeline.*

plusieurs compilateurs de langages data-parallèles utilisent le pipeline [Tse93, HKT94, BCG$^+$94, GB94, PSCB94, EJLC97]. Rappelons enfin que le pipeline est une technique assez proche de la transformation de boucles appelée *Tiling* [IT88, WL91, RP94].

La librairie `LOCCS`[18] a été conçue pour masquer l'utilisation de ces techniques au programmeur [Des94, DT94]. Une autre librairie, appelée `OPIUM`[19], a été développée pour le calcul de la taille de paquets optimale, paramètre critique pour l'efficacité du pipeline [DRR96].

Les travaux de Quinn et Hatcher [QH96] sont présentés plus en détail à la section 6.1.2 ; à notre connaissance, c'est la seule étude ayant cherché à mesurer le gain pouvant être obtenu par utilisation des recouvrements. Nos travaux étendent leurs résultats.

6.1.2 Les travaux de Quinn and Hatcher

Modélisation de l'architecture

Dans la mesure où ce travail étend les résultats présentés par Quinn et Hatcher dans [QH96], nous utilisons des notations similaires aux leurs. Nous supposons que la machine cible à une mémoire répartie et utilise un réseau de communications pour échanger les données entre les processeurs. Dans la suite de cette étude, nous noterons $T(n)$ le coût de communication de n éléments de données entre deux processeurs. C'est en général une fonction linéaire de la taille des messages et nous écrirons donc $T(n) = \lambda_T + \beta_T n$. De même, la latence des communications sera notée $L(n) = \lambda_L + \beta_L n$. Cette valeur inclut le coût de

18. Low Overhead Computation and Communication Subroutines.
19. Optimal Packet sIze Computation Methods.

bufferisation du message ainsi que le temps nécessaire pour l'appel à la fonction de communication (`send`). Nous appellerons également $L'(n)$, le coût de réception d'un message de taille n ; ce coût est supposé égal à $L(n)$.

Analyse de gains

Dans cette section, nous rappelons les principaux résultats décrits par Quinn et Hatcher dans [QH96] concernant le gain obtenu par recouvrement des calculs et communications. Cette analyse repose sur le modèle d'architecture décrit ci-dessus et concerne l'étude des gains en ne considérant qu'**un seul processeur**.

Un programme data-parallèle contient des segments de calculs mélangés avec des appels à des fonctions de communications. Les auteurs de [QH96] suppose que la structure du programme est toujours similaire à celle présentée sur la figure 6.4 (M est l'ensemble des variables constituant le message). Cet algorithme servira de référence pour l'étude des recouvrements. Une première optimisation peut être appliquée dans la mesure où le segment de code CS_2 peut recouvrir la communication Com_2. Com_2 est donc déplacée entre les calculs CS_1 et CS_2. Le recouvrement est appliqué en utilisant un appel à une routine de communication asynchrone et en plaçant juste après CS_2 et devant CS_3, une fonction de type `wait` qui permet de contrôler la terminaison de la communication. Après déplacement de Com_2, nous obtenons donc la configuration présentée figure 6.5.

$< Com_1 >$	`Communication 1`
$< CS_1 >$	`Code seg. 1 (incorporating var. in M)`
$< CS_2 >$	`Code seg. 2 (not incorp. var. in M)`
$< Com_2 >$	`Communication 2 with message list M`
$< CS_3 >$	`Code seg. 3 (not incorp. var. in M)`
$< CS_4 >$	`Code seg. 4 (incorp. var. in M)`
$< Com_3 >$	`Communication 3`

FIG. 6.4 – *Algorithme de référence pour l'étude des méthodes de recouvrement.*

Le segment de code, appelé C_o sur la figure 6.5, peut être utilisé pour le recouvrement de la communication 2. Par contre, en raison de dépendances de données, le segment de code noté C_r ne peut pas être directement utilisé.

$$
\begin{array}{l}
< Com_1 > \\
< CS_1 > \\
< Com_2 > \\
\left.\begin{array}{l} < CS_2 > \\ < CS_3 > \end{array}\right\} C_o \\
\left.< CS_4 >\right\} C_r \\
< Com_3 >
\end{array}
$$

FIG. 6.5 – *Algorithme de référence après la première optimization.*

Le tableau 6.1 présente les noms donnés dans [QH96] aux quatre configurations qui peuvent avoir lieu selon l'existence des calculs C_o et C_r. Les recouvrement *Systolic* et *General* sont assez similaires et utilisent tous deux un appel à une fonction de communication asynchrone pour recouvrir la communication. Le recouvrement *pipeliné* servira de base à notre étude. Enfin, notons que lorsque C_o et C_r sont vides, il est possible de concaténer les communications 2 et 3 de manière à effectuer un recouvrement avec de futurs calculs. Pour chacune de trois configurations retenues, nous allons rapidement rappeler les résultats proposés par Quinn et Hatcher concernant le gain maximal en performance qui peut être obtenu par les techniques de recouvrement sur un processeur.

		Segment C_o	
		Contains code	Empty
Segment C_r	Contains code	General overlap	Data dependencies prevent overlap
	Empty	Systolic overlap	No computations to overlap with communication

TAB. 6.1 – *4 configurations pour le recouvrement.*

Recouvrements *Systolic* et *General* Le gain maximal en performances pouvant être obtenu est donné par la formule suivante :

$$SG(n) = 1 + \frac{\min(C_0(n), T(n))}{L(n) + C_r(n) + \max(C_0(n), T(n))} \tag{6.1}$$

Quinn et Hatcher proposent deux corollaires sur les bornes de $SG(n)$. Le premier est que $SG(n) \leq 2$. Ce résultat, obtenu simplement à partir de la propriété $(A+B)/\max(A,B) \leq 2$, indique que le regroupement des communications par les calculs peut au mieux diviser le temps d'exécution par 2 ; exécuter deux actions de manière concurrente est au mieux deux fois plus rapide que de les exécuter l'une après l'autre. Le second corollaire indique que si $\lim_{n\to+\infty} T(n)/C_0(n) = 0$ et $\lim_{n\to+\infty} L(n)/C_0(n) = 0$ alors $\lim_{n\to+\infty} SG(n) = 1$. Lorsque le coût de calcul est vraiment grand devant le coût des communications, le recouvrement des communications n'apporte bien évidemment pas de gain intéressant [20].

Recouvrement pipeliné Le gain maximal en performances pouvant être obtenu par utilisation du recouvrement pipeliné, par des paquets de taille n_0, est donné par la formule suivante :

$$P(n) = \frac{L(n) + T(n) + C_r(n)}{\frac{n}{n_0}L(n_0) + \left(\frac{n}{n_0} - 1\right)\max(T(n_0), C_r(n_0)) + T(n_0) + C_r(n_0)} \tag{6.2}$$

En raison de la latence des communications, le temps d'exécution avec pipeline peut être plus important que le temps de la version non-pipelinée. Comme indiqué par Quinn et Hatcher, si la latence est telle que $\lambda_L < \min(T(n_0), C_r(n_0))$ alors le temps d'exécution sera toujours réduit. Si la latence du message est trop importante, il sera donc nécessaire d'envoyer un « gros » message au lieu de plusieurs petits. Il y a donc un compromis à trouver pour le calcul de la taille de grain optimale (ce n'est pas l'objet de cette étude).

En suivant de nouveau les hypothèses $\lim_{n\to+\infty} \frac{T(n)}{C_r(n)} = 0$ et $\lim_{n\to+\infty} \frac{L(n)}{C_r(n)} = 0$, nous obtenons le résultat $\lim_{n\to+\infty} P(n) \leq 1$. En effet, si le coût des communications est négligeable comparé au coût des calculs, il n'y a aucun gain à obtenir par recouvrement pipeliné sur un processeur et la latence des communications peut même entraîner une perte de performance. Par contre, nous verrons que les recouvrements calcul-communication et calcul-calcul que permet le pipeline sont bénéfiques lorsque l'on considère plusieurs processeurs.

6.1.3 Paramètres

Dans la suite de cette étude, nous ferons certaines hypothèses quant à la taille des données (matrices), leur distribution sur la topologie virtuelle de processeurs, les dépendances de données entre les calculs et le nombre de dimensions utilisées pour le pipeline. Les paramètres que nous ferons varier sont les suivants :

- N_m : nombre de dimensions des matrices.
- N_d : nombre de dimensions distribuées par bloc.

20. La distinction qui est faite entre les recouvrements *systolic* et *general* dans [QH96] se limite à la présence ou non de l'opérande $C_r(n)$ au dénominateur pour la formule indiquée.

– N_p : nombre de dimensions utilisées pour le pipeline.

Nous aurons bien évidemment la propriété $N_p \leq N_d \leq N_m$. Nous appelons $t_{np}^{N_d}$ le temps d'exécution obtenu avec la version *non pipelinée* d'une matrice distribuée sur N_d dimensions. De même, $t_p^{N_d}$ sera le temps d'exécution de la version *avec pipeline*.

Nous supposons qu'il existe au minimum un vecteur de dépendance $(d_1, d_2, \ldots, d_i > 0, \ldots, d_{N_m-1}, d_{N_m})$ pour chacune des dimensions distribuées d_i de la matrice. Lorsque cette hypothèse n'est pas vérifiée, plusieurs pipelines de plus petite dimension pourront être initiés en parallèle.

Les matrices manipulées sont supposées de taille n selon chacune de leurs dimensions. Le coût de calcul de la matrice complète s'écrit donc sous la forme $\tau_c \times n^{N_m}$ avec τ_c le coût de calcul d'un élément. Les études de complexité aux limites seront données avec la notation $o(n^{N_m})$.

6.2 Étude de gain du pipeline multi-dimensionnel

L'objectif de cette section est de fournir une analyse des recouvrements par la technique du pipeline lorsque l'on considère des matrices multi-dimensionnelles distribuées par blocs sur un ensemble de P processeurs. Étant donnée la complexité de l'analyse et des équations proposées, l'organisation de cette section sera la suivante : plusieurs résultats concernant les matrices bi-dimensionnelles seront dans un premier temps détaillés puis nous généraliserons la technique aux matrices de dimension N_m quelconque.

Remarque 6.1
Une partie des résultats de cette étude a été présentée lors de la conférence Renpar'9 [DZ97].

6.2.1 Matrice bi-dimensionnelle distribuée selon une seule dimension

La figure 6.6 montre l'amélioration en terme d'instant de démarrage des calculs pour chacun des blocs d'une matrice lorsque le pipeline est utilisé. Les traits gras schématisent la distribution de la matrice et le découpage du pipeline est représenté par des traits pointillés. La partie droite de la figure correspond au schéma d'exécution sans pipeline et la partie gauche décrit l'amélioration apportée par le fonctionnement avec pipeline.

9	18	27
8	17	26
7	16	25
6	15	24
5	14	23
4	13	22
3	12	21
2	11	20
1	10	19

9	10	11
8	9	10
7	8	9
6	7	8
5	6	7
4	5	6
3	4	5
2	3	4
1	2	3

FIG. 6.6 – *Instants de démarrage des calculs.*

La matrice est supposée carrée, de taille $n \times n$ et distribuée par blocs sur P processeurs. Le coût de calcul de chacunes des P portions de la matrice distribuée, noté C_r^n, est donc $\tau_c \times \frac{n^2}{P}$. Nous supposons dans la suite de ce chapitre que la taille de paquet optimale à été calculée, par exemple en utilisant la librairie `OPIUM` [DRR96], et est utilisée. Chacun des paquets est supposé de taille $n \times n_0$. Le coût de calcul de chaque paquet, noté $C_r^{n_0}$, est donc égal à $\tau_c \times \frac{n}{P} \times n_0$. Notons que la quantité de calculs effectués avec ou sans pipeline reste identique, puisque nous avons $\frac{n}{n_0} C_r^{n_0} = C_r^n$ [21].

Lemme 6.1
Soit $t_{np}^{1,2}$ le temps d'exécution total **sans pipeline** *pour une matrice bi-dimensionnelle distribuée par blocs selon une dimension. Nous avons :*

$$t_{np}^{1,2} = PC_r^n + (P-1)[2L(n) + T(n)] \tag{6.3}$$

Preuve 6.1

Pour évaluer le temps d'exécution total sans pipeline, nous devons prendre en compte le travail des P processeurs en séquentiel avec $P-1$ phases de communications (voir figure 6.7). *Q.E.D.*

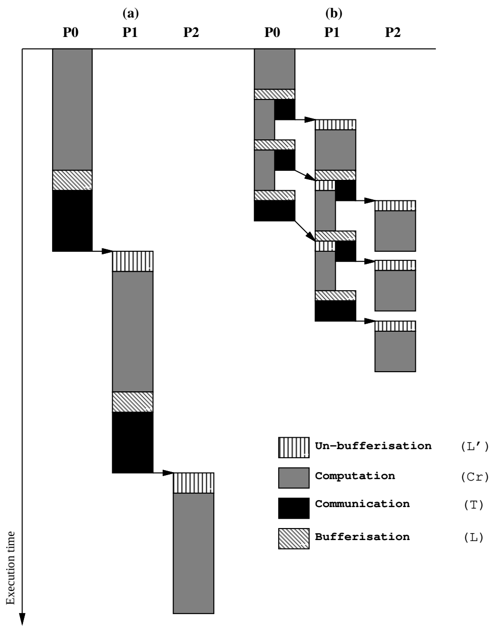


FIG. 6.7 – *Gain en performance pour P processeurs quand $C_r^n > T(n)$.*

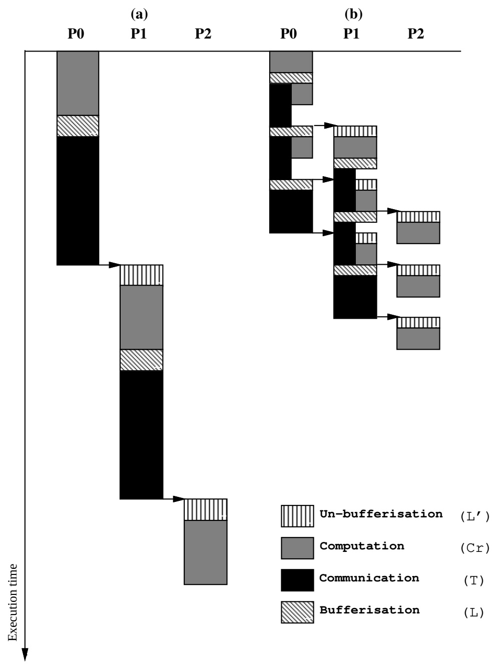


FIG. 6.8 – *Gain en performance pour P processeurs quand $C_r^n < T(n)$.*

Lemme 6.2
Soit $t_p^{1,2}$ le temps d'exécution total **avec pipeline** *pour une matrice bi-dimensionnelle distribuée par blocs selon une dimension. Nous avons :*

$$\begin{aligned} t_p^{1,2} &= PC_r^{n_0} + (P-1)[2L(n_0) + T(n_0)] \\ &\quad + \left(\frac{n}{n_0} - 1\right)[\max(L(n_0) + C_r^{n_0}, T(n_0)) + L(n_0)] \end{aligned} \tag{6.4}$$

Preuve 6.2

Les figures 6.7 et 6.8 présentent les diagrammes d'exécution avec pipeline. Une communication $T_i(n_0)$ peut être recouverte par la latence $L_{i+1}(n_0)$ et le calcul $C_{r,i+1}^{n_0}$. Selon que le coût de

21. En reprenant la notation de Quinn et Hatcher, nous avons $C_r^n = C_r(\frac{n^2}{P})$ et $C_r^{n_0} = C_r(\frac{n}{P} \times n_0)$.

communication domine ou non la somme des coûts de latence et calcul, le chemin critique d'exécution sera différent. Le temps total d'exécution inclut des termes décrivant le temps de démarrage du pipeline et d'autres termes correspondant aux étapes d'exécution pour un processeur.

$$\begin{aligned} t_p^{1,2} &= C_r^{n_0} + L(n_0) + T(n_0) + (P-2)[C_r^{n_0} + 2L(n_0) + T(n_0)] \\ &\quad + \left(\frac{n}{n_0} - 1\right)[\max(L(n_0) + C_r^{n_0}, T(n_0)) + L(n_0)] + L(n_0) + C_r^{n_0} \\ &= PC_r^{n_0} + (P-1)[2L(n_0) + T(n_0)] \\ &\quad + \left(\frac{n}{n_0} - 1\right)[\max(L(n_0) + C_r^{n_0}, T(n_0)) + L(n_0)] \end{aligned} \tag{6.5}$$

Q.E.D.

Lemme 6.3
Soit $G_1^2(n)$ le gain en performance pouvant être obtenu par utilisation du pipeline pour une matrice bi-dimensionnelle distribuée par blocs selon une dimension. Nous avons :

$$G_1^2(n) = \frac{PC_r^n + (P-1)[2L(n) + T(n)]}{PC_r^{n_0} + (P-1)[2L(n_0) + T(n_0)] + \left(\frac{n}{n_0} - 1\right)[\max(L(n_0) + C_r^{n_0}, T(n_0)) + L(n_0)]} \tag{6.6}$$

Preuve 6.3

Le gain en performance apporté par l'utilisation du pipeline peut être évalué en divisant le temps d'exécution sans pipeline ($t_{np}^{1,2}$) par le temps d'exécution avec pipeline ($t_p^{1,2}$).

Q.E.D.

Théorème 6.1
Si $\lim_{n\to+\infty} \frac{T(n)}{C_r^n} = 0$ et $\lim_{n\to+\infty} \frac{L(n)}{C_r^n} = 0$, alors $\lim_{n\to+\infty} G_1^2(n) = P$

Preuve 6.1

Comme Quinn et Hatcher, nous supposons ici que le coût de calcul est très grand comparé aux coûts de communication et de latence. Sous ces hypothèses, nous pouvons réécrire l'équation 6.6 sous la forme proposée ci-dessous dans l'équation 6.7.

$$\begin{aligned} G_1^2(n) &= \frac{PC_r^n + (P-1)[2L(n) + T(n)]}{PC_r^{n_0} + (P-1)[2L(n_0) + T(n_0)] + \left(\frac{n}{n_0} - 1\right)[2L(n_0) + C_r^{n_0}]} \\ &= \frac{PC_r^n + (P-1)[2L(n) + T(n)]}{PC_r^{n_0} + (P-1)[2L(n_0) + T(n_0)] + C_r^n + 2\frac{n}{n_0}L(n) - C_r^{n_0} - 2L(n_0)} \end{aligned} \tag{6.7}$$

Afin de calculer le gain en performance et dans la mesure où nous supposons que n est très grand, nous remplaçons les coûts par leurs complexités respectives (résumée dans le tableau 6.2).

$$\begin{aligned} \lim_{n\to+\infty} G_1^2(n) &= \frac{P \times o(n^2) + o(n)}{o(n^2) + o(n)} \\ \lim_{n\to+\infty} G_1^2(n) &= P \end{aligned} \tag{6.8}$$

Q.E.D.

	value	complexity
C_r^n	$n^2/p \times \tau_c$	$o(n^2)$
$T(n)$	$\lambda_T + \beta_T \times n$	$o(n)$
$L(n)$	$\lambda_L + \beta_L \times n$	$o(n)$
$C_r^{n_0}$	$n/p \times n_0 \times \tau_c$	$o(n)$
$T(n_0)$	$\lambda_T + \beta_T \times n_0$	$o(1)$
$L(n_0)$	$\lambda_L + \beta_L \times n_0$	$o(1)$

TAB. 6.2 – *Complexité des différents coûts.*

6.2.2 Matrice bi-dimensionnelle distribuée selon deux dimensions

Le pipeline peut également être utilisé lorsque le matrice est distribuée par blocs selon deux dimensions. Nous supposons ici qu'une grille carré de P processeurs est utilisée ($\sqrt{P}\times\sqrt{P}$) ; les résultats pourront être facilement étendus pour des grilles non carrées. Avec une matrice carrée de taille $n \times n$, chaque processeur reçoit une sous-matrice de taille $n' \times n'$ avec $n' = n/\sqrt{P}$. Le coût de calcul de cette sous-matrice, noté $C_r^{n'}$), est donc $\tau_c \times n' \times n'$. De même, le coût de calcul de chacun des blocs du pipeline, noté $C_r^{n'_0}$, est $\tau_c \times n' \times n'_0$.

Avec des dépendances de la forme $(d_1, 0)$ et $(0, d_2)$ telles que $d_1, d_2 > 0$ (par exemple le code de la figure 6.3), un pipeline mono-dimensionnel selon l'une des deux dimensions distribuées peut être appliqué. La figure 6.9 donne le schéma global de fonctionnement d'un pipeline sur la première dimension d'une matrice carrée.

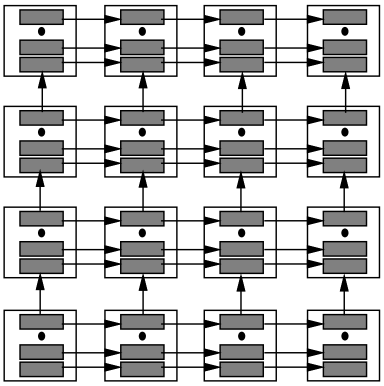


FIG. 6.9 – *Pipeline mono-dimensionnel pour une matrice carrée distribuée selon deux dimensions.*

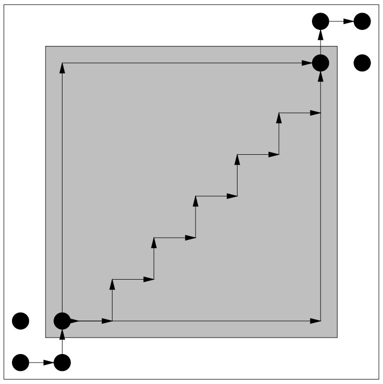


FIG. 6.10 – *Les chemins critiques d'exécution.*

Notre premier objectif est de calculer le temps d'exécution total *sans pipeline*. Tous les chemins à l'intérieur de la partie grisée sur la figure 6.10 sont équivalents en terme de temps d'exécution. Toutefois, nous devons considérer séparément les cas du démarrage et de la terminaison du pipeline en raison de l'absence de communications.

Lemme 6.4
Soit $t_{np}^{2,2}$ le temps d'exécution total ***sans pipeline*** *pour une matrice bi-dimensionnelle distribuée par blocs selon deux dimensions. Nous avons :*

$$t_{np}^{2,2} = (2\sqrt{P} - 1)C_r^{n'}) + (7\sqrt{P} - 8)L(n') + 2(\sqrt{P} - 1)T(n') \quad (6.9)$$

Preuve 6.4

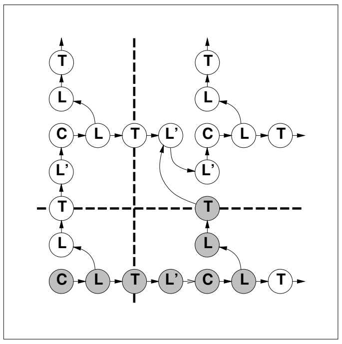


FIG. 6.11 – *Démarrage.*

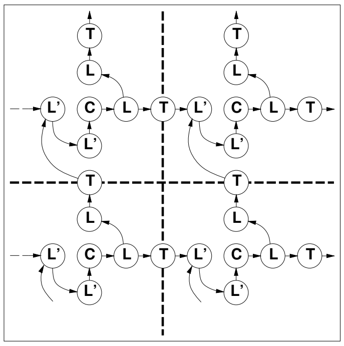


FIG. 6.12 – *Fonctionnement général.*

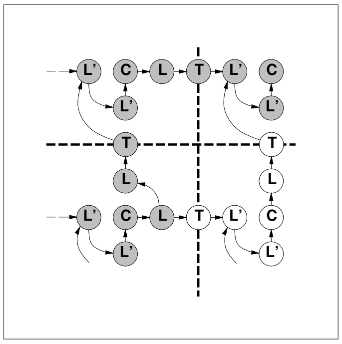


FIG. 6.13 – *Terminaison.*

Le temps total d'exécution sans pipeline est donc constitué de la somme de trois termes ; les temps de démarrage $t^{2,2}_{np1}$, de fonctionnement général $t^{2,2}_{np2}$ et de terminaison $t^{2,2}_{np3}$. Pour faciliter la compréhension du résultat à partir des figures, nous distinguons ici L_h, L'_h et T_h (resp. L_v, L'_v et T_v) les temps des latence et de communication selon la direction horizontale (resp. verticale).

$$\begin{aligned}
t^{2,2}_{np1} &= C^{n'}_r) + L_h(n') + T_h(n') + L'_h(n') + C^{n'}_r) + L_h(n') + L_v(n') + T_v(n') && (6.10)\\
t^{2,2}_{np2} &= (\sqrt{P} - 3) * (L'_h(n') + L'_v(n') + C^{n'}_r) + L_h(n') + L_v(n') + T_v(n') + \\
&\quad L'_h(n') + L'_v(n') + C^{n'}_r) + L_h(n') + T_h(n')) && (6.11)\\
t^{2,2}_{np3} &= L'_h(n') + L'_v(n') + C^{n'}_r) + L_h(n') + L_v(n') + T_v(n') + L'_h(n') + \\
&\quad + L'_v(n') + C^{n'}_r) + L_h(n') + T_h(n') + L'_h(n') + L'_v(n') + C^{n'}_r) && (6.12)
\end{aligned}$$

Q.E.D.

Lemme 6.5
Soit $t^{2,2}_n$ le temps d'exécution total **avec pipeline** *pour une matrice bi-dimensionnelle distribuée par blocs selon deux dimensions. Nous avons :*

$$\begin{aligned}
t^{2,2}_p &= (\sqrt{P} - 1) * [L'_h(n'_0) + C^{n'_0}_r + L_h(n'_0) + L_v(n') + T_v(n') + \\
&\quad \left(\frac{n'}{n'_0} - 2\right) \max(T_h(n'_0), L'_h(n'_0) + C^{n'_0}_r + L_h(n'_0))] + \\
&\quad (\sqrt{P} - 2) * [L'_h(n'_0) + L'_v(n') + C^{n'_0}_r + L_h(n'_0) + T_h(n'_0) + \\
&\quad \max(T_h(n'_0), L'_h(n'_0) + L'_v(n') + C^{n'_0}_r + L_h(n'_0)] + \\
&\quad \max(T_h(n'_0), L'_h(n'_0) + C^{n'_0}_r + L_h(n'_0)) + \\
&\quad L'_v(n') + C^{n'_0}_r + T_h(n'_0) + C^{n'}_r) + L'_h(n') && (6.13)
\end{aligned}$$

Preuve 6.5

De la même façon, le temps total d'exécution *avec pipeline* est donné par la somme de trois termes $t^{2,2}_{p1}$, $t^{2,2}_{p2}$ et $t^{2,2}_{p3}$. Les figures 6.14, 6.15 et 6.16 présentent l'enchaînement des différentes phases de calculs et de communications. La présence deux chemins critiques, représentée sur les figures par des fonds horizontaux et verticaux, entraîne l'apparition de termes en max dans les équations.

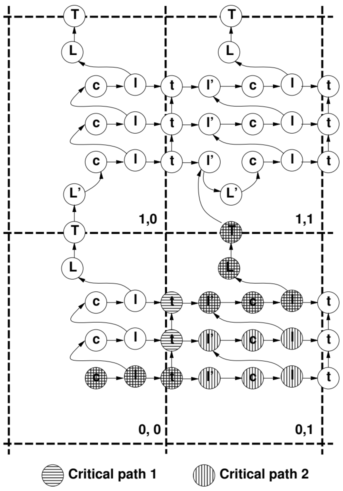


FIG. 6.14 – *Démarrage.*

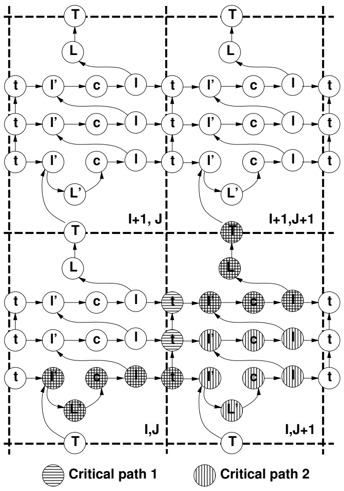


FIG. 6.15 – *Fonctionnement général.*

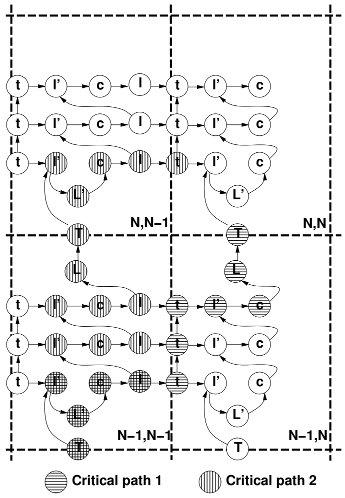


FIG. 6.16 – *Terminaison.*

$$
\begin{aligned}
t_{p1}^{2,2} &= C_r^{n_0'} + L_h(n_0') + T_h(n_0') + \left(\frac{n'}{n_0'} - 1\right) \max(T_h(n_0'), L_h'(n_0') + C_r^{n_0'} + L_h(n_0')) + \\
&\quad L_h'(n_0') + C_r^{n_0'} + L_h(n_0') + L_v(n') + T_v(n') && (6.14) \\
t_{p2}^{2,2} &= (\sqrt{P} - 3) * [L_h'(n_0') + L_v'(n') + C_r^{n_0'} + L_h(n_0') + T_h(n_0') + \\
&\quad \max(T_h(n_0'), L_h'(n_0') + L_v'(n') + C_r^{n_0'} + L_h(n_0')) + \\
&\quad \left(\frac{n'}{n_0'} - 2\right) \max(T_h(n_0'), L_h'(n_0') + C_r^{n_0'} + L_h(n_0')) + \\
&\quad L_h'(n_0') + C_r^{n_0'} + L_h(n_0') + L_v(n') + T_v(n')] && (6.15) \\
t_{p3}^{2,2} &= L_h'(n_0') + L_v'(n') + C_r^{n_0'} + L_h(n_0') + T_h(n_0') + \\
&\quad \max(T_h(n_0'), L_h'(n_0') + L_v'(n') + C_r^{n_0'} + L_h(n_0')) + \\
&\quad \left(\frac{n'}{n_0'} - 2\right) \max(T_h(n_0'), L_h'(n_0') + C_r^{n_0'} + L_h(n_0')) + \\
&\quad L_h'(n_0') + C_r^{n_0'} + L_v(n') + T_v(n') + \\
&\quad L_h'(n_0') + L_v'(n') + C_r^{n_0'} + \left(\frac{n'}{n_0'} - 1\right) (L_h'(n_0') + C_r^{n_0'}) && (6.16)
\end{aligned}
$$

Après quelque étapes de restructuration de l'équation, nous obtenons le résultat suivant :

$$\begin{aligned}
t_p^{2,2} &= t_{p1}^{2,2} + t_{p2}^{2,2} + t_{p3}^{2,2} \\
&= (\sqrt{P}-1) * [L'_h(n'_0) + C_r^{n'_0} + L_h(n'_0) + L_v(n') + T_v(n') + \\
&\quad \left(\frac{n'}{n'_0} - 2\right) \max(T_h(n'_0), L'_h(n'_0) + C_r^{n'_0} + L_h(n'_0))] + \\
&\quad (\sqrt{P}-2) * [L'_h(n'_0) + L'_v(n') + C_r^{n'_0} + L_h(n'_0) + T_h(n'_0) + \\
&\quad \max(T_h(n'_0), L'_h(n'_0) + L'_v(n') + C_r^{n'_0} + L_h(n'_0)] + \\
&\quad \max(T_h(n'_0), L'_h(n'_0) + C_r^{n'_0} + L_h(n'_0)) + \\
&\quad L'_v(n') + C_r^{n'_0} + T_h(n'_0) + C_r^{n'}) + L'_h(n') \qquad (6.17)
\end{aligned}$$

Q.E.D.

Théorème 6.2
Soit $G_2^2(n)$ le gain en performance pouvant être obtenu par utilisation du pipeline pour une matrice bi-dimensionnelle distribuée par blocs selon deux dimensions. Si $\lim_{n\to+\infty} \frac{T(n)}{C_r^{n'})} = 0$ et $\lim_{n\to+\infty} \frac{L(n)}{C_r^{n'})} = 0$, alors $\lim_{n\to+\infty} G_2^2(n) \leq 2$

Preuve 6.2
L'approche est exactement la même que celle de la section 6.2.1. En partant de l'hypothèse que n est très grand, il est possible de simplifier les termes max puis de remplacer les coûts par leurs complexités respectives.

$$\begin{aligned}
\lim_{n\to+\infty} G_2^2(n) &= \frac{(2\sqrt{P}-1) \times o(n^2) + o(n)}{\sqrt{P} \times o(n^2) + o(n)} \\
&= \frac{2\sqrt{P}-1}{\sqrt{P}} \\
&\leq 2 \qquad (6.18)
\end{aligned}$$

Q.E.D.

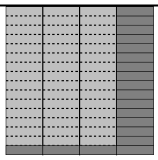
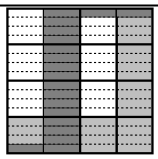
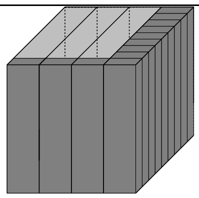
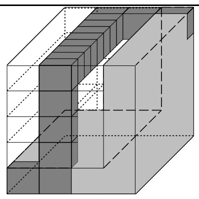
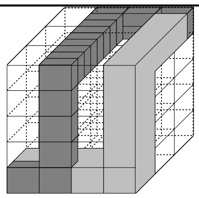
L'évaluation du gain en performance repose sur une comparaison entre les temps d'exécution avec et sans pipeline. La réduction du gain dans le cas de matrices carrées distribuées selon leurs deux dimensions n'est donc guère surprenante dans la mesure où le temps d'exécution sans pipeline bénéficie d'ores et déjà d'un degré de parallélisme (voir figure 6.9). Notons qu'il serait intéressant de mettre en place une méthode permettant de sélectionner la dimension par laquelle le pipeline donnerait les meilleurs résultats. Le choix de la direction de pipeline choisie serait également une donnée importante pour le cas des matrices multi-dimensionnelles présenté ci-dessous.

6.2.3 Extension aux matrices de dimension quelconque.

L'objet de cette section est de généraliser cette étude aux matrices multi-dimensionnelles distribuées par blocs selon plusieurs dimensions. Une seule dimension est pipelinée.

Proposition 6.1
Si $\lim_{n\to+\infty} \frac{T(n)}{C_r^n} = 0$ et $\lim_{n\to+\infty} \frac{L(n)}{C_r^n} = 0$, alors le gain maximal peut être approché par la formule suivante :

$$G_{N_d}^{N_m} \simeq \frac{N_d \times \sqrt[N_d]{P}}{NP_i} avecNP_i = \begin{cases} 1 & si\ N_d < N_m, \\ \sqrt[N_d]{P} & si\ N_d = N_m. \end{cases} \qquad (6.19)$$

Chemin critique	**Paramètres**	**Gain**
	$N_m = 2$ $N_d = 1$ $P \times 1$	$G_1^2 \simeq \frac{P \times o(n^2) + C_1 \times o(n)}{o(n^2) + C_2 \times o(n)} \simeq P$
	$N_m = 2$ $N_d = 2$ $\sqrt{P} \times \sqrt{P}$	$G_2^2 \simeq \frac{2\sqrt{P} \times o(n^2) + C_1 \times o(n)}{\sqrt{P} \times o(n^2) + C_2 \times o(n)} \simeq 2$
	$N_m = 3$ $N_d = 1$ $P \times 1 \times 1$	$G_1^3 \simeq \frac{P \times o(n^3) + C_1 \times o(n^2)}{o(n^3) + C_2 \times o(n^2)} \simeq P$
	$N_m = 3$ $N_d = 2$ $\sqrt{P} \times \sqrt{P} \times 1$	$G_2^3 \simeq \frac{2\sqrt{P} \times o(n^3) + C_1 \times o(n^2)}{o(n^3) + C_2 \times o(n^2)} \simeq 2\sqrt{P}$
	$N_m = 3$ $N_d = 3$ $\sqrt[3]{P} \times \sqrt[3]{P} \times \sqrt[3]{P}$	$G_3^3 \simeq \frac{3\sqrt[3]{P} \times o(n^3) + C_1 \times o(n^2)}{\sqrt[3]{P} \times o(n^3) + C_2 \times o(n^2)} \simeq 3$

TAB. 6.3 – *Gains en performance pour des matrices distribuées selon N_d dimensions.*

Explications 6.1

Nous nous intéressons au gain en performance apporté par le pipeline lorsque n est suffisamment grand, c'est-à-dire lorsque l'on peut supposer $\tau_c \times n^{N_m} \gg \lambda_T + \beta_T \times n^{N_m - 1}$. Les résultats obtenus pour des matrices bi-dimensionnelles peuvent être étendus de la manière suivante :

- le temps d'exécution sans pipeline est constitué de la somme de $N_d \times \sqrt[N_d]{P}$ étapes de calculs entrecoupées de phases de communications. Le calcul progresse séquentiellement le long des N_d dimensions distribuées sur $\sqrt[N_d]{P}$ processeurs. Nous avons donc $t_{np}^{N_d,N_m} \simeq N_d \times \sqrt[N_d]{P} \times o(n^{N_m}) + o(n^{N_m-1})$.
- Soit i la dimension choisie pour le pipeline. L'exécution le long de cette dimension est toujours séquentielle. Donc, si NP_i est le nombre de processeurs selon la dimension i, il y a NP_i étapes de calculs de complexité $o(n^{N_m})$. La complexité des calculs selon toutes les autres dimensions est seulement $o(n^{N_m-1})$ (la complexité du terme C_r^n est réduite par le pipeline à la complexité du terme $C_r^{n_0}$). Nous obtenons donc $t_p^{N_d,N_m} \simeq NP_i \times o(n^{N_m}) + o(n^{N_m-1})$.

Ces deux termes nous permettent donc d'aboutir à l'approximation du gain en performance suivante :

$$G_{N_d}^{N_m} \simeq \frac{N_d \times \sqrt[N_d]{P} \times o(n^{N_m}) + o(n^{N_m - 1})}{NP_i \times o(n^{N_m}) + o(n^{N_m - 1})} \simeq \frac{N_d \times \sqrt[N_d]{P}}{NP_i} \tag{6.20}$$

Le gain maximal quand n est très grand sera obtenu en choisissant pour le pipeline une dimension avec un nombre minimum de processeurs car l'exécution selon cette dimension est toujours séquentiel. Si toutes les dimensions ne sont pas distribuées, nous choisissons donc un pipeline selon une dimension non distribuée et $NP_i = 1$. Dans le cas contraire, nous avons $NP_i = \sqrt[N_d]{P}$.

6.2.4 Opportunités de pipeline selon N_p dimensions

Pour des matrices multi-dimensionnelles, le recouvrement des calculs et des communications peut être encore augmenté par l'application du pipeline selon plusieurs dimensions. Cela entraîne bien évidemment un surcoût en raison de l'augmentation du nombre de messages.

Il peut être difficile d'obtenir de bonnes performances grâce au pipeline lorsque les coûts de communications sont importants. En effet, lorsque n n'est pas suffisamment grand, l'hypothèse $C_r^n \gg T(n)$ n'est plus forcement respectée. Il devient alors nécessaire de réduire les termes de complexité $o(n^{N_m-1})$ présents au dénominateur sur la formule 6.20. Le pipeline multi-dimensionnel est efficace pour masquer le coût des communications dans la direction principale de pipeline (les termes $T(n')$ dans l'équation 6.17). D'autre part, le pipeline selon une seconde dimension permet de réduire le retard de démarrage des processeurs. La figure 6.18 montre par exemple que chaque processeur sur la dimension k peut démarrer plus tôt grâce au pipeline selon la dimension i.

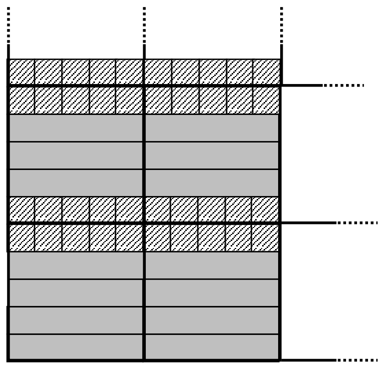


FIG. 6.17 – *2 dimensions pipelinées sur une matrice 2D.*

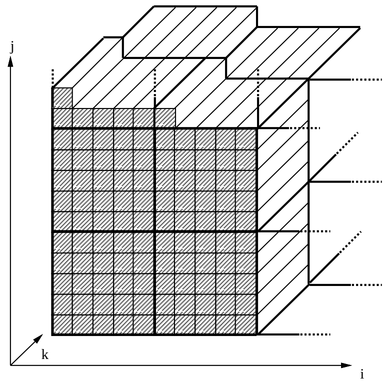


FIG. 6.18 – *2 dimensions pipelinées sur une matrice 3D.*

6.3 Implémentation dans le compilateur HPF - `adaptor`

L'utilisation du pipeline suppose l'ajout de communications de bas niveau qui rendent la programmation difficile et vont à l'encontre de l'idée de parallélisation automatique. D'autre part, l'exploitation « manuelle » du pipeline suppose que le programmeur soit capable d'identifier les schémas de type pipeline au sein de son programme. L'intégration dans un compilateur est donc nécessaire pour augmenter le champs d'application de cette technique tout en masquant sa complexité. J'ai implémenté cette technique dans le compilateur HPF ADAPTOR développé par Thomas Brandes (voir la description du compilateur donnée au chapitre 4).

Approche proposée L'implémentation dans ADAPTOR est fondée sur l'utilisation d'un *driver runtime* inséré dans la librairie DALIB. Cette approche a été préférée à la génération du code directement par le compilateur (proposée par exemple dans [Tse93] ou [EJLC97]). Elle permet, en effet, de traiter avec succès le cas d'une distribution inconnue à la compilation et facilite l'utilisation d'optimisations dépendantes de la machine. Un appel au driver, nommé `dalib_loccs_driver`, est inséré dans le programme. Les paramètres d'appel utilisés indiquent le nom de la routine locale contenant le code exécuté par chaque nœud ainsi qu'une description des tableaux à découper pour le pipeline (voir Figure 6.19). La détection automatique des boucles qui sont pipelinées peut être effectuée en utilisant une variation de l'algorithme de recherche des *Cross Processors Loops* [Tse93] adaptée au pipeline [Zor96].

L'implémentation accepte des matrices de dimension quelconque, distribuées par blocs selon une ou plusieurs directions. Le pipeline peut être appliqué selon autant de dimensions que souhaitées, mais la sélection de ces dimensions n'a pas été automatisée.

Jacobi : un exemple d'application Nous présentons ici quelques résultats obtenus pour la résolution de l'équation de Poisson par la méthode de Jacobi avec utilisation du pipeline. La figure 6.20 présente la structure du noyau de calcul. Les dépendances de données dans les deux directions empêchent la parallélisation des calculs (pas de redistribution possible). Nous avons mesuré les temps d'exécution obtenus sur un supercalculateur SP2 avec l'utilisation du pipeline, pour une distribution de la matrice B selon une dimension. Les figures 6.21, 6.22 et 6.23 présentent les résultats obtenus en faisant varier le nombre de processeurs pour respectivement des matrices carrées de taille 1680, 3360 et 5040. Les figures permettent de comparer les temps d'exécution avec et sans utilisation du pipeline. Les gains obtenus, présentés dans le tableau 6.4 sont loin des gains maximaux décrits lors de l'étude théorique de la section 6.2. En effet, ces mesures ont été effectuées en utilisant des communications bloquantes ; les communications non-bloquantes, bien que disponibles en MPI, ne sont pour l'instant implémentés ni dans les LOCCS ni dans le compilateur ADAPTOR.

Nombre de processeurs	Matrice 1680×1680	Matrice 3360×3360	Matrice 5040×5040
2	1.44	1.45	1.46
4	1.46	1.45	1.46
6	1.50	1.44	1.47
8	1.44	1.46	1.43
10	1.43	1.45	1.47
12	1.21	1.47	1.54
14	1.45	1.48	1.47
16	1.44	1.47	1.46

TAB. 6.4 – *Gains obtenus par le pipeline pour différentes tailles de matrices.*

6.4 Conclusion

Les techniques de recouvrement calculs-communications sont de plus en plus utilisées par les compilateurs data-parallèles afin de réduire le surcoût introduit par les communications. Quinn et Hatcher proposent dans [QH96] une étude analytique des gains pouvant être obtenus par recouvrement sur un processeur. Ils montrent que l'impact est faible dès que la complexité des calculs est importante comparée à la complexité des communications, pour un problème de grande taille. Nous avons proposé une extension de leurs travaux au problème du recouvrement par pipeline, en considérant le gain obtenu sur P processeurs. Nous avons montré que les recouvrements calcul-communication et calcul-calcul apportés par le pipeline aboutissent à des gains pouvant être importants. L'étude comporte une analyse de gain pour les matrices bi-dimensionnelles distribuées par blocs sur une puis deux dimensions et propose une extension aux matrices de dimension quelconque. Nous avons également implémenté le pipeline dans le compilateur HPF ADAPTOR.

Code initial

```
      REAL A(N,N), B(N.N)
!HPF$ DISTRIBUTE (*,BLOCK) :: A,B
      ...
      DO I = 2, N
         DO J = 1, N
             A(I,J) = A(I,J) + A(I,J–1)*B(I,J)
         ENDDO
      ENDDO
```

Appel au LOCCS driver

```
      CALL  DALIB_LOCCS_DRIVER  ( BLOCK , 2, 0,
                                  A(2:N,1:N), [0,1]
                                  B(2:N,1:N),[0,0] )
```

Fonction locale

```
      EXTRINSIC (HPF_LOCAL)  SUBROUTINE  BLOCK (A,B)
      REAL A(:,:), B(:,:)
!HPF$ DISTRIBUTE *(*,BLOCK) :: A,B
      DO J = lbound (A,2), ubound(A,2)
          DO I = lbound(A,1), ubound(A,2)
              A(I,J) = A(I,J) + A(I,J–1)*B(I,J)
          ENDDO
      ENDDO
      END SUBROUTINE
```

FIG. 6.19 – *Fonctionnement général du driver.*

```
 Do j
  Do i
   B(i,j) = B(i-1,j) + B(i,j-1) +
&           B(i+1,j) + B(i,j+1) + BETA*B(i,j)
 continue
```

FIG. 6.20 – *Résolution de l'équation de Poisson par la méthode de Jacobi.*

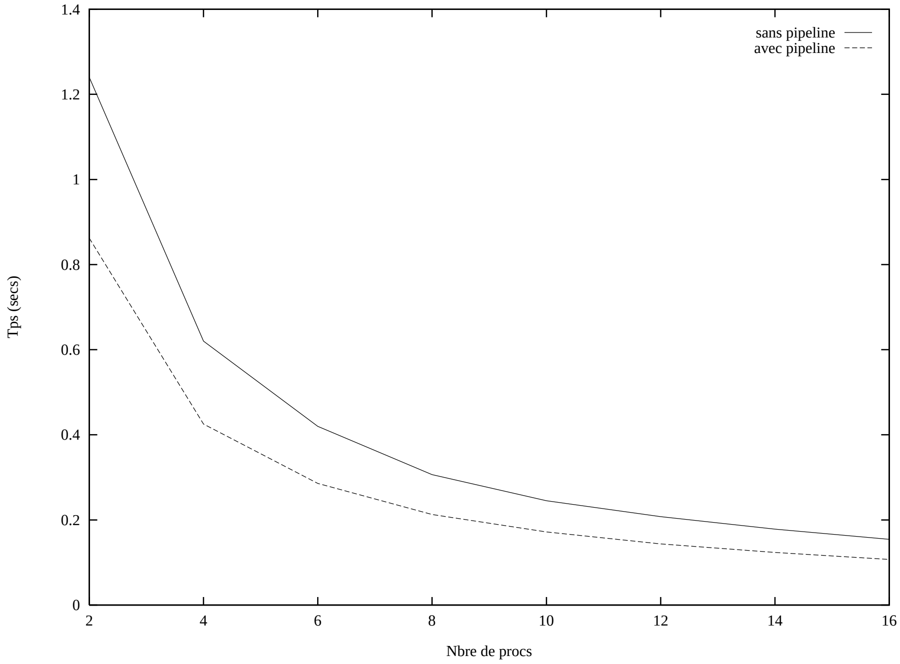


FIG. 6.21 – *Résultats obtenus pour une matrice de taille* 1680×1680.

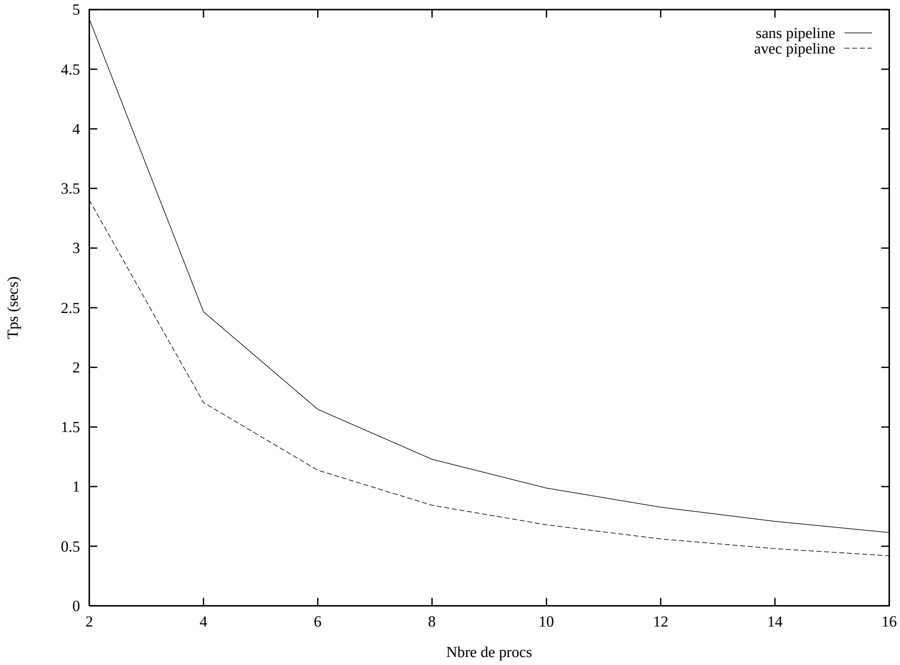


FIG. 6.22 – *Résultats obtenus pour une matrice de taille* 3360×3360.

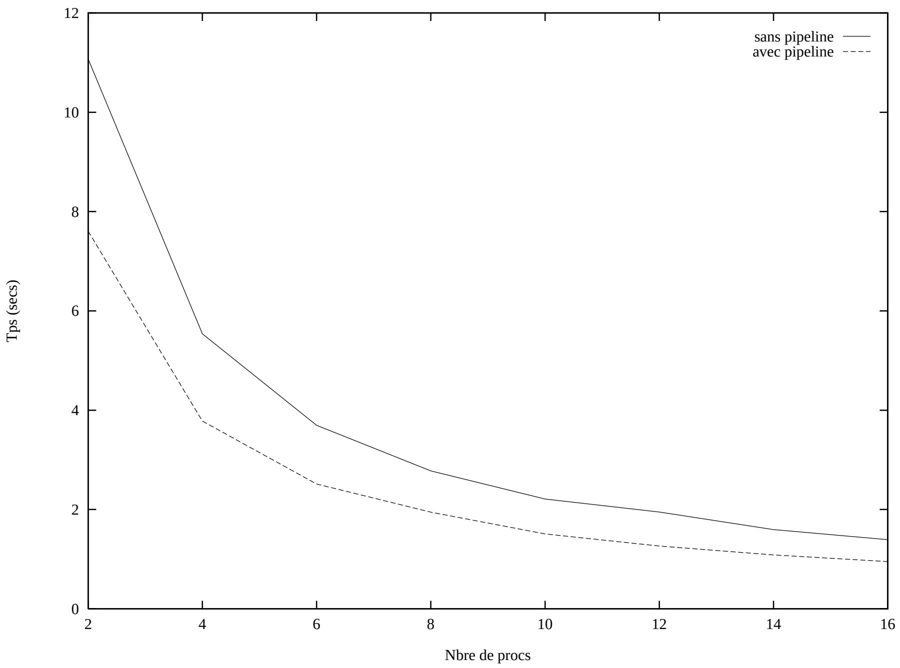


FIG. 6.23 – *Résultats obtenus pour une matrice de taille* 5040×5040.

Troisième partie

Evaluation d'expressions sur processeurs superscalaires

Chapitre 7

Évaluation d'expressions dans les applications scientifiques

7.1 Introduction

L'évaluation d'expressions se situe au cœur de la plupart des applications de calcul scientifique. En effet, la modélisation d'un phénomène physique quelconque aboutit à la construction d'une équation mathématique, à son tour traduite sous la forme d'une ou plusieurs expressions dans le programme. Or, dès lors que le domaine de travail considéré par le scientifique est important, les expressions doivent être évaluées un grand nombre de fois. La rapidité d'exécution du programme est donc fortement liée à l'efficacité d'évaluation de ces expressions.

Grâce aux propriétés algébriques telles que l'associativité et la commutativité des opérateurs, il existe en général plusieurs façons d'écrire puis d'évaluer une expression. Nous verrons, à travers les expériences présentées tout au long de cette partie, que les temps d'évaluation de deux expressions mathématiquement équivalentes peuvent être significativement différents. Cependant, le scientifique qui code son algorithme ne peut focaliser son attention uniquement sur des critères de performance. Il doit en effet respecter d'autres contraintes telles que la correction, la portabilité ou la lisibilité du code. Par ailleurs, l'auteur ne maîtrise pas toujours toutes les connaissances requises pour l'écriture d'un programme optimisé, connaissances qui évoluent avec les nouvelles architectures de machines. Une bonne maîtrise, d'une part, des techniques de compilation et, d'autre part, des capacités du matériel serait indispensable pour l'application manuelle des transformations qui sont présentées dans cette partie. Enfin, rappelons que le coût d'une optimisation manuelle par le programmeur peut devenir très important.

L'utilisation de transformations algébriques dans le but d'accélérer l'évaluation des expressions est souvent jugée par le programmeur comme « trop technique ». Or, du point de vue du compilateur, elles sont trop fréquemment considérées comme des techniques de « haut-niveau ». La plupart des compilateurs vendus aujourd'hui utilisent une représentation interne fondée sur du code 3-adresses. Ainsi, même si le passage de cette représentation à une autre sous forme d'arbre n-aire est possible, le code 3-adresses rend difficile l'exploitation des propriétés algébriques telles que la distributivité des opérateurs. Certaines opportunités d'optimisation sont donc sous-exploitées.

Par ailleurs, l'augmentation incessante des capacités offertes par les architectures modernes rend de plus en plus complexe le processus de compilation (cf. chapitre 1). Atteindre les performances maximales du matériel devient en effet de plus en plus délicat et demande la mise en place de nouvelles techniques. Les processeurs qui se trouvent actuellement sur le marché, ainsi que la nouvelle génération à venir, se caractérisent par une utilisation de plus en plus intensive du **parallélisme d'instructions** :

- la présence de plusieurs unités de calcul (entier ou flottant) permet au processeur d'exécuter simultanément plusieurs instructions ;
- le pipeline permet de lancer à chaque cycle des instructions de latence supérieure à 1 cycle (voir chapitre 1) ;

– les progrès du matériel permettent également d'exécuter des opérations complexes plus rapidement ("multiply-add", racine carré, etc.).

Grâce à ces évolutions architecturales, plusieurs opérations peuvent donc être exécutées en parallèle. Malheureusement, si les performances maximales théoriques ont considérablement augmenté, elles deviennent aussi de plus en plus compliquées à atteindre. Il est en effet difficile de « nourrir » en continu les unités de calcul (il faut par exemple 4 opérations flottantes indépendantes par cycle sur le Power2). Seules quelques applications de calcul très intensif et bien choisies par les constructeurs permettent en pratique d'obtenir les performances promises. Dans bien des cas, les applications scientifiques ne sont pas adaptées et on ne peut espérer atteindre les performances crêtes du processeur. De plus, les temps d'exécution obtenus en pratique sont souvent médiocres, loin d'exploiter les capacités maximales des processeurs. Il faut donc mettre en place de nouvelles techniques de compilation permettant de construire un code plus efficace.

Notre objectif est ici d'appliquer des transformations algébriques source-à-source pour tenter d'améliorer les temps d'exécution requis pour évaluer les expressions sur des architectures modernes. Nous cherchons à sélectionner, parmi toutes les expressions équivalentes au sens mathématique, celles qui donneront de bons résultats.

7.2 Deux exemples introductifs

Les expériences présentées ci-dessous ont été effectuées sur plusieurs processeurs de la famille RISC IBM-RS6000. Nous avons mesuré des performances sur des RS2[22] (nœud fin et nœud large) ainsi que sur des P2SC[23]. Les principales différences entre ces deux gammes de processeurs concernent la taille de la mémoire cache ainsi que la largeur et la fréquence des bus. Toutes les architectures Power2 sont composées de cinq unités :

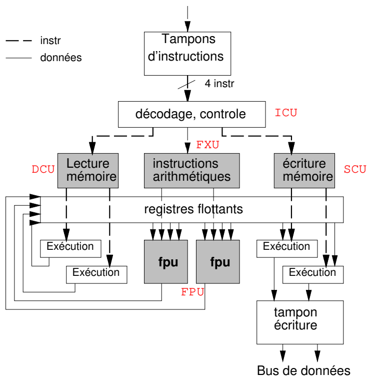


FIG. 7.1 – *Gestion des calculs flottants sur le Power2 (RS6K).*

– ICU (Instruction Control Unit) - unité chargée du décodage et du chargement des instructions (depuis le cache d'instruction). ;
– FXU (Fixed Point Unit) - unité chargée du décodage et de l'exécution de toutes les instructions, mis à part les calculs flottants et les exceptions ;
– FPU (Floating Point Unit) - unité chargée du décodage et de l'exécution des calculs flottants (voir figure 7.1) ;
– SCU (Storage Control Unit) - unité chargée de la communication avec les autres processeurs, des accès à la mémoire principale et des Entrées-Sorties ;
– DCU (Data Cache Unit) - unité contrôlant l'accès à la mémoire cache.

22. Power2 « multichip ».
23. Power2 « super chip ».

La FXU et la FPU contiennent deux unités de calcul [SM95, BGM90, HFH94] pouvant chacune décoder et exécuter une instruction (respectivement entière ou flottante) à chaque cycle d'horloge ; les opérations du Power2 sont pipelinées. La FPU peut également exécuter une instruction spécifique appelée *floating-point multiply-add* (*fma*) qui permet d'évaluer une expression du type $a \times b + c$. La latence de cette instruction est de deux cycles, c'est-à-dire la même que celle d'une addition ou d'une multiplication. La performance crête en MFLOP/s est donc égale à quatre fois la fréquence du processeur. Notons par ailleurs que le Power2 supporte deux nouvelles instructions (`quad load` et `store`) permettant de lire ou écrire en mémoire des mots de 128 bits (à des adresses contiguës).

Le compilateur natif IBM XL Fortran (`xlf`) version 4.1 a été utilisé pour les expériences. Le code optimisé a été obtenu par l'utilisation des options classiques du compilateur (`-qarch=pwr2 -O3 -qhot`) [Chi97, IBM95b]. Le préprocesseur source-à-source KAP [Kuc96] (version 3.3) pour IBM XL Fortran a également été employé sans effet notable pour les applications testées.

7.2.1 Expériences avec APPLU (SPEC FP95)

L'application que nous allons étudier ici est un solveur d'équations aux dérivées partielles[24]. APPLU est un benchmark, écrit en fortran 77 et extrait de la suite SPEC FP95. Les travaux présentés ici se concentrent sur un extrait de ce code présenté figure 7.2, mais la plupart de ses procédures contiennent du calcul intensif et ont pu également bénéficier de nos techniques (cf. chapitre 10). Cet extrait de programme met en jeu une expression, relativement grande, qui est évaluée pour chaque itération du nid de boucles. Cette expression apparaît en outre à quatre reprises dans le programme. Son évaluation naïve nécessite $12 \times 5n^3$ additions[25] et $30 \times 5n^3$ multiplications (voir Figure 7.3). Nous nous concentrons ici sur l'évaluation de cette expression et n'évoquons donc pas les diverses autres optimisations possibles concernant l'ordre de boucles, l'évaluation des bornes de boucles, etc.

```
[...]
do j = 2, ny - 1
 eta = ( dfloat(j-1) ) / ( ny - 1 )
 do i = 2, nx - 1
  xi = ( dfloat(i-1) ) / ( nx - 1 )
  do k = 1, nz
   zeta = ( dfloat(k-1) ) / ( nz - 1 )
   do m = 1, 5
    ue(m,k) =  ce(m,1)
            + ce(m,2) * xi
            + ce(m,3) * eta
            + ce(m,4) * zeta
            + ce(m,5) * xi * xi
            + ce(m,6) * eta * eta
            + ce(m,7) * zeta * zeta
            + ce(m,8) * xi * xi * xi
            + ce(m,9) * eta * eta * eta
            + ce(m,10) * zeta * zeta * zeta
            + ce(m,11) * xi * xi * xi * xi
            + ce(m,12) * eta * eta * eta * eta
            + ce(m,13) * zeta * zeta * zeta * zeta
   enddo
   [...]
continue
```

FIG. 7.2 – *Un extrait d'APPLU (SpecFP95).*

24. « five coupled parabolic-elliptic partial differential equations »
25. On suppose pour simplifier que nx=ny=nz=n.

Version	Temps d'évaluation (seconds)	Gain	Figure
Initiale	0.78	1.0	7.3
Déplacement de code	0.49	1.59	7.4
Déplacement de code + fma	0.48	1.63	7.5
Factorisation	0.35	2.2	7.6
Factorisation + fma	0.26	3.0	7.7

TAB. 7.1 – *Un extrait de Applu (sur P2SC 595 à 160 MHz).*

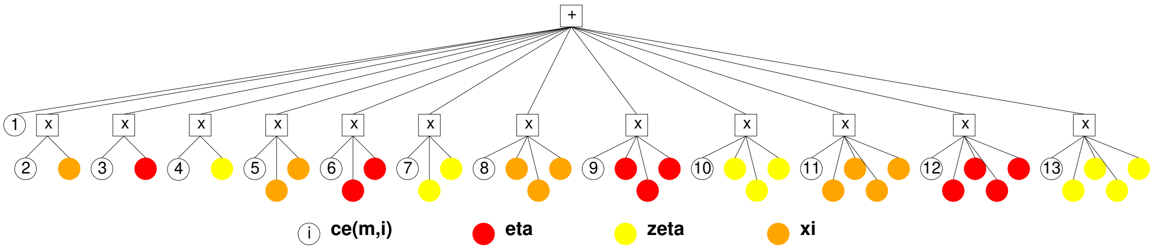


FIG. 7.3 – *APPLU - Structure originale.*

Les performances peuvent être simplement améliorées en effectuant un déplacement de code invariant. Les produits sur les variables *xi*, *eta* and *zeta* sont déplacés à l'extérieur des boucles. Le nombre de multiplications est ainsi réduit à $12 \times 5n^3 + 3n^3 + 3n^2 + 3n$ (voir figure 7.4). Le nombre d'additions ne change pas. Cette optimisation n'est cependant pas appliquée par le compilateur IBM `xlf`, car il effectue une atomisation associative-gauche (i.e. $x \times y \times z = (x \times y) \times z$). Les performances initiales obtenues sont donc très médiocres.

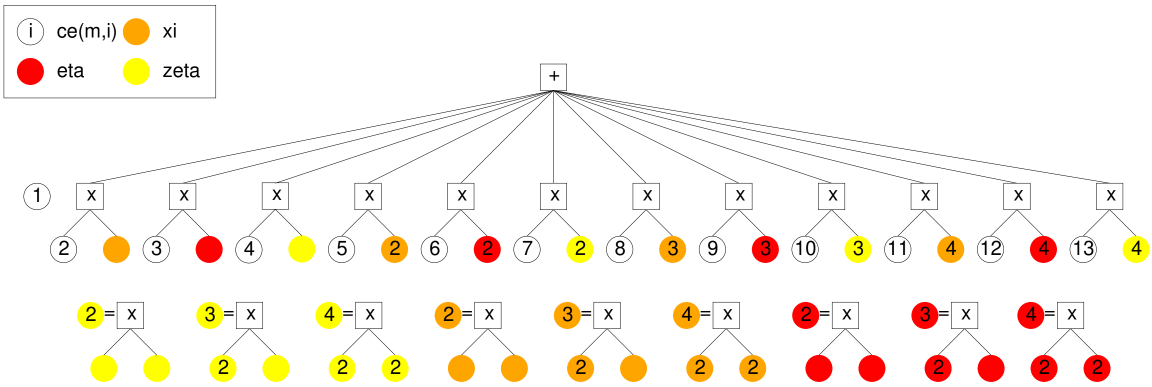


FIG. 7.4 – *APPLU - Déplacement de code invariant.*

L'utilisation de l'instruction *fma* permet de diminuer un peu plus le nombre d'opérations. Comme le montre la figure 7.5, il reste alors $3n^3 + 3n^2 + 3n$ multiplications et $12 \times 5n^3$ *fma*. Le tableau 7.1 résume les résultats expérimentaux que nous avons obtenus avec cet extrait de APPLU, pour $\mathrm{nx} = \mathrm{ny} = \mathrm{nz} = 100$ et des nombres flottants codés en double précision. L'application manuelle du déplacement de code nous a permis d'obtenir un gain en performance d'un facteur 1.63. L'ajout de parenthèses pour aider le compilateur à extraire les *fma* apporte un gain supplémentaire relativement faible. Le compilateur s'en sort en effet assez bien pour détecter les *fma* dans la version factorisée mais notre intervention est tout de même profitable avec un gain de 1.63 au lieu de 1.59.

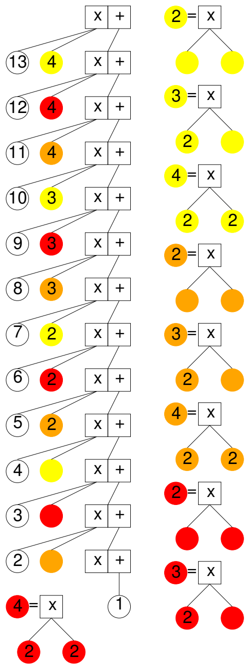


FIG. 7.5 – *APPLU - Déplacement de code invariant - extraction des* fma.

Grâce à la distributivité des opérateurs, le nombre d'opérations peut être encore réduit. La figure 7.6 montre la nouvelle structure de l'expression après factorisation [26]. Le nombre de multiplications est réduit à $12 \times 5n^3$, mais le nombre d'additions reste inchangé. Notons également que cette transformation permet d'obtenir un arbre mieux équilibré qui, comme nous le verrons plus tard, autorise une meilleure utilisation du parallélisme d'instructions. Cette transformation algébrique permet d'améliorer les performances d'un facteur 2.2.

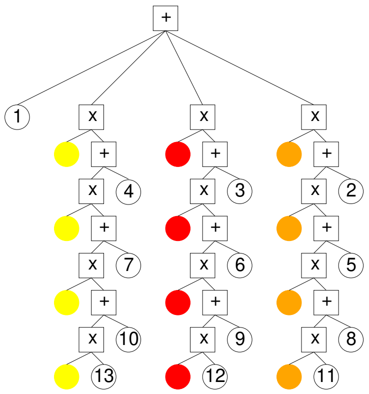


FIG. 7.6 – *APPLU - Factorisation.*

Le bénéfice apporté par l'utilisation des instructions de type *fma* après l'étape de factorisation est indiqué figure 7.7. Cette transformation réduit le nombre total d'opérations à $12 \times 5n^3$ *fma*. Le compilateur IBM `xlf` tente d'extraire du code ce genre d'opérations, mais il ne réussit pas à les trouver toutes. Par ailleurs, il ne profite pas des quelques choix disponibles pour sélectionner les expressions les mieux équilibrées. Nous avons pu améliorer les performances d'un facteur 3 en insérant manuellement

26. Cette transformation appliquée à un polynôme correspond à une évaluation par la méthode de Hörner.

des parenthèses afin d'aider le compilateur dans la recherche des *fma* et le guider dans ses choix. Une expression de la forme $x_1 \times (x_2 \times x_3 \times x_4 + x_5) + x_6 + x_7 \times x_8$ sera par exemple réécrite sous la forme $((x_1 \times (x_2 \times (x_3 \times x_4) + x_5) + x_6) + x_7 \times x_8)$.

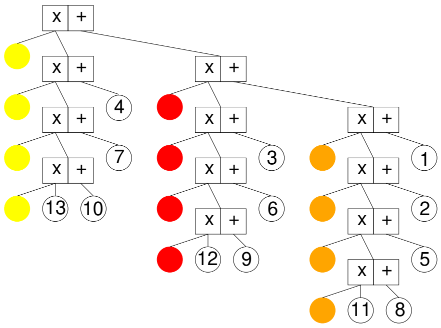


FIG. 7.7 – *APPLU - Factorisation + extraction des* fma.

Ces résultats démontrent l'augmentation significative de performances que peuvent apporter de simples transformations de la structure d'une expression. L'efficacité d'évaluation des expressions peut donc être améliorée grâce à l'utilisation des propriétés algébriques des opérateurs manipulés. L'expression obtenue grâce à la factorisation puis à l'extraction des *fma* permet une évaluation plus efficace qu'avec la version initiale sur l'architecture Power2. Cependant, il est intéressant de constater que l'expression après transformation ne contient toujours pas suffisamment de parallélisme, c'est-à-dire de calculs indépendants, pour « nourrir » les deux unités de calcul flottant en continu.

Les résultats de ces expériences sont utilisés par Alexandre MASSELOT dans le cadre d'un cours sur l'optimisation de programmes à l'Université de Genève.

7.2.2 Expériences avec Onde24 (IFP)

L'application Onde24 est utilisée pour la simulation d'une propagation d'onde en 2-dimensions (voir chapitre 2). La portion de code présentée figure 7.8 correspond au calcul de la solution en tout point du domaine de taille $NP \times NP$ (à partir des valeurs voisines et de la vitesse de propagation de l'onde). Elle occupe à elle seule la majeure partie du temps d'exécution de toute l'application (environ 75%). Évaluer cette expression nécessite l'évaluation de 10 additions[27] et 4 multiplications (cf. figure 7.8-c).

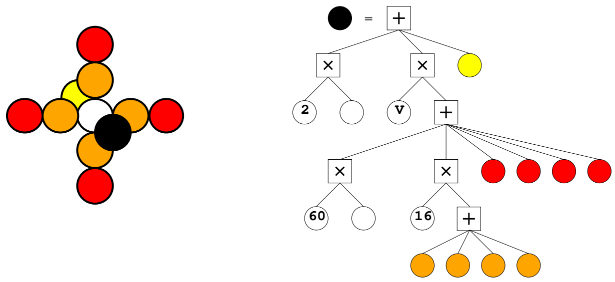
Nous avons vu ci-dessus que le Power2 peut exécuter deux *fma* à chaque cycle. Dans ces conditions, l'expression demande au minimum 5 cycles pour être évaluée (deux additions parmi 10 à chaque cycle). Cela donne pour ce code une performance maximale en MFLOP/s égale à $\frac{14}{5}$ fois la fréquence donnée en MHz. Or, cette performance crête théorique comparée aux performances réellement mesurées (données par le tableau 7.2) indique que le compilateur IBM n'a pas pu générer un code performant. En fait, le code assembleur généré par `xlf` montre que ce dernier est incapable de générer les instructions *fma* efficacement (seuls 3 *fma* sur 4 sont trouvés et l'équilibrage de l'arbre est médiocre).

L'examen de la figure 7.8 montre que le terme U(I,J,KM) apparaît à deux reprises au sein d'un produit dans l'expression. Donc, au lieu de factoriser l'expression sur la variable V, il est également possible de construire une nouvelle expression, mathématiquement équivalente, fondée sur la factorisation de U(I,J,KM). La figure 7.9 illustre cette propriété, avec en partie gauche (a), l'expression factorisée à partir de V(I,J) et, en partie droite (b), l'expression factorisée à partir de U(I,J,KM). Il y a, de plus, pour chacune des factorisations, quatre multiplications placées sous des additions (et donc quatre instructions *fma* potentielles).

27. les soustractions présentes dans l'expression initiale sont transformées en additions par utilisation de l'opérateur moins unaire. Ces derniers ne sont pas représentés sur les figures.

```
[...]
 DO J = 3,NP-2
   DO I = 3, NP-2
     U(I,J,KP) =
$       2 * U(I,J,KM) - U(I,J,KP)
$       - V(I,J) * ( 60*U(I,J,KM)
$       -16*(U(I+1,J,KM) + U(I-1,J,KM)
$       + U(I,J-1,KM) + U(I,J+1,KM))
$       + U(I+2,J,KM) + U(I-2,J,KM)
$       + U(I,J-2,KM) + U(I,J+2,KM))
   ENDDO
 ENDDO
[...]
```

(a) code



(b) pattern (c) structure originale (version 0)

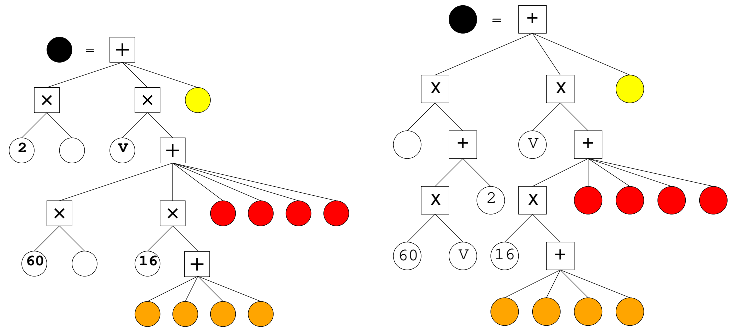
FIG. 7.8 – *Noyau de calcul de ONDE24.*

	Power2 590 (nœud large)	P2SC 595 (nœud fin)	P2SC 595 (nœud fin)
Fréquence (MHz)	66	120	160
MFLOP/s crête	266	480	640
MFLOP/s max	186	336	448
MFLOP/s mesuré	44	70	91

TAB. 7.2 – *Performances attendues et mesurées pour ONDE24.*

Considérons la sous-expression proposée figure 7.10 ; c'est une addition contenant trois opérandes dont deux sont des multiplications. En supposant que la première multiplication est combinée avec l'addition pour former un premier *fma*, il est possible de créer trois expressions différentes. Selon la position dans l'arbre des deux termes restants, un nouveau *fma* pourra ou non être créé. Cet exemple tout simple montre que le nombre d'expressions mathématiquement équivalentes construites à partir de l'extraction des *fma* peut devenir vite important ; une méthode de dénombrements est proposée à la section 7.3.

Afin de guider le compilateur, les instructions *fma* sont suggérées en insérant dans le code des parenthèses appropriées. Nos expériences montrent que les instructions *fma* sont très efficaces mais aussi très complexes à extraire. Il existe en effet un (trop) grand nombre de façons différentes d'écrire une expression. La figure 7.11 donne deux résultats obtenus sur un P2SC 595 cadencé à 160 MHz pour deux façons différentes d'écrire l'expression extraite de Onde24. L'analyse des performances obtenues est intéressante pour la compréhension de l'impact des transformations algébriques. En effet, il y a une différence de performance significative (19 MFLOP/s) entre ces deux versions de l'expression qui pourtant effectuent le même nombre d'opérations. La partie grisée sur la figure représente le sous-arbre commun aux deux expressions ; les différences se situent donc dans les parties non grisées. La position « au plus haut »



(a) Factorisation avec V(I,J) (b) Factorisation avec U(I,J,KM)

FIG. 7.9 – *Deux factorisations pour ONDE24.*

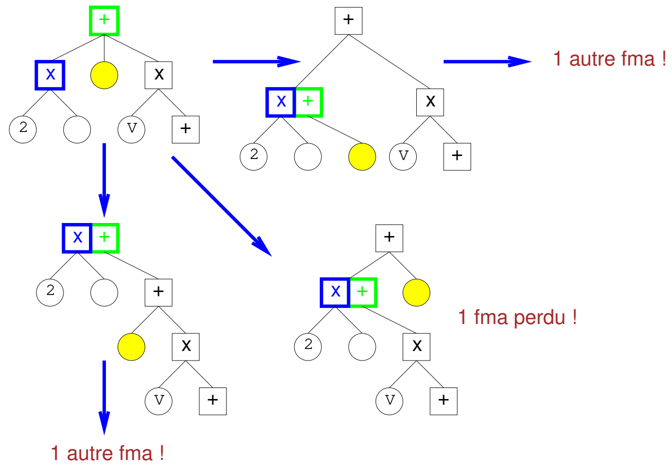


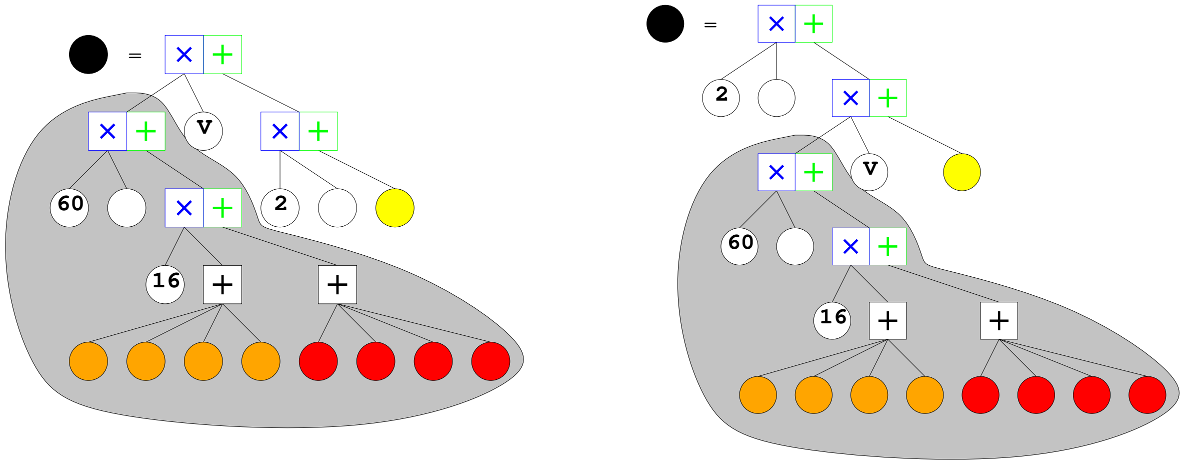
FIG. 7.10 – *Quelques choix pour l'extraction des* fma.

ou non de cette sous-expression grisée semble importante. En effet, l'étude du chemin critique [28] pour les deux expressions permet d'expliquer (en partie) les différences de performances mesurées. Le chemin critique pour l'expression de gauche vaut 5, alors qu'il vaut 6 pour celui de l'expression en partie droite de la figure.

La figure 7.12 donne deux nouveaux exemples permettant de démontrer une nouvelle fois l'impact de la structure de l'expression sur le temps d'évaluation. L'expression représentée sur la partie gauche de la figure correspond à la solution trouvée par le compilateur `xlf` pour l'extraction des *fma* (sans aide extérieure). La seconde expression, trouvée de manière automatique, est celle qui nous a permis de mesurer les meilleures performances. Deux facteurs principaux expliquent cette écart de 60 MFLOP/s. Tout d'abord, l'expression de gauche contient un *fma* de moins par rapport à celle de droite (il faut donc effectuer une opération de plus pour l'évaluer !). D'autre part, la profondeur et l'équilibrage des deux arbres sont très éloignés. Une amélioration des performances d'environ 60% a donc été atteinte par une simple transformation algébrique de l'expression.

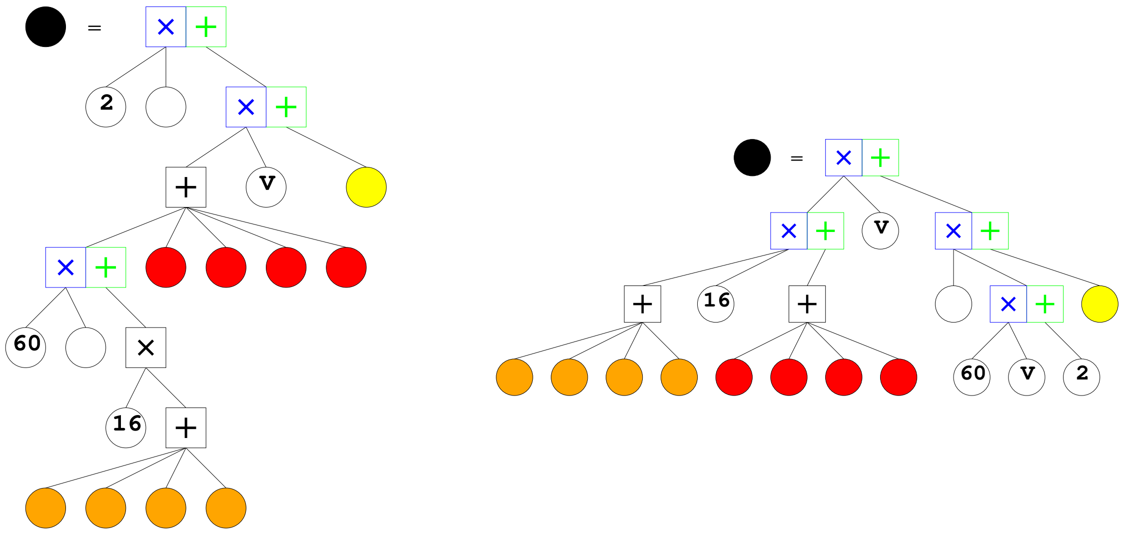
Il existe beaucoup de constructions possibles pour cette expression de Onde24 ; nous avons en effet dénombré **69300 expressions équivalentes** (cf. section 7.3). Afin de mieux illustrer le large éventail de performances obtenues pour cette expression, la figure 7.13 présente dix versions possibles obtenues à partir de l'une ou l'autre des deux factorisations. Pour chaque sous-expression de la forme $x_1 \times y_1 + x_2 \times y_2$,

28. chemin de longueur maximale partant de la racine de l'arbre



149 MFLOP/s **130 MFLOP/s**

FIG. 7.11 – *Analyse de résultats obtenus sur un P2SC 595 (160 MHz).*



102 MFLOP/s **162 MFLOP/s**

FIG. 7.12 – *Analyse de résultats obtenus sur un P2SC 595 (160 MHz).*

il faut choisir entre les deux constructions[29] fma$(x_1, y_1, (x_2 \times y_2))$ et fma$(x_2, y_2, (x_1 \times y_1))$. Cela nous donne donc pour l'expression factorisée sur V(I,J), quatre squelettes d'expressions possibles représentés sur les figures 7.13-1, 7.13-2, 7.13-3 et 7.13-4. De la même façon, la seconde factorisation donne lieu aux configurations 7.13-5 et 7.13-6. De plus, l'associativité de l'opérateur + permet d'obtenir encore un grand nombre de variations. Les figure 7.13-1a, 7.13-1b et 7.13-1c ne sont par exemple que des variation du squelette 7.13-1 pour lesquelles une addition (celle des quatre voisins situés à distance 2) est déplacée dans l'arbre. Les résultats obtenus pour ces diverses constructions sont synthétisés dans le tableau 7.3. Les expressions 7.13-1b et 7.13-5 donnent les meilleurs résultats pour respectivement la première (V(I,J)) et la seconde (U(I,J,KM)) factorisation. Notons également que l'écart entre la meilleure performance obtenue et la plus mauvaise (7.13-4a) pour un nombre identique d'opérations est de 20%.

Version	**P2SC 595** (160 MHz) MFLOP/s	**Power2 590** (66 MHz) MFLOP/s
initial	**102**	**47**
1	151	85
1-a	141	77
1-b	**164**	**98**
1-c	144	79
2	149	83
3	137	73
4	130	69
4a	*126*	*62*
5	**162**	**94**
6	142	78

TAB. 7.3 – *Performance du noyau de ONDE24.*

7.3 Analyse du problème

Les deux exemples proposés ci-dessus montrent que de très simples transformations peuvent améliorer de manière significative les performances d'évaluation d'une expression. Cependant, le nombre d'expressions équivalentes au sens algébrique est très important. En effet, l'utilisation de l'associativité et de la commutativité des opérateurs entraîne une véritable explosion combinatoire du nombre d'expressions.

7.3.1 Dénombrements

Comme nous cherchons à nous déplacer dans l'espace des expressions équivalentes au sens algébrique, il est intéressant d'obtenir des informations sur la taille de cet espace. En particulier, l'utilisation de l'associativité des opérateurs permet de construire différentes versions d'une même expression qui sont mathématiquement équivalentes mais structurellement différentes ; la position des opérandes dans l'arbre représentant l'expression varie par exemple. Nous souhaitons donc dénombrer le nombre d'arbre structurellement différents que l'on peut construire à partir d'une opération contenant n opérandes distincts. Les arbres équivalents à commutativité près ne seront comptabilisés qu'une seule fois car ils sont structurellement très proches.

Nombre d'arbres binaires différents à la commutativité près

Soit $\oplus$ un opérateur associatif-commutatif et $X_1, X_2, \ldots X_n$ des opérandes distincts et ordonnés. Nous appelons $\mathcal{P}(\oplus, n)$ la procédure de construction de l'ensemble des arbres binaires commutatifs différents

29. L'opération *fma* représentant le calcul $a \times b + c$ sera notée sous la forme fma(a,b,c)

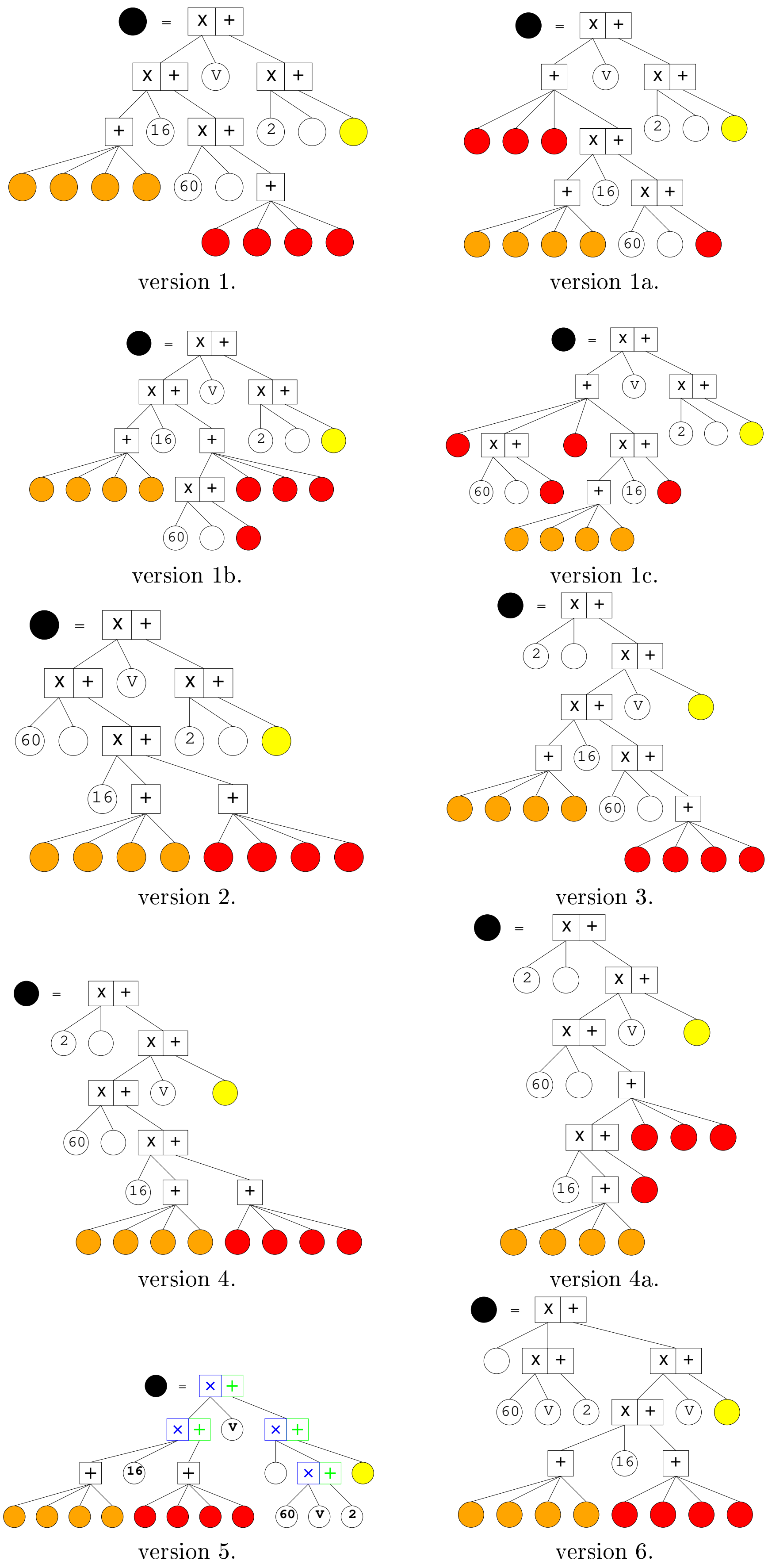


FIG. 7.13 – *Quelques constructions possibles.*

contenant n opérandes et $n-1$ opérateurs commutatifs identiques, décrite par l'algorithme 1. La construction est faite récursivement par introduction à chaque étape d'un nouvel opérande à toutes les positions possibles dans les arbres construits aux étapes précédentes. Nous appelons « arbres binaires différents à la commutativité près », deux arbres tels qu'il est impossible de passer de l'un à l'autre par application de la commutativité des opérateurs. Les arbres représentés sur la figure 7.14 ne sont comptabilisés qu'une seule fois car l'application de la commutativité permet de passer des uns aux autres. Le dénombrement proposé s'intéresse donc aux expressions telles que la profondeur des opérandes dans l'arbre est différente. Notons qu'il n'existe bien évidemment qu'un seul arbre *différent à commutativité et associativité près.*

Algorithme 1 Construction d'arbres binaires commutatifs.

Fonction $\mathcal{P}(\oplus, n)$
- ▷ - *$\oplus$: un opérateur commutatif*
- ▷ - *n : le nombre d'opérandes*
- ▷ - *retourne E_n l'ensemble des arbres construits*

Soit $\{X_1, X_2, \ldots, X_n\}$ les n opérandes distincts ordonnés
Si $n = 1$ **Alors**
  Retourner $\emptyset$
Fin Si
Si $n = 2$ **Alors**
  Retourner $\{\oplus(X_1, X_2)\}$
Sinon
  $E_{n-1} \leftarrow \mathcal{P}(\oplus, n-1)$, $E_n \leftarrow \emptyset$
  Pour tout $e_{n-1} \in E_{n-1}$ **Faire** ▷ *arbre binaire construit avec $\mathcal{P}(\oplus, n-1)$*
    Pour tout Y de l'arbre e_{n-1} **Faire** ▷ *Y est un opérande ou un sous-arbre quelconque*
      ▷ *Substituer Y par $\oplus(Y, X_n)$ dans l'arbre e_{n-1}*
      Ajouter l'arbre $e_{n-1}[Y \leftarrow \oplus(Y, X_n)]$ dans E_n
    Fin Pour
  Fin Pour
Fin Si
Fin

Lemme 7.1
La procédure de construction $\mathcal{P}(\oplus, n)$ ne construit aucun doublon.

Preuve 7.1

Cette propriété peut être montrée par récurrence. La propriété est vraie pour $n = 2$ (un seul arbre). Supposons qu'elle est soit pour $n = i - 1$.
Considérons un arbre e_{i-1} quelconque construit par $\mathcal{P}(\oplus, i-1)$. Par construction, le lieu d'insertion du nouvel opérande X_i est chaque fois différent et le nouvel opérande est toujours placé à droite de l'opérateur. Il est donc impossible de créer deux fois le même arbre ou deux arbres équivalents par commutativité en partant d'un unique e_{i-1}.
Soient e_{i-1} et e'_{i-1} deux arbres quelconques construits par $\mathcal{P}(\oplus, i-1)$. Par hypothèse, ces deux arbres sont différents. Or, la procédure $\mathcal{P}(\oplus, i)$ insère un opérande X_i qui n'appartient à aucun des arbres construits par $\mathcal{P}(\oplus, i-1)$. La différence existant entre e_{i-1} et e'_{i-1} sera donc toujours présente dans les nouveaux arbres créés. *Q.E.D.*

Lemme 7.2
*La procédure de construction $\mathcal{P}(\oplus, n)$ construit **tous** les arbres binaires différents à la commutativité près.*

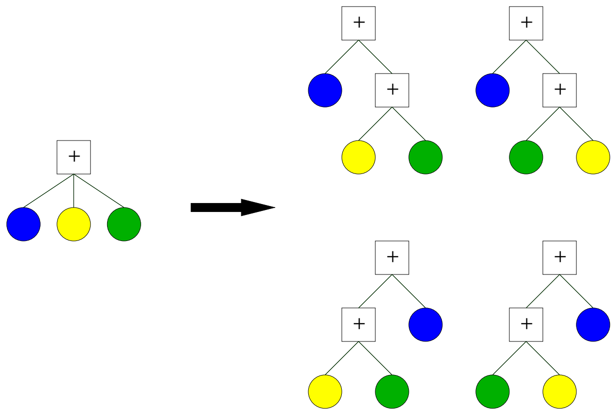


FIG. 7.14 – *Quatres arbres équivalents par commutativité.*

Preuve 7.2

Montrons, une nouvelle fois par récurrence, que la construction proposée par la procédure de construction $\mathcal{P}(\oplus, n)$ est exhaustive (tous les arbres sont construits).
Tous les arbres contenant 2 opérandes sont construits. Supposons que tous les arbres binaires différents contenant $i-1$ opérandes et $i-2$ opérateurs sont construits par $\mathcal{P}(\oplus, i-1)$.
Soit un arbre binaire quelconque e_i contenant i opérandes. Choisissons un sous-arbre e_{i-1} quelconque de e_i ne contenant que $i-1$ opérandes ; cet arbre est par hypothèse construit par $\mathcal{P}(\oplus, i-1)$.
Soient X l'opérande appartenant à e_i mais pas à e_{i-1} et Y le sous-arbre tel que $\oplus(X, Y)$ ou $\oplus(Y, X)$ est un sous-arbre de e_i. L'application de la procédure $\mathcal{P}(\oplus, i)$ permet de substituer Y par $\oplus(Y, X)$ dans e_{i-1} pour reconstruire e_i. Il est donc possible de construire n'importe quel arbre par application de cette procédure de construction. *Q.E.D.*

Lemme 7.3
Le nombre de subsitutions effectuées, dans un arbre donné e_{n-1}, par la procédure de construction $\mathcal{P}(\oplus, n)$ est $2n-3$.

Preuve 7.3

La procédure applique une substitution dans l'arbre soit au niveau des $n-1$ opérandes (feuilles) soit au niveau des $n-2$ opérations (nœuds internes). *Q.E.D.*

Théorème 7.1
Soit une opération formée de n opérandes et $n-1$ occurences d'un opérateur binaire associatif-commutatif. Le nombre d'arbres binaires différents à la commutativité près qu'il est possible de construire à partir de cette opération (noté $B(n)$) est :

$$B(n) = \frac{(2n-3)!}{2^{n-2}(n-2)!} \sim n! \times 2^n$$

Preuve 7.1

En utilisant la procédure de construction $\mathcal{P}(\oplus, n)$, on peut construire $(2n-3) \times B(n-1)$ arbres différents. Nous avons donc :

$$B(n) = (2n-3) \times (2n-5) \times \ldots \times 3 \times 1$$

Ce qui se ramène facilement à

$$B(n) = \frac{(2n-3)!}{2^{n-2}(n-2)!}$$

La formule de Stirling $n! \approx e^{n \times ln(n)+n}$ permet de transformer ce résultat de complexité sous une forme plus facile à interpréter. En supposant que n tend vers l'infini, on obtient alors :

$$\begin{aligned} B(n) &\sim \frac{e^{(2n-3)\times ln(2n-3)+(2n-3)}}{e^{(n-2)\times ln(2)} \times e^{(n-2)\times ln(n-2)+(n-2)}} \\ B(n) &\sim e^{2n\times ln(2n)+(2n)-n\times ln(2)-n\times ln(n)-n} \\ B(n) &\sim e^{n\times ln(2)+n\times ln(n)+n} \\ B(n) &\sim n! \times 2^n \end{aligned} \tag{7.1}$$

Q.E.D.

Prise en compte des *fma*

Le nombre d'arbres binaires commutatifs différents que nous venons de comptabiliser est déjà très (trop) important. Or, la présence d'opérations du type *fma* augmente encore ce nombre. Examinons, par exemple, l'expression représentée sur la figure 7.15. En considérant les deux multiplications comme de simples opérandes de l'addition, l'expression n'est comptabilisée qu'une seule fois par la technique présentée ci-dessus ($B(2) = 1$). Or, en raison des conflits sur les *fma*, deux arbres différents peuvent être construits.

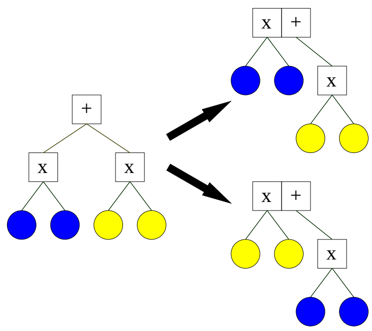


FIG. 7.15 – *Constructions d'arbres binaires en présence de* fma.

La prise en compte des conflits de construction des *fma* (voir chapitre 9) ajoute donc de nouvelles constructions possibles que nous allons chercher à comptabiliser. Nous proposons ici une méthode de dénombrement fondée sur la prise en compte des différents cas de conflits entre les *fma*.

Proposition 7.1
Soit une addition de n opérandes dont m multiplications. Le nombre d'arbre binaires commutatifs équivalents contenant au plus m fma *(noté Q(n,m)) est :*

$$Q(n,m) = B(n) + \sum_{i=1}^{m/2} \frac{m!}{(m-2i)!i!2^i} B(n-i)$$

Explications 7.1

Si l'expression manipulée ne contient qu'une seule multiplication, il n'y a pas de conflit. On a donc $Q(n,1) = B(n)$.

Si l'expression contient 2 multiplications, un conflit apparaît pour tous les arbres tels que les deux multiplications sont placées sous la même addition binaire. Pour chaque situation de conflit, il existe en fait deux arbres différents (cf. figure 7.15). Une de ces deux configurations possibles étant déjà comptabilisée dans le terme B(n), nous obtenons $Q(n,2) = B(n) + 1 \times$

$B(n-1)$. Le terme $B(n-1)$ représente ici le nombre d'arbres présentant un conflit entre deux *fma*.
De la même manière, pour $m = 3$, nous aurons à considérer tous les cas de conflits entre deux *fma* compte tenu de la présence des 3 multiplications. En nommant m_i la i^{eme} multiplication, les conflits apparaissent entre (m_1, m_2), (m_1, m_3) et (m_2, m_3). Nous avons donc $Q(n,3) = B(n) + 3 \times B(n-1)$. Encore une fois, $B(n-1)$ représente le nombre d'arbres contenant un conflit de factorisation.
Lorsqu'il y a 4 multiplications, le nombre d'arbres augmente encore. Il faut notamment prendre en compte les situations dans lesquelles deux conflits apparaissent pour un même arbre. Il est alors possible de construire quatre variations de cet arbre selon les multiplications choisies pour construire les *fma*. Il faut donc comptabiliser 4 fois cet arbre pour le calcul de Q(n,4). Cependant, cet arbre particulier a déjà été comptabilisé une fois dans $B(n)$ et 2 fois lors de l'ajout des conflits sur 2 multiplications. Pour m_1,m_2,m_3 et m_4, nous obtenons 6 paires différentes de 2 multiplications (conflit simple) et 3 groupes de 2 paires de multiplications ($[(m_1, m_2), (m_3, m_4)]$, $[(m_1, m_3), (m_2, m_4)]$ et $[(m_1, m_4), (m_2, m_3)]$) formant un conflit « double ». Nous obtenons donc $Q(n,4) = B(n) + 6 \times B(n-1) + 3 \times B(n-2)$.
Ce dénombrement se généralise à m multiplications. Tous les conflits simples (groupe de 1 paire), doubles (groupes de 2 paires), triples (groupes de 3 paires), etc... sont ajoutés au dénombrement du nombre d'arbres sans conflits. Ce qui nous donne la formule suivante :

$$Q(n,m) = B(n) + \sum_{i=1}^{m/2} \text{nbpaire}(m,i) \times B(n-i)$$

avec nbpaire(m,i) le nombre de groupes de i paires (non ordonnées) qu'il est possible de construire avec m multiplications. La sommation ne se fait que jusqu'à $\frac{m}{2}$ car on cherche à comptabiliser les conflits entre paires.
Pour construire i paires, il faut manipuler $2i$ éléments. Soit A_m^{2i} le nombre d'arrangements possibles des $2i$ éléments parmi m. Nous avons donc $A_m^{2i} = \frac{m!}{(m-2i)!}$.
Chacun de ces arrangements contient $2i$ éléments. Construisons les paires ordonnées à partir des éléments $2j-1$ et $2j$ pour j entre 1 et i, soit i paires ordonnées d'éléments. Pour ces i paires ordonnées, il existe $i!$ permutations possibles qui ont été dénombrées dans A_m^{2i}. Pour $i = 2$ et $m = 4$, nous aurons par exemple comptabilisé [(m1,m2)(m3,m4)] et [(m3,m4)(m1,m2)] alors qu'ils correspondent à deux arbres identiques à commutativité près. En divisant A_m^{2i} par $i!$ nous obtenons donc le nombre de groupes de i paires *ordonnées*.
Il faut donc encore retirer les 2^i permutations possibles correspondants aux i paires ordonnées pour obtenir le nombre de groupe de i paires *non ordonnées* que l'on peut construire avec m multiplications. Par exemple pour $i = 2$ et $m = 4$, nous avons comptabilisé [(m1,m2)(m3,m4)], [(m1,m2)(m4,m3)], [(m2,m1)(m3,m4)] et [(m2,m1)(m4,m3)] alors qu'une seule version est nécessaire.
Nous avons donc nbpaire$(m,i) = \frac{m!}{(m-2i)!i!2^i}$. Ce qui nous amène au dénombrement complet sous la forme suivante :

$$Q(n,m) = B(n) + \sum_{i=1}^{m/2} \frac{m!}{(m-2i)!i!2^i} B(n-i)$$

Ce résultat permet de comptabiliser tous les arbres binaires commutatifs différents constitués de n opérandes distincts dont m multiplications. Notons toutefois que tous ces arbres ne contiennent pas forcément m *fma* en raison des conflits évoqués.

Cette méthode de dénombrement permet de comptabiliser les expressions en présence de *fma*. On dénombre ici les expressions qui sont algébriquement équivalentes mais structurellement différentes, c'est-à-dire celles dont les temps d'évaluation sont potentiellement très différents. La formule obtenue nous a permis de dénombrer les expressions équivalentes que nous avons rencontrées dans les diverses applications traitées. Notons enfin que la prise en compte des *fma* augmente le nombre d'arbres puisque l'on a $\forall m, Q(n,m) \geq B(n)$.

7.3.2 État de l'art

Le travail présenté tout au long de cette partie repose sur l'utilisation de transformations algébriques pour accélérer l'évaluation des expressions sur des processeurs avec parallélisme d'instructions. A notre connaissance, il n'existe aucune étude basée sur la même approche et proposant d'autres algorithmes. Par contre, l'éventail des recherches en liaison plus ou moins directe avec l'émergence d'architectures présentant du parallélisme d'instructions, avec les techniques de compilations adaptées à ces dernières ou encore avec les optimisations à base de transformations algébriques, est véritablement immense. Nous allons tenter de décrire ici les évolutions majeures constatées sans chercher à faire la liste exhaustive des publications existantes.

Évolutions architecturales

Depuis le `CDC 6600`, les capacités en terme de parallélisme d'instructions ont augmenté progressivement [Bay98]. Les unités de calcul sont devenues indépendantes, ont été répliquées et utilisent un mécanisme de pipeline (voir les explications données au chapitre 1). Cette évolution a donné naissance aux modèles d'architecture VLIW et superscalaire. La plupart des processeurs utilisés en calcul scientifique (basé essentiellement sur du calcul flottant) dispose de parallélisme d'instructions et peuvent donc bénéficier des techniques proposées. C'est le cas notamment des processeurs superscalaires actuellement sur le marché : `DIGITAL Alpha 21264` [COM99], `MIPS R10K` [Inc96], `SUN UltraSparc IIi` [SUN99] ou encore `IBM P2SC` [Chi97]. La prochaine génération avec en autres le `MIPS R12K`, le `DIGITAL Alpha 21364` ou encore l'`IBM Power3` continue dans cette même direction. Notons que les processeurs de la famille `x86` (`Intel`, `AMD` ou `Cyrix`) ne présentent que peu de parallélisme d'instructions concernant le calcul flottant ; nos techniques pourront toutefois être utilisées sur ces processeurs dans le cadre du calcul entier et notamment dans les applications multimédia [LW97].

L'instruction *fma* est également de plus en plus fréquemment utilisée par les constructeurs pour améliorer les performances [JP98]. Le premier processeur à utiliser le *fma* a été l'`INTEL i860` [Bay98]. On trouve actuellement cette instruction sur les processeurs des gammes `IBM RS6K Power`, `DIGITAL MIPS R-family`, `HP PA-RISC` [Hun95, SBKB97] ou encore `ARM` [ARM98]. Certains processeurs dédiés aux applications embarquées tels que le `HITACHI SH-4` [Hit98] ou les `Texas Instrument TMS320C62X et TMS320C67X` [Tex99] utilisent également cette instruction. Notons enfin que les extensions multimédia telles que MMX-2 [30] de Intel MAX-2 [31] de HP ou encore MDMX [32] pour le jeu d'instruction MIPS-V utilisent abondamment cette technique d'opérations fusionnées permettant d'améliorer les performances [Eri96].

Remarque 7.1
L'évolution actuelle des architectures à travers l'augmentation du parallélisme d'instructions ne fait qu'augmenter la portée des techniques proposées dans cette thèse. Toutefois, cette augmentation de la capacité de calcul des processeurs doit être accompagnée d'une amélioration des accès aux données sur disque, en mémoire principale ou encore dans le cache. En effet, pour des applications présentant par exemple des problèmes de défaut de cache trop fréquents, l'effet de l'augmentation du parallélisme d'instructions est négligeable.

Techniques de compilations

L'évolution rapide de l'architecture des processeurs suppose une réaction de la part des compilateurs. Deux nombreuses techniques sont donc mises en place pour ordonnancer les calculs, accélérer les accès mémoire ou encore améliorer l'utilisation des registres [PH96]. L'augmentation du parallélisme d'instructions est à l'origine de nombreux travaux d'optimisation. En effet, les modèles *VLIW* et *superscalaires* souffrent d'un même mal : le parallélisme directement disponible dans les programmes scientifiques est insuffisant comparé aux capacités du matériel [SJH89, MSL92, Wal93]. Des travaux dans différentes directions ont donc été menés pour tenter d'extraire ce parallélisme d'instructions qui est difficile à exploiter

30. MultiMedia eXtensions.
31. Multimedia Architectural Extensions
32. MIPS V instruction set and Digital Multimedia Extensions

bien que réellement disponible dans de nombreuses applications [LW97]. L'analyse de dépendances (statique [AS92] ou dynamique [Ha95]), l'exécution spéculative [MDS92], l'exécution gardée [CHPC95] ou encore la réduction du chemin critique de contrôle [SMJ99] sont utilisés pour tenter d'exécuter de manière concurrente des calculs issus de différentes régions indépendantes du programme. Les techniques d'ordonnancement [LE95, NN95, BCGS96] et notamment le software pipelining [EGK+94, PH96, LAV98] permettent d'augmenter le débit des opérations indépendantes que peut exécuter le matériel.

Nos transformations algébriques permettent de construire des expressions présentant un fort degré de parallélisme. La réduction du nombre de dépendances entre les opérations offre donc plus de libertés aux techniques de compilation décrites ci-dessus. L'effort à fournir pour ordonnancer les calculs de manière à obtenir un débit d'instructions suffisant sera par exemple réduit.

Les transformations basées sur l'utilisation des propriétés algébriques dont l'associativité et la commutativité ne sont guère utilisées par les compilateurs [ASU86]. En effet, la représentation interne manipulée, en général du code 3-adresses, ne met pas en évidence les opportunités d'optimisation. Les expressions étant atomisées, les optimisations se limitent à la recherche de patterns simples tels que la présence de l'élément neutre ou la réduction de force sur une opération de base. Le caractère AC des opérateurs a été utilisé dans [BCGS96] pour réduire la hauteur d'un DAG. La méthode proposée se limite cependant à la reconnaissance de quelques patterns simples qui sont ensuite réécrits. Les travaux de [Aik88] autorisent une réduction de la longueur d'une chaîne d'instructions dépendantes manipulées sous la forme d'un code 3-adresses donné (i.e. représentation éclatée d'un DAG). Ses résultat utilisent l'approche proposée dans [KMC72, Bre74, KM74, KM75] ; on recherche une borne supérieure pour le temps d'évaluation d'une expression en supposant illimité le nombre d'unités de calcul. Des algorithmes pour l'évaluation d'une expression arithmétique sur des modèles de calcul tels que PRAM existent également aujourd'hui [Ber98] mais sont restreints à un contexte non AC.

De nombreux autres travaux visent à améliorer l'évaluation des expressions. Les auteurs de [KR95, Kes98, Ert99] se penchent sur le problème de la réduction du nombre de registres utiles pour l'évaluation d'une expression (problème NP-Complet d'après [Set75]). Les travaux dans le domaine de la génération de code [CT93, Can97] concernent quant à eux le problème de la sélection d'instructions. Cependant, ces travaux ne cherchent pas à exploiter les propriétés algébriques de opérateurs et viennent donc en complément de l'approche proposée ici. Bien que nos travaux cherchent à construire des expressions plus facile à évaluer car contenant moins de dépendances, leur interaction avec les techniques de bas-niveau telle que la génération de code, l'allocation de registres ou l'ordonnancement des calculs restent à étudier plus précisément. Nous évoquerons ces interactions au chapitre 17.

L'exploitation des propriétés algébriques des opérateurs apparaît également dans le cadre de la réduction du coût global d'évaluation des expressions ; les travaux se concentrent la plupart du temps sur la multiplication [BH94] et la division [GM94, Ama97].

La recherche des opérations fusionnées comme le *fma*, à partir d'un code 3-adresses, ne peut être que limitée à un contexte simple et aucune procédure de choix entre les différentes construction n'est mise en place (voir par exemple les documentations des compilateurs MIPSPro et HP). La littérature contient en revanche des études d'optimisation manuelle visant à profiter des *fma* [LF93, CFL94, PT96].

Autres applications des transformations algébriques

Les optimisations à base de transformations algébriques existent également dans d'autres domaines. La réassociation est par exemple efficace pour améliorer les performances des requêtes dans les bases de données [GLR97]. La conception de processeurs DSP[33] en technologie VLSI[34] utilise les propriétés algébriques pour réduire la hauteur des arbres manipulées [Par95] ou encore diminuer le nombre de composants, et donc la surface de silicium nécessaire, grâce à la détection de sous-expressions [Har96].

33. Digital Signal Processing.
34. Very Large Scale Integration.

Enfin, l'arithmétique est employée pour l'optimisation des calculs d'adresses dans le cadre de la conception de MMU [35] [MCJ+97].

7.3.3 Approche envisagée

La majorité des travaux cherchant à augmenter la densité de calculs qui peuvent être exécutés de manière concurrente par un processeur superscalaire, se concentrent sur l'élimination des « barrières » de contrôle (exécution gardée, spéculative, etc.). Constatant que le parallélisme d'instructions directement exploitable dans les blocs de base est peu important, ils tentent d'augmenter la taille de ces blocs (notion de superbloc) et d'éviter les branchements. Or, nous pensons qu'avant d'utiliser ces techniques, il est également nécessaire d'extraire le parallélisme d'instructions déjà présent à l'intérieur des boucles. Quelques techniques comme le *software pipelining* permettent une avancée dans cette direction. Cependant, nos expériences ont montré que l'utilisation des propriétés algébriques des opérateurs permet d'augmenter de manière significative le parallélisme d'instructions.

Nous nous plaçons ici dans le cadre de travaux pour la compilation de programmes scientifiques. Il n'est donc guère envisageable de parcourir tout l'espace des solutions pour chacune des expressions présentes dans un programme. Il faut également effectuer une évaluation statique à priori du temps d'exécution. Notre approche sera de nature *heuristique* et tentera de sélectionner, parmi un sous-ensemble des expressions équivalentes, une solution donnant de bons résultats. Il n'est pas question ici de « tester » la qualité des expressions par une exécution réelle du programme. Nous chercherons donc à établir des critères de sélection traduisant les caractéristiques (nombre d'opérations, équilibrage, etc.) que nous souhaitons obtenir compte tenu des spécificités des architectures modernes visées et de leurs compilateurs.

Notre objectif est donc d'exploiter les propriétés algébriques des opérateurs pour réécrire les expressions selon des critères de performance. Tout d'abord, réduire le nombre d'opérations est une transformation qui est la plupart du temps bénéfique. Il peut s'avérer aussi intéressant de remplacer des opérateurs coûteux par d'autres qui calculent la même valeur plus rapidement (réduction de force). D'autre part, en raison du parallélisme d'instruction, la prise en compte de la structure de l'expression étudiée devient primordiale.

Nous avons développé un prototype de transformation d'expressions, nommé **EOLE**, et contenant quatre phases principales. Son implémentation sera présentée au chapitre 10. La première phase, qui consiste à préparer les expressions manipulées pour les trois phases suivantes, est décrite au chapitre 8. Les deux phases suivantes utilisent des heuristiques destinées à améliorer les performances d'évaluation des expressions. Les deux techniques de factorisation puis d'extraction des *fma* sont décrites au chapitre 9. Une transformation supplémentaire ayant pour objectif de construire un arbre binaire équilibré à partir de la structure de données n-aire y est également présentée. Plusieurs problèmes rencontrés à travers ces méthodes d'optimisation seront discutés au chapitre 17.

35. Memory Management Unit

Chapitre 8

Mieux exploiter les opportunités de transformations

Pour une exploitation efficace des opportunités de transformations algébriques, les expressions contenues dans un code doivent être de taille suffisamment importante et présenter un grand degré de liberté. Or, le programmeur a d'autres contraintes que celle de la performance ; il fait donc souvent des choix qui limitent ensuite les transformations possibles. De plus, la conversion des expressions sous une forme *normale* facilite leur optimisation. La recherche de propriétés spécifiques ou encore la comparaison de sous-expressions entre elles s'en trouvent alors simplifiées.

Dans cette perspective, nous avons mis en place deux processus nous permettant de mieux exploiter les opportunités de transformations pour un programme donné. La section 8.1 décrit une phase de *forward substitution* qui est appliquée en premier lieu. La normalisation et la simplification des expressions seront ensuite exposées à la section 8.2. Ce chapitre donnera également un aperçu des techniques de pattern-matching utilisées par nos travaux pour la normalisation et la simplification (section 8.3).

8.1 Forward substitution - une transformation préalable

L'application manuelle de différentes techniques d'optimisation a pour conséquence une réduction de la portabilité du programme. En effet, de très nombreuses optimisations dépendent de l'architecture utilisée. Or, la durée d'utilisation d'une application scientifique est souvent bien supérieure à la durée de vie d'une génération d'ordinateurs. Beaucoup d'applications ayant été « adaptées » pour une exécution efficace sur des calculateurs vectoriels se retrouvent par exemple mal adaptées aux caractéristiques des processeurs actuels.

Les exemples présentés au chapitre 7 ont montré que l'exploitation du parallélisme d'instructions nécessite une grande liberté dans la sélection de la structure des expressions. Or, plus les expressions sont de taille importante, plus l'éventail des transformations augmente. Nous souhaitons donc pouvoir reconstruire des expressions de grande taille à partir de codes atomisés pour diverses raisons historiques. La technique couramment utilisée est la « substitution en avant » ou *forward substitution*. Notons que cette technique peut être également utilisée pour augmenter les opportunités de réassociation présentées au chapitre 14.

Présentation de la technique Chaque expression figurant en partie droite d'une instruction d'affectation est propagée le plus loin possible vers ses utilisations. L'utilisation d'une variable (membre gauche de l'affectation) est donc remplacée par celle de l'expression dont elle contient le résultat. La validité de cette transformation suppose bien évidemment qu'aucune des références de l'expression propagée ne soit modifiée sur le chemin de propagation. Sur l'exemple de la figure 8.1, les références t_0, c et t_2 sont substituées respectivement par les expressions $a + b$, 3 et $\frac{a+b}{d}$ dans l'expression affectée à X.

Cette transformation permet d'offrir plus d'opportunités aux transformations algébriques qui seront développées par la suite. La forme plus ou moins atomisée (à la main ou automatiquement) d'un code

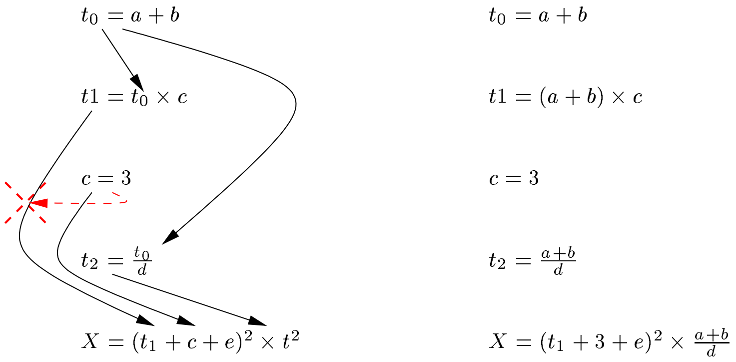


FIG. 8.1 – *Forward substitution.*

rend en effet plus complexe l'utilisation de l'associativité des opérateurs et l'extraction d'opérations complexes telles que le *fma*. Un exemple d'utilisation de la forward substitution sera donné au chapitre 11 section 11.3.

La *forward substitution* est une technique souvent employée dans le cadre de l'optimisation de programmes, mais ce n'est pas véritablement une optimisation. En effet, la duplication des expressions peut entraîner une expansion importante de la taille du code et un grand nombre de calculs inutiles. De plus, la propagation d'expressions de l'extérieur vers l'intérieur des boucles augmente le volume de calculs. Cependant, la *forward substitution* permet d'ajouter de nouvelles opportunités de transformations. De plus, l'application du déplacement de code invariant et de l'élimination de sous-expressions communes détaillés en partie IV annuleront tout ou partie des effets négatifs de cette transformation.

8.2 Normalisation et simplifications algébriques

La manipulation d'expressions équivalentes au sens mathématique est fondée sur l'utilisation des propriétés algébriques. Le point central de notre travail est évidemment l'exploitation de l'associativité et de la commutativité des opérateurs. Ces deux propriétés fondamentales nous permettent de modifier la structure des expressions. La distributivité, également au cœur de nos transformations, nous permet de réduire le nombre d'opérations réellement utiles pour évaluer une expression. D'autres propriétés algébriques telles que l'*élément neutre* ou encore l'*élément absorbant* permettent également des transformations profitables. Nous appliquons deux types de transformations : les premières visent à obtenir une forme normalisée pour les expressions tandis que les secondes recherchent des simplifications algébriques.

8.2.1 Vers une forme d'expression normalisée

La conversion d'une expression sous une forme normalisée est à la base de nombreux travaux d'optimisation existants. On trouve derrière le mot « normale » des notions souvent assez différentes. Le besoin d'obtenir une forme normale dans un système réécriture traduit par exemple la volonté d'aboutir à un état dans lequel plus aucune règle de réécriture ne peut être appliquée, un point fixe. La notion de forme normale décrit d'autres fois la recherche d'une forme unique à atteindre quelle que soit la forme initiale de l'expression. D'autres travaux se contentent de manipuler les expressions sous une forme bien adaptée à leurs objectifs sans pour autant imposer une forme unique.

L'approche que nous suivons ici est un mélange de ces dernières dans la mesure où notre objectif est simplement de pouvoir manipuler une expression ayant certaines propriétés qui nous « arrangent » ; pour atteindre cette forme « pseudo-normale », nous réécrivons les expressions jusqu'à ce que plus aucune des transformations sélectionnées ne soit applicable. La conversion des expressions dans cette forme « pseudo-normale » sera composée de plusieurs transformations.

Manipulation d'opérateurs n-aires

Exploiter l'associativité des opérateurs nous amène tout naturellement à préférer des opérateurs représentés sous une forme n-aire. Une opération associative-commutative (l'acronyme `AC` est utilisé par la suite) sera donc vue comme un arbre composé d'un nœud (l'opérateur) et d'un nombre arbitraire de feuilles (les opérandes). Toutes les transformations (insertion, suppression, fusion, etc.) appliquées à un opérateur AC seront effectuées de manière à conserver cette structure n-aire. L'approche usuelle en compilation consiste à ne manipuler que des expressions « mono-opérateur » (résultat de l'atomisation). Nous manipulons ici des expressions complexes, c'est-à-dire des arbres de profondeur quelconque, contenant des opérations hétérogènes (binaire ou n-aire, commutative ou non, etc.).

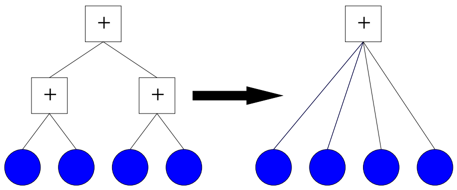


FIG. 8.2 – *Manipulation d'opérations n-aires.*

Substitution d'opérateurs

Notre objectif est ici de remplacer des opérateurs contraignants, car non associatif-commutatif, par d'autres opérateurs laissant plus de liberté aux transformations ; nous utilisons pour cela les propriétés algébriques des opérateurs. Nous reprenons la notion d'opérateur « complément » définie par Frailey dans [Fra70].

Définition 8.1
Un opérateur est défini comme étant un opérateur complément *si :*

1. *Il est unaire (un seul opérande)*
2. *Il est involutif : appliqué deux fois à un opérande quelconque, il produit le même opérande.*

L'opérateur `moins unaire` rempli par exemple ces deux propriétés puisqu'il est unaire, et vérifie $-(-X) = X$. De la même façon l'opérateur `inverse` de la division des nombres réels $(1/(1/X) = X)$ et l'opérateur booléen `NOT` (NOT(NOT(b))=b) sont des opérateurs compléments. Les opérateurs compléments sont utilisés pour substituer une opération « contraignante » par d'autres opérations plus souples. Par exemple, le `moins binaire` sera remplacé par l'addition (qui est plus flexible car associative et commutative) grâce à l'utilisation du `moins unaire` (voir figure 8.3).

Définition 8.2
*Un opérateur o_1 est **substituable par le couple d'opérateurs** (o_2,c) (avec o_2 un opérateur associatif-commutatif et c un opérateur complément) si l'opération (A o_1 B) peut être remplacée par (A o_2 (c B)).*

Le tableau 8.1 donne quelques exemples de substitutions possibles. Cette technique permet de d'obtenir une plus grande liberté dans l'application des transformations grâce à l'associativité des nouveaux opérateurs manipulés.

Remarque 8.1
Ces techniques de substitution d'opérations seront également utilisées dans le cadre des travaux d'optimisation de boucles présentés dans la partie IV. Pour extraire la sous-expression $A - C$ commune aux deux expressions $A - (B + C) + D$ et $A - C$, un passage par les expression $A + (-B) + (-C) + D$ et $A + (-C)$ est par exemple profitable.

Opérateur de départ	Opérateurs d'arrivée	Exemple	Résultat
"-" binaire	(+,"-" unaire)	A - B	A + (-B)
"/" (réels)	(*,"1/")	A/B	A*(1/B)
"implies"	(or, not)	A implies B	(not A) or B
"nand"	(or, not)	A nand B	(not A) or (not B)
"nor"	(and, not)	A nor B	(not A) and (not B)

TAB. 8.1 – *Substitution d'opérateurs.*

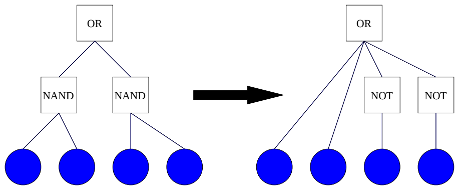


FIG. 8.3 – *Substitution d'opérateurs.*

Remarque 8.2
Les substitutions d'opérateurs sont très importantes pour augmenter les opportunités de transformations algébriques. Cependant, l'insertion d'opérateurs complément tels que (1/X) peut porter préjudice aux performances. Ainsi, l'expression $\frac{a\times b}{c\times d}$ *sera réécrite sous la forme* $a\times b\times(1/c)\times(1/d)$ *afin de construire une opération associative-commutative offrant de nouvelles opportunités de transformation. Cependant, il est nécessaire de faire le travail inverse à posteriori, lorsque la substitution n'a permis aucune transformation. En effet, l'évaluation de* $\frac{a\times b}{c\times d}$ *nécessite une seule division alors que celle de* $a\times b\times(1/c)\times(1/d)$ *en demande 2.*

Développement

Comme nous l'avons vu au paragraphe 7.2.2, une même expression peut parfois être écrite sous plusieurs formes factorisées différentes. Or, en présence d'un conflit entre plusieurs factorisations, un choix entre l'une ou l'autre est nécessaire. Bien souvent, le programmeur se contente de la formule mathématique initiale sans se soucier des performances d'évaluation de telle ou telle forme d'expression. De même, en présence de codes générés automatiquement, la sélection d'une forme factorisée plutôt qu'une autre n'est pas forcement axée sur la performance. C'est pourquoi, pour bénéficier de toutes les opportunités de transformations possibles, les expressions doivent être, dans un premier temps, développées. Une phase de (re)factorisation est ensuite appliquée (cf. section 9.2), sélectionnant éventuellement une forme factorisée différente.

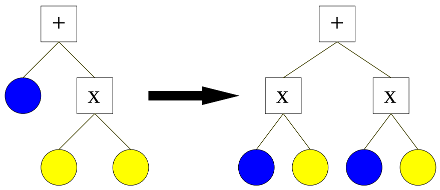


FIG. 8.4 – *Développement des opérateurs.*

8.2.2 Simplifications algébriques

De nombreuses simplifications algébriques élémentaires peuvent être appliquées dans le but de réduire le nombre d'opérations nécessaires pour évaluer une expression. Toutes les propriétés algébriques permettant de simplifier une expression peuvent être utilisées sur un large éventail d'opérations ($+$, $\times$, $-$, AND, OR, NOT, ...) et pour différentes algèbres (par exemple l'algèbre $(\max, +)$ utilisé dans [BCOQ92, OSMG98]). Le tableau 8.2 donne quelques exemples de simplifications.

Expression initiale	Expression simplifiée
X + (-X)	0
X * (1/X)	1
X * 1	X
(A implies B) or A	(not A) or B or A = true
A or (not A)	true
A and (not A)	false
max(X,min(X,Y))	X

TAB. 8.2 – *Exemples de simplifications algébriques.*

Réduction de force Certaines transformations ne réduisent pas le nombre d'opérations (voire même l'augmentent), mais remplacent une opération coûteuse en temps d'évaluation par une (ou plusieurs) autre moins coûteuse. Cette technique bien connue s'appelle réduction de force. On remplacera par exemple le calcul du carré d'un nombre (X^2) par une multiplication ($X \times X$). Le temps de calcul d'une multiplication est en effet beaucoup plus faible que celui de la puissance sur la plupart des processeurs. Cependant, si cette transformation est appliquée par la plupart des compilateurs, d'autres configurations sont en revanche beaucoup plus difficiles à détecter sans une structure de donnée adaptée. Le calcul effectué figure 8.5 en partie gauche peut être amélioré. La transformation des deux carrés sous forme de multiplications serait d'ores et déjà profitable. Cependant, en transformant cette expression sous la forme proposée en partie droite de la figure, l'évaluation peut être encore accélérée sur de nombreux processeurs ayant une latence pour l'addition plus faible que celle pour la multiplication (par exemple sur PentiumPro).

```
Do I = 1,NP
   C(I) = A(I)**2 - B(I)**2
Enddo
```

(a) Expression initiale.

```
Do I = 1,NP
   C(I) = (A(I) + B(I))*(A(I) - B(I))
Enddo
```

(b) Expression transformée.

FIG. 8.5 – *Réduction de force par transformation algébrique complexe.*

Remarque 8.3
La réduction de force devra venir après le déplacement de code invariant. En effet, en supposant déplaçable l'un des deux carrés de la figure 8.5 (ce qui n'est pas le cas ici), il serait préférable de déplacer ce dernier en dehors de la boucle plutôt que d'appliquer la réduction de force.

Évaluation partielle Que ce soit dans une application scientifique ou dans un programme généré automatiquement, un grand nombre de constantes ont une valeur connue à la compilation. La propagation de ces valeurs vers toutes les instances d'utilisation de ces constantes permet donc d'évaluer un grand nombre de calculs dès la compilation. L'évaluation partielle des expressions (avec des opérandes de valeurs connues) permet de diminuer le nombre de calculs nécessaires à l'exécution. D'autre part, elle offre également de nouvelles opportunités de simplifications algébriques. Ainsi, une évaluation partielle de l'expression $x + 4x - 2x$ (avec factorisation intermédiaire) donne le résultat $x * 1$ qui est simplifié en x.

8.3 Réécriture et Pattern-Matching associatif-commutatif

Les techniques de pattern-matching et de réécriture sont tout à fait adaptées à notre volonté de retrouver des caractéristiques particulières dans les expressions manipulées. Nous cherchons simplement dans les expressions des patterns dont nous connaissons les propriétés (pattern-matching) et que nous souhaitons transformer par la suite (réécriture). Les transformations sont de différents types (simplifier, substituer, supprimer, etc.).

8.3.1 État de l'art

Le pattern-matching associatif-commutatif est un problème connu pour sa grande complexité. Il peut être exprimé de la façon suivante :

Définition 8.3
*Le problème du pattern-matching associatif-commutatif (noté **ACM**[36]) est défini comme suit :*

Instance : *Soit V un ensemble de variables et F un ensemble de fonctions associatives et commutatives. Soit $T(F,V)$ l'ensemble de tous les termes que l'on peut construire en utilisant F et V. Soit t_1 et t_2 deux termes quelconque de $T(F,V)$. Soit ρ une fonction de V vers $T(F,V)$ représentant une substitution d'un ensemble de variables par un ensemble de termes.*

Question : *Existe-t-il une substitution ρ telle que $\rho(t_1) \underset{AC}{=} t_2$? (Existe-t-il une substitution des variables de t_1 (le pattern) par un ensemble de termes tels que $\rho(t_1)$ et t_2 sont équivalents modulo AC).*

Par exemple, pour une expression $a+b+c$ et un pattern $x+y$, il existe une substitution ρ des variables x et y par respectivement $a+c$ et b telle que $\rho(x+y) \underset{AC}{=} a+b+c$. Notre besoin en terme de pattern matching consiste à déterminer pour un ensemble de patterns $t_1, t_2, \ldots, t_n$ si l'un d'eux matche un terme donné s ; ce problème est couramment appelé *many-to-one matching*.

Benanav, Kapur et Narendran ont montré que ce problème est NP-Complet dans [BKN87] (par réduction du problème Monotone-3SAT vers ACM). Le problème de l'unification Associative-Commutative [Fag86] (dont ACM est une restriction) est également montré comme NP-Complet par Kapur et Narendran dans [KN90]. Malgré la NP-complétude du pattern-matching associatif-commutatif, il existe plusieurs algorithmes ayant une complexité exponentielle pour le pire cas, mais tout à fait utilisables en pratique. D'autre part, nous ne manipulons que des expressions de taille raisonnable, avec (beaucoup) de patterns de petite taille. Plusieurs techniques différentes sont proposées [SRR92, BCR93, MUHT96, CH96] et de nombreux outils ont été développés [WM91, WZJS94, CA95].

Les outils de pattern-matching utilisent en entrée des patterns, et donnent comme résultat une ou plusieurs substitutions possibles (si la recherche a abouti). Nous avons donc également besoin d'un mécanisme permettant de remplacer une expression qui présente certaines caractéristiques (décrites par un pattern) et la transforme en une nouvelle expression, donc d'un mécanisme de réécriture.

Les techniques de réécritures sont très utilisées dans divers domaines tels que la programmation fonctionnelle, la preuve de théorèmes ou encore la vérification de programmes. Le principe de base de la réécriture est évident ; le lecteur est invité à consulter les articles faisant une synthèse des travaux en réécriture [JL87, Plu98, Der93] pour de plus amples informations. De nombreux outils de réécriture existent également [Zha92, KZ95, Bel94, Kor98].

36. Associative-Commutative Matching.

8.3.2 Utilisation d'un simplificateur

Nous avons choisi d'utiliser l'outil de pattern-matching STORM [CA95], développé dans le cadre d'une collaboration entre les universités d'Orléans et de Stony-Brook (New-York). C'est un outil de pattern-matching associatif-commutatif (de type *many-to-one*), utilisant un mécanisme appelé *discrimination nets* [BCR93, BCR+95]. Le problème ACM peut être résolu en temps polynomial si les patterns sont *linéaires*, c'est-à-dire restreints à des termes ne contenant qu'une seule occurence de chaque variable. Cette propriété permet de mettre en place des algorithmes fonctionnant en deux temps [BCR+95]. Ils recherchent dans un premier temps les patterns qui *matchent* « modulo non-linéarité » pour une expression donnée. La seconde phase consiste ensuite à vérifier, pour le sous-ensemble de patterns sélectionné par la première phase, si ces derniers sont corrects vis-à-vis des multiples occurences de chaque variable. La NP-complétude du problème se situant au niveau de la seconde étape, l'objectif est de réduire le nombre d'appels à cette phase coûteuse grâce au filtrage.

Ce moteur de pattern-matching a été écrit de manière à faciliter son intégration dans un système de réécriture ou un prouveur de théorèmes. Ce travail d'intégration a été effectué par P. Baudin et J. Raguideau (CEA-DEIN/SLA) pour l'écriture d'un simplificateur de théorèmes dans le cadre d'un projet appelé CAVEAT [APC+98] (outil d'aide à la compréhension, vérification et génération de test pour logiciels). Le simplificateur [BR96] comporte un système de réécriture d'expressions algébriques et un parseur d'expressions [37]. La structure de données prend en compte les opérateurs associatifs-commutatifs grâce à la manipulation d'arbres n-aires. Les fonctionnalités offertes permettent de manipuler facilement les expressions (création, copie, comparaison, etc.) dans un contexte AC.

Plusieurs bases de règles Le simplificateur permet de manipuler plusieurs bases de règles de réécriture. Cette fonctionnalité nous permet de mettre en place des phases distinctes de transformations d'expressions ; nous pouvons notamment ainsi appliquer des règles « contradictoires » (qui feraient « boucler » le mécanisme de réécriture si elles étaient placées dans la même base) en les plaçant dans différentes bases, utilisées au moment opportun. Notre implémentation actuelle définit trois bases de règles :

① Normalisation des expressions
  - Normalisation
  - Simplifications algébriques
  - Évaluation partielle

② Développement à la demande

③ Mise en forme finale

Mécanisme de gardes Un mécanisme de gardes (réécriture conditionnelle) est également fourni par le simplificateur. Il permet d'appliquer des transformations plus complexes que celles de la réécriture AC classique. Ainsi, chacun des termes manipulés peut être sélectionné en fonction de son pattern, mais également en fonction de certaines propriétés. Voici deux exemples de propriétés recherchées dans les règles de réécritures.

- x est une *variable* de type `flottant`.

  La transformation d'une expression x/y en $x*(1/y)$ dont nous avons parlé précédemment n'est valide que pour des variables de type réel. Or, un pattern de la forme x/y ne permet pas de distinguer la division entière et la division flottante, qui utilisent le même symbole. La transformation peut être appliquée en ajoutant une garde à la règle de réécriture.
- x est une *constante* `entière positive et paire`.

  L'expression $(-a)^x$ peut être simplifiée sous la forme a^x si l'exposant est un nombre pair.

En pratique, ces gardes sont implémentées sous la forme de fonctions qui sont appliquées aux différents termes de la solution proposées. Elles permettent soit de vérifier la conformité de la solution trouvée aux propriétés recherchées soit de guider la réécriture. La figure 8.6 présente la syntaxe utilisée pour les règles de réécriture dans le module de simplification d'expressions de l'outil CAVEAT.

37. Les fonctionnalités présentées sont celles qui étaient disponibles au moment de l'intégration de cet outil dans notre prototype en Novembre 1997. De nouvelles possibilités ont été ajoutées depuis.

```
// règle NON GARDÉE
// insérée dans la base de règle NORM
REGLE_NORMALE (NORM, (-(-a?) → a?;) )

// règle avec PATTERN GARDÉ
// insérée dans la base de règle LATE
REGLE_GARDEE (square, LATE, (a?**b? → let x$ = a? in (x$*x$);) )

// règle avec PATTERN GARDÉ et RÉÉCRITURE GUIDÉE
// insérée dans la base de règle NORM
REGLE_GARDEE_APPLY (b_integer, let_c_eq_minus_b, NORM, -(b?*d?) → c?*d?;) )
```

FIG. 8.6 – *Syntaxe des règles de réécriture du simplificateur.*

Le moteur de pattern-matching AC (STORM) ne travaille qu'au niveau de la racine d'une expression. La réécriture est donc effectuée grâce à un parcours de l'arbre géré par le simplificateur. L'expression est transformée pas à pas, par chacune des règles applicables. Le processus de réécriture s'arrête lorsqu'aucune règle ne peut plus être appliquée.

8.3.3 Discussion

Le pattern-matching et la réécriture sont bien adaptés aux transformations algébriques que nous souhaitons appliquer dans le cadre de la normalisation et de la simplification des expressions. Cependant, ce sont des techniques dont l'utilisation peut s'avérer problématique à plusieurs niveaux que nous allons détailler ci-dessous.

Les limites du pattern-matching

Terminaison : Un inconvénient majeur des systèmes de réécriture vient de la difficulté à prouver que l'ensemble des règles proposées permet toujours au processus de réécriture de terminer. En effet, il existe des bases de règles pour lesquelles la dérivation d'une expression en une autre est infinie. Par exemple, avec une base contenant à la fois une règle de factorisation $x \times y + x \times z \to x \times (y + z)$ et une règle de développement $x \times (y + z) \to x \times y + x \times z$, le terme $a \times (b + c)$ serait indéfiniment réécrit. Il est assez facile d'éviter l'utilisation concurrente de deux règles « contradictoires », mais il est par contre difficile de prouver qu'il n'existe aucune suite de réécritures qui amène le système à boucler. L'étude de ces problèmes de terminaison d'un système de réécriture est décrite dans la littérature [Der93]. Nous n'avons pas cherché à montrer de manière formelle que l'ensemble des règles que nous utilisons abouti à un système de réécriture valide. Cependant, l'utilisation combinée de plusieurs bases de réécritures et de règles simples visant pour la plupart à diminuer la taille de l'expression fait que nous n'avons, en pratique, rencontré aucun problème.

Expressivité : L'absence de généricité dans la définition des patterns utilisés par STORM nous a posé des problèmes à plusieurs niveaux. Voici deux exemples de fonctionnalités qui nous ont manqué :

1. Terme **facultatif**
   La plupart des patterns existent sous une forme « réduite » et sous une forme « étendue ». La première décrit simplement le pattern, tandis que la seconde prend en compte les éventuels termes ajoutés qui n'interviennent pas directement dans le pattern. Par exemple, le pattern $x \times y_1 + x \times y_2$ matche l'expression $a \times b + a \times c \times d$ mais ne matche pas l'expression $a \times b + a \times c \times d + e$ qui contient un terme « parasite ». Il faut donc utiliser également le second pattern $x \times y_1 + x \times y_2 + z$.
2. Terme **répétitif**
   De la même façon, il serait utile de pouvoir exprimer directement dans le pattern le fait qu'un terme peut apparaître un nombre *indéterminé* de fois. Cette fonctionnalité serait par exemple

nécessaire pour l'écriture du pattern de la factorisation $x \times y_1 + x \times y_2 \ldots + x \times y_n$. Le nombre n étant non borné, il est impossible d'écrire toutes les règles contenant $1, 2, \ldots, n-1$ puis n termes.

Problèmes d'explosion combinatoire

Malgré la NP-Complétude du pattern-matching associatif-commutatif, l'application de règles de réécriture pour effectuer une normalisation et une simplification des expressions mathématiques rencontrées dans un programme scientifique est tout à fait possible en pratique. En effet, pour ce contexte particulier dans lequel nous appliquons *à la volée* des règles de réécriture, les temps d'exécution restent très faibles.

En revanche, nous souhaitons également appliquer des transformations pour lesquelles nous voulons pouvoir choisir entre **toutes** les opportunités de transformations qui existent à chaque étape du processus. La complexité devient alors beaucoup plus problématique. En effet, le choix doit avoir lieu entre toutes les substitutions possibles pour chaque pattern différent qui matche une expression donnée. L'explosion combinatoire est alors trop difficile à gérer.

Factorisation Afin d'améliorer les performances d'évaluation des expressions sur les processeurs modernes, nous souhaitons pouvoir choisir entre les différentes formes factorisées d'une expression donnée. Or, pour ce problème, les techniques de pattern-matching AC ne sont pas adaptées. En effet, il y a une explosion combinatoire du nombre de matches obtenus. Considérons une expression de la forme suivante :

$$i_1 \times i_2 \ldots \times i_p \times j_1 + i_1 \times i_2 \ldots \times i_p \times j_2 + \cdots + i_1 \times i_2 \ldots \times i_p \times j_n \tag{8.1}$$

La recherche à partir d'un pattern simple (non générique) de la forme $x \times y_1 + x \times y_2 + z$ de toutes les formes factorisées de l'expression 8.1 se fera par un processus itératif. Or, **pour chaque étape**, il y a $(2^p - 1)\frac{n(n-1)}{2}$ matches.

Remarque 8.4
Le dénombrement est établi de la manière suivante :

- *Pour chaque sous-expression $i_1 \times i_2 \ldots \times i_p \times j_1$, le pattern $x \times y_1$ propose $(2^p - 1)$ substitutions ($x = i_1, x = i_2, \ldots x = i_p, x = i_1 \times i_2, x = i_1 \times i_3, \ldots$). La solution $x = \emptyset, y = i_1 \times i_2 \ldots \times i_p \times j_1$ n'est pas comptabilisée (d'où le -1).*
- *Le pattern $x \times y_1 + x \times y_2 + z$ matche $\frac{n(n-1)}{2}$ fois avec les expressions de la forme $x \times j_1 + x \times j_2 + \ldots + x \times j_n + z$ (nombre de paires (y_1, y_2) que l'on peut constituer à partir de n éléments grâce à l'associativité et la commutativité).*

De même, la factorisation de l'expression 8.1 avec un pattern générique de la forme $x \times y_1 + x \times y_2 + \ldots + x \times y_K + z$ (avec K générique) nécessite $(2^p - 1)(2^n - n - 1)$ matches.

Remarque 8.5
Le dénombrement est établi de la manière suivante :

- *Le terme $(2^p - 1)$ est le même que pour le pattern simple (cf. ci-dessus).*
- *Le nombre de matches entre le pattern $x \times y_1 + x \times y_2 + \ldots + x \times y_K + z$ et l'expression de la forme $x \times j_1 + x \times j_2 + \ldots + x \times j_n + z$ comprend le nombre de groupes de 2 parmi n (C_n^2), plus le nombre de groupes de 3 parmi n (C_n^3), etc. jusqu'au nombre de groupes de n parmi n ($C_n^n = 1$). Avec la formule permettant l'évaluation de $(a+b)^n$ nous obtenons :*
  $(1+1)^n = 1 + C_n^1 + \ldots + C_n^n$
  Ce qui nous amène directement au terme $2^n - 1 - C_n^1$ du dénombrement indiqué.

Extraction des *fma* L'extraction des *fma* en utilisant le pattern-matching donne également lieu à une explosion combinatoire du nombre de matches. L'expression recherchée est de la forme :

$$i_1 + i_2 + \ldots + i_n + j_1 \times j_2 \times \ldots \times j_p$$

Les *fma* combinent une addition et une multiplication ; le pattern recherché sera de la forme $x \times y + z + k$ pour construire l'expression $fma(x, y, z) + k$. $(2^p - 2) \times (2^n - 1)$ seront donc nécessaires pour l'extraction de ce *fma*. Ce nombre de matches est ensuite à multiplier par le nombre de multiplications qui existent simultanément pour une même addition.

Remarque 8.6
Le pattern $x \times y$ permet d'obtenir 2^p substitutions avec la sous-expression $j_1 \times j_2 \times \ldots \times j_p$. Les solutions avec $x = \emptyset$ et $x = j_1 \times j_2 \times \ldots \times j_p$ sont supprimées.

La substitution des termes z et k à partir des n opérandes i_x nécessite $2^n - 1$ matches ; le cas $z = \emptyset$ est interdit.

L'application de la factorisation et de l'extraction des *fma*, lorsque l'on souhaite contrôler le processus en sélectionnant une transformation précise à chaque étape, entraîne une explosion combinatoire du nombre de matches. Les techniques de pattern-matching, bien adaptées pour la normalisation des expressions, ne seront donc pas utilisées pour factoriser et extraire les *fma*.

8.4 Conclusion

Les transformations algébriques, introduites au chapitre 7 à travers deux exemples, supposent la manipulation d'expressions de grande taille, présentant un grand degré de liberté. Nous avons implémenté une phase de *forward substitution* permettant de reconstruire des expressions optimisables à partir de codes fortement atomisés. Cette technique est nécessaire pour offrir de véritables opportunités de transformations algébriques augmentant le parallélisme d'instructions des expressions, notamment lorsque les applications ont été optimisées pour des architectures de machines aujourd'hui obsolètes. Nous proposons également une technique fondée sur le pattern-matching et la réécriture associatifs-commutatifs et visant à mettre les expressions sous une forme adaptée aux transformations algébriques envisagées par la suite. Les expressions sont manipulées sous la forme d'arbres n-aires dans un contexte associatif-commutatif. Ce dernier offre de nouvelles opportunités pour l'application des transformations classiques telles que les simplifications algébriques, la réduction de force ou encore l'évaluation partielle. Ce travail a été implémenté dans le cadre du prototype EOLE et utilise les fonctionnalités des outils STORM et CAVEAT.

L'explosion combinatoire, décrite au paragraphe 8.3.3.0, interdit l'utilisation du pattern-matching dès lors qu'une procédure de sélection entre les différentes solutions est mise en place. La réécriture ne sera donc appliquée que pour l'étape décrite dans ce chapitre et visant à augmenter les opportunités de transformations. Le chapitre 9 présente deux heuristiques permettant une sélection entre les différentes transformations possibles, tout en échappant à l'explosion combinatoire.

Chapitre 9

Transformations algébriques des expressions

Ce chapitre présente deux heuristiques et un algorithme qui transforment automatiquement les expressions dans le but de les factoriser, d'en extraire les *fma* et de les équilibrer. Les transformations algébriques présentées au chapitre précédent sont appliquées « pas à pas » ; nous ne cherchons pas à choisir entre plusieurs transformations incompatibles. Les techniques de réécriture associative-commutative sont donc bien adaptées à la situation. Cependant, pour les transformations présentées dans ce chapitre, nous souhaitons prendre des décisions afin de sélectionner des expressions dont la structure est bien adaptée à une évaluation performante sur une certaine classe de processeurs. Nous avons vu au chapitre 8 que, dans cette situation, le nombre de matches devient beaucoup trop grand, ce qui rend problématique l'utilisation du pattern-matching dans notre contexte, la compilation en un temps raisonnable de programmes scientifiques.

Dans la mesure où les critères de sélection des expressions peuvent varier selon le type d'architecture, nous souhaitons proposer une technique s'appuyant sur une « fonction d'évaluation » paramétrique. Le seul algorithme générique permettant d'obtenir la meilleure expression (en terme de temps d'évaluation minimal) consisterait donc à parcourir tout l'espace des solutions en calculant pour chacune le coût associé. Or, le nombre d'expressions équivalentes au sens algébrique, compte tenu des choix possibles concernant la factorisation et l'extraction des *fma* dans un cadre associatif-commutatif, est trop important. L'explosion combinatoire de ce nombre d'expressions ne nous permet pas de parcourir tout l'espace des expressions équivalentes. L'approche proposée ici pour la factorisation et l'extraction des *fma* est donc heuristique et repose sur un processus *itératif* de type *glouton* :

- Les décisions sont toujours prises localement. Nous ne cherchons pas à atteindre la meilleure solution « finale » mais à progresser itérativement vers une bonne solution.
- L'ensemble des transformations élémentaires autorisées à chaque étape du processus itératif est restreint. En effet, la seule utilisation de l'associativité et de la commutativité des opérateurs permettrait de construire un nombre de transformations élémentaires considérable. Seules quelques transformations dont nous choisissons les caractéristiques sont conservées. Le choix de ces transformations peut varier pour différentes classes d'architectures.
- Les décisions à chaque étape sont prises à partir de critères simples et efficaces que nous avons déterminés en tenant compte des architectures ciblés et qui ont été validés expérimentalement.

Comme le montre la figure 9.1, nous appliquons itérativement des transformations élémentaires pour atteindre l'expression finale après un certain nombre d'étapes. La transformation complète de l'expression peut ne pas être optimale, mais nous verrons que cette technique donne en pratique de bons résultats.

Avant d'entrer dans l'exposé des techniques proposées, ce chapitre commence par la présentation de différents critères de sélection des expressions. Les heuristiques de factorisation et d'extraction des *fma* seront respectivement décrites en détail aux sections 9.2 et 9.3. Un algorithme visant à favoriser l'équilibrage d'une expression sera également introduit à la section 9.4. Notons que les interactions entre

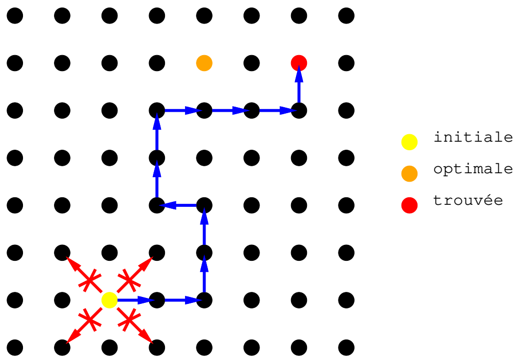


FIG. 9.1 – *Processus itératif de type glouton.*

ces différentes transformations algébriques et plus généralement avec les optimisations de code seront discutées au chapitre 17.

9.1 Critères de sélection

Ce paragraphe introduit différentes valeurs qui seront utilisées comme critères de sélection pour les transformations présentées dans la suite de ce chapitre. L'objectif de nos transformations est de diminuer le temps d'évaluation des expressions que l'on peut trouver dans un programme scientifique. Or, étant données deux expressions, il est assez difficile de prédire celle dont l'évaluation sera la moins coûteuse en temps d'exécution. Les paramètres qui interviennent sont en effet très nombreux ; on y trouvera notamment, le nombre et la latence des opérations, le nombre de registres, le cache, etc. Une expression a bien évidemment un *coût réel* correspondant à son temps d'évaluation sur un processeur donné (pour une compilation et dans un état donnés). Cependant, le contexte dans lequel se situent ces travaux, à savoir la compilation, rend impossible un choix entre les expressions fondé sur le temps d'exécution réel.

Nous pouvons cependant utiliser des valeurs théoriques permettant de décrire plus ou moins précisément les expressions et donnant un *coût estimé*. Une estimation très précise des temps d'évaluation des expressions manipulées suppose bien évidemment une modélisation de tous les paramètres pouvant entrer en jeu [AKR98, GmWHR98] (architecture de la machine, ordonnancement des opérations, etc.). Cependant, le critère de choix ainsi déterminé devient très fortement lié à un contexte particulier (compilateur x, processeur y, option de compilation z, etc.). Ne perdons pas de vue ici que les transformations algébriques que nous souhaitons appliquer restent de haut niveau. Elles visent à rechercher une bonne solution tenant compte de l'évolution des architectures actuelles, sans pour autant prétendre déterminer l'évaluation idéale pour un processeur donné.

Considérons une expression représentée par un arbre. Nous appelons w_i le *poids* associé à un nœud i dans l'arbre représentant l'expression. Cette valeur modélise le coût nécessaire à l'évaluation d'une opération donnée ; elle s'approche donc ainsi de la latence d'une opération. En pratique, les poids sont attribués de manière à donner une indication de la complexité de l'opération. Par exemple, le poids de l'addition est fixé à 1 tandis que celui de la division est fixé à 10 (des exemples de latences sont donnés dans le tableau 1.1 au chapitre 1). Il est bien évident que, selon l'architecture utilisée, le poids d'un opérateur pourrait varier (notamment selon que l'opération est effectuée au niveau logiciel ou matériel). Nous notons d_i la distance d'un nœud i à la racine de l'arbre.

Remarque 9.1
Pour déterminer le poids et/ou la profondeur d'un opérateur associatif avec une représentation n-aire, il est important de prendre en considération également le nombre d'opérandes de celui-ci. Ainsi, le poids affecté à un nœud tiendra compte de son poids « unitaire » et du nombre d'opérandes ; le poids d'une

opération $\oplus(x_1, \ldots, x_n)$ *sera égal à* $(n-1) \times w_{\oplus}$*. De la même façon, sa hauteur peut être approchée ; la hauteur moyenne d'un arbre binaire est proportionnelle à* $n\sqrt{n}$ *et sa hauteur minimale est de l'ordre de* $n \times ln(n)$ *[Knu68].*

Définition 9.1
Le ***poids total*** *d'une expression contenant n nœuds (opérations), noté* W_E*, est défini par :*

$$W_E = \sum_{i=1}^{n} w_i$$

Le **poids total** d'une expression permet de caractériser l'effort de calcul à appliquer pour évaluer une expression donnée. Cette valeur est particulièrement importante pour une architecture ne disposant que d'une seule unité de calcul.

Définition 9.2
Le ***chemin critique*** *d'une expression contenant n nœuds, noté* C_E*, est défini par :*

$$C_E = \max_{i=1}^{n} d_i$$

Le **chemin critique** d'une expression traduit la longueur maximale des dépendances existant entre les différentes opérations qui doivent s'enchaîner de manière séquentielle. Si l'on suppose que l'architecture contient une « infinité » d'unités de calculs ou au moins autant que l'on en veut (architecture s'approchant du modèle VLIW), cette mesure est particulièrement importante.

La tendance actuelle veut que les processeurs modernes disposent de plusieurs unités de calculs, chacune d'elles étant pipelinée, mais sans toutefois avoir une capacité de calcul illimitée. Le poids total et le chemin critique d'une expression doivent donc être tous deux pris en considération. Pour cela, nous utilisons une mesure nouvelle appelée **gravité** :

Définition 9.3
La ***gravité*** *d'une expression contenant n nœuds, notée* G_E*, est définie par :*

$$G_E = \frac{\sum_{i=1}^{n} w_i \times d_i}{W_E}$$

Cette mesure décrit la profondeur moyenne des opérations dans l'arbre. Ce « coût estimé » est une bonne estimation tenant compte à la fois du poids des calculs et de leur position dans l'arbre. Les décisions prises à partir du chemin critique ou de la gravité sont très proches. Cependant, la gravité permet de différencier deux expressions de même chemin critique mais avec un nombre différent d'occurences de ce chemin critique. Tous les chemins de l'arbre représenté sur la partie gauche de la figure 9.2 sont « critiques » (de longueur maximale égale à 5) ; l'évaluation de cette expression avec deux unités de calculs demande 6 cycles. L'évaluation de l'expression représentée à droite, de même chemin critique, ne requiert quant à elle que 5 cycles. L'utilisation de la gravité permet de différencier ces deux expressions. La gravité est donc une mesure importante pour des processeurs superscalaires disposant de parallélisme d'instruction. Son utilisation a prouvé en pratique que c'est un critère de décision efficace.

Les *coûts estimés* que nous utilisons modélisent donc les caractéristiques que nous souhaitons mettre en avant dans une expression, sans pour autant être très précis. Les expériences menées n'ont en outre pas démontré l'utilité d'une nouvelle mesure plus précise. Toutefois, les techniques présentées dans les sections suivantes peuvent être utilisées sans modification des algorithmes avec toute autre mesure estimée.

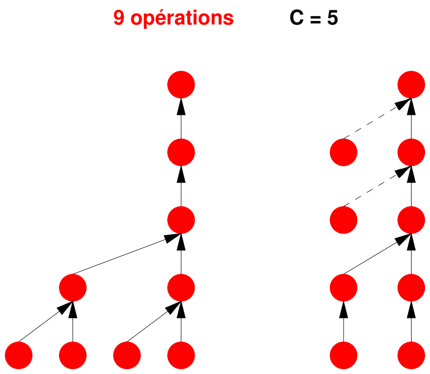


FIG. 9.2 – *Critères de décisions.*

9.2 Heuristique de Factorisation

La factorisation d'une expression est une transformation permettant de diminuer le nombre d'opérations à effectuer pour son évaluation. La propriété de distributivité s'applique sur divers couples d'opérateurs tels que $(+,\times)$, $(min,+)$, (`OR`,`AND`), etc. En plus de cette réduction du nombre d'opérations, nous souhaitons pouvoir sélectionner, parmi toutes les formes factorisées d'une expression, celle qui est la mieux adaptée pour une exécution sur des processeurs modernes.

9.2.1 Recherche des factorisations

Comme nous l'avons vu au chapitre précédent, les techniques de pattern-matching ne sont pas adaptées à nos besoins en raison de l'explosion combinatoire du nombre de matches ; beaucoup de ces matches sont inutiles ou redondants. Nous proposons ici une heuristique de recherche des formes factorisées d'une expression. Elle n'est pas optimale et risque de « manquer » certaines formes factorisées complexes, par exemple $(x + y) \times (z + k)$, mais reste efficace en pratique.

La technique utilisée consiste à rechercher, pour un nœud donné de l'arbre, tous les candidats potentiels pour la factorisation. La recherche prend en compte la présence d'opérateurs compléments (cf. chapitre 8). L'algorithme de la figure 9.3 retourne la liste de toutes les factorisations trouvées pour un nœud donné. Nous notons sous la forme $(x, \text{list of } y_i, z)$ la factorisation d'une expression E telle que $E = x \times \sum_i y_i + z$. La notation $n_1[n_2 \rightarrow n_3]$ décrit la substitution du nœud n_2 par le nœud n_3 à l'intérieur de n_1.

Pour un nœud avec n termes contenant chacun m « sous-termes », la complexité de cet algorithme est $\mathcal{O}(n^2m^2)$. Cette valeur est donnée en supposant que la comparaison entre deux sous-arbres est effectuée en temps constant $\mathcal{O}(1)$. Même si ce n'est pas vraiment le cas, cette hypothèse devient raisonnable en pratique grâce à l'utilisation combinée d'une table de hachage et d'une structure de donnée adaptée.

Remarque 9.2
Notre implémentation de cette heuristique a été optimisée à plusieurs niveaux. Il est par exemple plus efficace de garder trace de la liste des candidats à la factorisation durant tout le processus de recherche. Par ailleurs, nous construisons une liste des termes X qui ont déjà été testés afin de supprimer toutes les factorisations partielles qui ne nous intéressent pas. Par exemple, pour l'expression $a\times b+a\times c+a\times d$, seule la factorisation $a \times (b + c + d)$ sera conservée ; les factorisations partielles telles que $a \times (b + c) + a \times d$ seront rejetées. Rappelons enfin que l'implémentation est paramétrique et fonctionne pour tout couple d'opérateurs vérifiant la propriété de distributivité (ainsi que l'associativité et la commutativité).

9.2.2 Gestion des conflits

Les expériences décrites au paragraphe 7.2.2 ont montré que plusieurs formes factorisées peuvent exister pour une même expression. Il est dans ce cas nécessaire d'effectuer un choix ; deux factorisations peuvent être en effet incompatibles. Par exemple, une seule factorisation à la fois peut être appliquée à

```
Function list_of_candidates(n)
    - Input: n is a node
    - Output: a list of factorizable candidates
if (operator(n) is "-") then n = child of(n) end if
if (operator(n) is "*") then return children of(n)
else return singleton(n)
end if

Function Factorization(n)
    - Input: n is a node
    - Output: a list of possible factorizations for this node
if (operator(n) is "+") then
    let list of factors = (nil)
    let list of terms = children of(n)
    let Z1 = "0"
    for each term in list of terms
        let list of successors = successors of term in list of terms
        for each X in list_of_candidates(term)
            let Z2 = "0"
            let list of Y = (nil)
            for each successor in list of successors
                if (X appears in list_of_candidates(successor)) then
                    add successor[X_found→"1"] to list of Y
                else
                    Z2 = Z2 "+" successor
                end if
            end for each successor
            if (list of Y is not empty) then
                add term[X_tested→"1"] to list of Y
                add (X,list of Y,Z1 "+" Z2) to list of factors
            end if
        end for each X
        Z1 = Z1 "+" term
    end for each term
    return list of factors
end if
```

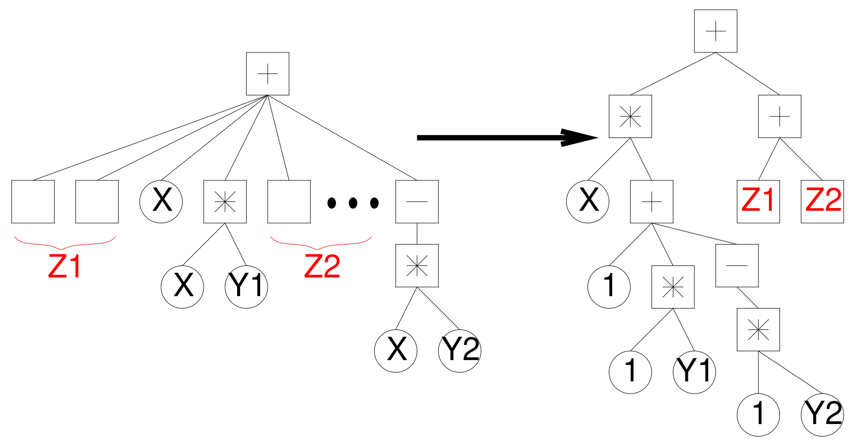


FIG. 9.3 – *Heuristique de factorisation.*

l'expression $a \times b + a \times x_1 + b \times x_2$. Le résultat obtenu est alors soit $a \times (b + x_1) + b \times x_2$, soit $a \times x_1 + b \times (a + x_2)$. Même si le nombre d'opérations nécessaires pour évaluer cette expression sous ses deux formes factorisées est identique, les temps d'évaluation peuvent être significativement différents. Les coûts respectifs des termes mis en jeu (a, b, x_1 et x_2) interviennent en effet différemment dans le coût total de l'expression selon la factorisation choisie.

Le choix entre les différentes formes factorisées d'une expression sera effectué en utilisant un des critères de sélection évoqués section 9.1. Ainsi, si l'on suppose que G est le critère choisi, la factorisation retenue sera celle qui minimise la gravité de manière à favoriser l'exploitation du parallélisme d'instructions.

La démarche proposée ici est fondée sur une progression itérative par transformations élémentaires vers une expression ayant de « bonnes » propriétés (poids, chemin critique, gravité, etc.). La prise successive des meilleures décisions *locales* vis à vis d'un critère donné ne constitue cependant pas obligatoirement la meilleure décision *globale*. Considérons une expression contenant trois candidats x_a, x_b et x_c pour la factorisation. L'expression est de la forme suivante :

$$\begin{aligned}
&x_a.x_b.x_c.Fabc_1 + \ldots x_a.x_b.x_c.Fabc_{n_{abc}}\\
&+x_a.x_b.Fab_1 + \ldots x_a.x_b.Fab_{n_{ab}}\\
&+x_a.x_c.Fac_1 + \ldots x_a.x_c.Fac_{n_{ac}}\\
&+x_b.x_c.Fbc_1 + \ldots x_b.x_c.Fbc_{n_{bc}}\\
&+x_a.Fa_1 + \ldots + x_a.Fa_{n_a}\\
&+x_b.Fb_1 + \ldots + x_b.Fb_{n_b}\\
&+x_c.Fc_1 + \ldots + x_c.Fc_{n_c}\\
&+F_1 + \ldots + F_n
\end{aligned} \tag{9.1}$$

Il existe 6 formes factorisées différentes pour cette expression. La figure 9.4 présente la structure de l'arbre pour une factorisation sur x_a puis x_b et x_c. L'expression factorisée est donnée par l'équation 9.2. Le nombre d'opérations est identique pour chacune des 6 factorisations possibles, mais la position des termes x_a, x_b, x_c, $Fabc_i$, Fab_i, Fac_i, etc. varie selon la factorisation choisie.

$$\begin{aligned}
&x_a.[x_b.(x_c.\Sigma_{i=1}^{n_{abc}} Fabc_i + \Sigma_{i=1}^{n_{ab}} Fab_i) + x_c.\Sigma_{i=1}^{n_{ac}} Fac_i + \Sigma_{i=1}^{n_a} Fa_i]\\
&+x_b.(x_c.\Sigma_{i=1}^{n_{bc}} Fbc_i + \Sigma_{i=1}^{n_b} Fb_i)\\
&+x_c.\Sigma_{i=1}^{n_c} Fc_i\\
&+\Sigma_{i=1}^{n} F_i
\end{aligned} \tag{9.2}$$

La notion de transformation élémentaire dans le cadre de la factorisation se limite à une factorisation partielle (factorisation d'un seul terme à chaque étape). En cas de conflits de factorisation, les décisions sont prises localement entre toutes les factorisations élémentaires disponibles. Sur notre exemple, la factorisation complète à partir des trois termes x_a, x_b et x_c se fera en trois étapes. Les choix sont donc réalisés entre les factorisations partielles sur un terme, x_a par exemple, avec le reste de l'expression (c'est-à-dire les termes contenant x_b et x_c) laissé sous forme développée. La factorisation partielle sur x_a est présentée sur la figure 9.5. Le tableau 9.1 indique la profondeur de chacun des termes de l'expression après la factorisation partielle sur x_a ou x_b (la position des opérateurs + et $\times$ n'est pas donnée, car elle ne varie pas en fonction de la factorisation choisie).

Remarque 9.3
La présence de plusieurs occurrences d'un même terme (les termes x_b et x_c apparaissent respectivement 2 et 4 fois dans l'équation 9.2) doit être prise en compte dans l'évaluation du critère. L'extraction des sous-expressions communes permet en effet de n'effectuer l'évaluation de ces termes qu'une seule fois. Seul le terme de profondeur maximale est ici comptabilisé pour l'évaluation du critère.

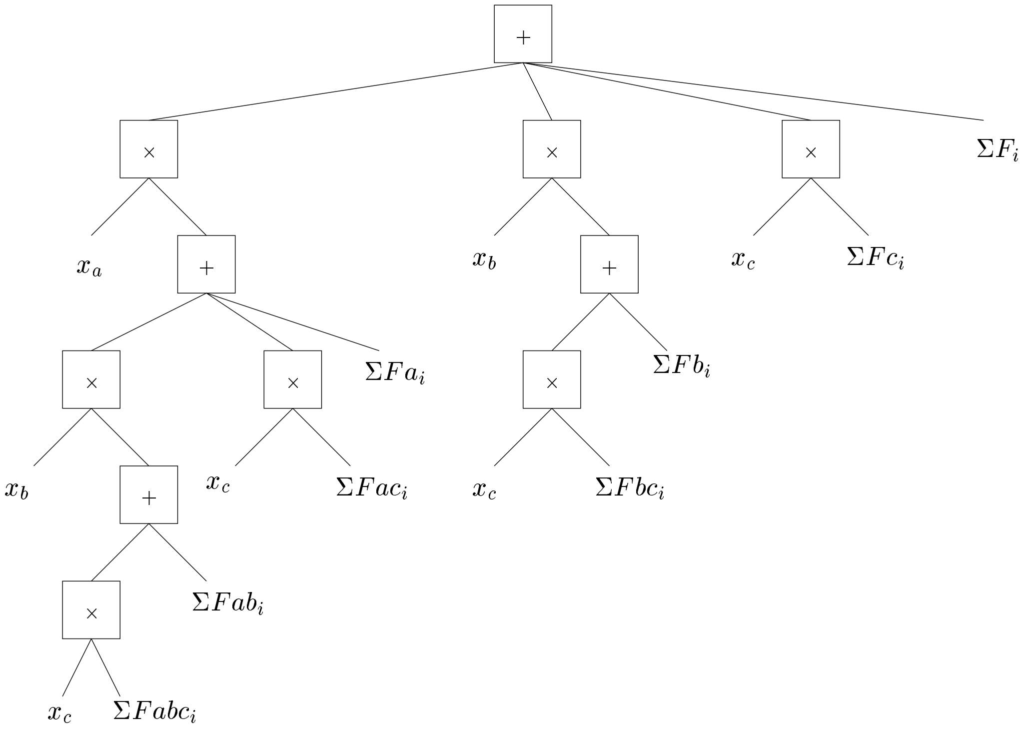


FIG. 9.4 – *Factorisation par x_a puis x_b et x_c.*

Profondeur	Factorisation sur x_a	Factorisation sur x_b
1	$F_1 \dots F_n$	$F_1 \dots F_n$
2	x_a, $Fbc_1 \dots Fbc_{n_{bc}}$, $Fb_1 \dots Fb_{n_b}$, $Fc_1 \dots Fc_{n_c}$	x_b, $Fac_1 \dots Fac_{n_{ac}}$, $Fa_1 \dots Fa_{n_a}$, $Fc_1 \dots Fc_{n_c}$
3	$Fa_1 \dots Fa_{n_a}$	$Fb_1 \dots Fb_{n_b}$
4	x_b, x_c, $Fabc_1 \dots Fabc_{n_{abc}}$, $Fab_1 \dots Fab_{n_{ab}}$, $Fac_1 \dots Fac_{n_{ac}}$	x_a, x_c, $Fabc_1 \dots Fabc_{n_{abc}}$, $Fab_1 \dots Fab_{n_{ab}}$, $Fbc_1 \dots Fbc_{n_{bc}}$

TAB. 9.1 – *Profondeur des termes dans les factorisations partielles.*

En supposant que le critère *gravité* est utilisé pour choisir entre les factorisations partielles, nous obtenons le résultat suivant pour la comparaison entre la factorisation sur x_a et celle sur x_b (à partir des éléments du tableau 9.1) :

$$\begin{gathered} G_a \leq G_b \\ \Sigma_{i=1}^{n_a} W(Fa_i) + 2W(x_b) + 2.\Sigma_{i=1}^{n_{ac}} W(Fac_i) \leq \Sigma_{i=1}^{n_b} W(Fb_i) + 2W(x_a) + 2.\Sigma_{i=1}^{n_{bc}} W(Fbc_i) \end{gathered} \tag{9.3}$$

Les poids respectifs des termes x_a et x_b interviennent évidemment dans le résultat. Cependant, nous allons voir que la présence des autres termes est en fait fortement liée à la forme développée (non définitive) de l'expression. En effet, si la décision est prise en considérant l'expression sous sa forme factorisée finale, présentée figure 9.4, ces termes n'interviennent plus dans le processus de choix. Le tableau 9.2 indique la profondeur de chacun des termes après les factorisations complètes sur, d'une part, x_a puis x_b et x_c et, d'autre part, x_b puis x_c et x_a.

L'utilisation de la *gravité* comme critère de sélection entre les deux factorisations *abc* et *bca* permet d'aboutir au résultat donné par l'équation 9.4. Seuls les poids respectifs des termes x_a, x_b et x_c inter-

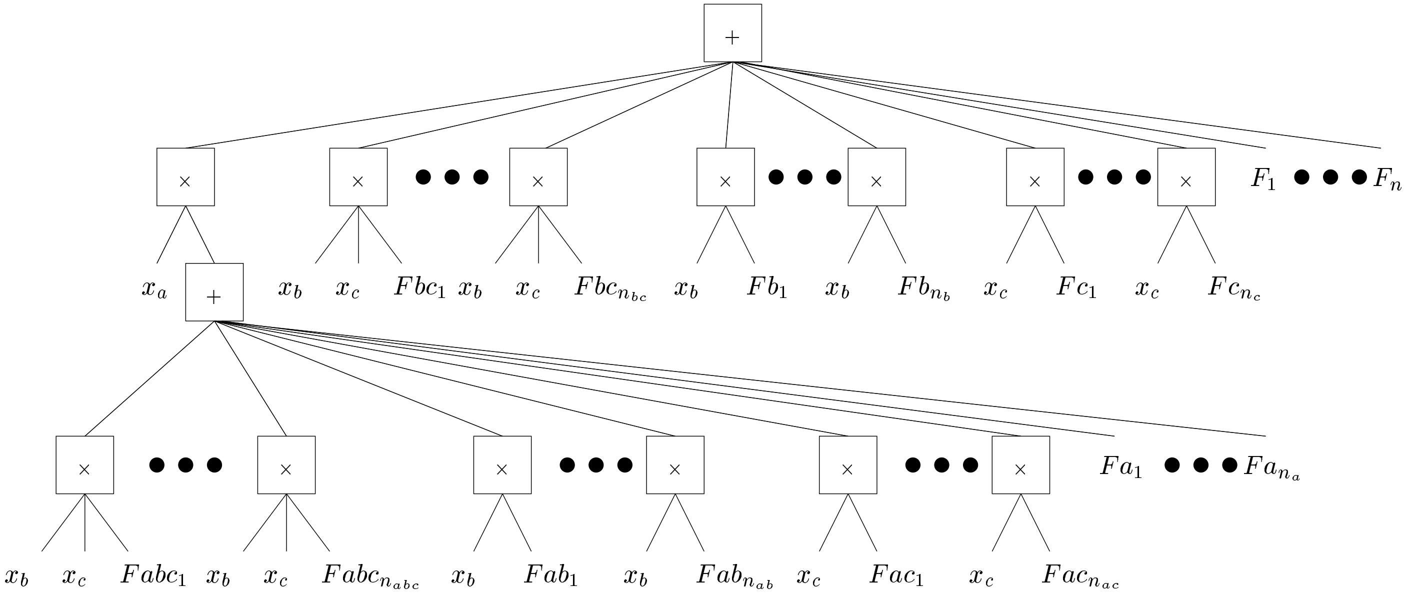


FIG. 9.5 – *Factorisation partielle sur x_a.*

Profondeur	Factorisation sur x_a, x_b puis x_c	Factorisation sur x_b, x_c puis x_a
1	$F_1 \dots F_n$	$F_1 \dots F_n$
2	x_a	x_b
3	$Fa_1 \dots Fa_{n_a}$, $Fb_1 \dots Fb_{n_b}$, $Fc_1 \dots Fc_{n_c}$	$Fa_1 \dots Fa_{n_a}$, $Fb_1 \dots Fb_{n_b}$, $Fc_1 \dots Fc_{n_c}$
4	x_b	x_c
5	$Fab_1 \dots Fab_{n_{ab}}$, $Fac_1 \dots Fac_{n_{ac}}$, $Fbc_1 \dots Fbc_{n_{bc}}$	$Fab_1 \dots Fab_{n_{ab}}$, $Fac_1 \dots Fac_{n_{ac}}$, $Fbc_1 \dots Fbc_{n_{bc}}$
6	x_c	x_a
7	$Fabc_1 \dots Fabc_{n_{abc}}$	$Fabc_1 \dots Fabc_{n_{abc}}$

TAB. 9.2 – *Profondeur des termes dans les factorisations complètes.*

viennent dans le choix. L'évaluation du critère de décision dans le cas de factorisation partielle prend donc en considération des termes qui n'interviennent plus si l'on considère directement la forme factorisée finale. La prise de décisions locales n'est pas optimale vis à vis du critère choisi.

$$\begin{gathered} G_{abc} \leq G_{bca} \\ 2W(x_b) + 2W(x_c) \leq 4W(x_a) \end{gathered} \tag{9.4}$$

Toutefois, si l'on se contente de rechercher la factorisation complète minimisant la *gravité*, il est possible d'obtenir ce résultat directement sans utiliser de factorisation partielle. En effet, comme le montre le tableau 9.3, une simple analyse comparative de la profondeur et du poid des termes mis en jeu par chacune des combinaison possibles pour la factorisation complète, permet de faire le meilleur choix (vis à vis de G). Par exemple, pour choisir entre la factorisation sur a, puis b et c et celle sur c puis b et a, il suffit de comparer les résultats de $2 \times W(x_a) + 4 \times W(x_b) + 6 \times W(x_c)$ et $6 \times W(x_a) + 4 \times W(x_b) + 2 \times W(x_c)$. La technique présentée ici pour 3 candidats est facilement étendue à une expression contenant n candidats en conflits pour la factorisation.

Factorisation	Profondeur $\times W(x_a)$	Profondeur $\times W(x_b)$	Profondeur $\times W(x_c)$
abc	2	4	6
acb	2	6	4
bac	4	2	6
bca	6	2	4
cab	4	6	2
cba	6	4	2

TAB. 9.3 – *Termes à prendre en compte dans la comparaison des gravités pour les 6 factorisations.*

Remarque 9.4
Les résultats présentés ci-dessus sont donnés pour une expression mise sous une forme « généralisée ». Il est évident que l'absence de l'un des termes $\Sigma_{i=1}^{n} Fx_i$ [38] ou la réduction d'un terme à un seul élément (par exemple disparition d'une addition au profit d'un seul opérande telle que $\Sigma_{i=1}^{n} Fx_i = C$) peuvent modifier les positions, et ainsi intervenir sur le choix final.

Nous avons vu que la transformation élémentaire consistant à appliquer les factorisations partielles les unes après les autres n'est pas idéale en présence de conflits de factorisation. Les décisions locales peuvent en effet ne pas aboutir à la meilleure factorisation complète vis à vis du critère choisi. Ce

38. Fx_i représente ici l'un quelconque des termes $Fabc_i$, Fab_i, Fac_i, Fa_i, etc.

problème pourrait être résolu en n'appliquant que des factorisations complètes choisis par comparaison des poids et des positions respectifs des candidats pour la factorisation. Cependant, cette technique n'est applicable que pour un critère de sélection déterminé (notre étude concernait ici la gravité G). Le caractère paramétrique de l'heuristique proposée est alors perdu. Nous avons donc choisi de conserver pour les expérimentations le mécanisme de factorisations partielles. Notons qu'en pratique, les conflits de factorisations faisant intervenir plus de 2 termes factorisables sont assez rares.

9.3 Heuristique d'extraction des multiply-adds

Nous allons maintenant décrire notre approche concernant l'extraction « efficace » des opérations du type multiply-add. En plus d'une augmentation du parallélisme d'instruction, les processeurs modernes offrent de nouvelles instructions permettant l'exécution simultanée de plusieurs opérations. L'instruction que nous manipulons ici, le multiply-add (*fma*), effectue une addition et une multiplication sous la forme $x \times y + z$. Cette instruction est de plus en plus souvent proposée par les constructeurs dans la mesure où elle coïncide avec un type de calcul fréquemment rencontré dans les programmes scientifiques. D'autres instructions permettant ce type de calculs « fusionnés » existent également dans certains jeux d'instructions (par exemple $a \times b + c \times d$ dans le jeux d'instructions MIPS-IV). Selon l'implémentation matérielle réalisée, l'instruction aura une latence plus ou moins intéressante comparée à la latence des opérations non fusionnées. Notons que certains constructeurs ne proposent pas de gain au niveau de la latence, mais se contentent d'ajouter une instruction spécifique afin de réduire le nombre d'instructions à charger.

La présence de ces instructions permet aux constructeurs de « vendre » une performance crête nettement augmentée (doublée dans le cas des *fma* sur l'architecture Power2). Cependant, si le compilateur n'est pas capable d'extraire efficacement ces instructions, les performances obtenues en pratique ne sont pas si bonnes. Nos expériences ont montré que les compilateurs commerciaux ne détectent les *fma* que pour des cas simples (cf. chapitre 7).

9.3.1 Approche proposée

Les transformations algébriques fondées sur l'associativité et la commutativité de l'addition et de la multiplication offrent de nouvelles opportunités dans l'extraction des *fma*. L'efficacité de ce travail dépend alors des décisions prises lorsque plusieurs choix d'extraction sont en conflit. Le premier objectif de notre heuristique est de déterminer la transformation qui aboutit au nombre maximal de *fma*. Nous sélectionnons ensuite, parmi les solutions présentant un nombre maximal de *fma*, la meilleure expression vis à vis du critère de sélection secondaire choisi, par exemple l'équilibrage de l'arbre.

Remarque 9.5
Notons que, dans certains cas, une réduction du nombre de fma *extraits de l'expression permettrait d'obtenir une meilleure solution vis à vis du critère choisi (arbre mieux équilibré par exemple). Cependant, nous préférons privilégier le nombre d'opérations au détriment de la structure de l'expression.*

Partant d'une expression quelconque contenant des additions et des multiplications, nous cherchons à construire des *fma*. Nous avons vu que la factorisation n'offre pas d'alternatives dans l'écriture de la nouvelle expression ; la position des termes dans l'arbre est en effet figée (l'expression $x \times y_1 + x \times y_2 + z_1 + z_2$ sera toujours réécrite en $x \times (y_1 + y_2) + z_1 + z_2$). Le processus de sélection n'apparaît qu'en présence de plusieurs factorisations différentes. Or, l'extraction des *fma* repose sur une grande liberté dans la construction de la nouvelle expression. Pour un seul et même *fma*, il existe plusieurs façons de réécrire l'expression ($x \times y + z_1 + z_2$ peut être réécrit sous la forme $\mathrm{fma}(x, y, z_1 + z_2)$ ou $z_1 + \mathrm{fma}(x, y, z_2)$). La position de certains des termes de l'expression peut donc varier en fonction de l'écriture choisie.

Cependant, comme nous l'avons expliqué au début de ce chapitre, seules quelques transformations élémentaires seront autorisées. En effet, en raison de l'associativité des opérateurs manipulés, le nombre de transformations élémentaires qu'il est possible de construire est très important. Notre objectif est donc de sélectionner celles qui permettent d'améliorer l'expression vis à vis du critère choisi ; il est également important de pouvoir couvrir à partir de ces transformations élémentaires une grande partie de l'espace des expressions équivalentes.

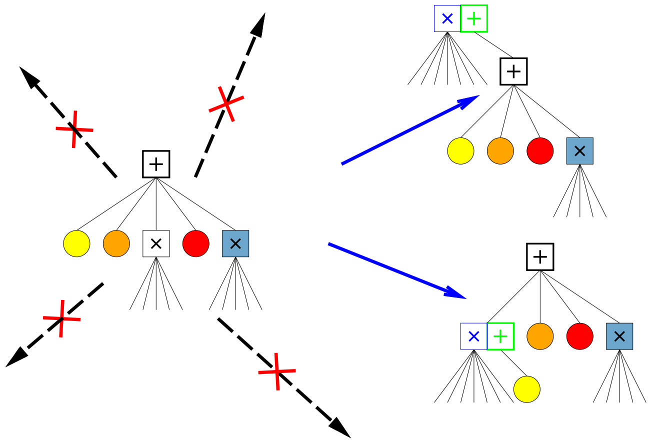


FIG. 9.6 – *Exemple de transformations élémentaires.*

La figure 9.6 présente deux transformations élémentaires intéressantes que nous avons sélectionnées pour tenter d'intervenir sur la position dans l'arbre des différents termes du *fma*. L'expression de départ est composée d'une addition avec 5 opérandes dont 2 multiplications et 3 termes quelconques (non multiplicatifs). Chacune des deux multiplications permet de construire un *fma*, les termes restants peuvent être placés indifféremment en dessus ou en dessous du *fma* ; de nombreuses expressions peuvent donc être construites (par application d'une transformation élémentaire). Considérons la construction d'un *fma* à partir de l'addition et de la multiplication non colorées sur la figure 9.6. Parmi toutes les expressions qu'il serait possible d'atteindre par application d'une transformation élémentaire, seules deux d'entre elles sont conservées. La transformation élémentaire en partie haute de la figure place la multiplication dans la plus haute position (tous les autres termes sont placés en dessous). La seconde transformation (en partie basse de la même figure) positionne le plus bas possible la multiplication et le terme (non multiplicatif) de valeur minimale vis à vis du critère de sélection choisi (par exemple le terme de gravité minimale).

Remarque 9.6
Il est important de distinguer les multiplications et les termes non multiplicatifs afin d'empêcher la suppression d'une opportunité de fma *(dans le cas où une multiplication serait séparée du terme additif car placée sous un* fma*). Cependant, si aucun terme non multiplicatif n'est disponible, la multiplication de valeur minimale pour le critère retenu sera utilisée pour la construction.*

Notre implémentation actuelle de cette heuristique ne comporte que les deux transformations élémentaires présentées sur la figure 9.6 ; ce choix s'est avéré suffisant en pratique. Il est toutefois possible d'ajouter de nouvelles transformations élémentaires (et donc obtenir un nombre n arbitraire de transformations) tout en interdisant l'application de **toutes** les transformations possibles (complexité exponentielle). Pour chacune des p multiplications placées sous une addition, n transformations élémentaires sont donc envisageables. La transformation élémentaire à appliquer est sélectionnée de manière à choisir, parmi les $n \times p$ expressions « autorisées », la meilleure vis à vis du critère retenu (voir section 9.1).

9.3.2 Présentation de l'heuristique

Notre heuristique applique itérativement la fonction d'extraction des *fma* qui est présentée sur la figure 9.7. Étant donnée une expression E, cette fonction retourne une nouvelle expression contenant un *fma* supplémentaire. La notation `fma`(x, y, z) est utilisée pour décrire la construction *fma* correspondant à l'opération $x \times y + z$. La fonction `MinVal` retourne l'expression minimisant le critère choisi.

Lorsque la multiplication contient plus de deux opérandes, l'associativité et la commutativité peuvent être utilisées pour construire un arbre plus ou moins bien équilibré (fonction `heuristic_balance` de la

Function multiply-add extraction(n)
- *Input:* n is a node
- *Output:* n after **one** multiply-add extraction

if (operator(n) is "+") **then**
  let *list of Mult* = list_of_multiplications(n)
  if (Not_empty(*list of Mult*)) **then**
    let *list of Rem* = list_of_remaining_nodes(n,*list of Mult*)
    if (Not_empty(*list of Rem*)) **then**
      let R_1 = MinVal(*list of Rem*)
    else
      let R_1 = MinVal(*list of Mult*)
      remove R_1 from *list of Mult*
    end if
    let *list of Candidates* = (*nil*)
    for each M_i **in** *list of Mult*
      (M_{i1},M_{i2}) = heuristic_balance(M_i)
      add `fma`(M_{i1},M_{i2},$n[M_i \rightarrow$"0"]) to *list of Candidates*
      add $n[M_i \rightarrow$ `fma`$(M_{i1},M_{i2},R_1)][R_1 \rightarrow$"0"] to *list of Candidates*
    end for each
    return MinVal(*list of Candidates*)
  else
    return n
  end if
end if

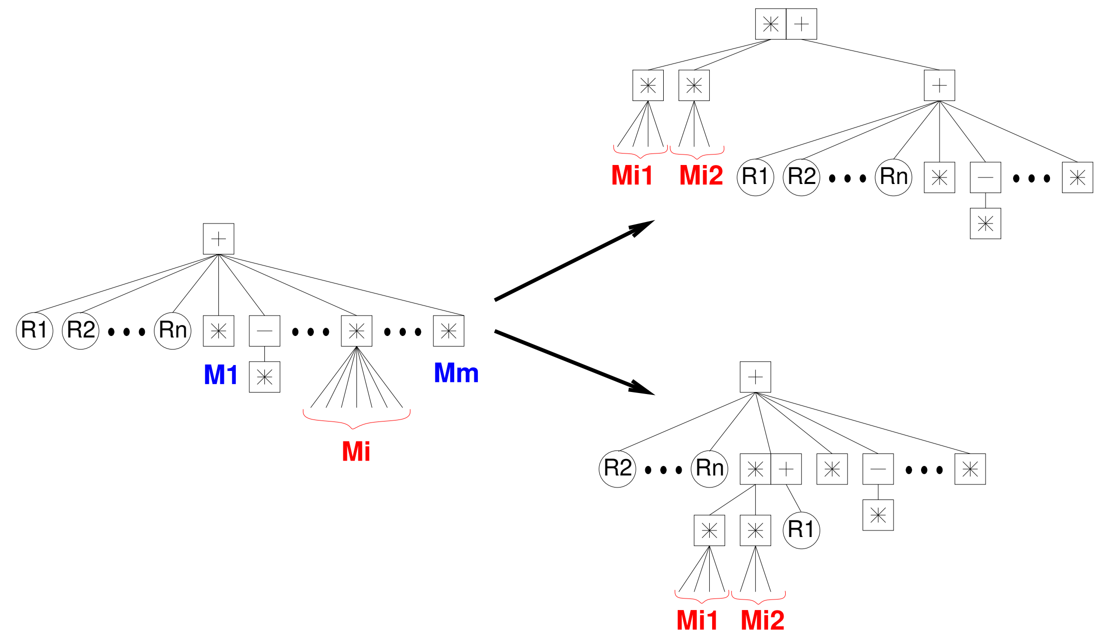


FIG. 9.7 – *Heuristique d'extraction des* fma.

figure 9.7). Il faut en effet construire, à partir des n opérandes de la multiplication, les deux opérandes « multiplicatifs » du *fma*. Étant donnés n termes et une fonction de coût, nous cherchons à construire deux paquets de termes ayant des coûts totaux équivalents. La fonction de coût utilisée est paramétrique et peut reprendre indifféremment le poids W, le chemin critique C, la gravité G ou toute autre mesure. Ce problème est en tous points similaire au problème PARTITION, bien connu pour être NP-Complet [Kar72].

Nous utilisons donc ici une heuristique classique de type glouton. Les deux paquets sont remplis en utilisant les termes par ordre de coût décroissant ; le terme courant étant systématiquement placé dans le paquet de coût total minimal. Cette technique est illustrée par la figure 9.8. La mesure utilisée dans cet exemple est le poids W des opérateurs ; les deux opérandes du *fma* construit sont de poids équivalents.

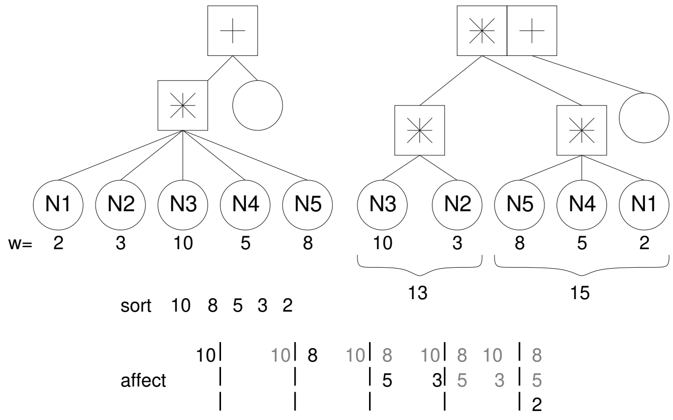


FIG. 9.8 – *Heuristique gloutonne de partitionnement.*

Remarque 9.7
Cette heuristique permet de construire deux paquets de coûts totaux équivalents ; le coût total correspondant ici à la somme des coûts élémentaires de chacuns des éléments du paquet. Elle peut donc être utilisée de manière paramétrique à partir d'une fonction de coût quelconque. Or, si le critère que l'on cherche à minimiser est la gravité, *il est préférable d'utiliser directement la variation de l'algorithme de Huffman qui est présentée ci-dessous, à la section 9.4.*

Étant donnée une expression avec n termes non multiplicatifs et m multiplications ayant chacunes p fils, la complexité de cette fonction est en $\mathcal{O}(n + m \times p.ln(p))$, en supposant l'utilisation d'un tri de p éléments en $\mathcal{O}p.ln(p)$. La fonction doit bien sûr être appliquée à tous les nœuds de l'arbre, jusqu'à ce que aucun *fma* ne puisse plus être construit.

9.4 Équilibrage de l'arbre

Le travail que nous proposons ici vise à utiliser les propriétés algébriques des opérateurs afin de transformer les expressions dans une forme mieux adaptée à une évaluation efficace sur les processeurs modernes. Pour cela, la manipulation d'une structure de donnée contenant des opérations *n-aires* est très pratique. Cependant, la majorité des jeux d'instructions ne supportent les opérations classiques que sous la forme d'un code 3-adresses. Une transformation de la structure n-aire des expressions en arbres binaires sera donc obligatoirement effectuée avant l'exécution. Or, tous les arbres binaires que l'on peut construire à partir d'un arbre n-aire contiendront un nombre d'opérations identique, mais n'auront pas les mêmes caractéristiques (profondeur, équilibrage, etc.). Les temps d'exécution peuvent donc également varier de manière significative.

9.4.1 Algorithme de Huffman

Afin de sélectionner un arbre binaire adapté à une exécution sur un processeur avec parallélisme d'instructions, nous cherchons à intervenir sur la forme de l'arbre et la position des opérandes. Nous proposons pour cela d'utiliser une variation de la technique de codage introduite par Huffman [Huf52]. Ses travaux aboutissent à une méthode de codage d'un message à partir d'un ensemble de n symboles ayant des probabilités quelconques ; l'algorithme est fondé sur la construction d'un arbre binaire minimisant $\Sigma_{i=1}^{n} P_i \times L_i$ avec P_i la probabilité du symbole i et L_i sa profondeur dans l'arbre. La figure 9.9 illustre cette technique pour le codage d'un message contenant 13 symboles.

Algorithme 2 Algorithme de codage de Huffman.

Fonction Huffman(L)
- ▷ - *L est une liste de n symboles*
- ▷ - *Créé un arbre binaire*
- Trier les n symboles de L par ordre de probabilité croissante
- **Soit** P_i la probabilité du i^{eme} symbole de L (noté s_i)
- **Pour** $i \leftarrow 1, n-1$ **Faire**
  - Extraire s_1 et s_2 de L
  - Concaténer s_1 et s_2 dans un nouvel élément r
  - **Soit** $P_r = P_1 + P_2$ la probabilité de r
  - Chercher une position j telle que $P_j \leq P_r$ et $\forall k > j, P_k > P_r$
  - Insérer r dans L à la position j
- **Fin Pour**

Fin

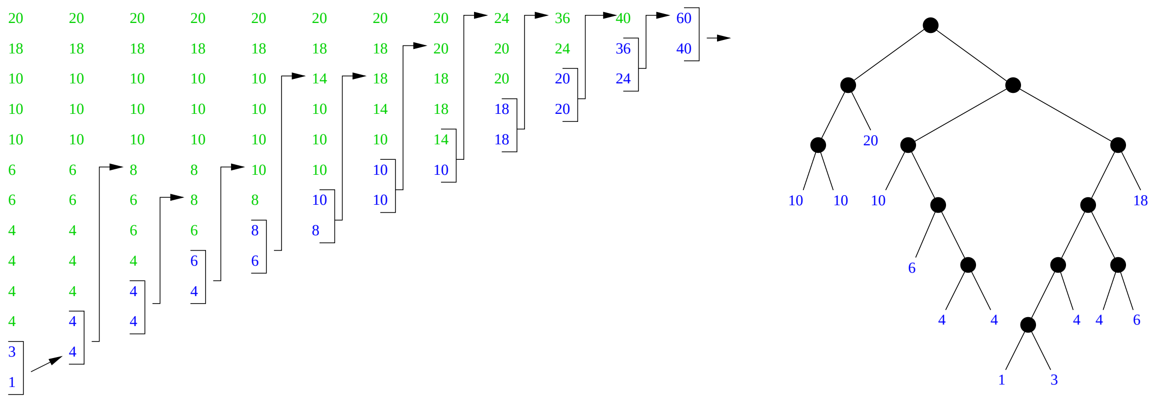


FIG. 9.9 – *Codage par la méthode de Huffman.*

9.4.2 Variation de l'algorithme de Huffman

Les similitudes entre ce problème de codage et la construction de la représentation binaire d'une opération sur n opérandes nous ont amenés à mettre en place l'algorithme 3. L'approche utilisée est similaire à celle de l'algorithme 2. Toutefois, le regroupement de deux symboles dans le cas du codage se caractérise simplement par une somme de probabilités ; aucun coût supplémentaire n'est introduit. Or, dans le cadre de l'évaluation du coût d'une opération binaire, nous souhaitons faire intervenir à la fois le coût des deux opérandes et le coût élémentaire de l'opérateur.

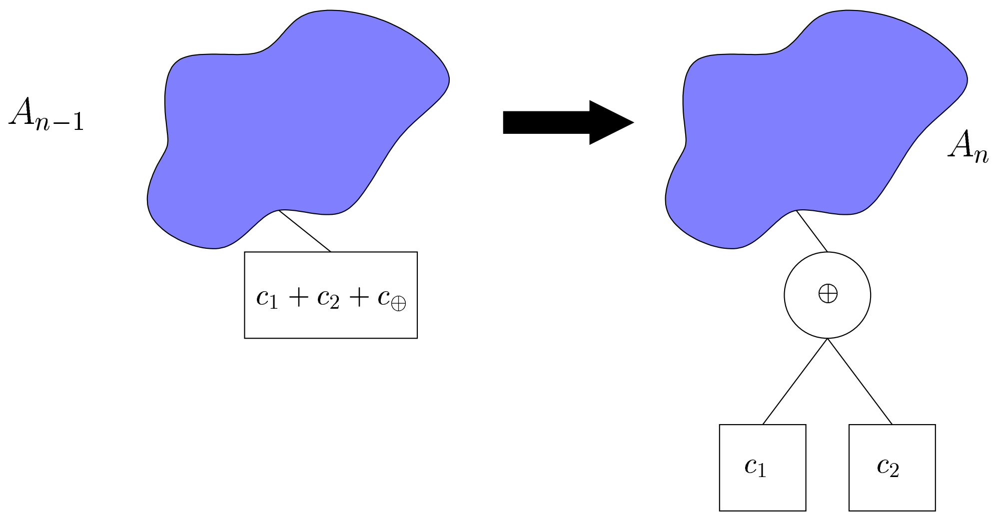


FIG. 9.10 – *Processus de construction.*

Théorème 9.1
Soit $\mathcal{B}(n)$ la procédure de binarisation d'une opération $\oplus$ pour n opérandes décrite par l'algorithme 3. Soient c_i et d_i respectivement le coût et la profondeur d'un opérateur ou d'un opérande i dans l'arbre binaire. Soit $c_\oplus$ le coût de l'opérateur $\oplus$. Par construction, $\mathcal{B}(n)$ minimise la fonction de coût :

$$C = \Sigma_{i=1}^{n} c_i \times d_i + c_\oplus \times \Sigma_{j=1}^{n-1} d_j$$

Preuve 9.1

La preuve est construite par récurrence sur n. Notons que, dans la mesure où la procédure de construction $\mathcal{B}(n)$ est déterministe, elle ne peut construire qu'un seul arbre de n opérandes.

• $\mathcal{B}(2)$ construit évidemment l'arbre de 2 opérandes minimisant la fonction C.

• Supposons que la procédure de binarisation $\mathcal{B}(n-1)$ soit optimale (au sens minimisation de C), puis montrons que $\mathcal{B}(n)$ l'est aussi.

Soit $c_1 \leq c_2 \leq c_3 \leq \ldots \leq c_n$ les coûts des n opérandes de l'opération que l'on cherche à binariser. Par hypothèse, la procédure de construction $\mathcal{B}(n-1)$ appliquée aux opérandes de coûts $\{c_1 + c_2 + c_\oplus, c_3, \ldots, c_n\}$ construit un arbre optimal. Soit A_{n-1} cet arbre.

Soit A_n le sous-arbre construit en remplaçant l'opérande de coût $c_1 + c_2 + c_\oplus$ par le sous-arbre constitué d'un opérateur $\oplus$ et de deux opérandes de coûts c_1 et c_2. Cette construction, qui correspond à la première étape de l'algorithme 3, est illustrée par la figure 9.10. La procédure $\mathcal{B}(n)$ est constituée de cette transformation, puis de l'application de $\mathcal{B}(n-1)$. Nous cherchons à montrer que l'optimalité de l'arbre A_{n-1} implique l'optimalité de A_n.

Supposons que la transformation (1) de la figure 9.11 nous fait passer de A_{n-1} optimal à A_n non optimal. Soit C_{n-1} le coût optimal de A_{n-1} et C_n le coût non optimal de A_n. Par construction nous avons $C_n = C_{n-1} + c_1 + c_2$ (profondeur augmentée de 1 pour ces deux opérandes). Si $A(n)$ n'est pas optimal, il est possible de construire un arbre A'_n optimal tel que $C'_n < C_n$ (transformation (2) sur la figure 9.11). Comme c_1 et c_2 sont les coûts minimaux, il est toujours possible de les regrouper dans un même sous-arbre sans augmenter C. Nous pouvons donc construire un arbre A'_n optimal et tel que les opérandes de coûts c_1 et c_2 se trouvent placés sous la même opération. La substitution du sous-arbre contenant les coûts c_1 et c_2 par un opérande de coût $c_1 + c_2 + c_\oplus$ à partir de A'_n construit un nouvel arbre A'_{n-1} de coût $C'_{n-1} = C'_n - (c_1 + c_2)$ (étape (3) sur la figure 9.11). D'après $C'_n < C_n$, nous obtenons

Algorithme 3 Construction d'arbres binaires par une variation de l'algorithme de Huffman.

Fonction Binarisation($\oplus, L, C$)
  ▷ - $\oplus$: *Un opérateur associatif-commutatif*
  ▷ - L : *Une liste de n opérandes*
  ▷ - C : *Une fonction de coût*
  Trier les n éléments de L par ordre croissant sur C
  Pour $i \leftarrow 1, n-1$ **Faire**
    ▷ ***Soit*** *$L[i]$ le i^{eme} élément de L*
    Extraire $L[1]$ et $L[2]$ de L
    Construire l'opération binaire $\oplus(L[1], L[2])$
    Soit $c = C(L[1]) + C(L[2]) + C(\oplus)$ le coût de l'opération
    Chercher la position j telle que $P_j \leq P_r$ et $\forall k > j, P_k > P_r$
    Insérer $\oplus(L[1], L[2])$ dans L à la position j
  Fin Pour
  Retourner $\oplus(L[1], L[2])$
Fin

donc $C'_{n-1} < C_{n-1}$, ce qui est contraire à l'hypothèse de départ puisque C_{n-1} est supposé de coût minimal.

La construction proposée par l'algorithme 3 permet donc bien de construire un sous-arbre optimal A_n à partir de A_{n-1} optimal. Par récurrence, $\mathcal{B}(n)$ minimise C. *Q.E.D.*

La figure 9.12 présente la construction d'une expression binaire en utilisant pour le coût, la notion de poids définie à la section 9.1. Le poids de l'opérateur est fixé à 1 et les six opérandes (pouvant représenter un sous-arbre quelconque) sont de poids $w_i = i$. Le processus de création et l'expression obtenue sont présentés respectivement en partie gauche et droite de la figure. La minimisation de la fonction C pour avec $c_i = w_i$ permet ici de minimiser la gravité de l'expression $G = \Sigma_{i=1}^{2n-1} w_i \times d_i$.

L'utilisation du poids des opérandes comme coût permet de construire un arbre *minimisant* la gravité de l'expression. De la même manière, il est possible de construire un arbre *maximisant* la gravité de l'expression en utilisant cette variation de l'algorithme de Huffman. Pour cela, il suffit de construire une fonction de coût telle que $\forall i \in [1, n-2], c(o_i) + c(o_{i+1}) + c(\oplus) < c(o_{i+2})$ et $w(o_i) > w(o_j) \Rightarrow c(o_i) < c(o_j)$. De cette manière, chaque nouvelle opération créée reste de coût minimal et l'algorithme construit progressivement un arbre de la forme $(o_n + (o_{n-1} + \ldots + (o_3 + (o_2 + o_1))\ldots))$. On obtient par exemple le résultat cherché en utilisant $c_\oplus = 0$ et la fonction $c_i = 2^{-i}$ avec i le rang obtenu par le tri des w_i décroissants. L'arbre binaire obtenu est présenté figure 9.13. Contrairement à la version de la figure 9.12 qui est équilibrée, on obtient ici une expression déséquilibrée au maximum.

Application à une expression complexe Lorsque l'expression manipulée est complexe, c'est-à-dire composée de différents opérateurs, la technique de binarisation proposée peut être appliquée localement à tous les opérateurs associatifs-commutatifs, tout en donnant un résultat optimal pour l'expression complète. Cela suppose bien évidemment que les propriétés algébriques impliquant plusieurs opérateurs différents, telles que la distributivité, ne soient pas utilisées en cours de binarisation. Ces transformations doivent faire l'objet d'un travail préalable tel que ceux décrits aux sections 9.2 et 9.3.

Théorème 9.2
Soit A un arbre composé de m opérations n-aires associatives-commutatives ; l'opérateur j possède n_j opérandes. L'application locale de la procédure $\mathcal{B}(n_j)$ à toutes les opérations permet de construire un arbre minimisant la fonction de coût :

$$C = \Sigma_{i=1}^{p} c_i \times d_i + \Sigma_{j=1}^{q} c'_j \times d'_j$$

avec $p = \Sigma_{k=1}^{m} n_k$ et $q = p - m$

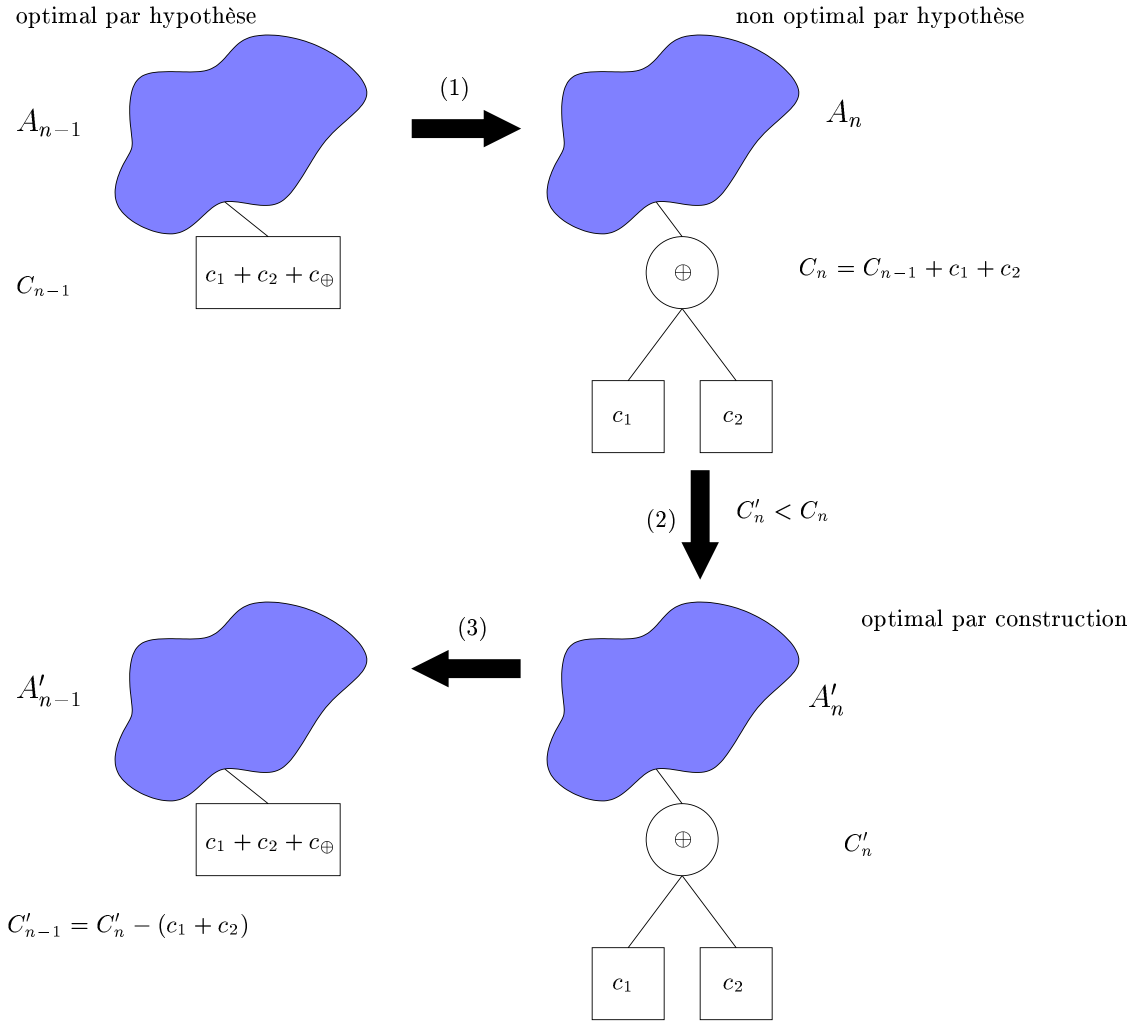


FIG. 9.11 – *Preuve par contradiction avec l'hypothèse de départ.*

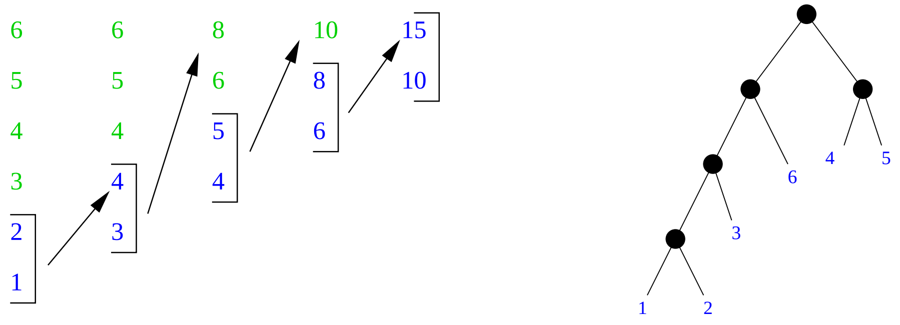


FIG. 9.12 – *Binarisation de l'arbre avec $c_i = w_i$ et $c_\oplus = 1$.*

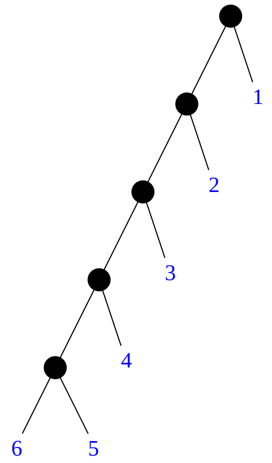


FIG. 9.13 – *Binarisation de l'arbre avec $c_i = 2^{-i}$ et $c_\oplus = 0$*

Preuve 9.2

La procédure $\mathcal{B}(n)$ permet de minimiser le coût de chacun des m sous-arbres indépendamment. Nous avons donc minimisé chacun des m termes de la forme :

$$C = \Sigma_{i=1}^{n_k} c_i \times d_i + \Sigma_{j=1}^{n_k-1} c'_k \times d'_j \text{ pour } k \in [1,m]$$

Appelons A_1 l'arbre construit à partir de toutes les binarisations locales. Supposons que A_1 n'est pas optimal. Sachant qu'il est impossible de permuter les opérandes de différents opérateurs, il existe une permutation de deux sous-arbres S et S' d'un même opérateur $\oplus$ telle que l'arbre A_2 construit par permutation vérifie $C_{A_2} < C_{A_1}$. Si l'écart de profondeur entre S et S' est noté k et si S est pris, sans perte de généralité, plus haut que S', la permutation nous donne $C_{A_2} = C_{A_1} + k \times C_S - k \times C_{S'}$. Nous obtenons donc $C_S < C'_S$. Or, dans ce cas, en permutant S et S' dans le sous-arbre construit pour l'opérateur $\oplus$, nous pouvons minimiser sa fonction de coût. Cela est contradictoire avec l'hypothèse de départ qui suppose que toutes les binarisations locales (dont celle sur notre opérateur $\oplus$) sont optimales.

Q.E.D.

Notons que, comme chaque sous-arbre optimal est traité comme un opérande de coût $c' = \Sigma_{i=1}^{n_j} c_i + (n_j - 1) \times c_\oplus$ par l'opérateur de niveau supérieur, l'application locale de la procédure de construction pour chacun des opérateurs associatifs-commutatifs *pris dans n'importe quel ordre* est optimale.

Remarque 9.8
Ce travail d'équilibrage doit être fait au dernier moment, car il détruit la structure n-aire des opérateurs manipulés.

9.5 Transformations algébriques et évaluation d'expressions

Les expériences présentées au chapitre 7, prouvent l'efficacité des transformations algébriques pour accélérer l'évaluation des expressions sur les processeurs modernes. Cependant, l'application manuelle de ces transformations est aussi difficile que fastidieuse (et plus ou moins portable !). Nous proposons donc une automatisation de ces transformations grâce à la mise en place de plusieurs techniques.

9.5.1 Un processus en plusieurs étapes

Le processus complet de transformation algébriques comporte 4 étapes principales :

1. La première phase, présentée au chapitre 8 permet de mieux exploiter les opportunités de transformations qui existent au sein des expressions manipulées. La `forward substitution` permet de reconstruire des expressions complexes lorsque celles-ci ont été cassées par le programmeur ou par un travail d'optimisation (souvent devenu obsolète entre temps). La phase de normalisation augmente le degré de liberté des transformations à venir et effectue un travail de simplifications.
2. Une heuristique de factorisation permettant de réduire le nombre d'opérations a été mise en place. Cette transformation cherche également à construire la *meilleure* expression vis à vis d'un critère donné (traduisant par exemple l'équilibrage de l'arbre).
3. L'objectif de la construction des *fma* est d'extraire un nombre maximum de ces opérations, tout en sélectionnant les expressions répondant le mieux possible au critère défini.
4. Enfin, la construction d'arbres binaires à partir de la représentation n-aire permet d'intervenir sur la position de chacun des opérandes. L'algorithme proposé minimise la gravité G_E de l'expression. Notons cependant que le nombre d'opérations de change pas.

Suite à l'application de la *forward substitution*, ces quatre étapes de transformations doivent être accompagnées d'un travail de *déplacement de code invariant* et d'*élimination des sous-expressions communes*. L'utilisation de ces techniques dans un contexte associatif-commutatif sera présentée dans la partie IV.

9.5.2 Mise en place de stratégies d'optimisations

L'application de transformations algébriques permet de construire des expressions structurellement « mieux adaptées » à une exécution sur une architecture avec parallélisme d'instruction. Ce travail est donc potentiellement utilisable avec tout processeur superscalaire disposant de plusieurs unités de calcul et/ou un pipeline de plusieurs étages. Nos expériences ont montré que la structure de l'expression (profondeur, largeur, nombre d'opérateurs, ...) intervient dans l'efficacité de l'exécution. Cependant, malgré des caractéristiques très proches, différents couples processeur-compilateur peuvent obtenir de meilleurs résultats avec telle ou telle structure pour évaluer l'expression. Les caractéristiques du processeur ainsi que les techniques d'ordonnancement des instructions employées peuvent en effet intervenir sur le choix des expressions. Nous avons par exemple pu remarquer que les applications ONDE24 et APPLU compilées avec le compilateur `MIPS` sur un processeur `R10000` obtenaient de meilleurs performances avec des expressions déséquilibrées. La raison principale de ce phénomène est la décision prise par le compilateur de dérouler fortement la boucle malgré un nombre insuffisant de registres. L'augmentation du parallélisme d'instructions obtenue avec une expression équilibrée aurait pourtant dû permettre de réduire le facteur de déroulage de boucles et donc, dans le même temps, réduire le nombre de registres utiles. Nous avons pu également constaté que le compilateur Fortran SUN version 5.0 déroule systématiquement deux les boucles rencontrées sans prendre en compte la pression de registres.

La définition de stratégies pour les optimisations présentées dans ce chapitre peut faire intervenir différents paramètres. Le critère de sélection (poids, chemin critique, gravité, etc.) permet tout d'abord d'intervenir sur la forme finale de l'expression. Il est également possible d'intervenir sur la définition des transformations élémentaires qui sont autorisées (voir section 9.3). En pratique, nous avons utilisé des stratégies non pas spécifiques à un processeur mais visant à faire évoluer les expressions vers une structure plutôt qu'une autre. On pourra par exemple chercher à construire un arbre équilibré, un arbre de hauteur maximale avec les éléments de poids fort placés au plus haut ou au contraire au plus bas, etc. Ce mécanisme de stratégies d'optimisations a été implémenté dans notre prototype (voir chapitre 10).

9.5.3 Nombre d'opérations vs équilibrage

Comme nous l'avons déjà indiqué à plusieurs reprises, le seul critère idéal permettant de choisir entre toutes les expressions mathématiquement équivalentes est le temps d'exécution réellement mesuré. Malheureusement, le nombre important d'expressions et les contraintes de durée que nous impose la compilation de programmes scientifiques rendent quasiment inexploitable la mesure du temps d'exécution. Nous avons donc établi des préférences parmi les paramètres permettant de caractériser une expression. Dans un premier temps, nous souhaitons réduire au maximum la charge de calcul :

- La réduction de force permet par exemple de réduire la puissance de calcul utile pour l'évaluation d'une opération donnée.
- La factorisation diminue le nombre d'opérations à évaluer pour l'obtention du résultat.
- L'exploitation des opérations fusionnées comme le *fma* permet également de réduire la charge de calcul (ou tout au moins le nombre d'instructions à charger).

À nombre d'opérations égal, les techniques proposées cherchent ensuite à intervenir sur la forme de l'expression (équilibrée ou non). Le nombre d'opérations reste constant, mais l'on espère pouvoir mieux exploiter le parallélisme d'instructions. En choisissant par exemple la gravité comme critère de sélection, la construction des *fma* ainsi que la factorisation aboutiront à des expressions bien équilibrées.

Le choix de ces priorités repose sur les résultats expérimentaux que nous avons obtenus. Cependant, dans certains cas (rares en pratique), la réduction du nombre d'opérations va à l'encontre de l'équilibrage de l'expression. Par exemple, la construction des *fma* pour une expression de la forme $x_1 \times x_2 + x_3 \times x_4 + \ldots + x_{n-1} \times x_n$ donne une expression contenant un nombre important de *fma* ; la diminution de la charge de calcul est donc profitable. Cependant, l'équilibrage de l'expression devient médiocre, car l'arbre prend la forme d'un râteau $\mathrm{fma}(x_1, x_2, \mathrm{fma}(x_3, x_4, \mathrm{fma}(\ldots$, et toutes les opérations sont alors séquentialisées. La figure 9.14 illustre ce phénomène pour une somme de 10 multiplications. Deux solutions existent pour contourner ce problème : si le nombre de registres n'est pas critique, un déroulage de boucles et l'application du *software pipelining* permet d'obtenir suffisamment de parallélisme d'instructions. Dans le cas contraire, le « sacrifice » d'un ou plusieurs *fma* permet de « casser » le râteau en un ou plusieurs endroits afin de retrouver une expression équilibrée (cf. figure 9.14). Nous avons par exemple utilisé cette technique pour un code de l'Institut Français du Pétrole dont le noyau de calcul est une expression constituée d'un râteau de 16 *fma*. Le découpage du râteau en 4 parties a permis d'améliorer les performances du noyau de calcul de 11% sur un processeur P2SC cadencé à 160MHz. Malgré tout, le choix de privilégier le volume de calcul donne en pratique de bons résultats pour les architectures actuelles qui ne disposent pas d'un nombre très important d'unités de calculs.

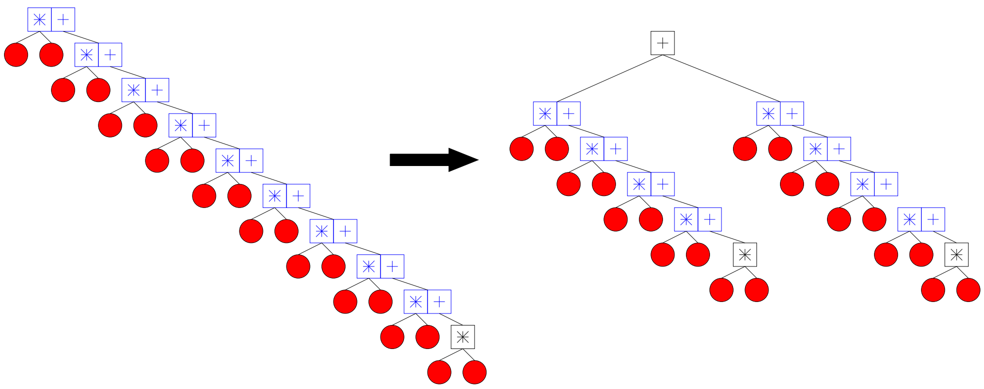


FIG. 9.14 – *Expression mise sous forme de râteau par l'extraction des* fma

Chapitre 10

Implémentation et expérimentations

Les travaux présentés dans cette partie ont donné lieu, dans un premier temps, à des expérimentations manuelles permettant de mieux définir les opportunités d'amélioration. Les techniques mises en place ont ensuite été implémentées afin de permettre une expérimentation plus systématique grâce à l'application automatique des transformations.

10.1 Implémentation

Nous avons vu aux chapitres 8 et 9 que le processus de transformations algébriques proposé est fondé, d'une part, sur des techniques de réécriture et, d'autre part, sur des transformations spécifiques. Toutes les techniques présentées dans cette partie ont été implémentées sous la forme d'un prototype appelé `EOLE` (*Evaluation Optimization for Loops and Expressions*). Le parseur d'expressions dans EOLE permet de travailler directement sur n'importe quelle expression arithmétique. Nous avons également mis au point une interface avec l'outil `PIPS`, présenté en annexe B, de manière à permettre l'application automatique de nos transformations sur un programme Fortran au sein d'un processus plus général de transformations. La figure 10.1 propose un schéma global du prototype.

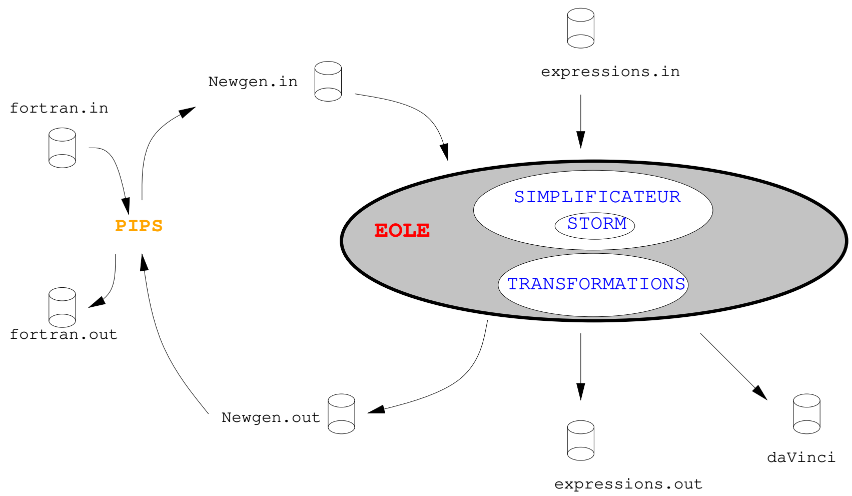


FIG. 10.1 – *Le prototype EOLE.*

10.1.1 EOLE

L'implémentation repose, pour la partie réécriture, sur le simplificateur (environ 25000 lignes de code) développé par Patrick Baudin au CEA, dans le cadre du projet CAVEAT. Ce dernier utilise le moteur de pattern-matching associatif-commutatif `STORM` développé par les universités d'Orléans et Stony Brook (environ 10000 lignes de code). Les transformations spécifiques ainsi que le cœur du prototype (environ 5000 lignes de code) ont été implémentés sur la même structure de données que le simplificateur. Le travail a été réalisé en C++ et utilise les fonctionnalités de la librairie standard STL.

`EOLE` accepte en entrée un fichier contenant une liste d'expressions, applique des transformations algébriques puis écrit le résultat soit sur la sortie standard, soit dans un fichier dont le nom est spécifié par l'utilisateur sur la ligne de commande. L'utilisateur peut indiquer la liste des transformations qu'il souhaite voir appliquées aux expressions ainsi que l'ordre dans lequel elles doivent l'être. Un ordre par défaut est défini.

Normalisation Les expressions sont mises sous une forme adaptée à l'application des transformations suivantes, par utilisation des techniques de réécriture. Tous les travaux de normalisation présentés au chapitre 8 ont été implémentés. Des simplifications algébriques, de l'évaluation partielle et de la réduction de force sont également appliquées. Le mécanisme de réécriture du simplificateur de CAVEAT est directement réutilisé. Deux bases de règles sont manipulées afin de séparer les transformations que nous souhaitons appliquer au début et à la fin du processus global d'optimisation. L'ajout de nouvelles transformations algébriques appliquées par réécriture est immédiat.

Développement Le développement des expressions est également effectué à partir du mécanisme de réécriture. Les expressions sont développées au maximum. Notons que cette transformation est placée dans une base de règle spécifique afin de permettre à l'utilisateur de développer ou non ses expressions.

Factorisation L'heuristique de factorisation présentée à la section 9.2 a été implémentée de manière paramétrique. Elle fonctionne pour n'importe quel couple d'opérateurs distributifs et les opérateurs complément, comme par exemple le `moins unaire`, sont également pris en compte.

Extraction des *fma* Les opérations de type *fma* sont extraites des expressions par l'heuristique présentée à la section 9.3. Un nouvel opérateur a été introduit dans la structure de donnée du prototype `EOLE`.

Équilibrage de l'expression La variation de l'algorithme de Huffman, présentée à la section 9.4 et permettant de binariser les expressions, n'a pas été implémentée directement dans EOLE pour des raisons pratiques. En effet, les opérateurs associatifs-commutatifs sont manipulés sous une forme n-aire dans la représentation interne du simplificateur de CAVEAT ; l'écriture binaire pour un opérateur associatif-commutatif est impossible en raison de la n-arisation automatisée. Nous avons donc préféré implémenter cette technique directement dans l'outil `PIPS` (voir paragraphe 10.1.2 et annexe B).

Le prototype EOLE permet d'afficher le résultat des différentes étapes de transformation (option "-i") et d'obtenir des statistiques (option "-s") sur le nombre d'opérations, le chemin critique et la gravité des expressions après chacune des transformations. L'option "-g" permet également de générer un fichier contenant la représentation sous forme de graphe coloré `daVinci` des expressions manipulées ; la figure 10.2 illustre le graphe obtenu pour l'expression du noyau de calcul de l'application Onde24 (cf. section 7.2.2).

Stratégies d'optimisation Afin de permettre une expérimentation fondée sur différentes hypothèses concernant la forme recherchée des expressions, l'implémentation permet de définir puis d'utiliser des *stratégies d'optimisation* : il est possible de choisir le critère de sélection, par exemple le chemin critique ou la gravité, qui sera utilisé par les différents algorithmes implémentés. De même, le choix des transformations élémentaires autorisées pour l'heuristique d'extraction des *fma* est paramétrique.

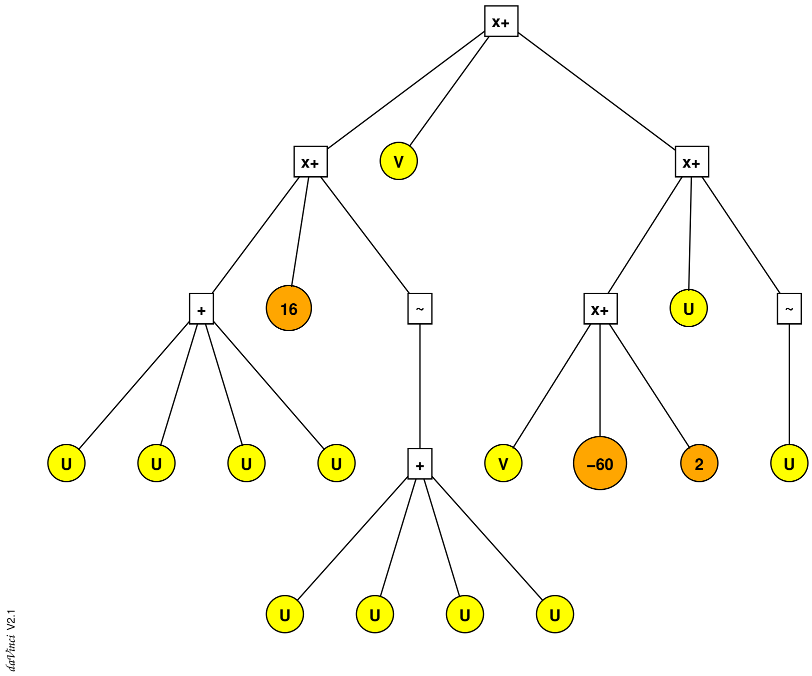


FIG. 10.2 – *Représentation des expressions avec daVinci.*

Cette fonctionnalité nous a, par exemple, été utile pour définir deux stratégies, l'une construisant des expressions *équilibrées* et l'autre aboutissant à des expressions *déséquilibrées*. La première s'est avérée efficace pour une évaluation sur processeurs P2SC ou UltraIIi, tandis que la seconde donne de bons résultats sur processeur R10K.

10.1.2 Interface avec `PIPS`

Une interface avec l'outil `PIPS` permet d'appliquer les transformations algébriques à toutes les expressions d'une application réelle écrite en Fortran. La communication entre `PIPS` et `EOLE` se fait, dans les deux sens, par manipulation de fichiers temporaires contenant la représentation des expressions dans la structure de données `NEWGEN` (voir figure 10.1). Les outils `PIPS` et `NEWGEN` sont décrits en annexe B. Notons que `PIPS` permet également d'appliquer la passe préalable de *forward substitution* présentée au chapitre 8.

10.2 Expériences et résultats

L'interfaçage entre `EOLE` et `PIPS` permet d'appliquer les transformations présentées dans cette thèse, de manière automatique, à toutes les expressions d'un module ou d'un programme complet écrit en Fortran. Les résultats présentés ça et là, tout au long de cette partie, ont été obtenus par optimisation manuelle d'expressions spécifiquement choisies. Nous ajoutons ici des résultats obtenus automatiquement en considérant les applications dans leur intégralité. Les expériences décrites concernent trois applications : ONDE24, benchmark de l'Institut Français du Pétrole[39], APPLU, benchmark de la suite SpecFP 95 et CX3D, application industrielle d'origine allemande. Les temps d'exécution ont été mesurés pour la version initiale et pour les versions obtenues automatiquement à partir de deux stratégies d'optimisation différentes construisant respectivement des expressions équilibrées et deséquilibrées.

39. Les résultats sont indiqués pour deux jeux de données de tailles différentes.

Expériences réalisées sur un processeur IBM P2SC Les processeurs de la gamme P2SC, décrits à la section 7.2 présentent un fort degré de parallélisme d'instructions, grâce aux deux unités de calcul flottant pipelinées et aux instructions *fma*. Le compilateur IBM `xlf` version 4.1 a été utilisé avec les options "-O3 -qarch=pwr2 -qhot". Les mesures obtenues par l'exécution des applications sur un processeur P2SC cadencé à 160MHz sont présentées dans le tableau 10.1. Le premier constat à tirer de ces résultats est que la modification de la structure des expressions influt de manière significative sur les temps d'exécution des programmes. Un écart de 32% est par exemple mesuré entre la version initiale et la version équilibrée de l'application Onde24. Il faut noter également que le passage a une forme équilibrée des expressions semble être un bon choix pour l'évaluation des expressions sur un processeur P2SC.

Code	Version initiale	Version équilibrée	Version déséquilibrée
APPLU	35.75s	31.13s	35.55s
ONDE24 (MODELE 4)	31.52s	21.28s	23.85s
ONDE24 (MODELE 5)	217.14s	154.76s	216.28s
CX3D	37.30s	41.72s	42.13s

TAB. 10.1 – *Temps d'exécution mesurés sur P2SC 160MHz.*

Les temps d'exécution mesurés pour l'application CX3D montrent que les transformations appliquées ne sont pas bénéfiques. La taille du code et, en particulier, celle des quatre expressions formant le noyau de calcul, rend très difficile, voire même impossible, une optimisation manuelle de ce code. Nous pensons que les transformations algébriques appliquées ont engendré une augmentation du nombre d'opérations à cause d'une mauvaise factorisation ou de la transformation d'un carré en produit sans extraction des sous-expressions communes. Un examen plus poussé de cette application sera nécessaire pour mieux comprendre ce résultat.

Expériences réalisées sur un processeur MIPS R10K Les processeurs de la gamme MIPS R10K ont une architecture relativement proche de celle des processeurs P2SC. Notons toutefois que l'instruction *fma* proposée dans le jeu d'instructions MIPS IV ne permet pas de réduire la latence de cette opération fusionnée ; le gain se situe uniquement au niveau du chargement des instructions, i.e. une instruction à charger au lieu de deux. Les expériences ont été réalisées avec le compilateur MIPSPro version 7.1 en utilisant les options "-O3 -TARG:MADD=ON:proc=r10000:platform=ip27 -OPT:IEEE_arithmetic=3". Les résultats obtenus pour l'exécution des applications [40] sur processeur MIPS R10K cadencé à 180 MHz sont présentés dans le tableau 10.2. Même si les écarts sont plus faibles que ceux obtenus sur P2SC, les transformations algébriques appliquées interviennent ici aussi sur les temps d'exécution mesurés. Les résultats montrent que l'évaluation des expressions mises sous forme déséquilibrée donne de meilleurs résultats. Ce comportement peut paraître étrange dans la mesure où la forme déséquilibrée d'une expression permet d'extraire peu de calculs indépendants, ce qui est problématique dans la perspective d'une exécution des opérations en parallèle. Nous pensons que ce phénomène s'explique par le facteur de déroulage important choisi par le compilateur. En effet, en appliquant un fort déroulage des boucles et du software pipelining, le compilateur tente d'augmenter le nombre de calculs indépendants, mais ne se préoccupe pas du nombres de registres. Or, choisir la version équilibrée de l'expression devrait permettre de réduire le facteur de déroulage nécessaire et ainsi limiter la pression de registres.

Expériences réalisées sur un processeur SUN UltraIIi Les processeurs de la gamme SUN UltraIIi comportent également deux unités de calcul flottant pipelinées. Il n'existe cependant pas d'instruction du type *fma*. Nous avons mesuré les temps d'exécutions obtenus par utilisation du compilateur SUN WorkShop version 5.0 avec l'option "-fast". Le tableau 10.3 présente les résultats obtenus pour un processeur cadencé à 270MHz. Les transformations appliquées ont encore une fois un impact significatif sur les temps d'exécution mesurés avec des écarts allant jusqu'à 16%. La forme équilibrée des expressions

40. Un problème de mauvaise initialisation des variables du programme dans la version initiale du code a rendu impossible les expériences avec le code CX3D sur ce processeur.

Code	Version initiale	Version équilibrée	Version déséquilibrée
APPLU	65.07s	67.26s	63.90s
ONDE24 (MODELE 4)	38.19s	38.40s	37.98s
ONDE24 (MODELE 5)	122.01s	121.47s	122.64s
CX3D	initialisations incorrectes		

TAB. 10.2 – *Résultats sur R10K 180MHz.*

permet, comme sur le P2SC, d'obtenir de meilleurs résultats. Une fois encore, les mesures effectuées avec CX3D mettent en évidence un problème dans le processus de transformation, déjà observé sur P2SC.

Code	Version initiale	Version équilibrée	Version déséquilibrée
APPLU	65.60s	64.16s	69.88s
ONDE24 (MODELE 4)	72.40s	62.17s	65.09s
ONDE24 (MODELE 5)	619.34s	519.11s	609.05s
CX3D (-O3)	433.64s	437.98s	437.83s

TAB. 10.3 – *Résultats sur UltraSparc IIi 270MHz.*

Notons qu'il serait intéressant d'effectuer d'autres expériences sur des processeurs tels que le PA-RISC 8000 ou le processeur DSP Texas 6700 qui présentent des capacités importantes en terme de parallélisme d'instructions et supportent les instructions *fma*. Les perspectives de gain en performance sont importantes, mais nous ne disposons malheureusement pas de ces processeurs.

10.3 Conclusion

Les différentes techniques présentées dans cette partie ont été implémentées sous la forme d'un prototype, appelé `EOLE`, permettant d'appliquer automatiquement des transformations algébriques. La phase de normalisation ainsi que les heuristiques de factorisation et d'extraction des *fma* ont été implémentées. L'algorithme d'équilibrage des expressions est également disponible. Des expériences réalisées à partir de plusieurs applications écrites en Fortran 77, sur trois architectures différentes, ont montré l'impact significatif des transformations algébriques. Les techniques présentées, utiles sur les processeurs superscalaires largement utilisés actuellement, pourraient être également exploitées dans le cadre de l'optimisation sur processeurs DSP.

Quatrième partie

Transformations algébriques et optimisations de boucles

Chapitre 11

Motivations

L'optimisation de programmes est un vaste domaine étudié depuis des années. De nombreuses techniques permettant de considérablement améliorer les performances des programmes sont disponibles aujourd'hui. Parmi les plus courantes se trouvent notamment *l'élimination de code mort, le déplacement de code invariant, l'élimination de sous-expressions communes, la réduction de force, la propagation de constantes*, etc. [ASU86]. Ces techniques, qui pour la plupart ont été introduites dans les années 70, sont partiellement implémentées dans la majorité des compilateurs commerciaux actuels. Cependant, en raison de la complexité de mise en œuvre de ces optimisations et de leur grande efficacité, d'autres travaux ont été menés dans la continuité, avec un regain d'intérêt notable ces dernières années. Un nombre important d'articles concernant le déplacement de code invariant a par exemple été publié en 1998 [Rut98b, KRS98, Rut98a, BGS98, PSS98, Agr98].

Les travaux présentés ici se concentrent sur l'exploitation des propriétés algébriques pour créer de nouvelles opportunités d'optimisation. Cette partie repose principalement sur l'étude du déplacement de code invariant (chapitre 12, 13 et 14). Des travaux concernant l'élimination de sous-expressions communes sont également présentés au chapitre 15. D'autres opportunités d'optimisations seront évoquées au chapitre 18. Quelques résultats ainsi que des informations sur l'implémentation de ces travaux dans l'outil d'analyses et de transformations automatiques `PIPS` (voir annexe B) sont fournis au chapitre 16.

Les travaux présentés sont particulièrement motivés par les deux types de problèmes que sont le calcul scientifique et la génération automatique de code. Afin de mieux illustrer l'intérêt de nos travaux, un extrait du programme Onde24 sera également étudié dans ce chapitre introductif.

11.1 Optimisation d'applications de calcul scientifique

La tendance actuelle est à l'utilisation de langages de plus en plus haut-niveau, permettant au programmeur de s'abstraire des spécificités de la machine utilisée pour exécuter le code. De plus, dans le domaine du calcul scientifique, le programmeur écrit son code en ayant pour base une théorie mathématique ou physique. Or, la transcription de la formule mathématique en expression évaluable par l'ordinateur n'est pas toujours faite avec des objectifs de performance. Le scientifique souhaite en effet conserver la lisibilité de son programme, et n'est pas forcément averti de l'importance que peut prendre la manière qu'il a de transcrire son algorithme.

L'utilisation des propriétés algébriques et plus particulièrement la commutativité et l'associativité des opérateurs est insuffisamment exploitée. Elle permettrait en effet dans bien des cas de transformer des formules mathématiques *lisibles* en formules mathématiques *évaluées efficacement*.

En outre, de nombreux codes ayant été optimisés « à la main » pour une architecture donnée (à l'époque des machines vectorielles notamment) offrent de bien mauvaises performances pour les processeurs modernes. Il est donc souvent nécessaire aujourd'hui de défaire ces optimisations, au moyen de techniques telles que la *forward substitution*[41]. Les nouveaux codes ainsi construits contiennent à leur tour de nombreuses opportunités d'optimisation utilisant des transformations algébriques.

41. Cf. chapitre 8

11.2 Optimisation de codes générés automatiquement

La parallélisation automatique de programmes scientifiques cherche à construire à partir d'un code séquentiel une nouvelle version exécutable en parallèle. Une des techniques utilisées consiste à générer un nouveau programme reprenant le contenu du code original et incluant de nouvelles portions de programme dont la tâche principale est de partager les données et de gérer les communications entre les processeurs. De nombreux compilateurs HPF fonctionnent sur ce principe [Bra94, Coe96]. L'énumération de polyèdres utilisée pour décrire les portions de données impliquées dans une communication est fréquemment utilisée. Or, la performance du code parallèle peut être considérablement dégradée si le code généré automatiquement n'est pas performant.

La qualité du code généré automatiquement est un objectif décrit dans de nombreux articles récents. Certains comme [AI91, Cha93c, Cha93b, Cha93a, LP94, Xue94, CL96, DR96, BF98] tentent d'améliorer les techniques « en amont » par l'utilisation de nouvelles techniques de génération (des boucles notamment). D'autres, « en aval » essaient d'appliquer des transformations de programmes permettant d'améliorer le code considéré [KK94, BEF$^+$96, MCA98, KS98]. L'approche « amont » permet d'améliorer la qualité du code généré dans le cas général (i.e. quels que soient le contexte, l'instanciation des paramètres de l'algorithme, etc...). L'approche « aval » traitera à l'aide d'une large palette d'outils une instance donnée de code généré. Le traitement d'un code « non générique » autorise l'utilisation d'optimisations plus complexes. Nous nous plaçons ici dans le contexte des optimisations en « aval ».

11.3 Un exemple extrait de ONDE24

La figure 11.1 présente une boucle extraite de l'application Onde24 (une vue globale de cette application est donnée au chapitre 2). Cette boucle ne constitue pas le point « sensible » de l'application, mais la même structure se retrouve à plusieurs reprises dans le code. Le nombre d'opérations nécessaires pour exécuter les $(NP-3)$ itérations de la boucle (rappelons que NP est grand puisqu'il s'agit de la taille du domaine traité) est donné sur la partie droite (b) de la même figure. L'auteur de ce code a manifestement tenté d'améliorer les performances et/ou la lisibilité du programme en extrayant la sous-expression commune V(I,NP) × DELTAT[42]. Cependant, ce nid de boucle présente de nombreuses autres opportunités d'optimisation, à condition d'utiliser pour cela les propriétés algébriques des opérateurs manipulés.

```
DO I = 2,NP-2
  V(I,NP)  = 2/( (1/V(I,NP)) + (1/V(I,NP-1)) )
  W        = V(I,NP)*DELTAT
  B(I,1,1) = 1 - (1/(W+H)) * ( 2*H + W*W/H)
  B(I,1,2) = 2*H/(W+H)
  B(I,1,3) = W*W/(2*H*(W+H))
ENDDO
```

(a) Code initial.

	$\times 1$	$\times(NP-3)$
additions	-	5
multiplications	-	8
divisions	-	7

(b) Nombre d'opérations.

FIG. 11.1 – *Opportunités d'optimisations de boucles dans ONDE24. (1/2)*

Une version optimisée de cette boucle extraite de ONDE24 est présentée figure 11.2. Les techniques utilisées sont tout à fait classiques, mais les opportunités d'optimisation n'apparaissent, pour la plupart, qu'avec l'utilisation de l'associativité et de la commutativité. Un premier travail de *forward substitution* est nécessaire pour propager l'expression V(I,NP) × DELTAT vers ses utilisations. Cette transformation, qui augmente dans un premier temps le nombre d'opérations, est indispensable pour exploiter tout le potentiel d'optimisation. Une phase d'élimination de code mort, fonctionnant par analyse des *use-defs* [ASU86], sur les permet de supprimer aisément l'instruction d'affectation à W, devenue inutile. Le déplacement de

42. Notons que la valeur affectée à la variable temporaire W est inutilisée à l'extérieur de la boucle

code invariant, puis l'élimination de sous-expressions communes permettent d'aboutir au code optimisé. Le nombre d'opérations calculées pendant l'exécution de ce fragment de code est réduit de manière significative, dès lors que le paramètre NP est important (ce qui est le cas en pratique). Le gain en performance obtenu pour cet extrait de programme sur plusieurs architectures avec différents compilateurs est présenté dans le tableau 16.2. Les transformations de programme appliquées pour obtenir la version optimisée du code ont été implémentées (voir le chapitre 16).

```
T1 = DELTAT*DELTAT
T2 = 2*H
T3 = T1/H
DO I = 2,NP-2
  V(I,NP)   = 2/( (1/V(I,NP)) + (1/V(I,NP-1)) )
  C1 = 1/(V(I,NP)*DELTAT+H)
  C2 = V(I,NP)*V(I,NP)*C1*T3
  C3 = T2*C1
  B(I,1,1) = 1 - (C3-C2)
  B(I,1,2) = C3
  B(I,1,3) = C2*0.5
ENDDO
```

(a) Code transformé manuellement.

	$\times 1$	$\times(NP-3)$
additions	-	4
multiplications	2	7
divisions	1	4

(b) Nombre d'opérations.

FIG. 11.2 – *Opportunités d'optimisations de boucles dans ONDE24. (2/2)*

Processeur	Compilateur f77	Options de compilation	Gain	Temps (sec) Initial/Optimisé
P2SC 160 MHz	xlf release 4.1	-O3 -qhot -qarch=pwr2	2.88	0.49/0.17
UltraSparc-IIi 270 MHz	SUN WorkShop 5.0	-O3	1.85	0.76/0.41
		-fast	3.45	0.69/0.20
		-fast -O5	3.45	0.69/0.20
PentiumII 300MHz	GNU g77 v0.5.24	-O3	2.43	1.19/0.49
		-O5	2.43	1.19/0.49
R10K 175 MHz	MIPSpro release 7.1	-O3	2.40	1.44/0.60

TAB. 11.1 – *Gains obtenus pour l'optimisation de boucles sur ONDE24.*

11.4 À propos de l'organisation de cette partie

Les travaux présentés tout au long de cette partie proposent des transformations de programmes ayant recours, dès que possible, aux propriétés algébriques des opérateurs. Notre étude se concentre principalement sur le déplacement de code invariant, mais les techniques sont également introduites pour d'autres optimisations. L'organisation de cette partie est donc la suivante.

Le chapitre 12 introduit les deux chapitres suivants en offrant un aperçu des opportunités de déplacement de code recherchées. Ce chapitre préliminaire permet également de mieux situer nos travaux dans ce vaste domaine. Le chapitre 13 introduit une nouvelle technique de déplacement d'instructions invariantes, tandis que le chapitre 14 présente notre travail de réassociation des expressions dans le but de construire des sous-expressions invariantes.

Le chapitre 15 se concentre sur la recherche de sous-expressions communes dans un contexte associatif-commutatif. Un premier travail d'étude de complexité est suivi de la mise en place d'une nouvelle heuristique.

Les résultats expérimentaux obtenus ainsi que l'implémentation dans l'outil `PIPS` sont présentés à travers le chapitre 16.

Une étude des interactions concernant les optimisations de boucles et d'autres transformations est également présentée au chapitre 17.

Chapitre 12

Techniques de déplacement de code invariant

Ce chapitre sert d'introduction aux chapitres 13 et 14. En effet, les techniques présentées dans ces derniers, à savoir le déplacement d'instructions invariantes et le découpage en sous-expressions invariantes, concernent la même optimisation, mais peuvent être étudiées et appliquées de manière complètement indépendante. Elles feront donc l'objet de deux chapitres distincts. Cependant, un certain nombre d'informations communes aux deux techniques seront regroupées à travers ce chapitre. Ainsi, après un aperçu des opportunités de déplacement de code invariant donné à la section 12.1, une discussion sur le positionnement de nos travaux dans ce domaine sera menée à la section 12.2. Enfin, quelques rappels et notations utiles à la bonne compréhension des deux chapitres suivants seront donnés aux sections 12.3 et 12.4.

12.1 Introduction

Le déplacement de code invariant est une optimisation puissante qui a prouvé depuis longtemps son efficacité. Cette technique permet de déplacer en dehors d'une boucle l'évaluation d'une expression qui prend la même valeur pour toutes les itérations de la boucle. La valeur de l'expression est toujours référencée dans le corps de la boucle, mais n'est plus (re)calculée inutilement. Cependant, la présence d'expressions arithmétiques de taille importante rend parfois difficile la détection des calculs invariants. Un travail de réassociation des sous-expressions est dans ce cas nécessaire. Les techniques proposées dans la littérature reposent sur une représentation interne du programme sous forme d'un code 3-adresses. Or, l'atomisation des expressions nécessaire pour traduire le code source dans cette représentation ne tient pas compte des opportunités offertes par l'associativité et la commutativité des opérateurs.

L'objet de nos travaux est de fournir des techniques permettant d'exploiter les propriétés algébriques des opérateurs pour améliorer l'efficacité du déplacement de code invariant. Considérons l'exemple présenté figure 12.1. Cet extrait de programme a été généré automatiquement par un algorithme de parcours de tuile [AI91, AI92]. De nombreuses expressions similaires à celle constituant l'instruction S[43] apparaissent dans ce code. La variable O_2 reçoit le résultat de l'évaluation d'un minimum sur trois valeurs. Plusieurs transformations sont possibles pour améliorer les performances de ce code. Tout d'abord, les techniques présentées au chapitre 9 permettent de factoriser la partie commune à deux des trois membres du terme min (ce qui autorise l'évaluation d'une partie du min dans la boucle L_J). Le résultat obtenu est présenté figure 12.2.

L'expression factorisée comporte des opérations pouvant être évaluées à l'extérieur de la boucle la plus interne. Cependant, l'ordre d'apparition des termes impliqués dans les différentes additions ne permet pas d'extraire immédiatement tous les calculs potentiellement déplaçables. Une atomisation classique de type *associative-gauche*[44] ne permet pas de remonter au mieux toutes les additions. En effet, comme le montre

43. Nous utilisons la lettre S pour nommer une instruction afin de rester homogène avec le *statement* en anglais
44. L'expression $x = a + b + c$ sera évaluée comme $x = ((a + b) + c)$ ou encore $t = a + b$ et $x = t + c$

```
        Do I0 = ...
          Do J0 = ...
            Do LJ = ...
              Do L = ...
(S)             O2 = Min(99, 9924 - 100*BANK_ID - 400*L + 25*I0,
   &                         24 - 100*BANK_ID + 2500*J0
   &                         - 400*L + 25*I0 + 100*LJ)
```

FIG. 12.1 – *Découpage en sous-expressions : code initial.*

```
        Do I0 = ...
          Do J0 = ...
            Do LJ = ...
              Do L = ...
(S)             O2 = Min(99, -100*BANK_ID - 400*L + 25*I0 +
   &                       Min(9924,24+ 2500*J0 + 100*LJ))
```

FIG. 12.2 – *Découpage en sous-expressions : code factorisé.*

la figure 12.3, plusieurs additions sont évaluées dans la boucle la plus interne, en raison de la présence du terme $400*L$. Notons qu'une atomisation *associative-droite* ne donnerait pas de meilleurs résultats.

Le résultat obtenu en effectuant une réassociation *réfléchie* des termes additifs est donné figure 12.4. Cette transformation permet d'éviter l'évaluation de $2n^4 - (n^3 + n)$ additions (cf. tableau 12.1). Pour aboutir à ce résultat, il est nécessaire de connaître la position dans le nid de boucle à partir de laquelle chacun des termes de l'addition peut être évalué. Concernant cet exemple, l'analyse de la présence des variables inductives I_0, J_0, L_J et L s'avérerait suffisante. Néanmoins, dans le cas général, une analyse plus fine des dépendances portées par les instructions est obligatoire. En effet, la présence combinée d'accès à des éléments de tableaux et l'utilisation de variables scalaires dans le cadre de réductions rend plus complexes les décisions de réassociation.

```
t1 = -100*BANK_ID
Do I0 = ...
   t4 = 25*I0
   Do J0 = ...
      t6 = 2500*J0
      t7 = 24 + t6
      Do LJ = ...
          t8 = 100*LJ
          t9 = t7 + t8
          t10 = Min(9924,t9)
          Do L = ...
             t2 = -400*L
             t3 = t1 + t2
             t5 = t4 + t3
             t11 = t5 + t10
             O2 =  Min(99,t11)
```

FIG. 12.3 – *Déplacement de code avec atomisation associative-gauche.*

L'objectif de notre approche est donc d'analyser les expressions arithmétiques présentes dans un nid de

```
t_1 = -100*BANK_ID
Do I_0 = ...
   t_2 = t_1 + 25*I_0
   Do J_0 = ...
      t_3 = 24 + 2500*J_0
      Do L_J = ...
          t_4 = t_2 + Min(9924, t_3 + 100*L_J)
          Do L = ...
             t_5 = t_4 + 400*L
             O_2 = Min(99,t_5)
```

FIG. 12.4 – *Déplacement de code avec réassociation en sous-expressions invariantes.*

	I_0 (n)	J_0 (n^2)	L_J (n^3)	L (n^4)
associatif-gauche (12.3)	-	1	1	3
réassociation (12.4)	1	1	2	1

TAB. 12.1 – *Position des additions dans les boucles*

boucles pour tenter d'en extraire des sous-expressions invariantes pouvant être déplacées et donc évaluées en dehors des boucles. Pour offrir de réelles opportunités, ce type de transformation nécessite évidemment l'utilisation de propriétés algébriques telles que l'associativité et la commutativité des opérateurs. D'autres propriétés telles que la distributivité seront également utilisées. Les principales opérations concernées sont l'addition, la multiplication, le minimum, le maximum, ainsi que les opérateurs logiques (AND, OR). Les opérateurs unaires de type « complément » (opposé, inverse, NOT, etc.) seront manipulés selon les principes exposés au chapitre 8.

Pour que le travail de réassociation soit véritablement efficace, il est nécessaire que chacune des instructions présentes dans le nid de boucles soit d'ores et déjà placée dans la boucle la plus externe possible. Cette condition (sur laquelle nous reviendrons au chapitre 14) est illustrée par la figure 12.5. Avant de pouvoir déclarer le calcul A(i_2)*N déplaçable dans la boucle d'indice i_2, il est nécessaire de déterminer que la valeur A(i_2) est disponible dans cette boucle. Pour cela, l'instruction S_1 doit être déplacée dans la boucle d'indice i_2. Notons ici que la valeur B(i_2) ne sera pas disponible dans la boucle i_2 car l'instruction S_2 ne peut être déplacée (c'est une réduction). Pour que la réassociation soit efficace, nous avons donc également besoin « en amont » d'un algorithme de déplacement d'**instructions** invariantes (contenu du chapitre 13).

```
(S_3)  N = ...
       Do i_2 = ...
          Do i_1 = ...
(S_1)         A(i_2) = f(i_2)
(S_2)         B(i_2) = B(i_2) + 1
(S_3)         C(i_1, i_2) = A(i_2) * B(i_2) * i_1 * N
          Enddo
       Enddo
```

FIG. 12.5 – *De l'importance de la position des instructions.*

12.2 Positionnement de nos travaux

Étant donnée l'étendue des travaux existant dans le domaine du déplacement de code invariant, il est nécessaire de préciser d'ores et déjà le positionnement de nos propres travaux. Après un rapide aperçu de l'évolution des techniques utilisées au cours du temps, nous nous pencherons plus particulièrement sur les caractéristiques qui motivent nos travaux et les distinguent des autres.

12.2.1 État de l'art

Dès 1965, Gear introduit dans [Gea65] les optimisations fondamentales sur les boucles telles que le déplacement de code à l'extérieur des boucles. Plusieurs articles fondamentaux comme ceux d'Allen [All69] ou Waite [Wai74] apporteront ensuite une étude plus exhaustive des techniques de ce domaine.

Morel et Renvoise dans [MR79] proposent un algorithme d'élimination des redondances partielles[45]. Cette technique permet à la fois de supprimer des boucles les calculs redondants et d'extraire les calculs invariants hors des boucles. L'algorithme est fondé sur une analyse bidirectionnelle des flots de données[46] et fonctionne par résolution d'équations booléennes. Aucune modification de la structure du code n'est effectuée, et l'approche est de type heuristique.

Ferrante et Ottenstein [FO83] proposent un algorithme de déplacement de code s'appuyant sur une représentation du programme appelée *extended data flow graph*. Cytron, Lowry et Zadeck présentent dans [CLZ86] plusieurs algorithmes permettant un déplacement de code invariant plus agressif. Ils proposent en particulier le déplacement de structures de contrôle et autorisent des optimisations non conservatives. Ces techniques seront réutilisées dans le cadre de l'utilisation du *Program Dependance Graph* (PDG) par Ferrante et al. dans [FOW87].

Les heuristiques utilisées par Morel et Renvoise posent des problèmes d'efficacité en pratique. Dhamdhere dans [Dha91] ainsi que Drechsler et Stadel dans [DS88] signalent ces difficultés et proposent quelques alternatives. La solution viendra réellement de la technique présentée par Knoop, Rüthing et Steffen dans [SKR90] puis [SKR91, KRS92, KRS94]. Elle repose sur une analyse unidirectionnelle à partir du *value flow graph*. L'algorithme est optimal en terme de calculs et durée de vie des registres (pas de déplacement de code « inutile »).

Un nouvel algorithme optimal de PRE permettant de travailler directement sur une représentation interne avec assignation unique[47] (la PRE devient ainsi plus homogène avec les autres optimisations travaillant déjà sous SSA) est proposé par Chow et al. dans [CCK$^+$97]. Par ailleurs, une approche interprocédurale de la PRE, motivée par la génération de code pour machines à mémoire distribuée, est abordée dans [ASD95, Agr98]. Enfin, constatant l'échec des techniques de PRE classiques face aux contraintes imposées par le flot de contrôle, Bodik, Gupta et Soffa ont récemment proposé dans [BGS98] une approche combinée de PRE et de restructuration du contrôle.

12.2.2 Notre approche

Plusieurs caractéristiques distinguent nos travaux de l'approche qui a été menée jusqu'à présent en PRE. Nous nous plaçons en effet dans un contexte particulier, pour lequel certaines contraintes vont disparaître tandis que d'autres vont au contraire venir s'ajouter.

Un travail axé sur les boucles

Les travaux menés dans le domaine de la PRE fonctionnent sur un principe de *profitabilité* très stricte. La transformation appliquée ne doit en aucun cas ajouter un calcul sur un chemin du graphe de flot de contrôle. Cette condition garantit qu'à l'exécution le programme transformé n'effectue pas plus d'opérations que le programme initial, ceci quel que soit le contexte d'exécution. Le déplacement des calculs en dehors des boucles n'est donc pas toujours autorisé (en particulier lorsque plusieurs boucles sont imbriquées). Considérons l'exemple de la figure 12.6. Les calculs $t = a + b$ et $h = c + d$ peuvent être

45. Partial Redundancy Elimination (PRE)
46. bidirectional data flow analysis
47. static single assignment form (SSA)

extraits du cycle d'exécution représenté en pointillés. En revanche, seul le calcul $t = a + b$ est extrait du second cycle et placé dans le nœud 1 ; le calcul $h = c + d$ n'est pas déplacé car il n'apparaît pas sur le chemin passant par les nœuds 1 et 4.

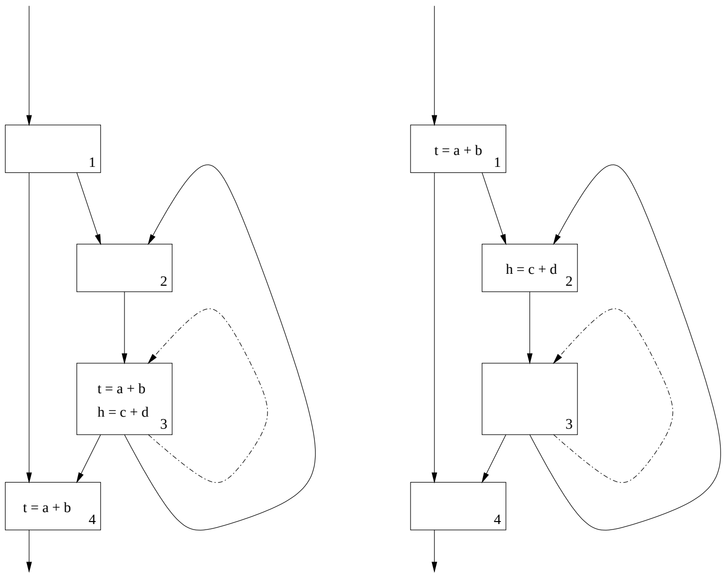


FIG. 12.6 – *Exemple d'application de la PRE.*

Click propose dans [Cli95] une approche différente du déplacement de code invariant et s'éloigne ainsi de la PRE. Son algorithme ne respecte pas le critère d'optimalité utilisé dans [KRS94] car il autorise l'ajout de calculs sur certains chemins. L'objectif de son travail est en effet de concentrer le travail d'optimisation sur les boucles, au risque de perdre en efficacité dans certains cas (rares en pratique). Ce travail se rapproche en cela plus d'articles anciens tels que [FO83] ou [CLZ86]. Horspool et Ho s'appuient dans [HH97] sur une estimation de la fréquence d'exécution des opérations pour appliquer une PRE fondée sur l'analyse d'une fonction de coût.

Nous avons également choisi de nous recentrer sur un problème crucial du calcul scientifique, à savoir l'optimisation des boucles. Contrairement aux travaux de PRE visant à définir des techniques applicables à un graphe quelconque, nous nous concentrons ici sur des programmes bien structurés contenant simplement des boucles, des tests et des séquences d'instructions. Les transformations plus agressives sont justifiées par le contexte favorable dans lequel nous nous plaçons. Le déplacement de code devient profitable dès lors que le corps de la boucle considérée est exécuté au moins une fois. Cependant, nous proposons un mécanisme permettant de préserver le comportement du programme ; la discussion concernant l'agressivité des transformations sera reprise à la section 13.4.

Déplacement d'instructions invariantes.

L'algorithme présenté au chapitre 13 se place au niveau des **instructions** et non au niveau des **expressions**. Il se distingue en cela de la majorité des travaux de PRE. L'extraction des sous-expressions invariantes pour lesquelles toute l'instruction ne peut être déplacée d'un bloc sera effectuée par la technique de réassociation présentée au chapitre 14.

Le déplacement d'instructions invariantes s'approche en fait plus des techniques de déplacement des assignations ([Dha88, KRS95]). En revanche, contrairement à ces dernières qui se contentent de déplacer des affectation de la forme $a = b + c$ (sous forme de code 3-adressses), nous désirons pouvoir déplacer des instructions contenant des accès à des éléments de tableaux ou des appels de fonctions. A notre connaissance, il n'existe aucun algorithme permettant ces transformations et nous avons donc mis en place une nouvelle technique de déplacement d'instructions invariantes. Notre travail se situant à un haut niveau, le déplacement d'instructions invariantes (chapitre 13) et les techniques de réassociation (chapitre 14) gèrent directement les accès à des éléments de tableaux. Cette fonctionnalité suppose l'utilisation d'une analyse fine concernant les relations définition-utilisation, à l'aide de l'analyse de dépendances ou des effets (voir chapitre 5) par exemple.

Vers de nouvelles opportunités d'optimisation

La majorité des articles publiés dans ce domaine repose sur une représentation interne du code sous forme de code 3-adresses. Seules les expressions élémentaires de la forme $a+b$ sont considérées. Quelques propositions axées sur des expressions de taille plus importante existent cependant. L'apport des propriétés algébriques des opérateurs est introduit dans un premier temps dans [Fra70]. L'auteur montre qu'un grand nombre de transformations élémentaires peuvent faciliter l'évaluation des expressions. Scarborough et Kolsky décrivent dans [SK80] une technique permettant de générer les expressions impliquées dans les calculs d'adresse (accès aux éléments de tableaux), sous forme d'une somme de produits. Les termes des sommes sont ensuite réassociés pour favoriser le déplacement de code invariant. Cocke et Markstein mentionnent également l'idée de réassociation dans [CM80]. Les auteurs de [FO83] évoquent dans leur algorithme la possibilité de réassocier les expressions en utilisant l'associativité des opérateurs. Plus tard, Briggs et Cooper recherchent dans [BC94] de nouvelles opportunités de déplacement de code. Ils essaient de reconstruire des sous-expressions «déplaçables» à partir du code 3-adresses. Leurs travaux étendent les résultats de [SK80]. Rüthing se penche quant à lui dans [Rut98b] sur le comportement de techniques de [KRS94] vis à vis de la pression de registres en présence de « grandes » expressions.

Les travaux de Briggs et Cooper constituent la référence actuelle dans le domaine de la réassociation. Après un exposé de la technique proposée, et un aperçu des résultats expérimentaux, les auteurs discutent également les limites de leur approche. Leur travail s'insère dans le cadre du développement d'un optimiseur utilisant une représentation intermédiaire de bas niveau (ILOC [BCT87]) avec des opérations sous forme de code 3-adresses. Pour revenir à des opérations n-aires, la technique proposée consiste à utiliser une phase de *forward propagation*. Cette méthode présente plusieurs inconvénients. Tout d'abord, certains calculs introduits dans les boucles par la *forward propagation* ne sont pas (re)extraits des boucles par les techniques de PRE. De plus, l'expansion de code introduite par la *forward propagation* risque de dégrader considérablement les performances.

L'approche que nous proposons à travers nos travaux (chapitre 14) vise à préserver la structure des expressions complexes (sous forme d'un arbre n-aire de profondeur quelconque). En effet, le passage à une représentation sous forme de code 3-adresses est un handicap pour l'exploitation des propriétés algébriques des opérateurs ; la reconstruction des opérations n-aires a montré ses limites dans [BC94]. D'autre part, en raison des interactions existant avec les autres optimisations, il est préférable de conserver le plus longtemps possible un degré de liberté maximum. Ainsi, il est par exemple préférable d'appliquer l'équilibrage de l'arbre (présenté au chapitre 9) après le déplacement de code invariant (dont le gain potentiel est plus important). Or, un passage par une forme atomisée compromettrait fortement les possibilités d'équilibrage de l'arbre. Les travaux de Briggs et Cooper utilisent la notion de rang. Le premier bloc prend le rang 1, le rang du second bloc est 2, etc. Cette définition ne peut en fait fonctionner que dans le cadre de boucles parfaitement imbriquées. Nous étendons cette idée en introduisant la notion de "disponibilité", qui permettra de traiter une imbrication de boucles quelconque.

La décomposition d'expressions de taille importante en sous-expressions telle que nous l'envisageons pour le déplacement de code invariant a également servi de base à quelques travaux dans le domaine de la vectorisation. En effet, des instructions non vectorisables dans leur globalité peuvent contenir des sous-expressions exploitables. Quelques techniques abordent le problème en passant par une représentation sous forme de code 3-adresses [Ban79, Wol90]. L'introduction d'un grand nombre de variables temporaires pose cependant des problèmes pour les temps de traitement des phases successives de vectorisation (une phase d'expansion de scalaires pour réduire le nombre de dépendances est notamment requise). Dehbonei dans [Deh91] propose une approche de « haut niveau », fondée sur la construction de sous-expressions *k-vectorisables* (vectorisables pour la boucle de profondeur k). Ces différentes techniques n'exploitent cependant pas les propriétés algébriques des opérateurs mises en jeu.

Une transformation de haut niveau

Les techniques de réassociation des expressions présentées dans cette thèse reposent sur une structure de données permettant de manipuler des opérateurs n-aires, associatifs-commutatifs. Elles peuvent indifféremment être appliquées directement dans l'optimiseur, ou sous la forme d'une passe préliminaire (*front-end*). Notre choix a été de mettre en place des transformations source-à-source, et ce pour les raisons suivantes :

1. La construction de sous-expressions invariantes n'améliore pas directement les performances du programme. Elle permet cependant d'augmenter l'efficacité du déplacement de code appliqué par la suite. Travailler source-à-source permet de bien séparer la réassociation de la technique de déplacement utilisée. La transformation reste donc valide quel que soit le compilateur choisi ensuite.
2. En ce qui concerne les codes générés automatiquement, il nous a paru intéressant de fournir une transformation source-à-source utilisable directement sur la sortie du générateur de code. De nombreux compilateurs génèrent un code source (Fortran par exemple) qui peut être indifféremment compilé sur plusieurs architectures. Seule une partie du code (la librairie de communication par exemple) sera spécifique à une architecture donnée. Ainsi, il est important de pouvoir améliorer la qualité du code généré pour tous les types d'architectures en une seule fois. Par exemple, le code de la figure 12.1 qui est généré par un compilateur HPF sera optimisé une fois pour toutes, quelle que soit l'architecture (parallèle) sur laquelle sera ensuite compilé le code Fortran (SPMD).
3. Une des principales qualités des transformations source-à-source provient de son caractère « interactif ». En effet, ce type d'approche permet au programmeur de contrôler les modifications apportées à son code, tout en conservant une version originale intacte (ce qui n'est pas le cas lorsque les optimisations sont intégrées directement au compilateur). Ainsi, le programmeur peut appliquer automatiquement une transformation et vérifier que la nouvelle version du programme respecte bien certains critères (dans le cadre de la stabilité numérique notamment).
4. Enfin, les transformations source-à-source sont les mieux adaptées dans l'optique d'un travail d'expérimentation et de prototypage.

12.3 Quelques rappels

Les techniques présentées aux chapitres 13 et 14 manipulent directement des accès à des éléments de tableaux. Cette fonctionnalité est rendue possible par l'utilisation de techniques qui sont rappelées ci-dessous.

12.3.1 Analyse de dépendances

Un programme contient des instructions qui lisent et/ou écrivent dans des scalaires ou des tableaux. Une dépendance est une relation entre deux instructions qui impose une contrainte sur leur ordre d'exécution. L'analyse de dépendances sera donc utile pour contrôler si une transformation particulière peut être appliquée sans changer la sémantique des calculs [Ber66, Ban88].

Une même instruction peut être exécutée plusieurs fois lorsqu'elle est contenue dans une ou plusieurs boucles. Dans ce cas, chaque exécution de l'instruction sera appelée instance d'instruction.

Les boucles sont supposées normalisées, c'est-à-dire avec un pas et une borne inférieure valant 1, la borne supérieure étant fixée arbitrairement à N. Les auteurs de [ZC91] ou [Wol96] rappellent la technique à utiliser pour normaliser une boucle.

Dépendance de contrôle

Il y a une **dépendance de contrôle** entre une condition C et une instruction S lorsque C détermine si S sera ou non exécutée. Ces informations peuvent être synthétisées dans un graphe appelé *graphe de flot de contrôle*. Étant donné que nos travaux sont axés sur les nids de boucles contenant des séquences d'instructions et des tests, les informations de type contrôle que nous aurons à manipuler sont très simples.

Dépendance de données

Il y a une **dépendance de données** entre une instruction S_1 et une instruction S_2 lorsqu'il y a utilisation conflictuelle d'une même variable. Pour toute instruction S, on définit $\mathcal{M}(S)$ (resp. $\mathcal{L}(S)$) l'ensemble des cellules mémoires modifiées (resp. lues) par l'instruction S.

Soient S_1 et S_2 deux instructions telles que S_1 s'exécute avant S_2 dans le programme séquentiel. Trois sortes de dépendances sont en général utilisées[48] :

- Il existe une **dépendance de flot**[49] de S_1 vers S_2 si S_2 accède en lecture à un emplacement mémoire qui a été modifié par S_1. Elle sera notée $S_1 \xrightarrow{\delta} S_2$.
  $\mathcal{M}(S_1) \cap \mathcal{L}(S_2) \neq \emptyset$
- Il existe une **anti-dépendance**[50] de S_1 vers S_2 si S_2 accède en écriture à un emplacement mémoire qui est lu par S_1. Elle sera notée $S_1 \xrightarrow{\overline{\delta}} S_2$.
  $\mathcal{L}(S_1) \cap \mathcal{M}(S_2) \neq \emptyset$
- Il existe une **dépendance de sortie**[51] de S_1 vers S_2 si S_2 modifie un emplacement mémoire précédemment modifié par S_1. Elle sera notée $S_1 \xrightarrow{\delta_o} S_2$.
  $\mathcal{M}(S_1) \cap \mathcal{M}(S_2) \neq \emptyset$

Remarque 12.1
Dans de nombreux travaux effectués sur l'analyse de dépendances, on considère que la seule véritable dépendance est la dépendance de flot (appelée souvent **dépendance vraie***). Les autres dépendances ne sont que les conséquences de la ré-utilisation d'une zone de mémoire, et sont communément appelées pseudo-dépendances.*

Soient deux instructions S_1 et S_2 apparaissant dans la même boucle (ayant pour indice i). Nous dirons que $S_2 \succ S_1$ si S_1 apparaît le premier dans la boucle et si $S_1 \neq S_2$. Soient $A(f(i))$ (resp. $B(g(i))$) une référence lue ou modifiée[52] par S_1 (resp. S_2) avec A (resp. B) une variable et $f(i)$ (resp. g(i)) une expression contenant l'indice i.

Définition 12.1
Il existe une **dépendance portée par la boucle**[53] *entre S_1 et S_2, si et seulement si il existe i_1 et i_2 tels que $1 \leq i_1 < i_2 \leq N$ et $f(i_1) = g(i_2)$.*

Définition 12.2
Il existe une **dépendance indépendante de la boucle**[54] *entre S_1 et S_2, si et seulement si il existe une itération i $(1 \leq i \leq N)$ telle que $S_2 \succ S_1$ et $f(i) = g(i)$.*

L'égalité $f(i_1) = g(i_2)$ est équivalente à un système contenant autant d'équations que les tableaux A et B ont de dimensions. Le test de dépendance revient donc à déterminer si un système d'équations à coefficients entiers (appelé *équations aux indices*) a des solutions entières. En plus des équations aux indices, il faut ajouter les contraintes de bornes de boucle ($1 \leq i_1 \leq N$ et $1 \leq i_2 \leq N$), la contrainte de différenciation d'itération ($i_1 \neq i_2$, en fait $i_1 < i_2$, par symétrie) et les prédicats vrais introduits par le reste du programme ; cela donne donc un système d'équations et d'inéquations à coefficients entiers.

48. Le cas $\mathcal{L}(S_1) \cap \mathcal{L}(S_2) \neq \emptyset$ appelé **input-dependence** n'est pas considéré comme une dépendance, car inverser l'ordre d'exécution de deux opérations différentes lisant la même cellule mémoire ne change pas le résultat du programme. Il peut par contre être utile dans le cadre de l'optimisation de la localité des calculs.
49. Flow-dependence
50. Anti-dependence
51. Output-dependence
52. Les références ne doivent pas être lues toutes les deux sinon on se ramène au cas d'une *input-dépendance.*
53. loop carried dependence
54. loop independent dependence

Le test utilisé pour la recherche des dépendances portées par les boucles se généralise au traitement d'un nid de boucles. La fonction d'accès $f(i)$ devient $f(x_1, x_2, \ldots, x_{n_1})$ où $(x_1, x_2, \ldots, x_{n_1})$ est un vecteur d'itération[55] (avec x_1 l'indice de la boucle la plus externe).

Définition 12.3
Il existe une dépendance portée par la boucle k entre les instructions S_1 et S_2 si il existe $(i_1, i_2, \ldots, i_{k-1})$, $(j_{k+1}, j_{k+2}, \ldots, j_{n_1})$, $(l_{k+1}, l_{k+2}, \ldots, l_{n_2})$ et deux entiers α_1 et α_2 sur les domaines suivants[56]
$1 \leq i_q \leq N_{q_1}, \forall q_1 \in [1, k[$
$1 \leq j_q \leq M_{q_2}, \forall q_2 \in]k, n_1]$
$1 \leq l_q \leq N_{q_3}, \forall q_3 \in]k, n_2]$
$1 \leq \alpha_1 < \alpha_2 \leq N_k$ *et tels que*

$$f(i_1, i_2, \ldots, i_{k-1}, \alpha_1, j_{k+1}, j_{k+2}, \ldots, j_{n_1}) = g(i_1, i_2, \ldots, i_{k-1}, \alpha_2, l_{k+1}, l_{k+2}, \ldots, l_{n_2})$$

L'efficacité des algorithmes de parallélisation automatique repose sur la précision du test de dépendances. Dans notre cas, une analyse de dépendance peu complexe sera suffisante. De nombreuses études traitent le problème spécifique du test de dépendances. On citera notamment le *Test de Fourier-Motzkin* ([DE73, Duf74]), le *Test du GCD*[57] ([Ban88]), le *Test de Banerjee* ([Ban88]), le *Test PIP* ([Fea88]) ou encore le *Test Omega* ([Pug90]). Pour plus de détails voir la thèse de Yi-Quing Yang [Yan93].

Le but de l'analyse de dépendances est donc de déterminer pour chaque instruction du programme les instructions avec lesquelles elle est en dépendance. La structure utilisée pour mémoriser l'ensemble des dépendances d'un programme est le **graphe de dépendances**.

Définition 12.4
Le **graphe de dépendances détaillé** *(noté GDD) d'un programme est un graphe orienté dont les sommets sont des* instances d'instructions *et les arcs sont des couples (x,y) tels que x et y sont en dépendances et $x \succ y$.*

Ce graphe est en pratique inutilisable en raison de sa taille trop importante et non bornée statiquement (chaque instruction nécessite autant de sommets dans le graphe qu'il y a d'itérations de boucles). Cette représentation trop coûteuse est donc remplacée par une (sur)approximation.

Définition 12.5
Le **graphe de dépendances résumé** *(noté GDR) d'un programme est un graphe orienté dont les sommets sont des* instructions *et les arcs représentent les relations de dépendances entre les instructions.*

Le GDR est donc une forme condensée du GDD. Il contient un sommet par instruction du programme initial et ses arcs sont étiquetés par des informations sur la nature de la dépendance. Plusieurs abstractions existent dans la littérature ; on retiendra essentiellement le *niveau de dépendance* [AK87], le *vecteur de direction de dépendances* [Wol90], ou encore le *cône de dépendances* [IT87].

Définition 12.6
On dira qu'il existe une dépendance de niveau k entre deux instructions s'il existe une dépendance portée par la boucle k. *Cette dépendance sera notée :*
- $S_1 \xrightarrow{\delta\ (k)} S_2$ *pour une dépendance de flot*
- $S_1 \xrightarrow{\overline{\delta}\ (k)} S_2$ *pour une anti-dépendance*
- $S_1 \xrightarrow{\delta_o\ (k)} S_2$ *pour une output-dépendance*

Une dépendance indépendante des boucles est dite de niveau infini (∞).

55. idem pour $g(i)$ transformé en $g(x_1, x_2, \ldots, x_{n_2})$.
56. Soit M_i la borne supérieure de la i^{eme} boucle entourant S_1 et N_i la borne supérieure de la i^{eme} boucle entourant S_2. $M_i = N_i$ pour $1 \leq i \leq n$ où n est le nombre de boucles communes entourant S_1 et S_2.
57. Greatest Common Divisor

Le graphe de dépendances que nous utiliserons par la suite est le graphe de dépendances résumé dont les arcs sont étiquetés par le niveau de dépendance. Lorsque deux références R et R' contenues respectivement dans deux instructions S et S' génèrent des dépendances pour n différents niveaux, celles ci seront regroupées sur le même arc du graphe de dépendances et notées $S \xrightarrow{\delta\ (k_1,k_2,\dots,k_n)} S'$.

Remarque 12.2
Dans le cadre de notre travail de réassociation des expressions, la simple connaissance des dépendances entre instructions est insuffisante. Il nous faut également connaître les références (de la forme $A(f(x_1,x_2,\dots,x_n))$) qui sont à l'origine de ces dépendances. L'information sera donc ajoutée si nécessaire à l'étiquette portée par les arcs du GDR (sous la forme $S_1 \xrightarrow{\delta\ (k)[A(f(x_1,\dots,x_{n_1}))-B(g(x_1,\dots,x_{n_2}))]} S_2$).

12.3.2 Techniques de parallélisation

Les travaux de déplacement de code invariant présentés chapitre 13 s'inspirent des techniques de vectorisation et parallélisation automatique de code. L'algorithme d'Allen et Kennedy [AK87] est en particulier exploité. Cet algorithme cherche à faire apparaître du parallélisme[58] en appliquant une transformation classique appelée *distribution de boucle*. Il repose sur l'analyse des composantes fortement connexes du graphe de dépendances.

Définition 12.7
Soit G=(V,E) un graphe orienté dont l'ensemble des sommets est V et l'ensemble des arcs est E. On appelle ***composante fortement connexe*** *de G, tout sous-ensemble maximal C de V ayant la propriété suivante : Si $v_1, v_2 \in C$ avec $v_1 \neq v_2$, alors il existe dans G un chemin de v_1 vers v_2.*

L'algorithme de Tarjan donne une méthode de calcul des composantes fortement connexes d'un graphe [Tar72].

Algorithme d'Allen et Kennedy

La stratégie employée consiste à appliquer une transformation sur la boucle la plus externe, puis à rappeler récursivement la méthode sur les boucles internes en faisant abstraction du travail déjà accompli. L'algorithme est fondé sur les deux principes suivants :

1. La boucle la plus externe est parallèle si elle ne contient aucune *dépendance portée par la boucle* (pas de dépendance de niveau 1).
2. Toutes les instances d'une instruction S_1 peuvent être exécutées avant n'importe quelle instance d'une instruction S_2 si le GDR ne contient pas de dépendance entre S_2 et S_1.

La première propriété permet de marquer une boucle comme séquentielle (DOSEQ) ou parallèle (DOALL). Le second principe permet d'affirmer que la détection du parallélisme peut être faite de manière indépendante dans chacune des composantes fortement connexes du GDR. Une version simplifiée de la méthode est donnée par l'algorithme 4. L'algorithme fonctionne par descente récursive le long des composantes fortement connexes du graphe ; le premier appel est ALLEN_KENNEDY(D,1) où D est le GDR pour le nid de boucle traité. Darte et Vivien ont montré dans [DV96] que l'algorithme d'Allen et Kennedy est optimal lorsque les dépendances sont représentées par les niveaux de dépendances. Aucun algorithme, le plus sophistiqué soit-il, ne peut détecter plus de parallélisme tant que seul le niveau de dépendance est connu.

D'autres algorithmes de parallélisation

Il existe d'autres algorithmes de parallélisation fonctionnant sur d'autres représentations des dépendances et appliquant d'autres transformations de programmes. Nous ne détaillerons pas ici ces techniques[59]. Le *niveau de dépendances* étant une abstraction suffisamment précise dans le cadre de nos transformations, nous utiliserons l'algorithme d'Allen et Kennedy [AK87] qui est optimal lorsque cette abstraction est utilisée [DV96].

58. Cet algorithme a été initialement utilisé pour vectoriser un nid de boucles.
59. [DV95] est un bon point d'entrée

Algorithme 4 Algorithme de parallélisation d'Allen et Kennedy

---

Fonction ALLEN-KENNEDY(D,k)
  $\triangleright$ *- D : le graphe de dépendances $D = (V, E)$*
  $\triangleright$ *- k : le niveau minimum pour les dépendances de D*
  Chercher l'ensemble $\{C_1, \ldots, C_m\}$ des composantes fortement connexes maximales sur D
  Construire D_Π le graphe de dépendances tel que C_i est un nœud unique (Π-block)
  Soit $\{\Pi_1, \ldots, \Pi_m\}$ les nœuds de D_Π dans l'ordre topologique
  Pour $i \leftarrow 1, m$ **Faire**
    Si Π_i contient au moins un arc **Alors**
      Soit $l = l_{min}(\Pi_i)$ $\triangleright$ *niveau minimum des dépendances de* Π_i
      Pour $t \leftarrow k, l-1$ **Faire**
        Générer l'entête de la boucle parallèle de profondeur t
      Fin Pour
      Générer la boucle séquentielle de profondeur l
      Soit D'_i la restriction de D aux dépendances internes à Π_i et de niveau $l+1$ ou plus
      ALLEN-KENNEDY(D'_i, $l+1$)
    Sinon $\triangleright$ Π_i *est réduit à une seule instruction - plus aucun arc*
      Pour $i \leftarrow 1, \rho(\Pi_i) - k$ **Faire** $\triangleright$ $\rho(\Pi_i)$ *est le nombre de boucles contenant* Π_i
        Générer l'entête de la boucle parallèle de profondeur $k+i$
      Fin Pour
      Générer le code de Π_i
    Fin Si
  Fin Pour
Fin

12.4 Notations utilisées

Voici quelques notations qui seront utilisées dans le chapitres 13 et 14. D'autres notations seront également introduites au fur et à mesure de l'avancement de l'exposé.

- D_L : Domaine d'itération de la boucle
- $\mathrm{Instr}(L)$: Ensemble des instructions contenues dans une boucle[60] L
- $\mathrm{Pred}(S)$: Ensemble des prédécesseurs d'une instruction S.

  $S' \in \mathrm{Pred}(S)$ si $S' \xrightarrow{\delta} S$
- $\mathrm{Pred}_G(S)$: Restriction de l'ensemble des prédécesseurs d'une instruction S aux nœuds du graphe G.

  $S' \in \mathrm{Pred}_G(S)$ si $\exists S' \in G$ tel que $S' \xrightarrow{\delta} S$
- $\mathrm{Succ}(S)$: Ensemble des successeurs d'un instruction S.

  $S' \in \mathrm{Succ}(S)$ si $S \xrightarrow{\delta} S'$
- $\mathrm{Ref}(S)$: Ensembles des références appartenant à une instruction S.
- $\mathrm{Ind}(S)$: Ensemble des indices présents syntaxiquement dans une instruction S.
- $\mathrm{Def}(R)$: Ensemble des définitions vivantes d'une référence R (cf remarque 12.2).

  $S' \in \mathrm{Def}(R)$ si $R \in \mathrm{Ref}(S)$, $R' \in \mathrm{Ref}(S')$ et $\exists k$ tel que $S' \xrightarrow{\delta\ (k)[R'-R]} S$.

Organisation

L'application de la *forward substitution* pour les travaux présentés dans la partie III et notre objectif d'optimisation des codes générés automatiquement supposent l'utilisation de techniques de déplacement de code invariant. Nous avons présenté à travers ce chapitre les caractéristiques qui distinguent notre approche des travaux déjà existant dans ce domaine. Les rappels et notations présentés dans ce chapitre seront utilisés dans les chapitres 13 et 14. Une méthode de déplacement d'instructions invariantes fondée sur l'analyse du graphe de dépendances sera présentée au chapitre 13. L'approche proposée permet de manipuler directement des accès aux éléments de tableaux ou des appels de fonctions et autorise le déplacement d'instructions non traitées par les techniques classiques de PRE, tout en préservant le comportement du programme. Le chapitre 14 présente une technique de découpage en sous-expressions invariantes consistant à réassocier différemment les opérandes des opérations associatives-commutatives. Ce travail peut être réalisé avec une imbrication de boucles quelconques et n'effectue pas d'atomisation inutile.

60. L pour *loop* en anglais

Chapitre 13

Déplacement d'instructions invariantes

Cet algorithme de déplacement d'instructions invariantes est fondé sur l'étude des propriétés d'invariance des instructions à partir d'informations extraites du graphe de dépendances. La première étape consiste à simplifier le graphe en supprimant des dépendances portées *inutilement* par des instructions invariantes (section 13.1). Cette transformation permet ensuite la réutilisation des techniques classiques de génération de code. Les boucles placées habituellement autour des instructions invariantes sont supprimées si cela est possible. La technique de génération de code présentée à la section 13.2 est inspirée des algorithmes de parallélisation d'Allen et Kennedy [AK87]. Elle fonctionne par descente récursive dans les composantes fortement connexes du graphe de dépendances, et utilise la distribution de boucles.

Le travail présenté peut être appliqué à un nid de boucles contenant des séquences d'instructions, éventuellement gardées par des tests. Cependant, afin de simplifier la présentation, la technique sera dans un premier temps expliquée pour le déplacement d'instructions de type affectation dans un nid de boucle sans tests. Les possibilités de déplacement des appels de fonctions seront évoquées à la section 13.3. Enfin, l'application des techniques en présence de tests ainsi que des discussions concernant la conservation du comportement du programme seront introduites à la section 13.4.

Les techniques présentés dans ce chapitre ont été implémentées dans l'outil d'analyses et de transformations `PIPS`. Les détails de l'implémentations seront exposés au chapitre 16.

13.1 Simplification du graphe de dépendances

La présence d'instructions invariantes dans un nid de boucle génère un grand nombre de dépendances pouvant être supprimées. En effet, l'analyse de dépendances présentée dans la littérature repose sur une recherche **syntaxique** des instructions accédant à une même adresse mémoire. La connaissance d'informations sémantiques complémentaires permet de relâcher les contraintes imposées sur l'ordonnancement des calculs par des dépendances pas toujours justifiées. Ainsi, certaines propriétés des instructions (dont l'invariance pour une boucle) permettent d'effectuer des simplifications sur le graphe de dépendances, laissant alors plus d'opportunités aux transformations de programme suivantes.

13.1.1 Définition de l'invariance d'une instruction

Soit une instruction S quelconque appartenant à une boucle L d'indice I. Soient $\mathcal{R}$ le domaine des références (espace adressable) et $\mathcal{V}$ le domaine des valeurs. Soit $\mathcal{M} : \mathcal{R} \to \mathcal{V}$ le domaine des états mémoires. Une instruction peut être vue comme une fonction d'un état mémoire m vers un autre état mémoire m'. Notons m_i (resp. m'_i) l'état mémoire **avant** (resp. **après**) l'exécution de la i^{eme} instance de l'instruction S (notée S[i]). L'état mémoire précédent la 1^{ere} instance de S est fixé arbitrairement à un état mémoire m_0.

$$m_0 = m_1 \xrightarrow{S[1]} m'_1 \xrightarrow{\cdots} m_2 \xrightarrow{S[2]} m'_2 \xrightarrow{\cdots} \ldots m_N \xrightarrow{S[N]} m'_N$$

Soit Write$[S]_m$ l'ensemble des références écrites par S dans l'état mémoire m. Notons w_i = Write$[S]_{m_i}$ l'ensemble des références écrites par S lors de l'exécution de l'instance i de S. La notation $m|_w$ représente l'état mémoire du domaine des références restreint à w (avec $w \in \mathcal{R}$).

Remarque 13.1
Les notations utilisées pour formaliser les notions d'état mémoire, d'instruction, etc. sont simplifiées au maximum. Le lecteur trouvera dans [Jou87, Cre96] une formalisation beaucoup plus complète visant à proposer une approche sémantique de ces domaines.

Définition 13.1
Une instruction S est dite **invariante pour une boucle L** *(noté $S \in$ Inv(L)) si, et seulement si,*

1. $\forall i \in D_L, w_i = w_1$
2. $\forall i \in D_L, m'_i|_{w_1} = m'_1|_{w_1}$

Remarque 13.2
De manière intuitive, cette définition comporte les deux idées suivantes :

1. *La (ou les) adresse mémoire à laquelle écrit l'instruction est la même pour toutes les itérations de la boucle (une affectation sera donc de la forme $C_1 = B(g(y))$).*
2. *La (ou les) valeur calculée par l'instruction (puis écrite) est identique pour toutes les itérations de la boucle (l'instruction est donc de la forme $A(f(x)) = C_2$).*

L'invariance d'une instruction telle que nous la donnons dans la définition 13.1 permet de construire un sous-ensemble des instructions du programme qui contient les instructions **potentiellement** déplaçables à l'extérieur de la boucle englobante. Cependant, cela n'indique en aucun cas que cette transformation est autorisée. Ce n'est qu'au moment de la génération du code (après simplification du graphe de dépendance), que la transformation sera éventuellement appliquée. L'exemple de la figure 13.1 illustre le cas d'une instruction *invariante* S_2 qui ne peut être extraite de la boucle par simple déplacement de code.

```
       x = 0
       DO i = 1, 100
(S1)     B(i) = f(x,A(i-1))
(S2)     x = 5
(S3)     A(i) = g(x,B(i))
(S4)     x = 6
       ENDDO
```

(a) code initial.

FIG. 13.1 – *Instruction invariante non déplaçable.*

13.1.2 Détection d'instructions invariantes

Afin de simplifier le graphe de dépendances (lorsque cela est possible), nous cherchons à établir une condition suffisante (mais non nécessaire) pour qu'une instruction soit déclarée **invariante pour une boucle**. Le principe de détection des instructions invariantes est exposé à travers la propriété 13.1. Il est ensuite affiné, à deux reprises en 13.3 et 13.4, avec l'objectif d'accélérer la recherche des invariants.

Propriété 13.1
Soit une instruction S quelconque appartenant à une boucle L d'indice i.

– *Si $i \notin$ Ind(S) (i n'apparaît pas syntaxiquement dans l'instruction[61])*

61. La notation Ind(S) est définie à la section 12.4

– *et si* $\forall S' \in Pred(S)$, $S' \in Inv(L)$ *(nous savons prouver que les prédécesseurs* S' *de* S *sont des instructions invariantes pour* L*)*

alors $S \in Inv(L)$ *(*S *est une instruction* ***invariante pour la boucle*** L*).*

Preuve 13.1

Par hypothèse, les prédécesseurs $\{S'_1, \ldots, S'_p\}$ de S sont invariants pour la boucle L. L'invariance de S'_i indique que l'ensemble w_i des références écrites par S'_i est le même pour toutes les itérations de la boucle. De même, l'état mémoire $m|_{w_i}$ après l'exécution des instances de S'_i est toujours le même.

Soit $\mathrm{Read}[S]_m$ les références lues par l'instruction S dans un état mémoire m. Nous savons que $S'_i \in \mathrm{Pred}(S)$ si S utilise une valeur qui est écrite par S'_i. Les p instructions S'_i forment l'ensemble des prédécesseurs de S. Il ne peut donc exister d'autre instruction modifiant une référence lue par S. Les références lues par S forment donc un sous-ensemble des références écrites par tous les S'_i. $\forall m, \mathrm{Read}[S]_m \subset \Sigma_{i=1}^{p} w_i$.

Toutes les instances de S accèdent donc uniquement à une restriction du domaine des références pour lequel l'état mémoire est inchangé pour le domaine d'itération de L. Comme par hypothèse, l'indice i de la boucle n'est pas utilisé par S, cette instruction ne peut écrire à des adresses mémoires différentes. De la même façon, les valeurs calculées restent inchangées sur D_L. S est donc invariante pour L.

Q.E.D.

Détecter les instructions invariantes nécessite donc d'établir, pour chacune des instructions, la propriété d'invariance en fonction de celles de ses prédécesseurs. Or, il n'est pas nécessaire de prendre en compte **tous** les prédécesseurs d'une instruction. Nous pouvons en effet restreindre le parcours aux prédécesseurs *appartenant à la même boucle* que l'instruction examinée.

Propriété 13.2
$\forall S \notin Instr(L), S \in Inv(L)$

Preuve 13.2

Toute instruction n'appartenant pas à la boucle L vérifie immédiatement les conditions de la définition 13.1 puisqu'il n'existe qu'une seule instance de l'instruction qui soit vivante dans le corps de la boucle L.

Q.E.D.

Propriété 13.3
Soit une instruction S *quelconque appartenant à une boucle* L *d'indice* i*.*

– *Si* $i \notin Ind(S)$
– *et* $\forall S' \in Instr(L) \cap Pred(S)$, $S' \in Inv(L)$

alors, $S \in Inv(L)$ *(*S *est une instruction* **invariante pour la boucle** L*).*

Preuve 13.3

Immédiat d'après les propriétés 13.1 et 13.2.

Q.E.D.

Nous pouvons de la même façon nous restreindre aux prédécesseurs S' tels que S et S' appartiennent à la même composante fortement connexe du graphe de dépendances.

Propriété 13.4
Soit une instruction S *quelconque appartenant à une boucle* L *d'indice* i*. Soit* Π *la composante fortement connexe du graphe de dépendances contenant* S*.*

– *Si* $i \notin Ind(S)$

– *et* $\forall S' \in Pred_\Pi(S)$, $S' \in Inv(L)$

alors $S \in Inv(L)$ *(S est une instruction* **invariante pour la boucle** *L).*

Preuve 13.4

Si deux instructions S et S' telles que $S' \xrightarrow{\delta} S$ n'appartiennent pas à la même composante fortement connexe du graphe de dépendances, alors il n'existe aucune dépendance (de type flot, anti, ou output) de S vers S'.
Dans ces conditions, toutes les itérations de S' peuvent être exécutées **avant** toutes les itérations de S. Un ordonnancement tel que toutes les itérations de S lisent la même valeur (résultat de la dernière instance de S') est donc correct. Les instructions en dehors de la composante fortement connexe contenant S n'apportent donc que des valeurs constantes vers S.
Cette propriété est utilisée par l'algorithme de vectorisation d'Allen et Kennedy [AK87]). La boucle L contentant S et S' est scindée en deux boucles L_1 et L_2, contenant respectivement S' et S. Cette transformation nous ramène directement à la propriété précédente concernant une instruction S' n'appartenant pas à la même boucle que S. Ceci prouve une nouvelle fois la propriété ci-dessus.

Q.E.D.

Les propriétés ci-dessus présentent une détection d'instructions invariantes fondée sur la notion de propagation de la propriété d'invariance le long des dépendances de flot. Cette approche est efficace en pratique et permet la détection d'une grande proportion des instructions invariantes. Cependant, cette technique ne permet pas de détecter **toutes** les instructions vérifiant la condition donnée par la définition 13.1 ; nous ne traitons en effet ici que le cas d'invariance *syntaxique* et non les cas d'invariance *sémantique*.

Considérons l'exemple de la figure 13.2-(a). L'instruction S_2 vérifie les conditions de la définition 13.1 dans la mesure où l'adresse mémoire écrite (x) reçoit une valeur constante (5) pour toutes les itérations. Or, cette instruction ne remplit pas les conditions formulées par les propriétés 13.1, 13.3 et 13.4. En effet, d'une part l'instruction S_1 qui est un prédécesseur de S_2 n'est pas déclarée invariante (elle ne l'est pas !), et d'autre part, l'indice i apparaît syntaxiquement dans l'instruction. Les valeurs utilisées ne sont pas constantes sur le domaine d'itération, mais le résultat du calcul l'est. Or, reconnaître l'invariance de l'instruction S_2 permettrait à l'aide des techniques présentée ci-après (simplification du graphe de dépendances puis génération de code), la transformation du code sous la forme présentée figure 13.2-(b). Le cube de x pourrait alors être évalué entre les deux boucles (pour cet exemple, on devrait bien évidemment éliminer la première boucle !).

```
      t = 0
      DO i = 1, 100
(S1)    t = t + 2
(S2)    x = 5 + t/2 - i
(S3)    A(i) = x**3
      ENDDO
```

(a) code initial.

```
      t = 0
      DO i = 1, 100
(S1)    t = t + 2
(S2)    x = 5 + t/2 - i
      ENDDO
      DO i = 1, 100
(S3)    A(i) = x**3
      ENDDO
```

(b) code transformé.

FIG. 13.2 – *Invariance non détectée.*

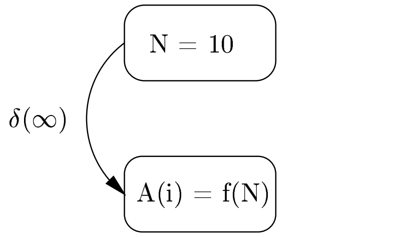
13.1.3 Suppression de dépendances

La présence d'une instruction invariante dans une boucle génère des dépendances que nous souhaiterions supprimer. En effet, si ces contraintes d'ordonnancement des instructions sont indispensables dans le cas général, elles peuvent être relaxées lorsque toutes les instances d'une instruction sont « identiques » (exécuter une instance de l'instruction plutôt qu'une autre ne modifie pas le comportement du programme).

L'exemple de la figure 13.3 montre une instruction S_2 utilisant une valeur définie à l'extérieur de la boucle par une autre instruction S_1. Cette relation *définition-utilisation* est transcrite sur le graphe de dépendances par une dépendance de flot $S_1 \xrightarrow{\delta\ (\infty)} S_2$. Cette dépendance est bien entendu indispensable pour déterminer l'ordre d'exécution des deux affectations. Aucune instance de S_2 ne peut être exécutée avant S_1. La figure 13.4 reprend les deux mêmes instructions, mais cette fois-ci, S_1 se trouve à l'intérieur de la boucle. La variable N reçoit donc la valeur 10 pour chacune des 100 itérations de la boucle. Le graphe de dépendances est quant a lui modifié par l'ajout de trois nouvelles dépendances. L'anti-dépendance $S_2 \xrightarrow{\overline{\delta}\ (1)} S_1$ indique par exemple que S_2 lit à l'adresse mémoire de N pour une itération k de la boucle alors que S_1 écrit à cette même adresse pour une itération k' (avec $k > k'$). Cette dépendance impose donc d'exécuter l'instance k de S_2 avant l'instance k' de S_1. Or, cette contrainte n'est pas réellement justifiée en raison de l'invariance de S_1.

```
(S1) N = 10
      Do i = 1, 100
(S2)    A(i) = f(N)
      Enddo
```

(a) Code source



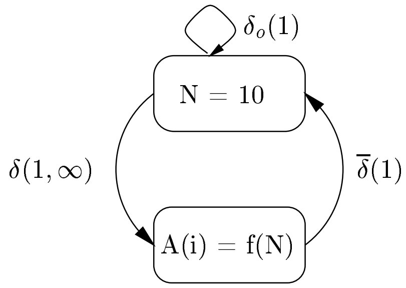
(b) Graphe de dépendances associé

FIG. 13.3 – *Invariance et graphe de dépendances (1/2).*

Notre objectif premier est d'effectuer une simplification du graphe de dépendances de la figure 13.4 pour se ramener à celui de la figure 13.3[62]. La suppression des dépendances $S_1 \xrightarrow{\delta\ (1)} S_2$ et $S_2 \xrightarrow{\overline{\delta}\ (1)} S_1$ autorisera l'exécution de toutes les instances de S_1 **avant** celle de toutes les instances de S_2. De plus, la suppression de la dépendance $S_1 \xrightarrow{\delta_o\ (1)} S_1$ permet d'exécuter toutes les instances de S_1 *indépendamment* les unes des autres. Notons que la dépendance $S_1 \xrightarrow{\delta\ (\infty)} S_2$ n'est pas supprimée. Cette simplification du graphe de dépendances permet de construire un ordonnancement tel que toutes les instances de S_1 sont exécutées (dans n'importe quel ordre) avant toutes les instances de S_2. La génération de code nous permettra de générer un code dans lequel seule une instance de S_1 est exécutée.

```
      Do i = 1, 100
(S1)   N = 10
(S2)   A(i) = f(N)
      Enddo
```

(a) Code source



(b) Graphe de dépendances associé

FIG. 13.4 – *Invariance et graphe de dépendances (2/2).*

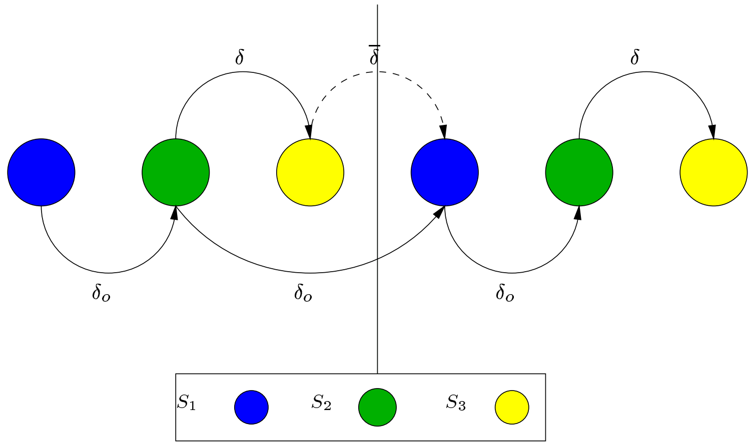
62. Dans cet exemple, une simple privatisation de la variable N permettrait la simplification du graphe de dépendances. Cependant, en présence d'affectations de tableaux ou si N n'est pas privée (instruction utilisant N placée après la boucle), la privatisation ne fonctionne plus !).

L'invariance d'une instruction donnée par la définition 13.1 n'est pas suffisante pour autoriser la suppression de dépendances similaires à celles de la figure 13.4. En effet, plusieurs instructions peuvent écrire de manière concurrente aux mêmes adresses. Dans ce cas, les modifications apportées à l'ordonnancement risquent de compromettre le fonctionnement correct du programme.

Considérons l'exemple de la figure 13.5-(a). Une seconde affectation de N a été ajoutée par rapport à la figure 13.4. Or, cette simple modification interdit d'appliquer une simplification du graphe de dépendances similaire à celle faite ci-dessus. En effet, la suppression de l'anti-dépendance entre S_2 et S_3 (en pointillés[63] sur la figure 13.5-(b)) n'est pas valide. Comme le montre la figure 13.6 la suppression de l'anti-dépendance permet de construire un ordonnancement non valide (l'instruction S_3 reçoit la valeur 10 au lieu de 20).

```
      Do i = 1, 100
(S1)    N = 10
(S2)    N = 20
(S3)    A(i) = f(N)
      Enddo
```

(a) Code source



(b) graphe de dépendances détaillé

FIG. 13.5 – *Deux écritures concurrentes à la même adresse.*

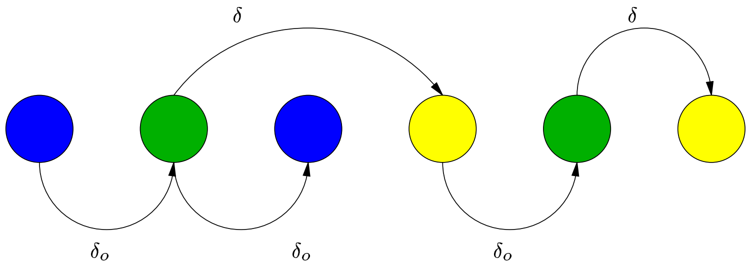


FIG. 13.6 – *Modification interdite du graphe de dépendances.*

La présence de deux accès concurrents en écriture dans une même boucle n'interdit cependant pas systématiquement toute simplification du graphe de dépendances. Nous devons donc rechercher dans quelles conditions la modification du sous-espace d'adressage par une autre instruction interdit ou n'interdit pas la simplification du graphe. Cependant, *la technique de simplification proposée ici ne sera appliquée à aucun code contenant plusieurs instructions appartenant à la même boucle et écrivant à la même adresse mémoire. Les techniques présentées dans cette partie pourront être appliquées, dans la plupart des cas malgré cette restriction, en appliquant simplement d'autres techniques telles qu'un renommage de variables, une élimination de code mort, etc. (voir [Wol96, ZC91]).*

Propriété 13.5
Soit une instruction S appartenant à une boucle L. Si $S \in \mathrm{Inv}(L)$ alors

63. Seules les dépendances non redondantes ont été représentées sur cette figure.

la dépendance $S \xrightarrow{\delta_o\ (1)} S$ *peut être supprimée.*

Preuve 13.5

Immédiat du fait de l'invariance de l'instruction.

Q.E.D.

Cette propriété indique simplement que les différentes instances d'une instruction invariante peuvent être exécutées dans un ordre quelconque. Elle est cependant importante car la suppression de la dépendance de cette instruction vers elle-même efface un cycle de dépendance, ce qui est impératif pour le bon fonctionnement de la génération de code.

Propriété 13.6

Soit une instruction S *appartenant à une boucle* L. *Si* $S \in Inv(L)$ *et si* $\nexists S'' \in Instr(L)$ *telle que* $S'' \xrightarrow{\delta_o} S$ *alors*

$\forall S' \in Instr(L)$ *telle que* $S \xrightarrow{\delta\ (1,\infty)} S'$, $S' \xrightarrow{\overline{\delta}\ (1)} S$, *les dépendances* $S' \xrightarrow{\overline{\delta}\ (1)} S$ *et* $S \xrightarrow{\delta\ (1)} S'$ *peuvent être supprimées*[64].

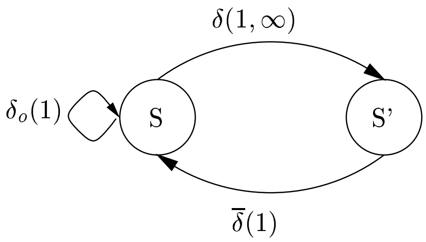


FIG. 13.7 – *Le graphe de dépendances résumé.*

Cette propriété s'applique pour un graphe de dépendances tel que celui présenté figure 13.7. Pour simplifier, l'explication sera construite sur le graphe de dépendances **détaillé**. L'ordonnancement présenté par les deux types de représentation est supposé équivalent. La simplification proposée transforme le graphe de la figure 13.8 en celui de la figure 13.9.

Preuve 13.6

La suppression des dépendances est correcte si pour tout ordonnancement construit à partir du nouveau graphe, l'instruction S' reçoit les mêmes valeurs de la part de l'instruction S que dans l'ordonnancement initial.

64. La dépendance $S \xrightarrow{\delta\ (\infty)} S'$ est conservée

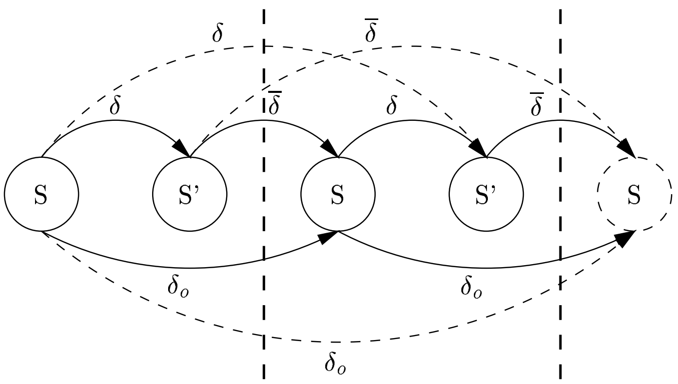


FIG. 13.8 – *Le graphe de dépendances détaillé.*

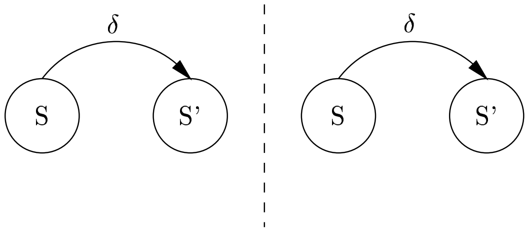


FIG. 13.9 – *Le graphe de dépendances détaillé simplifié.*

La conservation de la dépendance $S \xrightarrow{\delta\ (\infty)} S'$ garanti que la i^{eme} instance de S est exécutée **avant** la i^{eme} instance de S'. Cependant, comme le montre la figure 13.9, toutes les instances de S pourraient par exemple être exécutées **avant** toutes les instances de S'.
Par hypothèse, S est une instruction invariante pour L (voir définition 13.1). Par ailleurs, il n'existe aucune instruction S'' appartenant à L et telle que $S'' \xrightarrow{\delta_o\ (1)} S$ (c'est-à-dire écrivant à une même adresse que S). L'état mémoire $m|_w$ (restriction aux références écrites par S) n'est donc pas modifié entre la première instance de S et l'exécution de la dernière instance de la dernière instruction de la boucle.
L'existence d'une dépendance $S \xrightarrow{\delta\ (\infty)} S'$ indique que l'instruction S est placée *textuellement devant* l'instruction S' dans le corps de la boucle. Toutes les instances de S' (y compris la première) utilisent donc des valeurs définies par S[65]. L'état mémoire restreint aux références écrites par S et lues par S' est donc constant. Quelle que soit l'instance de S exécutée immédiatement avant S', la valeur lue par une instance quelconque de celle-ci sera donc la même que pour l'ordonnancement initial.

Q.E.D.

La simplification effectuée autorise un ordonnancement dans lequel toutes les instances de S sont exécutées **avant** toutes les instances des instructions S'. Il n'existe par ailleurs plus d'ordre à respecter entre les différentes instances de S (propriété 13.5).

Généralisation aux boucles imbriquées : La technique présentée ci-dessus est directement transposable à une imbrication de boucles. La condition d'invariance présentée est simplement étendue de la manière suivante :

Propriété 13.7
Soit une instruction S quelconque appartenant à une suite ordonnée de boucles $L_1, L_2, \ldots, L_n$, où L_1 est la boucle la plus externe. Les indices sont notés $i_1, i_2, \ldots, i_n$. Soit Π la composante fortement connexe du graphe de dépendances contenant S. Soit une profondeur k quelconque ($k \in [1, n]$).

- *Si* $\forall j \in [k, n], i_j \notin Ind(S)$
- *et* $\forall j \in [k, n], \forall S' Pred_{\Pi}(S)\ ,\ S' \in Inv(L_j)$

alors $\forall j \in [k, n], S \in Inv(L_j)$ *(S est* **invariant pour les boucles** $L_k, \ldots, L_n$*).*

Preuve 13.7

Cette propriété peut être validée par récurrence sur la profondeur k. Nous appelons C_k la condition exprimée pour le niveau k et P_k la propriété d'invariance pour les boucles de k à n :

- $C_k : \forall j \in [k, n], i_j \notin \mathrm{Ind}(S)$ et $\forall j \in [k, n], \forall S'\mathrm{Pred}_{\Pi}(S)\ ,\ S' \in \mathrm{Inv}(L_j)$
- $P_k : \forall j \in [k, n], S \in \mathrm{Inv}(L_j)$

65. Le cas particulier des affectations gardées par un test sera traité par la suite.

La propriété est vraie au niveau $k = n$, puisque l'on se ramène aisément à la propriété 13.4 pour une boucle L_n d'indice i_n.

- $C_n : i_n \notin \text{Ind}(S)$ et $\forall S' \text{Pred}_\Pi(S),\ S' \in \text{Inv}(L_n)$
- $P_n : S \in \text{Inv}(L_n)$

$$C_n \Rightarrow P_n$$

Posons l'hypothèse suivante : la propriété est vraie pour une profondeur ($k = t+1$) quelconque. Nous avons donc

$$C_{t+1} \Rightarrow P_{t+1}$$

Nous voulons montrer que la propriété est vraie pour le niveau ($k = t$). Nous cherchons à prouver que C_t implique P_t.

- $C_t : \forall j \in [t,n], i_j \notin \text{Ind}(S)$ et $\forall j \in [t,n], \forall S' \in \text{Pred}_\Pi(S)\ ,\ S' \in \text{Inv}(L_t)$
- $P_t : \forall j \in [t,n], S \in \text{Inv}(L_j)$

Nous savons que C_t est vraie et que $C_{t+1} \Rightarrow Pt+1$. Par construction de C_k, nous avons immédiatement $C_t \Rightarrow C_{t+1}$. Nous obtenons donc $C_t \Rightarrow C_{t+1} \Rightarrow P_{t+1}$. L'instruction S est invariante pour les boucles $L_{t+1}, \ldots, L_n$.

L'état mémoire restreint aux références écrites par S reste donc le même pour toutes les itérations des boucles $L_{t+1}, \ldots, L_n$. L'invariance de S pour la boucle L_t ne dépend donc que de l'évolution de l'état mémoire entre deux itérations de cette boucle. On se ramène donc à l'étude des propriétés d'invariance de l'instruction S pour la boucle L_t en faisant abstraction des autres boucles de profondeur supérieure à t. Or, d'après L_t, nous avons $i_t \notin \text{Ind}(S)$ et $\forall S' \in \text{Pred}_\Pi(S),\ S' \in \text{Inv}(L_t)$. La propriété 13.4 nous donne alors $S \in \text{Inv}(L_t)$.

Les deux propriétés P_{t+1} et $S \in \text{Inv}(L_t)$ forment la propriété Pt. Nous aboutissons donc au résultat recherché $C_t \Rightarrow P_t$.

Q.E.D.

13.1.4 Algorithme de simplification

D'après la propriété 13.7, l'invariance de chaque instruction est conditionnée par celle de ses prédécesseurs. La détection des invariants, en une passe, nécessite donc d'établir un ordre entre les instructions. Pour cela, notre algorithme utilise le graphe de dépendances restreint aux dépendances de flot, représentation immédiate de la notion de prédécesseurs. Il recherche ensuite les composantes fortement connexes de ce graphe et établit un tri topologique[66]. Cette technique garanti qu'au moment où l'invariance d'une instruction est testée, celle de tous ses prédécesseurs est déjà connue. Lorsqu'il existe un cycle de dépendances de flot porté par une ou plusieurs instructions, la propriété d'invariance de la ou des instructions est infirmée. Un cycle apparaît notamment dans les cas de réductions dans des variables scalaires.

Considérons l'exemple donné figure 13.10. Le graphe de dépendances est composé d'une seule et unique composante fortement connexe. La figure 13.11 présente le graphe restreint aux dépendances de flot, à partir duquel un ordre est déterminé entre les instructions. Les composantes fortement connexes sont schématisées par des cercles tracés en pointillés, et les chiffres présentés indiquent l'ordre obtenu. Les informations sont construites progressivement :

- L'instruction S_1 ne contient pas l'indice i, et n'a aucun prédécesseur. Elle est donc déclarée *invariante.*
- L'instruction S_2 qui ne comporte qu'un prédécesseur invariant est elle aussi *invariante.*
- Les instructions S_3 et S_4 forment un cycle de dépendances de flot. Toutes deux sont donc déclarées *non invariantes.*
- S_5 ayant un prédécesseur non invariant est elle aussi déclarée *non invariante.* De même pour S_6 qui utilise i et a un prédécesseur non invariant.

Des simplifications du graphe de dépendances peuvent apparaître pour différents niveaux de dépendances. L'algorithme 5 tente de simplifier toutes les composantes fortement connexes du graphe de

66. Le graphe des composantes fortement connexes est acyclique.

Algorithme 5 Simplification du graphe de dépendances

Fonction Simplifier_graphe(D,k)
- ▷ *entrée : - D : le graphe de dépendances $D = (V, E)$ et k : le niveau minimum pour les dépendances de D*
- ▷ *sortie : - E' ($E' \subset E$) : les arcs à supprimer dans D*
- ▷ $\forall e \in E'$ *on note* $e =< S_1, S_2, t, n >$ *pour* $S_1 \xrightarrow{t\ (n)} S_2,\ t \in \{\delta, \overline{\delta}, \delta_o\}$
- **Soit** $\{C_1, \ldots, C_m\}$ les composantes fortement connexes maximales sur D
- $E' \leftarrow \emptyset$
- **Pour** $i \leftarrow 1, m$ **Faire**
  - **Si** C_i est fortement connexe **Alors**
    - ▷ *graphe de dépendances restreint aux nœuds de C_i*
    - **Soit** $D_i = D|_S$ pour $S = \{v \in C_i\}$
    - $E'_i \leftarrow$ Supprimer_dépendances(D_i, k)
  - **Fin Si**
- **Fin Pour**
- **Retourner** $E' = \cup_{i=1}^{m} E'_i$

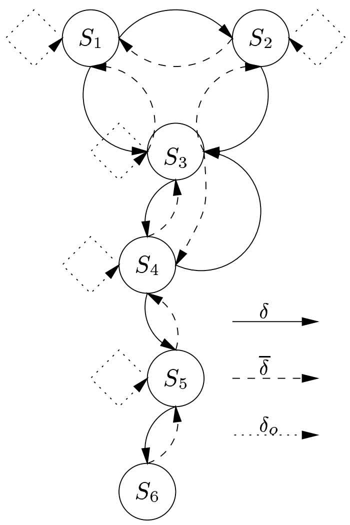
Fin

Fonction Supprimer_dépendances(D,k)
- $E_1 \leftarrow \emptyset$
- ▷ *graphe de dépendances restreint aux flots*
- **Soit** $D' = (V, E')$ pour $E' = \{< S_1, S_2, t, k > | t = \delta\}$
- **Soit** $\{C_1, \ldots, C_m\}$ les composantes fortement connexes maximales sur D'
- Construire $\{\Pi_1, \Pi_2, ..., \Pi_m\}$ un ordre topologique sur $\{C_1, C_2, ..., C_m\}$
- **Pour** $i \leftarrow 1, m$ **Faire** ▷ *dans l'ordre topologique*
  - **Si** Π_i est fortement connexe **Alors**
    - **Pour tout** $S_j \in \Pi_i$ **Faire**
      - Marquer S_j comme non invariant ($S_j \notin \mathrm{Inv}(L)$)
    - **Fin Pour**
  - **Sinon** ▷ *Π_i n'est pas fortement connexe (aucun arc !)*
    - ▷ *test syntaxique sur les indices supérieurs à i_k*
    - ▷ *soit $\rho(\Pi_i)$ le nombre de boucles contenant Π_i (qui est forcement une instruction !)*
    - **Si** $\forall i_j,\ k \leq j \leq \rho(\Pi_i),\ i_j \notin \mathrm{Ind}(\Pi_i)$
    - et $\forall S' \in Pred(\Pi_i), S' \in Inv(L_k)$ **Alors**
      - Marquer Π_i comme invariant ($\Pi_i \in \mathrm{Inv}(L)$)
      - **Pour tout** $S'' \in Succ(\Pi_i)$ **Faire**
        - **Si** $\Pi_i \xrightarrow{\delta\ (k,\infty)} S''$, $S'' \xrightarrow{\overline{\delta}\ (k)} \Pi_i$ et $\Pi_i \xrightarrow{\delta_o\ (k)} \Pi_i$ **Alors**
          - ajouter $\Pi_i \xrightarrow{\delta\ (k)} S''$, $S'' \xrightarrow{\overline{\delta}\ (k)} \Pi_i$ et $\Pi_i \xrightarrow{\delta_o\ (k)} \Pi_i$ à E_1
        - **Fin Si**
      - **Fin Pour**
    - **Fin Si**
  - **Fin Si**
- **Fin Pour**
- ▷ *graphe de dépendances restreint aux arcs $E - E'$ de niveau $k+1$ ou plus*
- **Soit** $D'' = (V, E'')$ pour $E'' = \{e =< V_1, V_2, t, j > | j \geq k + 1$ et $e \in E - E'\}$
- $E_2 =$ Simplifier_graphe(D'',$k+1$)
- **Retourner** $E_1 \cup E_2$

Fin

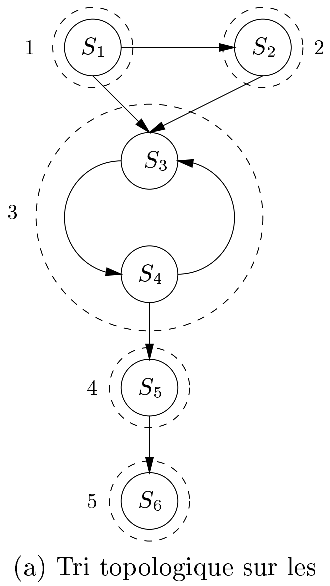
```
     C = 30
     Do i = 1, 100
(S1)  N = 10
(S2)  M = N * C
(S3)  X = Y + M + N
(S4)  Y = X**2
(S5)  T = Y + 1
(S6)  A(i) = T / 2
     Enddo
```

(a) Code source

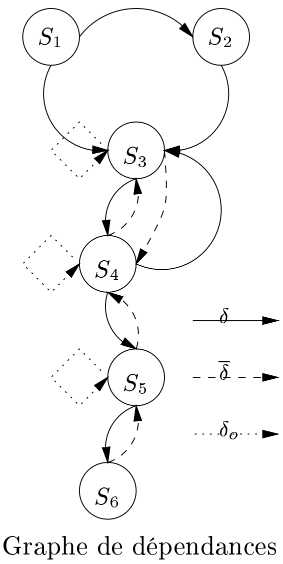


(b) Graphe de dépendances associé

FIG. 13.10 – *Exemple de simplification du graphe (1/2).*



(a) Tri topologique sur les dépendances de flot



(b) Graphe de dépendances après simplification

FIG. 13.11 – *Exemple de simplification du graphe (2/2)*

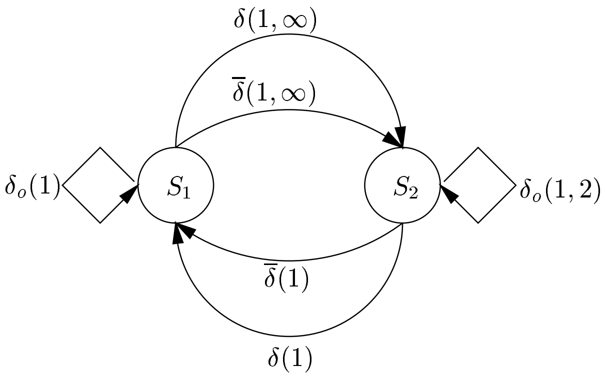
dépendances. Pour chacune d'elles, il recherche les arcs à supprimer et fonctionne de manière récursive (en travaillant donc pour tous les niveaux de dépendances). Il effectue une descente progressive dans l'imbrication des boucles pour y rechercher des invariants (fonction *Simplifier_graphe*(D,k)). L'appel initial travaille pour un niveau 1 de dépendances, et sur le graphe complet.

Pour une profondeur donnée, la fonction *Supprimer_dépendances* étudie l'invariance de chaque instruction (suivant le tri topologique), puis élimine les arcs suivant les propriétés décrites par la propriété 13.6.

L'existence d'un cycle de dépendances pour une étape donnée de l'algorithme ne préjuge pas de l'invariance des instructions pour les étapes suivantes. Les instructions S_1 et S_2 présentées figure 13.12 forment un cycle de dépendances de flot au niveau 1. Leur invariance pour la boucle L_1 est donc infirmée. L'instruction S_2 est cependant invariante pour la boucle L_2. Une analyse basée sur les use-def/def-use serait insuffisante pour traiter ce type de réduction (le niveau de dépendance est indispensable).

```
      C = 30
      Do i = 1, 100
(S1)    X = Y + 5
          Do j = 1, 100
(S2)        Y = X ** 3
(S3)        ...
          Enddo
      Enddo
```

(a) Code source



(b) Graphe de dépendances associé

FIG. 13.12 – *Exemple de réduction sur deux niveaux de boucles.*

13.1.5 Extensions

La simplification du graphe de dépendances peut être également effectuée pour des instructions vérifiant d'autres propriétés. Le cas intéressant des instructions qui sont potentiellement déplaçables *après* la boucle sera présenté ici. La caractéristique principale de ces instructions est que l'on peut se contenter de n'exécuter que leur « dernière instance ».

Définition 13.2
Une instruction est dite **partiellement invariante pour une boucle** L *(notée* $S \in Par\_Inv(L)$*) si, et seulement si,*

$$\forall i \in D_L, w_i = w_1$$

Cette notion d'invariance *partielle* nous permet de définir l'ensemble des instructions qui écrivent à la même adresse mémoire pour toutes les itérations de la boucle. La recherche de ces instructions est très similaire à la recherche des instructions invariantes ; une distinction importante existe néanmoins. L'étude des dépendances entre instructions n'est plus suffisante, puisqu'il faut pouvoir se limiter aux dépendances ne concernant que les références impliquées dans des opérations d'écriture mémoire (ce qui correspond au membre gauche pour une instruction d'affectation). Pour chacune de ces références, il faut contrôler d'une part l'absence syntaxique des indices de boucles, et d'autre part l'absence de dépendance de flot issue d'une instruction non invariante et concernant cette référence. Notre implémentation utilise les effets et régions *write* d'une instruction pour déterminer l'ensemble des références impliquées dans une écriture mémoire (cf. chapitre 5).

Détection des instruction redondantes

Définition 13.3

Une instruction est dite **redondante pour une boucle** L *(noté $S \in Red(L)$) si, et seulement si,*

1. $S \in Par\_Inv(L)$
2. $\forall S'$, $S \xrightarrow{\delta} S'$, $S' \in Red(L)$

De manière intuitive, cette définition reprend les deux idées suivantes :

1. l'instruction écrit toujours à la même adresse mémoire.
2. seules des instructions redondantes utilisent le résultat de l'instruction considérée.

Une illustration de cette propriété des instructions est donnée figure 13.13. Les instructions S_2, S_3 et S_4 écrivent toujours à la même adresse mémoire (pour toutes les itérations de la boucle). En revanche, la valeur écrite est modifiée à chaque itération. La variable t_3 n'étant pas utilisée dans le corps de la boucle, seule la dernière instance de S_4 est indispensable. L'instruction S_4 peut donc être déplacée après le corps de la boucle. La propriété de redondance se propage à l'instruction S_2 qui vérifie les propriétés de la définition 13.3. Par contre, l'instruction S_3 dont le résultat est utilisé par S_5 (non redondante), ne peut être déplacée.

```
       Do i = 1, 100
(S1)     x = x + 1
(S2)     t1 = x ** 2
(S3)     t2 = x * 2
(S4)     t3 = t1 + t2 + 1
(S5)     A(i) = t2 + ...
       Enddo
```

(a) Code source.

```
       Do i = 1, 100
(S1)     x = x + 1
(S3)     t2 = x * 2
(S5)     A(i) = t2 + ...
       Enddo
(S2)   t1 = x ** 2
(S4)   t3 = t1 + t2 + 1
```

(b) Code transformé.

FIG. 13.13 – *Exemple de déplacement de code après une boucle.*

La recherche des instructions *partiellement redondantes* est effectuée dans un premier temps (elle peut être faite parallèlement à la recherche des instruction *invariantes*). La méthode utilisée pour contrôler la redondance des successeurs d'une instruction est fondée sur un tri topologique **inverse** à partir du graphe restreint aux dépendances de flot.

Transformation du graphe de dépendances

Nous avons utilisé la propriété d'invariance d'une instruction pour effectuer des simplifications sur le graphe de dépendances. Le graphe modifié est ensuite utilisé pour regénérer une nouvelle version du programme.

Dans le cas des instructions redondantes, la suppression de dépendances n'est pas possible directement. Considérons l'exemple de la figure 13.14. Il existe une output-dépendance de l'instruction S_2 vers elle-même ; cette dépendance impose un ordre d'exécution entre toutes les instances (en pointillé sur la figure 13.15). Le cycle de dépendances interdit la génération d'une boucle parallèle autour de l'instruction. Mais, la suppression de l'output-dépendance (qui serait effectuée si l'instruction était invariante) est interdite : l'ordonnancement présenté sur la partie inférieure de la figure 13.15 n'est pas valide ! Contrairement au cas des instructions invariantes pour lesquelles il est possible de conserver *n'importe laquelle* des instances, l'instruction redondante nous amène à ne conserver que la **dernière** instance de l'instruction [67].

Pour appliquer une transformation du graphe de dépendances qui soit valide et qui permette ensuite de générer un code dans lequel une instruction redondante est extraite de la boucle, il est donc nécessaire de faire intervenir la notion de *dernière* instance. À chaque instruction du graphe de dépendances est

67. Cela correspond à l'application combinée de *loop splitting* puis élimination de boucle inutile.

```
     Do i = 1, 100
(S1)   A(i) = ...
(S2)   t = f(A(i))
     Enddo
```

FIG. 13.14 – *Exemple d'instruction redondante.*

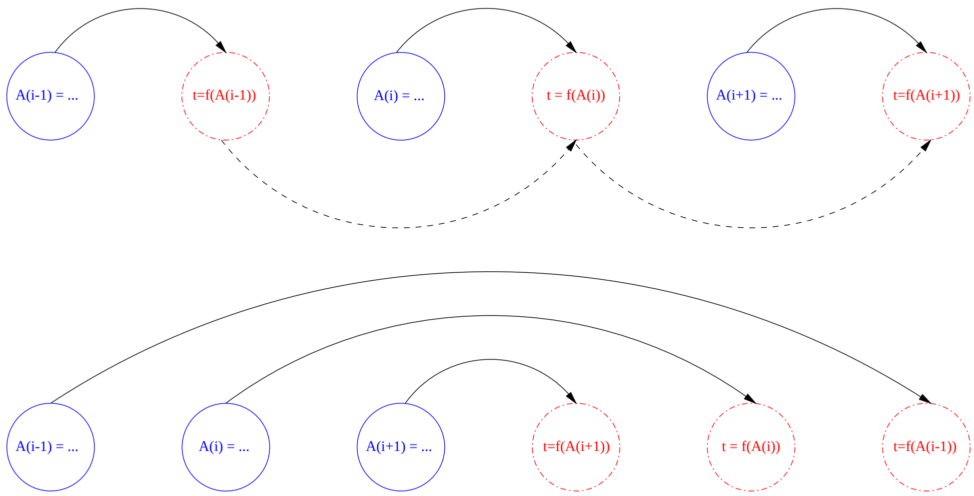


FIG. 13.15 – *Suppression interdite des dépendances.*

associée une information permettant de décrire la présence de l'instruction dans la boucle initiale. La génération de code utilise ensuite cette information (notamment, pour connaître les bornes de boucles). Pour indiquer que la transformation recherchée ici ne doit conserver que la dernière instance d'une instruction, nous devons donc modifier l'information qui lui est associée. Or, cette modification n'est valide que si l'instruction peut-être placée **seule** dans une boucle. Cette condition est vérifiée si la composante fortement connexe du graphe de dépendances ne contient que l'instruction considérée.

Propriété 13.8
Soit une instruction appartenant à une boucle L d'indice i, de bornes lb_i et ub_i et de pas s_i. Si $S \in Red(L)$ et si $\nexists S'$, $S' \xrightarrow{\delta_\circ\ (1)} S$ alors le graphe de dépendances est modifié comme suit :

- *la dépendance $S \xrightarrow{\delta_\circ\ (1)} S$ est supprimée.*
- *$\forall S' \in Instr(L)$ telle que $S' \xrightarrow{\delta\ (1,\infty)} S$, $S \xrightarrow{\overline{\delta}\ (1)} S'$, les dépendances $S' \xrightarrow{\delta\ (1)} S$ et $S \xrightarrow{\overline{\delta}\ (1)} S'$ peuvent être supprimées.*
- *l'instruction S est placée dans une boucle L' d'indice $i' = i$. Les bornes de la boucle [ANS83] sont :*
$$lb'_i = ub'_i = lb_i + [\max(INT(\tfrac{ub_i - lb_i + s_i}{s_i}), 0) - 1] \times s_i \text{ et } s'_i = s_i.$$

Preuve 13.8

- La suppression de l'output-dépendance est évidemment correcte puisqu'il n'existe plus qu'une seule instance de l'instruction.
- La suppression des dépendances $S' \xrightarrow{\delta\ (1)} S$ et $S \xrightarrow{\overline{\delta}\ (1)} S'$ permet l'exécution de toutes les instances de S après celles de ses prédécesseurs S'.
  Toutes les instructions utilisant une valeur écrite par S sont redondantes, ou extérieures à la boucle. Donc, aucune instruction (non redondante) présente dans le nid de boucle

n'utilise une valeur écrite par S. Il nous faut alors simplement vérifier que la simplification du graphe ne modifie rien pour les instructions *extérieures* à la boucle. Or, ces dernières utilisent les valeurs écrites par la dernière instance de S, ce qui sera toujours le cas.

– Nous devons montrer que l'instruction S se trouve **seule** dans sa composante fortement connexe. D'après la propriété 13.3, nous pouvons affirmer qu'il n'existe aucune instruction $S' \in \text{Instr}(L)$ telle que $S \xrightarrow{\delta} S'$[68]. Par hypothèse nous savons également qu'il n'existe aucune instruction S' telle que $S' \xrightarrow{\delta_o\ (1)} S$. Enfin, toutes les anti-dépendances $S \xrightarrow{\bar{\delta}\ (1)} S'$ ont été supprimées. Partant de ces trois restrictions, il est impossible de construire un cycle de dépendances entre S et une autre instruction S'.

Q.E.D.

13.2 Génération de code

La simplification du graphe de dépendances est une étape indispensable dans la transformation du code source. Cependant, elle n'est pas suffisante et il faut également générer une nouvelle version du code. Pour cela, nous utilisons les techniques de génération de code à partir du graphe de dépendances qui sont proposées dans [AK87] (voir la section 12.3 en début de chapitre). *Nous supposons que les boucles considérées sont toujours exécutées au moins une fois ; les techniques d'instructions gardées présentées à la section 13.4 permettront de traiter correctement les cas de boucles vides.*

13.2.1 Présentation de la technique utilisée

La simplification du graphe de dépendances permet d'extraire certaines instructions invariantes des composantes fortement connexes. La disparition des cycles de dépendances permet à l'algorithme de parallélisation d'Allen et Kennedy (algorithme 4) de générer des boucles parallèles. Or, par construction de l'algorithme, il est possible de montrer que si l'indice d'une borne de boucle parallèle n'apparaît dans aucune des instructions de cette boucle, alors elle est inutile (seul le corps de la boucle est repris).

Propriété 13.9
Soit une boucle **parallèle** *L d'indice i contenant p instructions. Si i n'apparaît syntaxiquement dans aucune des p instructions de cette boucle, alors l'entête de la boucle peut-être supprimé (non générée par l'algorithme).*

Preuve 13.9

La suppression de l'entête de la boucle L est correcte si les p instructions sont invariantes pour cette boucle. Or, par construction, l'algorithme de parallélisation d'Allen et Kennedy garantit l'absence de composante fortement connexe pour le niveau traité entre les p différentes instructions. Les conditions données par la propriété 13.7 et permettant d'affirmer l'invariance d'une instruction se ramènent donc immédiatement à l'étude de la présence syntaxique de l'indice de la boucle.

Pour chaque boucle d'indice i supprimée (entête supprimé et corps conservé), une nouvelle instruction d'initialisation de i est ajoutée afin que sa valeur à la sortie de la boucle reste inchangée par rapport à la version originale du code. Cette valeur est déterminée en fonction des bornes et du pas de la boucle [ANS83].

Q.E.D.

Notons que, dans la plupart des cas, l'indice de boucle ne sera pas réutilisé dans la suite du code. L'utilisation d'une phase d'élimination de code mort ou le contrôle de l'existence d'effets `OUT` (cf. chapitre 5) concernant cet indice seront suffisants pour supprimer cette initialisation lorsque l'on sait prouver qu'elle

68. On suppose que toutes les instructions redondantes utilisant un résultat de S' ont déjà été déplacée.

est inutile. Un exemple d'initialisation de l'indice de boucle est donné pour Fortran sur le figure 13.16. L'initialisation de l'indice i qui est utilisé après la boucle doit être ajoutée ; en revanche, l'initialisation de j est inutile car la variable j est ré-écrite avant d'être lue.

```
Do i = N, M, 2
  A(i) = f(i)
  B = 10
Enddo
Do j = 1, 100
  C = 5
  D(j) = g(i,j)
Enddo
```

(a) Code source.

```
C instruction non déplacée
    Do i = N, M, 2
      A(i) = f(i)
     Enddo
C code généré par le déplacement de B=10
     B = 10
     i = N+MAX(INT((2-N+M)/2), 0)*2
C code généré par le déplacement de C=5
     C = 5
     j = 101
C instruction non déplacée
    Do j = 1, 100
      D(j) = g(i,j)
     Enddo
```

(b) Code transformé.

FIG. 13.16 – *Initialisation de l'indice de boucle.*

L'algorithme 6 fonctionne par descente récursive dans les composantes fortement connexes du graphe de dépendances. Il reprend l'algorithme de génération de code de [AK87], tout en ajoutant des conditions sur la génération des boucles parallèles.

Extension

Nous avons vu précédemment qu'il est possible d'effectuer des transformations du graphe de dépendances pour des instructions appelées *redondantes* (voir la section 13.1.5). Ces transformations permettront d'extraire la plupart des instructions redondantes des boucles. Mais, si l'on se contente de la génération de code proposée ci-dessus, les boucles contenant des instructions redondantes de la forme $t = f(i)$ (où i est l'indice de boucle) ne seront pas supprimées (d'après la propriété 13.9). Pour profiter de la redondance de l'instruction dans ce cas précis, il sera en effet nécessaire d'insérer dans le code, **devant** l'instruction redondante, une initialisation de l'indice de chacune des boucles supprimées. Cette initialisation reprend la valeur de l'indice pour la dernière instance de la boucle (cf. section 13.1.5).

La transformation de l'exemple donné figure 13.17 donne le nouveau code présenté figure 13.14.

```
     Do i = n, m, s
(S1)    A(i) = ...
     Enddo
     i = n + (max(INT( (m - n + s)/s),0) -1)*s
(S2)t = f(A(i))
     i = n + max(INT( (m - n + s)/s),0)*s
```

FIG. 13.17 – *Exemple d'instruction redondante extraite de la boucle.*

13.2.2 Variation de l'algorithme sans distribution de boucle

La technique de génération de code présentée ci-dessus repose sur la distribution des boucles. Cette démarche est intéressante dans la mesure où elle permet d'extraire des calculs invariants des boucles qui

Algorithme 6 Génération de code

Fonction Générer_code(D,k)
- ▷ *- D : le graphe de dépendances $D = (V, E)$*
- ▷ *- k : le niveau minimum pour les dépendances de D*
- Chercher l'ensemble $\{C_1, \ldots, C_m\}$ des composantes fortement connexes maximales sur D
- Construire D_Π le graphe de dépendances tel que C_i est un nœud unique (Π-block)
- Soient $\{\Pi_1, \ldots, \Pi_m\}$ les nœuds de D_Π dans l'ordre topologique
- **Pour** $i \leftarrow 1, m$ **Faire**
  - **Si** Π_i est fortement connexe **Alors**
    - **Soit** $l = l_{min}(\Pi_i)$ ▷ *niveau minimum des dépendances de Π_i*
    - **Pour** $t \leftarrow k, l-1$ **Faire**
      - Générer_boucle_parallèle(Π_i,t)
    - **Fin Pour**
    - Générer la boucle de niveau l
    - **Soit** D'_i la restriction de D aux dépendances internes à Π_i et de niveau $(l+1)$ ou plus
    - Générer_code(D'_i, $l+1$)
  - **Sinon** ▷ *Π_i est réduit à une seule instruction - plus aucun arc*
    - ▷ *soit $\rho(\Pi_i)$ le nombre de boucles contenant Π_i*
    - **Pour** $i \leftarrow 1, \rho(\Pi_i) - k$ **Faire** ▷ *boucles restantes*
      - Générer_boucle_parallèle(Π_i,$k+i$)
    - **Fin Pour**
    - Générer le code de l'instruction Π_i
  - **Fin Si**
- **Fin Pour**

Fin

Fonction Générer_boucle_parallèle(C,k)
- **Si** $\forall S \in C$, $i_k \notin \mathrm{Ind}(S)$ **Alors**
  - Ne pas générer l'entête de la boucle de niveau k
- **Sinon**
  - Générer la boucle de niveau k
- **Fin Si**

Fin

ne pourraient pas l'être avec les techniques classiques (qui ne modifient pas la structure de contrôle). La figure 13.18 illustre ce phénomène : l'instruction S_2 ne peut être extraite de la boucle que par utilisation de la distribution de boucle. En effet, ce calcul ne peut être déplacé ni devant ni derrière la boucle.

```
       Do i = 1, 100
(S1)     A(i) = ...
(S2)     t  = A(1)
(S3)     B(i) = f(t)
       Enddo
```

(a) Code source.

```
       Do i = 1, 100
(S1)     A(i) = ...
       Enddo
(S2)   t  = A(1)
       Do i = 1, 100
(S3)     B(i) = f(t)
       Enddo
```

(b) Code transformé.

FIG. 13.18 – *Déplacement de code impossible sans distribution.*

Cependant, la distribution de boucles peut engendrer un surcoût non négligeable en raison de l'introduction de nouvelles boucles, c'est-à-dire plus de contrôle, et de la dégradation de la localité. Ainsi, pour des programmes présentant peu d'opportunités de déplacement de code, ce surcoût peut devenir problématique.

Une première solution consisterait à appliquer une phase de fusion de boucles après l'utilisation du déplacement de code. Les boucles pour lesquelles la distribution n'a pas permis d'extraire des instructions « bloquées » (i.e. non extraites par les techniques classiques) peuvent être aisément (re)fusionnées.

Il serait également possible d'utiliser une variation de l'algorithme de génération de code n'effectuant pas de distribution de boucle. Les opportunités de déplacement de code sont alors réduites, mais le surcoût de la distribution disparaît. Pour ce faire, au lieu de travailler sur les composantes fortement connexes du graphe de dépendances, il suffit de manipuler le sous-graphe restreint aux instructions présentes dans le corps de la boucle courante. L'algorithme génère un ordonnancement entre les instructions qui respecte les dépendances. Enfin, une instruction peut être « extraite » de la boucle si elle est *invariante* (resp. redondante) pour la boucle et si il existe un ordonnancement la plaçant *en tête* (resp. en fin) de boucle.

13.3 Déplacement des appels de fonctions

Les travaux présentés ci-dessus pour les instructions de type *affectation* sont transposables aux cas plus complexes que représentent les appels de fonctions. Pour cela, nous manipulons des informations interprocédurales (fournies par Pips) permettant de s'assurer de la validité des transformations.

13.3.1 Motivations

Même lorsque la quantité de calculs effectués est faible, un appel de fonction est toujours relativement coûteux en raison du surcoût introduit par le changement de contexte d'exécution. Une alternative possible est alors d'appliquer une phase d'*inlining* ; cette technique consiste à remplacer l'appel de procédure par le corps de celle-ci [Sch77]. Cependant, une augmentation exponentielle de la taille du programme risque d'introduire des surcoûts majeurs. C'est pourquoi, le déplacement des appels de fonctions en dehors des boucles est une transformation très profitable. Nos travaux sont motivés à deux niveaux.

Évaluation de fonctions mathématiques. La présence d'appels de fonctions dans les applications scientifiques apparaît la plupart du temps sous la forme d'évaluation de fonctions mathématiques. Les exemples les plus fréquemment rencontrés sont les fonctions trigonométriques (cos, sin, ...), les extrema (min, max, ...), la racine carrée, le logarithme, etc. L'évaluation de telles fonctions coûte cher en temps d'exécution, et donc leur déplacement à l'extérieur des boucles est vraiment important.

Génération automatique de code. Le recours à des librairies de toutes sortes est une constante de la génération de code automatique. Par exemple, pour la parallélisation automatique, la majorité des compilateurs génèrent des appels à des fonctions chargées de la gestion des communications entre les processeurs. Des appels à des fonctions permettant de connaître l'identité du processeur « voisin », la zone de données à communiquer, etc. sont introduits dans le code, et appelés à l'exécution. L'ensemble des données à communiquer ne peut par exemple pas toujours être connu à la compilation (si le nombre de processeurs n'est pas connu ou si des paramètres symboliques sont utilisés pour les bornes de boucles). Des fonctions dont le rôle est de déterminer cet ensemble sont alors appelées à l'exécution. Or, si l'ensemble des éléments à communiquer ne change pas au cours des différentes itérations d'une boucle (les valeurs changent mais pas le « pattern » à communiquer), la détermination de la zone à communiquer peut n'être effectuée qu'une seule fois. Ce phénomène est rencontré régulièrement dans les problèmes irréguliers ([Coe96, HKK+92, BSCV95]). Le code généré contient alors une phase appelée *inspector* chargée de la construction des schémas de communication, et une phase appelée *constructor* effectuant les communications.

Ce phénomène apparaît dans la dérivation d'un code SPMD à partir de la version HPF de Onde24 (voir chapitre 4). Le noyau de calcul de l'application est une boucle parallèle qui ne nécessite des communications que pour chaque pas de temps ; il n'y a pas de communication à l'intérieur des boucles i et j. La figure 13.19 donne un aperçu de la structure générale du code HPF (à gauche) et du code SPMD généré par le compilateur. Or, les zones de données communiquées à chaque pas de temps (itérations de la boucle d'indice Time) sont toujours les mêmes (bien que les données elles-mêmes changent de valeurs). Ces zones de données sont représentées figure 13.21. Or, comme le montre la figure 13.20, la construction du pattern à communiquer (inspector) peut être faite à l'extérieur de la boucle, tandis que la communication (executor) reste à l'intérieur.

```
  Do Time = 1, NT
    [...]
!hpf$ independent
    Do j = 2, NP-3
!hpf$ independent
      Do i = 2, NP-3
        // Calcul de la solution
        // U(i,j,KP)
      Enddo
    Enddo
    [...]
  Enddo
```

(a) code HPF.

```
Do Time = 1, NT
  [...]
  // Déterminer la zone à communiquer
  // Communiquer les données
  Do j = local_lb_j, local_ub_j
    Do i = local_lb_i, local_ub_i
      // Calculer la solution
      // U(i,j,KP)
    Enddo
  Enddo
Enddo
```

(a) code SPMD.

FIG. 13.19 – *Gestion des communications pour Onde24.*

13.3.2 Détection de l'invariance

L'invariance d'une instruction contenant un appel de fonction sera déterminée par plusieurs conditions :

1. La fonction manipulée doit être **pure**, c'est-à-dire sans effets de bord. Cette condition nous permet de garantir que seuls les arguments de la fonction peuvent être lus et écrits (i.e. pas de variable globale de type common utilisée par exemple). Les fonctions d'entrées-sorties (qui écrivent dans des pseudo variables globales) seront donc également déclarées non-invariantes (car non pures). Notons

```
// Ensemble des données à communiquer (type dérivé)
call MPI_TYPE_VECTOR(count, blocklength, stride,
                       MPI_FLOAT, overlap, ierror)
call MPI_TYPE_COMMIT(overlap, ierror)
Do Time = 1, NT
  // Communiquer
  call MPI_SEND(U(1,1,KM), 1, overlap, dest, tag,
                 MPI_COMM_WORLD, ierror)
  Do j
    Do i
      // Calcul de la solution U(i,j,KP)
    Enddo
  Enddo
Enddo
// Libérer
call MPI_TYPE_FREE(overlap, ierror)
```

FIG. 13.20 – *Détermination de l'ensemble de données à communiquer.*

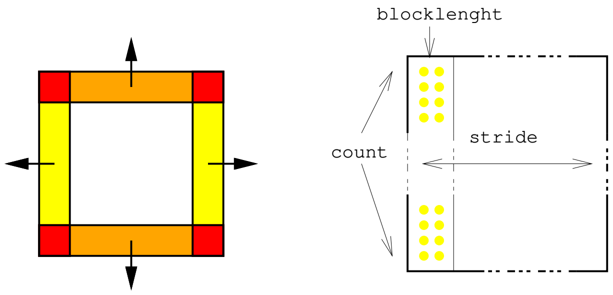


FIG. 13.21 – *Ensembles de données à communiquer dans Onde24.*

que toute expression classique peut être vue comme une fonction pure qui lit des paramètres (les opérandes) et retourne un résultat.

2. Tous les prédécesseurs pour cet appel de la fonction (c'est-à-dire les instructions définissant au moins un des arguments) doivent être invariants pour la boucle.
3. Les indices de bornes de boucles ne doivent pas apparaître syntaxiquement dans les arguments de la fonction.

Considérons l'exemple donné figure 13.22. L'appel à la procédure INC ne peut être déplacé car l'analyse de dépendances (interprocédurale) montrera qu'il existe un cycle de dépendances de cette instruction vers elle-même. L'appel à FOO sera par contre extrait des deux boucles car c'est une procédure sans effets de bords, avec des arguments qui sont définis à l'extérieur de la boucle. L'évaluation de la fonction SIN sera également déplacée d'un niveau de boucle. Enfin, l'utilisation d'un common par la fonction EXP interdit son déplacement.

```
PROGRAM FOO
 COMMON /C1/ M
 [...]
 M = 0
 DO I = 1, N
   DO J = 1, N
     call INC(K)
     call FOO(B,C,D)
     T1 = SIN(B*I*Π/2)
     T2 = EXP(T1)
     A(K) = T1 + T2
   ENDDO
 ENDDO
END

SUBROUTINE INC(I)
 INTEGER I
 I = I+1
END

SUBROUTINE FOO(X,Y,Z)
 INTEGER X,Y,Z
 X = (Y+Z)*(Y-Z)
END

INTEGER EXP (T)
 INTEGER T,EXP
 M = M+1
 EXP = T**M
 RETURN
END
```

(a) Code initial.

```
PROGRAM FOO
 COMMON /C1/ MAXO
 [...]
 call FOO(B,C,D)
 DO I = 1, N
   T1 = SIN(I*Π/2)
   DO J = 1, N
     call INC(K)
     T2 = MAXV(T1)
     A(K) = T1 + T2
   ENDDO
 ENDDO
END

[...]
```

(b) Code transformé.

FIG. 13.22 – *Déplacement d'appels de fonctions.*

13.4 Application de ces techniques en présence de tests

Les techniques de déplacement de code étudiées ci-dessus reposent sur l'analyse du graphe de dépendances. Ce dernier permet de synthétiser les informations utiles portées par le flot de données. Nous avons jusqu'ici supposé que le code manipulé ne contenait que des boucles imbriquées, ainsi que des instructions de type affectation comme dans [AK87]. L'information utile concernant le flot de contrôle était réduite à la connaissance de l'imbrication des boucles. Cette section se concentre maintenant sur l'application des techniques précédemment exposées dans le cas de boucles contenant des tests. En effet, la présence de tests à l'intérieur des boucles introduit de nouvelles contraintes sur la validité des transformations appliquées.

13.4.1 État de l'art

Les travaux d'Allen et Kennedy dans le domaine de la vectorisation puis de la parallélisation de programmes font appel à une technique appelée *if-conversion* [AKPW83, AK87, KM90], permettant de convertir des informations de type contrôle sous forme de dépendances de données. Leurs analyses se concentrent ensuite uniquement sur le graphe de dépendances de données. La *if-conversion* consiste simplement à associer à chaque instruction, un prédicat traduisant les conditions sous lesquelles l'instruction doit ou non être exécutée [WMmWHR93].

Afin de regrouper l'information concernant le flot de données et le flot de contrôle dans une seule et unique représentation de programmes Ferrante, Ottenstein et Warren introduisent dans [FOW87] le *Program Dependence Graph (PDG)* et l'utilisent dans le cadre de l'optimisation de programmes. Horwitz, Prins et Reps en confirment l'intérêt dans [HPR88]. En raison de sa complexité importante, le PDG est également à l'origine d'articles présentant des techniques permettant de le construire plus facilement [HMR93b, HMR93a, OE92].

Ballance, Maccabe et Ottenstein introduisent le *Program Dependence Web* [BMO90] qui est une combinaison du PDG et d'une forme en assignation unique (SSA) [CFR+89]. Contrairement au PDG, le PDW contient un modèle d'exécution. D'autres représentations de programmes plus ou moins compactes, et adaptées à différents types d'optimisation ont été proposées par la suite. On trouve notamment les *Value Flow Graph (VFG)* [SKR90], *Dependence Flow Graph (DFG)* [PBJ+91], *Gated Single Assignment (GSA)* [Hav93], *Value Dependence Graph (VDG)* [WCES94], *Simple Graph-Based Intermediate Representation (IR)* [CP95] ou plus récemment *Value Name Graph (VNG)* [BA98].

Notre travail étant axé principalement sur les boucles, l'information manipulée concernant le flot de contrôle est très réduite comparée à celle requise pour des optimisations travaillant sur un code non structuré. Nous souhaitons cependant pouvoir également utiliser le déplacement d'instructions invariantes en présence de tests. La suite de ce chapitre reposera sur l'utilisation de la *if-conversion* qui présente l'avantage de considérablement simplifier la complexité de la structure de donnée manipulée. Le passage des dépendances de contrôle aux dépendances de données nous permet également de ne pas devoir reformuler les algorithmes présentés ci-dessus. Cependant, similairement à Baxter et Bauer qui proposent dans [BB89] une variation de la vectorisation [AK87] fondée sur le PDG, les techniques présentées ici pourraient probablement être adaptées à d'autres représentations de programmes. Notons que l'utilisation de représentations de programmes telles que la if-conversion ou le PDG entraîne des problèmes de regénération du code.

13.4.2 Déplacement des instructions gardées

La présence de tests conditionnant l'exécution des instructions dans un nid de boucle rend plus complexe l'étude de l'invariance d'une instruction pour une boucle donnée. Si le résultat du test est identique pour toutes les itérations de la boucle au même titre que l'instruction contenue dans ce test, tous deux peuvent être évalués une seule fois. En revanche, si l'un ou l'autre (test ou instruction) n'est pas invariant pour la boucle, aucune transformation ne peut être appliquée.

L'utilisation de la *if-conversion* permet de gérer efficacement ce problème. Ainsi, si le prédicat gardant l'instruction est invariant au même titre que l'instruction (non-gardée) alors l'instruction gardée est invariante pour la boucle. L'instruction **et** le test associé peuvent donc être déplacés en dehors de la boucle. En revanche, si l'instruction non-gardée **ou** le prédicat ne sont pas invariants, l'instruction gardée ne peut être déplacée.

La figure 13.23 présente le mécanisme de *if-conversion*. Les gardes sont représentées sur la partie droite de la figure par l'ajout des prédicats après l'instruction sous la forme : (p_i). Le prédicat regroupe toutes les informations de contrôle portées par les tests conditionnant l'exécution d'une instruction. Supposons que p_1 soit invariant pour la boucle d'indice i, et p_2, p_3 non invariants. L'instruction gardée S_1 est invariante pour la boucle et peut donc être déplacée en dehors de celle-ci. Les instructions gardées S_2, S_3 et S_5 ne peuvent pas être extraites de la boucle à cause des prédicats non invariants. Enfin, S_6 n'est pas non plus invariante en raison de la présence d'un prédécesseur non invariant (et malgré l'absence de test).

```
DO I = 1, 100
  IF ( p1 )
    A = ...
    IF ( p2 )
      B = ...
    ELSE
      C = ...
    ENDIF
  ENDIF
  IF (x>I)
    D = ...
  ENDDO
  E = D
ENDDO
```

(a) format initial.

```
       DO I = 1, 100
(S1)     A = ...    : (p1)
(S2)     B = ...    : (p1 & p2 )
(S3)     C = ...    : (p1 & ¬ p2 )
(S4)     p3 = x>I
(S5)     D = ...    : (p3)
(S6)     E = D  : (True)
       ENDDO
```

(b) instructions gardées.

FIG. 13.23 – *Conversion des tests.*

La transformation des dépendances de contrôle sous forme de dépendances de données telle que la présentent les auteurs de [AKPW83] nous permet donc d'étendre le déplacement d'instructions invariantes aux nids de boucles contenant des tests. En pratique, l'instruction gardée est simplement considérée comme une instruction classique contenant un opérande qui se trouve être le prédicat. Une instruction responsable de l'évaluation du prédicat est introduite pour chaque test. Les dépendances de flot entre cette instruction et toutes les instructions gardées par le prédicat garantissent que les instructions gardées ne peuvent pas être déplacées au delà de l'évaluation du prédicat (qui est devenu un prédécesseur et intervient donc dans la décision concernant la propriété d'invariance).

13.4.3 Respect de la sémantique du programme

L'objectif de ces travaux est d'extraire des instructions invariantes des boucles afin d'améliorer les performances du programme. L'approche utilisée en PRE interdit le déplacement de code dès que celui-ci ajoute un calcul sur un chemin du graphe de flot de contrôle (voir la section 12.2). Or, cette restriction limite considérablement le nombre d'expressions pouvant être extraites des boucles. Bodik, Gupta et Soffa ont montré par leurs expériences dans [BGS98] que 75% des expressions invariantes ne peuvent être déplacées par la PRE. Nous souhaitons donc pouvoir augmenter le domaine d'application de nos techniques et permettre plus de déplacement de code invariant en autorisant notamment l'extraction de calculs invariants de plusieurs boucles à la fois. Cependant, il est important de respecter la sémantique du programme considéré. La technique utilisée par les auteurs de [Ste96] et [BGS98] consiste à dupliquer des morceaux de code afin de transformer les calculs *partiellement redondants* (non redondants sur tous les chemins du graphe) en calculs *complètement redondants*. Si cette technique permet d'éliminer tous les calculs partiellement redondants, elle entraîne une importante augmentation de la taille du programme. Le contexte (plus restreint) dans lequel nous nous plaçons permet de restructurer le contrôle sans introduire une trop forte expansion de code.

Modification de la structure de contrôle

Le déplacement d'instructions invariantes présenté aux sections 13.1 et 13.2 peut en effet entraîner l'évaluation d'une expression qui ne l'était pas dans la version originale du code. Le schéma général d'une boucle est représenté sur la partie gauche de la figure 13.24. A chaque itération de la boucle (y compris la première), un test (P) décide de l'exécution ou non du corps de boucle. Le calcul invariant $a + b$ ne peut être extrait de la boucle par la PRE car cette transformation ajoute une calcul sur un chemin du graphe (Front-P-Back). Une simple transformation de la structure du code telle que celle présentée sur la partie droite de la figure 13.24 permet de contourner le problème. Un test vérifiant que le domaine d'itération n'est pas vide est ajouté, et un nouveau nœud (en pointillé sur la figure) est inséré pour y placer les éventuels calculs invariants.

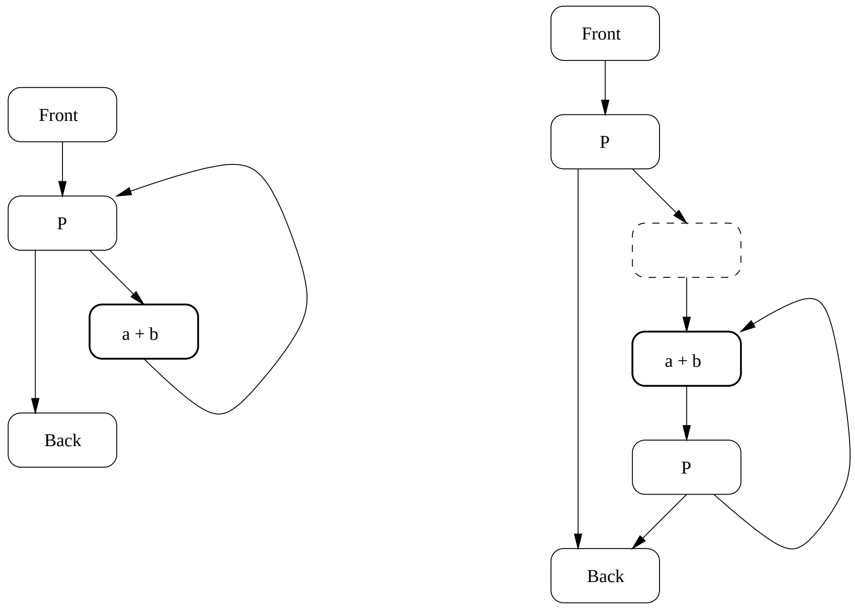


FIG. 13.24 – *Insertion d'un test vérifiant l'existence d'une itération.*

Cependant, cette transformation du graphe de contrôle est insuffisante pour permettre le déplacement d'un calcul de plusieurs niveaux de boucles. Or, ce sont les déplacement les plus bénéfiques puisqu'ils permettent de gagner $N_1 \times N_2 \ldots \times N_k$ opérations (avec N_i le nombre d'itérations pour chacune des k boucles). La figure 13.25 représente le graphe de contrôle obtenu par la technique précédente pour deux boucles imbriquées entre elles. Le corps des boucles est schématisé par un trait plus épais. Le calcul invariant $a+b$ peut être déplacé en dehors de la boucle interne (P) mais ne peut sortir de la boucle externe (Q). Il existe en effet un chemin (FrontQ-Q- -FrontP-P-BackP-Q-BackQ) interdisant ce déplacement.

Technique utilisée Supposons que le résultat du test P soit identique pour toutes les itérations de la boucle externe (Q) ; les bornes de la boucle ne dépendent donc pas de l'indice de la boucle englobante. Nous cherchons à évaluer le calcul invariant $a + b$ à l'extérieur des boucles, tout en évitant de le calculer dans le cas où le domaine d'itérations d'une des boucles est vide. Pour cela, nous effectuons un déplacement « groupé » du calcul invariant et du test. Le test est simplement dupliqué au niveau du nœud qui accueille le calcul invariant (voir figure 13.26). Cela permet de garantir que le comportement du programme est véritablement inchangé.

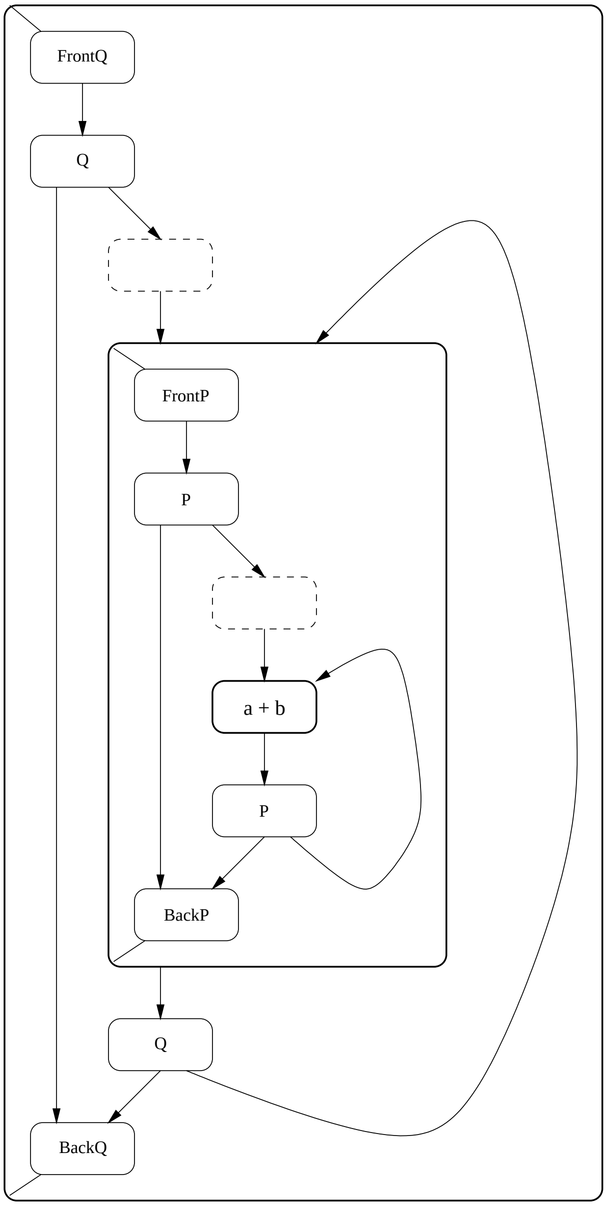


FIG. 13.25 – *Insertion de tests vérifiant l'existence d'une itération.*

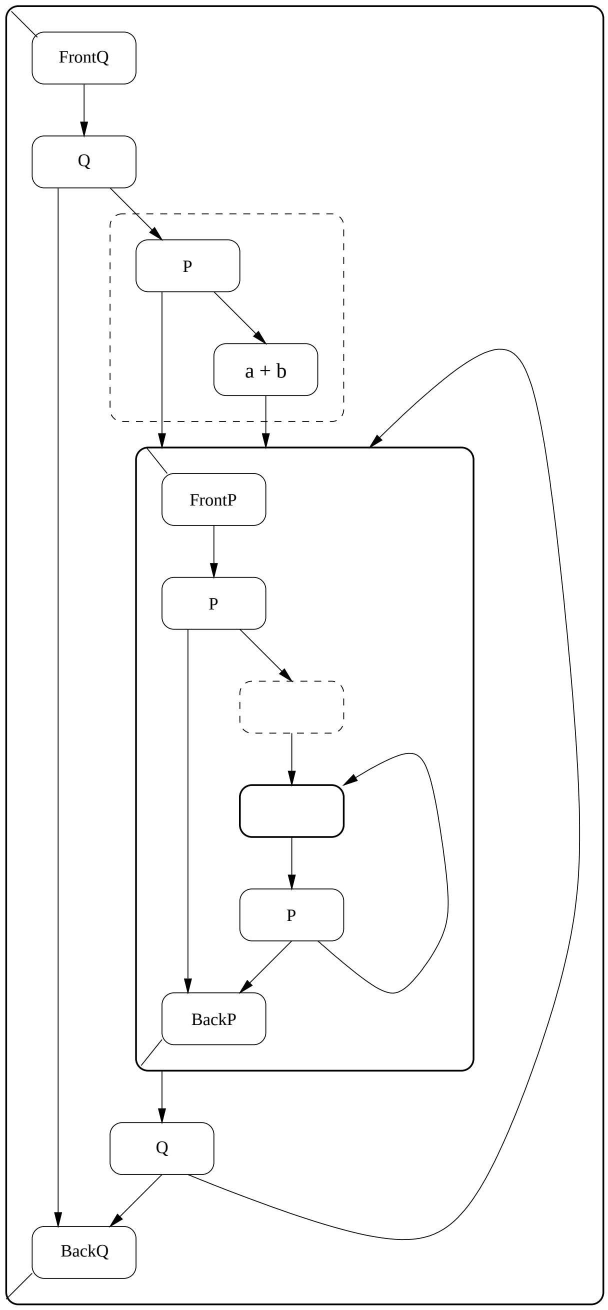


FIG. 13.26 – *Technique utilisée en présence de tests.*

Cas particulier des boucles parfaitement imbriquées Un nid de boucles parfaitement imbriquées possède une propriété intéressante : l'exécution de toutes les instructions présentes dans le corps de boucle est conditionnée par les tests sur toutes les bornes de boucles. Cette caractéristique permet d'appliquer des transformations visant à inverser l'ordre des tests et des boucles. Ainsi, lorsque les dépendances de données le permettent, un test concernant une boucle de profondeur x peut être placé autour d'une autre boucle de profondeur $x' < x$. La figure 13.27 illustre ce phénomène en reprenant la situation de la figure 13.25. Le nid de boucles étant parfaitement imbriqué, les nœuds FrontP et BackP ont été supprimés. Si le résultat du test P est identique pour toutes les itérations de la boucle externe (Q), il est possible d'intervertir l'ordre de la boucle Q et du test P (le test P est placé autour de la boucle Q). Ainsi, le calcul invariant $a + b$ est extrait des deux boucles sans duplication du test P.

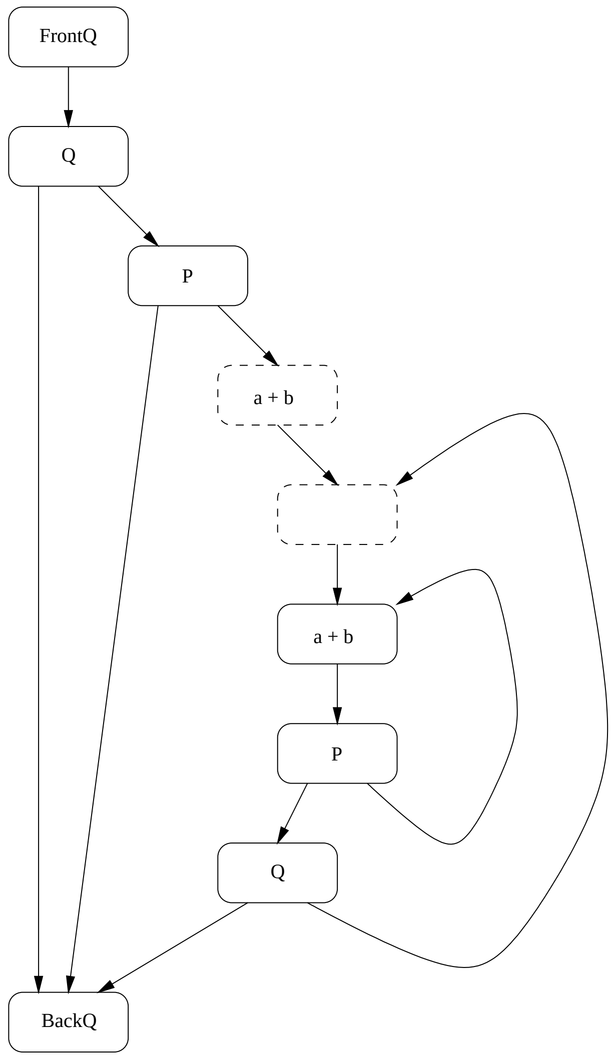


FIG. 13.27 – *Cas particulier des boucles parfaitement imbriquées.*

Application de la if-conversion.

Pour appliquer la modification de la structure de contrôle proposée ci-dessus, nous utilisons les techniques de if-conversion présentées précédemment. Nous introduisons pour chaque boucle un test vérifiant si son domaine d'itération est vide ou non ; toutes les instructions du corps de boucle deviennent ainsi gardées par ce test. Donc, si le prédicat correspondant est invariant pour les boucles englobantes (la condition sur les bornes de boucles est évaluable au plus tôt), les instructions pourront être remontées de plusieurs niveaux de boucles. En revanche, si le prédicat n'est pas invariant, les instructions seront remontées, au mieux, d'un seul niveau de boucle. Cette restructuration du contrôle permet d'augmenter l'applicabilité du déplacement d'instructions invariantes, sans pour autant nuire au comportement du programme.

Remarque 13.3
Des analyses suffisamment fines (fournies par PIPS) permettent de prouver dans bien des cas que le domaine d'itération d'une boucle n'est pas vide. Pour celles-ci, l'ajout du test n'est pas nécessaire. Considérons l'exemple donné figure 13.28. Le calcul interprocédural des préconditions ajouté à l'évaluation partielle (pour le modulo) permettent de prouver que la première boucle n'est jamais *vide, tandis que la seconde boucle est* toujours *vide. Les préconditions sont ici affichées par* `PIPS` *sous forme de commentaires dans le code.*

```
      PROGRAM MAIN
      INTEGER N,P
      PARAMETER (N=2)
      [...]
      P = 1
      [...]
      CALL FOO(N,P,...)
      END

      SUBROUTINE FOO(N,P,...)
      INTEGER N,M,P

      M = MOD(N,4)
C  P(M) N==2,M==2,P==1
      DO I = P, M
         [..]
      ENDDO
C  P(I,M) I==3,N==2,M==2,P==1
      DO I = M+1, N, 4
         [...]
      ENDDO

      END
```

FIG. 13.28 – *Boucles non vides.*

Grâce à cette technique, une instruction n'est exécutée que si elle l'était aussi dans le programme original. La figure 13.29 donne une illustration de cette technique. Le prédicat correspondant à la boucle d'indice I_i est noté b_i. L'instruction évaluant le prédicat b_2, qui est invariante pour la boucle d'indice I_1, est déplacée ; Cela permet à l'instruction S_1, gardée par b_1 et b_2, de remonter à son tour de deux niveaux de boucles. L'instruction évaluant le prédicat b_3 n'étant pas invariante, l'instruction S_2 reste à l'intérieur de la boucle d'indice I_1 (mais est tout de même déplacée en dehors de la boucle d'indice I_3).

Re-génération du code source. Pour des machines ne comportant pas de mécanisme d'exécution conditionnelle, ou dans le cadre de transformations de programmes source-à-source, il est nécessaire de reconstruire les tests à partir des gardes. S'il est très facile de régénérer le code par duplication des tests pour chacune des instructions, il est en revanche plus difficile de limiter le surcoût introduit par l'augmentation des instructions de contrôle. Cependant, si le code initial ne présente pas d'opportunités de

```
        b1 = 1.leq.N1
        DO I1 = 1, N1
          b2 = 1.leq.N2  : (b1)
          DO I2 = 1, N2
(S1)        A = ...    : (b1 & b2)
            ...
          ENDDO
          b3 = I1.leq.N2  : (b1)
          DO I3 = I1, N3
(S2)        B = ...    : (b1 & b3)
            ...
          ENDDO
        ENDDO
```

(a) code initial.

```
        b1 = 1.leq.N1
        b2 = 1.leq.N2  : (b1)
(S1)    A = ...    : (b1 & b2)
        DO I1 = 1, N1
          DO I2 = 1, N2
            ...
          ENDDO
          b3 = I1.leq.N2  : (b1)
(S2)      B = ...    : (b1 & b3)
          DO I3 = I1, N3
            ...
          ENDDO
        ENDDO
```

(b) code transformé.

FIG. 13.29 – *Respect de la sémantique du programme.*

déplacement de code, le travail de reconstruction est quasiment sans problème. En effet, aucun déplacement n'ayant été effectué, les instructions gardées par les mêmes prédicats sont « limitrophes » (elles sont donc mises dans le corps d'un seul et même test). Par ailleurs, si à l'opposé, le nombre d'instructions déplacées est important, le surcoût introduit par la duplication des tests devient faible comparé au gain apporté par la réduction du nombre d'opérations dans le corps de la boucle. Enfin, notre technique permettant d'extraire des instructions gardées hors des boucles, le nombre de tests effectués peut être globalement réduit (puisque les tests sont effectués moins souvent).

Les problèmes engendrés par le surcoût de la if-conversion dans le domaine de la vectorisation ont été dénoncés à plusieurs reprises. Des études concernant la traduction des instructions gardées sous forme d'un code contenant des tests peuvent être trouvées notamment dans [WMmWHR93]. L'utilisation d'une représentation de programme plus évoluée (comme par exemple le PDG) permettrait de limiter ce type de surcoût, au prix d'une plus grande complexité de manipulation.

Inversion test/boucle Si au moment de reconstruire le code source, toutes les instructions « utiles » (effets `OUT` non vides) du corps d'une boucle ont une garde contenant un prédicat commun p, il est possible d'insérer le test IF(p) autour de la boucle considérée. Ceci permet d'éviter la duplication intempestive des tests dans le cas des boucles parfaitement imbriquées (voir ci-dessus).

13.4.4 Relâchement des contraintes

L'introduction de tests pour conserver la sémantique du programme est indispensable pour les instructions donnant lieu à de véritables écritures mémoires. En effet, de nombreux calculs intermédiaires utilisés uniquement au sein du nid de boucles, souvent appelés *privés*, peuvent être déplacés sans mettre en péril le comportement du programme. Ainsi, il est dans certaines conditions possible de relâcher les contraintes pour les calculs temporaires. Seules les véritables contraintes correspondant au contrôle initial (et non celles introduites par nos soins et testant les bornes de boucles) doivent absolument être respectées.

Le relâchement des contraintes sera donc appliqué aux instructions qui sont des affectations à des variables privatisables, utilisées par aucune instruction à l'extérieur de la boucle. C'est le cas notamment des instructions qui seront introduites par la phase de réassociation présentée au chapitre 14. Cela permet de relâcher les contraintes pour des instructions qui ne sont utilisées que dans le cadre de calculs locaux. Pour rechercher de ce type d'instructions, nous testons l'existence d'*effets out* [Cre96] (voir chapitre 5).

Dans le cadre de la if-conversion, le relâchement des contraintes se fait par la simplification des gardes. Suite à la simplification des prédicats, certaines instructions peuvent être déplacées et donc exécutées alors que le domaine d'itération de la boucle initiale est vide. L'instruction est donc exécutée inutilement, puisque le programme initial ne le faisait pas. Ce changement de comportement du programme est à étudier de deux points de vues :

- Le nombre d'opérations effectuées par le programme est potentiellement augmenté. Cependant, dans le contexte du calcul scientifique, le gain apporté par le déplacement de calculs à l'extérieur des boucles nous semble en général profitable et outrepasse le risque d'augmenter le nombre d'opérations pour des boucles dont le corps n'est jamais exécuté. Cependant, cette transformation non conservative est bien évidemment de nature heuristique et comporte une part de prise de risque. Cette approche spéculative a été discutée par les auteurs de [GDF98].

- L'exécution de nouvelles instructions par la version transformée d'un programme présente un risque pour la stabilité du programme. Le relâchement des gardes ne sera donc appliqué que dans le cadre des instructions « sans effets de bord », telles que les opérations classiques (additions, multiplication, etc.) ; il faudra également vérifier que ni le logiciel ni le matériel ne provoquent d'interruptions en cas de dépassement de capacité.

Notons que certaines normes de manipulation des nombres flottants (notamment le standard IEEE) font que des langages (comme Fortran) renvoient NaN[69] *ou* ∞ *lorsque les opérandes d'une opération sont invalides (par exemple dans le cas d'une division par 0) et peuvent ne pas générer d'exceptions. Donc, dans la mesure où les écritures mémoires sont toujours faites de manière conservative, le déplacement de la division (ou encore la racine carrée, le logarithme, etc.) ne présente pas réellement de risque (pour cette classe de langages) ; le comportement global du programme reste inchangé, même si des calculs inutiles (et « erronés ») peuvent apparaître.*

13.5 Conclusion

Notre approche au cours de ces travaux a été de concentrer nos efforts sur l'optimisation des boucles. Contrairement aux techniques classiques de PRE, nous souhaitons en effet permettre le déplacement d'un plus grand nombre d'opérations tout en respectant le comportement du programme, avec le risque cependant d'augmenter le nombre d'opérations lorsque le corps de la boucle est vide. Nous pensons que cette approche non conservative est bénéfique dans la plupart des cas rencontrés en pratique. D'autre part, les techniques présentées sont des transformations de haut niveau permettant de déplacer des instructions invariantes contenant des accès à des éléments de tableaux et des appels de fonctions. Les techniques de réassociation des opérandes dans le but de former des sous-expressions invariantes, présentées au chapitre 14, viennent en complément de ce travail. La simplification du graphe de dépendances (algorithme 5) et la génération de code (algorithme 6) ont été implémentés ; les détails sont données au chapitre 16. Le mécanisme de traitement des boucles contenant des tests n'a cependant pas été implémenté car les instructions gardées ne sont pas supportées dans `PIPS`.

69. Not A Number.

Chapitre 14

Découpage en sous-expressions invariantes

La réassociation des opérandes d'une expression pour former des sous-expressions invariantes est rendue possible par les propriétés d'associativité et de commutativité des opérateurs. Nous souhaitons sélectionner parmi toutes les façons d'écrire une expression, celles qui offrent le plus d'opportunités pour le déplacement de code invariant. Cependant, l'explosion combinatoire due à l'utilisation des propriétés algébriques des opérateurs interdit un parcours exhaustif de l'espace des solutions.

Notre approche consiste à parcourir les expressions complexes présentes dans le code source pour en extraire des sous-expressions évaluables à l'extérieur des boucles courantes. La construction des sous-expressions repose sur un principe de tri des références selon un critère déterminé : les références sont regroupées en fonction de la position (dans l'imbrication de boucles) à partir de laquelle elles peuvent être évaluées.

L'efficacité de la réassociation des *expressions* dépend de la position pertinente ou non des *instructions* dans le nid de boucle considéré. En effet, une expression ne peut pas remonter au dessus de toute instruction définissant l'un au moins de ses opérandes. C'est pourquoi nous supposons qu'une phase de déplacement d'instructions invariantes a été appliquée au préalable ; cela permet d'assurer que les instructions sont déjà placées au mieux. Cependant, deux paramètres importants sont à considérer :

- Bien que nous proposions un nouvel algorithme de déplacement d'instructions invariantes au chapitre 13, les techniques de réassociation de code présentées peuvent fonctionner après l'application de n'importe quel algorithme capable de déplacer des **instructions**.
- Ce travail ne requiert pas véritablement l'usage préalable d'un algorithme complet de déplacement d'instructions invariantes. En effet, l'information utile pour la réassociation se limite à la « meilleure » position de chaque instruction dans l'imbrication de boucles. Or cette position peut être connue sans nécessairement déterminer l'ordonnancement final des instructions entre elles. Il suffirait donc d'utiliser les techniques proposées au chapitre 13, sans aller jusqu'à la génération effective du code.

14.1 Position d'une instruction

Les travaux de Briggs et Cooper [BC94] font référence à la notion de *rang* d'une expression. Leur objectif est qu'une expression *invariante* ait un rang plus petit qu'une expression *variante*. Pour calculer le rang des variables, les auteurs passent par une représentation en assignation unique, puis effectuent un parcours[70] du graphe de flot de contrôle. Le premier bloc prend le rang 1, le rang du second bloc est 2,

70. reverse post-order

etc. Un rang est attribué à chaque expression dans un bloc en suivant les trois règles suivantes :

1. Le rang d'une constante est 0.
2. Le rang d'un ϕ-node[71] ou d'un appel de fonction est celui du bloc courant.
3. Une expression reçoit le rang de son opérande de rang maximal.

Nos travaux étendent cette technique à une imbrication de boucles quelconque. En effet, la notion de rang telle que la présentent Briggs et Cooper n'est utilisable que dans le cadre d'une imbrication de boucles parfaite. La notion de rang ne permet pas de distinguer deux boucles de même profondeur, mais placées dans deux imbrications différentes. D'autre part, nous ne travaillons pas dans un cadre d'assignation unique, et donc nous devons tenir compte de l'existence de plusieurs définitions possibles pour chaque opérande.

Dans la suite de ce chapitre, nous appellerons $\mathcal{A}$ l'arbre définissant l'imbrication des boucles pour la portion de programme examinée. Le critère de réassociation que nous utiliserons repose sur la position des instructions dans cet arbre. Soit $\mathrm{Pos}(S)$ la position d'une instruction S dans $\mathcal{A}$.

Nous cherchons à construire une relation entre chaque instruction S et sa position dans $\mathcal{A}$. L'imbrication des boucles étant quelconque, une notation reposant sur une liste de boucles $L_1, \ldots, L_n$ n'est pas satisfaisante. Nous utiliserons donc une notation permettant d'identifier de manière unique tout imbrication de boucles. Chaque boucle sera nommée de manière à pouvoir identifier directement à partir de son nom, toutes les boucles qui l'entourent. Nous utilisons pour cela l'index de Dewey ; les règles de nommages sont les suivantes :

- une boucle fictive $L_\emptyset$ définissant la racine de $\mathcal{A}$ est ajoutée autour du code initial. Une instruction n'appartenant à aucune boucle aura donc la position $\emptyset$.
- Soit une boucle $L_{\boldsymbol{X}}$ avec $\boldsymbol{X} = x_1x_2\ldots x_n$ un préfixe quelconque défini pour $x_i \in \{\emptyset \cup [a-z]\}$. Les boucles contenues dans $L_{\boldsymbol{X}}$ (et non imbriquées entre elles) seront notées $L_{\boldsymbol{X}a}$, $L_{\boldsymbol{X}b}$, $L_{\boldsymbol{X}c}$, etc. La boucle $L_{\boldsymbol{X}y_1}$ est placée syntaxiquement devant la boucle $L_{\boldsymbol{X}y_2}$ dans le corps de boucle de $L_{\boldsymbol{X}}$ si $y_1 \succ y_2$ (avec $\succ$ l'ordre lexicographique).
- Par construction, la boucle $L_{x_1x_2\ldots x_n}$ sera donc imbriquée successivement dans les boucles L_{x_1}, $L_{x_1x_2}$, $\ldots$, $L_{x_1x_2\ldots x_{n-1}}$.

Les notations utilisées pour représenter les boucles et leur indices sont telles que la boucle $L_{\boldsymbol{X}}$ aura pour indice $i_{\boldsymbol{X}}$. La position d'une instruction placée dans le corps d'une boucle $L_{\boldsymbol{X}}$ sera donc la chaîne de caractères alphanumériques $\boldsymbol{X}$. Les définitions suivantes seront utilisées par la suite :

Définition 14.1
Soit $X = x_1x_2\ldots x_n$ et $X' = x'_1x'_2\ldots x'_{n'}$ deux positions quelconques. La relation d'ordre sur les positions est définie par l'ordre lexicographique telle que :

$$X = X' \text{ si } n = n' \text{ et } \forall i \in [1,n], x_i = x'_i. \tag{14.1}$$

$$X \succ X' \text{ si } \begin{cases} & \text{si } n > n' \text{ et } \forall i \in [1,n'], x_i = x'_i, \\ \textit{ou} & \text{si } n \leq n' \text{ et } \exists k \in [1,n] \text{ tel que } \forall i < k, x_i = x'_i \text{ et } x_k \succ x'_k. \end{cases} \tag{14.2}$$

Définition 14.2
Soit $\{X_1, \ldots X_n\}$ un ensemble de positions. La fonction $\max_\succ$ sur un ensemble de positions est définie telle que

$$\max_\succ\{X_1, \ldots, X_n\} = X_k \textit{ si, et seulement si, } \forall i \in [1,n], i \neq k, X_k \succ X_i.$$

71. technique utilisée pour le passage en SSA lorsqu'il existe deux définitions d'une même variable

Définition 14.3
Soient deux instructions S et S' telles que Pos$(S) = x_1 x_2 \ldots x_n$ et Pos$(S') = x'_1 x'_2 \ldots x'_{n'}$. La position commune entre deux instructions est définie (sous la forme d'un préfixe) telle que

$$PCom(S, S') = x_1 x_2 \ldots x_k \text{ si } \forall i \leq k, x_i = x'_i \text{ et } x_{k+1} \neq x'_{k+1}.$$

Un exemple de code utilisant la notation proposée est présenté figure 14.1. Deux instructions telles que S_4 et S_5 se trouvant dans la même imbrication de boucles se verront attribuer la même position ($\text{Pos}(S_4) = \text{Pos}(S_5) = ab$). Cette notion de *position commune* est similaire à la notion de *niveau commun* utilisée dans le cadre du calcul de dépendances par Allen et Kennedy ; l'information recherchée est plus complète mais ne présente pas plus de difficulté à récupérer.

```
Do i_a = lb_a , ub_a
   Do i_aa = lb_aa , ub_aa
      Do i_aaa = lb_aaa , ub_aaa
         (S_1)
      Enddo
      Do i_aab = lb_aab , ub_aab
         (S_2)
      Enddo
   Enddo
   (S_3)
   Do i_ab = lb_ab , ub_ab
      (S_4)
      (S_5)
   Enddo
Enddo
```

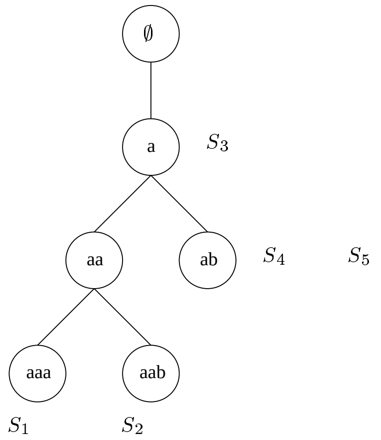


FIG. 14.1 – *Notations utilisées pour décrire l'arborescence*

14.2 Critère de réassociation

Le critère de réassociation doit permettre de distinguer les opérandes d'une opération qui sont « disponibles » à l'extérieur des boucles de ceux qui ne le sont pas. L'information recherchée est en effet l'instant à partir duquel un opérande donné est disponible pour l'évaluation d'un certain calcul.

L'exécution d'une instruction S n'est possible que si toute instruction $S' \in Pred(S)$ est exécutée **avant** S. De même, l'évaluation d'une sous-expression contenant plusieurs références ne peut être démarrée qu'après l'exécution de toute instruction définissant au moins l'une de ces références. Par ailleurs, toutes les définitions vivantes d'une même référence doivent être prises en compte pour déterminer le critère de réassociation. Enfin, la présence syntaxique des indices des boucles dans les opérandes doit être aussi prise en compte.

Définition 14.4
*Soit O un opérande quelconque contenue dans une instruction S. La **disponibilité** de cet opérande (notée $\mathcal{D}$) est définie telle que*

$$\mathcal{D}(O) = \max_{\succ}\{\max_{\succ}\{\boldsymbol{X} | i_{\boldsymbol{X}} \in Ind(O)\}, \max_{\succ}\{PCom(S, S') | S' \in Def(O)\}\}$$

Plusieurs paramètres importants sont pris en considération dans le calcul de la disponibilité d'un opérande :

- La disponibilité d'un opérande est calculée en fonction de la position de **toutes** les instructions le définissant.
- La disponibilité d'un opérande est contrainte par la position de la définition **la plus profonde** dans l'imbrication de boucle.

– Seule la **partie commune** de l'imbrication est prise en compte pour le calcul de $\mathcal{D}$.
– Les indices syntaxiquement présents dans un opérande limitent sa disponibilité.

La figure 14.2 illustre la disponibilité des cinq opérandes d'une addition. La disponibilité de l'opérande $A(i_a)$ est fixée à a en raison de la présence syntaxique de l'indice i_a. Notons également que la disponibilité de N_3 est calculée à partir de la position des **deux** instructions S_3 et S_7. Enfin, la disponibilité de l'opérande $B(i_a, i_{aba})$ est a bien que sa définition ait la position aa, puisque seule la partie commune aux positions de l'opérande et de sa définition est prise en compte.

```
(S1)  N1 = [...]
(S2)  N2 = [...]
(S3)  N3 = [...]
      Do ia = lba , uba
         Do iaa = lbaa , ubaa
(S4)         B(ia,iaa) = [...]
         Enddo
         Do iab = lbab , ubab
(S5)         C(ia,iab) = [...]
             Do iaba = lbaba , ubaba
(S6)           [...] = A(ia) + N1 + C(ia,iab) +
   &                   B(ia,ia) + N2 + N3 + D(ia,iab,iaba)
             Enddo
         Enddo
(S7)     N3 = [...]
       Enddo
```

Opérande	$\mathcal{D}$
A(i_a)	a
N_1	$\emptyset$
C(i_a,i_{ab})	ab
B(i_a,i_a)	a
N_2	$\emptyset$
N_3	a
D(i_a,i_{ab},i_{aba})	aba

FIG. 14.2 – *Disponibilité des opérandes*

14.3 Construction des sous-expressions invariantes

La disponibilité des opérandes est l'information nécessaire à l'extraction de sous-expressions invariantes. Il s'agit maintenant de mettre en place les techniques permettant de construire les nouvelles instructions à partir de cette information. Le cas d'une opération (un seul type d'opérateur) sera dans un premier temps étudié, puis nous étendrons la méthode à des expressions de taille plus importante.

14.3.1 Découpage d'une opération associative-commutative

Proposition 14.1
Soit $\oplus$ une opération ***associative-commutative*** *appartenant à une instruction S et contenant n opérandes (notés O_i).*

– *Les opérandes de $\oplus$ sont triés en fonction de leur* ***disponibilité*** *$D(O_i)$ dans l'ordre croissant.*
– *Une sous-expression est créée pour chaque groupe d'opérandes de disponibilité identique.*
– *Chaque sous-expression est affectée à une variable temporaire qui devient un nouvel opérande pour le groupe de disponibilité immédiatement supérieure.*

La construction des sous-expressions invariantes fonctionne par un tri sur les opérandes utilisant le critère de réassociation $\mathcal{D}$ défini ci-dessus. Tous les opérandes ayant la même disponibilité sont regroupés et affectés à des variables temporaires et forment ainsi des expressions telles que tous les opérandes sont disponibles à la même position dans l'arbre (même disponibilité). Une nouvelle instruction est ainsi créée pour chacune des valeurs de disponibilités différentes obtenues pour les opérandes, sauf pour la disponibilité correspondant à la position de l'instruction S (cette sous-expression ne pouvant pas être déplacée).

Remarque 14.1
Ce processus de découpage doit tenir compte des propriétés des opérateurs manipulés. Par exemple, pour respecter l'arité de l'opération, les additions ne contenant qu'un seul terme ne sont pas générées. Le terme isolé sera groupé avec les termes de disponibilité supérieure.

Revenons à l'exemple de la figure 14.2. Le découpage de l'addition (contenant 6 termes) pour en extraire des sous-expressions invariantes est fait à partir d'un tri sur la disponibilité des opérandes. Une première sous-expression invariante contenant l'opération $t_1 = N_1 + N_2$ est créée, pour la disponibilité $\emptyset$. La seconde instruction, de disponibilité a, est l'évaluation de $t_2 = t_1 + A(i_a) + B(i_a, i_a) + N_3$. L'instruction $t_3 = t_2 + C(i_a, i_{ab})$, de disponibilité ab est crée. Enfin, l'opérande $D(i_a, i_{ab}, i_{aba})$ de disponibilité aba reste dans l'instruction originale et est ajouté à t_3.

Une étude de l'optimalité de ce découpage des expressions est présentée en annexe A. Sous l'hypotèse que les domaines d'itérations des boucles ne sont pas vides, nous cherchons à montrer que le découpage par réassociation des opérandes selon leur disponibilité permet de minimiser globalement le nombre d'opérations à évaluer.

14.3.2 Manipulation d'expressions complexes

La technique présentée pour une opération n-aire peut également s'appliquer directement à une expression « complexe », c'est-à-dire une expression contenant n opérations associatives-commutatives ou non (représentée par un arbre de profondeur quelconque).

Il existe plusieurs avantages à manipuler directement les expressions complexes plutôt que de travailler sur une forme atomisée des expressions :

- La conservation de la structure complexe des expressions est importante pour appliquer d'autres transformations de haut niveau par la suite, comme par exemple la recherche des *fma*.
- Le code ne contient pas systématiquement des instructions invariantes. En revanche, le programmeur n'a pas forcément cherché à extraire explicitement les **sous-expressions** invariantes. C'est le cas dans la majorité des codes scientifiques qui ne sont pas mal écrits mais sont plutôt non optimisés ou se contentent de suivre les formules mathématiques modèlisant le problème étudié.

  La réassociation à partir d'une forme atomisée des expressions, de manière à ne manipuler que des opérations n-aires, pose des problèmes pour le calcul de la disponibilité ou du rang dans [BC94]. En effet, ce dernier s'appuyant sur la disponibilité des prédécesseurs, il est nécessaire d'appliquer une phase de déplacement d'instructions invariantes dans un premier temps ; le travail de réassociation ne vient qu'ensuite.

  La manipulation d'expressions complexes permet dans ce cas d'appliquer immédiatement (sans transformation préalable) la réassociation, puisque l'information de disponibilité sera propagée au fur et à mesure de la progression dans l'expression.
- Dans l'optique d'un travail source-à-source « contrôlé » permettant à l'utilisateur de suivre la transformation appliquée (pour vérifier les problèmes de stabilité numérique par exemple), il est préférable de ne modifier le code initial **que** lorsque cela est profitable. L'atomisation du code nuit en effet très souvent à la lisibilité du programme.

L'extraction des sous-expressions invariantes à partir d'une expression complexe se fait de manière ascendante. L'information de disponibilité des opérandes est propagée progressivement vers la racine de l'arbre. Une nouvelle expression peut être construite :

- pour un opérateur **associatif et commutatif** avec chaque groupe d'opérandes de même disponibilité.
- pour un opérateur **associatif et non commutatif** avec chaque groupe d'opérandes *consécutifs* de même disponibilité.
- pour un opérateur **non associatif et non commutatif** si tous les opérandes ont une disponibilité égale (en particulier les opérateurs unaires comme sin, log, exp, etc.)

Il n'est pas utile de créer une nouvelle instruction dans tous les cas ; tous les opérandes et toutes les opérations de l'expression complexe ayant une disponibilité égale à la position de l'instruction courante formeront au final la nouvelle instruction :

- Si tous les opérandes d'une opération ont une disponibilité inférieure à la position de l'instruction courante, l'opérateur est supprimé de l'arbre. Il est remplacé par la variable temporaire (nouvel opérande) introduite pour le groupe d'opérandes de niveau maximum (parmi tous les groupes d'opérandes construits pour cette opération). La disponibilité de la variable temporaire est bien évidemment connue.
- En revanche, si il existe au moins un opérande ayant une disponibilité égale à la position de l'instruction courante, l'opération est conservée dans l'arbre, et sa disponibilité est maximale (égale à la position de l'instruction courante).

Exemple

La figure 14.3 illustre le découpage d'expression complexes. Chaque groupe d'opérandes de disponibilité identique forme une nouvelle sous-expression affectée à une variable temporaire (par exemple t_1 sur la figure). Celle-ci est ajoutée comme opérande au groupe de disponibilité immédiatement supérieure (ici t_1 est un nouvel opérande pour la multiplication qui sera ensuite affectée à t_2). Si tous les opérandes d'une opération ont une disponibilité inférieure à celle de l'instruction courante, l'opération est affectée à une variable temporaire (c'est le cas de la multiplication affectée à t_2 dans cet exemple). En revanche dans le cas contraire, la structure de l'expression est conservée (cas de la seconde multiplication qui n'est pas affectée à une variable temporaire).

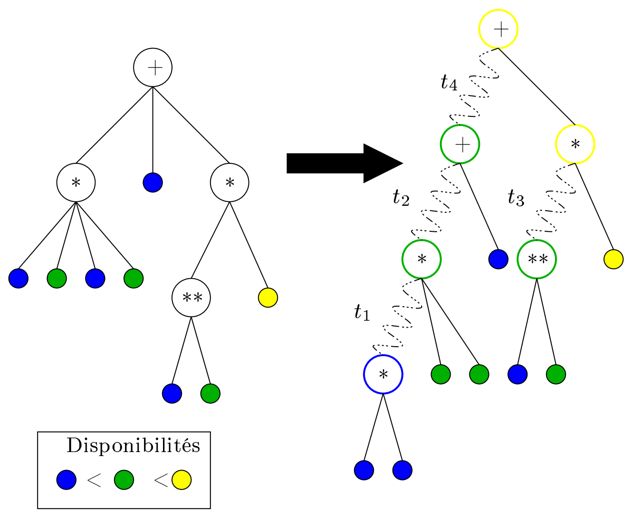


FIG. 14.3 – *Manipulation d'expressions complexes.*

Méthode utilisée

Les algorithmes 7 (fonction `Extraire_sous_expressions`) et 8 (fonction `Eval_disponibilité`) montrent de manière synthétique la méthode utilisée. Les appels récursifs autorisent un parcours de bas en haut. Les informations sur la disponibilité des opérandes et des opérateurs sont propagées au fur et à mesure de la remontée dans l'arbre.

L'algorithme 7 effectue une atomisation parfois excessive. En effet, celui-ci crée une nouvelle instruction pour toute sous-expression déplaçable (tout en conservant la partie non-invariante intacte). Or, une sous-expression « complexe » (i.e. contenant plus d'une opération) invariante peut être déplacée d'un seul bloc, avec un seul temporaire crée. Pour cela, il suffit de prendre la décision d'atomiser la sous-expression un peu plus tard (on atomise un opérande isolé si celui-ci est de type « expression » et non s'il est de type « variable »).

Étant donnée la technique de réassociation proposée, chaque nouvelle expression est donc affectée à une variable temporaire. Deux techniques peuvent alors être utilisées :

1. La nouvelle instruction (de type affectation) est insérée immédiatement devant l'instruction courante. L'utilisation d'un quelconque algorithme de déplacement d'instructions invariantes (et en particulier celui présenté au chapitre 13) permet dans un deuxième temps de déplacer les temporaires nouvellement crées à l'extérieur des boucles.
   Grâce à l'approche de type bottom-up, ce placement « immédiatement devant » respecte les contraintes « définition-utilisation » entre toutes les nouvelles instructions générées pour une même expression complexe.
2. La nouvelle instruction est insérée directement dans le corps de la boucle correspondant à sa disponibilité. L'insertion a lieu juste devant la boucle de profondeur minimale pour laquelle l'instruction est invariante. Les dépendances sont respectées en prenant les instructions l'une après l'autre dans l'ordre lexicographique et grâce à l'exploration bottom-up de l'arbre.

14.4 Transformations algébriques plus complexes

L'utilisation de l'associativité et de la commutativité des opérateurs permet de construire assez facilement des sous-expressions. Cependant, d'autres propriétés algébriques telles que la distributivité des opérateurs permettent également d'extraire de nouvelles opportunités de déplacement de code. Ainsi, la factorisation ou le développement d'une expression donnée peuvent engendrer l'apparition de nouvelles sous-expressions déplaçables, inexistantes dans la forme originale de l'expression.

Considérons les deux expressions équivalentes $a \times b + a \times c$ et $a \times (b + c)$. La forme factorisée de l'expression sera intéressante si $\mathcal{D}(a) > \mathcal{D}(b+c)$. En revanche, si $\mathcal{D}(c) > \mathcal{D}(a \times c)$, la forme développée de l'expression permettra d'extraire plus de calculs. De même, la transformation de l'expression $(a-b)(a+b)$ en $a^2 - b^2$ (puis $a \times a - b \times b$) est intéressante si $\mathcal{D}(a) < \mathcal{D}(b)$.

Les transformations algébriques présentées au chapitre 8 apportent également des opportunités nouvelles pour le déplacement de code invariant. La substitution de la division flottante par le couple d'opérateurs $(\times, 1/)$ sera par exemple bénéfique dans certains cas, en permettant l'évaluation de la division (de latence importante) à l'extérieur des boucles. L'expression $\frac{a}{b}$ est réécrite sous la forme $a \times (1/b)$ si $\mathcal{D}(b) < \mathcal{D}(a)$.

14.5 Conclusion

Notre approche permet d'effectuer un découpage des expressions en sous-expressions invariantes. Le travail réalisé directement sur les expressions complexes permet de conserver au maximum la structure initiale du code et évite toute atomisation inutile. Notre travail étend les résultats obtenus dans [BC94] à une imbrication de boucles quelconque et pour des expressions complexes. L'approche haut niveau permet

Algorithme 7 Extraction de sous-expressions invariantes - *Fonction Extraire_ sous_ expressions*

Fonction Extraire_sous_expressions(O , pos)
  ▷ *entrée :*
  ▷ *- une opération O (arbre représentant une expression)*
  ▷ *- la position (pos) de l'instruction contenant cette opération*
  ▷ *sortie :*
  ▷ *- l'opération transformée*
  ▷ *- sa disponibilité*
  $\{(o_1, d_1), \ldots, (o_k, d_k)\} \leftarrow$ Eval_disponibilités(O,pos)
  Si O est une opération AC **Alors** ▷ ***Soit*** $\oplus$ *l'opérateur*
    Construire un groupe d'opérandes pour chaque valeur **différente** de disponibilité
    Soit $\{G_1, \ldots, G_m\}$ ces groupes (non vides) triés selon disponibilité croissante
    Pour $i \leftarrow 1, (m-1)$ **Faire** ▷ *pour tous les groupes sauf le dernier*
      Soit d_{G_i} la disponibilité des opérandes du groupe G_i
      Soit $\{o_1, \ldots, o_{n_i}\}$ les n_i opérandes de G_i
      Si $n_i < 2$ **Alors** ▷ *nombre minimum d'opérandes*
        Déplacer l'opérande $\{o_1$ dans le groupe G_{i+1}
      Sinon
        Créer la sous-expression $\oplus(o_1, \ldots, o_{n_i})$
        Insérer le temporaire $t_i = \oplus(o_1, \ldots, o_{n_i})$
        Ajouter t_i dans le groupe G_{i+1}
      Fin Si
    Fin Pour
    Si $d_{G_m} = pos$ **Alors** ▷ *disponibilité maximale*
      ▷ *préserver la structure complexe de la partie variante de l'expression*
      $O' \leftarrow \oplus(o_1, \ldots, o_{n_m})$
      Retourner (O',pos)
    Sinon
      Créer sous-expression $\oplus(o_1, \ldots, o_{n_m})$
      Insérer temporaire $t_m = \oplus(o_1, \ldots, o_{n_m})$
      Retourner (t_m,d_{G_m})
    Fin Si
  Sinon ▷ *non AC -* ***Soit*** $\oslash$ *l'opérateur*
    $d_{max} = \max_{i \in [1,k]} d_i$
    Si $d_{max} = pos$ **Alors** ▷ *disponibilité maximale*
      ▷ *préserver la structure complexe de la partie variante de l'expression*
      Retourner (O, pos)
    Sinon
      Créer sous-expression $\oslash(o_1, \ldots, o_k)$
      Insérer temporaire $t = \oslash(o_1, \ldots, o_k)$
      Retourner (t,d_{max})
    Fin Si
  Fin Si
Fin

Algorithme 8 Extraction de sous-expressions invariantes - *Fonction Eval_ disponibilité*

Fonction Eval_disponibilité(O , pos)
 ▷ *entrée :*
 ▷ *- une opération O*
 ▷ *- une position pos*
 ▷ *sortie :*
 ▷ *- une liste L de couples (opérande,disponibilité)*
 $L \leftarrow \emptyset$
 Pour tout opérande o_i de O **Faire**
  Si o_i est une opération **Alors**
   $(o_i', d_i) \leftarrow$ Extraire_sous_expressions(o_i , pos)
   $L \leftarrow L \cup \{(o_i', d_i)\}$
  Sinon ▷ o_i *est une référence*
   $d_i = \mathcal{D}(o_i)$
   $L \leftarrow L \cup \{(o_i, d_i)\}$
  Fin Si
 Fin Pour
 Retourner L
Fin

en outre d'éviter les problèmes rencontrés par Briggs et Cooper pour reconstruire des opérations n-aires à partir d'un code 3-adresses. Notre technique de réassociation a été implémentée et, en complément du travail présenté au chapitre 13, permet d'effectuer un déplacement de code invariant source-à-source de programmes Fortran. Les détails de l'implémentation sont donnés au chapitre 16.

Chapitre 15

Élimination des sous-expressions communes

Rechercher puis éliminer les sous-expressions communes dans un programme est un travail aussi efficace que difficile. Chaque sous-expression commune détectée permet de réduire le nombre d'opérations nécessaires pour évaluer l'ensemble des expressions. Comme pour le déplacement de code invariant, les propriétés algébriques des opérateurs apportent de nouvelles opportunités d'optimisation. Mais, en conséquence, la complexité du problème est encore augmentée ; lorsqu'il existe plusieurs sous-expressions communes en conflit, la recherche de la solution optimale est un problème combinatoire.

15.1 Motivations

Revenons sur l'extrait de Onde24 introduit au chapitre 11. La figure 15.1 présente une étape intermédiaire du processus de transformation. Le déplacement de code invariant utilisant les techniques de réassociation du chapitre précédent a été appliqué. La phase de transformation suivante consiste à éliminer les sous-expressions communes du code. Outre le calcul redondant de $2 \times$ H qui ne présente pas de difficulté à éliminer, ce code illustre l'intérêt d'utiliser l'associativité et la commutativité des opérateurs (ici de la multiplication). Conformément aux idées présentées au chapitre 8, la division du produit DELTAT $\times$ DELTAT par le terme H a été substituée en utilisant l'opérateur inverse. Cette transformation permet de commuter le terme 1/H parmi les termes de la multiplication. Ainsi, la sous-expression DELTAT $\times$ DELTAT $\times$ 1/H qui apparaît dans T_1 et T_4 peut être détectée grâce à l'associativité et la commutativité de l'opération. Notons que les calculs (1/(V(I,NP) $\times$ DELTAT + H) et V(I,NP) $\times$ V(I,NP) sont également redondants, mais la détection ne nécessite pas l'usage de propriétés algébriques.

15.2 État de l'art

L'élimination des sous-expressions communes est une technique décrite à de nombreuses reprises dans la littérature [ASU86, Coc70] et utilisée aujourd'hui par la plupart des compilateurs. Les travaux de Ullman [Ull73] introduisent une méthode permettant de déterminer l'ensemble des expressions disponibles en un point du programme. Bruno et Sethi [BS76] puis Aho et Johnson [AJ77] s'intéressent au problème de la *génération de code* en présence de sous-expressions communes et montrent que le problème est NP-complet pour les « machines à un registre ». Un algorithme efficace pour une classe particulière de graphes (*colapsible graphs*) sera donné par les auteurs de [PS80]. Depuis, la plupart des travaux se sont concentrés sur le problème de l'équivalence entre deux expressions. Le *Value Numbering* consiste à affecter à deux expresssions la même « valeur », si l'on est capable de prouver que les deux expressions produisent toujours les même résultats. Les techniques utilisées reposent soit sur un mécanisme de hachage [CS70], soit sur la construction de classes d'équivalences [AWZ88]. *Value Numbering* et *Partial Redundancy Elimination*

```
T1 = DELTAT * DELTAT * 1/H
T2 = 2*H
T3 = 2*H
T4 = DELTAT * DELTAT * 1/2 * 1/H
DO I = 2,NP-2
  V(I,NP)   = 2/( (1/V(I,NP)) + (1/V(I,NP-1)) )

  B(I,1,1)  = 1 - (1/(V(I,NP)*DELTAT+H)) *
&              (T2 + V(I,NP)*V(I,NP)*T1)

  B(I,1,2) = T3 /(V(I,NP)*DELTAT+H)

  B(I,1,3) = (V(I,NP)*V(I,NP) * T4 )/
&             ((V(I,NP)*DELTAT+H))
ENDDO
```

FIG. 15.1 – *Recherche de sous-expressions communes : motivations.*

(cf. chapitre 12) sont ensuite combinés pour l'élimination des sous-expressions communes [BC94, Cli95, BCS97].

La compilation n'est pas le seul domaine concerné par la détection des sous-expressions communes. Le problème a été étudié d'un point de vue plus théorique par les auteurs de [Dow80] et [FSS90] pour analyser les relations d'équivalences existant sur les sommets d'un graphe quelconque. La compactification maximale d'un arbre contenant des sous-arbres communs est étudié dans [NP93]. On trouve également des études concernant les sous-expressions communes dans le domaine des bases de données [Fin82, Jar85, AR94] ou encore pour la conception de processeurs DSP [MMA95, MJ96, PSDD97, PSDD99]. Les travaux réalisés pour la vision par ordinateur, la biologie moléculaire ou encore le traitement des langages naturels cherchent également à détecter des sous-structures communes [WSS+98] mais cette fois-ci de manière approchée par l'introduction de la notion de distance entre 2 sous-arbres.

Tous ces travaux concernent la recherche de sous-expressions communes à partir d'une représentation atomisée des expressions, la plupart du temps sous la forme de code 3-adresses. Les techniques existent à la fois pour des blocs d'instructions et pour un graphe de flot de contrôle quelconque. L'objectif de nos travaux est d'étendre la recherche des sous-expressions communes en utilisant les opportunités offertes par l'associativité et la commutativité des opérateurs. Le travail réalisé ici vient donc compléter les techniques traditionnelles. Étant donné l'augmentation de complexité due à cette hypothèse, nous proposons une heuristique se limitant à un bloc d'instructions.

15.3 Analyse de la complexité

La complexité de la recherche des sous expressions communes dans un contexte associatif-commutatif est très grande. En effet, étant donnée une liste d'expressions quelconque, rechercher l'ensemble des sous-expressions communes qui minimise le nombre d'opérations est un problème NP-complet.

Définition 15.1
Le problème **ENSEMBLE COMPUTATION** *(Garey et Johnson [GJ79], page 274) est définit comme suit :*

Instance : *Soit C une collection de sous-ensembles pour un ensemble fini A. Soit J un entier positif.*

Question : *Existe-t-il une séquence $S = (z_1 \leftarrow x_1 \cup x_2, z_2 \leftarrow x_2 \cup y_2, \ldots, z_j \leftarrow x_j \cup y_j)$ de $j \leq J$ opérations d'union, où chaque x_i et y_i est soit le singleton $\{a\}$ pour $a \in A$, soit un z_k pour $k < i$ tel que x_i et y_i sont disjoints $(1 \leq i \leq j)$ et tel que pour chaque sous-ensemble $c \in C$ il existe un z_i $(1 \leq i \leq j)$ identique à c.*

Théorème 15.1
Le problème ENSEMBLE COMPUTATION est NP-Complet.

Preuve 15.1

La preuve est construite par transformation du problème **VERTEX COVER** en **ENSEMBLE COMPUTATION**. La démonstration est donnée dans [GJ79] page 66. *Q.E.D.*

Remarque 15.1
Ce problème reste également NP-complet dans le cas particulier où $\forall c \in C, |c| \leq 3$.

Nous appellerons **AC-CSE**[72] la recherche des sous-expressions communes dans un contexte associatif-commutatif. Résoudre ce problème est en fait équivalent à la résolution du problème ENSEMBLE COMPUTATION. En effet, considérons la formulation de AC-CSE donnée par la définition 15.2. Une liste d'opérations pour un opérateur associatif-commutatif donné peut être vue comme une collection de sous-ensembles sur un ensemble fini (de variables).

Définition 15.2
Le problème **AC-CSE** *est définit comme suit :*

Instance : *Soit O un liste d'opérations pour un opérateur AC (noté $\oplus$) sur un ensemble fini de variables V. Soit J un entier positif.*

Question : *Existe-t-il une séquence d'instructions $t_i = x_i \oplus y_i$ où chaque opérande x_i et y_i est soit une variable $v \in V$, soit le résultat d'une instruction t_k pour $k < i$ et tel que pour chaque opération $o \in O$ il existe un t_i $(1 \leq i \leq j)$ identique à o (c'est-à-dire une instruction qui évalue o).*

Le problème que nous souhaitons aborder n'est donc qu'une reformulation du problème ENSEMBLE COMPUTATION qui a été montré NP-Complet. La recherche de sous-expressions communes dans un contexte associatif-commutatif est donc un problème NP-Complet.

15.4 Heuristique proposée

La recherche de sous-expressions communes pour des opérations associatives-commutatives étant un problème NP-complet, nous savons qu'il n'existe pas d'algorithme optimal calculable en temps polynomial (sauf si P=NP). Nous allons donc nous intéresser ici à la mise en place d'une heuristique d'élimination des sous-expressions communes ayant une complexité raisonnable et permettant d'obtenir en pratique de bons résultats.

Pour concevoir notre heuristique, nous supposons que les techniques de déplacement de code invariant présentées aux chapitres 11, 12, 13 et 14 ont été appliquées. Cette hypothèse permet de garantir que s'il existe deux instances d'une opération quelconque $\oplus(x_1, x_2)$ alors elles ont été placées par le déplacement de code dans la même imbrication de boucles, c'est-à-dire dans la même séquence d'instructions. Nous discuterons ce choix à la section 15.5. D'autre part, nous supposons que les expressions manipulées sont mises sous forme d'arbres n-aires grâce aux techniques présentées au chapitre 8.

Présentation

Notre heuristique recherche les sous-expressions communes à l'intérieur d'une *séquence d'instructions* de type affectation. La structure complexe des expressions est conservée intacte, *sans atomisation inutile* : seules les sous-expressions communes sont affectées à des variables temporaires. La recherche des sous-expressions utilise l'associativité et la commutativité des opérateurs et fonctionne par des intersections sur les ensembles d'opérandes. En cas de conflits entre plusieurs sous-expressions communes, celle concernant le plus grand nombre d'opérandes est sélectionnée. La méthode proposée est décrite par les algorithmes 9, 10 et 11.

La structure de données utilisée pour la recherche des expressions est une table d'éléments définis par un opérateur, une liste d'opérandes, un nom et un état. Afin de ne pas effectuer une atomisation inutile

72. Associative-Commutative Common Subexpression Elimination

Algorithme 9 Élimination de sous-expressions communes (1/3).

Fonction CSE
  Soit $\mathcal{T}$ une table contenant des éléments de la forme (opérateur, liste d'opérandes, nom, état)
  Soit $S_1, S_2, \ldots, S_n$ une séquence d'instructions (affectations) $\forall i > j, S_i \succ S_j$
  Pour $i \leftarrow 1, n$ **Faire** ▷ *dans l'ordre lexicographique*
    Soit LE_i l'expression correspondant au membre gauche de l'instruction
    Soit RE_i l'expression correspondant au membre droit de l'instruction
    Si RE_i est une opération **Alors**
      $nom \leftarrow$ Analyser_expression(RE_i)
      Ajouter (=,{LE_i,nom}, nil, "réel")
    Sinon
      Ajouter (=,{LE_i,RE_i}, nil, "réel")
    Fin Si
    ▷ *Mettre à jour $\mathcal{T}$ en prenant en compte l'écriture de LE_i*
  Fin Pour
Fin

Algorithme 10 Élimination de sous-expressions communes (2/3).

Fonction Analyser_expression(E)
  Soit $\oplus$ l'opérateur à la racine de l'expression E
  Soit L_1 la liste des opérandes de $\oplus$
  $L_2 \leftarrow \emptyset$
  Pour tout $o_i \in L_1$ **Faire** ▷ *tous les opérandes*
    Si o_i est une opération **Alors**
      $L_2 \leftarrow L_2$+Analyser_expression(o_i)
    Sinon
      $L_2 \leftarrow L_2 + o_i$
    Fin Si
  Fin Pour
  (L_3, $sous - expr$) = Rechercher_sous_expression($\oplus$,L_2)
  Si Card(L_2)=Card(L_3) **Alors** ▷ *2 opérations identiques*
    Retourner $sous - expr$
  Sinon
    new = Nommer_temporaire
    Si Card(L_3)=0 **Alors** ▷ *pas de sous-expression*
      ajouter ($\oplus$, L_2, new,"virtuel") dans $\mathcal{T}$
    Sinon
      ajouter ($\oplus$, $L_2 - L_3 + \{sous - expr\}$, new,"virtuel") dans $\mathcal{T}$
    Fin Si
    Retourner new
  Fin Si
Fin

Algorithme 11 Élimination de sous-expressions communes (3/3).

Fonction Rechercher_sous_expression($\oplus$,L_1)
  Si $\oplus$ est associatif-commutatif **Alors**
    ▷ *Rechercher dans $\mathcal{T}$ l'élément permettant une intersection maximale sur les opérandes avec $\oplus$*
    Soit $t = (\oplus,\ L_2,\ nom,\ etat)$ l'élément choisi
    Soit $L_{1\cap 2}$ la liste des opérandes communs à L_1 et L_2
    Si Card($L_{1\cap 2}$) ≥ 2 **Alors** ▷ *sous-expression exploitable*
      Si Card($L_{1\cap 2}$) = Card(L_2) **Alors** ▷ *opérations identiques*
        Mettre à jour t avec ($\oplus$, L_2, nom, "réel")
        Retourner ($L_{1\cap 2}$,nom)
      Sinon ▷ *sous-expression commune*
        new = Nommer_temporaire
        Insérer ($\oplus$, $L_{1\cap 2}$, new,"réel") devant t dans $\mathcal{T}$
        Mettre à jour t avec ($\oplus$, $L_2 - L_{1\cap 2} + \{new\}$, nom,$etat$)
        Retourner ($L_{1\cap 2}$,new)
      Fin Si
    Sinon
      Retourner ($\emptyset$,nil)
    Fin Si
  Sinon ▷ *non AC*
    ▷ *Rechercher l'opération complète dans la liste*
    Si l'expression est déjà présente dans $\mathcal{T}$ **Alors**
      Soit $t = (\oplus,\ L_2,\ nom,\ etat)$ l'élément trouvé
      Mettre à jour t avec ($\oplus$, L_2, nom, "réel")
      Retourner (L_2,nom)
    Sinon ▷ *nouvelle opération*
      Retourner ($\emptyset$,nil)
    Fin Si
  Fin Si
Fin

des expressions, nous utilisons la règle de nommage suivante : un nom est attribué à chaque opération mais la création d'une instruction d'affectation, c'est-à-dire l'atomisation réelle, n'est effectuée que lorsqu'il existe une sous-expression commune. L'état de l'opération devient alors *réel*. Une opération restée dans l'état virtuel à la fin de l'application de l'heuristique ne sera pas atomisée et restera au sein de l'expression complexe initiale. La regénération du code se fait simplement à partir de la table ; les expressions complexes sont reconstruites tant que l'atomisation d'un opérande n'est pas *réellement* motivée par une sous-expression commune. Dans le cas contraire, les expressions référencent les temporaires qui ont été créées. Ce mécanisme d'atomisation virtuelle permet d'éviter la comparaison récursive entre sous-arbres, tout en conservant la structure complexe des expressions.

Quelques précisions

• Afin de simplifier les explications, les algorithmes 9, 10 et 11 ne prennent pas en compte la présence possible d'accès à des éléments de tableaux. En pratique, une référence peut être traitée comme une fonction non associative-commutative quelconque. Les expressions $x+A(i,j,k)$ et $y+A(i,j,k)$ contiennent par exemple une sous-expression commune $A(i,j,k)$ qui correspond en fait au calcul de l'adresse.

• Dans la mesure où nous ne travaillons pas en assignation unique, nous ajoutons dans la table un drapeau supplémentaire permettant d'indiquer si une opération donnée (i.e. un élément dans la table) est encore *vivante* à une étape donnée de l'avancement de l'heuristique. Les instructions sont traitées dans l'ordre lexicographique et donc, pour chaque référence écrite par l'instruction courante, il faut invalider dans la table toutes les opérations qui utilisent une ancienne valeur. Cette analyse, assez facile pour les variables scalaires, est plus complexe pour les instructions écrivant des éléments de tableaux ; il faut en effet invalider uniquement les opérations sur les éléments de tableaux écrits et non les accès à toute la matrice. L'analyse de régions `WRITE` d'une instruction (cf. chapitre 5) nous apporte l'information nécessaire.

• Suite à l'application préalable d'autres transformations ou par simple choix du programmeur, une même expression peut avoir été affectée à deux variables différentes. Considérons l'exemple de la figure 15.2. Une recherche naïve permet de détecter les sous-expressions communes $a+b$ et $c*e$. La présence des références t_1 et t_2 n'est pas interprétée correctement malgré le résultat $t_1 = t_2$.

Tous les éléments de la table $\mathcal{T}$ de la forme (=,$\{x_1, x_2\}$,nil,"réél") définissent en fait un phénomène d'alias entre les références x_1 et x_2. Pour chaque liste d'alias $\{(x_1,x_2),\ (x_2,\ x_3),\ \ldots\ (x_{n-1},\ x_n)\}$, les insertions d'opérations dans la table $\mathcal{T}$ ainsi que la recherche des intersections communes se font donc directement en manipulant la référence x_n[73]. Par exemple, lors de l'analyse des opérandes de l'instruction d'affectation à y sur la figure 15.2, la recherche de l'intersection maximale se fait donc à partir de $\{t_1, t_3, f\}$ et non à partir de $\{t_2, t_3, f\}$.

```
t1 = a+b
x = t1 * c * d * e
t2 = a+b
x = t2 * c * e * f
```

(a) Séquence d'instructions initiale.

```
t1 = a+b
t3 = c * e
x = t1 * t3 * d
t2 = t1
x = t2 * t3 * f
```

(b) Résultat d'une élimination « naïve ».

FIG. 15.2 – *Gestions des alias.*

• La méthode proposée permet de traiter les opérations, associatives-commutatives ou non, mais le cas des appels de fonctions n'a pas été pris en compte pour l'instant.

73. Afin d'assurer qu'il n'existe jamais d'alias de la forme (x,x') et (x,x''), nous devons encore une fois tenir compte de la durée de vie limitée des variables et invalider les alias au fur et à mesure.

Illustration de la méthode

La figure 15.3 présente un exemple de fonctionnement de l'heuristique pour une séquence d'instructions contenant les sous-expressions communes $a+c$, $a+b+c$ et $d+e$. Le tableau 15.1 donne le contenu de la table après traitement de la 1^{ere} instruction. L'atomisation reste à un stade virtuel permettant simplement de nommer chacune des opérations rencontrées.

```
x = (a+b+c) * (d+e) + f
y = (a+c) * (d+e) + g
z = (a+c)
k = z+b+f
```

(a) Séquence d'instructions initiale.

t_5 = a+c
t_1 = t_5 + b
t_2 = d+e
x = t_1 * t_2 + f
y = t_5 * t_2 + g
z = t_5
k = t_1 + f

(b) Résultat obtenu par l'heuristique.

FIG. 15.3 – *Illustration de la méthode utilisée.*

opération	opérandes	nom	état
$+$	$a\ b\ c$	t_1	v
$+$	$d\ e$	t_2	v
$\times$	$t_1\ t_2$	t_3	v
$+$	$t_3\ f$	t_4	v
$=$	$x\ t_4$	*nil*	r

TAB. 15.1 – *Après traitement de la 1^{ere} instruction.*

Le traitement de la seconde instruction fait apparaître la sous-expression $a+c$. Un nouvel élément est donc inséré dans la table 15.2 ; il est marqué comme réel (atomisation nécessaire). Les opérandes a et c sont remplacés par le nouvel opérande t_5 dans la liste des opérandes de l'opération affectée à t_1. De même, l'opération $d+e$ étant déjà présente dans la table, l'élément (+,$\{d,e\}$,t_2,v) est marqué comme réellement utile et t_2 est utilisé directement comme opérande pour le calcul de t_6.

opération	opérandes	nom	état
$+$	$a\ c$	t_5	r
$+$	$t_5\ b$	t_1	v
$+$	$d\ e$	t_2	r
$\times$	$t_1\ t_2$	t_3	v
$+$	$t_3\ f$	t_4	v
$=$	$x\ t_4$	*nil*	r
$\times$	$t_5\ t_2$	t_6	v
$+$	$t_6\ g$	t_7	v
$=$	$y\ t_7$	*nil*	r

TAB. 15.2 – *Après traitement de la 2^{nde} instruction.*

Après traitement des instructions 3 et 4, la méthode donne le résultat de la table 15.3. La sous-expression $a+c$ est découverte dans la troisième instruction et crée un alias entre z et t_5. La recherche de l'intersection maximale pour les opérandes z, b et f est donc faite directement avec t_5, b et f. Cela permet de détecter la sous-expression commune $a+b+c$ (l'atomisation dans t_1 sera donc effectuée).

opération	opérandes	nom	état
$+$	$a\ c$	t_5	r
$+$	$t_5\ b$	t_1	r
$+$	$d\ e$	t_2	r
$\times$	$t_1\ t_2$	t_3	v
$+$	$t_3\ f$	t_4	v
$=$	$x\ t_4$	nil	r
$\times$	$t_5\ t_2$	t_6	v
$+$	$t_6\ g$	t_7	v
$=$	$y\ t_7$	nil	r
$=$	$z\ t_5$	nil	r
$+$	$t_1 + f$	t_8	v
$=$	$k\ t_8$	nil	r

TAB. 15.3 – *Après traitement des 4 instructions.*

15.5 Discussions

La recherche des sous-expressions communes dans un contexte associatif-commutatif est un problème de grande complexité. L'heuristique que nous avons définie et implémentée permet d'exploiter efficacement les opportunités offertes sans entrer dans la combinatoire, au prix de quelques restrictions.

Restriction à une séquence Nous avons choisi de restreindre la recherche des sous-expressions communes à une séquence. Ce choix est motivé par notre décision d'appliquer préalablement un déplacement de code invariant. Les techniques présentées aux chapitres 11, 12, 13 et 14 garantissent l'absence de sous-expressions communes de la forme $\oplus(x_1, x_2)$ appartenant à deux imbrications de boucles différentes. Cependant, ce choix est discutable dans la mesure où peuvent exister des sous-expressions communes concernant des opérandes de ***disponibilités*** différentes (cf. chapitre 14). Considérons l'exemple de la figure 15.4. La sous-expression commune $x+z(j)+k(i)$ concerne des opérandes de disponibilités différentes qui sont donc placés dans différentes boucles par le déplacement de code invariant en (b) sur la figure 15.4. Notre travail sur la séquence ne pourra donc détecter ces sous-expressions communes. En effet, il existe ici un conflit entre le déplacement de code invariant et l'élimination de sous-expressions communes. Si par contre l'extraction des sous-expressions communes est faite sur le code initial, comme présenté en (c) sur la figure, l'expression commune peut être éliminée. La version (c) permet d'effecteur $n-1$ additions de moins que la version (b). Ces interactions seront étudiées plus en détail au chapitre 17.

La gestion des alias est également problématique dans le cadre de l'élimination des sous-expressions communes restreinte à une séquence. Ce phénomène est illustré par la figure 15.5 : l'information concernant l'égalité des variables t_1 et t_2 est indispensable pour la détection complète de la sous-expression commune (x(j) + y(j))*z(i)*k(i) dans la boucle la plus interne. La connaissance des alias doit donc être « propagée » à travers l'imbrication des boucles, d'une séquence à l'autre.

Pour que la propagation des alias soit efficace, un traitement des différentes séquences dans l'ordre lexicographique est imposé. Cependant, dès lors que l'on cherche à franchir des « barrières » de contrôle, par exemple les boucles, se pose le problème du respect du comportement du programme. En effet, la propagation des alias est sans problème de l'extérieur vers l'intérieur d'une boucle mais doit être appliquée avec précaution de l'intérieur vers l'extérieur des boucles ou en présence de tests. Notre approche ne permet pas de résoudre facilement ces problèmes de propagation des alias le long du graphe de contrôle. Les techniques de PRE classiques, applicables à n'importe quel graphe de contrôle, permettront de supprimer les sous-expressions communes de ce type (tant que l'associativité et la commutativité ne sont pas nécessaires).

Opérateurs compléments La manipulation des opérateurs compléments telle que nous l'avons proposée au chapitre 8 permet d'offrir plus d'opportunités aux optimisations de boucles telles que le dé-

```
Do j = 1, N
 Do i = 1, N
  A(i,j) = x + y + z(j) + k(i)
  B(i,j) = x + z(j) + k(i)
continue
```

(a) Code initial.

```
t1 = x + y
Do j = 1, N
 t2 = t1 + z(j)
 t3 = x + z(j)
 Do i = 1, N
  A(i,j) = t2 + k(i)
  B(i,j) = t3 + k(i)
continue
```

(b) Code après icm.

```
Do j = 1, N
 t1 = x + z(j)
 Do i = 1, N
  t2 = t1 + k(i)
  A(i,j) = t2 + y
  B(i,j) = t2
continue
```

(c) Code après cse puis icm.

FIG. 15.4 – *Limites de la restriction à une séquence.*

```
Do j = 1, N
 Do i = 1, N
  A(i,j) = (x(j) + y(j))*z(i)*k(i) + ...
  B(i,j) = (x(j) + y(j))*z(i)*k(i) + ...
continue
```

(a) Code initial.

```
Do j = 1, N
 t1 = x(j) + y(j)
 t2 = x(j) + y(j)
 Do i = 1, N
  A(i,j) =  t1*z(i)*k(i) + ...
  B(i,j) =  t2*z(i)*k(i) + ...
continue
```

(b) Code après icm.

```
Do j = 1, N
 t1 = x(j) + y(j)
 t2 = t1
 Do i = 1, N
  t3 = z(i)*k(i)
  A(i,j) =  t1*t3 + ...
  B(i,j) =  t2*t3 + ...
  continue
```

(c) Code après cse.

FIG. 15.5 – *Propagation indispensable de l'information sur les alias.*

placement de code invariant et l'élimination des sous-expressions communes. L'extrait de l'application Onde24 présenté figure 15.1 en est un bon exemple. Toutefois, cette manipulation n'est malheureusement pas toujours profitable. Prenons comme exemple la substitution de la division par l'opérateur inverse (1/). Si la substitution n'a apporté aucune opportunité nouvelle d'optimisation, repasser sous la forme initiale est très simple. En revanche, elle peut introduire une nouvelle opportunité dont le gain est insuffisant pour compenser l'augmentation potentielle du nombre de divisions à évaluer. La figure 15.6 illustre ce problème avec une sous-expression commune supplémentaire détectée au prix de l'augmentation du nombre de divisions. Le nombre total d'opérations à été réduit de 4 à 3, mais deux multiplications ont été remplacées par une division, ce qui n'est pas du tout souhaitable !

```
x = a/(b.c)
y = a/(b.d)
```

Code initial.

```
t = a/b
x = t/c
y = t/d
```

Code après cse.

FIG. 15.6 – *Manipulation négative des opérateurs compléments.*

Intersection maximale L'existence de conflits entre sous-expressions communes dans un contexte AC rend très difficile la recherche de la solution qui minimise le nombre d'opérations. Nous avons choisi de sélectionner chaque fois que cela est possible, la sous-expressions concernant un nombre maximum d'opérandes. Considérons l'exemple de table $\mathcal{T}$ donné par le tableau 15.4 avant l'insertion de l'opération $a+b+c+d+e+f+g$. Les opérations $a+b+c+d$, $a+e+f$ et $b+d+f$ sont déjà disponibles dans la table. En sélection l'intersection maximale, l'élément (+,$\{t_1, e, f, g\}$,t_4,v) est inséré dans la table (tableau 15.5). Or, l'élément (+,$\{t_2, t_3, g\}$,t_4,v) permettrait de réduire le nombre d'opérations (tableau 15.5). Notons touefois que si l'opération $e + f + g$ existe également dans le code, la décision est un nouvelle fois différente. La décision de privilégier localement les sous-expressions communes de taille maximale n'est donc pas optimale globalement.

opération	opérandes	nom	état
+	$a\ b\ c\ d$	t_1	v
+	$a\ c\ e$	t_2	v
+	$b\ d\ f$	t_3	v

TAB. 15.4 – *Avant insertion de l'opération $a + b + c + d + e + f$.*

opération	opérandes	nom	état
+	$a\ b\ c\ d$	t_1	r
+	$a\ c\ e$	t_2	v
+	$b\ d\ f$	t_3	v
+	$t_1\ e\ f\ g$	t_4	v

TAB. 15.5 – *Choix de l'heuristique.*

15.6 Conclusion

L'élimination de sous-expressions communes dans un contexte associatif-commutatif est un problème NP-complet. Nous proposons une heuristique travaillant sur une séquence d'instructions et choisissant de manière gloutonne l'expression commune de longueur maximale. Notre approche permet d'éviter les comparaisons entre sous-arbres et n'applique pas d'atomisation inutile. L'heuristique a été implémentée

opération	opérandes	nom	état
+	$a\ b\ c\ d$	t_1	v
+	$a\ c\ e$	t_2	r
+	$b\ d\ f$	t_3	r
+	$t_2\ t_3\ g$	t_4	v

TAB. 15.6 – *Choix idéal.*

et permet d'appliquer une élimination associative-commutative des sous-expressions communes présentes dans un code Fortran. Les détails de l'implémentation seront présentés au chapitre 16.

Chapitre 16

Implémentation et expérimentations

Les travaux présentés dans cette partie ont tout d'abord été motivés par l'analyse des résultats obtenus par optimisation manuelle d'applications comme Onde24. Les techniques permettant d'améliorer automatiquement les programmes manipulées ont par la suite été mises en place et implémentées dans l'outil d'analyses et de transformations de programmes `PIPS`, décrit dans l'annexe B. L'implémentation a permis d'expérimenter ces techniques de manière automatique sur des codes écrits en Fortran.

16.1 Implémentation

`PIPS` est un environnement d'analyses et de transformations automatiques dont l'une des principales fonctionnalités est de permettre le développement et l'intégration rapide de nouvelles transformations et/ou analyses.

Transformations implémentées

Nous avons regroupées les différentes transformations proposées dans cette partie pour former une nouvelle phase de transformations de programmes. Chacune des transformations peut être ou non activée par la manipulation des *properties* de `PIPS` (cf. annexe B section B.3.6).

Déplacement d'instructions invariantes L'algorithme de déplacement d'instructions invariantes présenté au chapitre 13 a été implémenté, avec l'aide d'Olivier Albiez. Il repose sur une variation des techniques de parallélisation automatique de code d'Allen et Kennedy qui étaient d'ores et déjà implémentées dans `PIPS`. L'implémentation utilise également l'analyse de dépendances interprocédurale de `PIPS` ainsi que l'analyse des effets et des régions de tableaux [Cre96]. Le déplacement des instructions contenant des appels de fonctions, décrit à la section 13.3, a été implémenté. L'application du déplacement des instructions invariantes en présence de tests, présentée à la section 13.4, n'a par contre pas été implémentée dans `PIPS` car le mécanisme des instructions gardées n'est pas supporté.

Réassociation La construction de sous-expressions invariantes grâce à l'utilisation de transformations algébriques a été implémentée dans `PIPS` avec l'aide de Fabien Coelho. Ce travail repose sur l'algorithme présenté au chapitre 9 et permet de réassocier les opérandes de toute opération associative-commutative. L'interface avec le prototype `EOLE` (cf. chapitre 10) permet d'appliquer une factorisation des expressions favorisant la construction de sous-expressions invariantes. Le mécanisme de communication entre les deux outils est présenté au chapitre 10.

Sous-expressions communes L'heuristique d'élimination des sous-expressions communes présentée au chapitre 15 a été implémentée par Fabien Coelho. Elle effectue une recherche des sous-expressions communes parmi les expressions complexes présentes dans un bloc d'instructions.

Exemples d'utilisation

L'outil PIPS permet d'appliquer très facilement une série de transformations à un programme Fortran. Il existe actuellement plusieurs interfaces graphiques disponibles, axées respectivement sur XView, Emacs et Java (en cours de développement), ainsi qu'un mode texte, appelé `tpips`, permettant d'appliquer des transformations depuis la ligne de commande ou par l'intermédiaire d'un fichier avec une extension ".tpips". La figure 16.1 présente un exemple de fichier tpips permettant d'optimiser le programme Fortran (supposé correct) contenu dans le fichier "file.f".

```
# create a workspace named work from file.f
create work file.f
# select EOLE transformations - FULL_BALANCED includes
#       INVARIANT CODE MOTION
#       COMMON SUBEXPRESSION ELIMINATION
#       EOLE algebraic transformations with the BALANCED strategy
setproperty EOLE_OPTIMIZATION_STRATEGY "FULL_BALANCED"
# apply the forward substitution to all MODULES
apply FORWARD_SUBSTITUTION[%ALL]
# apply dead code elimination using use-def information
apply USE_DEF_ELIMINATION[%ALL]
# apply selected EOLE transformations to ALL modules
apply OPTIMIZE_EXPRESSIONS[%ALL]
# write the optimized version of the code in a file
apply UNSPLIT
# close the workspace
close
```

FIG. 16.1 – *Exemple de fichier tpips*

16.2 Expériences et résultats

L'implémentation n'étant pour l'instant qu'au stade du prototype, nous avons limité nos expériences à des extraits de programmes. En effet, la vérification des résultats obtenus devient vite complexe lorsque des applications complètes sont manipulées. Les extraits de codes qui ont été présentés aux chapitres 11 et 12 ont été optimisés manuellement dans un premier temps. Nous présentons ici les résultats que nous avons obtenus de manière automatique en utilisant PIPS et EOLE. Les expériences ont été effectuées sur différentes architectures avec différents compilateurs, ce qui permet de mieux évaluer l'impact des transformations proposées. Pour chacun des résultats, le processeur et la version du compilateur utilisés ainsi que les options de compilation sont indiqués.

Extrait de Onde24 L'extrait de Onde24 que nous avons introduit à la section 11.3 présente de nombreuses opportunités d'optimisation qui ne sont mises en évidence que par l'utilisation des transformations algébriques. Le tableau 16.2 présente les gains mesurés entre la version initiale du code et la version optimisée automatiquement. La comparaison de ces résultats avec ceux de la page 124 montre que l'optimisation manuelle est légèrement meilleure que l'optimisation obtenue automatiquement. En effet, l'optimisation automatique manque des sous-expressions communes à cause de la présence de moins unaires. Cependant, les gains obtenus automatiquement restent significatifs pour un effort d'optimisation beaucoup plus faible.

Parcours de tuile Le code généré automatiquement par un algorithme de parcours de tuile [AI91] (tiling), dont certaines opportunités d'optimisation ont été présentés au chapitre 12, a également été

Processeur	Compilateur f77	Options de compilation	Gain	Temps (sec) Initial/Optimisé
P2SC 160 MHz	IBM xlf release 4.1	-O3 -qhot -qarch=pwr2	1.63	0.49/0.30
UltraSparc-IIi 270 MHz	SUN WorkShop 5.0	-O3	1.9	0.76/0.4
		-fast	2.09	0.69/0.33
		-fast -O5	2.09	0.69/0.33
PentiumII 300MHz	GNU g77 v0.5.24	-O3	2.13	1.19/0.56
		-O5	2.16	1.19/0.55
R10K 175 MHz	MIPSpro release 7.1	-O3	3.2	1.44/0.45

TAB. 16.1 – *Gains obtenus pour l'optimisation de boucles sur ONDE24.*

optimisé de manière automatique. Nous avons mesuré les temps d'exécution du code sans prendre en compte les appels aux fonctions de communications entre les différents processeurs. Les résultats obtenus sont présentés dans le tableau 16.2. Les gains obtenus sont importants et atteignent 36% sur le R10K. Notons que nous avons renoncé à l'optimisation manuelle totale de ce code en raison du nombre trop important de transformations nécessaires.

Processeur	Compilateur f77	Options de compilation	Gain	Temps (sec) Initial/Optimisé
P2SC 160 MHz	IBM xlf release 4.1	-O3 -qhot -qarch=pwr2	1.46	2.34/1.62
UltraSparc-IIi 270 MHz	SUN WorkShop 5.0	-fast	1.39	2.40/1.73
PentiumII 300MHz	GNU g77 v0.5.24	-O3	1.28	1.94/1.51
		-O5	1.39	2.05/1.47
R10K 175 MHz	MIPSpro release 7.1	-O3	1.58	1.71 / 1.08

TAB. 16.2 – *Gains obtenus pour l'optimisation d'un code de tiling.*

Code généré par `hpfc` Ce code est un extrait du programme SPMD généré par le compilateur HPFC comme résultat de la compilation d'un programme HPF effectuant une multiplication de matrices. Les tableaux étant distribués par blocs et alignés sur le même *template*, une redistribution est appliquée avant le calcul. L'extrait de code que nous avons optimisé correspond à cette redistribution et effectue une énumération de polyèdres. Les temps d'exécution mesurés, sans tenir compte des appels aux fonctions de communications, sont présentés dans le tableau 16.3. Bien que ce code soit semblable à celui du parcours de tuile, les gains obtenus sont plus faibles. En effet, le code présente structurellement un potentiel d'optimisation plus faible. Par ailleurs, certains calculs extraits des boucles par le déplacement de code invariant sont évalués inutilement car le domaine d'itération des boucles est vide dans certains cas. L'application des techniques présentées à la section 13.3 mais non implémentées, permettrait de résoudre partiellement ce problème.

Processeur	Compilateur f77	Options de compilation	Gain	Temps (sec) Initial/Optimisé
P2SC 160 MHz	IBM xlf release 4.1	-O3 -qhot -qarch=pwr2	1.07	0.29/0.27
UltraSparc-IIi 270 MHz	SUN WorkShop 5.0	-fast	1.22	0.84s/0.69s
R10K 175 MHz	MIPSpro release 7.1	-O3	1	0.21/0.21

TAB. 16.3 – *Gains obtenus pour l'optimisation d'un code généré par* HPFC.

16.3 Conclusion

Les algorithmes de déplacement d'instructions invariantes (chapitre 13), de découpage en sous-expressions invariantes (chapitre 14) et d'élimination des sous-expression communes (chapitre 15) ont été implémentés dans l'outils PIPS. Ce travail permet de mesurer de manière automatique le gain pouvant être obtenu par utilisation des propriétés algébriques dans le cadre des optimisations de boucles. Les premiers résultats obtenus sont prometteurs et montrent le réel intérêt de ces transformations. Une étude plus systématique des gains en performances obtenus pour des applications réelles sera faite ultérieurement.

Cinquième partie

Conclusion

Chapitre 17

Discussions

La mise en place des techniques présentées tout au long de ce document soulève diverses interrogations. L'objectif de ce chapitre est donc de discuter les solutions que nous avons retenues et d'introduire quelques problèmes partiellement traités. Les problèmes de stabilité numérique seront tout d'abord discutés à travers la section 17.1 puis nous évoquerons les problèmes rencontrés concernant le typage des expressions dans la section 17.2. Le positionnement de ces travaux dans le processus complet de compilation sera quant à lui discuté à la section 17.3. Enfin, la section 17.4 évoque le problème des interactions entre les différentes techniques présentées dans cette thèse.

17.1 Stabilité numérique

Les optimisations de programmes qui sont proposées tout au long de cette thèse font un usage intensif des propriétés algébriques des opérateurs. Cependant, ces propriétés incontestables du point de vue mathématique sont mises à rude épreuve par les erreurs de calculs des ordinateurs. En effet, les ensembles de nombres ne peuvent être définis qu'avec un nombre fini de chiffres (bits en base 2) ; les calculs sont donc effectués sur des valeurs arrondies, avec des risques de propagation de ces arrondis entraînant parfois de grosses erreurs. Nous ne parlerons ici que des réels qui sont fréquemment utilisés en calcul scientifique, mais la stabilité numérique concerne également les autres types de données. Tout réel $x \in \mathbf{R}$ peut s'écrire sous la forme :

$$x = \epsilon.m.b^e \text{ avec } b \in \mathbf{N},\ \epsilon \in \{-1,+1\},\ e \in \mathbf{Z} \text{ et } m \in [0, b[$$

Cette notation [CGV96] constitue la *représentation en virgule flottante* d'une nombre réel, où b est la base, ϵ le signe, e l'exposant et m la mantisse. Pour obtenir une représentation normalisée des réels, on impose $m \in [1, b[$ (le triplet $\{\epsilon, e, m\}$ devient alors unique). Le codage d'un réel sur un ordinateur utilise donc un nombre fini de bits pour représenter le triplet $\{\epsilon, e, m\}$. La norme `IEEE 754`, actuellement utilisée sur la majorité des ordinateurs, prévoit notamment des codages en simple précision (exposant codé sur 8 bits et mantisse codée sur 23 bits) et double précision (exposant codé sur 11 bits et mantisse codée sur 52 bits).

Cette représentation défini l'ensemble des flottants normalisés, noté $\mathbf{F}$. Or, le comportement des opérations arithmétiques sur $\mathbf{R}$ est différent de celui sur $\mathbf{F}$. Le caractère borné de l'ensemble $\mathbf{F}$ entraîne de problèmes de **sur-capacité** et **sous-capacité** ; le fait qu'il soit discret entraîne des **erreurs d'arrondis**. Soit X la représentation d'un réel x quelconque sur un ordinateur utilisant la base 2. Nous pouvons donc écrire :

$$X = \epsilon.M.2^E \text{ avec } X = x - \epsilon.2^{E-p}.\alpha$$

L'erreur absolue faite sur la mantisse finie M est $2^{E-p}.\alpha$, avec p la précision (nombre de bits utilisés pour coder la mantisse). Suivant le mode d'arrondi utilisé on aura $\alpha \in [-0.5, 0.5[$ (arrondi au plus près), $\alpha \in [0, 1[$ (arrondi vers zéro) ou $\alpha \in [-1, +1[$ (arrondi vers plus ou moins l'infini).

17.1.1 Étude de cas : la factorisation

Afin d'évaluer l'impact de nos transformations algébriques sur la précision des calculs effectués, nous allons examiner le cas particulier de la factorisation. Nous cherchons donc à comparer les erreurs dues, d'une part à une opération de la forme $x \times y + x \times z$ et d'autre part à l'opération sous sa forme factorisée $x \times (y + z)$.

Partant des arrondis $X = x - \epsilon.2^{E_x-p}.\alpha_x$, $Y = y - \epsilon.2^{E_y-p}.\alpha_y$ et $Z = z - \epsilon_z.2^{E_z-p}.\alpha_z$, nous pouvons évaluer l'erreur introduite par les calculs de $X \otimes Y$ et $X \otimes Z$, avec $\otimes$ la multiplication informatique. Notons la présence d'une erreur d'arrondi sur le résultat de l'opération, notée sous la forme $\epsilon_{xy}.2^{E_{xy}-p}.\alpha_{xy}$.

$$X \otimes Y = x.y - x.\epsilon_y.2^{E_y-p}.\alpha_y - y.\epsilon_x.2^{E_x-p}.\alpha_x + \epsilon_x.\epsilon_y.2^{E_x+E_y-2p}.\alpha_x.\alpha_y - \epsilon_{xy}.2^{E_{xy}-p}.\alpha_{xy}$$
$$X \otimes Z = x.z - x.\epsilon_z.2^{E_z-p}.\alpha_z - z.\epsilon_x.2^{E_x-p}.\alpha_x + \epsilon_x.\epsilon_z.2^{E_x+E_z-2p}.\alpha_x.\alpha_z - \epsilon_{xz}.2^{E_{xz}-p}.\alpha_{xz}$$

Remarque 17.1
Les termes du deuxième ordre en 2^{-2p} sont négligés par la suite.

De la même façon, l'erreur introduite par le calcul de $X \otimes\ Y \oplus\ X \otimes\ Z$ ($\oplus$ représentant l'addition informatique) est la suivante :

$$\begin{aligned} X \otimes Y \oplus X \otimes Z =& xy + xz \\ & -x.(\epsilon_y.2^{E_y-p}.\alpha_y + \epsilon_z.2^{E_z-p}.\alpha_z) \\ & -\epsilon_x.2^{E_x-p}.\alpha_x(y+z) \\ & -\epsilon_{xy}.2^{E_{xy}-p}.\alpha_{xy} - \epsilon_{xz}.2^{E_{xz}-p}.\alpha_{xz} \\ & -\epsilon_{xy+yz}.2^{E_{xy+yz}-p}.\alpha_{xy+yz} \end{aligned} \tag{17.1}$$

La forme factorisée de l'expression est également à l'origine d'erreurs que l'on peut évaluer. Le calcul de $Y \oplus Z$ introduit dans un premier temps l'erreur suivante :

$$Y \oplus Z = y + z - \epsilon_y.2^{E_y-p}.\alpha_y - \epsilon_z.2^{E_z-p}.\alpha_z - \epsilon_{y+z}.2^{y+z-p}.\alpha y + z$$

Puis, la multiplication par X nous amène à une erreur totale pour l'opération. Notons que les termes en 2^{-2p} sont ici aussi négligés.

$$\begin{aligned} X \otimes (Y \oplus Z) =& xy + xz \\ & -x.(\epsilon_y.2^{E_y-p}.\alpha_y + \epsilon_z.2^{E_z-p}.\alpha_z + \epsilon_{y+z}.2^{y+z-p}.\alpha_{y+z}) \\ & -\epsilon_x.2^{E_x-p}.\alpha_x(y+z) \\ & +\epsilon_x.\epsilon_y.2^{E_x+E_y-2p}.\alpha_x.\alpha_y + \epsilon_x.\epsilon_z.2^{E_x+E_z-2p}.\alpha_x.\alpha_z \\ & +\epsilon_x.\epsilon_{y+z}.2^{E_x+E_{y+z}-2p}.\alpha_x.\alpha_{y+z} \\ & -\epsilon_{x(y+z)}.2^{E_{x(y+z)}-p}.\alpha_{x(y+z)} \end{aligned} \tag{17.2}$$

Le erreurs obtenues par les équations 17.1 et 17.2 contiennent bien évidemment des termes communs. Le tableau 17.1 propose une comparaison des termes restants. La forme dévelopée de l'expression comporte deux erreurs sur le résultat des multiplications, auxquelles s'ajoute une erreur suite à l'addition des deux résultats intermédiaires. La forme factorisée de l'expression intègre quant à elle une erreur sur l'addition qui est multipliée par la valeur de x. L'erreur sur la multiplication n'apparaît cependant plus qu'une seule fois.

Ce résultat peut être facilement étendu aux cas de factorisations avec n variables. Comme le montre le tableau 17.2, on a dans le premier cas (forme développée) n erreurs sur les multiplications et $n - 1$ erreurs dues aux additions. La forme factorisée entraîne $n - 1$ erreurs dues aux additions et multipliées par x et seulement 1 erreur pour la multiplication.

$x \times y + x \times z$	$x \times (y+z)$
$-\epsilon_{xy}.2^{E_{xy}-p}.\alpha_{xy}$ $-\epsilon_{xz}.2^{E_{xz}-p}.\alpha_{xz}$ $-\epsilon_{xy+yz}.2^{E_{xy+yz}-p}.\alpha_{xy+yz}$	$-x.\epsilon_{y+z}.2^{y+z-p}.\alpha_{y+z}$ $-\epsilon_{x(y+z)}.2^{E_{x(y+z)}-p}.\alpha_{x(y+z)}$

TAB. 17.1 – *Comparaison des erreurs obtenues - factorisation de 2 termes.*

$\Sigma_{i=1}^{n} x.y_i$	$x.\Sigma_{i=1}^{n} y_i$
$-\Sigma_{i=1}^{n} \epsilon_{xy_i}.2^{E_{xy_i}-p}.\alpha_{xy_i}$ $-\Sigma_{i=1}^{n-1} \epsilon_{S_i}.2^{E_{S_i}-p}.\alpha_{S_i}$ avec $S_i = \Sigma_{j=1}^{i+1} xy_j$	$-x.\Sigma_{i=1}^{n-1} \epsilon_{S'_i}.2^{E_{S'_i}-p}.\alpha_{S'_i}$ $-\epsilon_{xS'_{n-1}}.2^{E_{xS'_{n-1}}-p}.\alpha_{xS'_{n-1}}$ avec $S'_i = \Sigma_{j=1}^{i+1} y_j$

TAB. 17.2 – *Comparaison des erreurs obtenues - factorisation de n termes.*

Cette comparaison formelle des erreurs respectives des formes développées et factorisées d'une somme de produits montre que ni l'une, ni l'autre des deux formes n'est potentiellement plus "précise" que l'autre ! Selon les valeurs utilisées, les problèmes d'arrondis seront différents. Supposons que les calculs sont effectués avec une représentation en base 10, avec une précision égale à 3. Nous avons donc des valeurs pour l'exposant telles que $E_{max} = 3$ et $E_{min} = -3$. De même, nous avons pour l'ensemble des réels *positifs* (sachant que **F** est symétrique par rapport à 0) une valeur maximum $v_{max} = 9,99.10^2 = 999$ et une valeur minimale $v_{min} = 1.00.10^{-3} = 0.001$.

Calculons les résultats obtenus pour $x = 10$, $y = 100$ et $z = -104$ en utilisant l'expression développée $x \times y + x \times z$ et factorisée $x \times (y+z)$. Le résultat exact du calcul est ici -40. Les résultats donnés par le tableau 17.3 montrent qu'un problème de *sur-capacité* provoque une énorme erreur pour la version développée du calcul. La version factorisée donne ici un résultat correct.

Opération mathématique	Valeur exacte	Opération informatique	Valeur calculée
$x \times y$	1.10^3	$X \otimes Y$	$9,99.10^2$
$x \times z$	$-1,04.10^3$	$X \otimes Z$	$-9,99.10^2$
$x \times y + x \times z$	-4.10^1	$X \times Y + X \times Z$	0
$y+z$	-4	$y \oplus z$	-4
$x \times (y+z)$	-4.10^1	$X \times (Y+Z)$	-4.10^1

TAB. 17.3 – *Précision des calculs en base 10, précision 3 ; $x = 10$, $y = 100$ et $z = -104$.*

Prenons maintenant les valeurs $x = 9$, $y = 1,2$ et $z = 0,144$. Le tableau 17.4 montre que le résultat obtenu grâce à la version développée de l'expression est cette fois-ci « meilleur » que celui obtenu pour la version factorisée. L'erreur d'arrondi sur le calcul de $y+z$ qui est multiplié par x entraîne une imprécision dans le résultat de la version factorisée. Notons toutefois que la précision utilisée ne permet pas ici d'atteindre le résultat exact.

Ces deux exemples montrent bien que ni la forme factorisée ni la forme développée de notre calcul ne peuvent être considérées comme « plus précise » (entachée de moins d'erreurs d'arrondis). Toutefois, notons que toute transformation permettant de diminuer le nombre d'opérations (la factorisation par exemple) limite potentiellement la quantité de calculs « intermédiaires » susceptibles de faire perdre en précision le résultat final. Rien ne permet cependant de se mettre complètement à l'abri de l'instabilité numérique intrinsèque de **F**.

Opération mathématique	Valeur exacte	Opération informatique	Valeur calculée
$x \times y$	$1,08.10^1$	$X \otimes Y$	$1,08.10^1$
$x \times z$	-1.296	$X \otimes Z$	-1.30
$x \times y + x \times z$	$9,504$	$X \times Y + X \times Z$	$9,50$
$y + z$	$1,056$	$y \oplus z$	$1,06$
$x \times (y + z)$	$9,504$	$X \times (Y + Z)$	$9,54$

TAB. 17.4 – *Précision des calculs en base 10, précision 3 ;* $x = 9$, $y = 1,2$ *et* $z = 0,144$.

17.1.2 Des transformations agressives à contrôler

L'arithmétique en virgule flottante n'est donc qu'une approximation plus ou moins bonne de l'arithmétique exacte ; les erreurs d'arrondis risquent d'entacher d'erreurs les résultats du programme. D'autre part, la stabilité numérique du programme est directement liée à l'algorithme utilisé (propagation des erreurs) et à l'écriture des formules mathématiques. Les transformations algébriques proposées dans cette thèse risquent donc de modifier la précision numérique du programme, en bien ou en mal. Deux cas de figure se présentent alors :

1. Si comme cela est malheureusement trop souvent le cas, le programmeur n'a pas réfléchi aux problèmes que pose l'arithmétique des ordinateurs, l'application de transformations algébriques peut soit améliorer la précision numérique (programmeur chanceux !), soit la dégrader (programmeur malchanceux).

   Notons que comme ces transformations visent pour la plupart à réduire le nombre d'opérations (nécessaires à l'évaluation d'une expression), il est probable que les pertes de précision numérique s'en trouvent réduites. Rappelons également que la précision numérique des *fma* est meilleure que celle obtenue suite à l'application successive de la multiplication et de l'addition. Cependant, ce type de transformations peut potentiellement engendrer une erreur « critique » suite à l'introduction d'un phénomène d'*absorption* ou d'*élimination* [Erh90].
2. Le principaux acteurs du domaine scientifique sont de plus en plus conscients des problèmes de stabilité numérique. En cas d'oublis, l'actualité est là pour le leur rappeler :

   Février 91 (guerre du golfe), un anti-missile *Patriot* rate un missile *Scud* en raison d'un décalage progressif de l'horloge. Il s'écrase sur une base américaine en faisant 28 morts.

   Juin 96, la fusée Arianne explose après seulement 40 secondes de vol en raison d'un dépassement de capacité concernant la vitesse horizontale de la platteforme.

   Diverses techniques ont été mises en place pour permettre aux scientifiques d'analyser les problèmes d'instabilité numérique [Col93, BBC$^+$97, Kra98]. La bibliothèque CADNA [CGV96] permet par exemple d'estimer l'erreur d'arrondi sur le résultat du programme par une approche stochastique. Dans la mesure où les transformations que nous proposons sont *source-à-source*, le programmeur peut sans difficulté contrôler le comportement numérique de l'application après nos transformations de la même manière que pour le programme initial (en introduisant des appels à CADNA dans son programme initial par exemple).

Cependant, toutes les précautions doivent être évidemment prise pour vérifier l'impact des transformations appliquées. En effet, comme cela est fait par tous les compilateurs, ce genre d'optimisation agressive (telle que -O3 ou plus) doit être accompagné d'un message d'avertissement en direction de l'utilisateur !

```
Optimization level 3 has the potential to alter the semantics of
a program. Please refer to documentation on -O3 and the STRICT
option for more information (IBM xlf compiler)
```

17.1.3 Les langages de programmation

Les problèmes de stabilité numérique ont été pris en compte au cours de l'évolution des langages de programmation. Les transformations « autorisées » pour la compilation et l'optimisation des programmes sont spécifiées à travers les normes des langages. L'utilisation de la commutativité des opérateurs est par exemple autorisée le plus souvent (les opérations informatiques sont commutatives). Fortran, le langage le plus utilisé en calcul scientifique, n'autorise par exemple que les transformations respectant les parenthèses (voir les paragraphes 6.6.4 de [ANS83] et 7.1.7 de [ISO91]). Le langage C (paragraphe 3.3 de [ANS89]) interdit quant à lui les transformations algébriques (après les avoir autorisées dans un premier temps).

Malgré le restrictions imposées par les normes de programmation, tous les compilateurs proposent deux sortes d'optimisations : celles qui sont « sans risque » et aboutissent à de faibles performances et celles qui sont plus agressives (« à risque ») et améliorent considérablement les performances. Tous les programmes scientifiques ne sont pas intrinsèquement instables numériquement ; ils peuvent donc la plupart du temps être compilés avec des optimisations agressives. En revanche, pour les applications instables, le numéricien doit s'assurer que les transformations respectent les contraintes numériques. Cela est rendu possible par l'utilisation de transformations source-à-source. L'utilisateur peut ainsi contrôler le processus d'optimisation « haut niveau ».

Certaines des transformations que nous employons ne peuvent être appliquées qu'avec une certaine « agressivité » (la factorisation par exemple). En revanche, il est possible d'appliquer un nombre important de nos transformations, comme par exemple l'extraction des *fma* ou des sous-expressions invariantes, tout en respectant les parenthèses introduites par l'utilisateur. Ceci permet alors de « protéger » les sous-expressions qui sont instables numériquement. Pour implémenter cette restriction, il suffit de traiter les parenthèses comme des fonctions unaires n'acceptant aucune transformations algébriques [74].

17.2 Typage des expressions

Les techniques présentées dans les parties III et IV exploitent les propriétés algébriques des opérateurs telles que l'associativité et la distributivité pour l'organisation de l'évaluation des calculs. Outre les problèmes de stabilité numérique présentés ci-dessus, les transformations proposées posent le problème du typage des sous-expressions et celui de l'évaluation d'opérations de types « non homogènes ».

Typage des sous-expressions Tout processus d'atomisation, notamment à travers la recherche des sous-expressions communes ou le déplacement de code invariant, repose sur la création de variables temporaires. Or, une expression peut contenir des variables de différents types. Le typage de la variable temporaire nécessite donc la recherche du type de la sous-expression. De plus, une expression peut inclure des opérations ou des appels de fonctions aboutissant à un résultat donc le type est différent de celui de ses opérandes ou paramètres ; il est par exemple facile de construire une fonction calculant la racine carrée d'un nombre entier et renvoyant le résultat de type réel. Le typage des temporaires suppose donc une analyse complète de l'expression.

Évaluation d'opérations de type « non-homogène » Nous appelons *opération de type « non homogène »*, toute opération avec n opérandes ayant k types différents. En présence d'opérateurs associatif-commutatif, le temps d´évaluation d'une expression dépend de la réassociation éventuelle des différents opérandes présents. En effet, le coût de calcul en simple ou en double précision peut varier considérablement. La latence de la division passe par exemple de 17 à 32 cycles sur un processeur R10000 et de 29 à 69 cycles sur un processeur Pentium Pro. En notant s_i (resp. d_i) un opérande en simple (resp. double) précision, l'opération $\frac{s_1 \times d_2 \times s_3}{s_4}$ pourrait être réécrite sous la forme $d_2 \times \frac{s_1 \times s_3}{s_4}$ afin que la division soit effectuée en simple précision et non plus en double précision. De plus, il n'est pas rare de trouver dans les calculs scientifiques impliquant à la fois des paramètres entiers et de variables réelles. Or, le mélange des opérandes entre eux entraîne un nombre non négligeable de conversions de types qui ne sont pas toujours utiles. La prise en compte de ce phénomène peut être vue sous deux aspects :

– Les techniques de réassociation peuvent être utilisées pour regrouper entre eux les opérandes de même types afin de réduire le nombre de conversions nécessaires. Le passage de divisions de double à simple précision est également automatisable. Enfin, appliquer une conversion de type au moment

74. Nous avons dans un premier testé cette fonctionnalité dans EOLE puis elle a été abandonnée pour simplifier l'implémentation.

de l'évaluation partielle peut permettre de réduire le nombre de conversions à l'exécution ; on transforme par exemple pour le langage FORTRAN le calul $2 \times d \times 3$ par sa formulation en double précision $6.D0 \times d$. En pratique, la plupart des applications rencontrées contiennent un ensemble non homogène de types. Cependant, les opérandes entiers sont souvent évaluables dès la compilation.

- L'utilisation de l'associativité pour, par exemple, équilibrer un arbre ou construire une sous-expression commune, devrait tenir compte des conversions de type introduites. La transformation ne doit pas être interdite car elle peut par exemple ajouter une conversion mais faire gagner n^3 opérations dans le cas du déplacement de code invariant. Cependant, les conversions de types pourraient être prises en compte dans le choix des transformations en incluant au coût d'évaluation d'une opération, le coût des conversions de types nécessaires ; ce n'est pas fait dans notre implémentation actuelle.

17.3 Des transformations de niveau intermédiaire ?

Les techniques présentées dans les parties III et IV se positionnent à un niveau intermédiaire dans le processus global d'optimisation. En effet, ce sont des transformations qui sont difficiles à gérer pour le programmeur. Les optimisations de programmes nécessitent un travail de longue durée diminuant sévèrement la lisibilité du programme. Les transformations d'expressions dans le cadre de l'évolution des processeurs modernes qui sont présentées dans la partie III demandent en plus une bonne connaissance de l'architecture cible et risque de dégrader la portabilité du code (en terme d'efficacité des transformations).

Du point de vue des optimisations traditionnelles telles qu'elles sont décrites notamment dans le fameux *Dragon Boook* [ASU86], les transformations que nous appliquons restent de « haut niveau ». En effet, la représentation interne sous forme de code 3-adresses n'est pas vraiment adaptée à l'exploitation des propriétés algébriques des opérateurs.

Nous nous plaçons donc à mi-chemin entre les transformations de haut-niveau souvent manuelles qui visent à produire un code performant et les optimisations de programmes du compilateur qui travaillent essentiellement au niveau de l'assembleur. Cependant, ces techniques reposent sur des notions que tout programmeur concerné par les performances de son application aurait avantage à connaître et qui pourraient être appliquées sur une représentation intermédiaire (d'assez haut niveau) par tout compilateur. Notre choix de nous placer dans le cadre de transformations source-à-source est principalement justifié par la commodité apportée dans la mise en place de prototypes avec des perspectives expérimentales.

Les transformations source-à-source présentent l'inconvénient de ne pas permettre l'optimisation des calculs d'adresses. Cependant, les techniques présentées dans cette thèse telles que l'extraction des *fma* ou la construction de sous-expressions invariantes peuvent être utilisées dans le cadre d'un préprocesseur ; cela suppose simplement que la structure de données manipulée reste de haut niveau.

17.4 Étude des interactions

Le problème des interactions se pose dès l'instant où l'on cherche à appliquer une série de transformations. En effet, en présence d'interactions, le choix et l'ordre d'application des transformations interviendra sur le résultat final obtenu. Certaines optimisations peuvent être considérées comme toujours profitable ; c'est le cas par exemple de l'élimination de code mort ou de l'évaluation partielle. En revanche, l'application de la majorité des transformations risque potentiellement de réduire les opportunités d'optimisations pour des phases ultérieures.

Transformations algébriques

Les expériences que nous avons réalisées ont montré tout l'intérêt des transformations algébriques dans le cadre de l'évaluation d'expressions sur processeurs superscalaires. Les simplifications algébriques, la factorisation, l'extraction des *fma* ou encore l'équilibrage de l'arbre sont des techniques permettant d'obtenir de meilleures performances. Néanmoins, des interactions existent d'une part entre ces transformations et d'autre part avec les optimisations ultérieurement appliquées par le compilateur.

Conflits entre transformations algébriques

Les transformations qui ont été présentées dans la partie III ne sont pas indépendantes les unes des autres. En effet, chacune des transformations est tributaire de l'état dans lequel elle reçoit l'expression à manipuler. Par exemple, on peut constater sur la figure 17.1 que l'extraction des *fma* n'est pas la même selon que l'expression de départ à été factorisée ou non. La forme développée de l'expression présente en effet plus d'opportunités en raison de la présence de multiplications supplémentaires. Comme on peut le voir sur cet exemple, les résultats obtenus en appliquant ou non la factorisation avant l'extraction des *fma* contiennent un nombre d'opérations identique. La factorisation permet un gain de $n-1$ multiplications et laisse une opportunité de construction de *fma*. L'extraction des *fma* permet de construire n opérations fusionnées (à condition qu'il existe un terme non multiplicatif). Cependant, il est préférable d'appliquer la factorisation avant l'extraction des *fma* car la latence du *fma* est, dans le meilleur des cas, égale à celle de la multiplication.

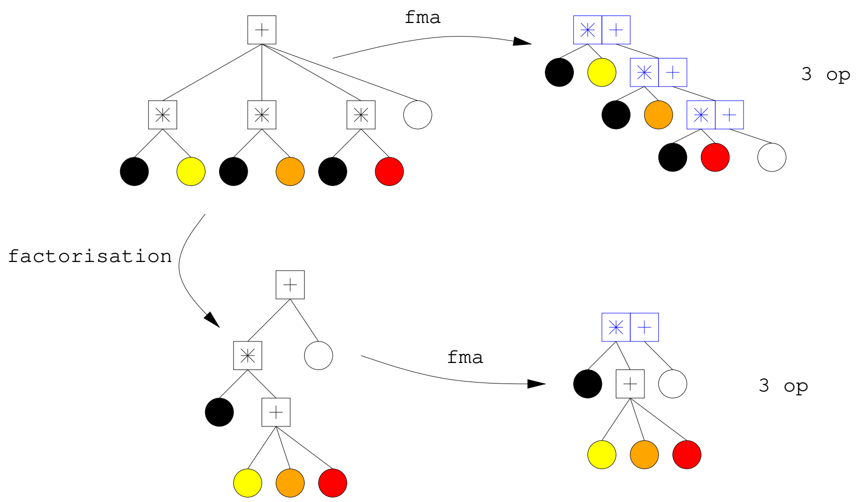


FIG. 17.1 – *Conflits entre la factorisation et l'extraction des* fma.

Par ailleurs, l'étape de normalisation, présentée au chapitre 8, est dupliquée de manière à appliquer des simplifications algébriques d'une part sur les expressions initiales et d'autre part après mise en évidence de nouvelles opportunités par la factorisation où l'extraction des *fma*. Rappelons également que le processus de binarisation des arbres doit être appliqué en dernier lieu, de manière à conserver la structure n-aire facilitant les transformations algébriques. Enfin, notons que la factorisation diminue le nombre d'opérateurs mais tend à reduire le champ d'application des techniques d'équilibrage d'arbre.

Interactions avec le *back-end*

A partir de l'instant où les transformations algébriques modifient la structure d'une expression, elles interviennent également indirectement sur le code assembleur qui sera généré par la suite. Les paramètres tels que le nombre de registre nécessaires, les dépendances entre les calculs, le nombre d'opérations à évaluer, etc. sont donc modifiés par nos transformations. Les interactions avec les optimisations du dorsal[75] du compilateur ne sont donc pas négligeables. Notre approche consiste à effectuer des transformations dans le but d'offrir plus d'opportunités d'optimisation au compilateur :

- La factorisation est une transformation le plus souvent profitable car elle permet de réduire le nombre total d'opérations. Le choix de la structure de l'expression factorisée permet également d'offrir plus de libertés au compilateur pour le choix des calculs indépendants. L'augmentation du chemin critique pouvant avoir lieu dans certaines configurations est masquée par la réduction du volume de calcul.

75. Traduction de *back-end* en français.

• L'extraction des *fma* permet au compilateur de détecter un maximum de *fma* et ainsi réduire le coût d'évaluation des expressions pour les processeurs disposant de telles opérations fusionnées.

• L'intervention sur l'équilibrage de l'expression existe au niveau des heuristiques de factorisation et d'extraction des *fma* ainsi qu'à travers la construction de la forme binaire des opérations. La motivation de ces travaux est de fournir au compilateur des expressions contenant un nombre important d'opérations évaluables de manière indépendante. Placer les calculs coûteux en haut de l'arbre, permet par exemple de ne pas bloquer l'exécution des calculs de moindre coût. Les techniques utilisées cherchent donc à faciliter le travail d'ordonnancement des calculs dans le but de « nourrir » au mieux les ressources de calculs des processeurs superscalaires.

La transformation des expressions avec l'objectif d'améliorer le parallélisme d'instructions devrait apporter au compilateur plus de libertés dans l'ordonnancement des calculs. En effet, l'application systématique des techniques de déroulage de boucles et de software pipelining n'est pas toujours heureuse en raison des problèmes de pression de registres. Réduire les dépendances entre les calculs en équilibrant au mieux l'expression permet de diminuer le facteur de déroulage de la boucle et donc de limiter le nombre de registres utiles. En effet, le nombre de registres est souvent insuffisant au regard de l'augmentation de la densité d'instructions nécessaire pour atteindre de bonnes performances avec des processeurs superscalaires.

Une des nombreuses applications possibles des transformations algébriques, que nous n'avons pas approfondie, est la construction d'une expression nécessitant un faible nombre de registres. C'est un problème difficile qui est déjà NP-complet en présence de sous-expressions commune, même en dehors du contexte AC [Set75].

Interactions avec les optimisations de boucles

L'application des transformations algébriques intervient également sur les opportunités d'optimisation de boucles présentées dans la partie IV. Les conflits notamment entre la factorisation et le déplacement de code invariant sont évoqués ci-dessous.

Optimisations de boucles

L'application de transformations algébriques dans le cadre des optimisations de boucles entraîne également des problèmes de « conflits d'intérêts ». Chacune des décisions prise, bien que bénéfique localement pour la transformation considérée avec une fonction de coût donnée, peut avoir une incidence négative pour la suite des optimisations.

Conflits entre optimisations de boucles

Des conflits apparaissent à partir de l'instant où les transformations appliquées modifient la position et la forme des expressions. Examinons par exemple les interactions entre le déplacement de code invariant (*icm*) et l'élimination de sous-expressions communes (*cse*). Selon l'ordre d'application de ces deux transformations, le nombre d'opérations de la solution finale peut être différent.

Nous cherchons à obtenir des éléments de comparaison concernant les gains obtenus par application successive de *icm* puis *cse* d'une part et *cse* puis *icm* d'autre part. Nous nous concentrons sur les interactions concernant deux opérations placées dans une boucle de profondeur 2, un contexte simple et correspondant à un cas fréquemment rencontré en pratique. Les opérandes peuvent être classés par catégories de la manière suivante :

- les opérandes formant une sous-expression commune et pouvant être déplacés d'un niveau, notés x_i^1
- les opérandes formant une sous-expression commune et pouvant être déplacés de deux niveaux, notés x_i^2
- les opérandes formant une sous-expression commune, notés y_i,

– les opérandes pouvant être déplacés d'un niveau, notés z_i^1,
– les opérandes pouvant être déplacés de deux niveaux, notés z_i^2,
– les autres opérandes, notés k_i.

Nos deux opérations associatives-commutatives peuvent donc aisément être mise sous la forme :

$$\begin{aligned} A &= \Sigma_{i=1}^{n_1} x_i^1 + \Sigma_{i=1}^{n_2} x_i^2 + \Sigma_{i=1}^{m} y_i + \Sigma_{i=1}^{p_1} z_i^1 + \Sigma_{i=1}^{p_2} z_i^2 + \Sigma_{i=1}^{q} k_i \\ B &= \Sigma_{i=1}^{n_1} x_i^1 + \Sigma_{i=1}^{n_2} x_i^2 + \Sigma_{i=1}^{m} y_i + \Sigma_{i=1}^{p_1} z_i^{1'} + \Sigma_{i=1}^{p_2} z_i^{2'} + \Sigma_{i=1}^{q} k_i' \end{aligned} \tag{17.3}$$

La figure 17.2 présente en partie gauche le résultat obtenu après élimination des sous-expressions communes. Le code après déplacement de code invariant (sur le résultat de la cse) est donné en partie droite de cette même figure.

```
Do i
 Do j
   t1 = Σ x_i^1 + Σ x_i^2 + Σ y_i
   A  = t1 + Σ z_i^1 + Σ z_i^2 + Σ k_i
   B  = t1 + Σ z_i^1' + Σ z_i^2' + Σ k_i'
continue
```

- CSE -

```
t2 = Σ x_i^2
t4 = Σ z_i^2
t6 = Σ z_i^2'
Do i
  t3 = t2 + Σ x_i^1
  t5 = t3 + Σ z_i^1
  t7 = t6 + Σ z_i^1'
  Do j
   t1 = t3 + Σ y_i
   A  = t1 + t5 + Σ k_i
   B  = t1 + t7 + Σ k_i'
continue
```

- ICM -

FIG. 17.2 – *Application de cse puis icm.*

La figure 17.3 présente les résultats obtenus cette fois-ci en appliquant dans un premier temps le déplacement de code invariant puis en effectuant une élimination des sous-expressions communes.

```
t1 = Σ x_i^2 + Σ z_i^2
t2 = Σ x_i^2 + Σ z_i^2'
Do i
  t3 = t1 + Σ x_i^1 + Σ z_i^1
  t4 = t2 + Σ x_i^1 + Σ z_i^1'
  Do j
   A = t3 + Σ y_i + Σ k_i
   B = t4 + Σ y_i + Σ k_i'
continue
```

- ICM -

```
t5 = Σ x_i^2
t1 = t5 + Σ z_i^2
t2 = t5 + Σ z_i^2'
Do i
  t6 = Σ x_i^1
  t3 = t6 + t1 + Σ z_i^1
  t4 = t6 + t2 + Σ z_i^1'
  Do j
   t7 = Σ y_i
   A  = t3 + t7 + Σ k_i
   B  = t4 + t7 + Σ k_i'
continue
```

- CSE -

FIG. 17.3 – *Application de icm puis cse.*

La comparaison des parties droites des figures 17.2 et 17.3 permet de constater que les termes en Σ se trouvent placés de manière identique quel que soit l'ordre choisi pour l'application des 2 transformations. Seule la position des additions entre ces termes diffère. Le tableau 17.5 indique le nombre d'additions placées dans chacune des imbrications de boucles. On peut constater que le fait de privilégier l'élimination des sous-expressions communes réduit le nombre d'occurrences de l'addition dans le code, mais augmente le nombre global d'évaluations nécessaires dans la boucle la plus interne.

Version	$\times 1$	$\times n$	$\times n^2$
icm-cse	2	4	4
cse-icm	0	3	5

TAB. 17.5 – *Nombre d'additions : icm-cse vs cse-icm.*

Les éléments de comparaison obtenus tendent à montrer que l'application du déplacement de code invariant en premier lieu est intéressante lorsque le domaine d'itérations des boucles n'est pas vide. Néanmoins, ces indications doivent être prise avec précaution dans la mesure où il existe de nombreux cas particuliers pour lesquels l'application de l'élimination des sous-expressions communes, en premier lieu, aboutit à un meilleur résultat. La figure 17.4 en donne un exemple : pour les deux instructions $A(i_2, i_1) = x + y + z(i_1) + k(i_2)$ et $B(i_2, i_1) = x + z(i_1) + k(i_2)$, le résultat de la transformation *icm-cse* nécessite $1 + 2n + 2n^2$ additions tandis que celui obtenu par *cse-icm* en demande seulement $n + 2n^2$. L'explication de ce résultat est l'absence de certains des termes qui étaient présents dans notre formulation « générale ». Donc, l'ordre *icm-cse* ne peut être considéré comme toujours meilleur que l'ordre *cse-icm*, et vice-versa. Par contre, dans le cas général, le gain obtenu en appliquant *icm* puis *cse* semble plus intéressant.

```
Do i1
 t2 = x + z(i1)
 Do i2
  t1 = t2 + k(i2)
  A(i2,i1) = t1 + y
  B(i2,i1) = t1
```

- CSE-ICM -

```
t1 = x + y
Do i1
 t4 = z(i1)
 t2 = t1 + t4
 t3 = x + t4
 Do i2
  t5 = k(i2)
  A(i2,i1) = t2 + t5
  B(i2,i1) = t3 + t5
```

- ICM-CSE -

FIG. 17.4 – *Un exemple de code mieux adapté à cse puis icm.*

Des interactions existent également avec d'autres optimisations. Par exemple, l'utilisation classique de la transformation d'une multiplication par la constante 2 en un shift et[76] le déplacement de code invariant, sont en conflit. L'expression $a \times b \times 2 \times c$ est en effet réécrite en $((a \times b) << 1) \times c$, ce qui réduit les opportunités de réassociation du chapitre 14.

Interactions avec les transformations algébriques

La factorisation et l'extraction de *fma* entraînent des modifications de la structure générale des expressions. Elles modifient donc les opportunités offertes pour les techniques d'optimisation de boucles. Nous avons vu au chapitre 14 que le choix entre les différentes formes factorisées d'une expression peut être réalisé de manière à favoriser les opportunités de déplacement de code invariant. De la même façon, la factorisation intervient dans l'existence ou non de sous-expressions communes. Considérons l'expression $a \times b + a \times c + d \times c$. La factorisation sur a ou c permet de créer les deux expressions $a \times (b + c) + d \times c$ et $a \times b + c \times (a + d)$. Or, la première forme factorisée créé une opération nouvelle $b + c$ qui n'existe ni dans la forme initiale de l'expression, ni dans la seconde forme factorisée.

76. Sorte de réduction de force.

Interactions avec le *back-end*

Le déplacement de code invariant et l'élimination des sous-expressions communes effectuent une *atomisation* des expressions. Cette étape peut s'avérer problématique dans la mesure où elle insère de nouvelles variables temporaires qui seront vraisemblablement transformées en registres ultérieurement. De plus, les nouvelles instructions rendent plus complexe les analyses du programme puisque le compilateur ne connaît pas les propriétés spécifiques des variables créées par l'atomisation (variables privées, assignation unique, etc.). Cependant, les techniques que nous avons proposées aussi bien pour le déplacement de code invariant que pour l'élimination des sous-expressions communes, sont basées sur un principe d'atomisation minimale. Seules les sous-expressions dont l'atomisation est véritablement utile et permet une optimisation, sont atomisées, ce qui réduit la pression de registres. Le véritable travail d'atomisation est laissé au compilateur.

Notons également que le déplacement de code invariant peut poser des problèmes en terme de défaut de cache. En effet, une donnée utilisée au cœur du nid de boucle peut-être disponible dans la mémoire cache alors qu'elle ne l'est plus lorsqu'elle est déplacée d'un ou plusieurs niveaux de boucles. Pour que la transformation reste efficace, il faut que le nombre d'opérations supprimées compense l'apparition du défaut de cache !

Calcul parallèle

La parallélisation semi-automatique de programmes scientifiques telle qu'elle est organisée pour le langage HPF, dépend fortement des optimisations de programmes appliquées.

- Nous avons pu constater au chapitre 4 que le programme obtenu en amont du processus de parallélisation influence considérablement le comportement du processus de compilation. Un code trop fortement optimisé sera difficile à analyser et donc la parallélisation risque d'échouer.
- L'efficacité du programme parallèle généré est également fortement dépendante des techniques disponibles pour l'améliorer. En effet, la majorité des codes générés automatiquement sont fortement optimisables. L'utilisation des propriétés algébriques pour mettre en évidence des opportunités nouvelles d'optimisations est tout particulièrement intéressante.

Suite à l'analyse des interactions présentées ci-dessus, nous avons choisi pour notre implémentation d'appliquer les transformations dans l'ordre suivant :

1. Forward substitution
2. Use-def Elimination
3. Simplification algébriques et normalisation
   Réduction de force
   Évaluation partielle
4. Factorisation des expressions
5. Déplacement de code invariant avec réassociation
6. Élimination des sous-expressions communes
7. Extraction des *fma*
8. Équilibrage des expressions

Rappelons que l'implémentation permet de choisir les transformations désirées. La *forward substitution* sur un code non atomisé ou l'extraction des *fma* en vue de l'exécution sur une architure sans *fma* ne sont par exemples pas utiles. Ce processus de transformations peut être appliqué directement sur une application ou sur le code généré automatiquement par un compilateur de langages data-parallèles tels que HPF.

Chapitre 18

Contributions et perspectives

Cette étude repose sur l'analyse, la conception, la réalisation et l'expérimentation de différentes techniques permettant de diminuer le temps d'exécution des applications de calcul scientifique. Ce dernier chapitre rappelle les différentes contributions apportées et suggère quelques voies futures de développement.

Contributions apportées

La puissance des ordinateurs a considérablement augmenté, au cours des dernières décennies, grâce à de nombreuses évolutions technologiques et architecturales. Malheureusement, les techniques logicielles disponibles actuellement ne permettent pas d'exploiter pleinement les capacités du matériel. Les travaux présentés dans cette thèse apportent des solutions permettant d'améliorer la qualité des programmes et prenant en compte les évolutions architecturales. De bons résultats ont été obtenus pour des applications de calcul scientifique réelles et concernent à la fois les modèles d'exécution parallèle (distribué) et séquentiel, ce qui augmente la portée des travaux.

L'analyse de l'origine des facteurs limitant les performances lors de l'exécution des applications de calcul scientifique sur des machines modernes est à la base de cette étude. Nous avons pu ainsi concevoir et mettre en place de nouvelles techniques efficaces et directement applicables aux problèmes réels.

Originalité de l'étude

Plusieurs idées directrices contribuent à l'originalité de cette étude :

• Nous souhaitions tout d'abord axer nos efforts sur la prise en compte du contexte dans lequel se plaçaient nos travaux ; les modifications architecturales ainsi que l'évolution du type de programmes manipulés étaient donc au cœur de nos préoccupations. Les résultats présentés dans cette thèse apportent des solutions logicielles à l'essor du parallélisme, tant au niveau du calcul distribué sur plusieurs processeurs qu'à travers l'exploitation des capacités de calcul concurrent des processeurs VLIW et superscalaires. Par ailleurs, l'augmentation du nombre de programmes générés automatiquement ainsi que la volonté d'écrire des programmes portables et lisibles, donc peu optimisés, en utilisant des langages de plus en plus haut niveau, supposent un travail différent, encore plus important, de la part du compilateur.

• Les progrès réalisés dans le domaine de l'analyse automatique de programmes depuis quelques années ont ouvert de nouvelles perspectives. Les contributions de ce travail s'appuient donc sur les analyses disponibles (notamment dans l'outil `PIPS`) pour améliorer automatiquement la qualité des programmes.

• Partant de la constatation que la majorité des applications de calcul scientifique reposent sur l'évaluation de formules mathématiques de façon répétée dans des nids de boucles, nos travaux profitent des propriétés algébriques des opérateurs manipulés chaque fois que cela est possible.

Contributions

L'analyse des problèmes de performance réalisée et les idées directives que nous avons suivies, ont engendré un certain nombre de contributions augmentant les connaissances et techniques désormais exploitables dans le domaine de la compilation d'applications de calcul scientifique :

• L'efficacité du data-parallélisme sur machine parallèle à mémoire distribuée est limitée par le surcoût des communications et le manque de maturité des compilateurs. Nos travaux dans le domaine de la compilation du langage High Performance Fortran ont permis de mieux appréhender l'état d'avancement de ce langage et proposent des solutions concernant la diminution du surcoût des communications.

Nous avons montré au chapitre 4, à travers l'étude de 3 compilateurs HPF, que le portage d'une application réelle de Fortran 77 vers HPF pose des problèmes de performance et de portabilité. Nous avons également contribué à la mise en avant de certaines directives qui ne sont que des *extensions approuvées* dans la norme HPF2, mais qui s'avèrent utiles en pratique. Le travail expérimental réalisé a été présenté à la communauté HPF dans le cadre de la conférence HUG'98.

Nos expériences ont montré que la gestion des entrées-sorties dans les programmes HPF est un point critique pour les performances. En effet, le surcoût introduit par de trop nombreuses communications annihile le gain obtenu par la parallélisation des calculs. Nos travaux concernant la compilation des entrées-sorties ont montré qu'une analyse des accès mémoire permet de réduire considérablement le nombre de communications générées. Notre approche, présentée chapitre 5, est fondée sur l'utilisation des *effets* et des *régions de tableaux.*

Toutes les communications ne pouvant être supprimées, les recouvrements des calculs et des communications constituent un point clé pour l'obtention de hautes performances sur les machines à mémoire distribuée. Nous proposons une étude analytique du gain en performance pouvant être atteint par utilisation du pipeline multi-dimensionnel (chapitre 6). Nous montrons que, selon le type de distribution utilisé, le pipeline permet, à travers un double recouvrement calcul-communication et calcul-calcul, d'obtenir des gains intéressants. Une partie de ces travaux a été présentée lors de la conférence Renpar'9 (1997).

• Malgré l'augmentation des capacités de calcul concurrent des processeurs VLIW et superscalaires, les performances des applications scientifiques restent limitées car les techniques de compilations sont insuffisantes et entraînent une trop faible utilisation de ces ressources. Nos travaux prouvent que les propriétés algébriques des opérateurs manipulés peuvent être utilisées pour réécrire les expressions sous des formes mathématiquement équivalentes et mieux adaptées à une exécution avec parallélisme d'instructions. Les résultats présentés dans cette thèse montrent l'impact considérable que peuvent entraîner ces transformations sur le temps d'exécution d'applications de calcul scientifique.

La manipulation d'expressions à partir des propriétés algébriques des opérateurs suppose une représentation bien adaptée : nous avons montré au chapitre 8 que l'utilisation conjointe d'une représentation sous forme d'arbres n-aires et d'un mécanisme de réécriture d'arbres fondé sur du pattern-matching associatif-commutatif, est une méthode efficace. Nos travaux permettent d'appliquer des optimisations telles que l'évaluation partielle où la réduction de force dans un contexte plus favorable que celui généralement utilisé par les compilateurs. Nous pouvons également appliquer toutes sortes de simplifications algébriques permettant de réduire le nombre d'opérations à évaluer.

Grâce aux propriétés algébriques des opérateurs telles que la commutativité, l'associativité ou la distributivité, une expression peut, en général, exister sous différentes formes mathématiquement équivalentes. Nos expériences (cf. chapitres 7 et 10) ont montré que la transformation d'une expression sous une forme bien adaptée à une exécution sur processeurs superscalaires permet d'obtenir des gains significatifs en performance. Nous proposons au chapitre 9 une sélection entre les différentes expressions équivalentes fondée sur l'utilisation de critères prenant en compte le nombre d'opérations, la profondeur de l'arbre ou encore la profondeur moyenne pondérée des opérateurs dans l'arbre. Deux heuristiques fonctionnant par un mécanisme itératif de type glouton ont été mises en place pour factoriser les expressions et extraire les opérations de type *fma*. Toutes deux permettent de construire des expressions en favorisant le critère choisi comme par exemple l'équilibrage de l'arbre. Un algorithme optimal fondé sur une variation de l'algorithme de codage de Huffman a été mis en place afin de construire des arbres binaires minimisant la *gravité* des expressions.

Ce processus en plusieurs étapes permet donc de construire des expressions offrant plus de choix au compilateur pour l'ordonnancement des calculs indépendants. Nous avons pu valider l'efficacité de cette approche par l'application automatique des transformations à des applications de calcul scientifique. Ces travaux ont été présentés lors de la conférence PACT'98 [ZC98].

Les recherches dans le domaine de la compilation se sont concentrées principalement sur l'étude des optimisations à partir d'une représentation sous forme de code 3-adresses. Or, de nombreuses opportunités d'optimisation sont trop complexes à gérer pour le programmeur et nécessitent l'utilisation d'une représentation de haut niveau. Nous proposons l'exploitation des propriétés algébriques des opérateurs pour extraire de nouvelles opportunités d'optimisations de boucles.

Nous proposons au chapitre 13 un algorithme de déplacement d'*instructions* invariantes fondé sur l'analyse et la simplification du graphe de dépendances. Cette technique permet de manipuler directement des accès à des éléments de tableaux et des appels de fonctions. Une restructuration du contrôle basée sur le mécanisme des instructions gardées permet également d'extraire des calculs invariants qui ne sont pas déplacés par les techniques classiques de PRE.

La réassociation des opérations associatives-commutatives permet d'effectuer un découpage en sous-expressions invariantes. De nombreuses opportunités nouvelles de déplacement de code invariant sont ainsi mises en évidence. Le chapitre 14 présente une technique permettant de travailler directement sur des expressions complexes sans atomisation inutile.

L'utilisation de l'associativité et de la commutativité des opérateurs permet de mettre en évidence de nouvelles opportunités d'élimination de sous-expressions communes (cf. chapitre 15). Nous proposons une heuristique travaillant sur une séquence d'instructions et recherchant les sous-expressions dans un contexte AC. L'approche proposée permet de gérer les phénomènes d'alias, ne nécessite pas de comparaison entre sous-arbres et permet d'éviter toute atomisation excessive.

L'approche source-à-source que nous avons choisie pour l'implémentation de ces techniques permet d'optimiser facilement le code généré automatiquement par un compilateur. Ce type de code présente en effet de nombreuses opportunités d'optimisations nécessitant l'utilisation des propriétés algébriques des opérateurs.

Réalisations effectives

Les techniques présentées dans cette thèse ont également fait l'objet d'un travail d'implémentation permettant de valider les résultats manuels ou théoriques obtenus par de véritables expérimentations sur des applications de calcul scientifique (cf. chapitre 5 section 5.3, chapitre 6 section 6.3, chapitre 10 et chapitre 16). Tous ces travaux sont disponibles sous la forme de prototypes qui reposent sur les outils `HPFC`, `PIPS`, `STORM`, `CAVEAT` et `EOLE`.

Travaux futurs

Les perspectives futures de recherche s'inscrivent tout naturellement dans la continuité des travaux présentés ici. D'une manière générale, nous chercherons tout d'abord à confirmer nos résultats par un plus grand nombre d'expériences. La validation des prototypes devrait permettre par la suite d'utiliser ces techniques de manière automatique sur des applications de grandes tailles, dans le cadre de contrats avec des partenaires industriels ou universitaires. Nous tâcherons également d'augmenter la portée des travaux présentés par l'utilisation des techniques proposées dans un contexte différent.

Outre les surcoûts engendrés par les communications entre processeurs, l'obtention des performances souhaitées pour les programmes data-parallèles est rendue difficile par la qualité médiocre des programmes générés automatiquement. Les techniques présentées dans cette thèse sont une première source d'amélioration possible. Toutefois, d'autres problèmes viennent de l'utilisation trop intensive de code de contrôle. Certaines boucles ont par exemple un domaine d'itération vide ou se comportent comme des simples tests. Un gros travail doit donc être fait pour permettre d'améliorer la qualité de ces codes.

Nous souhaitons poursuivre l'exploitation des transformations algébriques dans le but de sélectionner la structure des expressions. En effet, il serait intéressant d'évaluer à quel point le choix du couple (compilateur, processeur) intervient sur le processus de sélection des expressions. En effet, nous avons jusqu'à présent décidé de privilégier une approche fondée sur l'analyse des besoins en terme de parallélisme d'instructions, sans prendre en compte les spécificités d'un processeur donné. Cela permet d'augmenter la portée des transformations. Il serait toutefois intéressant de connaître l'impact de paramètres tels que le nombre de registres, le nombre d'opérations lancées à chaque cycle, le nombre d'opérations indépendantes nécessaires pour remplir en continu le pipeline d'exécution, etc. sur la sélection des expressions. Nous ne sommes actuellement pas en mesure de donner des réponses précises à ces questions.

Lorsque la pression de registres est très importante, nous avons pu constater qu'il est souvent intéressant de distribuer une boucle contenant un gros volume de calculs en deux boucles moins « denses ». En effet, le nombre de registres utiles pour le calcul d'une itération de la boucle est parfois trop important, ce qui aboutit à un nombre insuffisant de registres. Or, en appliquant un découpage des expressions pour former des sous-expressions de coûts équivalents, il est possible de réduire considérablement la pression de registres. Une partie des calculs est mémorisée dans une variable temporaire puis réutilisée dans la seconde boucle. Il est évident que la gestion des accès mémoire doit être analysée également pour ne pas entraîner une augmentation du nombre de défauts de cache ; selon le découpage choisi, le comportement vis-à-vis du cache peut être soit amélioré, soit au contraire dégradé. Nous n'avons pour l'instant pas cherché à caractériser plus précisément le contexte d'application de cette technique. Les constatations expérimentales obtenues ont été le fruit d'un découpage manuel. La figure 18.1 donne un exemple d'application de cette technique pour le benckmark `mgrid` de la suite NAS. Des mesures ont été effectuées sur un processeur P2SC cadencé à 160MHz avec la version 4.1 du compilateur `IBM xlf` en utilisant les options `-qarch=pwr2 -qhot -O3`. La transformation présentée figure 18.2 a permis d'obtenir un gain de 10% sur le temps d'exécution de la routine `resid` qui contient cette boucle et prend 42% du temps de calcul global. Les performances de l'application complète sont passées de 90 à 97 MFLOP/s. Nous avons également mesuré les performances sur un processeur UltraIIi cadencé à 270MHz en compilant avec la version 5.0 du compilateur `SUN`. Le gain obtenu pour l'application complète est de 34% sur le temps d'exécution total.

Nous pensons qu'il est possible d'aller encore plus loin dans le processus d'exploitation des transformations algébriques pour les optimisations de boucles. Tout d'abord, nous souhaitons étendre les traitements des opérateurs de type complément de manière à reconnaître des sous-expressions communes plus complexes. Par exemple, l'expression $b-(a+f-c)$ est calculée au signe près dans $a-(b+c+d)+e+f$. D'autre part, nous n'avons pour l'instant cherché à exploiter les transformations algébriques que pour deux optimisations de boucles : le déplacement de code invariant et l'élimination de sous-expressions communes. Il serait également intéressant d'évaluer les opportunités existant dans le cadre d'autres transformations comme par exemple l'élimination des variables d'induction. L'optimisation des calculs d'adresses est également susceptible de profiter des résultats présentés dans cette thèse.

```
     do 600 i3=2,n-1
     do 600 i2=2,n-1
     do 600 i1=2,n-1
600  r(i1,i2,i3)=v(i1,i2,i3)
    >  -a(0)*( u(i1,  i2,  i3  ) )
    >  -a(1)*( u(i1-1,i2,  i3  ) + u(i1+1,i2,  i3  )
    >       +  u(i1,  i2-1,i3  ) + u(i1,  i2+1,i3  )
    >       +  u(i1,  i2,  i3-1) + u(i1,  i2,  i3+1) )
    >  -a(2)*( u(i1-1,i2-1,i3  ) + u(i1+1,i2-1,i3  )
    >       +  u(i1-1,i2+1,i3  ) + u(i1+1,i2+1,i3  )
    >       +  u(i1,  i2-1,i3-1) + u(i1,  i2+1,i3-1)
    >       +  u(i1,  i2-1,i3+1) + u(i1,  i2+1,i3+1)
    >       +  u(i1-1,i2,  i3-1) + u(i1-1,i2,  i3+1)
    >       +  u(i1+1,i2,  i3-1) + u(i1+1,i2,  i3+1) )
    >  -a(3)*( u(i1-1,i2-1,i3-1) + u(i1+1,i2-1,i3-1)
    >       +  u(i1-1,i2+1,i3-1) + u(i1+1,i2+1,i3-1)
    >       +  u(i1-1,i2-1,i3+1) + u(i1+1,i2-1,i3+1)
    >       +  u(i1-1,i2+1,i3+1) + u(i1+1,i2+1,i3+1) )
```

FIG. 18.1 – *Un extrait du benchmark* `mgrid` *(NAS).*

```
  do i3=2,n-1
    do i2=2,n-1

       do i1=2,n-1
          tmp(i1) =
>      -a(1)*( u(i1-1,i2,  i3  ) + u(i1+1,i2,  i3  )
>         +  u(i1,  i2-1,i3  ) + u(i1,  i2+1,i3  )
>         +  u(i1,  i2,  i3-1) + u(i1,  i2,  i3+1) )
>      -a(2)*( u(i1-1,i2-1,i3  ) + u(i1+1,i2-1,i3  )
>         +  u(i1-1,i2+1,i3  ) + u(i1+1,i2+1,i3  )
>         +  u(i1,  i2-1,i3-1) + u(i1,  i2+1,i3-1)
>         +  u(i1,  i2-1,i3+1) + u(i1,  i2+1,i3+1)
>         +  u(i1-1,i2,  i3-1) + u(i1-1,i2,  i3+1)
>         +  u(i1+1,i2,  i3-1) + u(i1+1,i2,  i3+1) )
        enddo

        do i1=2,n-1
           r(i1,i2,i3)= v(i1,i2,i3)
>      -a(0)*( u(i1,  i2,  i3  ) )
>      + tmp(i1)
>      -a(3)*( u(i1-1,i2-1,i3-1) + u(i1+1,i2-1,i3-1)
>          +  u(i1-1,i2+1,i3-1) + u(i1+1,i2+1,i3-1)
>          +  u(i1-1,i2-1,i3+1) + u(i1+1,i2-1,i3+1)
>          +  u(i1-1,i2+1,i3+1) + u(i1+1,i2+1,i3+1) )
        enddo
    enddo
  enddo
```

FIG. 18.2 – *Distribution de l'expression sur 2 boucles pour réduire la pression de registres.*

The end.

Annexe A

Optimalité de la réassociation

Nous montrons que la réassociation des opérandes pour former des sous-expressions invariantes, présentée au chapitre 14 est optimale. Ce travail a été réalisé par Fabien Coelho, Corinne Ancourt et moi-même.

Note aux rapporteurs : La rédaction de la preuve n'étant pas terminée, elle sera integrée ultérieurement ou donnera lieu à une publication sous la forme d'un rapport technique interne au CRI.

Annexe B

PIPS Overview

PIPS est un outil expérimental permettant d'implanter puis évaluer des transformations interprocédurales dans le cadre de la compilation et de la parallélisation de programmes scientifiques. Ce travail repose sur de nombreuses analyses disponibles. Nous reproduisons simplement ici, avec la permission de son auteur principal Ronan Keryell, une description constituant le rapport interne *EMP CRI A-289* [KAC+96]. Une présentation plus récente sous forme de transparents et décrivant une partie des travaux de cette thèse est également disponible à partir de l'url `http://www.cri.ensmp.fr/pips` (dans la partie *Bibliography*).

B.1 Introduction

Detecting the maximum level of parallelism in sequential programs has been a major source of interest during the last decade. This process is of utmost importance when trying to cope with the increasing number of vector and parallel supercomputers and, perhaps unfortunately, the huge number of already existing "dusty deck" sequential programs. Nowadays, since the development of high-end parallel computers has not fulfilled the wildest hopes of entrepreneurial companies in the parallel supercomputing business, we can expect more down-to-earth developments of work-group parallel servers and workstations with few but powerful processors.

Together with this architectural evolution, three main directions in compiler developments have been pursued: compilers for sequential machines (with superscalar processors), parallelizers of sequential programs and compilers for explicit parallel programs. All these approaches benefit from deep global program analyses, such as interprocedural and semantical analyses, to perform optimizations, vectorization, parallelization, transformations, restructuring and reverse-engineering of programs. Since these approaches often require the same or share a common ground of analyses, it is interesting to factorize them out in a common development tool to get the benefit of code re-use and modular programming.

PIPS is such a highly modular workbench for implementing and assessing various interprocedural compilers, optimizers, parallelizers, vectorizers, restructurers, etc. without having to build a new compiler from scratch. It has been used in various compilation areas since its inception in 1988 as the PIPS project (Interprocedural Parallelization of Scientific Programs). PIPS has been developed through several research projects funded by DRET (French ARPA), CNRS (French NSF) and the European Union (ESPRIT programs). The initial design was made by Rémi TRIOLET, François IRIGOIN and Pierre JOUVELOT. It has proved good enough since then not to require any major change.

Our project aims at combining advanced interprocedural and semantical analyses [TFI86] with a requirement for compilation speed. The mathematical foundations of the semantical analysis implemented in PIPS are linear programming and polyhedra theory. Despite such advanced techniques, PIPS is able to deal with real-life programs such as benchmarks provided by ONERA (French research institute in aeronautics), or the Perfect Club in quite reasonable time. An excerpt of a fluid dynamics code for a wing analysis (Figure B.2) will be used as a running example in this paper.

PIPS is a multi-target parallelizer. It incorporates many code transformations and options to tune its features and phases. PIPS offers interesting solutions to solve programming design problems such as generating high-level data structures from specifications, co-generating their documentation, defining user interface description and configuration files, dealing with object persistence and resource dependences, being able to write parts of PIPS in various languages, etc.

This paper details how PIPS achieves these goals. We first present a general overview of PIPS in the next section. In Section B.3 we describe the mechanisms that deal with PIPS data structures, their dependences and their interprocedural relationships. The general design is discussed in Section B.3. The programming environment, including the data structures generator and the linear library, and the documentation process is presented in Section B.4. At last, the different user interfaces in Section B.5 are presented before the related work in Section B.6.1.

B.2 Overview

PIPS is a source-to-source Fortran translator (Figure B.1), Fortran still being the language of choice for most scientific applications. Beside standard Fortran 77, various dialects can be used as input (HPF) or output (CMF, Fortran 77 with Cray directives, Fortran 77 with HPF parallelism directives, etc.) to express parallelism or code/data distributions. Fortran can be seen here as a kind of portable language to run on various computers without having to deal with an assembly code back-end, although the PIPS infrastructure could be used as a basis to generate optimized assembly code. Other parsers or prettyprinters could be added and the internal representation could be extended if need be to cope with other imperative languages such as C.

Following the field research history, PIPS has first been used to improve the vectorization of Fortran code for parallel vector computers with shared memory (Fortran 77 with `DOALL`, Fortran 77 with Cray micro-tasking directives, Fortran 90). It is now mainly targeted at generating code for distributed memory machines, using different methods (processor and memory bank code for control distribution [AI92], CMF, CRAFT polyhedral method [Pla95a] or message-passing code from HPF [Coe94]).

As we see in Section B.3, the translation process is broken into smaller modular operations called phases in PIPS. For the user, these phases can be classified into various categories according to their usage. Since PIPS is an interprocedural environment, procedures and functions are very important in PIPS and are referred to as *module* in the sequel.

B.2.1 Analyses

An analysis phase computes some internal information that can be later used to parallelize (or generate) some code, some user information (such as a program complexity measure) to be later displayed by a prettyprinter, etc. The most interesting analyses available in PIPS are the semantical analyses described below.

Effects

Since an imperative language such as Fortran deals with memory locations, the basic semantical analyses in PIPS deal with the effects of instructions on memory. First, the *proper* effects are computed for each sub-statement of a module: a read or write operation on an array element, the update of an index variable in a `DO` loop, etc. These effects can be displayed to the user *via* a prettyprinter like:

```
C       <must be read   >: J NC S1 S2 S3
C       <must be written>: T(J,1,NC+5)
           T(J,1,NC+5) = S1*S2/((S3-S1)*(S3-S2))
```

Then, since the abstract syntax tree is recursively defined in PIPS, the *cumulated* effects are recursively aggregated from all the proper effects of the sub-statements in a statement. For the loop 300 on Figure B.2 this leads to:

```
C       <may be read    >: J J2 JH K L NC S1 S2 S3 T(*,*,*)
C       <may be written >: JH S1 S2 S3 T(*,*,*)
C       <must be read   >: J1 JA
C       <must be written>: J
```

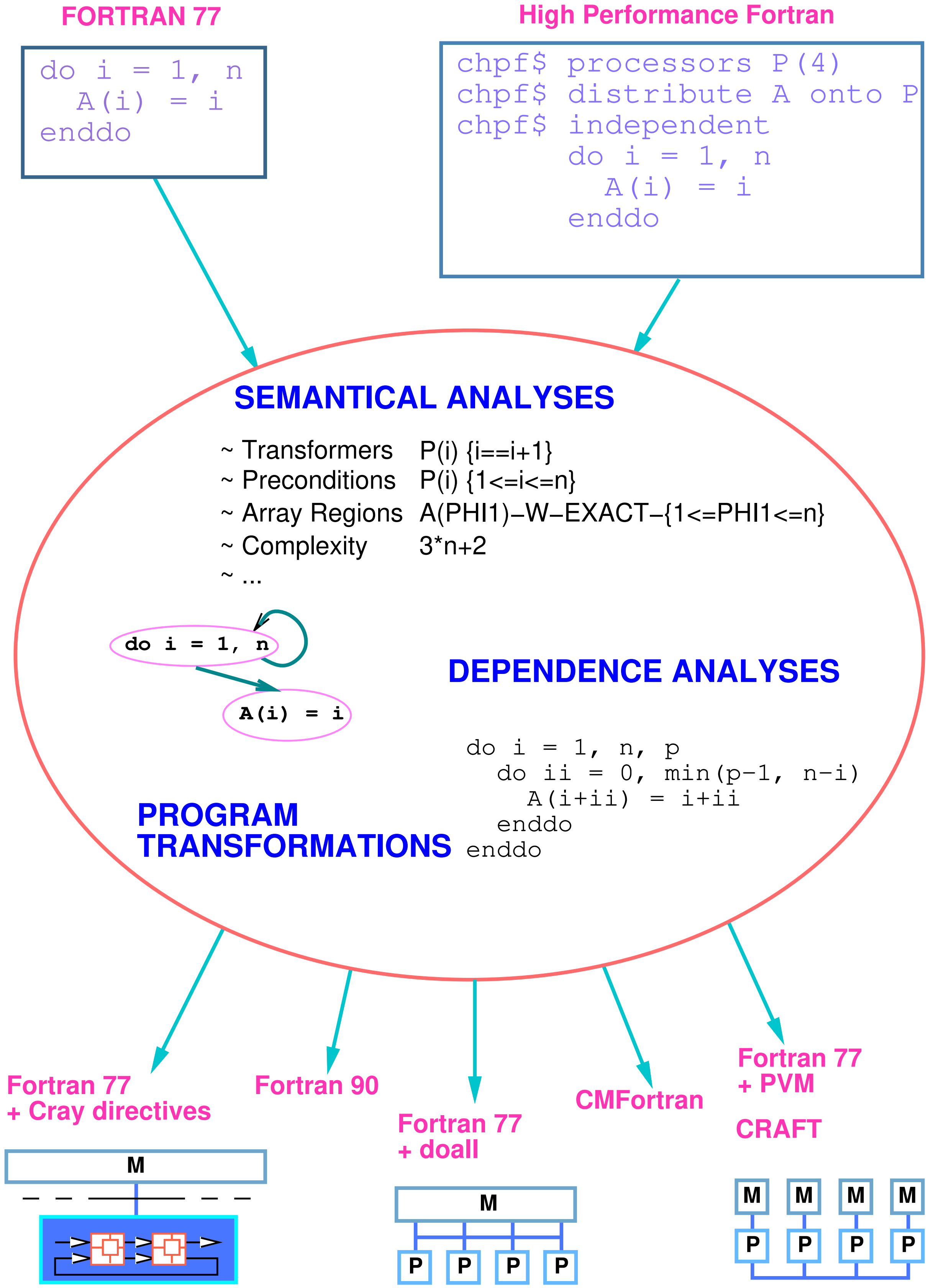


Fig. B.1 – *User view of the* Pips *system.*

```
c
      PROGRAM EXTRMAIN
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,...
      COMMON/CJ/J1,J2,JA,...
      COMMON/CK/K1,K2,...
      COMMON/CNI/L
      DATA N1,N4,N7,N17 /1,4,7,17/

      READ(NXYZ) I1,I2,J1,JA,K1,K2
      REWIND NXYZ
c
      IF(J1.GE.1.AND.K1.GE.1) THEN
         N4=4
         J1=J1+1
         J2=2*JA+1
         JA=JA+1
         K1=K1+1
         CALL EXTR(N7,N17)
         CALL EXTR(N4,N17)
         CALL EXTR(N1,N17)
      ENDIF
      END
c
c
c Compute the extrapolation coefficients
c for all the wing area (K=K1)
c
      SUBROUTINE EXTR(NI,NC)
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CI/I1,I2,...
      COMMON/CJ/J1,J2,JA,...
      COMMON/CK/K1,K2,...
      COMMON/CNI/L
      L=NI
      K=K1
      DO 300 J=J1,JA
         S1=D(J,K  ,J,K+1)
         S2=D(J,K+1,J,K+2)+S1
         S3=D(J,K+2,J,K+3)+S2
         T(J,1,NC+3)=S2*S3/((S1-S2)*(S1-S3))
         T(J,1,NC+4)=S3*S1/((S2-S3)*(S2-S1))
         T(J,1,NC+5)=S1*S2/((S3-S1)*(S3-S2))
         JH=J1+J2-J
         T(JH,1,NC+3)=T(J,1,NC+3)
         T(JH,1,NC+4)=T(J,1,NC+4)
         T(JH,1,NC+5)=T(J,1,NC+5)
 300  CONTINUE
      END
c
c
c Compute D=DISTANCE
c
      REAL FUNCTION D(J,K,JP,KP)
c
      DIMENSION T(52,21,60)
      COMMON/CT/T
      COMMON/CNI/L
c
      D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1      +(T(J,K,L+1)-T(JP,KP,L+1))**2
     2      +(T(J,K,L+2)-T(JP,KP,L+2))**2)
      END
```

FIG. B.2 – *Code excerpt from an* ONERA *benchmark.*

```
      DO 300 J = J1, JA
         ...
```

Since having information about effects on internal variables in a function is irrelevant to calling procedures, the *summary* effects of a procedure are constructed by removing these local effects from the cumulated effects of the procedure. Interprocedural propagation will be commented in Section B.3.4.

Transformers

To improve the accuracy of the effect information, to provide more information for dependence testing and to select better program transformations, statements are labeled with preconditions that express constraints on integer scalar variables. Since such variables are often used in array references and loop bounds, preconditions can often precisely define which subpart of an array is referenced by a memory reference expression, thus refining cumulated effects into Regions (Section B.2.1.0).

Mathematically speaking, PIPS preconditions are computed via transformers that are abstract commands mapping a store to a set of stores [IJT91, Iri92]. They are relations between the values of integer variables in an initial store and those in a final store. In PIPS, the considered relations are polyhedra, represented by systems of linear equalities and inequalities, because they provide a sound and general mathematical framework for reasoning about array accesses which are mostly linear in practice. Below, two statement transformers are given by the prettyprinter.

```
C  T(J1) {J1==J1#init+1}
         J1 = J1+1
C  T(J2) {J2==2JA+1}
         J2 = 2*JA+1
```

Constant propagation, inductive variable detection, linear equality detection or general linear constraints computation can be performed in this workbench using different operators to deal with PIPS basic control structures. To deal with unstructured control graphs, some fancier operators requiring a fix point computation are needed, although they may slow down the analysis. To provide a better trade-off between time and precision to the user, options are available to tune the precondition construction algorithms.

Unlike most analysis in the field, we are dealing with abstract commands instead of abstract stores for two reasons. First, variables appearing in references cannot be aggregated to build cumulated and summary effects unless they denote the same values, that is they refer to the same store. The module's initial store is a natural candidate to be the unique reference store and, as a result, a relationship between this store and any statement's initial store is needed. This relationship is called a precondition [Iri92] in PIPS and is constructed for a statement by aggregating all the transformers of the statements on the paths from the beginning of the module up to that statement. On the same example than the transformers we have:

```
C  P(J1,N) {N==1, 1<=J1, ...}
         J1 = J1+1
C  P(J1,N) {N==1, 2<=J1, ...}
         J2 = 2*JA+1
C  P(J1,J2,JA,N) {J2==2JA+1, N==1, 2<=J1, ...}
```

Second, dependence tests are performed between two statements, Although a relationship between each statement store would be more useful than two stores, each relative to one statement, this would not be realistic because too many predicates would have to be computed. Thus, a common reference to the same initial store seems to be a good trade-off between accuracy and complexity.

Dependence test

Since the dependence graph [YAI94] is used within many parts of PIPS, such as the parallelizers, the validity check for some transformations, the use-def elimination phase, etc., a multi-precision test has been implemented to provide various degrees of precision and speed. First, a crude by fast algorithm is used and if no positive or negative conclusion is found, a more precise but more compute-intensive test is used, and so on [Yan93].

PIPS dependence tests compute both dependence levels and dependence cones. Dependence cones can be retrieved to implement advanced tiling and scheduling methods.

Regions

Array Regions [CI95, CI96] are used in PIPS to summarize accesses to array elements. READ and WRITE Regions represent the effects of statements and procedures on sets of array elements. They are used by the dependence test to disprove interprocedural dependences. For instance, in our example, the READ regions for the sole statement of function D are:

```
C      <T(φ1,φ2,φ3)-R-EXACT-{φ1==J,K<=φ2<=K+1,L<=φ3<=L+2}>

            D=SQRT((T(J,K,L  )-T(JP,KP,L  ))**2
     1      +      (T(J,K,L+1)-T(JP,KP,L+1))**2
     2      +      (T(J,K,L+2)-T(JP,KP,L+2))**2)
```

where the ϕ variables represent the three dimensions of `A`. To compute this Region, interprocedural preconditions must be used to discover the relations between the values of `J` and `JP`, and `K` and `KP`.

However, since READ and WRITE Regions do not represent array data flow, they are insufficient for advanced optimizations such as array privatization. IN and OUT regions have thus been introduced for that purpose: for any statement or procedure, IN regions contain its imported array elements, and OUT regions contain its exported array elements. The possible applications are numerous. Among others, IN and OUT regions are already used in PIPS to privatize array sections [AC94, Cre95], and we intend to use them for memory allocation when compiling signal processing specifications based on dynamic single assignments.

Another unique feature of PIPS array Regions lies in the fact that, although they are over-approximations of the element sets actually referenced, they are flagged as exact whenever possible. Beside a theoretical interest, there are specific applications such as the compilation of HPF in PIPS.

Array Data Flow Graph

This sophisticated data flow graph extends the basic DFG with some kind of array Regions information with linear predicates. Thus, it is able to finely track array elements accesses according to different parameters and loop indices in programs. However, this is only valid when the whole control can be represented in a linear framework [Pla95b, Pla95a].

Complexities

When optimizing some code, or assessing whether to parallelize some program fragment or apply a transformation, it is quite useful to have some estimation of the program time complexity. A static complexity analysis phase has been added to PIPS to give polynomial complexity information about statements and modules [Zho94] from preconditions and other semantical information.

B.2.2 Code generation phases

The result of program analyses and transformations is a new code. These phases generate (1) parallel code from data such as sequential code, semantical informations and the dependence graph, or (2) a message passing code for distributed memory architectures.

HPF Compiler

HPFC is a prototype HPF compiler implemented as a set of PIPS phases. From Fortran 77, HPF static (`align distribute processors template`), dynamic (`realign redistribute dynamic`) and parallelism (`independent new`) directives, it generates portable PVM-based SPMD codes which have run on various parallel architectures (SUN NOW [Coe94], DEC alpha farm, TMC CM5, IBM SP2).

Implementing such a prototype within the PIPS framework has proven to be a fine advantage: First, the integration of the various phases (Figure B.3) within the PipsMake system allows to reuse without new developments all PIPS analyses, and to benefit from the results of these analyses for better code generation and optimizations; Second, the Linear C^3 library has provided the mathematical framework [ACIK93, ACIK96] and its implementation to develop new optimizations techniques targeted at generating efficient communication codes. It has been applied to I/O communications [Coe95] and general remappings [CA96].

Phase	Function
`hpfc_filter`	preprocessing of the file
`hpfc_parser`	Fortran and HPF directive parser
`hpfc_init`	initialization of the compiler status
`hpfc_directives`	directive analysis
`hpfc_compile`	actual compilation of a module
`hpfc_close`	generation of global runtime parameters
`hpfc_install`	installation of the generated codes
`hpfc_make`	generation of executables
`hpfc_run`	execution of the program under PVM

FIG. B.3 – HPFC *phases in* PIPS

Parallelization

The primary parallelization method used for shared-memory machines is based on Allen&Kennedy algorithm and on dependence levels. This algorithm was slightly modified to avoid loop distribution between statements using the same private variable(s). The resulting parallel code can be displayed in two different formats based on Fortran 77 and Fortran 90. Parallel loops are distinguished by the `DOALL` keyword.

Another parallelization algorithm was derived from Allen&Kennedy's to take into account the Cray specificities, the vector units and the micro-tasking library. Two levels of parallelism are selected at most. The inner parallelism is used to generate vector instructions and the outer one is used for task parallelism.

Code distribution

This parallelization method emulates a shared memory machine on a distributed architecture with programmable memory banks. It splits the program in two parts: a computational part and a memory part. More precisely, two programs are generated by using a method based on linear algebra: one SPMD program that does the computations in a block distributed control way and another SPMD program that manages the memory accesses.

The development run-time is implemented over PVM but the target is Inmos T9000 with C104 transputer-powered parallel machines with an hight speed interconnection network. This prototype was built as part of the European Puma project [AI92].

Parallelization with a polyhedral method

By using the Array Data Flow Graph, it is possible to track the movement of each value in a program where control flow can be statically represented in a linear way. Using this information, a schedule is generated to exploit the maximum parallelism and a placement is selected to reduce the number of communications according to their nature without discarding all the parallelism [Pla95b, Pla95a].

B.2.3 Transformations

A transformation in PIPS takes an object, such as the internal representation of the code, and generates a new version of this object. Transformations currently implemented in PIPS are:

- loop distribution;
- scalar and array privatization based on Regions ;
- atomizer to split statements in simpler ones such as `A=B op C`;
- loop unrolling;
- loop strip-mining;
- loop interchange using an unimodular transformation;
- loop normalization;
- partial evaluation;
- dead-code elimination based on preconditions;
- use-def elimination;
- control restructurer;
- reduction detection.

B.2.4 Pretty-printers

Pretty-printers are used to transform some PIPS internal representation into a human readable file that can be displayed by the user interface. There are pretty-printers to display the original parsed code, the sequential code, the parallel code in some parallel dialect, the dependence graph, the call graph, etc. Other specialized pretty-printers exist to add some informative decoration to the previous ones, such as displaying the code with regions, preconditions, complexities, etc. There are implemented by using hooks in the more classical pretty-printers.

B.3 PIPS general design

For a PIPS developer, PIPS can be viewed as in Figure B.4. Of course the user interfaces control PIPS behavior but, internally, all the compiling, analyzing and transformation operations are split into basic

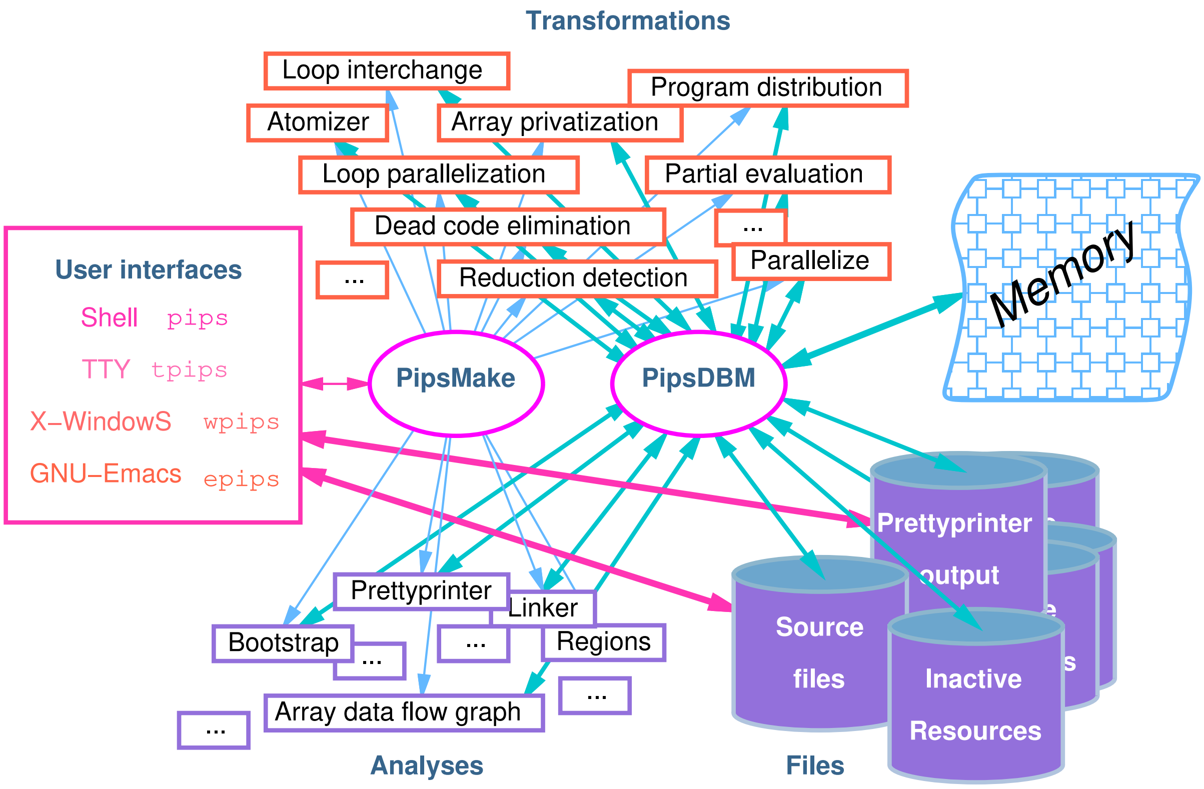


FIG. B.4 – *Overview of the* PIPS *architecture.*

independent *phases*. They are scheduled by PipsMake, an *à la* `make` interprocedural mechanism that deals with *resources* through the PipsDBM data base manager. An input program is first split into modules that correspond to its subroutines and functions. PipsMake can then apply the transformation phases on these modules according to some predefined rules. The result is a streamlined programming interface with as few side effects as possible between phases. This allows a highly modular programming style with several developers at the same time. To add a phase to PIPS, only a C function (which may be a wrapper if the execution part of the phase is written in another language) is added (which may of course call other functions) that is called by PipsMake with a module name as argument.

B.3.1 Resources

All the data structures that are used, produced or transformed by a phase in PIPS are called resources. A phase can request some data structures in PipsDBM, use or modify them, create some other resources, and then store with PipsDBM the modified or created resources to be used later. Examples of resources are the input code, abstract syntax tree, control flow graph, use-def chains, output code, semantic information such as *effects*, *preconditions*, *regions*, etc. These resources are described with the NEWGEN description language (Section B.4.1) that allows transparent access to persistent data in PipsDBM from different languages.

B.3.2 Workspace

Since all the resources about a user code may be stored on disk (Section B.3.5), resources are located in a workspace directory. This workspace is created from the source file given by the user and also contains a log file and a PipsMake data structure (Section B.3.4 that encodes a coherent description of its content).

B.3.3 Phases

Each phase in PIPS use the same interface and simply is a C function taking a module name as argument and returning a boolean describing its completion status (success or failure). Figure B.5 shows the example of the dead code elimination phase.

Since PIPS core is written in C, a wrapper function is needed to program PIPS in other programming languages. Currently, the only phase written in Common-Lisp is the reduction detection phase: it calls a Common-Lisp process to perform the job. Communications are handle through PipsDBM files.

B.3.4 PipsMake consistency manager

From a theoretical point of view, the object types and functions available in PIPS define an heterogeneous algebra with constructors (*e.g.* parsers), extractors (*e.g.* prettyprinters) and operators (*e.g.* loop unrolling). Very few combinations of functions make sense, but many functions and object types are available. This abundance is confusing for casual and even experienced users and it was deemed necessary to assist them by providing default computation rules and automatic consistency management.

The PipsMake library – not so far from the Unix make utility – is intended for interprocedural use. The objects it manages are resources stored in memory or/and on disk. The phases are described in a `pipsmake-rc` file by generic rules that use and produce resources. Some examples are shown on Figure B.6. The ordering of the computation is demand-driven, dynamically and automatically deduced from the `pipsmake-rc` specification file. Since it is a quite original part of PIPS, it is interesting to describe a little more the features of PipsMake.

Rule format

For each rule, that is in fact the textual name of a phase C function, it is possible to add a list of constraints composed with a qualifier, an owner name and a resource name. Each constraint has the following syntax:

$$< \textit{qualifyer} > < \textit{owner-name} > . < \textit{resource-name} >$$

```
bool suppress_dead_code(string module_name)

  /* get the resources */
  statement module_stmt = (statement)
      db_get_memory_resource(DBR_CODE, module_name, TRUE);
  set_proper_effects_map((statement_mapping)
      db_get_memory_resource(DBR_PROPER_EFFECTS, module_name, TRUE));
  set_precondition_map((statement_mapping)
      db_get_memory_resource(DBR_PRECONDITIONS, module_name, TRUE));

  set_current_module_statement(module_stmt);
  set_current_module_entity(local_name_to_top_level_entity(module_name));

  debug_on("DEAD_CODE_DEBUG_LEVEL");
  /* really do the job here: */
  suppress_dead_code_statement(module_stmt);
  debug_off();

  /* returns the updated code to Pips DBM */
  DB_PUT_MEMORY_RESOURCE(DBR_CODE, module_name, module_stmt);
  reset_current_module_statement();
  reset_current_module_entity();
  reset_proper_effects_map();
  reset_precondition_map();

  return TRUE;
```

FIG. B.5 – *Excerpt of the function of the dead code elimination phase.*

The qualifier describing the resource can be for each resource:

`<`: needed;

`>`: generated;

`#`: destroyed by the rule;

`=`: asserts that the resource remains valid even if other dependence rules would suggest to invalidate the resource;

`!`: asks for applying an other rule before executing the rule. Thus, this constraint has a special syntax since it composed with a rule instead of a resource name. In Figure B.6 for instance `hpfc_close` needs an alternate parser for HPF, the must Regions and the directives of all the modules of the program.

This qualifier is thus more complex than an Unix `make` but allows finer tuning to avoid useless computation (for example with the =) and be able to expand the rule dependence behavior with !.

An owner name specifies the owner of the needed resource. For example for the code resource, only the code of this module, or the code of all the modules, etc:

MODULE: the current module itself;

MAIN: the main module if any (that is the Fortran `PROGRAM`);

ALL: all the modules

CALLEES: the set of the called modules by the current one; it allows to describe bottom-up analyses on the callgraph.

CALLERS: the set of the calling modules of the current one; it expresses top-down analyses.

PROGRAM: a resource that is attached to the whole program, such as the entities of a program. This concept is orthogonal to the module concept;

SELECT: select a rule by default.

A rule (that is a phase for a PIPS user) is conventionally called a transformation when a resource name appears in the rule with both $<$ and $>$ qualifiers. For example dead code elimination which transforms the code both needs the code and writes it.

```
proper_effects    > MODULE.proper_effects
        < PROGRAM.entities
        < MODULE.code
        < CALLEES.summary_effects

cumulated_effects > MODULE.cumulated_effects
        < PROGRAM.entities
        < MODULE.code MODULE.proper_effects

summary_effects   > MODULE.summary_effects
        < PROGRAM.entities
        < MODULE.code
        < MODULE.cumulated_effects

hpfc_close          > PROGRAM.hpfc_commons
    ! SELECT.hpfc_parser
    ! SELECT.must_regions
    ! ALL.hpfc_static_directives
    ! ALL.hpfc_dynamic_directives
    < PROGRAM.entities
    < PROGRAM.hpfc_status
    < MAIN.hpfc_host

use_def_elimination > MODULE.code
        < PROGRAM.entities
        < MODULE.code
        < MODULE.proper_effects
        < MODULE.chains
```

FIG. B.6 – *Example of `pipsmake-rc` rules.*

It is possible to chose between several rules to make a resource. By default the first one found in `pipsmake-rc` is selected but the programmer can choose to activate an other default rule through the API (see Section B.3.4.0 or with the **SELECT** owner). This feature is used for example in the user interfaces to select the kind of decorations in the prettyprinters. For instance the sequential code can be built by several phases, one for the plain code, another one for the plain code with Regions, etc.

Up to date resources & API

PipsMake is responsible for dynamically activating rules to build up to date resources and iterates through the transitive closure of the resource dependences (something Unix's `make` is unfortunately unable to do). A small programming interface is available to drive PipsMake from user interfaces (Section B.5) or other phases :

- `make(string resource_name, string owner_name)` builds the resources `resource_name` for the set of modules or the program described by `owner_name`;
- `apply(string phase_name, string owner_name)` applies the phase `phase_name` to the set of modules or the program described by `owner_name`;
- `activate(string phase_name)` activates this rule as the new default one when more than one rule could be used to build some resources. Since with this new default rule these generated resources are no longer up to date, the resources that can generated by the new default rule and that existed before are recursively deleted.

Interproceduralty

Interprocedurality is dealt with by the PipsMake **CALLEES** and **CALLERS** owner features: A bottom-up algorithm on the interprocedural call graph such as effects or interprocedural transformers computation will have some rules using **CALLEES**, while a top-down algorithm such as interprocedural preconditions will have some rules using **CALLERS** so as to properly order the phase execution and propagate resources interprocedurally. The correct ordering of the computation is deduced from the `pipsmake-rc` file.

For instance, effects computation relies on the three dependence rules shown on Figure B.6: First the proper effects are computed from the code and the summary effects of the **CALLEES** to get the summary effects of the modules called in the current module. Second these proper effects, that only relate to each simple statement, are transformed into the summary effects that take into account the hierarchical structure of the abstract syntax tree. Third, the summary effects are computed to describe the effects that are only non local to the current module and that may later be used to compute the effects of a caller.

Persistence and interruption

Since PipsMake itself is implemented with NewGen, the PipsMake data structure describing the state of Pips is saved on disk when Pips exits. Thus, a Pips operation can continue from a previous saved state.

Furthermore, at a phase boundary, all Pips data structures are managed with PipsDBM. Thus the Pips process can be stopped cleanly at a phase boundary when the control is under PipsMake. Of course the `make()` operation asked to PipsMake fails, but next time the user asks for a `make()`, PipsMake will only launch the phases required to build the lacking resources. This mechanism is heavily used in the user interfaces to insure interactivity.

B.3.5 PipsDBM resource manager

All the data structures used by the phases to seamlessly communicate with each other are under the responsibility of the PipsDBM resource manager.

PipsDBM can decide to store a resource in memory or on disk. Typically, resources that have been modified are written on disk when Pips exits but the programmer can ask to immediately create a file resource to be viewed by a user interface. The programmer can also declare some resources as unloadable or as memory-only.

PipsDBM also manages a logical date for each resource for the dependence mechanism of PipsMake. Logical dates have been introduced to overcome the lack of Unix time stamp precision (1 second under SunOS4) and the difficulty to have very well synchronized client-server operations in an NFS environment. Of course, the Unix date is also used in order to detect resource files modified by the user (typically an edit through a user interface) where the 1 second precision is good enough.

The programming interface is reduced to

```
string db_get_resource(string ressource_name, string module_name, bool pure)
void db_put_resource(string ressource_name, string module_name, void* value)
```

`pure` is used to specify if the programmer wants the genuine resource or only a copy (for example to modify it and give back another resource).

B.3.6 PIPS properties

Global variables to modify or finely tune Pips behavior are quite useful but unfortunately are often dangerous. Thus these variables are wrapped in Pips as properties. A property can be a boolean, an integer or a string. Using the fact that properties are centralized, they are initialized from a default property file and possibly from a local `property.rc`. Since the behavior is modified by the properties, they are stored in the workspace in order to restart a Pips session later in the same way even if the global property file has been modified.

B.4 Programming environment

The Pips workbench is implemented to form a programming environment. Typically, each phase has its own directory and its own library so that it is easy to add a new phase or modify a phase without too much trouble-shooting. Furthermore, almost all the directory structure is duplicated to have both a production and a development version at the same time.

B.4.1 NewGen: data structure and method generation

In oder to ease data structure portability between different languages and also to allow data persistence, a tool and a language to declare data structures has been developed: NewGen [JT90]. Once a data description of the *domains* (that are the data structures that can be defined in NewGen) is written, NewGen constructs several methods to create initialized or not data values, access or modify certain parts of constructed values, write or read data from files, or recursively deallocate data values.

A domain can be a sum (`+` like a C-union) or a product (`x` like a C-struct) of other domains, list (`*`), set (`{}`), array (`[]`) or a map (`->`) of domains. Some domains can be declared as "persistent" to avoid being deleted and the "tabulated" attribute associates a unique identifier to each domain element, allowing unique naming through files and different program runs. The mapping of domains is used to attach semantic information to objects: `alignmap = entity->align`.

Of course, some external types can be imported (such as for using the linear C^3 library in Pips). The user must provide a set of functions for these domains to write, read, free and copy methods. The NewGen environment includes also various low-level classes such as lists, hash-tables, sets, stacks etc. with their various methods (iterators, etc.) that ease the writing of new phases.

An excerpt of the internal representation for Fortran in Pips is shown on Figure B.7. For example, a `call` is a `function entity` with a list of `arguments expressions`. An `expression` can be a `reference`, a `range` or a `call`, and so on. In fact an assignment is represented itself by a call to a pseudo-intrinsic function "=" with 2 arguments, the left hand side and the right hand side argument. And so on for all the intrinsics of the Fortran language. All the syntactic sugar of Fortran has been removed from the internal representation and in this way it is expected not to stick on Fortran idiosyncrasies. The representation is quite simple: there are only 5 cases of statements and 3 cases of expressions. This simplicity also benefits all the algorithms present in Pips since there are very few different cases to test.

```
expression = reference + range + call ;
reference = variable:entity x indices:expression* ;
range = lower:expression x upper:expression x increment:expression ;
call = function:entity x arguments:expression* ;
statement = label:entity x number:int x ordering:int x
            comments:string x instruction ;
instruction = block:statement* + test + loop + goto:statement +
              call + unstructured ;
test = condition:expression x true:statement x false:statement ;
```

FIG. B.7 – *Excerpt of the* PIPS *abstract syntax tree.*

An important feature of NEWGEN is the availability of a general multi domain iterator function (`gen_multi_recurse`). From any NEWGEN domain instance (e.g., `statement`, `instruction`, `expression`) one can start an iterator by providing the list of domains to be visited, and for each domain the function to be applied top-down, which returns a boolean telling whether to go on with the recursion or not, and a function to be applied bottom-up. The iterator is optimized so as to visit only the needed nodes, and not to follow paths on which no nodes of the required types might be encountered.

In the Figure B.8 example, some entities are replaced with others. Entities may appear as variable references, as loop indices, as called functions and in program codes. The recursion starts from `statement stat`: on each `loop`, `reference`, `call` or `code` encountered top-down, the function `gen_true` is applied. This function does nothing but returning true, hence the recursion won't be stopped before having processed all the required domains that can be reached from `stat`. While going back bottom-up, the `replace` functions are applied and perform the required substitution. From the user point of view, only replace functions are to be written.

```
static void replace(entity * e)
{ /* replace entity (*e) if required... */}

static void replace_call(call c)
{ replace(&call_function(c));}

static void replace_loop(loop l)
{ replace(&loop_index(l));}

static void replace_reference(reference r)
{ replace(&reference_variable(r)); }

static replace_code(code c) // code_declarations() is a list of entity
{ MAPL(ce, replace(&ENTITY(CAR(ce))), code_declarations(c));}

// args: (obj, [domain, filter, rewrite,]* NULL);
   gen_multi_recurse(stat,
        reference_domain, gen_true, replace_reference,
        loop_domain,      gen_true, replace_loop,
        call_domain,      gen_true, replace_call,
        code_domain,      gen_true, replace_code,
                    NULL);
```

FIG. B.8 – *`gen_multi_recurse` example*

An example of how NEWGEN defined types are used is shown on Figure B.9. It is a follow-up of Figure B.5. `suppress_dead_code_statement()` begins with a NEWGEN generic iterator to walk down the module statements applying `dead_statement_filter` on them and walk up applying `dead_statement_rewrite` on all the `statement` domains encountered. In `dead_statement_filter()` one can see that `statement_instruction()` is the accessor to the `instruction` of a `statement` generated by NewGen from the internal representation description. `instruction_tag()` describes the content of an `instruction` (which is a "union"). For loops the result is `is_instruction_loop`. NewGen also generates predicates such as `instruction_loop_p()` to test directly the content of an `instruction`.

NewGen can output data structure declarations and methods for C and Common-Lisp languages since they are the languages used to build Pips. The translation environment is described in Section B.4.4. Note that at the beginning of the Pips project, C++ was not considered stable enough so that all the object methods have been written in C, wrapped and hidden in macros and functions.

B.4.2 Linear C^3 library

An important tool in Pips is the Linear C^3 library that handles vectors, matrices, linear constraints and other structures based on these such as polyhedrons. The algorithms used are designed for integer and/or rational coefficients. This library is extensively used for analyses such as the dependence test, precondition and region computation, and for transformations, such as tiling, and code generation, such as send and receive code in HPF compilation or code control-distribution in wp65. The linear C^3 library is a joint project with irisa and prism laboratories, partially funded by cnrs. irisa contributed an implementation of Chernikova algorithm and prism a C implementation of pip (Parametric Integer Programming).

B.4.3 Debugging support

Developing and maintaining such a project requires a lot of debugging support from the programming environment at many levels. At the higher level, it is nice to be able to replay a Pips session that failed since a bug can appear after quite a long time. For this purpose there is a tool that transforms a Wpips log file into a Tpips command file to be re-executed later. Every night, a validation suite is run on hundreds of test cases to do some basic non-regression testing. It is not enough to do intensive debugging but is generally sufficient to ensure that a modification is acceptable.

At the programmer level there are some macros to enable or disable the debug mode in some parts of Pips according to some environment variables that indicate the debug level. Often, there is such a variable per phase at least and the debug modes can be nested to avoid debugging all the functions used in a phase to debug only this phase for example. The debug levels can also be modified according to the property of the same names.

At the NewGen level, the recursive type coherence of a data structure can be verified with `gen_consistent_p()` or even at each assignment or at creation time by setting the variable `gen_debug` to an adequate value.

At last, some Emacs macros and key bindings have been defined for the `gdb` Emacs-mode to display the type of a NewGen object, the type of a Pips entity, an expression of the Pips abstract syntax tree, a Pips statement, etc. to alleviate the development effort.

B.4.4 Documentation and configuration files

Such a big project (over 200,000 lines of code for the compilers, mathematical libraries and tools) with developers on different sites) needs to manage a strong documentation support of course, but also as much automatically generated and non-redundant as possible to keep the incoherence level low.

The main idea is to write all the configuration files as technical reports and automatically extract the configuration files from them. Figure B.10 is a simplified synopsis of the documentation flow. C headers are generated from `.newgen` files that come from some document describing the usage of the data structure, such as `ri.tex` for the internal representation . The menu files for the user interfaces are generated from `pipsmake-rc` and the property file. All the information needed to feed PipsMake and PipsDBM comes from the `pipsmake-rc` file. At last, LaTeX2HTML is used to generate most of `http://www.cri.ensmp.fr/pips` from all the LaTeX files.

```
static bool dead_statement_filter(statement s)

   instruction i = statement_instruction(s);
   bool ret = TRUE;

   if (!statement_weakly_feasible_p(s))       // empty precondition
      ret = remove_dead_statement(s, i);
   else

      switch (instruction_tag(i))
      case is_instruction_loop:

         loop l = instruction_loop(i);
         if (dead_loop_p(l))                    // DO I=1,0
            ret = remove_dead_loop(s, i, l);
         else if (loop_executed_once_p(s, l))  // DO I=N,N
            remove_loop_statement(s, i, l);
            suppress_dead_code_statement(body);
            ret = FALSE;

         break;

      case is_instruction_test:
         ret = dead_deal_with_test(s, instruction_test(i));
         break;
      case is_instruction_unstructured:
         dead_recurse_unstructured(instruction_unstructured(i));
         ret = FALSE;
         break;



   if (!ret) // Try to rewrite the code underneath
      dead_statement_rewrite(s);

   return ret;


void suppress_dead_code_statement(statement module_stmt)

    gen_recurse(module_stmt,            // recursion from...
                statement_domain,       // on statements
                dead_statement_filter,  // entry function (top-down)
                dead_statement_rewrite);// exit function (bottom-up)
```

FIG. B.9 – *Excerpt of the function of the dead code elimination phase.*

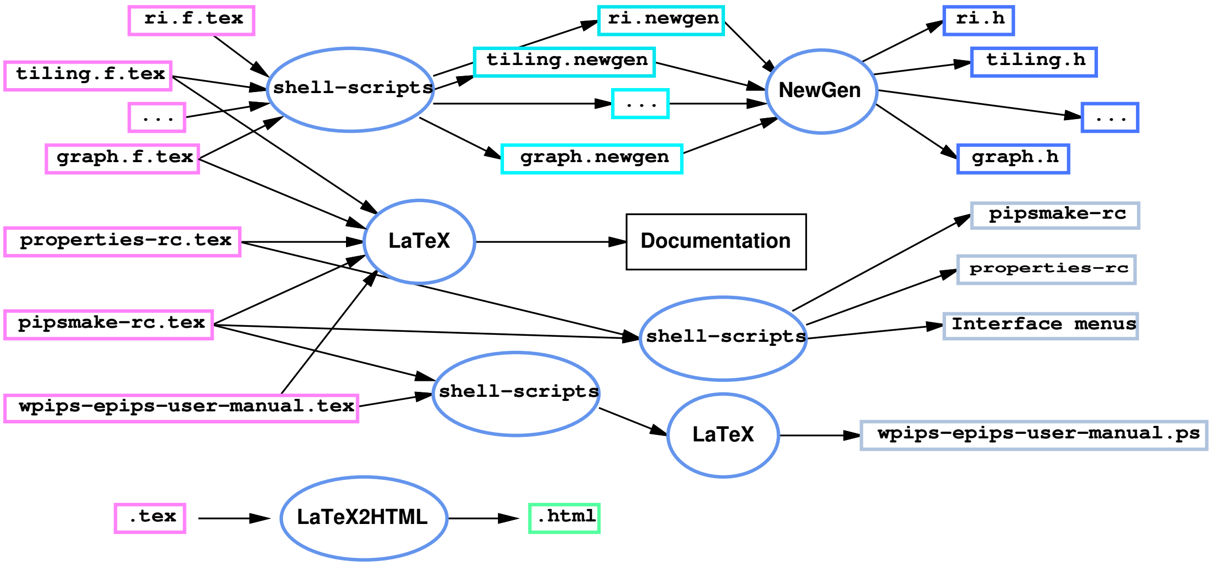


FIG. B.10 – *Documentation and configuration system.*

B.5 User interfaces

Typically, a user interface will use the `make()` command to build a resource to display, use `apply()` to apply a transformation on the code and select default rules, that are options, with `activate()` (Section B.3.4.0). There are as many as four interfaces to PIPS :

- `pips` is a shell interface, can do only one operation on a workspace and relies a lot on PipsDBM persistence since the `pips` process exits after each operation;
- `tpips` is a line interface with automatic completion. It's much faster to use for experimentation and debugging than a full graphical interface;
- `wpips` is an XView based easy interface interface;
- `epips` is an extended version of the previous one for GNU-Emacs users. The interface inherits the Fortran mode and color highlight features. Further more graphs are displayed with `xtree` and `daVinci`.

All these interfaces are configured according to the documentation B.4.4: when a programmer adds a transformation, it automatically appears in all the interfaces after reconfiguration of the `pipsmake-rc` file.

When dealing with big programs, it may be necessary to interrupt PIPS if a user chooses too costly an option or applies an heavy analysis by mistake. To solve this issue, the user interfaces can interrupt PipsMake at the phase interface level, without compromising the coherence through PipsMake and PipsDBM (Section B.3.4.0).

B.6 Conclusion

B.6.1 Related Work

Many compiler prototypes have been implemented to test analysis, transformation or code generation techniques.

Parafrase 2 (UIUC): This compiler is a source to source code restructurer for Fortran and C performing analysis, transformation and code generation. It allows automatic vectorization and parallelization.

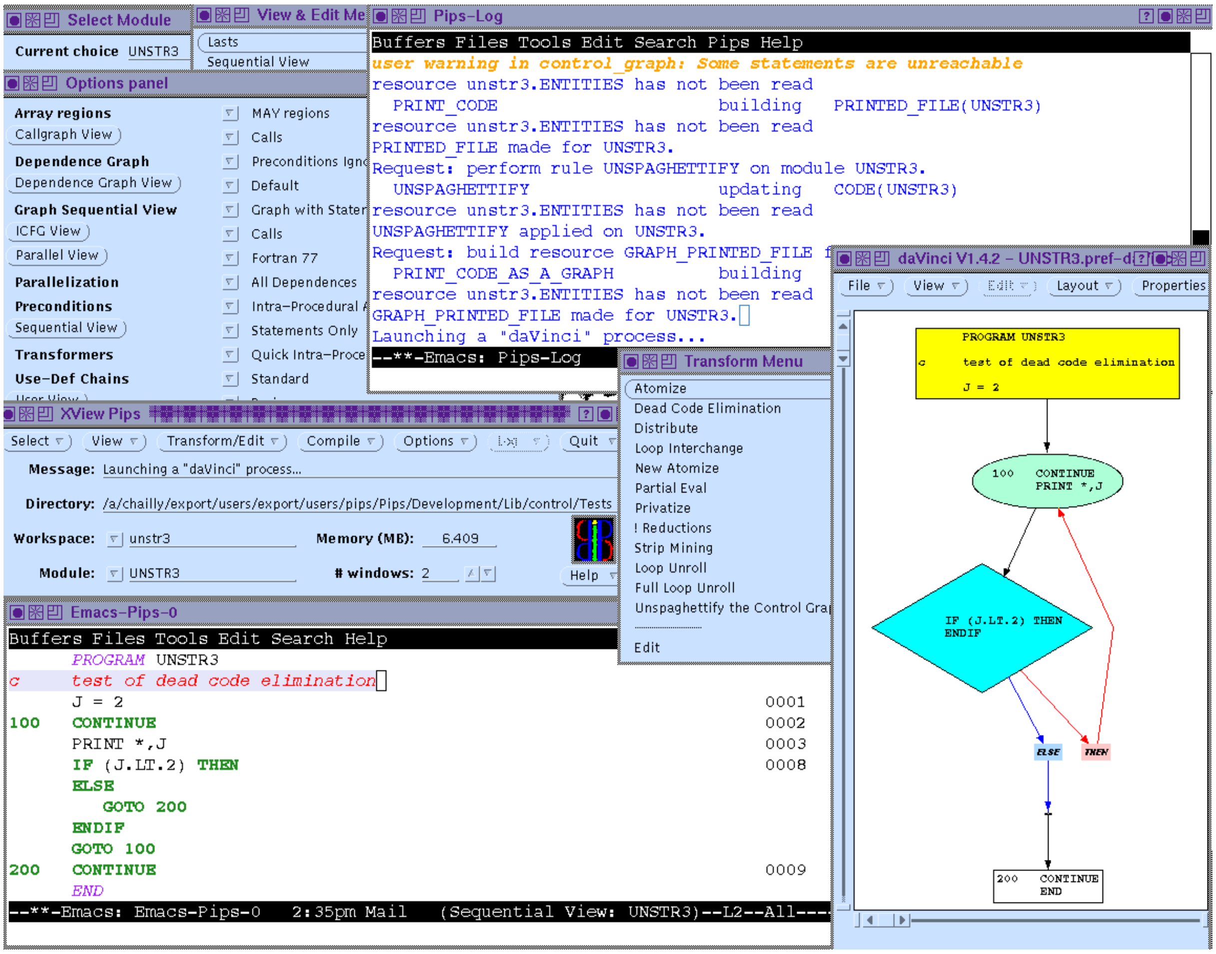


FIG. B.11 – *Emacs-Pips interface snapshot.*

Polaris (UIUC): This compiler is also a source to source restructurer developed for Fortran 77. It performs interprocedural analysis for automatic parallelization of programs, targeting shared-memory architectures.

SUIF (Stanford): The *Stanford University Intermediate Form* is a low level representation plus some high level additions that keep track of the program structure such as loops, conditions and array references. The input language is C and Fortran through the `f2c` translator. The compiler outputs machine code for different architectures. The compiler initial design and the current distribution of SUIF does not incorporate interprocedural analysis. Such an addition is being worked on [HMCCR93, HMA+95]. Another problem is that the consistency of analyses is not automatically inforced (for instance invalidation after code transformations).

The originality of PIPS with respect to these other prototypes is the conjunction of:

- Interprocedurality and coherency automatic management in a demand driven system combining database (PipsDBM) and dependence (PipsMake) functions.
- Advanced interprocedural semantic and data-flow analyses already implemented, and their mathematical foundation upon linear algebra (Linear C^3 library).
- Software foundation upon an high level description system (NEWGEN) which automatically generates functions (access, constructor, destructor) and can iterate on arbitrary structures. An additional by-product is that the code look and feel is homogeneous over the 150,000 lines of the compiler.
- Multi-target:
    - Automatic parallelization for shared-memory
    - Compilation for distributed-memory models (HPFC, WP65)
    - Automatic data mapping
- The availability of different interfaces (shell, terminal TPIPS, graphical WPIPS).

Compiler development is a hard task. Here are examples of many research prototypes. They would benefit a lot from already available infrastructures, and of the synergy of other analyses and transformations, especially interprocedural ones.

- (small) program analyzers and parallelizers: Bouclette (LIP, ENS-Lyon), LooPo (U. Passau), Nascent (OGI), PAF (PRISM, UVSQ), Petit (U. of Maryland), Tiny.
- Parallel Fortran (HPF spirit) compilers: Adaptor (GMD), Annai (CSCS-NEC), D system (Rice U.), Fx (Carnegie Melon U.), Paradigme (UIUC, based on Parafrase 2), SHPF (Southhampton), VFCS (Vienna)
- Parallel C compilers: Pandore (Irisa), EPPP (CRI Montréal)

The HTML version of this paper should be used to obtain many links to related work and the PIPS project.

B.6.2 Remarks

Eight years after its inception, PIPS as a workbench is still alive and well. PIPS provides a robust infrastructure for new experiments in compilation, program analysis, optimization, transformation and parallelization. The PIPS developer environment is described in [IJT+96], but it also is possible to develop new phases on top of but *outside* of PIPS since all (in fact, most...) PIPS data structures can be reloaded using NEWGEN primitives.

PIPS can also be used as a reverse engineering tool. Region analyses provide useful summaries of procedure effects, while precondition-based partial evaluation and dead code elimination reduce code size.

PIPS may be less robust than other publicly available source-to-source compilers but the stress is put on the ability to quickly add new phases such as program transformations, semantic analyses, parsers, parallelizers, etc. or new user interfaces without spending time developing a cumbersome infrastructure.

All the on-line reports of the CRI can be found at http://www.cri.ensmp.fr/rapports.html.

The PIPS software and documentation is available on `http://www.cri.ensmp.fr/pips`.

B.6.3 Acknowledgment

We would like to thank many people that have contributed to the PIPS project: Bruno BARON, Denis BARTHOU, Pierre BERTHOMIER, Arnauld LESERVOT, Guillaume OGET, Alexis PLATONOFF, Rémi TRIOLET, Yi-Qing YANG, Lei ZHOU.

Bibliographie

[ABH97] Ed Anderson, Jeff Brooks, and Tom Hewitt. The benchmarker's guide to single-processor optimization for cray t3e systems. Technical report, Cray Research, June 1997.

[AC94] Béatrice Apvrille-Creusillet. Régions exactes et privatisation de tableaux. Master's thesis, Université Paris VI, September 1994.

[ACIK93] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. December 1993.

[ACIK96] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. A Linear Algebra Framework for Static HPF Code Distribution. *Scientific Programming*, ??(??):??–??, ?? 1996. To appear.

[AGGL97] E. Ayguade, J. Garcia, M. L. Grande, and J. Labarta. Data distribution and loop parallelization for shared-memory multiprocessors. *Lecture Notes in Computer Science*, 1239:41–??, 1997.

[Agr98] G. Agrawal. Interprocedural partial redundancy elimination with application to distributed memory compilation. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):609–624, July 1998.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26, pages 39–50, Williamsburg, VA, April 1991.

[AI92] Corinne Ancourt and François Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers*, July 1992.

[Aik88] Alexander Aiken. Compaction-based parallelization. Technical Report TR88-922, Cornell University, Computer Science Department, June 1988.

[AJ77] A. V. Aho and S. C. Johnson. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1):146–160, January 1977.

[AJLA95] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, September 1995.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AKPW83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189. ACM, ACM, January 1983.

[AKR98] Shail Aditya, Vinod Kathail, and Ramakrishna Rau. Elcor's machine description system : Version 3.0. Technical Report HPL-98-128, Hewlett Packard, October 1998.

[All69] Fran E. Allen. Program optimization. In Mark I. Halpern and Christopher J. Shaw, editors, *Annual Review in Automatic Programming*, volume 5, pages 239–307. Pergamon Press, Oxford, 1969.

[Ama97] Saman Prabhath Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities*, phd thesis Modulo and Division optimization, pages 154–161. Stanford University, 1997.

[And96] F. André. A multi-threads runtime for the pandore data-parallel compiler. Technical Report 986, IRISA, 1996.

[ANS83] ANSI. *American National Standard Programming Language Fortran 77, ANSI x3.9-1978, ISO 1539-1980*. New-York, 1983.

[ANS89] ANSI. Rationale for American National Standard for Information Systems, Programming Language, C. included in ANSI X3.159-1989, 1989.

[APC+98] Arnould A., Baudin P., J.M. Collart, A. Pacalet, J. Raguideau, and D. Schoen. Assistance in software validation. In *DASIA'98: Data Systems in Aerospace*, Athens, May 1998.

[AR94] J. R. Alsabbagh and V. V. Raghavan. Analysis of common subexpression exploitation models in multiple-query processing. In Ahmed K. Elmagarmid and Erich Neuhold, editors, *Proceedings of the 10th International Conference on Data Engineering*, pages 488–497, Houston, TX, February 1994. IEEE Computer Society Press.

[ARM98] ARM. *ARM10 Thumb Family*, 1998.

[AS92] T.M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *19th International Symposium on Computer Architecture (ISCA-19)*, 1992.

[ASD95] Gagan Agrawal, Joel H. Saltz, and Raja Das. Interprocedural partial redundancy elimination and its application to distributed memory compilation. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, La Jolla, California, June 1995.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[AWZ88] B. Alpern, M. Wegman, and F. Zadeck. Detecting equality of variables in programs. In *Fifteenth POPL*. January 1988.

[BA98] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–251, San Diego, California, January 1998.

[Ban79] U. Banerjee. *Speedup of ordinary programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1979. Report No. 79-989.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publisher, New York, USA, 1988.

[Bay98] John Bayko. Great Microprocessors of the Past and Present (V 10.11). `http://www.cs.uregina.ca/~bayco/ design/design.html`, March 1998.

[BB89] William Baxter and Henry R. Bauer, III. The program dependence graph and vectorization. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 1–11, Austin, TX, USA, January 1989.

[BBB+98] Michel Barreteau, Francois Bodin, Peter Brinkhaus, Zbigniew Chamski, Henri-Pierre Charles, Christine Eisenbeis, John Gurd, Jan Hoogerbrugge, Ping Hu, William Jalby, Peter Knijnenburg, Michael O'Boyle, Erven Rohou, Rizos Sakellariou, André Seznec, Elena Stöhr, Menno Treffers, and Harry Wijshoff. Oceans: Optimising compilers for embedded applications. In *Proc. EuroPar'98, Parallel Processing, Workshop 23*, pages 1123–1130, 1998.

[BBC+97] Jean-Claude Bajard, Olivier Beaumont, Jean-Marie Chesneaux, Marc Daumas, Jocelyne Erhel, Dominique Michelucci, Jean-Michel Muller, Bernard Philippe, Nathalie Revol, Jean-Louis Roch, and Jean Vignes. *Qualité des Calculs sur Ordinateur: Vers des arithmétiques plus fiables? [*Quality of Computer Calculations: Towards More Viable Arithmetic*]*. Masson Editeur, Paris, France, 1997.

[BC94] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 159–170, New York, NY, USA, June 1994. ACM Press.

[BCG+94] P. Banerjee, J.A. Chandy, M. Gupta, J.G. Holm, A. Lain, D.J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *First International Workshop on Parallel Processing*, Bangalore - India, December 1994.

[BCGS96] David Berson, Pohua Chang, Rajiv Gupta, and Mary Lou Soffa. Integrating program optimizations and transformations with the scheduling of instruction level parallelism. In *Languages and Compilers for Parallel Computing*, pages 207–221, 1996.

[BCOQ92] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity*, chapter Max-Plus Algebra, pages 103–154. Wiley, 1992.

[BCR93] Leo Bachmair, Ta Chen, and I. V. Ramakrishnan. Associative-commutative discrimination nets. In *International Joint Conference on Theory And Practice of Software Development*, pages 61–74, April 1993.

[BCR+95] Leo Bachmair, Ta Chen, I.V. Ramakrishnan, Siva Anantharaman, and Jacques Chabin. Experiments with associative-commutative discrimination nets. In *International Joint Conference on Artificial Intelligence*, pages 348–354, August 1995.

[BCS97] R. Briggs, K. D. Cooper, and L. T. Simpson. Value numbering. *Software Practice and Experience*, 27(6):701–724, June 1997.

[BCT87] Preston Briggs, Keith D. Cooper, and Linda Torczon. Iloc'87. *IRn Programming Environnement Newsletter*, (44), September 1987.

[BD96] T. Brandes and F. Desprez. Implementing Pipelined Computation and Communication in an HPF Compiler. In *Europar'96*, August 1996.

[BDFH96] J.A. Bergstra, T.B. Dinesh, J. Field, and J. Heering. A complete transformation toolkit for compilers. Technical Report RC20342, IBM, January 1996.

[BEF+96] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jeajin Lee, Tom Lawrence, Jay Hoeflinger, David Padua, Yunheung Paek, Paul Peterson, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Restructuring programs for high-speed computers with polaris. In *International Conference on Parallel Processing Workshop*, pages 149–161, 1996.

[Bel94] Francoise Bellegarde. A transformation system combining partial evaluation with term rewriting. Technical Report CSE-94-010, Oregon Graduate Institute of Science and Technology, January 1994.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15:757–762, October 1966.

[Ber98] Laurent Bergogne. *Quelques algorithmes dans les modèles de calcul PRAM, BSR, BSP*. PhD thesis, Université de Picardie Jules Verne, 1998.

[BF98] Pierre Boulet and Paul Feautrier. Scanning polyhedra without do-loops. In *PACT'98*, pages 4–11, October 1998.

[BGM90] H. B. Bakoglu, G. F. Grohoski, and R. K. Montoye. The IBM RISC System 6000 processor: Hardware overview. *IBM Journal of Research and Development*, 34(1):12–22, January 1990.

[BGS98] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-98)*, volume 33, pages 1–14, New York, June 1998. ACM Press.

[BH94] Preston Briggs and Tim Harvey. Multiplication by integer constants. Technical report, Rice University, Houston, Texas, July 1994.

[BKN87] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. *Journal of Symbolic Computation*, 3:203–216, 1987.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web : a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (SIGPLAN '90)*, pages 257–271, White Plains, NY, USA, June 1990. ACM Press.

[Boa97] OpenMP Architecture Review Board. OpenMP : A proposed industry standard API for shared memory programming. Technical report, October 1997.

[BR96] Patrick Baudin and Jacques Raguideau. CAVEAT Algebraic Simplification Rules. Technical Report 0.2.1-7, Commissariat à l'Énergie Atomique - Département d'Électronique et d'Instrumentation Nucléaire, December 1996.

[Bra94] T. Brandes. Adaptor: A compilation system for data parallel Fortran programs. In Christoph W. Kessler, editor, *Automatic Parallelization — New Approaches to Code Generation, Data Distribution, and Performance Prediction*. Vieweg, Wiesbaden, 1994.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[BS76] J. Bruno and R. Sethi. Code generation for a one-register machine. *Journal of the ACM*, 23(3):502–510, July 1976.

[BSCV95] P. Brezany, K. Sanjari, O. Cheron, and E. Van Konijnenburg. Processing irregular codes containing arrays with multi-dimensional distributions by the PREPARE HPF compiler. In Bob Hertzberger and Giuseppe Serazzi, editors, *High-Performance computing and networking: International Conference and Exhibition, Milan, Italy, May 3–5, 1995: proceedings*, volume 919 of *Lecture Notes in Computer Science*, pages 526–531. Springer-Verlag, 1995.

[BZ94] T. Brandes and F. Zimmermann. Adaptor — A transformation tool for HPF programs. In K. M. Decker and R. M. Rehmann, editors, *Programming environments for massively parallel distributed systems: working conference of the IFIP WG10.3, April 25–29, 1994, Ascona, Italy*, pages 91–96, Cambridge, MA, USA, 1994. Birkhäuser Boston Inc.

[CA95] Ta Chen and Siva Anantharaman. STORM : A Many-to-one Associative-Commutative Matcher. *Lecture Notes in Computer Science*, 914:414–419, 1995.

[CA96] Fabien Coelho and Corinne Ancourt. Optimal compilation of HPF remappings. *Journal of Parallel and Distributed Computing*, 38(2):229–236, November 1996.

[Can97] Philippe Canalda. *Contribution à la production automatique de sélecteurs d'instructions optimisants fondés sur la réécriture d'arbres*. PhD thesis, Université d'Orléans, July 1997.

[CCK+97] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32 of *ACM SIGPLAN Notices*, pages 273–286, New York, June 1997. ACM Press.

[CCM95] C. Calvin, L. Colombet, and P. Michallon. Overlap of the Communications in Parallel Algorithms. In *Acte des journées 1995 du site expérimental en hyperparallélisme*, volume 1, page 161. 1995.

[CFL94] Don Coppersmith, Ephraim Feig, and Elliot Linzer. Hadamard transforms on multiply-add architectures. *IEEE Transactions on Signal Processing*, 42(4):969–970, April 1994.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth Zadeck. An efficient method of computing static single assignment form. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89)*, pages 25–35, Austin, TX, USA, January 1989. ACM Press.

[CGV96] Jean-Marie Chesneaux, Stéphane Guilain, and Gilles Vignes. La bibliothèque cadna - présentation et utilisation. Technical report, Université Pierre et Marie Curie, November 1996.

[CH96] R. Cole and R. Hariharan. Tree pattern matching and subset matching in randomized $o(nlog^3m)$ time. Technical Report TR1996-731, New York University, December 1996.

[Cha93a] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle*. PhD thesis, Université de Rennes 1, February 1993.

[Cha93b] Zbigniew Chamski. Fast and efficient generation of loop bounds. Research Report 2095, INRIA, October 1993.

[Cha93c] Zbigniew Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. Research Report 2094, INRIA, October 1993. In Proceedings of the 27th Annual Hawaii Int. Conf. on System Sciences, 1994, p. 14–22.

[Chi97] Stephen Chin. P2SC overview: Architecture, Application Tuning and Optimization. Technical report, "IBM", July 1997.

[CHPC95] Po-Yung Chang, Eric Hao, Yale N. Patt, and Pohua P. Chang. Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution. In Lubomir Bic, Wim Böhm, Paraskevas Evripidou, and Jean-Luc Gaudiot, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 99–108, Limassol, Cyprus, June 1995. ACM Press.

[CI95] Béatrice Creusillet and François Irigoin. Interprocedural Array Region Analyses. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 46–60. Springer-Verlag, August 1995.

[CI96] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming (special issue on LCPC)*, 24(6):513–546, 1996.

[CL96] M. Cosnard and M. Loi. A simple algorithm for the generation of efficient loop structures. *International Journal of Parallel Programming*, 24(3):265–289, 1996.

[Cli95] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 246–257, La Jolla, California, June 1995.

[CLS98] B. Chamberlain, E. C. Lewis, and L. Snyder. Array language support for wavefront and pipelined computations. Technical Report TR-98-11-02, University of Washington, Department of Computer Science and Engineering, November 1998.

[CLZ86] R. Cytron, A. Lowry, and K. Zadeck. Code motion of control structures in high-level languages. In *Symposium on Principles of Programming Languages*, pages 70–84. January 1986.

[CM80] J. Cocke and P. Markstein. Measurement of code improvement algorithms. *Information Processing 80*, pages 221–228, 1980.

[Coc70] A. J. Cocke. Global common subexpression elimination. *ACM SIGPLAN Notices*, 5(7):20–24, 1970.

[Coe94] Fabien Coelho. Experiments with HPF Compilation for a Network of Workstations. In *High Performance Computing and Networking Conference*, pages 423–428, April 1994.

[Coe95] Fabien Coelho. Compilation of I/O Communications for HPF. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 102–109, February 1995.

[Coe96] Fabien Coelho. *Contributions to High Performance Fortran Compilation*. PhD thesis, École des Mines de Paris, October 1996.

[Col93] Jean-François Colonna. The Subjectivity of Computers. *Communications of the ACM*, 36(8), August 1993.

[COM99] Computer Corporation COMPAQ. *Compiler Writer's Guide for the Alpha 21264*, 1999.

[CP94] Philippe Clermont and Nicolas Paris. HyperParallel technologies : HyperC: Portable parallel programming in C. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 682–687, Los Alamitos, CA, USA, April 1994. IEEE Computer Society Press.

[CP95] Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *ACM SIGPLAN Notices*, 30(3):35–49, March 1995.

[Cre95] Béatrice Creusillet. Array regions for interprocedural parallelization and array privatization. Report CRI A 279, CRI, École des Mines de Paris, November 1995.

[Cre96] Béatrice Creusillet. *Array Region Analyses and Applications*. PhD thesis, École des Mines de Paris, December 1996.

[CS70] John Cocke and J. T. Schwartz. Programming languages and their compilers. Technical report, New York, 1970.

[CT93] Weimin Chen and Volker Turau. Source-to-source code generation based on pattern matching and dynamic programming. Technical Report TR-93-047, International Computer Science Institute, Berkeley, CA, August 1993.

[DC98] J. Vayssiere D. Caromel, W. Klauser. Towards seamless computing and metacomputing in java. In *Concurrency Practice and Experience*, pages 1043–1061, September 1998.

[DDT95] F. Desprez, J.J. Dongarra, and B. Tourancheau. Performance Study of LU Factorization with Low Communication Overhead on Multiprocessors. *Parallel Processing Letters*, 5(2):157–169, 1995.

[DE73] G.B. Dantzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory*, A(14), 1973.

[Deh91] Babak Dehbonei. *Génération de code et analyse interprocédurale au sein d'un environnement de programmation parallèle.* PhD thesis, Université de Paris VI, October 1991.

[Der93] Nachum Dershowitz. A taste of rewriting. *Functional Programming, Concurrency, Simulation and Automated Reasoning, Lecture Notes in Computer Science*, 693:199–228, 1993.

[Des94] F. Desprez. A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures. In *IFIP 10.3 Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 365–371. Birkhaeuser Verlag AG, Basel, Switzerland, 1994.

[DFSLOJ94] Bacon David F., Graham Susan L., and Sharp Oliver J. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[DG95] F. Desprez and M. Garbey. Numerical Simulation of a Combustion Problem on a Paragon Machine. *Parallel Computing*, 21:495–508, 1995.

[Dha88] D. M. Dhamdhere. Register assignment using code placement techniques. *Computer Languages*, 13(2):75–93, 1988.

[Dha91] Dhananjay M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[Dow80] Downey, P. J. and Sethi, R. and Tarjan, R. E. . Variations on the Common Subexpression Problem. *Journal of the ACM*, 27(4):758–771, 1980.

[DP94] F. Desprez and M. Pourzandi. A Comparison of Three Matrix Product Algorithms on the Intel Paragon and Archipel Volvox Machines. In Bl. Sendov I.T.Dimov and P.S.Vassilevski, editors, *Advances in Numerical Methods and Applications NM&A - O(h^3)'94*, pages 234–244. World Scientific, 1994.

[DR96] S. Dutta and J. Ramanujam. Data structures for efficient execution of programs with block-cyclic distributions. In *Compiler for Parallel Computers*, pages 91–92, December 1996.

[DRR96] F. Desprez, P. Ramet, and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms. In *Europar'96*, August 1996.

[DS88] K. Drechsler and M. Stadel. A solution to a problem with Morel and Renvoisés "global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[DT94] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. *Future Generation Computer Systems*, 10(2&3):279–284, June 1994.

[Duf74] R.J. Duffin. On fourier's analysis of linear inequality systems. In *Mathematical Programming Study*, number 1, pages 71–95. North-Holland Publishing Company, 1974.

[DV95] Alain Darte and Frédéric Vivien. A comparison of nested loops parallelization algorithms. Technical Report 95-11, LIP, ENS-Lyon, May 1995.

[DV96] A. Darte and F. Vivien. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. *Lecture Notes in Computer Science*, 1123:379–??, 1996.

[DZ97] Frédéric Desprez and Julien Zory. Macro-pipelines dans les programmes data-parallèles. In *9ème Rencontres francophones du Parallélisme*, pages 17–20, May 1997.

[ECGS92] T. Von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Message: a Mecanism for Integrated Communication and Computation. In ACM IEEE Computer Society, editor, *The 19th Annual Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.

[EGK+94] Kemal Ebcioglu, Randy Groves, Ki-Chang Kim, Gabriel Silberman, and Isaac Ziv. VLIW compilation techniques in a superscalar environment. *SIGPLAN Notices*, 29(6):36–48, June 1994. Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.

[EJLC97] E. W. Evans, S. P. Johnson, P. F. Leggett, and M. Cross. Automatic code generation of overlapped communications in a parallelization tool. *Parallel Computing*, 23(10):1493–1523, October 1997.

[Erh90] Jocelyne Erhel. Erreurs de calcul des ordinateurs. Technical Report 552, INRIA, September 1990.

[Eri96] Grant Erickson. RISC for Graphics: A Survey and Analysis of Multimedia Extended Instruction Set Architectures. Electrical Engineering 8362, Dept. of Elec. Eng., University of Minnesota, December 1996.

[Ert99] M. Anton Ertl. Optimal code selection in dags. In *Principles of Programming Languages (POPL '99)*, 1999.

[Fag86] Francois Fages. Associative-commutative unification. *"Journal of Symbolic Computation"*, pages 194–208, 1986.

[FDT87] P. Feautrier, A. Dumay, and N. Tawbi. Paf : un paralléliseur pour fortran. Technical Report 185, Université Paris VI, 1987.

[Fea88] P. Feautrier. Parametric Integer Programming. In *RAIRO Recherche Opérationnelle*, volume 22, pages 243–268, September 1988.

[FHM95] T. Fahringer, M. Haines, and P. Mehrotra. On the Utility of Threads for Data Parallel Programming. In *International Conference on Supercomputing*, July 1995.

[Fin82] S. Finkelstein. Common subexpression analysis in database applications. In Mario Schkolnick, editor, *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*, pages 235–245, Orlando, Florida, June 1982.

[Fis84] J. A. Fisher. The VLIW machine: a multiprocessor for compiling scientific code. In *Computer*, volume 17, pages 45–53. 1984.

[FN92] E.W. Felten and D. Mc Namee. Improving the Performance of Message Passing Applications by Multithreading. In IEEE Computer Society, editor, *Scalable High-Performance Computing Conference*, pages 84–89, April 1992.

[FO83] Jeanne Ferrante and Karl Ottenstein. A program form based on data dependency in predicate regions. In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Lanuages*, pages 217–236, Austin, TX, January 1983. ACM Press.

[For93] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, May 1993. *Version 1.0.*

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Fra70] Dennis J. Frailey. Expression optimization using unary complement operators. In *Proceedings of a Symposium on Compiler Optimization*, volume 5, pages 67–85, July 1970.

[FSS90] Philippe Flajolet, Paolo Sipala, and Jean-Marc Steyaert. Analytic variations on the common subexpression problem. In Michael S. Paterson, editor, *Automata, Languages and Programming, 17th International Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 220–234, Warwick University, England, July 1990. Springer-Verlag.

[GB94] M. Gupta and P. Banerjee. Compile-Time Estimation of Communication Costs of Programs. *Journal of Programming Languages*, 1994.

[GBC+93] V. Getov, T. Brandes, B. Chapman, A. Dunlop, Anthony J. G. Hey, and David Pritchard. Comparison of HPF-like systems. PPPE Deliverable D4.3a (University of Southampton, GMD, University of Vienna), November 1993.

[GBD+93] Al Geist, Adam Beguelin, Jack Dongarra, Jiang Weicheng, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.

[GDF98] R. Gupta, Berson D., and J.Z. Fang. Path profile guided partial redundancy elimination using speculation. In *IEEE International Conference on Computer Languages*, pages 230–239, May 1998.

[Gea65] C. W. Gear. High speed compilation of efficient object code. *Communications of the ACM*, 8(8):483–488, August 1965.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability (A Guide to Theory of NP-Completeness).* Freeman, New York, 1979.

[GL97] W. Gropp and E. Lusk. Why are PVM and MPI so different? *Lecture Notes in Computer Science*, 1332:3–10, 1997.

[GLR97] César Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–74, March 1997.

[GM94] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 61–72, Orlando, Florida, June 1994.

[GmWHR98] John C. Gyllenhaal, Wen mei W. Hwu, and B. Ramakrishna Rau. Optimization of machine descriptions for efficient use. *International Journal of Parallel Programming*, 26(4):417–447, 1998.

[GUD96] Marc Gengler, Stéphane Ubéda, and Frédéric Desprez. *Initiation au Parallélisme. Concepts, architectures et algorithmes.* Paris, 1996.

[Ha95] Hwu and al. Compiler technology for future microprocessors. Technical report, University of Illinois, 1995.

[Har96] Richard Hartley. Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers. *IEEE Circuits and Systems - Part II: Analog and Digital Signal Processing Transactions*, 43(10), October 1996.

[HAS95] Laurent HASCOET. Specifications of PARTITA Analyses. Technical report, INRIA, 1995.

[Hav93] Paul Havlak. Construction of thinned gated single-assignment form. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 477–499, Portland, Oregon, August 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.

[HFH94] T. N. Hicks, R. E. Fry, and P. E. Harvey. POWER2 floating-point unit: Architecture and implementation. *IBM Journal of Research and Development*, 38(5):525–536, September 1994.

[HH97] Nigel Horspool and H. C. Ho. Partial redundancy elimination based on a cost-benefit analysis. In *Proceedings of 8th Israeli Conference on Computer Systems and Software Engineering (ICSSÉ97) - IEEE Computer Society*, pages 111–118, Herzliya, Israel, June 1997.

[Hit98] Hitachi. *SH7750 (SH-4 Series) Super H RISC Processor - Product Brief*, September 1998.

[HKK+92] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler Analysis for Irregular Problems in Fortran-D. In *Proceedings of 5th Workshop on Languages and Compilers for Parallel Computing*, pages 97–111, August 1992.

[HKT94] S. Hiranandani, K. Kennedy, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, 21:27–45, 1994.

[HMA+95] Mary Hall, Brian Murphy, Saman Amarasinghe, Shih-Wei Liao, and Monica Lam. Interprocedural analysis for parallelization. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 61–80. Springer-Verlag, August 1995.

[HMCCR93] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodríguez. FIAT : A framework for interprocedural analysis and transformation. In *Sixth International Workshop on Languages and Compilers for Parallel Computing*, August 1993.

[HMR93a] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. Constructing program dependence graphs using a parser. Technical Report 93-117, Department of Computer Science, Clemson University, November 1993.

[HMR93b] Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. Efficient construction of program dependence graphs. Technical Report 93-102, Department of Computer Science, Clemson University, January 1993.

[HPF96] HPF Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, November 1996. version 2.0.

[HPR88] S. Horowitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, pages 146–157, San Diego, CA, January 1988.

[Huf52] David Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, pages 1098–1101, September 1952.

[Hun95] D. Hunt. Advanced performance features of the 64-bit PA-8000. In IEEE, editor, *Digest of papers: Compcon '95: technologies for the information superhighway: March 5–9, 1995, San Francisco, CA, USA*, pages 123–128, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, March 1995. IEEE Computer Society Press.

[IBM95a] IBM. Ibm Parallel Environnement for AIX. MPL Programming and Subroutine Reference (version 2, release 1). Technical Report GC23-3893-00, International Business Machines Corporation, August 1995.

[IBM95b] Corporation IBM. Optimization and tuning guide for FORTRAN, C and C++. Technical Report SC09-1705-00, 1995.

[IBM96a] IBM. *XL High Performance Fortran for AIX - Language Reference.* IBM, March 1996.

[IBM96b] IBM. *XL High Performance Fortran for AIX - User's Guide.* IBM, March 1996.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, pages 144–151, June 1991.

[IJT+96] François Irigoin, Pierre Jouvelot, Rémi Triolet, Arnauld Leservot, Alexis Platonoff, Ronan Keryell, Corinne Ancourt, Fabien Coelho, and Béatrice Creusillet. *PIPS: Development Environnement.* CRI, École des Mines de Paris, 1996.

[Inc96] MIPS Technologies Incorporated. R10000 microprocessor - product overview (version 2.0). Technical report, September 1996.

[Iri92] François Irigoin. Interprocedural analyses for programming environment. In Jack J. Dongara and Bernard Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 333–350, Saint-Hilaire-du-Touvet, September 1992. North-Holland, Amsterdam, NSF-CNRS.

[ISO91] ISO/IEC. *Fortran 90 International Standard ISO/IEC 1539:1991 (E)*, second 1991-07-01 edition, 1991.

[IT87] François Irigoin and Rémi Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, CRI, École des Mines de Paris, 1987.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 15th Annual ACM Symposium on Principels of Programming Languages (POPL '88)*, page 319, San Diego, CA, USA, January 1988. ACM Press.

[Jar85] M. Jarke. Common subexpression isolation in multiple query optimization. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Sys.*, page 191. Springer-Verlag, 1985.

[JL87] Jean-Pierre Jouannaud and Pierre Lescanne. Rewriting systems. *Technology and Science of Informatics*, 6(3):181–199, 1987. French version: "La réécriture", *Technique et Science de l'Informatique* (1986), vol. 5, no. 6, pp. 433–452.

[Jou87] Pierre Jouvelot. *Parallélisation Sémantique : Une Approche Dénotationnelle Non-Standard pour la Parallélisation de Programmes Impératifs Séquentiels.* PhD thesis, Université Paris VI, 1987.

[JP98] Romesh M. Jessani and Michael Putrino. Comparison of single- and dual-pass multiply-add fused floating-point units. *IEEE Transactions on Computers*, 47(9):927–937, September 1998.

[JT90] Pierre Jouvelot and Rémi Triolet. *NewGen User Manual.* CRI, École des Mines de Paris, December 1990.

[KAC+96] Ronan Keryell, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, and Pierre Jouvelot. Pips: A framework for building interprocedural compilers, parallelizers and optimizers. Technical Report 289, CRI, École des Mines de Paris, April 1996.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[KCN88] C.T. King, W.H. Chu, and L.M. Ni. Pipelined Data Parallel Algorithms - Concept and Modeling. In *International Conference on Supercomputing*, pages 385–395, July 1988.

[Kes98] Christoph W. Kessler. Scheduling expression DAGs for minimal register need. *Computer Languages*, 24(1):33–53, April 1998.

[KK94] V. Konda and A. Kumar. A framework for dependence based optimization and parallelization of practical DO loops. In Jagdish Chandra, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, pages 97–101, Boca Raton, FL, USA, August 1994. CRC Press.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *The High Performance Fortran Handbook.* Scientific and Engineering Computation Series, 1994.

[KM74] D. Kuck and Y. Muraoka. Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity. *Acta Informatica 3*, 3:203–216, October 1974.

[KM75] David J. Kuck and K. Maruyama. Time Bounds on the Parallel Evaluation of Arithmetic Expressions. *SIAM Journal of Computing*, 4(2):147–162, 1975.

[KM90] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing*, pages 407–416, November 1990.

[KMC72] D. J. Kuck, Y. Muraoka, and S.-C. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speedup. *IEEE Trans. Computers*, c-21:1293–1310, 1972.

[KN90] Deepak Kapur and Paliath Narendran. Complexity of associative-commutative unification check and related problems. Technical report, State Uiversity of New York, 1990.

[Knu68] D. E. Knuth. *The Art of Computer Programming Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968.

[Kor98] Fabrice Kordon. Méta scribe : Un outil pour la génération de moteurs de réécriture. Technical Report 037, Laboratoire d'Informatique de Paris 6, July 1998.

[KR95] C. W. Kessler and T. Rauber. Optimal contiguous expression DAG evaluations. *Lecture Notes in Computer Science*, 965:293–??, 1995.

[Kra98] Walter Kramer. A priori worst case error bounds for floating-point computations. *IEEE Transactions on Computers*, 47, 1998.

[KRS92] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 224–234, New York, NY, July 1992. ACM Press.

[KRS94] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, July 1994.

[KRS95] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. The power of assignment motion. *ACM SIGPLAN Notices*, 30(6):233–245, June 1995.

[KRS98] Jens Knoop, Oliver Ruthing, and Bernhard Steffen. Code motion and code placement: Just synonyms? In *ESOP*, 1998.

[KS98] D. Kulkarni and M. Stumm. Optimizing computational and spatial overheads in complex transformed loops. *Lecture Notes in Computer Science*, 1511:367–377, 1998.

[Kuc96] Kuck and Associates. *KAP for IBM Fortran - User Guide*, 3.3 edition, 1996.

[KZ95] Deepak Kapur and Hantao Zhang. An overview of rewrite rule laboratory (rrl). *J. of Computer and Mathematics with Applications*, 29(2):91–112, 1995.

[LAV98] Josep Llosa, Eduard Ayguadé, and Mateo Valero. Qualitative evaluation of register pressure on software pipelined loops. *International Journal of Parallel Programming*, 26(2):121–142, April 1998.

[LE95] Jack L. Lo and Susan J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 151–162, La Jolla, California, June 1995.

[LF93] Elliot Linzer and Ephraim Feig. Implementation of efficient FFT algorithms on Fused Mulitply-add Architectures. *IEEE Transactions on Signal Processing*, 41(1):93–107, January 1993.

[LM97] Sylvain Lelait and Bruno Marmol. Allocation cyclique de registres et déroulage de boucles. *Techniques et Sciences Informatiques*, 16(5):583–608, 1997.

[LP94] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.

[LP97] Pascale Launay and Jean-Louis Pazat. A framework for parallel programming in java. Technical Report RR-3319, INRIA, 1997.

[LW97] Heng Liao and Andrew Wolfe. Available parallelism in video applications. In *International Symposium on Microarchitecture*, pages 321–329, December 1997.

[MCA98] John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code. *International Journal of Parallel Programming*, 26(5):613–638, 1998.

[MCJ+97] Miguel Miranda, Francky Catthoor, Martin Janssen, Stefan Janssens, Martin Kaspar, Lode Nachtergaele, and Sven Wuytack. Survey on high-level address optimization and synthesis techniques for data-intensive applications. Technical Report IMEC/VSDM/HASS/MM970901, September 1997.

[MCT96] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[MDS92] Monica S. Lam Michael D. Smith, Mark A. Horowitz. Efficient superscalar performance through boosting. In *5th ASPLOS*, October 1992.

[MJ96] Potkonjak Miodrag and Rabaey Jan. Maximally fast and arbitrarily fast hardware efficient implementation of linear and feedback linear computations. Technical Report 960008, University of California, Los Angeles, Computer Science Department, March 1996.

[MMA95] Potkonjak Miodrag, Srivastava Mani, and Chandrakasan Anantha. Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination. Technical Report 950058, University of California, Los Angeles, Computer Science Department, December 1995.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, September 1979.

[MSL92] Robert P. Wilson Monica S. Lam. Limits of control flow on parallelism. In *19th ISCA*, pages 19–21, May 1992.

[MUHT96] Mitsunori Matsushita, Motohide Umano, Itsuo Hatono, and Hiroyuki Tamura. A fast pattern-matching algorithm using matching candidates for production systems. In Norman Foo and Randy Goebel, editors, *Proceedings of the Fourth Rim International Conference on Artificial Intelligence (PRICAI-96)*, volume 1114, pages 646–656, August 1996.

[NN95] Steven Novack and Alexandru Nicolau. A hierarchical approach to instruction-level parallelization. *International Journal of Parallel Programming*, 23(1):35–62, February 1995.

[NP93] Nivat and Podelski. Another variation on the common subexpression problem. *DMATH: Discrete Mathematics*, 114, 1993.

[OE92] Karl J. Ottenstein and Steven J. Ellcey. Experience compiling Fortran to program dependence graphs. *Software Practice and Experience*, 22(1):41–62, January 1992.

[OSKO95] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal Tile Size Adjustement in Compiling General DOACROSS Loop Nests. In ACM Press, editor, *International Conference on Supercomputing*, pages 270–279, Barcelona, Spain, July 1995. ACM SIGARCH.

[OSMG98] Geert Jan Olsder, Subiono, and Michael Mc Gettrick. On large scale max-plus algebra model in railway systems. Technical report, CAS - École des Mines de Paris", 1998.

[Par95] Keshab K. Parhi. High-Level Algorithm and Architecture Transformations for DSP Synthesis. *Journal of VLSI Signal Processing*, 9:121–143, 1995.

[PBJ+91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 67–78, Orlando, FL, USA, January 1991. ACM Press.

[PH96] David A. Patterson and John L. Hennessy. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, second edition, 1996.

[Pla95a] Alexis Platonoff. Automatic data distribution for massively parallel computers. pages 555–570, June 1995.

[Pla95b] Alexis Platonoff. *Contribution à la Distribution Automatique de Données pour Machines Massivement Parallèles*. PhD thesis, Université Paris VI, March 1995.

[Plu98] Detlef Plump. Term graph rewriting. Technical Report CSI-R9822, Computing Science Institute, University of Nijmegen, September 1998.

[PS80] Bhaskaram Prabhala and Ravi Sethi. Efficient computation of expressions with common subexpressions. *Journal of the ACM*, 27(1):146–163, January 1980.

[PSCB94] D. J. Palermo, E. Su, J. Chandy, and P. Banerjee. Communication optimizations used in the PARADIGM compiler for distributed memory multicomputers. In *Proceedings of the 23rd International Conference on Parallel Processing. Volume 2: Software*, pages 1–10, Boca Raton, FL, USA, August 1994. CRC Press.

[PSDD97] R. Pasko, P. Schaumont, V. Derudder, and D. Durackova. Optimization method for broadband modem FIR filter design using common subexpression elimination. In *Proceedings 10th International Symposium on System Synthesis - IEEE Computer Society*, pages 100–106, 1997.

[PSDD99] R. Pasko, P. Schaumont, V. Derudder, and D. Durackova. A new algorithm for elimination of common subexpressions. *IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18, 1999.

[PSS98] Vineeth Kumar Paleri, Y. N. Srikant, and Priti Shankar. A simple algorithm for partial redundancy elimination. *ACM SIGPLAN Notices*, 33(12):35–43, December 1998.

[PT96] Bernd Pfrommer and Taku Tokuyasu. Slow fourier transforms on fast microprocessors, February 1996. CS252 project - Berkeley University - http://http.cs.berkeley.edu/~tokuyasu.

[Pug90] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings Supercomputing'91*, pages 4–13, November 1990.

[QH96] Michael J. Quinn and Philip J. Hatcher. On the Utility of Communication–Computation Overlap in Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 33(02):197, 1996.

[Ric95] Harvey Richardson. *High Performance Fortran: history, overview and current developments*. Thinking Machines Corporation, 245 Firts Street, Cambridge, USA, March 1995.

[RP94] Anne Rogers and Keshav Pingali. Compiling for distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):281–298, March 1994.

[Rut98a] Oliver Ruthing. Bidirectional data flow analysis in code motion: Myth and reality. In *SAS*, 1998.

[Rut98b] Oliver Ruthing. Optimal code motion in the presence of large expressions. In *IEEE International Conference on Computer Languages*, pages –, May 1998.

[SBKB97] Anne P. Scott, Kevin P. Burkhart, Ashok Kumar, and Richard M. Blumberg. Four-way superscalar pa-risc processors. *Hewlett-Packard Journal*, 48(4), August 1997.

[Sch77] R.W. Scheifler. An analysis of inline substitution for a structured programming language. *Communications of the ACM*, 20(9):647–654, September 1977.

[Sch97] Robert Schreiber. High performance fortran, version 2. *Parallel Processing Letters*, 7(4):437–449, 1997.

[SD99] Georges-André Silber and Alain Darte. The Nestor library: A tool for implementing Fortran source to source transformations. In *High Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 653–662. Springer Verlag, April 1999.

[Set75] R. Sethi. Complete register allocation problems. *SIAM Journal of Computing*, 4(3):226–248, September 1975.

[SGH97] Vatsa Santhanam, Edward Gornish, and Wei-Chung Hsu. Data prefetching on the hp pa-8000. In *International Symposium on Computer Architecture*, pages 264–273, June 1997.

[SJH89] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on multiple instruction issue. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 24, pages 290–302, New York, NY, May 1989. ACM Press.

[SK80] Randolph G. Scarborough and Harwood G. Kolsky. Improved optimization of FORTRAN object programs. *IBM Journal of Research and Development*, 24(6):660–676, November 1980.

[SKK+96] A. Sohn, J. Ku, Y. Kodama, M. Sato, H. Sakane, H. Yamana, S. Sakai, and Y. Yamaguchi. Identifying the capability of overlapping computation with communication. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 133–138, Boston, Massachusetts, October 1996. IEEE Computer Society Press.

[SKR90] Bernhard Steffen, Jens Knoop, and Oliver Rüthing. The value flow graph: A program representation for optimal program transformations. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 389–405, Copenhagen, Denmark, May 1990. Springer.

[SKR91] B. Steffen, J. Knoop, and O. Ruthing. Efficient code motion and an adaptation to strength reduction. In *International Joint Conference on Theory And Practice of Software Development*, pages 394–415, April 1991.

[SL96] André Seznec and Fabien Lloansi. étude des architectures des microproceseurs MIPS R10000, Ultrasparc et Pentium Pro (in french). Technical Report 1024, IRISA Rennes, France, May 1996.

[SM95] André Seznec and Yann Mével. Etude des architectures des microprocesseurs DEC21164, IBM Power2 et MIPS R8000. Technical Report 932, IRISA, June 1995.

[SMJ99] Michael Schlansker, Scott Mahlke, and Richard Johnson. Control CPR: A branch height reduction optimization for EPIC architectures. In *ACM SIGPLAN Notices - PLDI'99*, volume 34, pages 155–168, May 1999.

[SOHL+96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The complete reference.* MIT, MIT Press edition, 1996.

[SRR92] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptative pattern matching. In W. Kuich, editor, *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, number 623 in Lecture Notes in Computer Science, pages 247–260, Wien, Austria, 1992. Springer-Verlag, Berlin.

[SS94] B.S. Siegel and P.A. Steenkiste. Automatic Generation of Parallel Programs with Dynamic Load Balancing. In *Third International Symposium on High-Performance Distributed Computing*, San Fransisco, August 1994. http://www.cs.cmu.edu/afs/cs/project/nectar/WWW/appl_papers.html.

[SS95a] B.S. Siegel and P.A. Steenkiste. Controlling Application Grain Size on a Network of Workstations. In *Supercomputing'95*, 1995. http://www.cs.cmu.edu/afs/cs/project/nectar/WWW/gnectar_papers.html.

[SS95b] James Smith and Gurindar Sohi. The microarchitecture of superscalar processors. In *Proceedings of the IEEE*, August 1995.

[Ste96] Bernhard Steffen. Property oriented expansion. In *SAS/ALP/PLILP'96*, volume 1145, pages 22–41, Aachen (D), September 1996. Springer Verlag. Proc. Int. Static Analysis Symposium (SAS'96),.

[SUN99] microsystems SUN. *UltraSPARC-IIi CPU - Data Sheet*, May 1999.

[SV94] André Seznec and Thierry Vauleon. Etude comparative des architectures des microprocesseurs Intel Pentium et PowerPC 601. Technical Report RR-2320, Inria, Institut National de Recherche en Informatique et en Automatique, 1994.

[sys96] The C++// system. D. caromel, f. belloncle, y. roudier. In *Parallel Programming Using C++*, pages 257–296, 1996.

[Tar72] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), June 1972.

[Tex99] Texas Instrument. *TMS320C6000 CPU and Instruction Set Reference Guide*, March 1999.

[TFI86] Rémi Triolet, Paul Feautrier, and François Irigoin. Direct parallelization of call statements. In *ACM SIGPLAN Symposium on Compiler Construction*, volume 21(7), pages 176–185, July 1986.

[Tri84] Rémi Triolet. *Contribution à la parallélisation automatique de programmes Fortran comportant des appels de procédures.* PhD thesis, Paris VI University, 1984.

[Tse93] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.

[Ull73] J. D. Ullman. Fast algorithms for the elimination of commons subexpressions. *Acta Informatica*, pages 191–213, 1973.

[VNL97] Steven VanderWiel, Daphna Nathanson, and David Lilja. Complexity and performance in parallel programming languages. In *High-Level Programming Models and Supportive Environnements*, pages 3–12, April 1997.

[VWL97] Steven P. Vander Wiel and David J. Lilja. When caches aren't enough: Data prefetching techniques. *Computer*, pages 23–30, July 1997.

[Wai74] W. M. Waite. *Optimization*, pages 549–602. Springer Verlag, 1974.

[Wal93] David W. Wall. Limits of instruction-level parallelism. Technical Report 93/6, DEC-WRL, November 1993.

[WCES94] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Principles of Programming Languages*, pages 297–310, Portland, Oregon, January 1994. ACM Press.

[WFW+94] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, Shih-Wei Liao, Chau-Wen, Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: an infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[WL91] Michael E. Wolf and Monica S. Lam. Loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[WM91] Sun Wu and Udi Manber. AGREP - A fast approximate pattern-matching tool. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 153–162, Berkeley, CA, USA, January 1991. Usenix Association.

[WMmWHR93] Nancy J. Warter, Scott A. Mahlke, Wen mei W. Hwu, and B. Ramakrishna Rau. Reverse if-conversion. In Robert Cartwright, editor, *Proceedings of the Conference on Programming Language Design and Implementation*, pages 290–299, New York, NY, USA, June 1993. ACM Press.

[Wol90] Michael Wolfe. *Optimizing supercompilers for supercomputers.* The MIT Press, Cambridge, MA, 1990.

[Wol96] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.

[WSS+98] Jason T. L. Wang, Bruce A. Shapiro, Dennis Shasha, K. Zhang, and K. M. Currey. An algorithm for finding the largest approximately common substructures of two trees. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 20, pages 889–895, August 1998.

[WZJS94] J. T. L. Wang, K. Zhang, K. Jeong, and D. Shasha. A system for approximate tree matching. In *IEEE Transactions on Knowledge and Data Engineering*, volume 6, pages 559–571, August 1994.

[Xue94] Jingling Xue. Automating non-unimodular loop transformations for massive parallelism. *Parallel Computing*, 20(5):711–728, April 1994.

[YAI94] Yi-Qing Yang, Corinne Ancourt, and François Irigoin. Minimal data dependence abstractions for loop transformations. In *Languages and Compilers for Parallel Computing*, July 1994. Also available as TR EMP A/266/CRI.

[Yan93] Yi-Qing Yang. *Tests des Dependances et Transformations de Programme.* PhD thesis, Université Paris VI, November 1993.

[YFA97] H W Yau, G C Fox, and Hawick K A. Evaluation of high performance fortran through application kernels. In *High Performance Computing and Networking*, pages 772–780, April 1997.

[ZC91] H. Zima and B. Chapman. *Supercompilers for parallel and vectors computers.* ACM Press, New York, 1991.

[ZC98] Julien Zory and Fabien Coelho. Using algebraic transformations to optimize expression evaluation in scientific codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[Zha92] Hantao Zhang. Herky: High performance rewriting in RRL. In Deepak Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607, pages 696–700, Saratoga Springs, NY, June 1992. Springer.

[Zho94] Lei Zhou. *Statical and Dynamical Analysis of Program Complexity.* PhD thesis, Université Paris VI, September 1994.

[Zor96] Julien Zory. Macro-pipelines dans les programmes data-parallèles. Master's thesis, LIP ENS-Lyon and INSA-Lyon, 1996.

[Zor98] Julien Zory. Porting a 2d acoustic wave modeling application to hpf : issues and problems. In *HPF User Group Meeting*, June 1998. Also available as technical report number CRI-ENSMP A-308.