Optimisation Globale du Placement d'Applications de Traitement du Signal sur Architectures Parallèles en Utilisant la Programmation Logique avec Contraintes

Version définitive

Christophe Guettier

à Sabine

Remerciements

Cette thèse est le résultat de trois années de recherches conduites au Laboratoire Central de Recherches de Thomson, Groupe Informatique ainsi qu'au Centre de Recherches en Informatique de l'Ecole des Mines de Paris. Ce document scelle une collaboration pluridisciplinaire exemplaire entre chercheurs académiques et industriels. Elle se concrétise aussi bien par une avancée technologique utilisable que par des acquis et résultats scientifiques défiant toute autre approche.

Je remercie Jean Jourdan (Thomson-LCR, Groupe Informatique, Laboratoire Architectures Technologies Systèmes), premier artisan de cette synergie. En me plaçant au cœur même de ces recherches, il m'a transmis l'ambition de résoudre des problèmes informatiques de grande échelle, ainsi que la philosophie pour y parvenir.

Je remercie François Irigoin (Centre de Recherches en Informatique de l'Ecole des Mines de Paris), mon directeur de thèse, dont la sagesse contribue aux fondations de cette thèse. Il a su me conduire, grce à son recul et son expertise, dans un monde de défis complexe et très vaste qu'est le *calcul de hautes performances*.

Un grand merci à Juliette Mattioli (Thomson-LCR, Groupe Informatique, Laboratoire Architectures Technologies Systèmes), dont les conseils et remarques pertinentes, ainsi que son esprit de synthèse toujours constructif et positif ont bien souvent permis de consolider et concevoir les modèles utilisés dans cette thèse.

L'intuition et les capacités de dialogue de Corinne Ancourt (Centre de Recherches en Informatique de l'Ecole des Mines de Paris) ont été un atout précieux durant l'ensemble de mes investigations. Qu'elle soit ici vivement remerciée pour son aide et les connaissances qu'elle m'a apporté sur ce sujet.

Les Professeurs Franois Fages et Patrice Quinton ont acceptés de rapporter ce document, je les remercie pour ce long travail, certainement difficile.

Les explications de Jean Sageloli (Laboratoire de Modélisation Accoustique, CNRS) m'ont été d'un grand secours sur la compréhension du domaine applicatif, qu'il en soit ici remercié.

Je remercie aussi Bertrand Jeannet (Thomson-LCR, Groupe Informatique, Laboratoire Architectures Technologies Systèmes) et Denis Barthou (Laboratoire PRISM de l'UVSQ) pour leur étroite et "chaude" collaboration.

Les conseils et encouragements de Franois Laburthe (Ecole Normale Supérieure) et les nombreux échanges que nous avons eu sur l'optimisation combinatoire m'ont beaucoup aidé dans la rédaction de ce document, qu'il en soit aussi remercié.

Je tiens de même à remercier Beatrice Creusillet, Alexis Platonoff, Fabien Coelho (Centre de Recherches en Informatiques de l'Ecole des Mines de Paris) pour leurs encouragements ainsi que les fructueuses discussions que nous avons eu.

Je remercie Michel Coatrieux (Thomson-Airsys) et Didier Rozzonelli pour leur aide dans l'étude des placements manuels, leurs encouragements et leur enthousiasme.

Je tiens à remercier les dirigeant successifs du CET (Thomson-CSF) traitement du signal numérique pour leur aide et leur interêt dans ces travaux : Thierry Brizard, Gilbert Edelin, ainsi que l'équipe de Thomson Marconi Sonar : Alain Demeure pour avoir exposé sa vision du problème, Bruno Marchand pour ses précisions techniques concernant les critères de performance, enfin Alain Albarello pour avoir financé cette thèse.

Je remercie Béatrice Bacconnet (Thomson-LCR, Groupe Informatique) pour son soutien, en souhaitant à l'ensemble du Groupe Informatique de connaître l'essor qu'il mérite. Merci aussi à tous ceux qui ont su faire du Groupe Informatique un lieu caleureux où l'on contribue scientifiquement à des travaux de grande envergure.

Résumé

Le traitement du signal digital numérique exige dorénavant des machines parallèles spécialisées de hautes performances. Leur exploitation exige de la part des experts non seulement une parfaite connaissance de l'architecture mais aussi l'emploi au mieux du potentiel des ressources matérielles (taille mémoire, puissance de calcul, réseau de communications), tout en satisfaisant les contraintes temps-réels et de coût.

Les performances sont donc difficiles à obtenir, de par la finesse et le nombre des optimisations à réaliser comme par la quantité et la diversité des contraintes à satisfaire. Bien connus des chercheurs, ces problèmes de placement ont fait l'objet de nombreuses techniques spécifiques : partitionnement, alignement ou ordonnancement. Selon les approches, la résolution automatique tient compte de manière indépendante, soit de l'allocation des données, soit des communications, soit des paramètres temps-réels. Cependant, la résolution s'avère dans la plupart des cas NP-difficile et les techniques n'adressent qu'une partie du problème général.

Aborder globalement le problème du placement automatique revient à considérer simultanément les différents formalismes sous-jacents aux techniques mentionées ci-dessus. Le problème général est décomposé en différents modèles (partitionnement, ordonnancement, alignement, communications, architecture, allocation mémoire), chacun d'eux disposant d'une combinatoire propre. La Programmation Logique par Contraintes (PLC) s'impose alors de par la nature des formalismes considérés, par sa capacité à résoudre des problèmes combinatoires, enfin par son aptitude à supporter des modèles concurrents. Nous montrons sous quelles conditions chaque modèle peut être raffiné et exprimé en PLC à partir des formalismes issus de l'état de l'art en calcul de haute performance. De plus, il est nécessaire de garantir la coopération et la concurrence entre les modèles pour obtenir une solution globale cohérente. Pour cela, les contraintes non-linéaires formalisant les relations entre les différents modèles sont mises en évidences. Des stratégies globales, combinées à différentes heuristiques, coordonnent et supervisent la recherche de solutions sur l'ensemble des modèles.

Un prototype de placement a été réalisé qui exploite la modélisation multiple. Outre le calcul automatique de solutions, il permet l'optimisation de paramètres-clefs comme la latence, le cot de l'architecture ou bien l'efficacité du placement tout en considérant les contraintes de ressources (mémoires, processeurs, bande passante). Le prototype se révèle optimal sur des cas d'écoles et efficace sur des applications réelles. Ces dernières optimisations fournissent des solutions équivalentes à celles obtenues manuellement par les experts en traitement du signal.

Table des matières

Première partie

Présentation de la thèse

Avant-propos

Contexte et importance industrielle du thème de recherches

Au sein des systèmes radars et sonars, plusieurs domaines technologiques sont sollicités afin d'exploiter au mieux l'information présente dans les signaux captés. En premier lieu, la maîtrise des aspects physiques et mathématiques permet de définir les fonctions capables d'analyser le signal. Un radar ou sonar se compose de plusieurs niveaux de traitement, s'échelonant du bas niveau (le traitement du signal numérique) vers le haut niveau (traitement de données). Le traitement du signal numérique est caractérisé par des fonctions essentiellement numériques comme la *Transformée de Fourier, la Formation de Faisceaux, l'équation Doppler*, parmi les plus connues. Le haut niveau peut comporter par exemple des fonctions de reconnaissance de formes spectrales. En second lieu, il est nécessaire d'adapter les algorithmes développés selon les propriétés du milieu, le type de cibles, les contraintes temps réel, les contraintes liées à l'embarquement. En somme, autant d'éléments intervenant tant dans les choix de conception que de réalisation : sélection, dimensionnement et configuration des matériels adaptés, définition des fonctions composant le logiciel. Les ingénieurs doivent alors nécessairement gérer des compromis, optimiser les performances, le pouvoir de résolution, la fidélité, la fiabilité des systèmes de traitement du signal. Dans ce sens, l'utilisation des architectures parallèles permet un gain de performances, particulièrement dans le cadre du traitement du signal numérique, et d'ambitieuses réalisations industrielles ont vu le jour.

Toutefois, cette ingéniérie déjà difficile, est complexifiée aussi bien par la constante évolution des architectures que par l'ajout de nouvelles fonctions de traitement du signal. De ce fait, les capacités de traitement s'accroîent avec les volumes de données à traiter. Dans les années 70, de telles architectures atteignent le "Mega-flops"[1], dans les années 80 l'ordre de grandeur est le "Giga-flops", contre le "Tera-flops" dans les années 90. Face à cette évolution, la maîtrise des temps et coûts de développement sur ces architectures est prépondérante.

Exploiter le parallélisme et ainsi les performances de telles architectures est un problème reconnu complexe, l'ensemble des choix possibles étant très important, de même que le nombre de contraintes à satisfaire par l'ingénieur. Différents outils et langages automatisent certaines fonctions séparément, comme la distribution du code et des données, l'ordonnancement, l'allocation mémoire. Cependant, une grande partie des décisions importantes reste encore du ressort de l'ingénieur. Placer une application en exploitant la combinatoire globale du problème – et ainsi résoudre la totalité des fonctions plutôt que d'adresser séparement plusieurs problèmes fortement combinatoires – est la problématique étudiée par cette thèse.

[1]flops : floating operation per second, en Français opérations flotantes par seconde.

Historique du projet de recherches

En 1993, sur la classe applicative du traitement du signal numérique, un premier prototype de placement automatique fut réalisé au laboratoire Central de Recherche de Thomson-CSF à la demande d'une unité opérationnelle. Si le prototype démontrait la faisabilité, le niveau de granularité de l'analyse n'était pas satisfaisant pour pouvoir apporter des solutions suffisament pertinentes. Le principal intérêt de cette étude est d'avoir établi que le problème du "placement" n'est pas soluble par des méthodes d'optimisation numériques pures mais qu'il nécessite une approche de type Intelligence Artificielle.

En 1994, plusieurs expérimentations sur ce premier prototype ont démontré la modularité de l'approche, apportant plus de facilité et de souplesse à la modélisation. En 1995 et 1996, avec le concours du Centre de Recherches en Informatique de l'école des Mines de Paris, nos efforts se sont alors focalisés sur la définition des modèles de placement (partitionnement, ordonnancement, communications,...) et les liens qui les régissent.

L'objectif de cette thèse (soutenue par contrat DRET G1 91/306/094) est de présenter un système d'aide à la conception et à la configuration de machines parallèles *de type SPMD* pour le placement automatique d'applications de traitement de signal.

La prise en compte de l'ensemble des ressources de la machine, de toutes les composantes du placement – tel que le partitionnement, l'alignement, les communications, l'allocation, la distribution des données – nécessite de combiner les différentes techniques de placement. Cette combinaison peut être réalisée par composition de contraintes, c'est-à-dire en introduisant les méthodes issues de la programmation avec contraintes. Cette dernière offre un cadre cohérent permettant la composition des contraintes via des connecteurs logiques ou règles de propagation. Le haut niveau d'expression qu'elle offre permet de concevoir des modèles intrinsèques à la fonction placement, comme le partitionnement des données, le séquencement des calculs et des communications, *etc....*

La résolution de ce problème utilisant cette approche multi-modèles est le fil conducteur de ce travail et est finalisée par un démonstrateur. Elle nous permet d'appréhender la complexité du placement automatique de traitement de signal sur machine SPMD de manière globale. Nous avons étudié dans une première phase différents modèles (partitionnement, ordonnancement des calculs et des communications, architecture d'une machine de type SPMD, contraintes applicatives) afin d'en avoir une modélisation mathématique claire. L'implantation de ces modèles dans un langage de contraintes concurrentes est l'objet de la deuxième phase du travail. Celle-ci repose sur l'écriture de conditions suffisantes que doivent satisfaire les différents modèles. Enfin, des résultats équivalents à ceux des experts sur des exemples réels ont permis de valider l'approche.

Introduction

Les architectures parallèles dédiées au traitement du signal n'échappent pas à la difficulté de programmation. Les machines sont soumises à des flots de données extrêmement intensifs sur lesquels il faut effectuer des traitements numériques quasiment systématiques. La vitesse d'acquisition des données ainsi que les temps de réponses nécessitent de tirer partie au mieux de l'architecture. Pour cela, le développeur définit finement l'emplacement des calculs et des données sur les différents processeurs, et configure les communications. Ainsi, la plupart des décisions sont prises statiquement à la conception, évitant ainsi les décisions dynamiques. L'architecture parallèle est donc entièrement dédiée aux calculs. Diverses méthodes sont nées des nombreuses recherches pour optimiser le placement d'une application sur une machine parallèle. Ces méthodes consistent à exprimer soit ce qui est parallèle, soit ce qui est séquentiel dans une application. Lorsqu'elles sont automatiques, l'expression du problème et sa résolution reposent généralement sur l'algèbre linéaire et la programmation en nombre entier qui permettent de considérer le degré de finesse recherché. Les contraintes de ressources comme la taille mémoire, le nombre de processeurs ou la latence sont par essence non linéaires, et une vision unique du problème ne suffit plus.

Il faut donc représenter le problème sous toutes ses facettes pour en maîtriser la complexité globale. Seule la Programmation avec Contraintes, grâce à ses propriétés de compositionalité et de généricité permet d'une part d'établir facilement le problème sous ses différents aspects, d'autre part de maintenir la cohérence générale afin de résoudre efficacement le problème.

Les domaines sollicités

Cette thèse se trouve donc à l'intersection de trois domaines : le **traitement de signal** (domaine applicatif), le **parallélisme à grain fin** (domaine de validation) et la **modélisation concurrente** (domaine de résolution).

Le **traitement du signal** fournit une classe applicative privilégiée pour expérimenter une approche globale dont l'ambition est la résolution de l'ensemble des problèmes composant le placement. En effet, les applications sont régulières, voire cycliques, et présentent des fonctions d'accès linéaires dans les tableaux. Cependant, les paramètres temps réel du signal sont à prendre en compte : les cadences auxquelles les résultats doivent être délivrés, ou les cadences d'acquisition de données sont importantes mais la latence est un point encore plus important car il définit l'actualité d'un résultat. Ces contraintes temps réel font l'objet de nombreux travaux, dont l'environnement de prototypage rapide Ptolemy [LM87] [BSL96]. Dans le même sens, différents système comme SynDEx tirant partie des caractéristiques du traitement du signal ont été développés [LS92] . La sélection des solutions est influencée principalement par les contraintes temps-réel ou bien l'optimisation de la mémoire [BML95] [MBL96]. Malgré cela, les solutions obtenues par les experts en traitement du signal en milieux industriels sont plus pertinentes, résultant d'une plus grande finesse d'analyse et d'optimisation. Les calculs sont finement partitionnés et distribués sur des architectures parallèles performantes. Les solutions obtenues par ces experts permettent de calculer finement ordonnancement et allocation ainsi que les communications. En réalité, exception faite du très récent compilateur FX [OWS97], très peu de travaux établissent aujourd'hui explicitement un lien entre le parallélisme à grain fin et les contraintes temps réel de traitements du signal. Cette dernière approche est basée sur les directives usuelles en calcul de hautes performances et parallélisation automatique : alignement et distribution. En revanche, aucune approche ne résoud le problème globalement.

Les techniques de parallélisation automatique cherchent à extraire, à l'aide d'analyses statiques, un maximum de parallélisme d'un programme séquentiel [Ber66] [BC86] [Ban88] [IT87] [LYZ89] [Fea91] [Pug92] [AYI94]. La complexité grandissante des architectures parallèles impose une finesse d'analyse déterminante pour la qualité du placement. En effet, ce dernier consiste à distribuer automatiquement les calculs et les données sur la machine cible multi-processeurs en conservant le maximum de parallélisme. Pour cela, une première méthode consiste à exprimer l'ensemble des opérations obligatoirement séquentielles en calculant un ordonnancement à grain fin [Fea92b] [Fea92a] [DR92] [DR94a]. La représentation duale consiste à exprimer le parallélisme, par exemple en déterminant un partitionnement [IT88b] [SD90] [FGMS92] [LL94] . Dans les deux cas, la transformation de programme ne doit pas modifier les dépendances de flots de données entre les opérations du programme.

Bien que le parallélisme de l'application soit susceptible d'être mis en évidence, le placement automatique, constitué de la distribution des calculs sur les processeurs et de leur ordonnancement, est néanmoins un problème difficile. Il faut tenir compte du nombre de processeurs, des dépendances, de l'alignement des données, des communications, de l'allocation en mémoire et des performances du couple architecture/application. Chacune de ces composantes implique un sous-problème intrinsèque, si bien que considérer l'un d'eux et trouver un ordonnancement ou une distribution est NP-difficile [Dar93] [Kre93a]. Les approches proposées consistent à calculer soit un ordonnancement, soit une distribution selon l'une de ces composantes essentielles au problème (alignement, dépendances, allocation mémoire, communications).

Cependant, l'expression et la résolution du placement dépendent principalement du type d'applications et de la finesse de leur description de même que pour les architectures. La granularité joue ainsi un rôle prépondérant. Or, les applications de Traitement du Signal peuvent être représentées à l'aide de boucles parallèles et séquentielles. De plus, les architectures se composent de plusieurs processeurs et d'une mémoire répartie. Elles sont généralement du type SIMD "Single Instruction flow, Multiple Data flow". Dans ces conditions, l'algèbre linéaire ainsi que les graphes orientés permettent l'expression des sous-problèmes qui composent le placement. De plus, ces expressions permettent de situer le placement au **niveau de granularité requis par le couple architecture-application**. Les manipulations de matrices et de graphes correspondent alors à des transformations de programme.

La résolution de chacun de ces sous-problème repose usuellement sur des techniques de Programmation Linéaire en Nombre Entier ou de Recherche Opérationnelle, adaptées aux transformations de matrices ou de graphes [Fea88] [LC90] [KM93] [BKK94] [BKPS94]. Néanmoins le manque de généricité et de compositionalité de ces techniques ne permettent pas de combiner et de résoudre efficacement plusieurs composantes sur des exemples probants. En fait, peu de travaux s'attachent à résoudre plusieurs sous-problèmes de placement. Lorsque c'est le cas, le placement se résume à **la résolution séquentielle d'une suite de sous-problèmes**. Par exemple, alignement des données puis distribution, ou bien calcul des dépendances suivit de l'ordonnancement, fusion de boucles et distribution. La combinatoire des transformations est tellement importante que des solutions-heuristiques[2] sont souvent employées comme la minimisation des communications pour résoudre le moindre sous-problèmes.

Aussi, devant la difficulté de l'automatisation, plusieurs langages de programmation on été conçus, laissant l'utilisateur confronté à la résolution du problème de placement. Programmer dans un langage comme High Performance Fortran (HPF) [KLS$^+$94] nécessite donc de spécifier des directives efficaces sur des programmes aux nombreuses dimensions de calcul en tenant compte des contraintes d'architectures. Ce travail demande là encore un haut niveau d'expertise et une connaissance de l'architecture élevée.

Pourtant, quelques travaux [BKK94] menés à l'aide de la Programmation Linéaire en variables $0-1$ dénotent l'interêt de résoudre deux problèmes simultanément : l'alignement et la redistribution des données. Mais le nombre de variables explose très rapidement. De plus, le contrôle de la résolution et la modélisation ne sont pas séparés, ce qui ne permet pas de résoudre efficacement le problème.

Alain Darte a démontré [Dar93] que résoudre simultanément la distribution et l'ordonnancement n'est pas du ressort de l'algèbre linéaire, même lorsque les dépendances sont caractérisées à l'aide de vecteurs uniformes. De plus, Paul Feautrier [Fea94a] a mis en évidence que la prise en compte de ressources de natures différentes pour calculer un ordonnancement requiert des contraintes non-linéaires. En conclusion, il résume la difficulté de résolution en une phrase :

"The question now is: is there a better way of solving directly the non-linear constraints [...] than by a combination of integer programming and search?"[3]

Cette question montre avant tout que la programmation linéaire en nombre entier ne peut pas être l'unique moyen de résolution. De plus, elle soulève la difficulté de considérer à la fois recherche et résolution de contraintes. Ce problème est bien connu dans le domaine de la programmation par contraintes. Très tôt, les chercheurs de ce domaine se sont rendus compte de la nécessité d'allier le non-déterminisme et la résolution des contraintes. Déjà selon cette idée, Hervé Gallaire explique en 1985 :

"Manipulating constraints, deciding which or when to use them, is a complex matter ... We know the limitations of approaches relying totally on constraints (e.g. scheduling in Operations Research). What must be found then – and this is where logic programming will show at its best – is how to mix problem-solving with constraint handling."[4]

Ainsi, c'est bien le contrôle de la recherche et de la résolution de contraintes qui permet d'aborder la complexité d'un problème. En l'occurence, il s'agit des différentes fonctionnalités complexes qui composent le placement. Or, la Programmation Logique avec Contraintes concurrentes dispose désormais d'opérateurs de synchronisation et d'activation portant aussi bien sur la résolution de contraintes que sur la recherche de solutions. Il est donc possible de dissocier les différentes parties du problème du placement et de combiner ces opérateurs afin de le résoudre globalement.

L'état de l'art démontre que l'algèbre linéaire est adapté à l'expression du problème de

[2]On appele ici solutions-heuristiques, des solutions obtenues directement en appliquant une heuristique sans parcourir tous les choix possibles.

[3]La question est : Y a-t-il un meilleur moyen de résoudre directement les contraintes non-linéaires [...] que par une combinaison de programmation en nombre entier et de recherche ?

[4]Manipuler les contraintes, décider quand ou comment les utiliser est un problème complexe ... Nous savons les limitations des approches reposant totalement sur les contraintes (i.e. ordonnancement en Recherche Opérationnelle). Ce qui doit être trouvé alors – et c'est là que la programmation logique montrera sa force – est comment mixer la résolution de problèmes et la manipulation de contraintes.

placement. Elle permet en effet de formuler un lien entre la granularité de l'architecture et celle de l'application. Plutôt que d'associer vainement plusieurs algorithmes de résolution, il est préférable de résoudre le placement en considérant la globalité de sa combinatoire. Cette globalité nécessite de composer plusieurs modèles formels à l'aide de contraintes, où chacun d'entre eux représente une fonctionnalité du placement (distribution des calculs, ordonnancement, ...). Complexité et résolution globales exigent alors un environnement de développement permettant de combiner résolution de contraintes et recherche de solutions. Un langage concurrent de contraintes, associé à une modélisation multiple du placement consituent une réponse à ces nécessités.

L'approche suivie dans cette thèse considère le placement programme de traitement du signal exposant un parallélisme potentiel. Cela consiste à résoudre l'ordonnancement et le partitionnement des opérations de ce programme sur l'architecture cible en tenant compte des contraintes de ressources et de temps-réel. Notre approche exploite donc simultanément les différentes composantes du placement : ordonnancement, partitionnement, communications, capacité mémoire, architecture cible, temps réel. Contrairement aux autres approches, **la résolution est globale et concurrente sur l'ensemble des modèles constituant le placement automatique**. Cela nécessite que la granularité des différents modèles puisse concorder. De ce fait, face aux nombreuses composantes et à la complexité du problème, la finesse de la modélisation à l'aide de contraintes est déterminante pour la résolution du problème. Si bien que plusieurs questions se sont posées : Quels est le langage adapté pour exprimer les modèles issus des techniques de parallélisation/placement ? Ne faut-il pas adapter ces modèles aux caractéristiques du traitement du signal ? Comment composer les modèles pour concevoir la résolution ? Le nombre de contraintes et de variables n'entravent-ils pas la résolution ?

En fait l'expression d'un modèle doit être rendue suffisament générique pour qu'il puisse être composé avec un autre. Les relations entre ceux-ci ainsi que les contraintes de ressources se traduisent alors par des contraintes complexes, bien souvent non-linéaires. Modèles et relations nécessitent donc d'introduire un véritable langage de programmation disposant d'un haut niveau d'expressivité. Enfin, chaque modèle constitue un problème NP-difficile, et la solution globale doit être cohérente vis-à-vis de chaque modèle. Pour cela, il faut être capable d'articuler la résolution entre les différents modèles, en tenant compte de la non-linéarité.

La **Programmation Logique avec Contraintes** apporte un cadre formel privilégié [JL87] afin de développer un environnement d'optimisation pour les problèmes de placement. Ses propriétés de compositionalité, de modularité et de généricité [Sar87a] [VSD92a] [Jou95] correspondent au besoin d'exprimer et d'articuler facilement la résolution des différents modèles complexes de placement. Le système de Programmation Logique par Contraintes est en effet très indiqué pour exprimer les contraintes propres aux techniques de placement/parallélisation automatique, généralement basées sur l'algèbre linéaire. Le contrôle de la résolution, fondé sur l'échange d'informations partielles et la coordination des procédures de résolution [JVP90], permet de tirer partie de l'information présente dans chaque modèle. La résolution s'appuie de cette façon simultanément sur les différents modèles, et considère une combinatoire globale pour résoudre le problème du placement. Une bonne illustration de l'utilisation de notre approche est l'optimisation de la latence ou bien de fonctions de coût comme celui de l'architecture parallèle.

Cependant, la Programmation Logique avec Contraintes seule n'est pas suffisante pour résoudre directement le problème du placement. Le nombre de contraintes atomiques mises en œuvre et la complexité du problème sont bien trop important pour raisonner à ce niveau élémentaire. Cela que ce soit au niveau de la résolution comme de la compréhension du problème. Comme nous l'avons déjà subodoré, il faut associer au langage une méthodologie de modélisation concurrente proposée en [Jou95]. Déjà éprouvée en milieu industriel, elle permet de définir des modèles multiples concurrent. Les modèles peuvent être :

- Coopérant : les modèles décrivent des sous-parties d'un même problème.
- Redondant : les modèles représentent un même concept. La résolution peut alors tirer partie de différents aspects du problème.
- Hiérarchiques : un modèle peut se décomposer en sous-modèles.

Cette méthodologie permet aux modèles de placement d'hériter des propriétés du langage (généricité, compositionalité, modularité). Elle définit en fait un niveau d'abstraction améliorant la compréhension du problème et autorise l'articulation autour des modèles de la résolution. En résumé, l'approche confère à l'environnement d'optimisation du placement automatique de nombreux avantages :

- Meilleure compréhension du problème car tous les aspects sont représentés sous formes de modèles aisément manipulables.
- Résolution efficace, articulée autour des critères pertinents de placement et de leur composition. La modélisation par contraintes concurrentes laisse libre cours à la spécialisation des recherches, comme pour des optimisations particulières.
- Modularité des composantes du placement, héritage de la propriété de compositionalité de la Programmation Logique avec Contraintes. Chaque modèle peut être vue comme un composant logiciel.
- Maintenabilité et évolutivité. Nous montrerons par exemple comment il est aisé de changer d'architecture cible ou de modèle mémoire.

L'objectif de cette thèse est donc la réalisation à partir de modèles formels issus de l'état de l'art en placement/parallélisation d'un système de placement d'applications de traitement de signal. Le système considère une application ainsi qu'une architecture et ses ressources (processeurs, mémoires, réseau) et résoud le problème du placement. L'optimisation de fonctions de coût est envisageable, comme la latence de l'application ou l'efficacité des solutions. L'ensemble des contributions de la thèse peut se résumer dans les points suivant :

- Adapter au contexte du Traitement du Signal, pour chacune des composantes, les techniques de parallélisation/placement.
- Spécialiser et adapter les modèles à la Programmation Logique avec Contraintes.
- Implanter les différents modèles.
- Définir des stratégies de résolution efficaces selon les critères à optimiser.
- Exploiter la modularité et la maintenabilité de l'approche.

Enfin, ceci a fait l'objet de nombreuses expérimentations, en particulier sur des applications industrielles réelles. Les résultats sont caractérisés par des solutions proches de celles des experts de Thomson-CSF.

Architecture du document

La partie [II] constitue un état de l'art sur le *placement automatique des applications de traitement du signal.*

Les résultats sur la **parallélisation et le placement automatique** (chap. 1), sur l'exploitation des architectures parallèles et sur les langages de programmation parallèle, sont présentés. Cet état de l'art expose comment sont usuellement représentés les problèmes de placement et par exemple quel rôle joue la granularité. De plus, il introduit la complexité du placement et permet ainsi d'appréhender la structure du problème dans sa globalité. Dans le même temps, nous expliquons comment et pourquoi la plupart des modèles formels détaillés en partie [III] sont inspirés de ces résultats.

La partie de l'état de l'art sur le **traitement du signal** (chap. 2) caractérise les algorithmiques ciblés et donne le niveau de granularité requis pour l'exploitation des ressources de calculs. Elle présente aussi comment les experts parviennent à résoudre le problème du placement des applications de traitements du signal sur des architectures parallèles. Cette connaissance est détaillée sur un exemple classique du domaine et témoigne de la globalité de leur approche. Nous donnons dans ce chapitre les travaux sur le placement automatique dédiés au traitement du signal. Ils illustrent le décalage entre la finesse des optimisations effectuées par l'expert et les systèmes automatisés actuels.

Le chapitre (3), présente **comment divers sous-problèmes de placement sont résolus** à l'aide de la Programmation Linéaire en Nombre Entier ou de la Recherche Opérationnelle. Au travers d'exemples, nous donnons les différentes approches considérées et les questions qu'elles suscitent.

La partie suivante donne un éclairage sur toutes les fonctions sous-jacentes au placement (distribution des calculs et des données, séquencement des traitements, *etc...*). Cette partie [III] détaille le contenu de chaque modèle. Pour chacun d'entre eux, nous montrons **comment sont transformées des méthodes de parallélisation automatique en formalismes adaptés au traitement du signal**. En particulier, le choix des approximations, le niveau de granularité, la représentation des modèles, et enfin les liens entre modèles seront explicités. Le plan de chacun de ces chapitre reflète notre démarche : on rappelle brièvement l'état de l'art et l'on montre comment à partir de celui-ci et des caractéristiques applicatives, on obtient des modèles constitués de contraintes.

Si chaque modèle est en apparence indépendant, disposant de ses propres niveaux de granularité, des liens apparaissent entre eux. Ces liens, bien souvent composés de contraintes non-linéaires permettent de maintenir la cohérence de la modélisation globale. Ils sont sous-jacents à la modélisation dans la partie [III] mais apparaîtrons explicitement lors de la partie [IV].

C'est bien l'objet de cette partie [IV], que de présenter **la réalisation basée sur la Programmation Logique avec Contraintes reflètant la modélisation et ses inter-connections**. Nous rappelons donc les fondements de la Programmation Logique avec Contraintes et de la méthodologie de modélisation. Nous présentons les problèmes liés à la conception du système (spécialisation de contraintes, approximations, validation ou invalidation de la modélisation). L'optimisation automatique de fonction de coûts, la manipulation de solutions partielles ainsi que la modularité des différentes composantes de l'implantation nous permettent d'introduire un système proche d'un environnement d'optimisation pour le placement.

En partie [V], **différents niveaux d'expérimentations** permettent de valider les modèles et de tester l'efficacité du système de placement. En premier lieu, une interprétation des résultats sur une série de cas d'école permet d'illustrer les modèles et de comprendre leurs interactions au-delà des définitions mathématiques. En deuxième lieu, des stratégies de recherches spécialisées, des heuristiques favorisant les solutions satisfaisantes sont alors définies sur la base de résultats obtenus. Celles-ci accélèrent la recherche et se combinent aisément avec la modélisation coucurrente. La partie [V] démontre combien la réalisation dépasse finalement de loin la simple notion d'arbre de recherche. Plusieurs expérimentations sur des applications industrielles prouvent la maturité de l'approche.

Le résultat d'une collaboration

Différents modèles n'auraient pas pu voir le jour sans une forte collaboration[5] entre les équipes du Centre de Recherches en Informatique (CRI) de l'École Nationale Supérieure des Mines de Paris et celle du Laboratoire Architectures et Technologies Systèmes (A& TS) de Thomson-CSF/Laboratoire Central de Recherches. Denis Barthou et François Irigoin (CRI), Juliette Mattioli et Bertrand Jeannet (A& TS) ont apporté de nombreux éléments dans différents modèles. Corinne Ancourt (CRI) a supervisé le déroulement de cette thèse en apportant son aide sur l'ensemble des modèles. Enfin, François Irigoin (CRI) et Jean Jourdan (A& TS) ont respectivement assuré le suivi des travaux concernant le parallélisme et la modélisation concurrente. De plus, ce travail a fait l'objet d'un soutien actif du réseau *traitement de signal numérique* de Thomson-CSF, de nombreux experts de Thomson-CSF ainsi que de la DRET (aujourd'hui DSP/DGA).

Les travaux effectués dans le cadre de cette collaboration ont permis la conception d'un système d'aide au placement pour applications de traitement de signal sur machine SPMD. La mise en œuvre informatique de ce dernier ainsi que sa validation, constitue une importante partie (en temps de travail fourni) de ma thèse.

Ce système a permis de trouver, sur l'ensemble des "benchmarks" testés, des solutions équivalentes à des placements manuels réalisés par des experts de Thomson-CSF.

L'analyse du comportement du système s'est en effet, faite en étroite collaboration avec Jean Jourdan pour l'analyse des méthodes de résolution, Juliette Mattioli pour les applications *cas d'écoles* et avec Corinne Ancourt et François Irigoin pour les applications réelles.

[5]Cette collaboration est active depuis plus de trois ans.

Deuxième partie

États des arts

Introduction

Les caractéristiques des applications de traitement du signal peuvent être exprimées à l'aide de l'algèbre linéaire en nombres entiers. Bien connues des chercheurs travaillant sur le placement et/ou la parallélisation automatique, cette technique permet de représenter et d'exploiter le parallélisme d'une application. Il semble donc naturel d'utiliser les résultats existants, afin de les adapter en complétant si le besoin s'en fait sentir au cas du traitement du signal, et des architectures considérées. Cet état de l'art présente de quelle manière les techniques de placement et/ou parallélisation permettent de considérer les applications de traitement du signal. Cependant, la difficulté de la combinatoire sous-jacente au problème de placement nécessite un outil puissant permettant de coordonner les étapes de résolution et de recherche de solutions. Outre ces deux propriétés, la programmation avec contraintes permet de surcroît d'exprimer les formalismes de placement basés sur l'algèbre linéaire. Ainsi, l'état de l'art, ou plutôt ces états de l'art se composent de trois parties.

Le premier chapitre (chap. 1) rappelle les représentations de placement et/ou parallélisation couramment utilisées. Nous ennonçons celles qui seront utilisées par la suite dans la thèse, tout en les replaçant dans un contexte d'état de l'art afin de mieux justifier les choix et orientations que nous prendrons.

Le second chapitre (chap. 2) est consacrée au traitement du signal. Plus qu'un simple état de l'art décrivant des techniques usuelles permettant d'augmenter les performances, nous avons voulu confronter la pertinence d'une approche théorique à un savoir-faire relevant de l'expertise. Nous nous appuyons pour cela sur un exemple d'échelle réelle qui illustre les choix des experts.

Le troisième chapitre (chap. 3) présente les techniques usuelles de résolution utilisées pour ce type de problème.

Chapitre 1

Le Placement Automatique

1 Complexité du placement

Le placement automatique d'applications de traitement du signal combine plusieurs difficultés. D'une part, le placement *optimal* de nids de boucles présentant un parallélisme potentiel sur une architecture parallèle est un problème reconnu NP-difficile [Kre93a] [Dar93]. D'autre part, les spécificités dues aux traitements du signal apportent des contraintes complexes à satisfaire, comme l'acquisition de données et la latence de l'application[1]. Enfin, des contraintes impératives de ressources (taille mémoire, nombre de processeurs,..) s'ajoutent au problème.

1.1 Complexité intrinsèque

Le problème général du placement optimal est par essence NP-complet, cela a été démontré dans différents contextes, que ce soit d'un point de vue global lorsque l'on considère le placement d'une application complète i.e. plusieurs nids de boucles [Kre93b], ou d'un point de vue local, lorsqu'il s'agit d'un seul nid de boucles [DR94b]. Le placement consiste à associer aux traitements et données les différents nœuds d'une machine parallèle et à les ordonnancer dans le temps, en exploitant au mieux le parallélisme de l'application. Les méthodes de placement proposées établissent comment maximiser le parallélisme ou/et comment éviter des communications. Les contraintes de dépendances, ainsi que de nombreuses techniques de placement, se traduisent par des systèmes d'inéquations pouvant être résolus à l'aide de la programmation linéaire en nombres entiers (PLNE). Peu de travaux ont été effectués abordant le placement avec des contraintes de ressources [Fea94a] parcequ'il introduit de la non-linéarité.

1.2 Contraintes sur les applications de traitement du signal

Les applications de traitement de signal posent des problèmes de compilation et de placement. Outre les problèmes de ressources (taille de la mémoire, nombre de processeurs, taille de la bande passante) de nombreuses contraintes temps réel apparaissent et compliquent encore le problème (cf. chapitre 2, § 2.2). Elles exécutent des macro-instructions définissant un ensemble d'opérations correspondant en

[1] la latence correspond au temps écoulé entre l'acquisition d'une donnée et le dernier résultat auquel elle a contribué.

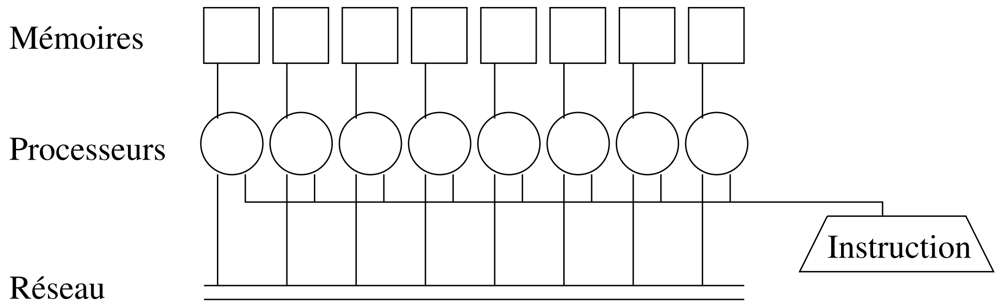


FIG. 1.1. – *Architecture SIMD*

général à des fonctions de librairie de traitement du signal. Les opérandes d'entrées seront appelées par la suite données d'entrée. Les résultats seront appelées données de sortie.

1.3 Architecture des machines

Dans le cadre de cette étude, les machines cibles sont des multi-processeurs SIMD ou SPMD à mémoire distribuée. Chaque processeur dispose d'une mémoire locale. Les communications entre la mémoire et les différents processeurs s'effectuent grâce à un réseau d'interconnections à haut débit. Les architectures parallèles ciblées pour le traitements de signal sont de type Multiple Instruction Multiple Data (MIMD) exploitées en mode SPMD ou Single-Instruction Multiple Data (SIMD). L'architecture SIMD exécute un flot unique d'instructions élémentaires, tandis que l'architecture MIMD est capable d'exécuter plusieurs flots d'instructions simultanément.

En traitement du signal, il existe aussi des machines hybrides combinant par exemple des cartes MIMD et SIMD. Usuellement appelées "machines à coudre" par les programmeur de traitement du signal, les hautes performances sur ces machines sont atteintes en plaçant statiquement et manuellement les applications (cf. chapitre 2, § 4).

1.4 Les contraintes de placement

Différentes contraintes applicatives conditionnent les choix de placement, ce sont les dimensions de la machine cible et les contraintes temps réel. Un placement valide doit vérifier l'ensemble de ces contraintes. Ces contraintes sont présentées de façon détaillée (cf. chapitre 2, § 2). Elles sont introduites ici brièvement pour préciser le contexte de la thèse par rapport à l'état de l'art.

1.4.1 Architectures cibles

Dans le cadre de cette thèse, nous nous restreignons aux modèles de programmation SIMD et SPMD. A un instant donné, les macro-instructions exécutées en parallèle sur chacun des processeurs sont identiques. Les données nécessaires au traitement de la macro-instruction doivent résider dans la mémoire locale du processeur qui l'exécute.

1.4.2 Le signal

Le temps réel se traduit par des contraintes d'acquisition du signal. La principale contrainte d'acquisition est la récurrence d'entrée, elle donne le temps écoulé entre deux acquisitions. La latence de l'application correspond à un temps de réponse : c'est le temps écoulé entre l'acquisition d'une donnée et le dernier résultat auquel elle a contribué.

1.4.3 Dimension de l'architecture cible

Pour les applications de traitement du signal, les dimensions de l'architecture cible sont des contraintes impératives de placement. Toutefois, elles ne sont pas prises en compte par les méthodes classiques de placement automatique et ne sont donc que partiellement traitées dans l'état de l'art actuel. Les points suivants définissent le niveau de représentation caractérisant l'architecture cible :

Nombre de processeurs : Le nombre de processeurs disponibles est considéré. Ceux-ci sont de puissance égale.

Puissance d'un processeur : Pour les applications temps réel de traitement du signal, la latence des calculs est très importante. Elle peut être bornée par une valeur maximale. Son calcul dépend de la puissance du processeur (nombre de cycles de calculs par seconde).

Mémoire : La mémoire est répartie uniformément sur l'ensemble des processeurs.

Communications : La bande passante donne le nombre de cycles machine nécessaires à la communication d'un paquet de données entre une paire de processeurs. Par la suite, la durée des communications n'est pas fonction de la topologie.

Nous précisons dans un contexte de traitement du signal ces paramètres dans les chapitre correspondants (cf. chapitre 2, § 2.1, et chapitre 9).

2 Analyse des dépendances

Une dépendance est une relation associée à un couple d'instructions x et y. On peut associer plusieurs types de dépendances à un tel couple. Il est en général impossible de construire ce type de relation en extension (liste exhaustive de tous les couples d'instances d'instructions dépendantes) pour des programmes exécutant des centaines de milliers d'opérations. Des abstractions représentant des approximations de ces ensembles sont donc plus couramment utilisées. Les dépendances permettent de restructurer et de générer par des transformations légales un code parallèle. Une transformation est dite légale si elle conserve l'ordre partiel entre les opérations induit par les dépendances du code séquentiel.

2.1 Définition des dépendances

Afin d'exprimer ces relations, définissons l'ensemble $R^w(z)$ des variables modifiées par l'instruction z et l'ensemble $R^r(z)$ des variables référencées en lecture par cette instruction.

Il n'existe pas de dépendance entre 2 instructions si les conditions de Bernstein [Ber66] sont vérifiées :

- L'instruction y ne lit aucune variable modifiée par x : $R^w(x) \cap R^r(y) = \emptyset$
- L'instruction x ne lit aucune variable modifiée par y : $R^r(x) \cap R^w(y) = \emptyset$

<table>
<tr>
<td>

```
B[0]=0
DO I=1,10
  DO J=0,9
c inst 1
    B[I]=B[I]+ ABS(A[I,J]-A[I-1,J])
  ENDDO
c inst 2
  B[I]=(B[I-1]+B[I])/2
ENDDO
```

</td>
<td>

```
DO I=1,10
  DO J=1,10
c inst 1
    T[I,J]=T[I,J]+T[I-1,J-1]
          +T[I-1,J] +T[I,J-1]
  ENDDO
ENDDO
```

</td>
</tr>
<tr>
<td>Programme Norme :
nid de boucles mal structuré</td>
<td>Programme Relax :
nid de boucles parfait</td>
</tr>
</table>

FIG. 1.2. – *Comparaison entre deux nids de boucles*

- Les instructions x et y ne modifient pas les même variables : $R^w(x) \cap R^w(y) = \emptyset$

En utilisant les même critères, les dépendances sont classifiées en plusieurs types :

Définition 2.1 (Définition des dépendances) *Supposons que l'instruction x soit exécutée avant l'instruction y.*

Une dépendance de flot plus connue sous le nom de flow dependence *est définie par*

$R^w(x) \cap R^r(y) \neq \emptyset$ *L'instruction y lit au moins une variable écrite par x*

Une dépendance en écriture plus connue sous le nom de output dependence *est définie par*

$R^w(x) \cap R^w(y) \neq \emptyset$ *L'instruction y écrit au moins une variable écrite par x*

Une dépendance en lecture plus connue sous le nom de input dependence *est définie par*

$R^r(x) \cap R^r(y) \neq \emptyset$ *L'instruction y lit au moins une variable lue aussi par x*

Une anti- dependence *est définie par*

$R^r(x) \cap R^w(y) \neq \emptyset$ *L'instruction x lit au moins une variable modifiée par y*

2.2 Différentes représentations des dépendances

Cette section donne les représentations usuelles des dépendances associées à un programme composé de nids de boucles. Chaque boucle itère selon un indice ou itérateur sur un domaine entier défini par les bornes de boucles. Les accès aux tableaux sont généralement des fonctions affines des indices de boucles. On distingue les nids de boucles parfaits des nids de boucles mal imbriqués (fig. 1.2).

Le vecteur d'itérations associé à l'instruction `inst 1` du programme `Norme` est (i, j), avec $0 < i \leq 10,\ 0 < j \leq 10$. Les dépendances sont traditionnellement exprimées par rapport aux itérateurs communs à une paire de macro-instructions.

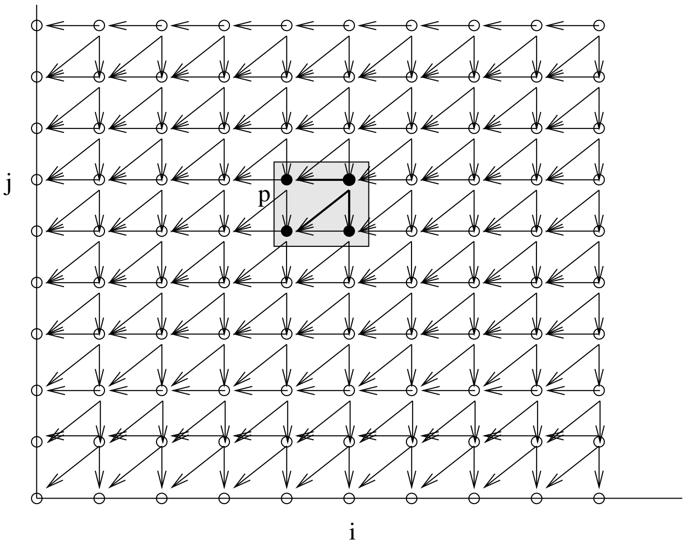


FIG. 1.3. – *Dépendances pour une itération* (i, j) *donnée du programme* `Relax`

2.2.1 Dépendances entre itérateurs

La représentation en extension des dépendances est l'abstraction la plus simple mais la plus coûteuse en espace mémoire. Elle consiste à énumérer exhaustivenent chaque couple d'itérations dépendantes. Pour le programme `Norme` et les opérations correspondant à l'instruction `inst 2` l'ensemble des couples d'itérations flot dépendantes est :

$$\{(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9),(9,10)\}$$

On dénombre exactement 81 couples de dépendances pour le programme `Relax`. On se rend compte ainsi que pour des programmes importants, il est impossible de manipuler une telle représentation. Des représentations plus compactes, symboliques, ont donc été proposées. Le problème du choix d'une représentation, ou d'une modélisation réaliste en terme de calculabilité pour le problème de placement fait partie des priorités.

2.2.2 Les vecteurs de distance de dépendance

Les vecteurs de distance de dépendance représentent une abstraction approchée des itérations de dépendances. Il est possible d'abstraire l'ensemble des couples d'itérations dépendantes par un ou plusieurs "vecteurs de distance" (DV)[2]. Ces vecteurs expriment l'ensemble des itérations dépendantes par rapport à une itération donnée.

$$D(\mathbf{i}_1, \mathbf{i}_2) \rightarrow \exists \mathbf{d} \in DV \;\;|\;\; \mathbf{i}_1 = \mathbf{i}_2 + \mathbf{d} \tag{1.1}$$

où D est la relation de dépendance, et DV l'ensemble des vecteurs de distance de dépendance. Lorsqu'un nid de boucles présente un seul vecteur de distance de dépendance (i.e. celui-ci est con-

[2]Par abus de notation, nous désignerons par DV l'ensemble des vecteurs de dépendances ainsi qu'un vecteur de dépendances (DV : Dependance Vector)

stant), le nid de boucles est dit *uniforme*. Plusieurs méthodes de placement ont proposé des solutions automatiques pour des programmes dont les dépendances sont uniformes [DR94b]. Voici quelques exemples de DV de calculs pour des dépendances lecture/écriture :

- L'ensemble des DV correspondant au programme `Relax` est :

  $$\{(0 \quad -1), (-1 \quad 0), (-1 \quad -1)\}$$

  Ils sont représentés en gras sur la figure (1.3).
- Tandis que l'ensemble des DV correspondant au programme `Norme` est
  - pour `inst 1` : $\varnothing$
    Il n'y a pas de dépendances lecture-écriture sur cette instruction.
  - pour `inst 2` : $\{(-1)\}$
    Ce qui signifie qu'il faut avoir exécuter l'itération $i-1$ impérativement avant l'itération i.
- Selon la définition des dépendances en lecture, on peut par exemple calculer les (DV) pour le programme `Norme`:
  - `inst 1 - Tableau ABS`, $\{(0 \quad -1)\}$
  - `inst 2 - Tableau B`, $\{(-1)\}$

  Dans les deux cas, ceci signifie que l'on réutilise la donnée utilisée à l'itération précédente.

2.2.3 Les vecteurs de direction de dépendances

Une approximation [Wol82] très répandue des DV est le vecteur de direction de dépendances (DDV). Chaque composante est du type $<, >, =$. Elle est calculée à partir du signe des vecteurs de distance de dépendances. Pour le programme `Norme` et l'instruction `inst2`, l'unique composante du DDV est $<$.

2.2.4 Les profondeurs de dépendances

Pour un nid de boucles donné, cette représentation donne la profondeur de la première boucle qui porte des dépendances [AK87]. Selon cette définition, les deux nids boucles `Relax` et `Norme` possèdent une profondeur de dépendance à 1.

2.2.5 Le polyèdre de dépendances

Le polyèdre de dépendances est une forme approchée des DV. Pour le programme `Relax`, l'ensemble des DV peut s'exprimer par le système :

$$\begin{cases} d_i = -k & 0 \leq k \leq 1 \\ d_j = -k' & 0 \leq k' \leq 1 \end{cases}$$

Dans certains cas, il faut calculer l'enveloppe convexe des dépendances. Le polyèdre de dépendances (DP) est alors bien approprié. Ce polyèdre est généralement représenté par son système générateur [YAI94]. La relation de dépendance peut toujours être déterminée par rapport à l'équation (1.1), en remplaçant les DV par une combinaison convexe des vecteurs générateurs de DP :

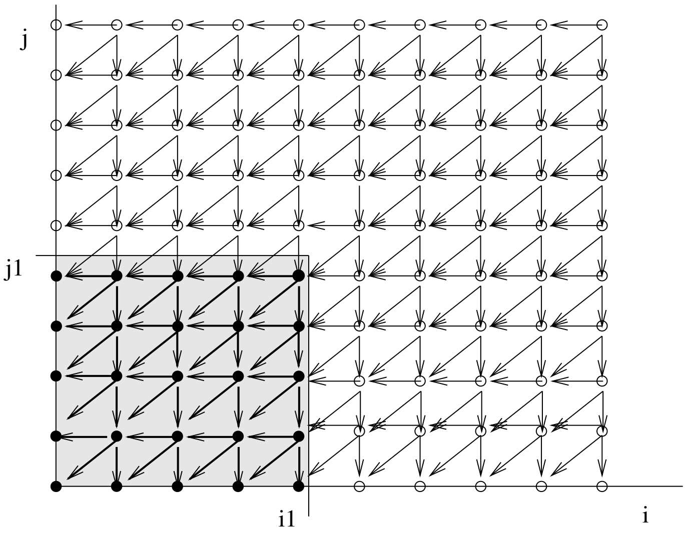


FIG. 1.4. – *Cône de dépendances sur-approximant les dépendances en lecture-écriture pour une itération (i,j) du programme* `Relax`

$$D(\mathbf{i}_1, \mathbf{i}_2) \to \exists \mathbf{d} \in \mathbb{Z}^n, \mathbf{v} = \sum_1^k \lambda_i.\mathbf{d}_i,\ \lambda_i \geq 0,\ \sum_1^k \lambda_i = 1,\ \mathbf{d}_i \in DV \tag{1.2}$$

Cette représentation introduit généralement de nouveaux vecteurs par rapport à la représentation sous forme de DV. Sur la figure (1.3), le polyèdre de dépendances lecture-écriture du programme `Relax` est représenté par le carré.

2.2.6 Le cône de dépendances

L'approximation des dépendances représentée par le cône de dépendances [IT87] relaxe les contraintes exprimant l'appartenance au polyèdre de dépendances à une combinaison linéaire positive :

$$D(\mathbf{i}_1, \mathbf{i}_2) \to \exists \mathbf{d} \in \mathbb{Z}^n, \mathbf{v} = \sum_1^k \lambda_i.\mathbf{d}_i,\ \lambda_i \geq 0,\ \sum_1^k \lambda_i \geq 1,\ \mathbf{d}_i \in DV \tag{1.3}$$

Le cône de dépendance (DC) est représenté par les droites de la figure (1.4). L'ensemble des dépendances relatives à l'itération (i_1, j_1) est représenté en gras.

Cette dernière approximation ajoute encore des arcs entre itérations dépendantes qui n'entravent pas la parallélisation [Iri87]. Les approximations, sous forme de polyèdres et de cônes de dépendances, sont utiles lorsque le nombre de vecteurs de distance de dépendances est grand. L'avantage est d'avoir une seule structure à manipuler pour représenter les dépendances. Cependant, la manipulation des systèmes générateurs nécessite l'utilisation d'algorithmes coûteux comme le Simplexe ou Fourier-Motzkin.

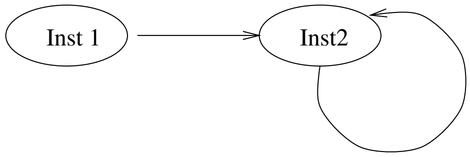


FIG. 1.5. – *DFG du programmes* `Norme`

2.2.7 Les graphes de flots de données

Les graphes de flots de données ou *Data-Flow Graph* (DFG) sont des graphes orientés dont les sommets sont les opérations et dont les arcs expriment une dépendance de flot. Cette représentation des dépendances a fait l'objet de nombreuses études. Dans le cadre de cette étude, il est possible de calculer un graphe *quotient* :

- Les sommets sont les instructions.
- Un arc exprime une dépendance potentielle entre deux instructions. Il est décoré par un prédicat donnant les paires d'itérations effectivement en dépendance.

Les arcs du DFG sont annotés par les dépendances. P. Feautrier calcule pour l'ensemble des dépendances leur *source* [3][Fea91]. Le DFG de `Norme` contient un seul arc entre les sommets `inst 1` et `inst 2`, et un arc bouclant sur `inst 2` (fig. 1.5), car on réutilise une donnée calculée à l'itération précédente pour cette instruction.

Les sommets du graphe sont :

Instrutions	Indices de boucles	domaines des boucles
inst 1	i,j	$1 \leq i \leq 10,\ 0 \leq j \leq 9$
inst 2	i	$1 \leq i \leq 10$

Les arêtes du graphe sont :

Source	Destination
inst 1, i	inst 2, i
inst 2, i-1	inst 2, i

A partir du DFG, plusieurs informations peuvent être associées aux arêtes. On en déduit des équations et inéquations linéaires affines [Fea94a] permettant de caractériser les dépendances.

2.2.8 Les dépendances intra-procédurales

Dans le cas où l'on souhaite distribuer l'exécution de procédures ou macro-instructions sur une machine parallèle, et non plus des opérations élémentaires, les dépendances présentées peuvent apparaître complexes à cause des nombreuses références intra-procédurales. Il faut alors pouvoir tester si deux données sont utilisées par la même procédure, ou si une même donnée est utilisée par deux procédures. Pour cela différentes représentations à plusieurs niveaux d'abstractions ont été définies :

[3] La source correspond à la dernière instance d'instructions ayant mise à jour la donnée.

Représentations explicites : Certaines abstractions représentent les accès aux éléments de tableau de manière exacte en marquant chaque élément référencé (ces représentations sont coûteuses, trop "fine grain"), soit de manière trop grossière (i.e. trop "coarse grain") en considérant le tableau comme un scalaire [Cal87][CK88]. Des abstractions représentant une approximation ont alors été introduites. On associe aux données le fait qu'elles sont *peut être* accédées (accès may). Sur la figure (1.6), l'ensemble des points est $\{(1,1),\ (3,2),\ (3,4)\}$.

Représentation explicite formelle : Pour caractériser les accès aux tableaux, Burke et Cytron [BC86] proposent de représenter formellement par un ensemble d'expressions arithmétiques les références aux tableaux. Sur la figure (1.6), cela correspond aux expressions :

$$\{(X,Y),\ (X+2,Y+1),\ (X+2,Y+3)\}$$

"Bounds & Strides" : Cette représentation permet d'abstraire les données accédées par des équations linéaires indépendantes calculées à partir des dimensions de tableau et des indices de boucles[Cal87][HK90], elles sont appelées les "Regular Section Descriptor". Le niveau de granularité que fixe cette représentation est en fait celui d'HPF. La figure (1.6) présente bien l'ensemble des accès approximés par les RSD.

$$\{\ Tab[\varphi_x,\ \varphi_y]\ \mid\ \varphi_x\ =\ i,\ \ \varphi_y\ =\ 2.j,\ \ (i,\ j)\ \in\ \mathbb{Z}^2\ \}$$

où φ_x et φ_y sont les coordonnées de tableaux.

Triolet, Creusillet : Les régions de tableau

Une région de tableau permet, pour une portion de code donnée, de caractériser les accès (lecture ou écriture) aux éléments de tableau par un système d'équations et d'inéquations linéaires définissant un polyèdre convexe [Tri84][IJT91][AC94][CI96].

Les inéquations portent sur les itérateurs de boucles et paramètres de structures externes à la portion de code examinée. Analyser une portion de code de la sorte revient à éliminer récursivement les variables décrivant les accès élémentaires internes à la portion de code analysé. A cette fin, un ensemble d'opérateurs permettant de manipuler les régions de tableau sont définis comme $\cap$, $\cup$ et permettent de manipuler les régions[Cre96]. Les opérateurs comme $\cap$ conservent l'exactitude des données référencées, les régions sont alors appelées `MUST`. D'autres opérateurs considérent une sur-approximation: l'enveloppe convexe. Les régions résultantes sont sur-approximées et dénommées `MAY`, par opposition aux régions `MUST`[AC94]. Voici le système correspondant à la figure (1.6).

$$\{\ Tab[\varphi_x,\ \varphi_y]\ \mid\ \ -2.\varphi_y\ +\ 3.\varphi_x\ -\ 1\ \geq\ 0,\ \ \varphi_x\ \leq\ 0,\ \ -\varphi_x\ +\ 2.\varphi_y\ -\ 1\ \geq\ 0\ \}$$

où φ_x et φ_y sont les coordonnées de tableaux.

Le choix du niveau d'approximation de la représentation[YAI94] conditionne la granularité de parallélisation et du placement (fig. 1.6)

2.3 Utilisation des dépendances pour le placement des applications de traitement de signal

Le traitement du signal systématique est proche du calcul scientifique puisqu'il met en œuvre des traitements numériques sur des échantillons de signal présents sous la forme de tableaux n-dimensionnels.

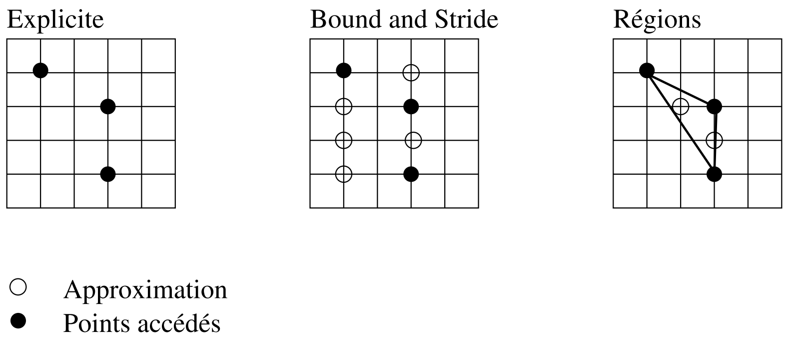


FIG. 1.6. – *Exemple de représentations*

Ces traitements enchaînent par exemple des intégrations, des décompositions temps-fréquence, des calculs de normes. Les fonctions d'accès sont généralement affines et présentent parfois des modulos. Le placement de ces applications sur des machines parallèles nécessite d'analyser différents types de dépendances.

2.3.1 Dépendances lecture-écriture intra nids de boucles - granularité des macro-instructions

Un traitement particulier du signal se compose d'un nid de boucles pas forcément bien structuré. Il peut contenir des appels de procédures et des dépendances lecture-écriture peuvent apparaître. Toutefois, celui-ci se compose généralement d'une boucle matérialisant les capteurs, une boucle caractérisant la dimension temps, et enfin une boucle représentant les fréquences, offrant plus ou moins de parallélisme selon les traitements.

Les procédures (aussi dénommées macro-instructions ou transformations élémentaires), appartenant au corps de ces boucles, sont des composants logiciels sélectionnés dans des bibliothèques du traitement du signal, pouvant faire l'objet de circuits spécialisés. On s'interdit donc la parallélisation du corps de ces procédures. Cependant, pour permettre la parallélisation manuelle des boucles englobantes, le programmeur doit de surcroît spécifier les ensembles d'éléments tableaux écrits et lues par la macro-instruction, dans un formalisme proche des "Regular Section Descriptor" (RSD) [HK90]. Ainsi intétégrées dans le corps du programme, ces macro-instructions définissent en fait la granularité des calculs parallélisables. Cette granularité pourrait bien entendu être calculée automatiquement avec des outils d'analyse automatique de code assimilant les ensembles d'éléments de tableaux à des régions de tableaux [CI96][Cre96].

Dans le cadre de la thèse **tout nid de boucles ne comporte pas de dépendances lecture-écritures**.

2.3.2 Dépendances en lecture

Les dépendances en lecture permettent de caractériser les opérations référençant en lecture un même élément de tableau. Il est, en effet, préférable de placer sur un même processeur plusieurs opérations accédant à un même élément de tableau (cf. chapitre 8-part. III), si l'on souhaite économiser la mémoire et réduire les communications. Les dépendances en lecture sont donc utiles pour guider les choix de placement (cf. chapitre 6-part. III).

Le choix d'une représentation des dépendances en lecture pose un problème délicat car les approximations telles que le polyèdre et le cône de dépendances ajoutent des dépendances fictives, ce qui tend à sur-estimer le nombre d'instructions accédant à une même donnée.

2.3.3 Anti-Dépendances

Les modélisations de la mémoire et de l'ordonnancement doivent tenir compte des anti-dépendances. En effet, soit x une opération ayant besoin de lire le contenu d'une adresse mémoire et y une opération mettant à jour cette adresse mémoire. Dans tous les cas de figure, à savoir même si x et y utilisent cette adresse pour stocker des variables différentes, y ne doit pas être exécutée avant x (cf. chapitre 8-part. III). De ce fait, les choix de séquencement des instructions ont beaucoup d'effets sur l'allocation mémoire.

2.3.4 Dépendances de flot inter-nids de boucles

Les applications de traitement du signal systématique se composent d'une succession de nids de boucles à assignation unique et de nombreux paramètres de placement tels que la taille de la mémoire, la latence, etc.. dépendent de la façon dont les macro-instructions sont ordonnancées globalement. Pour définir un ordonnancement global légal (qui ne modifie pas la sémantique du programme) de l'ensemble des macro-instructions, il est nécessaire de caractériser les dépendances de flot inter-nids de boucles. Les calculs de flot seront donc basés sur un DFG où chaque sommet se compose d'une (macro) instruction et d'un ensemble d'itérations. Au niveau de chaque arc, la représentation des opérations dépendantes est un polyèdre de dépendances (cf. chapitre 5-part. III). La combinaison d'un Graphe de Flot de Données et d'un ployèdre, ce dernier représentant des sur-ensembles d'itérations dépendantes, est originale et ne réduit pas l'ensemble des ordonnancements légaux.

2.3.5 Calcul des communications entre deux tâches placées

Les dépendances de flots inter-nids de boucles permettent de modéliser les communications nécessaires à la distribution de l'ensemble des instructions de deux nids de boucles. En particulier, les dépendances contribuent à déterminer le nombre de messages échangés par les processeurs enchaînant l'exécution de deux nids de boucles (cf. chapitre 6-part. III).

3 Transformations de boucles

Parmi l'ensemble des restructurations de code, la transformation de boucles est la plus utilisée en matière de parallélisation automatique et d'optimisation de code [Wol89, HC90]. En effet, les programmes scientifiques effectuant des traitements numériques passent beaucoup de temps dans les boucles. Les dépendances permettent de déterminer les transformations *légales* et de guider leur choix. Selon les niveaux d'abstractions définis en [YAI94], l'abstraction minimale nécessaire à la transformation est donné lorsque c'est possible.

<pre>B[0]=0
DO I=1,10
  S=B[I]
  DO J=0,9
c inst 1
    S=S+ABS(A[I,J]-A[I-1,J])
  ENDDO
  B[I]=S
ENDDO</pre>
	<pre>B[0]=0
	DO J=0,9
	  DO I=1,10
	c inst 1
	    B[I]=B[I]+ABS(A[I,J]-A[I-1,J])
	  ENDDO
	ENDDO</pre>

Début de Norme : un exemple d'inversion de boucles	Début de Norme restructuré

FIG. 1.7. – *Exemple d'échange de boucles*

3.1 Décalage de boucles

Cette transformation[4] opère un décalage sur le domaine d'itérations définit par une boucle.

3.2 Inversion de l'ordre d'exécution des itérations d'une boucle

L'inversion de l'ordre d'exécution des itérations d'une boucle se traduit par l'exécution des opérations dans le sens inverse.

3.3 Échange de boucles

L'échange de boucles est une transformation simple qui intervertit deux boucles. Cette transformation peut être intéressante dans différents cas de programme. Elle peut par exemple permettre la vectorisation[5]. Elle nécessite une représentation des dépendances simple, comme la profondeur de dépendances pour décider de la pertinence de son application. Prenons comme exemple la première partie du programme norme en figure (1.7).

Soit une machine avec un processeur et une mémoire cache, la première version du nid de boucles réutilise un même élément de tableau `A[I,J]` toutes les dix opérations `inst1`. La version restructurée utilise un élément de l'itération précédente. Dans le cas de gros tableaux, ce type d'optimisation permet d'éviter de nombreux accès en mémoire globale et *défauts de cache* en gardant les données les plus utilisées en mémoire cache.

3.4 Distribution

La distribution d'un nid de boucles mal structuré peut permettre la parallélisation de certaines instructions. Elle restructure des nids de boucles mal imbriqués en nids parfaits auxquels on peut ensuite

[4]Aussi dénommée le "Loop-Skewing".
[5]Une machine vectorielle exécute des opérations dont les opérandes sont des vecteurs

<table>
<tr>
<td>

```
B[0]=0
DO I=1,10
  DO J=0,9
c inst 1
    B[I]=B[I]+ABS(A[I,J]-A[I-1,J])
  ENDDO
ENDDO
DO I=1,10
c inst 2
  B[I]=(B[I-1]+B[I])/2
ENDDO
```

</td>
<td>

```
B[0]=0
DOALL I=1,10
  DO J=0,9
c inst 1
    B[I]=B[I]+ABS(A[I,J]-A[I-1,J])
  ENDDO
ENDDO

c Communications pour la Réduction

DOALL I=1,10
  COMMUNICATION(B[I],PROC0)
ENDDO

c Execution sur le processeur 0

DO I=1,10
c inst 2
  B[I]=(B[I-1]+B[I])/2
ENDDO
```

</td>
</tr>
<tr>
<td>Programme <code>Norme</code> distribué.</td>
<td>Programme <code>Norme</code> parallélisé.</td>
</tr>
</table>

FIG. 1.8. – *Exemple de distribution de boucles pour parallélisation*

appliquer d'autres transformations de code. Dans l'exemple (fig. 1.8, p. 33), le `DOALL` exprime les boucles parallèles.

Dans l'exemple ci-dessus, il faut générer une communication entre les deux nids de boucles distribués.

3.5 Fusion

La transformation inverse de la distribution est la fusion [KM93]. En factorisant les boucles, elle permet d'augmenter la localité des données et réduit le coût du contrôle des boucles. Dans certains cas cette transformation réduit le parallélisme d'un traitement, comme le montre l'exemple de la section précédente.

L'exemple donné en figure (1.9) illustre l'augmentation de la localité. Le tableau `B` dans la version fusionnée, ne comporte plus qu'une seule dimension. Ce type de fusion, associé à la réduction des tableaux temporaires est utilisé par les opérateurs de traitement du signal pour placer une succession de nids de boucles (cf. chapitre 2).

```
DO I=1,10
  DO J=1,5
    DO K=1,4
      c inst1
      B[I,J]=B[I,J]+A[I,J,K]
    ENDOO
  ENDDO
EMDDO

DO I=1,10
  DO J=1,5
    c inst2
    C[I]=C[I]+B[I,J]
  ENDDO
ENDDO
```

Programme Sommes distribué

```
DO I=1,10
  DO J=1,5
    DO K=1,4
      c inst1
      B[I]=B[I]+A[I,J,K]
    ENDDO
    c inst2
    C[I]=C[I]+B[I]
  ENDDO
ENDDO
```

Programme Sommes fusionné

FIG. 1.9. – *Exemple de restructuration utilisant la fusion*

3.6 Permutation

La permutation est une transformation unimodulaire [Ban89] permutant l'ordre des boucles. Elle est donc utilisée aux mêmes fins que l'échange de boucles. Pour garantir la légalité de cette transformation, l'abstraction minimale nécessaire est le DDV (vecteur de direction des dépendances).

3.7 Partitionnement

Le partitionnement d'un nid de boucles scinde les boucles de manière à regrouper les opérations en plusieurs sous-ensembles disjoints d'opérations. Dans le cas de nids de boucles, on s'intéresse à des partitionnements uniformes obtenus par pavage [Iri87][IT88b][RS90][SD90][AL93b][Dio96][HCF97] de l'espace d'itérations décrit par les nid de boucles, les traitements locaux sont alors des blocs d'opérations. Les partitionnements sont définis par des hyperplans parallèles. Les hyperplans définissent les facettes des blocs d'opérations. Le partitionnement peut aussi être construit sur un maillage de l'espace d'itération.

L'abstraction minimale du partitionnement sous forme de blocs est le cône de dépendances.

3.8 "Strip-Mining"

Le "strip-mining" est un cas de partitionnement découpant le domaine d'itérations en bandes.

Le nid de boucles (fig. 1.10) est partitionné selon deux nouveaux itérateurs `I1` et `I2` définis comme $I = 2 * I1 + I2 + 1$. Les références aux tableaux `A` et `B` sont donc modifiées selon ce "strip-mining". La boucle `I1` peut par exemple représenter le numéro du processeur sur lequel sera effectué un bloc

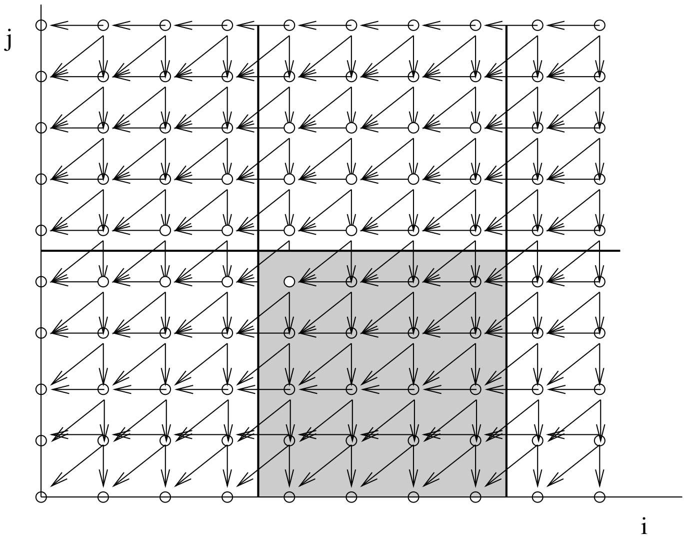


FIG. 1.10. – *Dépendances et partitionnement du nid de boucles* `Relax`

<table>
<tr>
<td>
<pre>
DO I=1,10
  DO J=1,5
    c inst1
    B[I,J]=B[I,J]+B[I,J-1]*A[I,J]
  EMDDO
EMDDO
</pre>
</td>
<td>
<pre>
DOALL I1=0,4
  DO I2=0,1
    DO J=1,5
      c inst1
      B[2*I1+I2+1,J]=B[2*I1+I2+1,J] +
          B[2*I1+I2+1,J-1]*A[2*I1+I2+1,J]
    EMDDO
  EMDDO
EMDDO
</pre>
</td>
</tr>
<tr>
<td>Programme <code>SommesDep</code></td>
<td>Programme <code>SommesDep</code> partitionné</td>
</tr>
</table>

FIG. 1.11. – *"strip-mining" d'un nid de boucles*

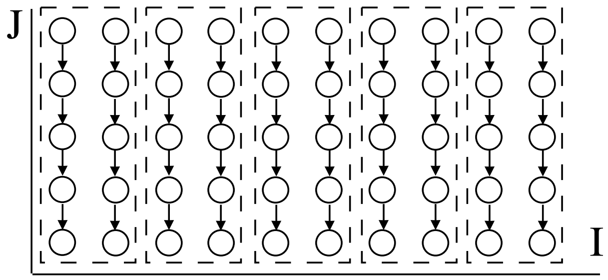


FIG. 1.12. – *Dépendances et "strip-mining" du nid de boucles* `SommesDep`

d'opérations locales (représenté par les itérateurs `I2`,`J`,`K` et l'instruction `inst1`). Comme la boucle `I1` partitionnée ne comporte pas de dépendances, le traitement est placé sur 5 processeurs.

3.9 Transformations unimodulaires

Plusieurs techniques de transformations de boucles ont été présentées pour optimiser localement des nids de boucles. L'utilisation d'une matrice unimodulaire permet d'unifier certaines transformations de boucles élémentaires [Ban90, AI91]. Les transformations unimodulaires pouvant être représentées par ce même formalisme, sont entres autres: l'inversion, l'échange, le décalage, la permutation.

3.10 Parallélisation des boucles

Diverses techniques et méthodes ont été utilisées pour paralléliser automatiquement un nid de boucles séquentiel. Nous en introduisons quelques unes dans cette section. Ces approches reposent sur les transformations de codes et représentations de dépendances introduites ci-dessus.

- La méthode de Kennedy & Allen [AK87] combine le réordonnancement d'instructions et la distribution de boucles. Cette méthode est utilisée dans de nombreux compilateurs et utilise à la fois un graphe de dépendance et les DDV.
- La méthode hyperplane [Lam74] décompose l'espace d'itérations par une famille d'hyperplans parallèles. Les itérations d'un même hyperplan sont exécutées en parallèles. Les hyperplans sont exécutés séquentiellement. Cette méthode utilise la permutation, l'inversion et le décalage de boucles.
- Le *cycle shrinking* utilise le strip-mining et l'échange de boucles.

3.11 Utilisation des transformations de boucles pour le placement des applications de traitement du signal

Dans le contexte de la thèse, l'ensemble du contrôle d'une application de traitements de signal placée doit être statique. Cela signifie que tous les aspects de l'implantion sont définis statiquement : le contrôle du code des instructions, l'allocation des données dans la mémoire et les communications entre processeurs. Le problème, rendu complexe par le nombre de contraintes que le placement doit satisfaire, était résolu auparavant manuellement. Une application placée se traduit par une combinaison de transformations de code. Les transformations sont utilisées tant pour l'optimisation d'une ressource locale (comme la mémoire cache), que pour exprimer un ordonnancement général des macro-instructions. Toutefois, l'ensemble des choix de transformations et de paramétrages augmente rapidement et le contrôle de l'explosion combinatoire n'est pas réaliste.

Un exemple de placement d'applications illustrant ces transformations manuelles est donné (cf. chapitre 2, § 4.3).

3.11.1 Partitionnement

Le partitionnement permet d'adapter les domaines d'itérations à la taille de la machine (cf. chapitre 4 - part. III) en exprimant :

- La granularité du parallélisme des nids de boucles,
- Le parallélisme du nid de boucle, afin que celui-ci puisse être exécuté sur une machine dont le nombre de processeur est contraint.
- L'ensemble des blocs d'opérations séquentielles dans un bloc, associé par exemple à une ressource de l'architecture cible. Cet ensemble permet de définir par exemple l'ensemble des données requises pour l'exécution du bloc.
- Les blocs d'opérations doivent être de la bonne taille par rapport à l'utilisation de la ressource dans le temps. Pour reprendre l'exemple de la mémoire, les blocs doivent être sérialisés dans le temps si la mémoire requise est trop importante.

3.11.2 Fusion

La spécification d'une application de traitement du signal correspond à une succession de nids de boucles distribués. La fusion est utilisée pour augmenter la localité des données afin de réduire l'allocation.

3.11.3 Permutation

Les boucles induisant des dépendances lecture/lecture doivent être placées le plus à l'intérieur possible du nid afin, par exemple, de maximiser la réutilisation des données au plus tôt (cf. chapitres 7-part. III et 8-part. III).

4 Placement automatique

Si la phase de parallélisation consiste à extraire le parallélisme potentiel d'un programme séquentiel, le placement automatique, tel qu'il est abordé dans cette thèse, consiste à trouver l'adéquation entre un programme, dont le parallélisme est spécifié, et une architecture cible. Les paramètres considérés sont généralement, le nombre de processeurs, la mémoire, la bande passante et le temps d'accès au réseau d'interconnection. Le critère d'optimisation est la performance du programme placé. Une première approche consiste à exprimer directement le parallélisme. Le placement est alors basé sur la distribution des calculs sur l'architecture, des heuristiques permettent de retenir les solutions de placement qui favorisent la localité. Une deuxième approche consiste à éviter les communications. Dans ce cas, c'est la distribution des données sur les processeurs qui exprime le placement.

4.1 Maximiser le parallélisme et la localité

Le partitionnement de nids de boucles représente une solution aux problèmes de placement puisqu'il permet de représenter le parallélisme d'un nid de boucles en fonction du nombre de ressources (processeurs, mémoires) disponibles d'une part et de définir l'ensemble des opérations devant être effectuées localement d'autre part [Lam74, IT88b, AL93b]. Un bloc d'instructions peut être alors associé à une ressource de la machine en fonction du nombre de processeurs. Les différents ensembles d'opérations doivent par la suite être ordonnancés de façon à restructurer légalement le code selon les dépendances lecture-écriture [LL97]. Parmi les solutions de partitionnement, on privilégie celles garantissant une certaine localité que ce soit pour optimiser les communications ou la réutilisation des données dans une mémoire locale. Favoriser la localité est alors une heuristique permettant de trouver les partitionnements pertinents. Par exemple, lorsque deux opérations référencent une même donnée, on favorise le partitionnement pour lequel ces opérations sont exécutées sur le même processeur. De même si deux données sont référencées par une même opération, elles doivent être associées au même processeur. En suivant ce type de choix, le placement peut rapidement converger vers un seul processeur ! Des compromis doivent donc être effectués [Dio96][Pla95] (fig. 1.10).

Lorsque deux instructions appartiennent à deux nids de boucles différents et référencent un même ensemble de données, elles doivent être placées de préférence sur le même processeur afin d'éviter les communications. L'opération de fusion permet alors de regrouper ces deux instructions qui seront affectées à une ressource identique. Pour résoudre le problème K. Kennedy et K.S.McKinley [KM93] proposent un algorithme de flot maximal/coupe minimale sur le graphe de contrôle du programme : chaque nœud de ce graphe se compose d'une seule boucle, une arête détermine si deux boucles peuvent être fusionnées.

4.2 Minimiser les communications

Exprimer l'agencement des tableaux sur la machine cible permet d'évaluer les communications relatives au placement. L'agencement des tableaux est caractérisé par la relation d'Alignement. Elle se définit entre dimensions de tableaux et permet d'exprimer la position des données d'un tableau par rapport à celles d'un autre. Les différents alignements induisent alors les mouvements de données. Lorsque l'ensemble des relations d'alignement sont données, la position de l'ensemble des données peut être exprimée dans un même repère virtuel, le “Template” qui doit alors être distribué sur une machine cible. Cette notion fort utile de “Template” est un concept rendu général dans le langage `High Performance Fortran` (§ 4.3).

- Une méthode d'alignement a été introduite par Li & Chen [LC90] avec le “Component Affinity Graph” (CAG). Chaque nœud de ce graphe représente une dimension de tableau. Une arête représente une relation potentielle d'alignement associée à un coût. Une partition de ce graphe fixe les alignements et deux dimensions de tableaux sont alignées si elles sont dans la même partition. Minimiser les communications revient alors à trouver une partition du graphe (fig. 1.13) pour laquelle la somme des coûts associés aux arêtes reliant les partitions est minimale. Le problème se révèle alors NP-complet. Cette technique est utilisée dans le cadre de la compilation du langage à assignation unique *Crystal* en utilisant des heuristiques.
- Cette approche a été étendue par la suite en [GB92] en ajoutant des contraintes (qui sont en fait des heuristiques qu'il faut essayer de satisfaire) portant sur l'équilibrage de charge et les communications.

```
DO I=0,19
  DO J=0,9
    DO K=0,29
c inst 1
      B[I,J,K]=A[I+K,J]*C[I,K+J]
    ENDDO
  ENDDO
ENDDO
```

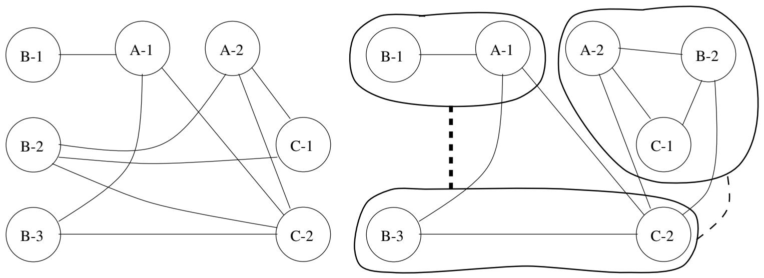


FIG. 1.13. – *Exemple de CAG et de Partition permettant de définir les alignements*

- L'approche de Knobe, Lucas et Steele [KLS90] propose un graphe dont les sommets sont les références aux tableaux et les arêtes des *préférences* d'alignements. On distingue alors deux types de préférences : la *préférence d'identité* traitant deux références au même tableau; la *préférence de conformité* traitant une donnée référencée par deux instructions. Le graphe traduit le mouvement des données engendré par une solution d'alignements.
- La méthode polyédrique de Feautrier calcule, à partir du "Array Data Flow Graph" (DFG) une fonction de placement visant à minimiser les communications [Pla95]. L'alignement des dimensions est exprimé à partir du DFG par une série d'équations linéaires [Fea94b]. Une fonction de placement sur un repère virtuel commun à l'ensemble des tableaux est calculée (par élimination de Gauss). Pour être légale, la restructuration du nid de boucles peut nécessiter un réordonnancement des itérations dans le temps, qui définie une base de temps[Fea92b, Fea92a]. Enfin, Feautrier présente le lien entre les contraintes non-linéaires et le placement avec des ressources finies. La résolution de contraintes non-linéaires ne peut être résolue dans le cadre de la programmation linéaire en nombres entiers (PLNE) [Fea94a].
- Anderson et Lam [AL93a] proposent un algorithme calculant des *décompositions*, ce sont des fonctions affines exprimant le placement des tableaux et calculs sur un "Template" de processeurs virtuels. Cette approche cherche à concilier le partitionnement des données et calculs avec l'alignement des dimensions de tableaux pour éviter statiquement les communications.
- Les travaux de Darte et Robert [Dar93, DR94b] consistent à contruire un *graphe de communications* où un nœud représente une donnée, un arc représente une communication. Le problème s'avère là encore NP-Complet même avec un vecteur de distance uniforme associé aux communications.
- L'expression sous forme d'un problème de programmation en 0-1 de l'alignement et la distribution des données afin de minimiser les communications a été introduite par Bixby, Kennedy et

Kremer [BKK94, Kre95]. Leur problématique consiste à générer automatiquement les directives d'alignement et de distribution du langage HPF. Leur approche revient à exprimer les resynchronisations et à considérer une représentation similaire à celle du CAG [KK93] de Li & Chen pour l'alignement :

- L'ensemble des resynchronisations et des alignements possibles sont décrits par des ensembles de variables $0-1$.
- Une fonction de coût que l'on pourrait appeler *latence* permet de lier les deux ensembles de contraintes sur les différentes variables.

L'utilisation d'un solveur générique basé sur le Simplex [BKK94] permet l'optimisation du placement en respectant les contraintes d'alignement.

4.3 Programmation parallèle

Devant la complexité des méthodes de placement automatique, différents travaux laissent la responsabilité de la spécification de l'alignement et de la distribution au programmeur. Nous citerons les primitives `ALIGN` et `DISTRIBUTE` du langage HPF [KLS+94] (pour High Performance Fortran) permettant respectivement d'exprimer l'alignement des tableaux dans un repère commun, le `TEMPLATE`, et de distribuer celui-ci sur une grille de processeurs virtuels. Dans le même cadre, la primitive `FORALL`[6] a été proposée pour spécifier les boucles parallèles, la primitive `INDEPENDENT` permet d'étendre le langage au parallélisme plus général.

Le compilateur doit alors traduire cette spécification de placement sous forme de contraintes linéaires, permettant d'allouer les données, d'engendrer les communications, et de synchroniser les traitements sur une architecture physique [ACIK93, Cha95].

5 Conclusion

Les problèmes de placement comme de parallélisation nécessitent de formuler des représentations en **intension** plutôt qu'en **extension**. La formulation du problème génère donc soit beaucoup de contraintes, soit des contraintes complexes. L'algèbre linéaire permet de formuler les différentes étapes de parallélisation ou de placement comme l'analyse des dépendances ou la distribution des calculs et des données. Ces formulations sont avantageuses car elles ne nécessitent pas de représenter explicitement chaque dépendance, itération ou élément de tableau. Quand à la résolution, elle donne lieu à une profusion d'algorithmes spécialisés selon la fonction à traiter, utilisant certains résultats de Recherche Opérationnelle, de la Programmation Linéaire et des heuristiques de domaines (recouvrement des communications par les calculs par exemple). La parallélisation et le placement se soldent alors par une succesion d'algorithmes déterministes appliqués à chacune des étape du placement. Or, si l'on considère le placement dans la globalité, chaque fonctionnalité (ordonnancement, distribution des calculs, allocation mémoire, complexité des communications) est par nature non déterministe. Différents travaux soulignent d'ailleurs l'insuffisance et la difficulté de l'approche algorithme pour résoudre le placement global avec contraintes de ressources [Dar93][Fea94a][BKK94][Dio96]. Ce problème est abordé manuellement par les programmeurs de traitement du signal, le chapitre suivant illustre l'approche utilisée.

[6] Cette primitive exprime les boucles parallèles.

Chapitre 2

Placement des applications de traitement du signal

Introduction

D'un point de vue physique, on distingue deux grandes familles de traitements du signal. D'une part, le traitement hyperfréquence analyse la réception continue de photons sur une antenne. D'autre part, le traitement en milieu accoustique de nature atmosphérique ou sous-marine est caractérisé par la propagation de molécules de ce milieu. Dans un contexte opérationnel, ces traitements sont utilisés pour la détection, ou la poursuite de cibles. Parmi ces systèmes, certains sont voués à rester au sol et d'autres sont destinés à être embarqués. Le choix et l'implantation des traitements sont conditionnés par ces critères : d'un côté, l'embarcation des systèmes qui contraint l'utilisation de certaines ressources de calculs, de l'autre la célérité du signal dans un milieu spécifique qui impose des contraintes temps réel plus ou moins fortes.

Célérité des signaux : Pour fixer des ordres de grandeur, la célérité du signal en milieu atmosphérique est de l'ordre de 300 m/s, en milieu sous-marin de l'ordre de 1400 m/s, tandis qu'en hyperfréquence il faut considérer la vitesse de la lumière.

Complexité du milieu : Le milieu marin offre un cadre complexe et hétérogène de propagation du signal : la variation de la salinité de l'eau et de sa température[1] combinée à de nombreux échos ou des amortissements locaux du signal (échos de surface, rivages, organismes marins, bulles de gaz ...)[Gaz89b]. Ces problèmes n'apparaissent pas en traitement radar[Far92].

Type de cibles : Si le domaine hyperfréquence est sujet à la détection de cible métallique, le milieu maritime offre un nombre de cibles variées[2] dont les propriétés accoustiques sont souvent méconnues[Gaz89b].

Aussi, les traitements doivent tenir compte de ces spécificités[3]. Si l'on dispose de plus de temps pour traiter l'information en milieu sous-marin, les traitements peuvent apparaître plus complexes.

[1] La temperature varie selon la profondeur et peut être sujette à de nombreuses fluctuations dues aux mélanges de courant chauds/froids.

[2] Fonds marins, bancs de poissons, plongeurs, sous-marins, tête de puits, pipe-line, etc...

[3] On pourrait ajouter la notion de traitements passifs et de traitements actifs. Dans la première situation, on cherche à capter un signal originalement non identifié. Les algorithmes utilisés tenteront alors d'identifier des régularités dans le signal. Dans la deuxième situation, on connait d'avance la fréquence d'émission.

Néanmoins, on peut dégager de nombreuses caractéristiques communes à ces traitements.

La nature des applications de traitement du signal est introduite dans ce chapitre. L'ensemble des paramètres applicatifs issus de la fonction du traitement du signal est ensuite présenté. Alors que ces applications sont caractérisées par un modèle de programmation, l'usage des transformations de code et des critères de décision est exposé. Enfin, les différentes approches de compilation/placement propres au traitement de signal (TS) sont abordées.

1 Nature du traitement du signal digital numérique

De manière générale, le traitement du signal utilise une grande partie de l'algorithmie qui peut aussi bien être numérique que symbolique [aG75][Gaz89b][Gaz89a][Far92][Bou]. Nous considérons, dans cette thèse, une classe algorithmique définie par un modèle de programmation dédié au **traitement du signal systématique** (TSS). Cette classe algorithmique ne permet pas de définir des bornes de boucles à partir de variables et n'autorise pas les instructions conditionnelles globales, ni les indirections dans les tableaux. Cependant, une application intègre différentes fonctions de traitement du signal comme les décompositions temps-fréquence, intégrations, filtres, ou calcul de normes issues de bibliothèques. Ils sont utilisés dans un contexte opérationnel où l'implantation est rendu délicate par la prise en compte du temps réel.

1.1 Traitements multidimensionnels

Les activités de détection nécessitent l'utilisation d'un réseau de capteurs exploités pour mesurer les déphasages entre les signaux reçus. Ce réseau de capteurs, aussi appelé antenne peut-être organisé spacialement pour favoriser les calculs de détection ou de poursuite [Mai94]. Par exemple, sur une antenne 2D, l'axe vertical et l'axe horizontal sont exploités différemment. Chaque capteur échantillone un signal, c'est-à-dire mesure une grandeur physique quelconque dans le temps. Dans les systèmes de détection, la structure de données associée à l'antenne est un tableau n-D (fig. 2.1).

Cependant, il existe une autre représentation où le signal se décompose, sur une portion de temps en amplitudes, phases et fréquences. Cette décomposition temps/fréquence comporte donc deux spectres : le spectre des phases et le spectre des amplitudes[Gaz89a]. Ils permettent de calculer respectivement les déphasages et l'énergie entre signaux. Dans un domaine de calculs discrets, le traitement généralement utilisé à cette fin est la Transformé de Fourier. Une implantation bien connue est la FFT (Fast Fourier Transform).

En considérant l'ensemble de ces aspects, le traitement du signal est multi-dimensionnel. Un tableau comportant les données à traiter peut se constituer de plusieurs dimensions : **l'axe temps, l'axe fréquence et le ou les axes capteurs**.

1.2 Plusieurs tâches de traitement

Pour extraire l'information pertinente du signal, une application de traitement du signal enchaîne généralement plusieurs tâches de calcul. L'ensemble de l'application peut être représentée par un graphe de traitements.

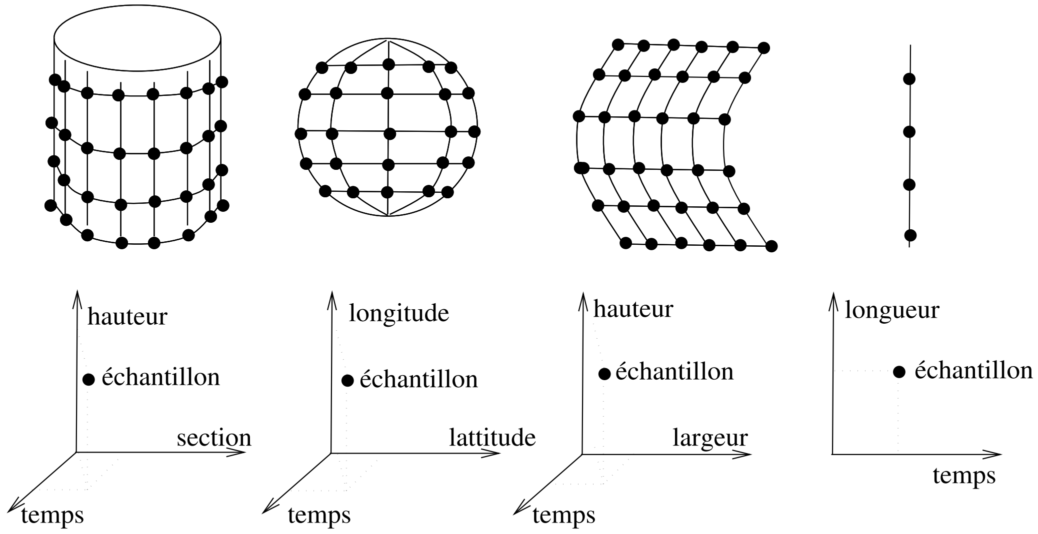


FIG. 2.1. – *Nature des capteurs et leur représentation tableaux*

1.3 Temps infini

La dimension temps est particulière car théoriquement tous les échantillons sont utiles à la production des résultats. Pratiquement, l'application doit se contenter des possibilités limitées qu'offre une machine dont les ressources sont finies. A un instant t, il est nécessaire de connaître les échantillons et résultats intermédiaires ne servant plus pour pouvoir réutiliser leur emplacement dans la mémoire. Directement lié aux dimensions du système, le nombre d'échantillons minimum requis dans le temps à la production d'un résultat doit être déterminé lors du placement de l'application. La dimension temps d'un tableau est potentiellement infinie (fig. 2.1) et correspond à une dimension de calcul aussi infinie.

1.4 Exemple de traitements

Différents outils numériques interviennent dans le cadre du traitement du signal. Les décompositions temps-fréquence les calculs de normes, les filtres sont très utilisés. D'autres traitements spécifiques au domaine sont aussi développés. Parmi ces traitements, la Formation des Voies permet de comparer les déphasages entre signaux issus de différents capteurs et permet ainsi d'obtenir l'amplitude du signal par voie (fig. 2.2. couplé avec la FFT, ce traitement permet de localiser la provenance du signal de plus grande énergie, selon une bande de fréquences donnée)[Bou][Far92]. Selon que les algorithmes sont soit de détection, soit de poursuite, la formation de voies est exécutée avant ou après la Transformée de Fourier. Une application de traitement du signal complète enchaîne donc plusieurs traitements. La nature des algorithmes mis en œuvre dépent fortement de la géométrie du problème : nombre et agencement des capteurs (cylindrique, en ligne, en tableau), fonction du système (identification d'un cap, d'une vitesse). Si la FFT représente un traitement important pour les calculateurs modernes [Tol96], la formation des voies n'est pas moins complexe à cause du l'importance du voisinage à prendre en compte à chaque instant [Far92]. Enfin composer l'ensemble de ces traitements engendre de nombreux problèmes d'implantation [Bou][Far92].

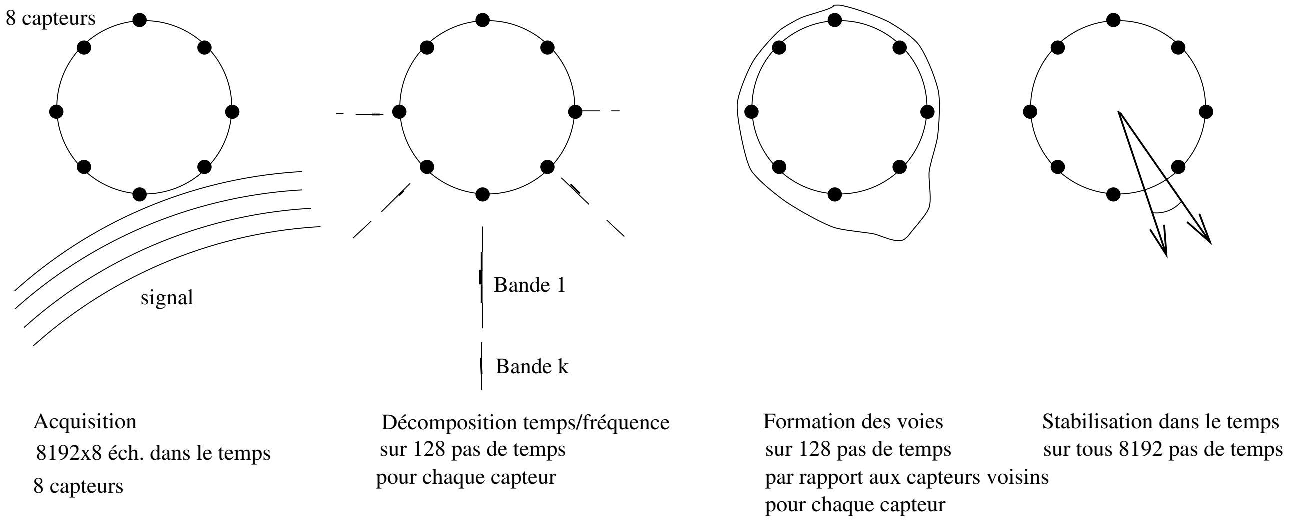


FIG. 2.2. – *Exemple de capteurs et traitements*

2 Contraintes de placement

Les exigences opérationnelles sont prises en compte sous la forme de directives à respecter lors d'un placement manuel, afin de trouver l'architecture adéquat, et sous la forme de contraintes si l'on considère le placement automatique. Ces contraintes sont sous-jacentes à la *fonction* du traitement du signal comme par exemple la détection ou/et la poursuite de cibles. Nous avons d'une part les contraintes de ressources, d'autres part les contraintes de temps réel.

2.1 Les contraintes de ressources

Les ressources caractérisent la puissance de calcul d'une architecture et disposent d'une série de paramètres que l'on peut dimensionner pour trouver l'adéquation machine/application. Le niveau de représentation des architectures a déjà été présenté dans l'état de l'art placement (cf. chapitre 1-part. III), il est ici opportun de rappeler ou compléter ces notions :

Les processeurs : Les paramètres considérés sont généralement la puissance de calcul, qui est fonction de la fréquence d'horloge. Lors des placements manuels, seul le nombre d'opérations réalisables en un temps donné est pris en compte. L'échelle est le Flops (nombre d'opérations flottantes/seconde). Le nombre de processeurs intervient aussi dans les décisions de placement. Dans différents contextes opérationnels le nombre de processeurs peut être limité, par exemple, à cause de l'encombrement de ceux-ci.

La mémoire : Elle est caractérisée par son temps d'accès et la quantité disponible par processeur. Si les temps d'accès mémoire sont pris en compte pour le placement manuel, ils seront intégrés aux temps de calcul élémentaire dans l'approche automatique développée dans cette thèse.

Les communications : Les communications dépendent fortement de l'architecture cible. Celle-ci a une grande influence sur la nature du code des communications à traiter et leur performance. Etant donné l'évolution technologique, il n'est pas déraisonnable de considérer une machine "*fully connected*"[4] où chaque processeur est capable d'émettre et de recevoir simultanément. Ces considérations sont valables pour le placement manuel. L'approche automatique de placement développée dans la thèse respecte ces hypothèses. Toutefois, des modèles d'interconnexion pourraient être ajoutés au modèle de communication dans le système de placement automatique, en représentant par des contraintes non-linéaires la complexité des algorithmes de communications [GB94][Tol96].

Dans le contexte du traitement du signal, le niveau de granularité auquel nous étudions le problème du placement, qui est en fait celui de la procédure ou de la macro-instruction, permet de manipuler l'application dans sa globalité. Dans le contexte de l'approche automatique organisée en modèles concurrents, les ressources sont uniformes (même nombre de processeurs d'une puissance de calcul égale, même taille mémoire locale, communications élémentaires identiques quelque soit la position de l'émetteur et du récepteur). Une contrainte de ressource porte sur la cardinalité d'un ensemble de ce type. Le placement automatique, à l'instar du placement manuel devra donc tenir compte des contraintes ressources en plus de la difficulté de trouver un placement correct ...

2.2 Les contraintes Temps Réel

Deux aspects du temps réel sont traités et pris en compte : la cadence des acquisitions et la restitution des données sont dénommées récurrence ou période. Le temps de réponse entre l'acquisition des données et le résultat correspondant est la latence. Ces différentes contraintes temps réel s'ajoutent au problème de placement général.

2.2.1 Les contraintes d'entrées-sorties

Au niveau d'une application complète, on distingue les contraintes d'entrées et de sorties. Elles sont motivées par des considérations matérielles (type de capteurs) ou par la fonction du système de traitement du signal. Par exemple, pour être fluide, la cadence d'un signal vidéo doit être supérieure à celle de l'œil. De surcroît, la cadence d'acquisition dépendra de la définition de l'image.

Les contraintes de sorties reposent sur les traitements produisant les résultats et modélisent la cadence de sortie de ces résultats (fig. 2.3). Cette cadence est aussi dénommée la **récurrence de sorties**.

Les contraintes d'entrées sont définies à partir des traitement d'acquisitions et modélisent la cadence d'entrée des échantillons de signal (fig. 2.3). Cette cadence est aussi appelée la **récurrence d'entrées**.

Dans les deux cas, une contrainte temps réel correspond à contraindre l'intervalle de temps physique séparant l'exécution de deux instructions. Pour chacune des boucles imbriquées, mais plus particulièrement pour la dimension infinie, une intervalle maximale de temps est donnée, dans laquelle tous les calculs du corps de boucle doivent être impérativement exécutés. Ainsi, des contraintes temps réel multi-dimensionnelles peuvent être définies. Dans le chapitre 10 (cf. partie III), nous précisons comment traiter ces contraintes au niveau d'une dimension particulière : la dimension infinie.

Si l'application dispose de plusieurs sorties ou de plusieurs entrées, comme sur la figure (2.3), de telles contraintes sont à spécifier pour chaque type d'instruction.

[4] Chaque processeur peut communiquer directement avec tout autre processeur

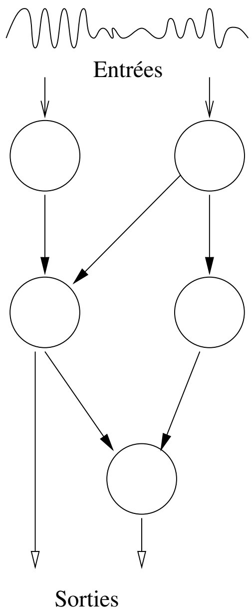


FIG. 2.3. – *Entrées-Sorties*

2.2.2 La latence de l'application

La **latence** de l'application modélise le temps de réponse de celle-ci une fois qu'elle est placée sur une architecture cible. C'est donc le critère de durée de vie de l'information. Dans le contexte du traitement du signal, la latence correspond à l'intervalle de temps physique maximal séparant deux dates : la date de production d'un résultat et la date de la plus ancienne acquisition nécessaire à l'élaboration de ce résultat. Lorsque ce critère est imposé, c'est une contrainte impérative à garantir. Il faut donc totaliser la durée de l'ensemble des opérations dépendantes intervenues entre ces deux dates et considérer l'intervalle maximal.

3 Modélisation des applications de traitement du signal

ALAIN DEMEURE *(Thomson-CSF) est à l'origine du formalisme de description d'application de traitement du signal systématique. Une synthèse de ces travaux est l'objet d'une publication [DLB+95].*

Les applications de traitement du signal sont formées de séquences de tâches, que l'on peut exprimer par des nids de boucles bien structurés et parallèles. Chaque nid de boucles contient un appel à une procédure ou macro-instruction correspondant en général à une fonction d'une librairie de traitement du signal telle qu'une FFT. Ces procédures ont un coût d'exécution fixé, inclu dans la spécification de l'application. Celle-ci est globalement représenté par un graphe de flot de données. Dans le contexte de la thèse, on ne retiendra uniquement les **graphes acycliques**, c'est une des caractéristiques applicatives données (§ 6.1, p. 67). L'application est sous forme *d'assignation unique*, c'est à dire que chaque élément de tableau n'est mis qu'une seule fois à jour par l'application.

```
DO I1=0,up(I1)
   ...
     DO In=0,up(In)
       APPEL_PROC(I1,I2,...In)
     ENDDO
   ...
ENDDO
```

où $up(I1) \in [0, \infty]$

Les indices externes sont passés en paramètres à l'appel de procédure.

FIG. 2.4. – *Nids de boucles englobant un appel de procédure*

3.1 Modèle de programmation

Plus précisement, une tâche se décompose en deux parties :

L'espace d'itérations externe décrivant le domaine de calculs. L'une des dimensions peut être infinie. Elle représente le temps. Ce domaine est représenté par un nid de boucles parfaitement imbriqué et totalement parallèle. Il n'y a aucune dépendance en écriture, par contre il peut exister des recouvrements en lecture. C'est ce domaine de calculs qu'il faut *placer* et ordonnancer sur la machine parallèle.

L'espace d'itérations interne décrivant l'ensemble des données qui sont utiles au calcul de la macro-instruction ou procédure. Les fonctions d'accès aux éléments de tableau sont des fonctions pseudo-affines. Des fonctions `modulo` sont parfois utilisées pour prendre en compte le caractère cyclique des capteurs.

3.1.1 Indices extra-procéduraux

Les espaces d'itérations des applications considérées comportent des bornes de boucles constantes. L'espace d'itérations est donc un hyper-parallépipède rectangle, défini dans $\mathbb{Z}^n$, que nous noterons $\mathcal{I}$. Seule la boucle temps peut posséder une borne infinie. Chaque itération exécute un appel de procédure (fig. 2.4) où chaque indice est un paramètre d'entrée de la procédure qu'elle englobe. Ces indices permettent de paramétrer les accès aux tableaux à l'aide de fonctions d'accès (cf. § 3.1.4, p. 48).

Définition 3.1 L'espace d'itérations *noté $\mathcal{I}$ associé à un nid de n boucles de bornes $up(i_1)$, $up(i_2)$, ... $up(i_n)$ est :*

$$\mathcal{I} = \{0, \,..,\, up(i_1) - 1\} \times \{0, \,..,\, up(i_2) - 1\} \times ... \times \{0, \,..,\, up(i_n) - 1\}$$

où $\forall\ i_k,\ \ 0 \leq i_k <\ up(i_k)$ et $up(i_k) - 1$ est la borne supérieure de la k^{me} boucle du nid.
On notera un vecteur $\mathbf{i}$ d'itérations appartenant à $\mathcal{I}$.

Ces indices expriment le parallélisme potentiel du nid de boucles. En effet, le placement ne cherchera pas à exploiter le parallélisme intra-procédural. Dans la figure (2.5), l'outil tiendra compte à la fois du parallélisme sur la dimension `temps` et/ou sur les dimensions `capteurs`, `frequences` ou autres.

```
DO TEMPS=0,INFINI
  DO CAPTEURS=0,N
    APPEL_FFT(TEMPS,CAPTEURS)
  ENDDO
ENDDO
```

FIG. 2.5. – *Appels de la procédure FFT*

3.1.2 Accès aux tableaux

Les fonctions d'accès aux éléments de tableau sont des fonctions pseudo-affines. Des fonctions `modulo` sont utilisées, par exemple, pour représenter le caractère cyclique des capteurs. Pour chaque dimension non infinie, l'application étant supposée correcte, les bornes de tableaux sont considérées comme les références maximales et sont connues statiquement.

Définition 3.2 Le tableau des données Φ *de* n *dimensions est défini par :*

$$\Phi = \{0,\ ..,\ up(\varphi_1) - 1 \times \{0,\ ..,\ up(\varphi_2) - 1\} \times ... \times \{0,\ ..,\ up(\varphi_n) - 1\}$$

où $\forall\ k,\ \ 0 \leq \varphi_k <,\ up(\varphi_k)$ *et* $up(\varphi_k) - 1$ *est la borne supérieure de la dimension* k *du tableau. On notera* $\boldsymbol{\varphi}$ *un vecteur appartenant à* Φ*.*

3.1.3 Accès intra-procéduraux

Les accès intra-procéduraux ne sont pas explicitement donnés par l'application. Les accès réels sont *cachés* au sein de la procédure. Seuls les accès décrivant les ensembles de données $\mathcal{S}$ et $\mathcal{E}$ référencés par la procédure sont explicités. En reprenant les notations issues de la parallélisation automatique, nous notons ces ensembles $\mathcal{R}^w$ et $\mathcal{R}^r$. La nécessité de représenter une sur-approximation des points accédés par une portion de code et de considérer cette portion comme une "boîte noire" est un problème déjà traité en parallélisation inter-procédurale (cf. chapitre 1). Notamment, ces motifs d'entrées et de sorties peuvent être calculés automatiquement par les régions de tableaux [Cre96]. Ce niveau de granularité et ce mode de description fixent le grain du placement tout en permettant de respecter les dépendances intra-procédurales. Cela permet de prendre des décisions de placement, d'effectuer les calculs de dépendances sans pour autant connaître la macro-instruction. Les accès intra-procéduraux peuvent parfois comporter des modulos [BFCM94], la valeur du modulo est alors constante et permet de représenter des accès par exemple circulaire sur certaines dimensions de tableaux (fig. 2.2).

3.1.4 Modélisation des accès

Définition 3.3 *Soit un nid parfait de* n *boucles définissant un espace d'itérations* $\mathcal{I}$*. Soit la procédure* `P` *comprise dans ce nid qui référence le tableau* Φ*. Soit* $\mathcal{M}$ *un espace d'itérations. L'***ensemble des données référencées**[5] *noté* $\mathcal{R}(\mathbf{i})$ *par la procédure* `P` *au cours de l'appel* $\mathbf{i}$ *est :*

$$\mathcal{R}(\mathbf{i}) = \{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.\mathbf{i} + M.\mathbf{m}\ mod\ \mathbf{k}\} \tag{2.1}$$

où $I,\ M \in \mathcal{L}(\mathbb{Z}^n,\ \mathbb{Z}^m)$ *et où* $\mathcal{L}(\mathbb{Z}^n,\ \mathbb{Z}^m)$ *désigne l'ensemble des applications linéaires de* $\mathbb{Z}^m \to \mathbb{Z}^n$*.*

[5] Aussi appelé motif.

Les matrices I et M sont des matrices d'accès. La première matrice permet de paramétrer les accès d'une procédure par un déplacement dans le tableau selon l'espace d'itération externe. La seconde matrice définit l'ensemble des éléments de tableaux accédés référencés par une procédure élémentaire (fig. 2.6).

3.1.5 Hypothèses sur les accès

Les lectures comme les écritures sont représentées par ces fonctions d'accès (exemple 2.6). Néanmoins, on ne permet pas d'écrire à deux reprises un même élément de tableau : l'application est dite en "*single assignement*"[6]. Les programmes ne présentent donc pas de dépendance en écriture. De plus, les fonctions d'accès sont linéairement indépendantes par dimension de tableau ce qui signifie que les matrices I et M sont diagonales à une permutation près.

Si l'on décompose la fonction d'accès :

$$\varphi = I.\mathbf{i} + M.\mathbf{m} \ mod \ \mathbf{k}$$

en autant d'équations qu'il y a de dimensions de tableau. Chaque équation correspondant à une dimension de tableau λ a pour forme :

$$\boldsymbol{\varphi}_\lambda \ = \ (I_{\kappa,\lambda}.i_\kappa \ + \ M_{\kappa',\lambda}.m_{\kappa'}) \ mod \ k_\lambda$$

Chaque élément de la matrice I ou de la matrice M, noté ci-dessus respectivement $I_{\kappa,\lambda}$ et $M_{\kappa',\lambda'}$ est entier. Enfin, les équations d'accès par dimension de tableau sont linéairement indépendantes, et les matrices d'accès M, I sont positives et diagonales à une permutation près. Ceci signifie qu'*au plus un coefficient des matrices I et M non nul est présent par ligne et par colonne.*

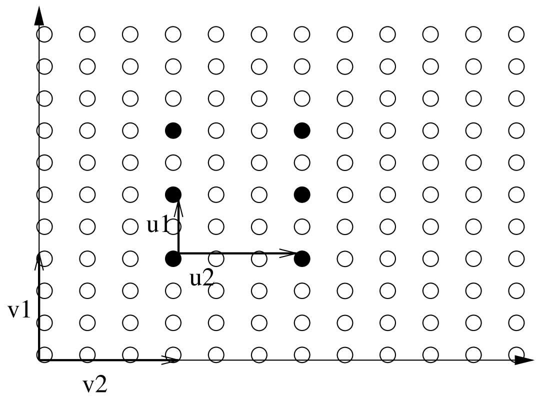
L'ensemble des travaux exposés par la suite se placent implicitement dans ce cadre applicatif. Les caractéristiques de nos applications permettent de définir automatiquement les buffers `IN` et `OUT` paramétrant l'appel de procédure. Toute application traitée peut être spécifiée au moyen de références, des instructions et déclarations usuelles de n'importe quel langage comportant boucles `do`, `procedures` et tableaux. Les langages comme `ALPHA` [Wil94] développé par l'INRIA ou `Fortran 90` [ABM+92] en sont de bons exemples.

Figure (2.7), le schéma général des accès intra-procéduraux est présenté. Il correspond à la procédure de la figure (2.4).

3.2 Exemple

Ces hypothèses peuvent paraître restrictives, mais en pratique sur l'ensemble des benchmarks soumis, très peu de tâches présentaient des accès d'une complexité supérieure. Dans la plupart des cas. Les zones accédées sont de simples segments de tableaux comme pour les principales procédures élémentaires de traitement du signal : la transformée de Fourier ou la formation des faisceaux ainsi que la plupart des calculs de somme. Aussi, la matrice M est souvent l'identité. Néanmoins, les adressages modulo sont très fréquents à cause de nombreuses applications devant calculer des directions. Les accès présentant des dépendances lecture-lecture sont aussi fréquents, y compris sur la dimension temps du tableau. Cette section présente un exemple classique (2.8) de traitement. On notera que l'antenne est représentée comme un tableau mono-dimensionnel.

[6]Traduire : affectation unique, un élément de un tableau ne peut être écrit qu'une seule fois.



$$I = \begin{pmatrix} 4 & 0 \\ 0 & 3 \end{pmatrix}, M = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}, up(\mathbf{m}) = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Ce dernier vecteur est donné par l'utilisateur, par conséquent l'ensemble référencé par une itération $\mathbf{i}$ est

$$\left\{\varphi | \varphi \in \ \Phi, \ \exists \ \mathbf{m} \leq \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \varphi = \begin{pmatrix} 4 & 0 \\ 0 & 3 \end{pmatrix} .\mathbf{i} + \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix} .\mathbf{p}\right\}$$

En l'occurence, sur la figure, cette itération est

$$\mathbf{i} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

pour cette itération, l'ensemble des références est

$$\{(3,3);(6,3);(3,5);(6,5);(3,7);(6,7)\}$$

FIG. 2.6. – *Exemple de références aux éléments de tableaux*

```
 ...
SUBROUTINE APPEL_PROC(I1,I2,...In)
DO M1=0, up(M1)
  ...
    DO Mm=0, up(Mm)
      INk(M1..Mm)=ARRAY_IN(I.(I1,..,In)+M.(M1,..Mm))
    ENDDO
  ...
ENDDO
  ...
PROC(..,INk,..,OUTk,..)
  ...
DO M1'=0, up(M1')
  ...
    DO Mp'=0, up(Mp')
      ARRAY_OUT(I'.(I1,..,In)+M'.(M1',..Mp'))=OUTk(M1',..Mp')
    ENDDO
  ...
ENDDO
 ...
```

où $up(I1) \in [0, \infty]$.

Le modèle général d'appel de procédures permet de lire les données dans le tableau d'entrée `ARRAY_IN`, grâce aux fonctions d'accès `I` et `M`. Ces données sont placées dans le tampon `IN`, paramètre de la procédure `PROC`. La procédure `PROC` effectue le calcul et place le résultat dans le tampon `OUT`. Enfin le résultat est écrit dans le tableau `ARRAY_OUT` grâce aux fonctions d'accès `I'` et `M'`. Enfin, il peut y avoir plusieurs références différentes au même tableau comme il peut y avoir plusieurs références dans différents tableaux. Les copies explicites ne sont naturellement pas effectuées à l'exécution.

FIG. 2.7. – *Références aux tableaux en écriture et en lecture*

```
SUBROUTINE APPEL_FFT(TEMPS,CAPTEURS)
DO TEMPS_LOCAL=0,511
  ECHANTILLONS(TEMPS_LOCAL)=ACQUISITION(511*TEMPS+TEMPS_LOCAL,CAPTEURS)
ENDDO
FFT(ECHANTILLONS,FREQUENCE)
DO FREQUENCE=0,255
  SPECTRES(TEMPS,FREQ_LOCALES,CAPTEURS)=FREQUENCES(FREQ_LOCALES)
ENDDO
```

Chaque appel de FFT calcule 256 fréquences représentées par des complexes,[a] à partir de 512 échantillons de temps. Le tableau résultant dispose donc de trois dimensions.

Soient $\mathbf{i_t}, \mathbf{i_c}, \mathbf{m}_{tl}, \mathbf{m_{fl}}$ les variables respectives associées aux paramètres `TEMPS, CAPTEURS` et aux indices internes `TEMPS_LOCAL, FREQUENCE`. Soient $\boldsymbol{\varphi}^{acq} = (\varphi_t^{acq}, \varphi_c^{acq})^t$ un élément du tableau `ACQUISITION` et $\boldsymbol{\varphi}^{spc} = (\varphi_t^{spc}, \varphi_f^{spc}, \varphi_c^{spc})^t$ un élément du tableau `SPECTRE`. Les équations suivantes définissent les accès :

Accès en lecture dans le tableau `SPECTRE`

$\varphi_t^{acq} = 512 \times \mathbf{i_t} + \mathbf{m}_{tl}$, avec $0 \leq \mathbf{m}_{tl} < 512$

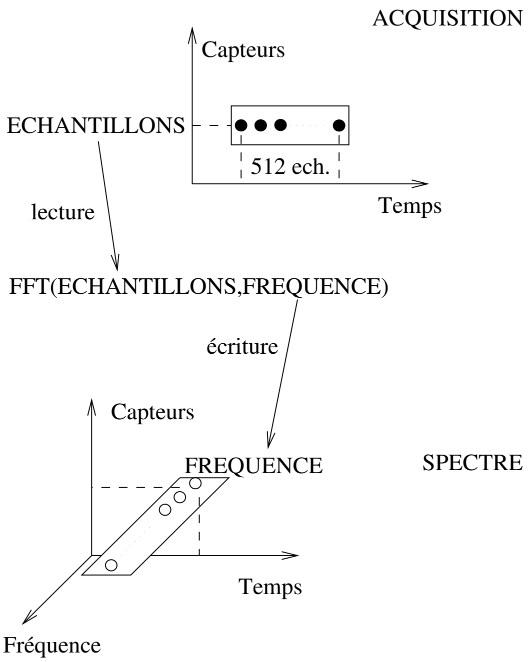
$\varphi_c^{acq} = \mathbf{i_c}$

Accès en écriture dans le tableau `ACQUISITION`

$\varphi_t^{spc} = \mathbf{i_c}$

$\varphi_f^{spc} = \mathbf{m}_{fl}$, avec $0 \leq \mathbf{m}_{fl} < 256$

$\varphi_c^{spc} = \mathbf{i_c}$



[a] Ainsi, on dispose des amplitudes et déphasages.

FIG. 2.8. – *Accès aux tableaux pour les paramètres de la FFT*

4 Mise en oeuvre du placement manuel

Dans cette section, l'approche manuelle du placement est présentée en deux parties : d'une part les transformations locales permettent de restructurer indépendamment un nid de boucles parfait, d'autre part les transformations globales restructurent un ensemble de boucles mal imbriquées. Ces notions sont illustrées sur un cas concret d'application, la Veille Bande Large (VBL). Cette section présente selon quels critères l'opérateur de traitement du signal utilise les transformations de boucles présentées au chapitre 1. Voici diverses questions qui se posent de manière intuitive à l'expert : Faut-il associer un processeur à chaque capteur ? Dans ce cas on devra faire beaucoup de communications entre la transformée de Fourier et la formation des faisceaux. Ou bien pourrait-on associer un processeur par fréquence ? Ce qui constitue une perte de parallélisme au niveau de la transformée de Fourier. Bien entendu des solutions intermédiaires sont envisageables, c'est ce qui est présenté sur cet exemple.

4.1 Transformations locales

Les transformations locales sont le partitionnement et la permutation de boucles (cf. chapitre 1). Ces transformations sont guidées par des préférences établies en fonction des dimensions des ressources et du maintien d'un niveau de performances. Les transformations locales utilisées sont légales car elles portent sur des indices ne comportant ni dépendances lecture/écriture, ni dépendances écriture/écriture.

4.1.1 Préférence et critères de décision

Choix des partitionnements : Le partitionnement permet d'exprimer le parallélisme qui peut apparaître sur une ou plusieurs dimensions, tout en minimisant les communications et la mémoire nécessaires. Pour cela, si des données sont référencées par plusieurs itérations elles devront plutôt résider sur le même processeur. En effet, si ces données sont placées sur deux processeurs différents, il faut les stocker sur chaque processeur et les communiquer aux deux processeurs. Dans ce cas, les dépendances en lecture sont utilisées pour établir les préférences. D'autre part, si la mémoire disponible est trop faible, il faudra calculer des partitions référençant peu d'éléments de tableaux, et éventuellement répartir les calculs sur un temps plus grand, ceci temps à faire diminuer la taille des partitions. Le partitionnement permet donc de jouer sur la granularité du problème, aussi bien sur un plan parallélisme que sur la localité et intervient donc directement sur les ressources sous-jacentes : processeurs et mémoire. Pour concilier les objectifs du partitionnement, il est nécessaire d'enchaîner successivement plusieurs partitionnements, et de paramétrer ceux-ci par rapport au parallélisme et à la localité.

Choix des permutations : Les permutations sont utilisées dans le même objectif que l'échange de boucles (cf. chapitre 1). Elles permettent de fixer un ordre d'exécution, une fois le parallélisme donné. Le critère de choix est ici de privilégier la localité des données accédées, en fonction de la mémoire locale ou pour éviter les communications.

4.1.2 Le cas de la formation de voies

Prenons comme exemple le cas de la Formation des Voies introduite en figure (2.9). Elle succède généralement directement ou indirectement à la FFT, elle prend donc en entrée un tableau de complexes (codés sur 2 octets) à trois dimensions : temps, capteurs, fréquences. Le système calcule un spectre d'amplitude pour chacune des 128 voies en sortie (codé sur 1 octet) (fig. 2.9).

```
DO T = 0, INFINI
 DO V = 0, 128-1
   DO F = 0, 200-1
     FORMATION_DE_VOIES(T,F,V)
   ENDDO
 ENDDO
ENDDO

...

SUBROUTINE FORMATION_DE_VOIES(T,F,V)

DO V_LOCAL = 0, 192-1
   IN(V_LOCAL) = SPECTRES(T,4 * V + V_LOCAL, F)
ENDDO

call FORME_VOIES(IN, OUT)

VOIES(V, F, T) = OUT

...
```

FIG. 2.9. – *La formation des voies*

Pour la `formation des voies`, plusieurs informations permettent de choisir un partitionnement, ainsi qu'un ordre lexical d'exécution :

Dépendances en lecture : Procédons à une analyse des dépendance en lecture du nid de boucle, permettant de rendre compte des dépendances en lecture (cf. § 2.3.2, p. 30, chap. 1). Le polyèdre de dépendances en lecture peut s'écrire :

$$\begin{cases} dt = 0 \\ -189 \leq 4.dv \leq -1 \\ df = 0 \end{cases}$$

où $(dt, dv, df)^t$ est un vecteur de dépendances en lecture correspondant aux indices `T,V,F`.

Si l'on procède à un calcul du nombre de références en lecture N_r^r et de données référencées N_d^r pour une itération donnée de la dimension infinie :

$$N_r^r = 127 * 191 * 200 = 4851400 \quad \text{et} \quad N_d^r = 139800$$

Nombre de références en écriture [7] : N_r^r pour un temps donné : $N_d^r = 25600$

Le polyèdre de dépendances et les volumes calculés indiquent que les données, qui ont été préalablement écrites, sont souvent réutilisées selon la dimension des voies. Par conséquent, la boucle `V` doit donc être interne pour maximiser la réutilisation des données. De plus, le parallélisme doit être sur les dimensions `T` ou/et `V`. Ci-dessous est exposé l'impact de ces critères sur le placement :

[7] Le nombre de références en écriture est égal au nombre de données référencées

La mémoire locale : Si l'on dispose d'une mémoire locale avec 256Ko, il faut définir des blocs de taille réelle afin que la totalité des blocs en écriture et en lecture puisse être placée dans la mémoire locale. Pour cela, on peut choisir de partitionner la dimension `F` à l'aide d'un coefficient de partitionnement L_f, il s'agit alors de résoudre la contrainte :

$$128 * L_f + 2 * 24257 * L_f < 262144$$

Le premier terme définit le nombre de données référencées en écriture dans la mémoire locale. Le second terme donne le nombre de données référencées en lecture dans le cache (sachant que les données sont codées sur deux entiers). Si l'on avait choisit de partitionner plus d'une dimension (par exemple le temps, à l'aide du coefficient P_t) la contrainte à résoudre devient non linéaire :

$$128 * L_f * P_t + 2 * 24257 * L_f * P_t < 262144$$

Un partitionnement avec uniquement $L_f = 5$ sera néanmoins retenu, la dimension fréquence se réecrit :

$$f \;=\; 5.f_L \;+\; f_l, \;\; f_l \;<\; 5$$

Où l est la dimension itérant selon la mémoire locale.

Le parallélisme : Le partitionnement en blocs de $C_f = 5$ fréquences n'autorise plus qu'un parallélisme maximum de $max(f_L) = 40$ sur la dimension fréquence. Si la machine dispose de 8 processeurs, il faudra combiner un deuxième partitionnement de coefficient $P_f = 8$:

$$f_L \;=\; 8.f_c \;+\; f_p, \;\; f_p \;<\; 8$$

Le partitionnement de la dimension fréquence s'écrit alors :

$$f \;=\; 40.f_c \;+\; 5.f_p \;+\; f_l, \;\; f_l \;<\; 5, \;\; f_p \;<\; 8, \;\; f_c \;<\; 5$$

Cela signifie que $L_f = 5$ itérations de la procédure de formation de voies utiliseront la mémoire locale, sur chacun des 8 processeurs résideront 5 blocs. Les variables f_l, f_c représentent respectivement : l'indice de l'itération dans un bloc présent sur un processeur, l'indice d'un bloc cyclique et parallèle sur les processeurs. Enfin, f_p représente le numéro logique du processeur. La figure (2.10) représente le nid de boucles restructuré par le partionnement présenté ci-dessus.

4.2 Transformations globales

Les décisions prises dans l'exemple de la formation des voies, ne prennent pas en compte l'ensemble de l'application. Par exemple, les décisions prises localement peuvent engendrer des communications importantes. Il faut donc tenir compte de l'ensemble des nids de boucles de l'application pour pouvoir calculer un placement global efficace. L'approche utilisée par les opérateurs de traitement du signal consiste à enchaîner des fusions de boucles.

```
DO T = 0, INFINI
c Boucle cyclique
DO FC = 0,5-1
  c Boucle sur les 8 processeurs
  DO FP = 0, 8-1
    c Boucle locale a un processeur
    DO FL = 0, 5-1
       DO V = 0, 128-1
         FORMATION_DE_VOIES(T,FP,FL,FC,V)
       ENDDO
    ENDDO
    c fin des traitements locaux a un processeur
  ENDDO
  c fin de l'execution parallele
ENDDO
c fin de tous les cycles paralleles
...

SUBROUTINE FORMATION_DE_VOIES(T,FP,FL,FC,V)

DO V_LOCAL = 0, 192-1
   IN(V_LOCAL) = SPECTRES(T,4 * V + V_LOCAL, 40*FP+8*FL+FC)
ENDDO

call FORME_VOIES(IN, OUT)

VOIES(V, F, T) = OUT

...
```

FIG. 2.10. – *La formation des voies partitionnée*

4.2.1 Heuristiques et critères de fusion

Dans l'état de l'art sur le placement (cf. chapitre 1), nous avons introduit les différentes techniques permettant de maximiser la localité. A cet effet, le programmeur d'applications de traitement du signal utilise les fusions de boucles et cherche à satisfaire les heuristiques (établis au chapitre 1). De plus, il faut tenir compte des communications pouvant résulter des choix de distribution exprimés par le partitionnement.

Objectifs des Fusions de boucles :

- à la réduction de l'allocation mémoire des tableaux
  Cet effet est obtenu en réduisant la durée de vie des données : Une donnée venant d'être produite est utilisée le plus tôt possible.
- au réordonnancement global de l'application
  La satisfaction des contraintes de latence et de récurrence nécessite le déclenchement des calculs le plus tôt possible, c'est-à-dire dès la production des données requises en entrée.
- au traitement de la dimension infinie
  Le traitement de la dimension infinie s'effectue par fusion de boucles, cela permet de n'avoir qu'une dimension infinie externe à l'ensemble de l'application.

Critères de fusion : Soit la fusion de deux boucles `I1` et `I2`, voici les critères de fusion généralement retenus :

- La boucle `I2` comporte des dépendances en lecture, si `I1` est fusionnée avec `I2`, la boucle résultante ne doit pas être parallélisée. En effet, si l'on parallélise une telle boucle, cela engendrera des communications et synchronisations qui diminuent les performances et compliquent le code à générer.
- Si la boucle `I2` ne comporte pas de dépendances en lecture, on peut procéder à la fusion des boucles, si la boucle résultante peut être parallélisée, on dispose de moins de parallélisme sur cette dimension.
- Enfin, la fusion permet d'envisager différents séquencements des macro-instructions. Le critère à ce niveau consiste à regrouper les composantes connexes du graphe de dépendances pour pouvoir effectuer les calculs le plus tôt possible et donc satisfaire les contraintes temps réel. En d'autres termes, ceci permet d'exécuter les calculs inutiles à l'élaboration d'un résultat après la production de celui-ci.

Pour réaliser les fusions, à partir de la spécification, le programmeur de traitement du signal utilise les ensembles d'éléments de tableaux référencés (en lecture et en écriture) pour chaque tâche. On calcule l'ensemble des macro-instructions nécessaires en analysant les éléments de tableaux devant être impérativement lue par une macro-instruction de sortie. L'opérateur procède ainsi récursivement selon l'orientation inverse du graphe de dépendances, jusqu'aux calculs de l'ensemble des macro-instructions d'acquisition nécessaires. Si cette technique est valable pour les chaines linéaires de flot de données, le problème est rendu beaucoup plus complexe lorsque le programmeur est confronté à un graphe quelconque.

Fusion pour éviter les communications : Considérons le partitionnement de deux nids de boucles, l'un écrit dans un tableau lu par l'autre. Le choix des partitionnements est guidé à nouveau par le critère suivant : on favorisera les couples de partitionnements pour lesquels deux procédures

```
c Traitement Carre
c ----------------
  DO T=0,INFINI
    DOALL I=0,3
c PARTIE INTRA PROCEDURALE
      TAB12[T,I]=TAB1[T,I]*TAB1[T,I]
c FIN PARTIE INTRA PROCEDURALE
    ENDDO
  ENDDO

c Traitement Diff
c --------------
  DO T=0,INFINI
    DOALL I=0,3
c PARTIE INTRA PROCEDURALE
      TAB23[T,I]=TAB12[2*T,I]-TAB12[2*T+1,I]
c FIN PARTIE INTRA PROCEDURALE
    ENDDO
  ENDDO

c Traitement Somme
c ---------------
  DO T=0,INFINI
c PARTIE INTRA PROCEDURALE
    S=0
    DO MI=0,3
      S=S+TAB23[T,MI]
    ENDDO
    TAB3[T]=S
c FIN PARTIE INTRA PROCEDURALE
  ENDDO
```

FIG. 2.11. – *Cas d'école*

élémentaires dépendantes sont distribuées sur le même processeur. Ceci permet de supprimer des choix de fusions dans l'arbre de décision issue de la réflexion du programmeur. Inversement, le programmeur établie des préférences entre boucles devant être fusionnées afin d'éviter des communications et de réduire le contrôle. Une fois ces boucles fusionnées, il en résulte une perte de parallélisme supprimant de même une partie des choix de partitionnements dans l'arbre de décision que le programmeur à en tête.

4.2.2 Exemple sur un cas d'école

L'illustration de ces critères sur les transformations de boucles nécessite l'introduction d'un "cas d'école" simplifié (fig. 2.11). On considère les transformations locales effectuées : la boucle `I` s'effectue en parallèle pour les deux traitements `Carre` et `Diff`.

Le placement du nid de boucles utilise ces critères (fig. 2.13) de fusion[8] :

- La boucle `T`, représentant le temps a pu être fusionnée sur l'ensemble du graphe. Elle ne comporte pas de dépendances en lecture, aussi la boucle `T1` résultante peut elle être parallèle. Cependant

[8]Nous ne précisions pas dans l'exemple la phase de communication

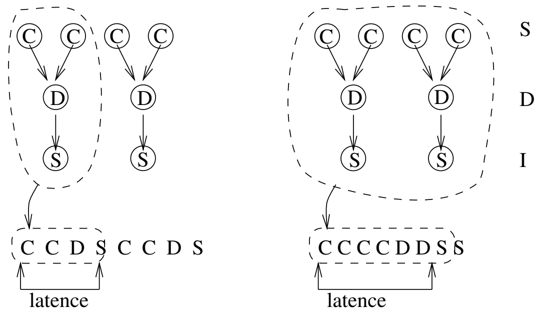


FIG. 2.12. – *Importance des composantes connexes du DFG*

si on la parallélise, on a besoin de 6 processeurs et de communications. On peut choisir de la laisser séquentielle.

- Un ordonnancement défini par ce nid de boucles s'exprime à l'aide d'une chaîne de traitement:

$$..CCDSCCDSCCDS..$$

les caractères C, D, S représentent respectivement l'élévation au carré, la différence et la somme. Le corps de la boucle `T` exécute une seule composante connexe du graphe de dépendances : $CCDS$. Aussi, cet exemple privilégie des dates d'exécution au plus tôt afin de réutiliser le plus vite possible les données, ce qui a un impact bénéfique sur la latence et sur la mémoire.

En l'absence d'autres contraintes temps réel, un mauvais choix peut avoir des conséquences désastreuses. Par exemple, effectuer plusieurs composantes non connexes du graphe de dépendances dans le corps de boucle : ... $CCCCDDS$ ... aurait entrainé l'allocation de plus gros tableaux et une détérioration de la latence. Un tel ordonnancement ne serait justifié que si l'acquisition successive de quatre données en entrée était impérative à cause de contraintes de signal (fig. 2.12).

4.3 Le cas de la Veille Bande Large

Nous présentons, dans cette section, un exemple d'application réaliste nommée la Veille Bande Large (fig. 2.14). Elle procède aux calculs de la FFT, de la Formation des voies, de l'énergie pour calculer le spectre d'amplitude comme expliqué précédemment. Le `Regroupement Large Bande` exécute un traitement par voie (en fait par direction) sur l'ensemble des fréquences. Les intégrations qui suivent permettent le calcul et la stabilisation dans le temps de paramètres purement opérationnels (fig. 2.15).

Dans l'exemple (fig. 2.15), on considère que tout nid de boucles lit dans le tableau écrit par le nid de boucles précédent. Les variables (φ_i) représentent les dimensions de ces tableaux intermédiaires.

Pour placer l'application de la figure 2.15, considérons un contexte de génération de code simple. Le code généré doit présenter un minimum de communications et conserver un maximum de parallélisme. Les dimensions finies des nids de boucles qui sont utilisées pour exprimer le parallélisme de l'application sont maintenant présentées :

```
DO T=0,INFINI
  DO T1=0,1
    DOALL I=0,3
      TAB12[T1,I]=TAB1[T1,I]*TAB1[T1,I]
    ENDDO
  ENDDO
  DOALL I=0,3
    TAB23[I]=TAB12[0,I]-TAB12[1,I]
  ENDDO

  c Communications

  S=0
  DO MI=0,3
    S=S+TAB23[MI]
  ENDDO
   TAB3=S
ENDDO
```

FIG. 2.13. – *Cas d'école placé sur les processeurs*

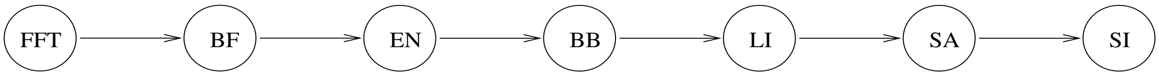


FIG. 2.14. – *L'application VBL schématisée*

Transformée de Fourier : La dimension parallèle est celle du capteur c. Chaque processeur dispose alors d'un ou plusieurs spectres.

Formation des Voies : La dimension parallèle est celle des fréquences f. Ceci entraîne une transposition de tableau avec la tâche précedente. Il est aussi possible de paralléliser selon les voies, mais cela demande un surcoût de communications due aux dépendances lecture-lecture.

Energie : La dimension fréquence f est parallèle, qui fusionne avec la dimension f de la Formation des voies.

Regroupement large bande : La dimension fréquence f est forcément séquentielle (c'est une dimension intra-procédurale). Le parallélisme sur les voies est donc nécessaire malgré une communication avec l'Energie.

Intégration courte : Seule la dimension voie v dispose de parallélisme sur les voies. Le parallélisme est exprimé selon cette dimension bien qu'une communication avec l'Energie soit nécessaire.

Le nombre de cycles requis pour l'exécution des nids de boucles suivants (Sabilisation azimutale, Intégration Longue) est négligeable par rapport à celui requis pour les premières tâches, sur cet exemple. Pour fusionner selon les dimensions temps de l'application, on détermine l'ensemble des éléments de tableaux nécessaires à l'exécution d'une instruction `integration longue`. Ainsi, on sait calculer l'ensemble des instructions de `regroupement large bande` nécessaire. On continue ce raisonnement récursivement en calculant tour à tour l'ensemble des éléments de tableaux nécessaires et l'ensemble des instructions nécessaires à un même nœud destination du graphe. Ce calcul s'arrête lorsqu'un nœud représente un traitement d'entrée. Une automatisation de ce calcul peut se baser par exemple à partir de la composition d'opérateurs portant sur les régions de tableaux[Cre96].

Transformée de Fourier

$entree\ :\ \{\ 512.t \leq\ \varphi_t\ <\ 512.t+512,\ \varphi_c\ =\ c\ \}$
$sortie\ :\ \{\ t\ =\ \varphi_t,\ \varphi_c\ =\ c,\ 0 \leq\ \varphi_f\ <\ 200\ \}$

```
doall t,c
 call FFT(t,c)
enddo
```

Formation des Voies

$entree\ :\ \{\ \varphi_t\ =\ t,\ 4.v\ \leq\ \varphi_c\ < 4.v+192,\ \varphi_f\ =\ f\ \}$
$sortie\ :\ \{\ \varphi_t\ =\ t,\ \varphi_v\ = v,\ \varphi_f\ =\ f\ \}$

```
doall t,f,v
  call FormationDeVoies(t,f,v)
enddo
```

Energie

$entree\ :\ \{\ \varphi_t\ =\ t,\ \varphi_v\ = v,\ \varphi_f\ =\ f\ \}$
$sortie\ :\ \{\ \varphi_t\ =\ t,\ \varphi_v\ = v,\ \varphi_f\ =\ f\ \}$

```
doall t,f,v
  call Energies(t,f,v)
enddo
```

Regroupement Large Bande

$entree\ :\ \{\ \varphi_t\ =\ t,\ \varphi_v\ = v,\ 0\ \leq\ \varphi_f\ <\ 200\ \}$
$sortie\ :\ \{\ \varphi_t\ =\ t,\ \varphi_v\ = v\ \}$

```
doall t,v
  call RLB(t,v)
enddo
```

Intégration courte

$entree\ :\ \{\ 8.t\ \leq\ \varphi_t\ <\ 8.t+8,\ \varphi_v\ = v\ \}$
$sortie\ :\ \{\ \varphi_t\ =\ t,\ \varphi_v\ = v\ \}$

```
doall t,v
  call IntegrationCourte(t,v)
enddo
```

Stabilisation azimutale

$entree\ :\ \{\ \varphi_t\ =\ t,\ v\ \leq\ \varphi_v\ <\ v+8\ \}$
$sortie\ :\ \{\ \varphi_t\ =\ t,\ \varphi_v\ = v\ \}$

```
doall t,v
  call StabilisationAzimut(t,v)
enddo
```

Integration longue

$entree\ :\ \{\ 8.t\ \leq\ \varphi_t\ <\ 8.t+8,\ \varphi_v\ =\ v\ \}$
$sortie\ :\ \{\ \varphi_t\ =\ t,\ \varphi_v\ = v\ \}$

```
doall t,v
  call IntegrationLongue(t,v)
enddo
```

FIG. 2.15. – *La VBL : une application réaliste*

Transformée de Fourier

$entreerequise$: $\{ 64.512.t \leq \varphi_t < 64.512.t + 64.512\}$

```
doall t,c
 call FFT(t,c)
enddo
```

Formation des Voies

$entreerequise$: $\{ 64.t \leq \varphi_t < 64.t + 64 \}$

```
doall t,f,v
  call FormationDeVoies(t,f,v)
enddo
```

Energie

$entreerequise$: $\{ 64.t \leq \varphi_t < 64.t + 64\}$

```
doall t,f,v
  call Energies(t,f,v)
enddo
```

Regroupement Large Bande

$entreerequise$: $\{ 64.t \leq \varphi_t < 64.t + 64 \}$

```
doall t,f,v
  call RLB(t,v)
enddo
```

Intégration courte

$entreerequise$: $\{ 64.t \leq \varphi_t < 64.t + 64,\ v \leq \varphi_v < v + 8 \}$

```
doall t,v
  call IntegrationCourte(t,v)
enddo
```

Stabilisation azimutale

$entreerequise$: $\{ 8.t \leq \varphi_t < 8.t + 8 \}$

```
doall t,v
  call StabilisationAzimut(t,v)
enddo
```

Integration longue

$entreerequise$: $\{ 8.t \leq \varphi_t < 8.t + 8 \}$

```
doall t,v
  call IntegrationLongue(t,v)
enddo
```

FIG. 2.16. – *Analyses statiques sur la VBL*

Le calcul des éléments de tableaux référencés (cf. fig. 2.16) permet d'établir qu'une exécution dans le temps de la procédure `IntegrationLongue` nécessite 64 exécutions dans le temps des procédures `FFT`, de `FormationDeVoies`, d'`Energie` et de `RLB`. La même itération `IntegrationLongue` nécessite 8 exécutions d'`IntegrationCourte` et de StabilisationAzimut. D'autre part il permet de **regrouper les composantes connexes du DFG** afin de déterminer une série de fusions appropriée. Le nid de boucles peut alors être entièrement généré, en tenant compte des phases de communications engendrées par les partitionnements des boucles finies.(fig. 2.17)

Une machine à huit processeurs est considérée, elle dispose d'une interface avec 512 capteurs, et 128 voies sont formées et analysées. Ce choix de fusion permet de traiter les données au plus tôt car le corps de la boucle se compose d'une même composante connexe du graphe de dépendance.

4.4 Utilisation des régions pour le placement manuel

Ce chapitre ne prétend pas apporter une approche méthodologique basée sur les transformations de boucles, mais présente la compilation manuelle des applications de traitement du signal telle qu'elle est abordée en milieu industriel. Une méthode de placement pourrait être développée sur la base des calculs de régions, en associant des transformations de programme comme cela a été présenté sur la VBL.

4.5 Composition des transformations globales et locales

Réaliser un placement manuel nécessite la composition des transformations locales et globales tout en tenant compte des effets sur les dimensions. Le programmeur de traitement du signal doit alors placer une application suivant un ensemble de compromis. Certains on déjà fait l'objet de discussions dans les sections précédentes, en voici les grandes lignes :

Nombre de processeurs/Bande passante : Augmenter le nombre de processeurs nécessite de disposer d'un réseau de communications plus performant.

Mémoire/temps réel : Retarder les calculs pour permettre d'enchaîner plusieurs acquisitions nécessite de conserver les données intermédiaires et échantillons en mémoire.

Nombre de processeurs/temps réel : Augmenter le nombre de processeurs permet de diminuer les temps d'exécution et de satisfaire les contraintes de temps réel.

Bande passante/temps réel : Si les calculs ne sont pas recouverts par les communications, alors la capacité du réseau à communiquer est prépondérante pour satisfaire les contraintes de temps réel.

Mémoire/Bande passante : Stocker un volume de données en mémoire important nécessite dans certains cas, de le communiquer par la suite.

Placer une application en sélectionnant les bonnes transformations de boucles est difficile sur des cas simples [KKB92], même en tenant compte de restrictions de traitement du signal [OWS97]. Ces remarques sont encore vraies lorsque l'on cherche un placement manuellement en prenant en compte l'ensemble de la problématique.

5 Approches automatiques

Dans cette section, les approches concernant le placement et l'optimisation des applications de traitement du signal sont exposées. Si ces approches bénéficient du caractère simplifié des applications, peu de travaux tentent d'exploiter un parallélisme à grain fin.

```
do t1=0,infini
  doseq t2=0,8-1
    doseq t3=0,8-1

      doall C1=8-1
        doseq C2=64-1

          fft(8*8*t1+8*t2+t3,64*C1+C2)

        enddo
      enddo

      c Communications : Il faut redistribuer les donnees en parallele
  selon les frequences

      doall f1=0,8
        doseq f2=0,25
          doseq voies=0,128-1

            FormationDeVoies(8*8*t1+8*t2+t3,25*f1+f2,voies)
            Energie(8*8*t1+8*t2+t3,25*f1+f2,voies)

          enddo
        enddo
      enddo

      c Communications : Il faut redistribuer les donnees en parallele
      selon les voies

      doall voies1=0,8-1
        doseq voies2=0,16-1

          RLB(8*8*t1+8*t2+t3,16*voies1+voies2)

        enddo
      enddo

    enddo

    c Communications : regroupement sur un seul processeur de toute les donnees

    IntegrationCourte(8*t1+t2,voies)
    StabilisationAzimut(t1,voies)

  enddo

  IntegrationLongue(t1,voies)

enddo
```

FIG. 2.17. – *La VBL partitionnée et fusionnée*

5.1 Théorie des graphes

Pour le placement en général, la théorie des graphes propose un cadre mathématique séduisant pour réaliser des optimisations locales sur des algorithmes de traitement du signal. De nombreux travaux reposent sur des concepts de résolution spécialisés manipulant les graphes. Ces approches modélisent les éléments du problème (graphe de tâches, architecture multi-processeur) en extension [LM87][MBL96][BSL96][PW96]. La taille des représentations explose rapidement.

5.1.1 SynDEx

Les travaux réalisés dans le cadre de la compilation du langage `Signal` correspondent à des manipulations de graphe. Une spécification `Signal` se compose d'états ou traitements élémentaires et de transitions associés aux flots mono-dimensionels entrant et sortant. Selon la sémantique du langage, un traitement peut être exécuté si l'ensemble des conditions spécifiées en entrée sont satisfaites.

L'outil SynDEx permettant le placement de `Signal` [BGJ89] permet le partitionnement automatique du graphe de flot de données à un niveau de granularité élevé. Un ordonnancement mono-dimensionnel est alors calculé selon différentes heuristiques comme l'ordonnancement au plus tôt. Pour un tel ordonnancement, le système de placement évalue les performances des traitements : bande passante nécessaire, latence, récurrence de sortie et d'entrée. Les traitements élémentaires ne peuvent pas être partitionnés en différents sous-traitements, si bien que la spécification d'entrée borne le parallélisme maximal au seul nombre de noeuds du DFG.

L'optimisation est manuelle et l'utilisateur a le choix entre ajouter des processeurs, augmenter la bande passante. Il peut aussi re-spécifier l'application en partitionnant manuellement les traitements élémentaires afin d'augmenter le parallélisme. Là encore, même dans un cadre dédié au traitement du signal, même si une grande partie du placement est calculé par `SynDEx`, la responsabilité des décisions repose sur l'utilisateur. SynDEx propose donc une méthodologie appelée "Adéquation Algorithme Architecture" (A3) [Sor94] basée sur des modèles de graphes autant pour spécifier l'algorithme (mise en évidence du parallélisme potentiel) et l'architecture matérielle (mise en évidence du parallélisme disponible), que pour décrire l'implantation en termes de transformations de graphes (réduction de parallélisme) [LS92] [LS93].

En revanche l'outil SynDEx permet de modéliser finement les communications puisqu'elles sont évaluées à partir du graphe résultant du partitionnement.

5.1.2 PEDIGREE

Le compilateur PEDIGREE [NS96] adresse le traitement de données dans un sens large : leur "Bench-Marking" inclus des algorithmes de traitement du signal, de classification, de détection voire de prédiction d'orbite. Si le compilateur permet d'exploiter, voire de combiner le parallélisme de données et de contrôle, la granularité de l'approche reste élevée. Celle-ci est basée sur un graphe "Program Dependence Graph" construit à partir du graphe de contrôle de l'application. Le compilateur est basé sur le partitionnement de ce graphe et l'ordonnancement de ses parties sur les processeurs, l'architecture cible étant MIMD. Le choix des partitionnements et ordonnancements sont guidés par une évaluation récursive des sous-latences associées aux parties du graphe.

5.1.3 Ptolemy

L'environnement Ptolemy permet la simulation, la modélisation, et la génération de code générique pour les applications de traitement du signal. L'environnement propose différents espaces de traitement du signal (signal mono ou multi-dimensionnel, image) et est organisé sur une architecture orientée objet. Les propriétés des différents traitements se transmettent par héritage de classes.

Synchronous DataFlow (SDF) [LM87] : Spécification des applications pouvant être décrites à l'aide de flots de données mono-dimensionnels. La génération de code est basée sur un ordonnancement statique des tâches spécifiées.

Multi-Dimensionnal Dataflow (MD-SDF) [Che94] : Extension de SDF et de son ordonnancement aux flots de données multi-dimensionnels.

Dynamic Dataflow (DDF) : Ce domaine caractérise les applications de traitement de signal pour lesquelles il est difficile de calculer un ordonnancement efficace statiquement, le domaine DDF propose un ordonnancement à l'exécution.

Il existe bien d'autres domaines dans Ptolemy pour lesquels l'environnement propose un modèle d'exécution permettant de générer un code efficace.

Les graphes représentant les applications des domaines SDF et MD-SDF sont des graphes quotient des dépendances de flots. Ce graphe modélise alors la ou les périodes de l'application[9] en représentant une composante connexe maximale du graphe de dépendances de flot. Une fois ce graphe de flot de données donné par l'utilisateur, différents algorithmes spécialisés selon la ressource à optimiser sont construits. Plusieurs machines cibles sont envisageables, bien que les travaux soient principalement orientés vers un parallélisme de contrôle sur machines MIMD.

Ces algorithmes reposent sur le partitionnement du graphe quotient et sur l'ordonnancement des parties de ce graphe [LM87][SL93]. La partition ainsi que les précédences entre les parties du graphe sont représentées de manière cyclique en intention. Ceci limite l'utilisation de ces techniques. Des algorithmes de listes spécialisés sont généralement utilisés pour résoudre le problème.

Les méthodes d'ordonnancements définies pour SDF et MD-SDF permettent l'optimisation d'une ressource, comme la mémoire [MBL96], la mémoire cache [LM87][BML95], le nombre de synchronisations, les communications ou l'optimisation de la latence [BSL96].

5.2 Fusion automatique

Les techniques proches de la fusion de boucles sont utilisées pour compiler efficacement dans le cadre des travaux de l'université d'Arizona [PW96]. Les applications ciblées sont des successions de filtres de signal. Chaque filtre lit et écrit dans des espaces mémoire intermédiaires. Pour chaque paire consécutive, le problème consiste à fusionner ou pas les deux filtres afin de supprimer ces accès. La fusion s'effectue alors sur le graphe de contrôle de l'application.

[9] La partie intra-périodique correspond en fait à un corps de boucles

5.3 Compilation d'un programme HPF pour le signal

Le compilateur FX de CMU adresse la même classe algorithmique de TS que celle définie précédemment [SV96]. La spécification de l'application est basée d'une part sur la description syntaxique des réductions, d'autre part par l'indication des boucles parallèles à l'aide de la primitive HPF : `INDEPENDENT`. Dans cette approche, seules les boucles indépendantes sont parallélisables. Le placement s'effectue à l'aide de la primitive d'HPF : `DISTRIBUTE` et des permutations de boucles. Ces travaux se placent dans des restrictions applicatives semblables à celles présentées dans ce chapitre et propose une approche basée sur le parallélisme de données [SV96]. Les techniques de restructuration globales sont aussi abordées et les aspects temps réel effectivement pris en compte [OWS97].

Cependant, les choix sont guidés par des tests d'efficacité liés aux sélections des permutations, aux choix des distributions et aux réductions. C'est donc une résolution basée sur le "*Generate and Test*"[10]. Cela consiste à parcourir un arbre de recherche pour instancier l'ensemble des paramètres des directives de placement, puis à vérifier par évaluation que les paramètres comme la latence sont satisfaits.

6 Conclusion

Nous avons circonscrit, dans ce chapitre, le domaine applicatif. L'ensemble des restrictions considérées sont résumées dans la section suivante, néanmoins le domaine couvert permet de prendre en compte un nombre important d'applications de traitement du signal.

6.1 Résumé des caractéristiques applicatives

6.1.1 Nids de boucles

- Nid de boucles uniforme, un appel de procédure dans le corps du nid de boucles.
- Un nid de boucle ne présente pas de dépendance en lecture/écriture, ni en écriture/écriture. Le langage est à assignation unique.
- Les domaine de calcul sont hyper-parallépipédiques, les bornes de boucles sont des constantes.
- A chaque dimension de tableau correspond au plus une dimension d'itération externe et au plus une dimension d'itération interne. Les d'accès sont des fonctions affines des indices d'itérations. Les matrices I et M d'accès sont diagonales à une permutation près. Les accès peuvent comporter des modulos.
- Les fonctions d'accès en écritures sont bijectives.
- On ne considère qu'un seul accès par tableau et par nids de boucles.
- Les procédures élémentaires ou macro-instructions ont une durée fixe spécifiée par l'utilisateur.

6.1.2 Graphe de flot de données

- Une séquence de nids de boucles, chacun possédant une dimension infinie. Le tout est représenté par un Graphe de Flot de Données qui est en fait une chaîne linéaire.
- Le Graphe de Flot de données est acyclique.
- Pas de dépendance écriture-écriture au sein du graphe acyclique.

[10] Comme nous le verrons dans le chapitre 11, ce mode de recherche consiste à parcourir exhaustivement tous les nœuds de l'arbre de recherche pour évaluer, dans les cas terminaux, la qualité des solutions.

6.2 Résumé des caractéristiques architecturales

6.2.1 Architecture homogènes SPMD

Les processeurs sont de puissance égales et disposent de la même taille de mémoire locale. On ne considère pas de mémoire cache.

6.2.2 Interconnection

Le temps de communication entre une paire de processeur ne dépend pas de leur position sur le réseau d'inter-connection. Les processeurs peuvent calculer et communiquer simultanément.

Conclusion

Les applications de traitement du signal, comme la plupart des programmes numériques sont exprimés à l'aide de nids de boucles imbriqués. Les accès aux tableaux sont des fonctions affines des itérateurs de boucles, ce qui nous place naturellement dans un contexte de placement à niveau fin de granularité.

Nous avons démontré que gérer les ressources, lors des phases de partitionnement de nids de boucles, nécessitait l'introduction de contraintes non-linéaires (formes disjonctives, *etc...*). Il a aussi été montré, dans ce chapitre, combien il était difficile de conserver la concurrence entre les fusions et les partitionnements utilisés par les programmeurs de traitement du signal, notamment lorsqu'il faut considérer les paramètres de communications et d'allocations.

L'exemple du placement de la `VBL` montre que l'approche de l'expert composant des transformations de boucles globales et locales est avant tout générale.

Or, à l'image des travaux sur Ptolemy, peu d'approches permettent l'automatisation du placement malgré la simplicité apparente de cette classe applicative. De plus, la granularité de ces approches, avec l'absence de fonctions d'accès, ne permet pas de traiter les problèmes de sur-échantillonage et sous-échantillonage très important tant du point de vue des allocations que des communications et que l'expert sait appréhender.

Nous étudions dans le chapitre suivant, comment les techniques traditionnelles de résolution ont été adaptées au placement dans un sens général et si celles-ci permettraient de résoudre le problème global à un niveau de granularité fin.

Chapitre 3

Méthodes de résolution dédiées au problème de placement

Introduction

Placer la succession de nids de boucles d'une l'application de traitement du signal nécessite de concilier des techniques aussi bien locales à un nid de boucle que globales à l'ensemble de l'application (cf. § 2). Chacune des techniques présentée dans les chapitres précédents (cf. § 1) ne considère qu'une partie de notre problématique générale. Lorsque les applications bénéficient des particularités de la classe applicative (cf. § 2), la modélisation se situe à un grain élevé et les représentations sont en extension. Dans ces conditions, la résolution dépend aussi bien de la dimension de l'architecture cible que de la complexité de l'application. Or, l'approche globale doit pouvoir combiner ordonnancement, distribution des calculs, allocation, alignement des donnés et durée des communications. Le placement doit aussi prendre en compte les dimensions de la machine cible ainsi que les aspects temps réel du signal. Résoudre séparement, chaque sous-problème est NP-difficile, comme par exemple l'alignement [Kre93a, LC90] ou la répartition des calculs [Dio96]. Pour un sous-problème isolé de tout contexte, un algorithme spécialement étudié peut trouver efficacement des solutions [Fea94a]. Dans notre cas, plus qu'un contexte, chaque solution partielle doit être conforme aux solutions des autres sous-problèmes pour que l'ensemble puisse constituer une solution globale. Aussi, tant pour les applications de traitement du signal que pour la résolution du placement, l'approche est globale. La difficulté de cette approche globale est de considérer un ensemble de représentations hétérogènes alors que par définition, la solution est une instanciation de ces représentations par des valeurs cohérentes.

Nous présentons dans ce chapitre les méthodes de résolution associés à diffèrents aspects du placement. Ces approches, bien que souvent très efficaces ne traitent que très partiellement la fonction placement et n'ont en conséquence que très peu d'impact industriel.

1 Approche Algorithmique de l'alignement et de la fusion de boucle

L'utilisation du graphe pour représenter et résoudre un problème de placement a fait l'objet de nombreux travaux [LC91] [DR92] [SM96] [PW96]. Plusieurs techniques ont été citées en (chap. 1, p. 21), et reposent sur des algorithmes spécialisés semblables à ceux de la Recherche Opérationnelle. Ces approches, associent manipulation de graphes et évaluation de solutions partielles. Par exemple, la résolution du problème de fusion de boucles [KM93] utilise un dérivé de l'algorithme de Ford-Fulkerson-Minty [LD56], classiquement utilisé en Recherche Opérationnelle. Elles traitent généralement un problème particulier et éloignent de toute généricité la formulation du problème ainsi que sa résolution.

2 Résolution par programmation linéaire en nombre entier (PLNE) du problème de réalignement

De même, le problème de l'alignement [BKK94] peut être exprimé et résolu à l'aide de variables $0-1$ et un résolveur basé sur le "Branch and Bounds". Ce résolveur est basé sur un outil de programmation linéaire en nombre entier `CPLEX` utilisant l'algorithme du Simplex [Chv83]. L'interêt de approche est de considérer les contraintes portant sur les variables $0-1$ comme des liens entre modèles de résolution. L'algorithme de "*Branch and Bound*" contrôle et coordonne alors *implicitement* la résolution entre les resynchronisations et les alignements 1. Si cette approche permet de combiner plusieurs modèles, elle ne différencie pas les variables du problème de celles contrôlant la résolution. Ceci entraine un manque de contrôle dans la recherche de solutions. De plus, la résolution en programmation $0-1$ a le désavantage d'engendrer beaucoup de variables lorsque le problème devient important.

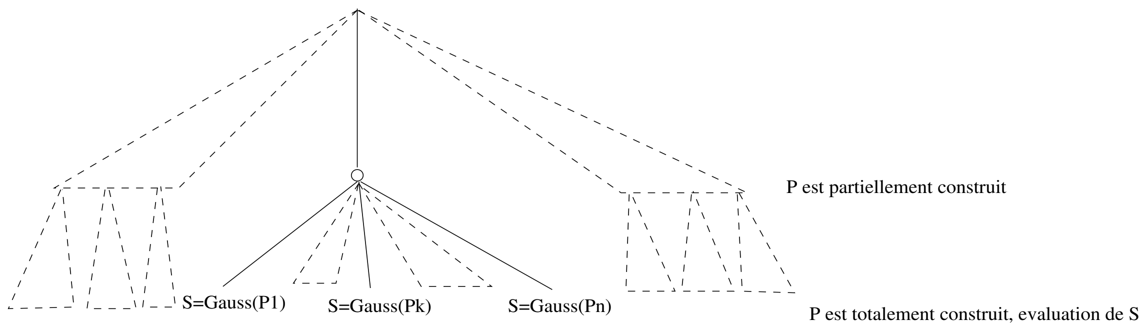
3 Résolution du problème des dépendances interprocédurales

Le système d'analyse des dépendances interprocédurales PIPS (Paralléliseur Inter-procédural de Programmes Scientifiques) développé par le Centre de Recherche de l'Ecole des Mines de Paris, utilise l'algorithme du Simplex [Chv83] pour résoudre le calcul des dépendances inter-procédurales [IT88a]. De plus, le même système utilise l'élimination de Fourier-Motzkin [Fou27] [CI96] pour l'implantation des opérateurs d'analyse de régions de tableaux.

4 Résolution de problèmes d'ordonnancement sous contraintes de ressource en combinant Programmation Linéaire en Nombre Entier et recherche de solutions

Beaucoup d'approches utilisent directement l'algèbre linéaire pour formuler et résoudre les problèmes de placement 1. Un résolveur basé sur l'algorithme du Simplex [Chv83] et la Programmation Linéaire en Nombre Entier [Fea88] sont utilisés pour analyser les dépendances, résoudre de manière indépendante les fonctions d'ordonnancement et de placement [DR94b, Fea92b, Fea92a].

Il est par exemple possible d'optimiser de façon mono-critère les communications en intégrant une heuristique dans l'élimination de Gauss [Pla95]. Le couplage avec la recherche de solutions se fait alors par évaluation et passage de valeurs [Fea94a] (i.e. "*generate and test*", fig. 3.1). Mais généralement la résolution de contraintes non-linéaires associées aux nombreuses contraintes de ressources, selon Paul Feautrier, nécessitent la combinaison d'algorithmes de recherche et de résolution [Fea94a].



L'algorithme `Gauss` accepte en paramètre un problème P_k, donnant une solution S, si la solution échoue, d'autres problème P_i sont générés et évalués.

FIG. 3.1. – *Recherche arborescente par passage de valeurs*

5 Utilisation et couplage d'heuristiques

Les problèmes de placement offrent beaucoup d'heuristiques de domaine (voir chap. 13, § 2 p. 234) : maximiser le parallélisme, éliminer les communications, ou au mieux, les recouvrir. Ils permettent alors de couper la recherche de solutions et sont exploités dans deux sens différents :

Au sens explicite : La méthode de résolution propose une solution heuristique, coupant tout autre solution ne satisfaisant pas les conditions de l'heuristique. Ajouter les conditions de l'heuristique au problème initial peut alors simplifier considérablement la complexité initiale. Dans certains cas, il peut s'agir d'un choix arbitraire, on ne garantit pas l'optimalité au sens général, mais seulement vis-à-vis de l'heuristique. Dans d'autre cas, on démontre que la solution optimale obtenue avec l'heuristique, est meilleure que tout autres solutions obtenues sans heuristiques. Dans ce sens, la classe des ordonnancements affines permettent de produire des ordonnancements quasi-optimaux [DKR91]. Dans tout les cas l'approche reposera sur la qualité de l'heuristique.

Au sens implicite : L'heuristique exprime une préférence permettant de borner efficacement la solution, mais la totalité de l'espace de recherche est considéré. Ce type d'utilisation permettrait de raffiner le calcul des bornes d'algorithmes de “Branch and Bound”. La preuve d'optimalité est alors calculé par la machine.

L'intégration de l'heuristique est aussi difficile que le couplage entre recherche et résolution. En ce qui concerne la Recherche Opérationnelle (RO), les algorithmes sont très souvent spécialisés selon l'heuristique, on perd d'avantage la généricité et par exemple, la possiblité de combiner plusieurs heuristiques de nature différentes. Dans le cas de la Programmation Linéaire en Nombre Entier (PLNE) ou d'une méthode de résolution basée sur l'algèbre linéaire, les heuristiques contribuant à borner la solution peuvent se révéler efficaces [Fea94a]. Cependant la combinaison d'heuristiques de diverses natures est difficilement inégrable dans le cadre linéaire de la PLNE.

6 Synthèse

Approche	*Exemple d'algorithmes*	*Exemple d'applications*	*Avantage/Inconvénients*
Recherche Opérationnelle	Flot Maximal, algorithmes spécialisés	Fusion de boucles [KM93]	Combinatoire importante, pas de généricité mais algorithmes facilement spécialisés
Résolution basée sur l'algèbre linéaire	Gauss	Placement de nids de boucles[Fea94b]	Formulation algébrique dans laquelle le problème doit être exprimable. Couplage possible entre espace de recherches uniquement par "passage de valeur".
Programmation Linéaire en Nombre Entier	"Branch and Bound"	Ordonnancement	Formulation algébrique et algorithme de recherche spécialisable. Le problème doit être exprimable en PLNE (ensemble d'inéquations linéaires). Représentations et méthodes de recherche sont uniques, donc couplage par passage de valeurs.
Programmation Linéaire 0-1	Simplex [Chv83]	Alignement, resynchronisations [BKK94]	La séparation de la formulation de l'alignement et des resynchronisations permettent de construire une recherche concurrente. Le problème doit être exprimable en $0-1$. Contrôle implicite de la résolution ammène à une résolution du problème délicate
Programmation Linéaire	Simplex [Chv83], Fourier-Motzkin [Fou27]	Analyse des dépendances	Utilisation d'algorithmes génériques pour diverses analyses de dépendances

Conclusion

Nous avons montré dans cette partie quelques exemples illustrant le manque de généricité des algorithmes de résolution basés sur la Programmation Linéaire en Nombre Entier ou la Recherche Opérationnelle. La liste des problèmes résolus à l'aide de la programmation linéaire n'est pas exhaustive et l'on assiste en fait à une profusion d'algorithmes. Très souvent spécialisés selon une heuristique, ils ne permettent pas de combiner simultanément la résolution sur différents sous-problèmes de placement. De plus, ces algorithmes intègrent difficilement le concept de recherche de solutions. Pourtant, la granularité de ces approches est adaptée au problème du placement des applications de traitement du signal (cf. 2). Quelle peut être la méthode de résolution adaptée pour le problème de placement global ?

Conclusion

La présentation des états des arts a montré la complexité du problème, associé à un nombre important de contraintes à satisfaire. L'état de l'art sur le placement souligne la difficulté de traiter plusieurs ressources et la latence de l'application simultanément. S'il est possible d'exprimer ces contraintes, dont certaines d'entre elles sont non-linéaires, les différentes approches présentées consistent à maximiser le parallélisme ou la localité à partir d'algorithmes spécialisés, en résolvant des problèmes particuliers de placement (partitionnement, ordonnancement, alignement...). Ce manque de généricité ne permet pas d'englober simultanément partitionnements, communications, ordonnancements, allocations, contraintes temps-réel. Or, nous avons montré que chacun de ces sous-problèmes pouvait être exprimé à l'aide de l'algèbre linéaire lorsque l'on cible les applications de traitement du signal. Pourtant, l'expert en traitement du signal plaçant manuellement une application de traitement du signal utilise bien simultanément deux techniques. L'une est basée sur le partitionnement, et répond au problème de la granularité entre l'architecture ciblée et l'application. L'autre est basée sur la fusion de boucles, mais son utilisation va bien au-delà de sa définition "académique" : elle traite globalement la privatisation des tableaux, les communications et restructure ainsi totalement le code de l'application afin que celle-ci satisfasse l'ensemble des contraintes. L'approche du placement automatique présentée dans cette thèse consiste justement à décrire le placement en modèles permettant de construire une résolution concurrente. Pour celà, cette modélisation et cette résolution doivent reposer sur un langage de programmation avec contraintes apte et suffisament éprouvé pour exprimer et résoudre les différentes fonctionnalités du placement.

La partie suivante (III) présente la phase modélisation et s'appuie sur les techniques de l'état de l'art en parallélisation/placement. De par la nature même de notre approche pour la résolution du problème global de placement, il ne peut y avoir de présentation linéaire de cette modélisation. Nous donnons donc, tant que possible, les relations et références aux modèles connexes.

La partie (IV) présente les fonctionalités du système de placement développé. C'est la mise en œuvre informatique de la partie (III). Ceci a donné naissance à un système de placement de traitement du signal sur architecture parallèle. Nous montrons comment son implantation reflète totalement la modélisation.

La partie (V) présente des essais sur des exemples applicatifs qui permettent de raffiner les stratégies de résolution et les heuristiques utilisés pour valider l'approche concurrente.

Troisième partie

Modélisation

Introduction

Les chapitres qui suivent présentent la modélisation des différentes composantes du problème de placement. Chaque modèle est dérivé de l'état de l'art en matière de placement et parallélisation automatique. Cette dérivation est motivée par l'implantation de chaque modèle dans un langage de Programmation Logique avec Contraintes (PLC), ce qui fera l'objet de la partie [IV].

Chacune des composantes du placement est assimilé dans cette partie à un modèle représentant mathématiquement un concept physique : partitionnement, ordonnancement, communications, mémoire *etc...* Les modèles sont interconnectés par des relations assurant la cohérence globale des solutions. Ces liens apparaissent dans la modélisation sous la forme d'expressions utilisant des variables de plusieurs modèles. Ils seront rappelés en conclusion de cette partie.

Pour résoudre (cf. part. [IV]) par la suite efficacement le problème du placement automatique, il est nécessaire de conserver une concurrence potentielle entre les modèles à travers leurs interconnexions. Ceci est parfois difficile lorsqu'il s'agit de contraintes non-linéaires comportant des quantificateurs universels. Il faut donc procéder à des sur-approximations ou des spécialisations des modèles. Celles-ci sont bien souvent proches de celles réalisées par les programmeurs en traitement du signal, compte-tenu des caractéristiques applicatives.

Ainsi, la dérivation mathématique des modèles issus de l'état de l'art est soumise à deux facteurs : les caractéristiques applicatives du traitement du signal et le maintien de la concurrence afin de pouvoir implanter les différents modèles dans un langage de programmation par contraintes. La modélisation a donc été itérative, à plusieurs reprises les contraintes n'avaient pas de comportement satisfaisant et nécessitait une reformulation.

Voici les modèles considérés dans cette thèse :

Le partitionnement exprime la distribution des calculs et des données sur les processeurs (chap. 4, p. 78).

Les dépendances caractérisent les itérations accédant aux mêmes donnée (chap. 5, p. 96).

L'interphase donne les communications entre deux phases de calcul partitionnées (chap. 6, p. 112).

L'ordonnancement consiste à organiser l'exécution des traitement parallèles dans le temps (chap. 7, p. 130).

La mémoire définit une contrainte de capacité pour laquelle on est sûr de pouvoir calculer une allocation (chap. 8, p. 142).

L'architecture est en fait un ensemble de paramètres comme le nombre de processeur, la bande passante (chap. 9, p. 151).

Le signal temps réel est constitué de la latence et des contraintes d'entrées-sorties (chap. 9, p. 158).

Le chapitre qui suit, le modèle de partitionnement, joue un rôle majeur car il conditionne la granularité de la solution par rapport à l'architecture cible. Celle-ci dispose généralement d'un ensemble de processeurs et d'une taille de mémoire finie. Ceci justifie de partitionner les nids de boucles afin d'adapter leurs dimensions à celles de l'architecture cible.

Chapitre 4

Modèle de Partitionnement 3D

François Irigoin *(CRI/ENSMP) est à l'origine du modèle de partitionnement 3D [IT88b, ACIK93] et de son expression sous forme d'algèbre linéaire. À la lumière de ces travaux, nous*[a] *avons initialement comparé ce modèle aux partionnements obtenus lors de placements manuels effectués par les experts de Thomson-CSF. Cette étude comparative a mis en exergue la pertinence des distributions blocs/cycliques de calculs et de données pour les applications de traitement du signal, et comment ces partitionnements pourraient être implantés dans un langage de contraintes.*

[a] Il s'agit ici d'une collaboratoin étroite entre le CRI (François Irigoin et Corine Ancourt), le LCR/Thomson-CSF (Jean Jourdan), des experts de Thomson-CSF (Alain Demeure et Didier Rozonelli) et moi-même.

1 Introduction

L'utilité du partitionnement pour adapter l'exécution d'un nid de boucles à une machine parallèle n'est plus à démontrer. C'est à cet effet que nous utiliserons le partitionnement. Les applications de traitement du signal sont spécifiées comme des suites de nids de boucles parfaitement imbriqués. Chacun d'eux doit être placé sur une machine à mémoire distribuée dont les dimensions ne concordent pas forcément aux dimensions du nid de boucles. Le modèle de partitionnement 3D s'applique indépendament à chaque nid de boucles permettant d'exprimer la localité, la distribution des itérations sur les processeurs ainsi que dans le temps. Chaque dimension d'itération est alors décomposée en trois nouvelles dimensions. Dans ce chapitre, sont présentées la fonction, la définition et les propriétés des partitionnements considérés.

La section (§ 2, p. 79) exprime la motivation du partitionnement 3D, après un rappel des contraintes mathématiques représentant un partitionnement (§ 3, p. 81), nous donnons son utilisation pratique dans le contexte du placement d'applications du traitement du signal (§ 4, p. 90).

2 Fonction et nature du partitionnement

Le partitionnement a tout d'abord un rôle local, il considère chaque nid de boucles du graphe de flot de données de façon indépendante. Il s'agit d'adapter un nid de boucles aux dimensions de l'architecture (processeur/mémoire) en exprimant l'ensemble des itérations distribuées par processeur. Chaque ensemble ainsi composé est local à un processeur et exécuté séquentiellement. De ce fait, le partitionnement représente le parallélisme et la localité (par rapport à un processeur). D'un point de vue global, le partitionnement permet d'éclater les nids de boucles et d'entrelacer leur exécution[1]. Ceci permet de retarder des calculs pour privilégier par exemple la récupération de la mémoire ou l'exécution de traitements d'acquisition. Les dates d'exécution associées aux différentes partitions sont du ressort de l'ordonnancement global.

2.1 Objectifs du partitionnement

Comme il a été introduit, le partitionnement requiert trois composantes liées aux dimensions physiques du problème, définissant des ensembles et sous-ensembles d'itérations à attribuer aux ressources de l'architecture.

2.2 Plusieurs dimensions physiques à considérer

Pour un nid de boucles donné, un premier partitionnement permet de définir des ensembles d'itérations qui recevront, grâce à l'ordonnancement global, la même date d'exécution. Ces ensembles seront alors considérés comme des "boîtes noires" par l'ordonnancement global et permettent de prendre en compte le recouvrement des communications par les calculs. Un second partitionnement permet de scinder chaque ensemble en plusieurs sous-ensembles qui seront exécutés en parallèle, chaque sous-ensemble d'itérations étant séquentiel à un processeur.

En résumé le partitionnement nécessite 3 composantes[2], la **composante localité** (par rapport à un processeur), la **composante parallélisme**, enfin la composante permettant l'ordonnancement global, appelée par la suite **composante temporelle**. Si un choix de partitionnement constitue une solution partielle pour le parallélisme, la localité, et l'ordonnancement global il définit aussi, pour chacune de ces composantes, un niveau de granularité. Cette formulation du partionnement est en fait proche de celui considéré pour la définition d'HPF[ACIK93], où l'on considère la distribution d'un `template` de données. Ces trois dimensions peuvent être résumées ainsi :

1. La possibilité d'ordonnancer globalement les calculs de différents nids de boucles : La *"schedulabilité"*. Il peut être nécessaire, si la dimension de la machine est trop faible, de différer les

[1] Le partitionnement est réellement une technique utilisée par de nombreux ingénieurs plaçant les algorithmes de traitement du signal sur machines parallèles. De ce fait, le modèle de partitionnement représenté ici est l'image de la réalité

[2] Il a cependant été envisagé de considérer une dimension de plus pour la mémoire cache. Ceci n'est pas le cas dans les travaux présentés dans la thèse. Néanmoins il est d'un usage courant tant dans le milieu industriel que le milieu académique de partitionner pour tenir compte de cette ressource. L'étude de l'impact de cette nouvelle dimension du partitionnement pourra faire l'objet de travaux futurs.

calculs dans le temps. Le vecteur $\mathbf{c}$ représente l'ensemble des calculs effectués à la même date d'ordonnancement. De plus, cette composante autorise le recouvrement partiel ou total des communications par des blocs de calcul cycliques. Cette composante est appelée la composante temporelle.

2. Le *parallélisme*, directement associé à l'ensemble de processeurs disponibles. Le vecteur $\mathbf{p}$ représente, pour un date logique donnée (i.e. pour un $\mathbf{c}$ donné), l'ensemble des calculs effectués séquentiellement sur un processeur.

3. La localité, pouvant être aussi vue comme l'exécution *séquentielle* d'une partie des itérations sur un processeur. Le vecteur $\mathbf{l}$ représente une itération effectuée sur un processeur à une date logique donnée (i.e. pour un couple $(\mathbf{c}, \mathbf{p})$ donné.

2.3 Spécificités des partitionnements

L'architecture ciblée étant homogène à tous les niveaux (réseau d'interconnection unique, mêmes processeurs et mémoire distribuée uniformément), on cherche à modéliser - au travers du partitionnement - une distribution uniforme des itérations. L'uniformité a deux avantages :

Représentation en intention plutôt qu'en extension : Au lieu d'exprimer explicitement tous les éléments d'une partition, la représentation en intention permet de caractériser chaque élément par un de ses représentants. Si ce choix supprime des solutions de partitionnement, les côuts associée à la représentation et à son traitement ne sont pas dépendant de la taille des ensembles. Cela signifie que l'obtention d'une solution ne dépendra ni de la taille de l'architecture cible, ni du nombre d'itérations.

Simplicité de génération de code : La génération de code associée à une distribution de nid de boucles peut s'écrire à l'aide de boucles imbriquées, évitant les problèmes de contrôle.

Pour cela, les travaux sont orientés vers l'expression algébrique des ensembles partitionnés. Les fondements théoriques du partitionnement sont préalablement rappelés (cf. § 3), leur adaptation au placement de nids de boucles indépendants est présentée en second lieu (cf. § 4, p. 90).

2.4 Partitionnement de boucles indépendantes

Seul l'espace d'itération $\mathcal{I}$, qui est constitué de boucles indépendantes est à partitionner. Ne seront pas abordés les problèmes posés par les dépendances lecture/écriture pouvant apparaître dans un nid de boucles car cela sort du cadre applicatif pris en compte. Toutes les solutions de partitionnement sont envisageables. Cependant, les caractéristiques applicatives nous amènent à considérer uniquement les partitionnements pour lesquels les matrices sont diagonales. La primitive `distribute` du langage `HPF` comporte en fait la même restriction. Ceci simplifie la formalisation, la modélisation, l'implantation, le code généré et surtout le calcul des dépendances de flot de données (chap. 5).

Dans le cas où nous serions confrontés a des dépendances lecture-écriture, le modèle de partitionnement supporterait de contraindre la sélection des partitionnements légaux. D'autant plus que de telles contraintes permettraient de couper la combinatoire associée au choix des partitionnements.

3 Modèle mathématique de partitionnement d'un espace discret

Les techniques de partitionnement pour les problèmes de placement et de parallélisation ont fait l'objet de nombreux travaux [Iri87][IT88b][RS90][SD90][AL93b][Dio96][HCF97]. Après avoir présenté quelques rappels mathématiques en § 3.1, p. 81 et § 3.2, p. 82, nous rappelons en § 3.3, p. 83 et en § 3.4, p. 84 la formulation mathématique du partitionnement uniforme d'un espace d'itération telle qu'elle a été définie en [Iri87][IT88b]. Nous donnons alors la dualité entre partitionnement cyclique et bloc (§ 3.4.3, p. 85). Ceci nous permet de mieux comprendre comment le partitionnement se recombine § 3.5, p. 88 pour obtenir un partitionnement 3D semblable à celui utilisé pour la définition d'`HPF` [ACIK93] afin d'expliciter les contraintes qui caractérisent ce modèle.

Dans ce paragraphe, on va considérer que l'ensemble E est inclus dans le module $\mathbb{Z}^n$. Afin de décrire de manière rigoureuse la notion de partitionnement de $\mathbb{Z}^n$, nous allons plonger (de manière canonique) $\mathbb{Z}^n$ dans $\mathbb{R}^n$. L'idée consiste à générer des tuiles pour partitionner uniformément l'espace $\mathbb{R}^n$. Nous allons supposer que le "paveur" remplit son espace avec un pavé type $\mathcal{P}_0$ (que nous appellerons dans la suite partition originelle, ou partition origine) dont il dispose d'un stock illimité d'exemplaires.

3.1 Partitionnement d'un espace de dimension fini par un ensemble quelconque

Rappelons tout d'abord la notion classique de partitionnement d'un espace de dimension finie.

Définition 3.1 *Soit $\mathcal{P}_0$ un compact connexe inclus dans $\mathbb{R}^n$ et d'intérieur non vide (i.e. $\mathrm{Int}(\mathcal{P}_0) \neq \emptyset$). On dit que $\mathcal{P}_0$ induit un* partitionnement uniforme *de* partition origine *$\mathcal{P}_0$, s'il existe un sous-groupe $\mathcal{G}(\mathbb{R}^n)$ inclus dans le groupe des translations de $\mathbb{R}^n$ tel :*

$$\mathbb{R}^n = \bigcup_{\tau \in \mathcal{G}(\mathbb{R}^n)} \tau(\mathcal{P}_0) \tag{4.1}$$

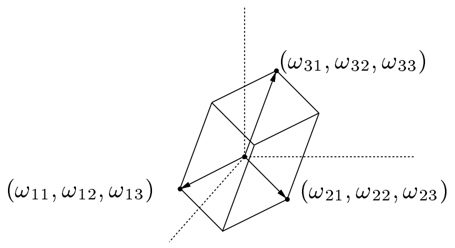
$$\forall \tau_i, \tau_j \in \mathcal{G}(\mathbb{R}^n), \qquad \tau_i(\mathrm{Int}(\mathcal{P}_0)) \cap \tau_j(\mathrm{Int}(\mathcal{P}_0)) \neq \emptyset \quad \Rightarrow \quad \tau_i(\mathcal{P}_0) = \tau_j(\mathcal{P}_0) \tag{4.2}$$

Remarque 3.2 *Contrairement à l'usage mathématique, un tel sous-ensemble $\tau(\mathcal{P}_0)$ sera appelé* partition. *Il est à noter qu'en algèbre classique, une partition de $\mathbb{R}^n$ est en fait une famille $(\mathcal{P}_i)$ satisfaisant l'équation (4.1) i.e., $\cup_i \mathcal{P}_i = \mathbb{R}^n$ mais aussi l'équation suivante :*

$$\forall i \neq j, \quad \mathcal{P}_i \cap \mathcal{P}_j = \emptyset$$

L'équation (4.1) signifie que l'on a recouvert tout l'espace à l'aide de "pavés" $\mathcal{P}_0$ et de ses translatés, l'équation (4.2) sous-entend que deux tuiles ne peuvent se recouvrir, et que les seuls points communs à deux partitions sont ceux de la frontière car $\mathcal{P}_0$ est compact.

Le type de partitionnement utilisé dans tout ce rapport sera induit par une partition origine de forme parallélépipédique. Nous allons donc rappeler brièvement comment construire un parallélipède, et rappeler quelques propriétés mathématiques utiles par la suite.



$$\Omega = \begin{pmatrix} \omega_{11} & \omega_{21} & \omega_{31} \\ \omega_{12} & \omega_{22} & \omega_{32} \\ \omega_{13} & \omega_{23} & \omega_{33} \end{pmatrix}$$

FIG. 4.1. – *Le parallélépipède associé à une matrice* Ω

3.2 Définition d'un parallélépidède dans un espace de dimension fini

Rappelons qu'un polyèdre convexe d'un espace affine de dimension, est un ensemble défini par une intersection finie de demi-espaces fermés. Un polyèdre est un polyèdre convexe compact et d'intérieur non vide.

Définition 3.3 *Soit* $(s_i)_{i=0}^n$ *une famille de* $(n+1)$ *points de l'espace affine* $\mathbb{R}^n$ *linéairement indépendants*[3]*. Le* parallélépipède *de sommets* $(s_i)_{i=0}^n$ *est alors l'ensemble :*

$$\left\{ s_0 + \sum_{i=1}^{n} \lambda_i \mathbf{s_0 s_i} \quad \textit{avec } \forall i,\ \lambda_i \in [0,1] \right\}$$

Remarque 3.4 *Un parallélépipède est un polyèdre.*

Parallélépipède origine : On suppose ici que le sommet s_0 est confondu avec l'origine de l'espace affine, et pour tout point s_i $(i \neq 0)$, on pose $\mathbf{u}_i = \mathbf{Os_i}$. Le parallélépipède ainsi défini et appelé par la suite le parallélépipède origine est alors entièrement caractérisé par la famille $(\mathbf{u}_1, ..., \mathbf{u}_i, ...\mathbf{u}_n)$ des n vecteurs libres.

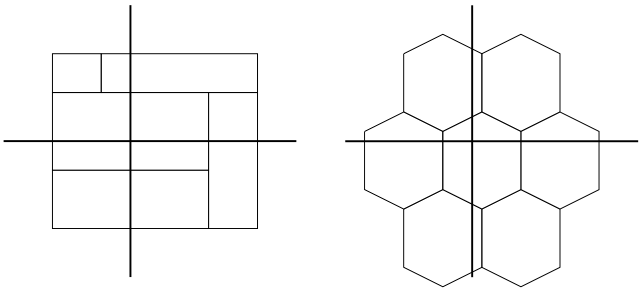
Matrice associée à un parallélépipède origine : Soit Ω la matrice dont la $i^{\text{ème}}$ colonne est composée du vecteur $\mathbf{u}_i$, *i.e.* $\Omega = (\mathbf{u}_i)_{i=1}^n$, on dira que la matrice Ω est la matrice associée[4].

Le partitionnement a un impact fondamental sur l'ensemble des contraintes de capacité de cette thèse (cf. chapitres 9, 8, 6). Le volume des parallélépipèdes [Ber90a,Ber90b,Str86] est l'outil mathématique qui permet d'exprimer ces contraintes de capacité.

Théorème 3.5 *Le volume d'un parallélépipède associé à une matrice* Ω *est égal au déterminant* $\det(\Omega)$ *de cette matrice (fig. 4.1).*

[3] *i.e.* les vecteurs $(\mathbf{s_i s_0})_{i=1}^n$ forment une base de l'espace vectoriel $\mathbb{R}^n$

[4] La matrice Ω est par construction une matrice carrée de taille $n \times n$ et de déterminant non nul. En effet, $\forall 1 \leq i \leq n,\ \ \mathbf{u}_i \in \mathbb{R}^n$ et $(\mathbf{u}_1, ..., \mathbf{u}_i, ...\mathbf{u}_n)$ forme une base de l'espace vectoriel de $\mathbb{R}^n$



Deux exemples de partitionnement interdits. Le partitionnement à base de rectangle n'est pas uniforme, tandis que les hexagones ne sont pas des parallépipèdes.

FIG. 4.2. – *Partitionnements non valides*

3.3 Partitionnement uniforme d'un espace de dimension finie par un parallélépipède

La fonction du partitionnement étant de distribuer un domaine d'itérations sur une architecture homogène, chaque processeur recevra exactement le même nombre d'itérations. Dans cette optique, l'interêt du partitionnement est de distribuer uniformément les itérations. En ce qui concerne les bornes des domaines d'itérations on distingue deux cas :

- le cas ou l'ensemble des parallélépipède recouvre exactement le domaine d'itérations, le partitionnement est dit exact.
- le cas ou les parallépipèdes ne sont pas de même nature aux bornes du domaine d'itération.

Les deux cas sont traités dans l'implantation du modèle.

Dans le cadre de notre approche, on impose certaines contraintes sur ces tuiles :

1. Les polyèdres décrivant les partitions sont des parallépipèdes.
2. Le même parallépipède est utilisé pour partitionner l'espace.

De ce fait, l'ensemble des partitions peuvent être obtenues par la définition d'une partition origine et de translations.

En utilisant la définition (3.1), on introduit donc un partitionnenement de l'espace affine de dimension n par un parallélépipède $\mathcal{P}_0$ origine associée à une matrice Ω.

Proposition 3.6 [Partitionnement uniforme induit par un parallélépipède] *Soit $\mathcal{P}_0$ un parallélépipède inclus dans $\mathbb{R}^n$ et dont un sommet coïncide avec l'origine. Soit Ω la matrice associée à $\mathcal{P}_0$. Soit $\mathcal{T}$ l'ensemble défini par : $\mathcal{T} = \{\mathbf{t} \mid \mathbf{t} = \Omega.\mathbf{v} \;\; avec\; \mathbf{v} \in \mathbb{Z}^n\}$. Alors :*

$$\mathbb{R}^n = \cup_{\mathbf{t}\in\mathcal{T}}(\mathcal{P}_0 + \mathbf{t})$$
$$\forall \mathbf{t}_1, \mathbf{t}_2 \in \mathcal{T} \qquad (\mathrm{Int}(\mathcal{P}_0) + \mathbf{t}_1) \;\cap\; (\mathrm{Int}(\mathcal{P}_0) + \mathbf{t}_2) \neq \emptyset \qquad \Longrightarrow \qquad \mathbf{t}_1 = \mathbf{t}_2$$

où $(\mathcal{P}_0 + \mathbf{t})$ représente le parallélépipède $\mathcal{P}_0$ translaté par le vecteur $\mathbf{t}$.

Cette proposition montre que l'on a bien défini un partitionnement (au sens de la définition (3.1)) de l'espace $\mathbb{R}^n$.

Démonstration – L'ensemble des translations possibles est donc le groupe isomorphe à $\mathcal{T}$. Comme $\mathbb{Z}^n$ est un $\mathbb{Z}$-module, c'est un groupe. De plus, il est immédiat de voir que deux parallélépipèdes $(\mathcal{P}_0 + \mathbf{t}_1)$ et $(\mathcal{P}_0 + \mathbf{t}_2)$ (où $\mathbf{t}_1, \mathbf{t}_2 \in \mathcal{T}$) n'ont de points communs que les points de frontière. En effet, les vecteurs de translations s'écrivent comme combinaisons linéaires à coefficients entiers des vecteurs associés aux sommets du parallélépipède origine $\mathcal{P}_0$.

3.4 Partitionnement uniforme de $\mathbb{Z}^n$

Dans ce paragraphe, nous allons transposer les notions précédentes définies dans l'espace affine $\mathbb{R}^n$ au module affine $\mathbb{Z}^n$ par simples restrictions des points de $\mathbb{R}^n$ ayant des coordonnées entières.

3.4.1 Parallélépipède discret origine

Soit $(\mathbf{u}_1, ...\mathbf{u}_i, ...\mathbf{u}_n)$ des n vecteurs libres de $\mathbb{Z}^n$. Alors le parallélépipède discret origine est défini comme l'ensemble $\mathcal{P}_0 \cap \mathbb{Z}^n$ où $\mathcal{P}_0$ est le parallélépipède (réel) de matrice $\Omega = (\mathbf{u}_i)_{i=1}^n$. Pour simplifier les notations, on le notera aussi par $\mathcal{P}_0$.

Définition 3.7 *Soit $(s_i)_{i=0}^n$ une famille de $(n+1)$ points du module affine $\mathbb{Z}^n$ linéairement indépendants. Le* parallélépipède *(discret) de sommets $(s_i)_{i=0}^n$ est alors l'ensemble :*

$$\left\{ x \in \mathbb{Z}^n \mid x = s_0 + \sum_{i=1}^{n} \lambda_i \mathbf{s_0 s_i} \quad avec\ \forall i,\ \lambda_i \in \mathbb{Q},\ et\ \lambda_i \mathbf{s_0 s_i} \in \mathbb{Z}^n \right\} \tag{4.3}$$

Proposition 3.8 *Soit $\mathcal{P}_0$ le parallélépipède origine associé à la matrice $\Omega = (\mathbf{u}_i)_{i=1,..,n}$ avec $\forall i,\ \ \mathbf{u}_i \in \mathbb{N}^n$. Alors pour tout point $x \in \mathcal{P}_0$, on a :*

$$\mathbf{0} \leq \Omega^{-1}.\mathbf{x} < \mathbf{1} \tag{4.4}$$

où $\mathbf{1}$ représente le vecteur dont toute les coordonnées sont égales à 1, $\mathbf{x} = \mathbf{Ox}$ et $\mathbf{v} \leq \mathbf{w}$ signifie que $\forall i$, on ait $v_i \leq w_i$.

Démonstration – Par définition (3.7), si $x \in \mathcal{P}_0$, alors il existe un vecteur de coordonnées $\mathbf{v} = (\lambda_1, ..., \lambda_n)^t$ avec $\lambda_i \in [0, 1[$, tel que $\mathbf{x} = \Omega.\mathbf{v}$, or

$$\begin{array}{rll} & \mathbf{0} \leq \mathbf{v} < \mathbf{1} & \\ \Leftrightarrow & \mathbf{0} \leq \mathbf{x} < \Omega\mathbf{1} & \text{car } \Omega \text{ est positive} \\ \Leftrightarrow & \mathbf{0} \leq \Omega^{-1}.\mathbf{x} < \mathbf{1} & \text{car } \det(\Omega) \neq 0 \end{array}$$

Attention, même si Ω est à coefficients entiers, comme elle est construite à partir de n vecteurs entiers de dimension n indépendants, les coefficients de son inverse ne seront pas nécessairement des entiers.

3.4.2 Partitionnement uniforme d'un espace discret induit par un parallélépipède

Définition 3.9 [Définition du partitionnement uniforme dans $\mathbb{Z}^n$] *Soit Ω une matrice carrée de taille $n \times n$ à coefficients entiers positifs et de déterminant non nul. Soit $\mathcal{P}_0$ le parallélépipède (discret) origine associé à Ω, i.e.*

$$\mathcal{P}_0 = \left\{ y \mid \mathbf{0} \leq \Omega^{-1}.\mathbf{y} < \mathbf{1} \right\}$$

et soit $\mathcal{T}$ l'ensemble des vecteurs de translations défini par

$$\mathcal{T} = \{\mathbf{t} = \Omega\mathbf{x} \quad \textit{avec } \mathbf{x} \in \mathbb{Z}^n\}$$

On dit que $(\mathcal{P}_0, \mathcal{T})$ forme un partitionnement uniforme *du module $\mathbb{Z}^n$.*

En utilisant la définition (3.9) et la proposition (3.8), on en déduit alors :

Proposition 3.10 *Sous les hypothèses précédentes on a :*

1. $\mathbb{Z}^n = \cup_{\mathbf{t}\in\mathcal{T}}(\mathcal{P}_0 + \mathbf{t})$ *et* $\forall \mathbf{t}_1 \neq \mathbf{t}_2$ *avec* $\mathbf{t}_1, \mathbf{t}_2 \in \mathcal{T}$ *on a :* $(\mathcal{P}_0 + \mathbf{t}_1) \cap (\mathcal{P}_0 + \mathbf{t}_2) = \emptyset$
2. $$\forall \mathbf{z} \in \mathbb{Z}^n, \quad \exists!(\mathbf{x}, \mathbf{y}) \ \ \textit{tel que} \ \ \mathbf{z} = \Omega\mathbf{x} + \mathbf{y} \ \ \textit{avec} \left\{ \begin{array}{l} \mathbf{0} \leq \Omega^{-1}.\mathbf{y} < \mathbf{1}, \\ \det(\Omega) \neq 0 \end{array} \right.$$

L'assertion (1) de la proposition (3.10) signifie que le partitionnement n'est rien d'autre qu'une "*partition*" au sens mathématique du terme (et non pas dans le sens donné dans ce document).

La définition de la partition origine et l'ensemble des translations possibles constituent une bijection entre l'espace $\mathbb{Z}^n$ et l'espace engendré par le couple de vecteurs $(\mathbf{t}, \mathbf{y})$ (cf. assertion (2) de la proposition (3.10)).

3.4.3 Dualité du partitionnement : cycles et blocs

Les différentes interprétations possibles des variables $\mathbf{x}$ et $\mathbf{y}$ permettent d'introduire un partitionnement supplémentaire : le partitionnement cyclique.

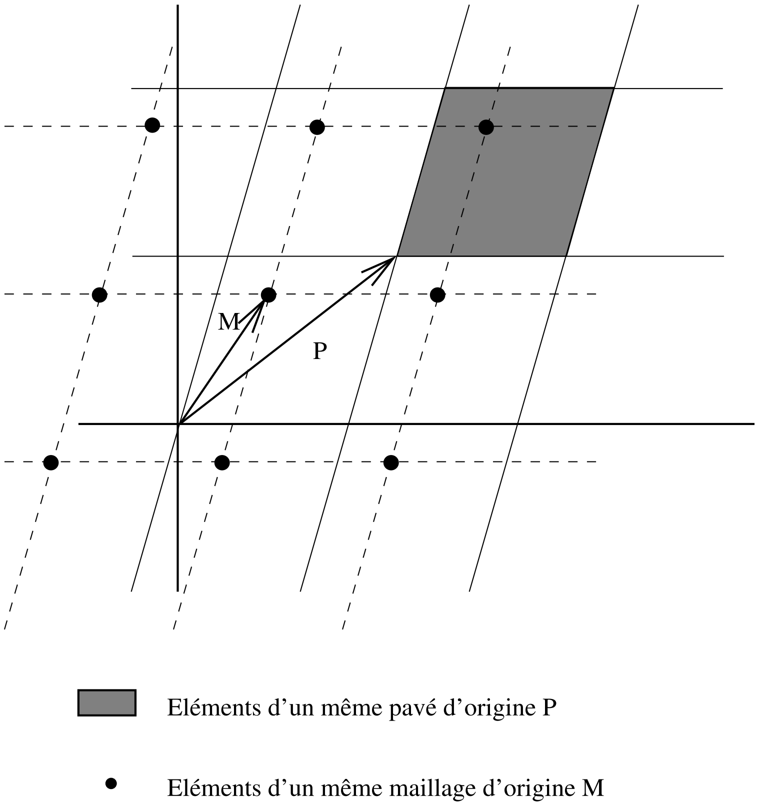
Partitionnement par blocs : Si $\mathbf{x}$ est considéré comme le représentant d'une partition, c'est le partitionnement par blocs. Le vecteur $\mathbf{y}$ donne alors le déplacement dans un bloc lorsque le vecteur $\mathbf{x}$ est fixe.

Partitionnement par cycles : Inversement, il est possible de considérer le vecteur $\mathbf{y}$ comme le représentant d'une partition, c'est le partitionnement par cycles. Le vecteur $\mathbf{x}$ donne alors le déplacement dans un cycle.

Relations d'équivalence associées aux partitionnements : Les notions de blocs et de cycles peuvent être interprétées à l'aide de classes d'équivalence. Ces classes d'équivalence structurent différement l'espace. Un bloc comme un cycle peut être considéré comme un représentant d'une classe d'équivalence.

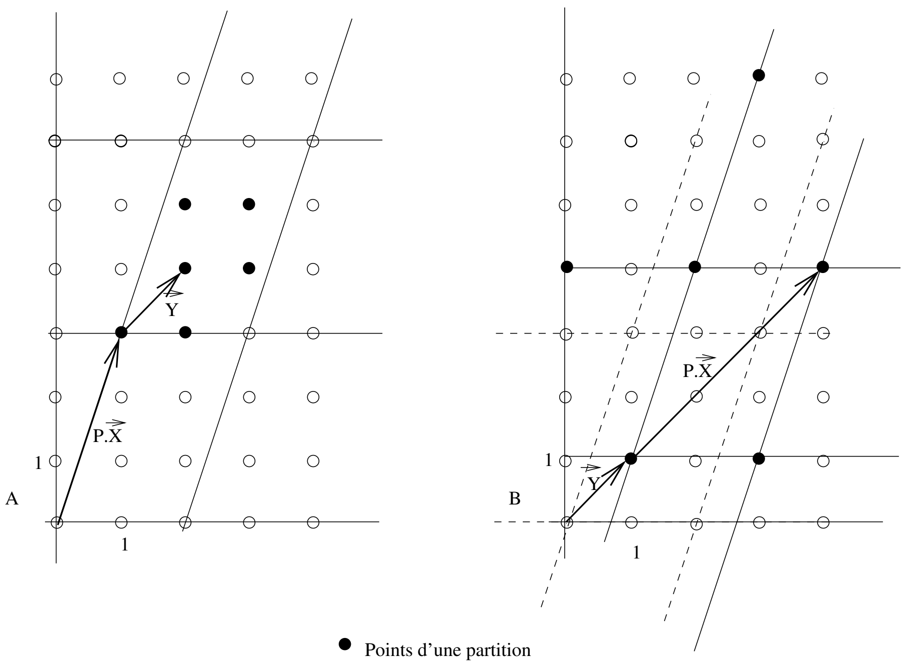
Soit deux points $\mathbf{z}$ et $\mathbf{z}'$, si $\mathbf{z} = \Omega\mathbf{x}+\mathbf{y}$ et $\mathbf{z}' = \Omega\mathbf{x}+\mathbf{y}'$ sous les contraintes préservant l'uniformité du partitionnement, alors $\mathbf{z}$ et $\mathbf{z}'$ appartiennent au même bloc (resp. au même cycle). On dira alors qu'ils sont équivalents au sens bloc (resp. au sens cycle).

Dans l'ensemble de cette présentation nous avons montré que le partitionnement d'un espace peut être caractérisé à l'aide d'un cycle ou d'un bloc représentant une classe d'équivalence. Il sera présenté par la suite comment ce bloc et ce cycle permet de dimensionner localement et globalement un ensemble de paramètres clefs du placement.



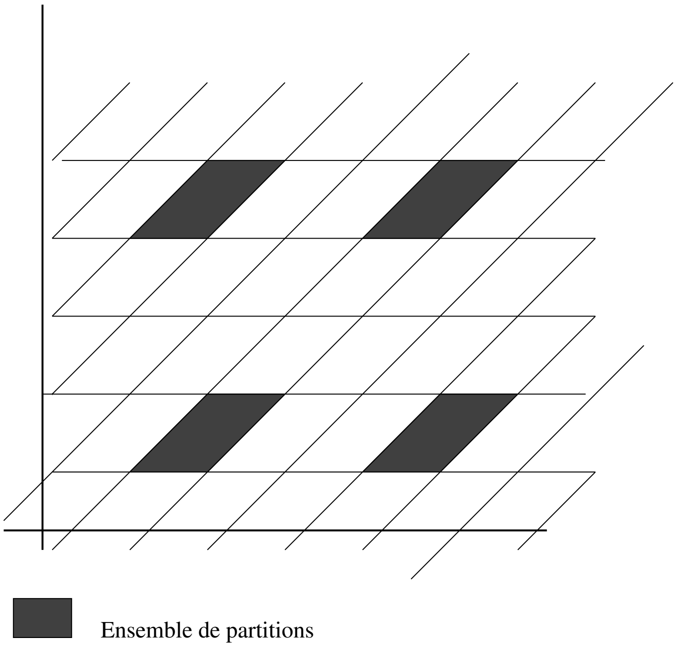
Un exemple de cycle et de bloc, la partition par bloc correspond au parallépipède, tandis que la partition par cycle est représenté par le treillis. De même que les différentes partitions, les différents cycles recouvrent totalement l'espace par translation.

FIG. 4.3. – *Cycles et blocs*



Dans les partitionnements par cycles (resp. par blocs), **Y** et $P.\mathbf{X}$ représente un élément dans un bloc et l'origine d'un cycle. Le vecteur $P.X$ représente la partition et l'origine de la partition.

FIG. 4.4. – *Sémantique des vecteurs*



La définition du partitionnement est réappliquée sur l'un des deux vecteurs. Un premier partitionnement engendre des blocs, un deuxième partitionnement permet de les regrouper par cycles. Par exemple, l'exécution des blocs en gris sera différée dans le temps et affecté à un même processeur. De nombreuses configurations sont donc permises augmentant considérablement la flexibilité du choix du partitionnements et des affectations.

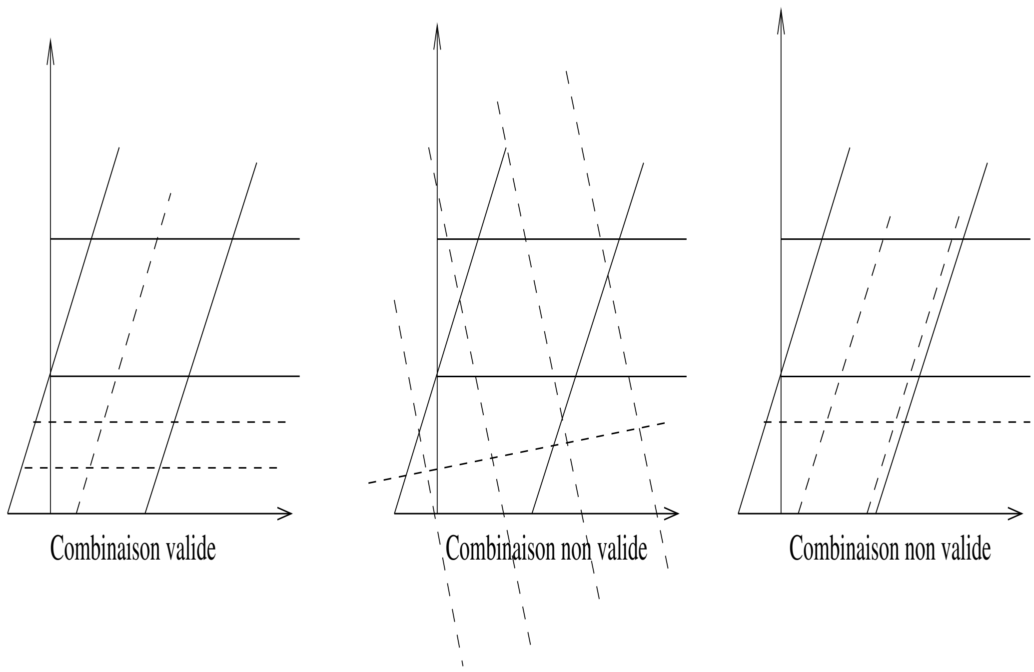
FIG. 4.5. – *Combinaison de partitionnements*

3.5 Combinaison de partitionnements

Il a précédement été expliqué en (§ 2, p. 79) qu'il était nécessaire de partitionner le domaine d'itérations en trois dimensions. A cet effet, la définition du partitionnement peut se réappliquer sur l'un ou l'autre des vecteurs $\mathbf{x}, \mathbf{y}$ engendrés par le premier partitionnement (fig. 4.5). Encore une fois, une condition nécessaire et suffisante doit être satisfaite afin d'obtenir un partitionnement uniforme. Nous présentons dans cette partie la généralisation du partitionnement.

À partir de la définition du partitionnement (§ 3.4, p. 84), deux cas sont possibles :
- réappliquer la définition du partitionnement à $\mathbf{x}$,
- ou réappliquer la définition du partitionnement à $\mathbf{y}$.

Envisageons donc les deux cas :
1. Si la définition du partitionnement (§ 3.4, p. 84) est réappliquée sur $\mathbf{y}$ on obtient la définition suivante : $\mathbf{z} = \Omega.\mathbf{x} + \Omega'.\mathbf{y_1} + \mathbf{y_2}$.
   La contrainte classique du partitionnement s'applique sur $\mathbf{y_2}$: $\mathbf{0} \leq \Omega'^{-1}.\mathbf{y_2} < \mathbf{1}$.
   De plus, la contrainte résultant du premier partitionnement se réécrit :
$$\mathbf{0} \leq \Omega^{-1}.(\Omega'.\mathbf{y_1} + \mathbf{y_2}) < \mathbf{1} \tag{4.5}$$



Le partitionnement obtenu après des combinaisons de divers partitionnements doit être basé sur des parallépipèdes identiques, ce qui contraint la matrice Λ.

FIG. 4.6. – *Combinaison de partitionnements valides ou non*

Enfin, il faut tenir compte de l'uniformité du partitionnement. Pour cela, le partitionnement local à $\mathbf{y}$ doit engendrer des tuiles parallépipédiques toutes identiques. Ainsi, la base Λ permettant ce deuxième partitionnement doit être orthonormée par rapport à la première base induite par la matrice Ω. D'autre part, $\Omega = \Omega'.\Lambda$, où Λ est inversible dans $\mathbb{Q}$.
Ainsi, la formulation du partitionnement devient :

$$\mathbf{z} = \Omega'.\Lambda\mathbf{x} + \Omega'.\mathbf{y_1} + \mathbf{y_2} \quad \text{avec} \quad \begin{cases} \mathbf{0} \leq \Omega'^{-1}.\mathbf{y_2} < \mathbf{1} \\ \mathbf{0} \leq \mathbf{y_1} + \Omega'^{-1}.\mathbf{y_2} < \Lambda.\mathbf{1} \end{cases}$$

par substitution de Ω dans l'inéquation (4.5), une telle substitution ne peut se faire que si Λ est diagonale, ce que nous supposerons par la suite.
Or, si on tient compte de $\mathbf{0} \leq \Omega'^{-1}.\mathbf{y_2} < \mathbf{1}$, cette dernière inégalité est équivalente à :

$$\mathbf{0} \leq \Lambda^{-1}.\mathbf{y_1} < \mathbf{1}$$

2. Dans le cas où l'on réapplique sur $\mathbf{x}$ la définition du partitionnement, on obtient:

$$\mathbf{z} = \Omega.(\Omega'.\mathbf{x_1} + \mathbf{x_2}) + \mathbf{y} \Leftrightarrow \mathbf{z} = \Omega.\Omega'.\mathbf{x_1} + \Omega.\mathbf{x_2} + \mathbf{y}$$

avec : $\mathbf{0} \leq \Omega'^{-1}.\mathbf{x_2} < \mathbf{1}$ et $\mathbf{0} \leq \Omega^{-1}.\mathbf{y} < \mathbf{1}$.

La formulation des deux cas de partitionnement traités est la même, désormais il est fait référence à une unique définition du partitionnement à ce niveau :

Définition 3.11 Les contraintes de partitionnements combinés *définissant une combinaison de partitionnements sont :*

$$\forall\ \mathbf{i} \in \mathbb{Z}^n \qquad \mathbf{i} = \Omega.\Lambda\mathbf{x} + \Omega.\mathbf{y} + \mathbf{z} \quad avec \begin{cases} \forall\ \mathbf{z},\ \mathbf{0} \leq \Omega^{-1}.\mathbf{z} < \mathbf{1}, \\ \forall\ \mathbf{y},\ \mathbf{0} \leq \Lambda^{-1}.\mathbf{y} < \mathbf{1}, \\ \det(\Omega) \neq 0 \\ \det(\Lambda) \neq 0 \ et\ \Lambda\ diagonale \end{cases}$$

4 Placement d'un nid de boucles à l'aide du partitionnement 3D

Cette section présente comment la notion de partitionnement 3D, dont l'expression est semblable à celle de la distribution des calculs dans HPF, s'adapte particulièrement au problème du placement sur une architecture donnée.

4.1 Partitionnement d'un espace d'itérations

Le placement d'un nid de boucle indépendant sur l'architecture est basé sur la formulation du partitionnement 3D, afin de considérer les dimensions physiques du problèmes. Le partitionnement d'un espace d'itérations $\mathcal{I}$ donné décompose l'ensemble associé $\mathcal{E}$ des itérations en sous-ensembles disjoints notés $\mathcal{P}_i$ formant une partition (au sens mathématique du terme) de $\mathcal{E}$.

Définition 4.1 *Soient s sous-ensembles notés $\mathcal{P}_i$, $0 \leq i < s$ d'un ensemble d'itérations $\mathcal{I}$, ces s sous-ensembles forment une* partition *(au sens mathématique du terme) de $\mathcal{E}$ si et seulement si :*

$$\mathcal{E} = \bigcup_{i=1}^{s} \mathcal{P}_i \tag{4.6}$$

$$\forall\ i,\ j,\ 1 \leq i,\ j < s,\ i \neq j, \qquad \mathcal{P}_i \cap \mathcal{P}_j = \emptyset \tag{4.7}$$

Remarque 4.2
- *L'équation (4.6) précise exactement que l'union des partitions recouvre totalement l'espace des itérations $\mathcal{I}$, tous les calculs seront donc distribués.*
- *Tandis que (4.7) interdit l'appartenance d'une itération à deux partitions, un calcul ne peut être dupliqué dans plusieurs partitions.*

La complexité du code à générer pouvant être très importante, on se limite à des modèles simples de partitionnement.

On ne traitera par la suite que le cas où l'espace $\mathcal{I}$ est recouvert par des parallépipèdes identiques (§ 3.4.2, p. 84), *i.e.*

1. Les polyèdres décrivant les partitions sont des parallépipèdes.
2. Le même parallépipède est utilisé pour partitionner l'espace.

On pourra alors parler de partitionnement uniforme. De ce fait, l'ensemble des partitions peuvent être obtenues par la définition d'une partition "origine" et de translations (3.10). Le domaine d'itérations à partitionner est entièrement défini par l'espace suivant :

$$\mathcal{I} = \{0,..,b_1 - 1\} \times \{0,..,b_2 - 1\} \times ... \times \{0,..,b_i - 1\} \times ... \times \{0,..,b_n - 1\}, \ \text{ où } 1 \leq i \leq n,\ b_i \in \mathbb{N}^\star$$

Remarque 4.3 $\forall i(i_1,..i_k,..i_n) \in \mathcal{I}$ *vérifie :* $\forall\ k,\ 0 \leq i_k < b_k,\ b_k \in \mathbb{N}^\star$.

La définition (3.11) restreinte au domaine $\mathcal{I}$ est alors :

Définition 4.4 Définition du partitionnement "3D" d'un domaine borné
Soit $\mathcal{I}$ le domaine inclus dans $\mathbb{Z}^n$ défini par : $\mathcal{I} = \{0,..,b_1-1\} \times \{0,..,b_2-1\} \times ... \times \{0,..,b_k-1\} \times ... \times \{0,..,b_n-1\}$, où $1 \leq k \leq n$, $b_k \in \mathbb{N}^\star$. Soit P (resp. L) une matrice carrée (resp. et diagonale) de taille $n \times n$ de déterminant non nul et à coefficients entiers (i.e. dans $\mathbb{N}$). Alors, pour tout point $i \in \mathcal{I}$, il existe un unique triplet (x,y,z) de points de $\mathcal{I}$ verifiant :

$$\boxed{\mathbf{i} = LP\mathbf{x} + L\mathbf{y} + \mathbf{z}}$$

$$\textit{avec} \begin{cases} \forall\ \mathbf{z},\ \mathbf{0} \leq L^{-1}.\mathbf{z} < \mathbf{1}, \\ \forall\ \mathbf{y},\ \mathbf{0} \leq P^{-1}.\mathbf{y} < \mathbf{1}, \\ \forall\ i \in \mathcal{I},\ \mathbf{0} \leq \mathbf{i} < \mathbf{b} \\ \det(L) \neq 0 \\ \det(P) \neq 0 \ \textit{et } P \textit{ diagonale} \end{cases}$$

où à tout point $s \in \mathcal{I}$ est associé le vecteur $\mathbf{s} = \mathbf{Os}$ et $\mathbf{b} = (b_1, b_2, ..., b_n)^t$. L'application qui à tout point i associe le triplet (x,y,z) sera appelée partitionnement 3D du domaine $\mathcal{I}$.

La simple formulation du partitionnement 3D ne suffit pas à exprimer la distribution des calculs, il faut aussi associer les dimensions physiques aux dimensions du partitionnements.

4.2 Possibilités d'affectations des partitions aux processeurs

Pour permettre de comprendre comment associer les composantes du partitionnement aux composantes physiques, intéressons-nous au cas simple du partitionnement bi-dimensionnel. Celui-ci met en jeu deux variables $\mathbf{x}, \mathbf{y}$, l'une représentant les partitions, l'autre un élément de la partie. Rappelons la forme générale[5] d'un partitionnement (simple) de l'espace $\mathcal{I}$:

$$\forall \mathbf{i} \in \mathcal{I},\quad \exists!(\mathbf{x}, \mathbf{y}) \ \text{ tel que } \ \mathbf{i} = L.\mathbf{x} + \mathbf{y} \ \text{ avec } \begin{cases} \mathbf{0} \leq L^{-1}.\mathbf{y} < \mathbf{1}, \\ \det(L) \neq 0 \end{cases} \tag{4.8}$$

L'affectation revient à associer aux vecteurs régissant le partitionnement les composantes physiques. Ainsi, le vecteur représentant les parties est affecté aux processeurs. Tandis que le vecteur indiquant un élément dans une partition est affecté à la dimension séquentielle. Nous en déduisons la partition placée sur un processeur, paramétré selon $\mathbf{p}$, et notée $\mathcal{P}(p)$:

- Dans le cas d'un partitionnement par blocs (§ 3.4.3, p. 85), le vecteur $\mathbf{x}$ est affecté au processeur p, le vecteur $\mathbf{y}$ est affecté à la mémoire l. L'ensemble des éléments résidant sur un processeur est :

$$\mathcal{P}(p) = \left\{ \mathbf{i} \in \mathcal{I} \mid \exists \mathbf{l},\ 0 \leq P^{-1}.\mathbf{l} < 1,\ \mathbf{i} = P.\mathbf{p} + \mathbf{l} \right\}$$

[5]Cette expression a été introduite dans $\mathbb{Z}^n$ (2), proposition (3.10).

Le partitionnement par bloc est le plus utilisé [Iri87, IT88b, RS90, SD90, AL93b, Dio96, HCF97]. En effet, il permet de conserver le voisinage des données : si deux itérations successives accèdent à une même donnée, celles-ci devront appartenir à une même partie disposée sur un même processeur.

- Dans le cas d'un partitionnement par cycles (§ 3.4.3, p. 85), le vecteur $\mathbf{y}$ est affecté au processeur p, le vecteur $\mathbf{x}$ est affecté à la mémoire l.

$$\mathcal{P}(p) = \left\{ \mathbf{i} \in \mathcal{I} \mid \exists \mathbf{p},\ 0 \leq \mathbf{l} < P^{-1}.up(\mathbf{i}),\ \mathbf{i} = P.\mathbf{l} + \mathbf{p} \right\}$$

On trouve quelques exemples d'application du partitionnement cyclique []. Il présente l'interêt de garantir l'équilibrage de charge, même lorsque le domaine d'itérations est polyèdrique.

Le choix d'une distribution par cycles ou par blocs revient donc à choisir l'affectation des variables physiques (la localité l et le parallélisme p) aux dimensions logiques du partitionnement.

4.2.1 Les différents types de partitionnement 3D possibles

Comme déjà présenté, le partitionnement 3D est plus complexe, il permet de combiner *cycles* et *blocs*. Ce type de partitionnement est utilisé par exemple dans la définition d'`HPF`[ACIK93] de la primitive `DISTRIBUTE` qui permet de distribuer les éléments du `TEMPLATE`[KLS+94] sur une grille de processeurs virtuels. Les notions de *cycle*, de *bloc*, ou de combinaisons $cycles - blocs$ sont confondues dans les classes que nous représentons ci-dessous. En effet, certains partitionnements peuvent être associés à diverses combinaisons possibles. Afin de ne pas construire inutilement plusieurs fois le même partitionnement, le système suivant de classification est considéré : À partir de la définition (4.4), deux vecteurs représentent toujours une partition, tandis que le troisième vecteur donne un élément dans la partition. De ce fait, l'interprétation des variables est assujetie à trois permutations possibles.

- Le couple de vecteurs ($\mathbf{x}$,$\mathbf{y}$) représente une partition, tandis que $\mathbf{z}$ représente un élément d'une partition (Ceci revient à considérer une combinaison de partitionnement bloc/bloc).
- Le couple de vecteurs ($\mathbf{z}$,$\mathbf{y}$) représente une partition, tandis que $\mathbf{x}$ représente un élément d'une partition (Ceci revient à considérer une combinaison de partitionnement cycle/cycle).
- Le couple de ($\mathbf{x}$,$\mathbf{z}$) représente une partition, tandis que $\mathbf{y}$ est un élément d'une partition. (Ceci revient à considérer une combinaison de partitionnement cycle/bloc ou bloc/cycle).

4.2.2 Affectation des partitions 3D aux processeurs

Les éléments d'une partition 3D ne peuvent être affectés sur plusieurs processeurs puisque l'on ne considère pas les duplications de calcul. Par conséquent le couple de vecteurs associé à une partition doit comporter un vecteur associé aux processeurs p (pour cpu). De plus, la deuxième composante de ce couple doit être affectée à la dimension datée c. Cette dernière dimension représente des ensembles d'itérations pouvant être différés dans le temps pour diverses raisons. En fait, l doit être constamment affecté au vecteur représentant un élément dans une partition. Considérant les vecteurs $\mathbf{x}, \mathbf{y}, \mathbf{z}$ de la définition (3.11), à chaque type de partitionnement énuméré en (4.2.1), correspond mathématiquement à $2 \times 1 = 2$ possibilités d'affectation, mais certaines n'ont aucune signification physique.

$$\mathbf{i} = LP\mathbf{x} + L\mathbf{y} + \mathbf{z} \quad \text{avec} \begin{cases} \forall\ \mathbf{z},\ \mathbf{0} \leq L^{-1}.\mathbf{z} < \mathbf{1}, \\ \forall\ \mathbf{y},\ \mathbf{0} \leq P^{-1}.\mathbf{y} < \mathbf{1}, \\ \forall\ i \in \mathcal{I},\ \mathbf{0} \leq \mathbf{i} < up(\mathbf{i}) \\ \det(L) \neq 0 \\ \det(P) \neq 0 \text{ et diagonale} \end{cases}$$

Permutation de (c, l, p) et significations physiques

(x, y, z)	type				Remarques
(c, p, l)	bloc/bloc	Grand bloc ← temps	Petit bloc ← processeur	Point ← mémoire	modèle acceptable
(c, l, p)	bloc/cycle	Gros bloc ← temps	Origine cycle ← processeur	Point ← mémoire	modèle acceptable
(p, c, l)	bloc/bloc	Grand bloc ← processeur	Petit bloc ← temps	Point ← mémoire	proc. ∞
(p, l, c)	bloc/cycle	Gros bloc ← processeur	Origine cycle ← temps	Point ← mémoire	proc. ∞
(l, c, p)	cycle/cycle	Origine grand cycle ← temps	Origine petit cycle ← processeur	Point ← mémoire	mémoire ∞
(l, p, c)	cycle/cycle	Origine grand cycle ← processeur	Origine petit cycle ← temps	Point ← mémoire	mémoire ∞

Ainsi, si l (resp. p) est affecté à $\mathbf{x}$, alors l, (resp. p) est borné par $L^{-1}.P^{-1}.up(\mathbf{i}) + \epsilon$. Compte tenu de la définition du nid de boucles (comportant une dimension infinie) les ressources nécessaires – taille mémoire, nombre de processeurs – sont infinies. Seules les affectations blocs-cycliques (c, l, p) et (c, p, l) sont à considérer :

$$\mathcal{P}_{(\mathbf{p},\mathbf{c})} = \left\{\mathbf{i} \in \mathcal{I} \mid \exists \mathbf{l},\ \mathbf{0} \leq L^{-1}.\mathbf{l} < \mathbf{1},\ \mathbf{i} = L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}\right\} \tag{4.9}$$

L'équation de ce partitionnement permet d'exprimer la sémantique de la primitive `DISTRIBUTE` d'HPF[Coe94][ACIK93].

$$\mathcal{P}_{(\mathbf{p},\mathbf{c})} = \left\{\mathbf{i} \in \mathcal{I} \mid \exists \mathbf{l},\ \mathbf{0} \leq P^{-1}.\mathbf{l} < \mathbf{1},\ \mathbf{i} = L.P.\mathbf{c} + L.\mathbf{l} + \mathbf{p}\right\} \tag{4.10}$$

<u>Démonstration</u> – Supposons en effet, qu'il existe une dimension de $\mathbf{c}$ qui soit infinie, *i.e.* $\exists j,\ \ c_j$ est non borné, alors si $\mathbf{c}$ est affecté soit à la variable $\mathbf{z}$ soit à la variable $\mathbf{y}$, on en déduit que les contraintes du modèle ne sont pas satisfaites sauf si le nombre de processeurs ou la mémoire est infini. Comme dans le cadre applicatif qui nous intéresse nous avons une dimension temporelle infinie, seul les modèles $(\mathbf{c}, \mathbf{p}, \mathbf{l})$ et $(\mathbf{c}, \mathbf{l}, \mathbf{p})$ seront à prendre en compte. □

4.2.3 Choix d'une affectation

Si la dimension infinie coupe un certain nombre de choix de partitionnements, il reste encore deux alternatives qui représente au sens PLC de réelles disjonctions. Lorsque l'on compare ce problème avec celui de la distribution des données en HPF, le langage HPF propose une distribution d'un TEMPLATE représentant les données. Alors que l'on propose un partitionnement qui représente une distribution des itérations. Comme l'on considère des applications à assignation unique, ces deux concepts sont équivalents. Il permet de jouer sur la localité et la cyclicité, d'autre part on retombe sur des solutions dites cycliques lorsque la matrice P est l'identité ou blocs lorsque c'est la matrice L qui est l'identité.

Une solution de partitionnement pour chaque nid de boucles correspond donc à l'instantiation de la matrice P et de la matrice L.

4.3 Distribution des données impliquée par le partitionnement

Un processeur doit disposer de la totalité des données dont il a besoin pour exécuter la partie de code qui lui est attribuée. Aussi, est-il nécessaire de pouvoir représenter l'ensemble des données référencées par une partition. Nous utilisons la définition des régions de tableaux [Tri84] [CI96] à cette fin. Si à chaque itération extra-procédurale i correspond une région de donnée $\mathcal{R}(\mathbf{i})$, nous pouvons construire la région référencée par une partition $\mathcal{P}(p)$. Au niveau de la thèse, nous utiliserons les vocabulaires d'ensembles d'éléments de tableaux ou de régions.

4.3.1 Distribution au sein d'un bloc temporel

Soit $R(\mathbf{i})$ l'ensemble des données référencées par l'itération $R(\mathbf{i})$.

$$\mathcal{R}(\mathcal{P}_{(0,0)}) = \bigcup_{\mathbf{i}\in\ \mathcal{P}_{(0,0)}} \mathcal{R}(\mathbf{i}) \tag{4.11}$$

$$\forall(\mathbf{p},\mathbf{c}),\quad \mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \bigcup_{\mathbf{i}\in\ \tau_{(\mathbf{c},\mathbf{p})}(\mathcal{P}_{(0,0)})} \mathcal{R}(\mathbf{i}) \tag{4.12}$$

La translation d'un bloc par rapport au vecteurs $\mathbf{c}$ et $\mathbf{p}$ est ici représenté par τ. D'après la définition des ensembles d'éléments de tableaux référencés (chap. 2, § 3.1.4, p. 48) par les macro-instructions, on peut écrire la région référencée par une partition comme :

$$\mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \bigcup_{i\in\ \mathcal{P}_{(\mathbf{c},\mathbf{p})}} \{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.\mathbf{i} + M.\mathbf{m}\}$$

ou bien d'après (4.12)

$$\mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \bigcup_{i\in\ \tau_{(\mathbf{c},\mathbf{p})}\mathcal{P}_{(0,0)}} \{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.\mathbf{i} + M.\mathbf{m}\}$$

donc

$$\mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \Big\{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{l},\ 0 \leq L^{-1}.\mathbf{l} < 1\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.\tau_{(\mathbf{c},\mathbf{p})}(\mathbf{l}) + M.\mathbf{m}\Big\} \tag{4.13}$$

Si l'onm choisi l'affectation $(\mathbf{c},\mathbf{p},\mathbf{l})$, cette union d'ensembles peut donc s'écrire directement, d'après les équations (4.9) :

$$\mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \Big\{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{l},\ 0 \leq L^{-1}.\mathbf{l} < 1\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.(L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}) + M.\mathbf{m}\Big\} \tag{4.14}$$

Dans le cas de l'affectation $(\mathbf{c},\mathbf{l},\mathbf{p})$, celle ci s'exprime :

$$\mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \Big\{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{l},\ 0 \leq P^{-1}.l < 1\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.(L.P.\mathbf{c} + L.\mathbf{l} + \mathbf{p}) + M.\mathbf{m}\Big\} \tag{4.15}$$

4.3.2 Distribution temporelle des données

De même, il est possible de définir les ensembles d'éléments de tableaux référencées par une partition temporelle :

$$\mathcal{R}(\mathcal{P}_{\mathbf{c}}) = \{\varphi \mid \varphi \in \Phi,\ \exists(l,\ p),\ 0\ \leq\ L^{-1}.\mathbf{l}\ <\ 1,\ 0\ \leq\ P^{-1}.\mathbf{p}\ <\ 1$$
$$\exists \mathbf{m} \in \mathcal{M},\ \varphi = I.(L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}) + M.\mathbf{m}\}$$

Si l'on recompose le partitionnement au niveau des blocs distribués sur les processeurs, on a une vue plus simple de la distribution temporelle des données, qui pourra servir pour le calcul des dépendances entre blocs de calculs de type indexés par $\mathbf{c}$. Il faut poser $\mathbf{s}\ =\ L.\mathbf{l}+\mathbf{p}$, on a alors $0\ \leq\ (L.P)^{-1}.\mathbf{s}\ <\ \mathbf{1}$

$$\mathcal{R}(\mathcal{P}_{\mathbf{c}}) = \left\{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{s},\ 0\ \leq\ (L.P)^{-1}.s\ <\ 1,\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.(L.P.\mathbf{c} + \mathbf{s}) + M.\mathbf{m}\right\} \tag{4.16}$$

4.3.3 Utilisation de la distribution des données

La distribution des données à un impact sur plusieurs modèles et elle permet de définir plusieurs relations dans la modélisation du problème global :

Le calcul des dépendances : Le calcul des dépendances est basé sur l'intersection des ensembles d'éléments de tableaux lus et écrits par deux taches différentes du programme.

La taille des allocations mémoire : A l'exécution d'une partition, correspond un ensemble de données préalablement allouées et disponibles en lecture, ainsi qu'un emplacement prévu pour les écritures de la partition.

Les communications : Après l'exécution d'une partition, une partie des données seront redistribuées vers les autres processeurs et une autre partie restera sur le même processeur. La taille des données à communiquer ainsi que la synthèse des instructions de communication nécessitent de connaitre l'emplacement des données sur les processeurs.

Recouvrement calculs/communications : Cette taille de données à communiquer intervient aussi dans le dimensionnement des communications et donc dans la possibilité de les recouvrir. C'est donc un paramètre essentiel pouvant infléchir la latence ou les acquisitions.

La gestion de la mémoire locale : Le voisinage des données est très important pour la mémoire locale. Ainsi, les données devant être réutilisées doivent être situées dans la mémoire locale.

5 Conclusion

Après avoir rappelé la formulation mathématique du partitionnement, permettant d'expliciter les contraintes nécessaires à considérer, nous avons donné quels était les choix de distribution pertinents pour le problème du placement. Encore une fois, nous soulignerons l'interêt d'avoir une représentation concise en intention de la distribution des données et des calculs. Cette représentation permet de manipuler un nombre illimité d'itérations à différents niveaux de granularité sans que cela nécessite une structure de donnée importante. Ceci reste valable pour les processeurs et les éléments de tableaux. Nous détaillerons l'utilisation de distributions de données pour les modèles capacitifs de mémoire (chap. 8), de communications (chap. 6) et la relation de dépendance de flot (chap. 5).

Chapitre 5

Caractérisation des dépendances

La caractérisation des dépendances de flot est le fruit d'une collaboration et d'une investigation itérative. En effet, Denis Barthou *et* François Irigoin *(CRI/ENSMP) ont initialement proposé de considérer les sommets du polytope de dépendances pour calculer l'ordonnancement événementiel. Après avoir constaté que les sommets, grâce aux caractéristiques applicatives, pouvaient être calculés à partir de polygones et des ensembles d'élements de tableaux référencés, j'ai mené divers essais infructueux pour calculer automatiquement ces sommets en Programmation par Contraintes à partir du partitionnement. Malgré les caractéristiques applicatives simplifiant les équations, il a été constaté qu'une forme symbolique des sommets serait plus appropriée pour concerver le caractère concurrent de la modélisation.* Juliette Mattioli *et* Bertrand Jeannet *(LCR/Thomson-CSF) ont alors permis la réalisation de ce modèle en calculant une forme explicite des sommets des polygones de dépendance. En revanche, la caractérisation des dépendances en lecture résulte d'une adaptation de techniques connues aux caractéristiques applicatives.*

1 Introduction

Nous donnons dans ce chapitre, une caractérisation des dépendances de flots existant entre deux nids de boucles ainsi que des dépendances en lecture présentent au sein d'un nid de boucles.

La relation de dépendances de flots (notée par la suite $D(\mathbf{i}^w,\ \mathbf{i}^r)$) est fondamentale pour la suite du travail. De nombreux modèles dépendent directement de la caractérisation de la dépendance de flot, et les dépendances permettent d'établir des arcs entre ces modèles. Les modèles d'ordonnancement, de communications, de mémoire et les heuristiques exploitent cette relation.

Etant donné le nombre de dépendances de flots entre paires d'itérations, une représentation en intention est ici encore requise. Compte tenu de la classe applicative et contrairement aux restrictions considérées en [DKR91, DR92, DR94c, DR94a], nous sommes dans le cas où les dépendances ne peuvent pas représentées par un vecteur constant mais dans celui où elles sont représentées par un ensemble de vecteurs appartenant à un polyèdre. Dans ce cas, la taille de la représentation dépend uniquement du nombre de sommets à manipuler. En revanche, le coût pour calculer les sommets de l'enveloppe convexe d'un polyèdre définit par un ensemble d'inéquations, est important. Cela nécessite de transformer l'ensemble des contraintes de flot en un système générateur. Une fois l'ensemble des sommets calculés dans $\mathbb{Q}$, il faut trouver les sommets dans $\mathbb{N}$. Si cette méthode reste valable lorsque l'approche de placement automatique est séquentielle, dans le contexte d'une approche concurrente, les dépendances et le partitionnement sont très liés. Le nombre et la position des sommets de ce polyèdre de dépendance variant dynamiquement en fonction du partitionnement, et donc lors de la phase de résolution. Une forme paramétrée de ces sommets est donc requise.

Le calcul des dépendances en lecture permet de caractériser la localité des données pour un nid de boucle. Il est donc très important pour guider la recherche de solutions de partitionnements. Si ces dépendances n'apparaissent pas en temps que modèles[1], elles sont utilisées en tant qu'heuristiques pour sélectionner les partitionnements qui minimisent allocations et communications.

Ce chapitre explique comment les dépendances de flot sont dégradées (§ 2, p. 97) et approximées pour les applications de traitements du signal, même si les accès contiennent des modulos fixés (§ 3, p. 103). Enfin, les dépendances lecture-lecture, fondamentales en traitement du signal sont étudiées (§ 4, p. 106).

2 Dépendances entre blocs séquentiels

Considérons deux nids de boucles, dont les vecteurs d'itérations sont respectivement $\mathbf{i}^w$ et $\mathbf{i}^r$ (avec $r \neq s$). Supposons qu'il y ait dépendance de flot entre les opérations des deux nids de boucles (cf. chapitre 1, § 2, p. 23). La relation de dépendances $D(\mathbf{i}^w, \mathbf{i}^r)$ entre $\mathbf{i}^w$ et $\mathbf{i}^r$ est alors définie par l'intersection des ensembles d'éléments de tableaux référencés en lecture et en écriture :

$$D(\mathbf{i}^w, \mathbf{i}^r) \Leftrightarrow \mathcal{R}^w(i_w) \ \cap \ \mathcal{R}^r(i_r) \ \neq \ \emptyset$$

Pour permettre de caractériser les dépendances, il est à présent nécessaire de rappeler l'expression des accès aux tableaux (chap. 2,§ 3.1.4, p. 48) :

$$\mathcal{R}(\mathbf{i}) = \{\varphi | \varphi \in \Phi, \ \exists \mathbf{m} \in \mathcal{M}, \ \varphi = I.\mathbf{i} + M.\mathbf{m} \ mod \ \mathbf{k}\} \tag{5.1}$$

Si l'on réecrit l'équation des dépendances en utilisant cette formulation des ensembles d'éléments de tableaux référencés, on obtient :

[1]bien qu'elles puissent servir à établir un modèle d'antémémoire

$$D(\mathbf{i}^w,\mathbf{i}^r) = \left(\begin{array}{c} \left[\left(\begin{array}{c} \{\boldsymbol{\varphi} \mid \exists \mathbf{m}^w,\ \boldsymbol{\varphi} = I^w\mathbf{i}^w + M^w\mathbf{m}^w,\ \mathbf{0} \leq \mathbf{m}^w\ <\ up(\mathbf{m}^w)\} \\ \cap \\ \{\boldsymbol{\varphi} \mid \exists \mathbf{m}^r,\ \boldsymbol{\varphi} = I^r\mathbf{i}^r + M^r\mathbf{m}^r,\ \mathbf{0} \leq \mathbf{m}^r < up(\mathbf{m}^r)\} \end{array} \right) \neq \emptyset \right] \\ \wedge \\ (\mathbf{0} \leq \mathbf{i}^w < up(\mathbf{i}^w)) \\ \wedge \\ (\mathbf{0} \leq \mathbf{i}^r < up(\mathbf{i}^r)) \end{array} \right) \tag{5.2}$$

Il faut alors tenir compte du partitionnement des nids de boucles (chap. 2, § 4.2.2) et de la distribution des données (chap. 4, § 4.3.1, p. 94) pour pouvoir représenter les dépendances. Nous tiendrons compte des caractéristiques applicatives (chap. 2,§ 6.1, p. 67) pour calculer symboliquement les sommets des polyèdres dans $\mathbb{N}$. D'une part, les fonctions d'accès aux données sont linéairement indépendantes. D'autres part, les matrices de partitionnement sont diagonales. L'ensemble des couples $(\mathbf{c}^w,\ \mathbf{c}^r)$ de blocs satisfiant le prédicat $\mathcal{D}_i(\mathbf{c}^w,\ \mathbf{c}^r)$ peut donc être défini par un simple produit cartésien.

Propriété 2.1 *Dans le cas des caractéristiques applicatives, si c^w_m et c^r_n sont deux composantes des vecteurs $\mathbf{c}^w$ et $\mathbf{c}^r$, et k est une dimension de tableau. Les boucles m et n sont associées à la dimension k du tableau. L'ensemble des couples de blocs dépendant est définit par le prédicat $\mathcal{D}$:*

$$\{(c^w,c^r) \mid \mathcal{D}(c^w,\ c^r)\} \ = \ \bigotimes_k \{(c^w_m,c^r_n) \mid \mathcal{D}_k(c^w_m,\ c^r_n)\} \tag{5.3}$$

où la $k^{\text{ème}}$ relation prend la forme :

$$\mathcal{D}_k(c^w_m,\ c^r_n) \ \Rightarrow\ \mathcal{R}^w_k(c^w_m)\ \cap\ \mathcal{R}^r_k(c^r_n)\ \neq\ \emptyset$$

Si nous considérons la distribution des données telle qu'elle a été formulée en (chap. 4, § 4.3.2, p. 95), la relation $\mathcal{D}_k$ s'écrit donc à partir des composantes des matrices P^r, P^w, L^r, L^w et des composantes des vecteurs $\mathbf{c}^r$, $\mathbf{c}^w$:

$$\begin{array}{rcl} \mathcal{D}_k(c^w_m,\ c^r_q) & \Rightarrow & \exists (m^w,m^r) \in \mathcal{M}_s \otimes \mathcal{M}_t, \\ & & \exists\ s_m,\ (L^w_{k,m}.P^w_{k,m})^{-1}.s^w_m, \\ & & \exists\ s^r_q,\ (L^r_{k,q}.P^r_{k,q})^{-1}.s^r_q, \\ & & I^w_{k,m}.(L^w_{k,m}.P^w_{k,m}.c^w_p + s^w_m) + M^w_{k,s}.m^w = I^r_{k,q}.(L^r_{k,q}.P^r_{k,q}.c^r_q + s^r_q) + M^r_{k,t}.m^r \end{array} \tag{5.4}$$

où k,m,s,q,t donnent les composantes des matrices I^w I^r M^w et M^r (chap. 2,§ 3.1.4,p. 48).

L'avantage des caractéristiques applicatives (chap. 2,§ 3.1.5,p. 49) ainsi que des partitionnements à l'aide de matrice diagonales permet de ne considérer uniquement des couples d'itérateurs partitionnés. On raisonne alors indépendament dans le plan formé par ces deux itérateurs. Le problème est donc à déterminer pour chaque plan k (correspondant à la $k^{\text{ème}}$ dimension de tableau), le polygone $\mathcal{P}_k$ surapproximant l'ensemble des blocs dépendants. Ce polygone varie dynamiquement car il dépend du partitionnement (par l'intermédiaire des matrices diagonales L^w, P^w, L^r et P^r). Seul un calcul fonctionnel des sommets, rendu possible grâce aux restrictions applicatives – notamment grâce aux fonctions d'accès linéairement indépendantes et aux matrices de partitionnement diagonales – permet de maintenir la concurrence entre les différents modèles de placement.

Calculer fonctionellement l'ensemble des sommets du polygone nécessite d'éliminer les quantificateurs $\exists$ et les variables associées de l'équation (5.4). C'est en fait une exécution manuelle de l'algorithme d'élimination de Fourier-Mortzkin [Fou27][2]. La deuxième étape consiste à calculer les intersections des droites dans $\mathbb{Q}^2$ et à déterminer les sommets entiers les plus proches. C'est à dire procéder manuellement à l'exécution de l'algorithme de Chernikova [3] [Che65] et des coupes de Gomory[4] [Gom63] dans le plan.

2.1 Utilisation des dépendances de flot

Pour comprendre l'importance de l'utilisation des dépendances, tant au niveau de la mémoire, que de l'ordonnancement ou que des communications, il est nécessaire d'expliciter l'utilité de cette représentation, en tant que lien entre modèles. Soient deux types d'opérations T^r et T^w. Les opérations T^r lisent les données écrites par T^w. Voici les principales utilisations des dépendances :

Légalité de l'ordonnancement global : Les dépendances permettent de connaître l'ensemble **suffisant** des blocs de la tâche T^w permettent d'effectuer un bloc donné de la tâche T^r. C'est l'utilisation classique de la dépendance de flot.

Définir le coût de calcul des communications : Le calcul de l'ensemble des blocs de la tâche T^r devant utiliser les données produites par un bloc de la tâche T^w permet de déterminer et d'approximer la durée des communications.

Taille de l'allocation mémoire : Le temps maximal écoulé entre la création d'une donnée par une opération de la tâche T^w et sa dernière utilisation par une opération de la tâche T^r permet de calculer l'emplacement mémoire requis.

Heuristiques : Il peut être utile de connaître l'ensemble des blocs de T^w devant être exécutés pour effectuer un bloc de la tâche T^r. Ceci permet de réserver la place mémoire nécessaire minimale et d'ordonnancer au plus tôt les blocs de calcul.

2.2 Elimination des quantificateurs

La première étape correspond en fait à l'élimination des variables définissant l'ensemble des comportements locaux à un bloc temporel. A partir de la définition des dépendances (5.4), les équations définissant $\mathcal{D}_k$ sont calculables. Ainsi, si l'on se place dans le $k^{\text{ème}}$ plan, en omettant les indices, (i^w dénote i^w_p, I^w, $I^w_{k,p}$, etc...), on a :

[2] L'algorithme de Fourier-Motzkin permet, à partir d'un système de contraintes linéaires comportant inégalités et égalités, d'éliminer les variables et quantificateurs associés. Le résultat de cette élimination est un système de contraintes définissant un sur-ensemble approximant les solutions du système initial

[3] L'algorithme de Chernikova permet de passer d'un système de contraintes linéaires représentant un polyèdre à un système générateur, dont les sommets, rayons et droites sont définies dans les rationnels et inversement

[4] Les coupes de Gomory permettent de déterminer itérativement les sommets entiers d'un polyèdre à partir de sa définition rationnelle

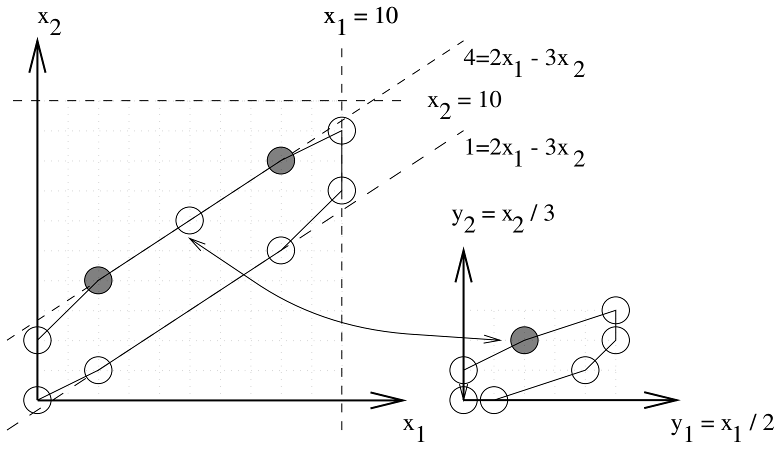


FIG. 5.1. – *Polygone et son transformé discret par division entière suivant Ox_1 et Ox_2*

$$\begin{cases} 0 \leq c^w \leq up(c^w) - 1 \\ 0 \leq c^r \leq up(c^r) - 1 \\ I^w(L^w P^w c^w + s^w) - I^r(L^r P^r c^r + s^r) = M^r m^r - M^w m^w \\ 0 \leq m^r < up(m^r) \\ 0 \leq m^w < up(m^w) \\ 0 \leq s^w \leq L^w P^w - 1 \\ 0 \leq s^r \leq L^r P^r - 1 \end{cases}$$

Par élimination des variables s^w, s^r, m^r et m^w, on engendre le système de contraintes suivant qui impliquent le précédent :

$$\begin{cases} 0 \leq c^w \leq up(c^w) - 1 & (Or \text{ et } Or') \\ 0 \leq c^r \leq up(c^r) - 1 & (Ow \text{ et } Ow') \\ -M^w(up(m^w) - 1) \leq I^w(L^w P^w (c^w + 1) - 1) - I^r L^r P^r c^r & (D_{inf}) \\ I^w L^w P^w c^w - I^r(L^r P^r (c^r + 1) - 1) \leq M^r(up(m^r) - 1) & (D_{sup}) \end{cases} \quad (5.5)$$

Avec 6 droites il y a normalement $C_6^2 = 15$ points d'intersections, mais comme on a 3 paires de droites parallèles, il n'en reste plus que 12, à savoir :$A = Or \cap Ow$, $B = Or \cap D_{inf}$, $C = Or \cap Ow'$, $D = Ow' \cap D_{inf}$, $E = Or' \cap Ow'$, $F = Or' \cap D_{sup}$, $G = Or' \cap Ow$, $H = Ow \cap D_{sup}$, $I = Or \cap D_{sup}$, $J = Ow \cap D_{inf}$, $K = Ow' \cap D_{sup}$, $L = Or' \cap D_{inf}$.

2.3 Intersection des droites dans $\mathbb{N}^2$

De plus, les sommets des polygones doivent être définis dans les entiers. Il est donc parfois nécessaire de considérer des sommets approchés. Le polygone obtenu caractérise alors une surapproximation des paires d'itérations dépendantes. Il faut donc calculer tous les points d'intersection potentiellement valides et sélectionner les bons en n'utilisant des min et des max et non des contraintes conditionnelles en raison du grand nombre de polygones sur lesquels il faut opérer, et d'une propagation de contraintes qui ne serait pas efficace. Ainsi, la représentation des polyèdres pourra être implantée dans un langage de contraintes. Ceci explique la technicité du résultat suivant :

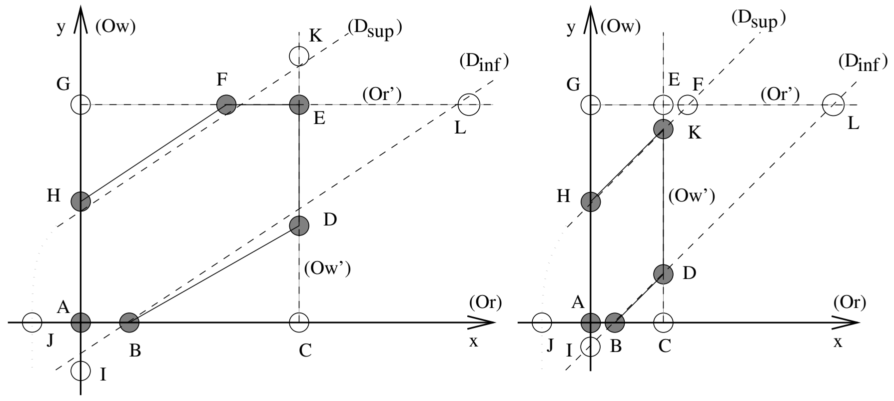


FIG. 5.2. – *Deux cas de figure parmi d'autres*

Proposition 2.2 *En se référant aux notations de la figure (5.2), les 6 points délimitant $\mathcal{P}_k$, (léger) sur-ensemble du plus petit polygone contenant les points satisfiant $\mathcal{D}_k$, ont les expressions suivantes :*

$$
\begin{array}{lcllcll}
A & = & (& 0 & , & 0 &) \\
BC & = & (& \min(x_B, up(x)) & , & 0 &) \\
CDL & = & (& \min(x_L, up(x)) & , & \max(0, \min(y_D, up(y))) &) \\
HG & = & (& 0 & , & \min(y_H, up(y)) &) \\
FGK & = & (& \max(0, \min(x_F, up(x))) & , & \min(y_K, up(y)) &) \\
EKL & = & (& \min(x_L, up(x)) & , & \min(y_K, up(y)) &)
\end{array}
$$

avec

$$
\begin{array}{rcl}
x_B & = & \left\lceil \dfrac{I^w(\mathsf{L^w P^w} - 1) + M^w(up(m^w) - 1)}{I^r \mathsf{L^r P^r}} \right\rceil \\
x_F & = & \left\lfloor \dfrac{I^w \mathsf{L^w P^w}(\mathsf{up(c^w)} - 1) - I^r(\mathsf{L^r P^r} - 1) - M^r(up(m^r) - 1)}{I^r \mathsf{L^r P^r}} \right\rfloor \\
x_L & = & \left\lceil \dfrac{I^w(\mathsf{L^w P^w up(c^w)} - 1) + M^w(up(m^w) - 1)}{I^r \mathsf{L^r P^r}} \right\rceil \\
y_D & = & \left\lfloor \dfrac{I^r \mathsf{L^r P^r}(\mathsf{up(c^r)} - 1) - I^w(\mathsf{L^w P^w} - 1) - M^w(up(m^w) - 1)}{I^w \mathsf{L^w P^w}} \right\rfloor \\
y_H & = & \left\lceil \dfrac{I^r(\mathsf{L^r P^r} - 1) + M^r(up(m^r) - 1)}{I^w \mathsf{L^w P^w}} \right\rceil \\
y_K & = & \left\lceil \dfrac{I^r(\mathsf{L^r P^r up(c^r)} - 1) + M^r(up(m^r) - 1)}{I^w \mathsf{L^w P^w}} \right\rceil \\
up(x) & = & \mathsf{up(c^r)} - 1 \\
up(y) & = & \mathsf{up(c^w)} - 1
\end{array}
$$

Les caractères gras dénotent les quantités variables, l'appellation CDL par exemple indique que ce point est confondu suivant les cas avec C, D ou L.

Les points I et J comportant des coordonnées négatives ne caractérisent pas le polygone de dépendances et ne sont pas considérées dans cette section. Les translations dues aux fonctions d'accès affines ne sont pas repréntées dans ce chapitre.

2.4 Sélection des points pertinents

Il s'agit maintenant de sélectionner suivant les différents cas les points qui appartiennent au polygone. On a comme propriété que D_{inf} et D_{sup} ont toujours une pente positive car les coefficients des matrices d'accès sont positifs ou nuls, et que la seconde est "au-dessus" de la première. Pour être certain d'avoir tous les points de $\mathcal{P}_k$, on prend pour les points de D_{inf} des approximations entières supérieures sur les x et inférieures sur les y, et *vice-versa* pour D_{sup}. Comme on a

$$\begin{aligned} x_B &= \left\lceil \frac{I^w(\mathsf{L}^\mathsf{w}\mathsf{P}^\mathsf{w} - 1) + M^w(up(m^w) - 1)}{I^r\mathsf{L}^\mathsf{r}\mathsf{P}^\mathsf{r}} \right\rceil & \geq 0 \\ y_H &= \left\lceil \frac{I^r(\mathsf{L}^\mathsf{r}\mathsf{P}^\mathsf{r} - 1) + M^r(up(m^r) - 1)}{I^w\mathsf{L}^\mathsf{w}\mathsf{P}^\mathsf{w}} \right\rceil & \geq 0 \end{aligned}$$

On ne considère pas $x_I \leq 0$ et $y_J \leq 0$ car I et J soit n'appartiennent à $\mathcal{P}_k$, soit sont confondus avec A ; on les considère donc pas. On a les propriétés suivantes entre les autres points :

$$\begin{aligned} x_B > x_C &\iff y_D < y_C \\ y_E > y_D &\iff x_E < x_L \\ x_E > x_F &\iff y_E < y_K \\ x_F > x_G &\iff y_H < y_G \end{aligned}$$

Considérons les différents cas qui déterminent les points à retenir :

- A : on a toujours $A \in \mathcal{P}_k$.
- Sélection des points de D_{inf}
  1. $x_B < x_C$: on retient B, on élimine C.
     (a) $y_D < y_E$: on retient D, on élimine L.
     (b) $y_D \geq y_E$: on retient L, on élimine D.
  2. $x_B \geq x_C$: on retient C, on élimine B, D et L
- Sélection des points de D_{sup}
  1. $y_H < y_G$: on retient H, on élimine G.
     (a) $x_F < x_E$: on retient F, on élimine K.
     (b) $x_F \geq x_E$: on retient K, on élimine F.
  2. $y_H \geq y_G$: on retient G, on élimine H, F et K
- Sélection de E On prend E ssi, $x_F < x_E < x_L$ et $y_D < y_E < y_K$

On vérifie enfin pour chaque cas qu'on obtient bien les points désirés éventuellement redondants avec les expressions de A, BC, CDL, GH, FGK et EKL donnés dans l'énoncé de la proposition. □

Remarque 2.3 *Si on connaît les bornes inférieures et supérieure des L^sP^s, on peut examiner les cas dans lesquels on est sûr statiquement qu'un point potentiel de $\mathcal{P}_k$ est à considérer ou à éliminer. La résolution étant concurrente, on pourrait faire ce genre d'optimisation au cours de la résolution lorsque les partitionnements pour deux nids de boucles dépendants sont connus.*

2.5 Interprétation du polyèdre de dépendances

La seule expression de l'ensemble de ces points ne permet pas d'appréhender la signification des dépendances.

Les modèles d'allocation, de latence et les heuristiques reposent en partie sur les sommets ou les droites qui composent ce polyèdre de dépendances. Soit un point en abscisse en lecture, par exemple le point $(1, 0)$ sur de la figure (5.3), soit la droite parallèle aux ordonnées passant par ce point, alors son intersection avec la "bande" formée par D_{sup} et D_{inf} représente tous les blocs en écriture requis pour exécuter ce point. Sur l'exemple, ce sont les points d'ordonnées $\{1, 2, 3\}$. Cette "bande" représente l'ensemble des paires d'itérations dépendantes.

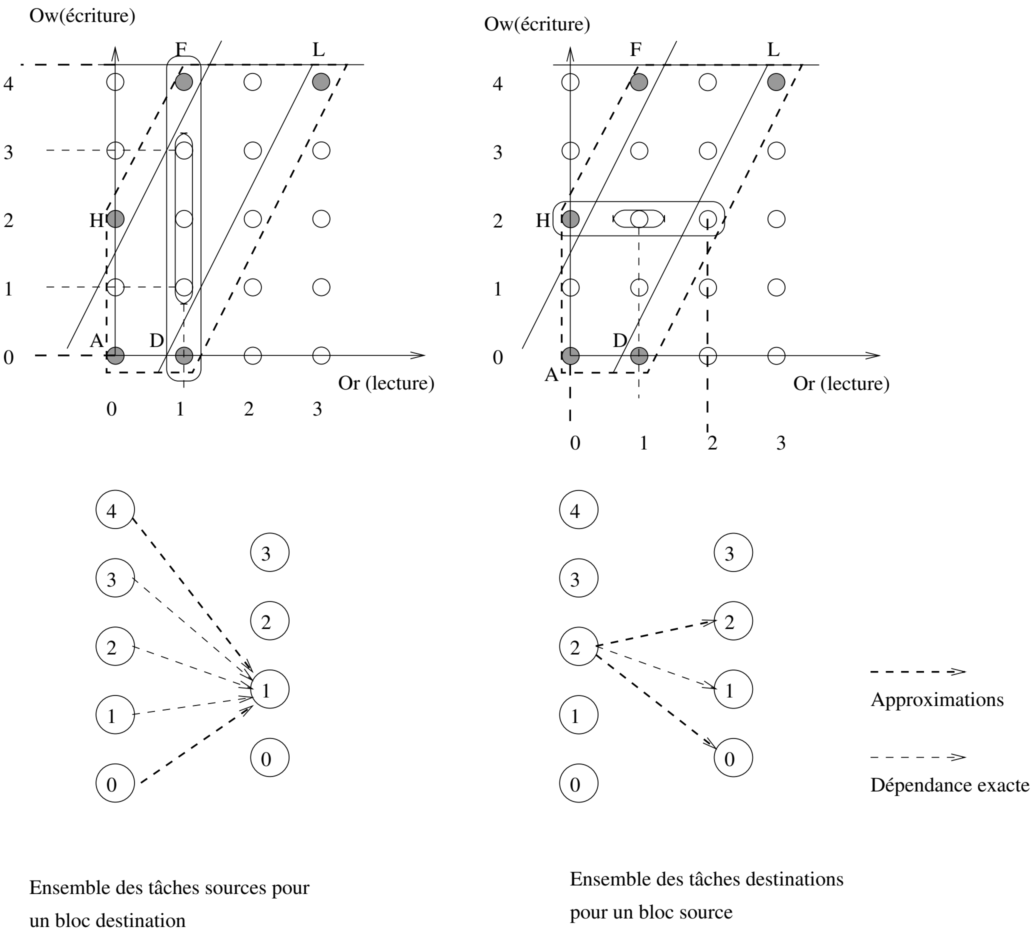
Si l'on fixe une convention d'ordonnancement croissante (comme il le sera effectivement spécifé en chap. 7), le point de cette droite le plus proche de D_{sup} (point $(3, 4)$) sera dit "dernier écrit" tandis que le point le plus proche de D_{inf} sera dit "premier écrit". De même, soit un point en ordonnée en écriture, par exemple le point $(2, 2)$ sur la figure (5.3), soit la droite parallèle à l'abscisse passant par ce point, alors son intersection avec la "bande" formée par D_{sup} et D_{inf} représente tous les blocs en lecture qui utiliseront les données produites par ce point. Ainsi, le point de cette droite le plus proche de D_{sup} sera dit "premier lu" tandis que le point le plus proche de D_{inf} sera dit "dernier lu". De ce fait les points et sommets situés sur ces droites sont très importants, car ils constituent la base des informations liées à la durée d'utilisation et la réutilisation de résultats de blocs en écriture.

2.6 Effets de la surapproximation polygone

Le calcul des dépendances avec en tenant compte de sommets approchés entraine différents effets sur les autres modèles. Ces sur-approximations contraignent davantage ceux-ci et le système global pourrait ne pas trouver de solution alors qu'il en existe. En ce qui concerne les modèles de communication et d'allocation, ces approximations entrainent un sur-coût risquant de sur-contraindre le problème global. Il est difficile de caractériser l'ensemble des solutions qui ne serons pas retenues à cause de la non-linéarité des contraintes qui composent ces modèles. En revanche, la sur-approximation des sommets est particulièrement gênante pour le cas où les ordonnancements définissent des fusions de boucles (cf. introduction du chapitre 7). Nous donnons la raison exacte en (7). Comme il sera étudié dans le modèle d'ordonnancement, nous serons amenés à relaxer artificiellement l'approximation dans certains cas. En fait, d'un point de vue plus général, le problème rencontré ici est la maîtrise des approximations réalisées. C'est à dire qu'une approximation ne devant pas ôter de solutions intéressantes pour le problème global.

3 Dépendances avec modulo

Les accès en lecture des tâches sont en fait de la forme $\varphi = (I^r.\mathbf{i}^r + M^r.\mathbf{m}^r) \bmod \mathrm{up}(\varphi)$. Il faut donc généraliser la démarche précédente. Soit φ le tableau sur lequel porte les dépendances. On raisonne



Dans la partie gauche, le bloc d'opérations 1, nécessite l'exécution des blocs $\{1, 2, 3\}$. L'approximation rajoute les blocs $\{4, 0\}$ à cet ensemble. Dans la partie droite, le bloc d'opérations 2, produit des données utilisés par le bloc $\{1\}$. Là encore, l'approximation rajoute les blocs $\{2, 0\}$ à cet ensemble.

FIG. 5.3. – *Interprétation du polyèdre de dépendances*

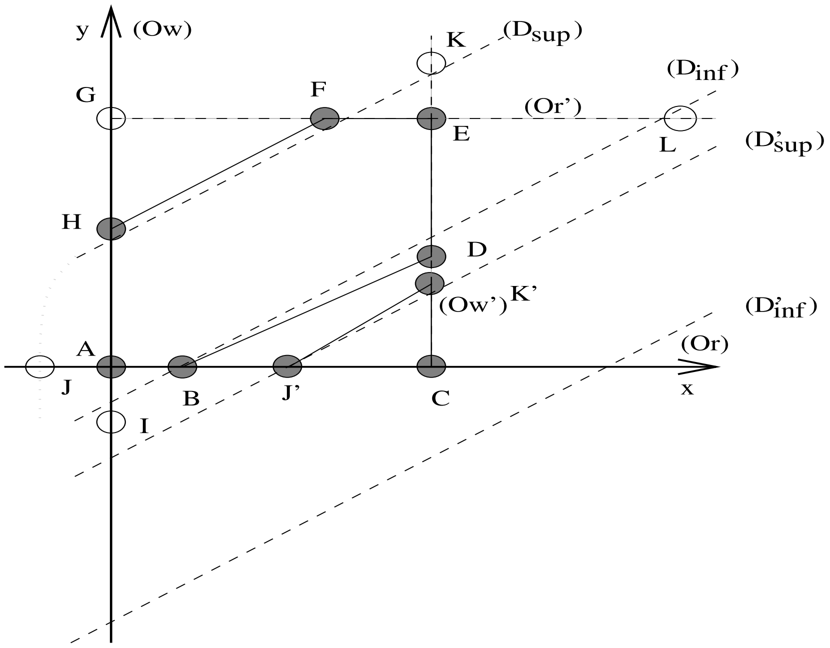


FIG. 5.4. – *Polygone des dépendances avec modulo*

toujours dans le plan, ne considérant qu'une seule dimension sur φ. On note $\mathrm{up}(\varphi)$ la taille de φ sur cette dimension. Les accès vérifient les hypothèses suivantes :

- La taille d'un motif n'excède pas la taille du tableau. En effet le contraire consisterait à inclure plusieurs fois le même élément dans un motif. Donc $M^r.(\mathrm{up}(m^r) - 1) < \mathrm{up}(\varphi)$.
- La base du dernier motif est adressée sans modulo, *i.e.* $I^r.(\mathrm{up}(i^r) - 1) < \mathrm{up}(\varphi)$.

Proposition 3.1 *Il faut prendre en compte la fonction modulo si et seulement si,*

$$I^r.(up(i^r) - 1) + M^r.(up(m^r) - 1) \geq up(\varphi)$$

Avec les hypothèses précédentes,

$$D(i^w, i^r) \Longleftrightarrow \exists (m^w, m^r) \in \mathcal{M}^w \otimes \mathcal{M}^r \ \ \exists 0 \leq \mu \leq 1 \ \mid \ I^w i^w + M^w m^w = I^r i^r + M^r m^r - \mu.\mathrm{up}(\varphi)$$

On a déjà calculé le polygone $\mathcal{P}_k$ correspondant à $\mu = 0$. Traitons exclusivement le cas $\mu = 1$ et calculons $\mathcal{P}'_k$. On obtiendra ainsi deux polygones convexes, sur les sommets desquels on pose les contraintes d'ordonnancement. On considère alors l'enveloppe convexe des dépendances, ce qui n'empêche pas de calculer des ordonnancements légaux.

Proposition 3.2 *Supposons les hypothèses de la proposition (3.1) vérifiées. En se référant aux notations de la figure (5.4), les 3 points délimitant une légère surapproximation du plus petit polygone contenant D'_k ont les expressions suivantes :*

$$\begin{array}{ll} C & = (up(c^r) - 1, 0) \\ J' & = (x_{J'}, 0) \\ K' & = (up(c^r) - 1, y_{K'}) \end{array}$$

avec

$$\begin{array}{lll} x_{J'} & = & \left\lfloor \dfrac{up(\varphi) - I^r(\mathsf{L^rP^r} - 1) - M^r(up(m^r) - 1)}{I^r\mathsf{L^rP^r}} \right\rfloor \\ y_{K'} & = & \left\lceil \dfrac{I^r\mathsf{L^rP^r}(\mathsf{up(c^r)} - 1) + M^r(up(m^r) - 1) - up(\varphi)}{I^w\mathsf{L^wP^w}} \right\rceil \end{array}$$

Démonstration – Montrons tout d'abord que $D'_{inf} \bigcap \mathcal{C} = \emptyset$ avec $\mathcal{C}$ le carré délimité par Or $O'r$ Ow et $O'w$. D'_{inf} a pour équation

$$I^wC^wc^w - I^rC^rc^r = I^r(C^r - 1) + M^r(\text{up}(m^r) - 1) - \text{up}(\varphi)$$

et $D' = D'_{inf} \bigcap O'w$ vérifie

$$\begin{array}{lll} I^wC^wc^w & = & I^rC^r(\text{up}(c^r - 1) - \text{up}(\varphi) - (C^w - 1) - M^w(\text{up}(m^m - 1) \\ & \leq & I^rC^r(\text{up}(c^r - 1) - \text{up}(\varphi) \quad < 0 \end{array}$$

Comme la pente de cette droite est positive, on en conclu qu'elle n'a aucune intersection avec $\mathcal{C}$. Par ailleurs les points J' et K' données dans l'énoncé sont toujours inclus dans $\mathcal{C}$. On en déduit que le polygone $\mathcal{P}'_k$ est un triangle dont les sommets sont $\{C, J', K'\}$. □

4 Dépendances en lecture au sein d'un nid de boucles

L'utilisation des dépendances Lecture/Lecture permet de définir quelles sont les données utilisées à plusieurs reprises par plus d'une itération. D'une part, ces données devraient résider dans une mémoire locale puisque l'on accède à celles ci plus souvent qu'aux autres. D'autre part, les données engendrant une dépendance lecture/lecture entre deux itérations parallélisées doivent être dupliquées si ces itérations sont placées sur des processeurs différents. Ces duplications entrainent des communications. L'intersection des régions référencées en lecture permet d'exprimer les dépendances entre deux itérations.

4.1 Cas général de dépendances

Selon la définition (chap. 2, § 3.1.4, p. 48) des accès aux tableaux, largement utilisée dans ce chapitre, on dira que deux itérations $\mathbf{i}^1, \mathbf{i}^2$ sont dépendantes en lecture si et seulement si elles référencent un même élément de tableau, *i.e.* $\exists \varphi \in \Phi$ tel que $\varphi \in \mathcal{R}(\mathbf{i}^1)$ et $\varphi \in \mathcal{R}(\mathbf{i}^2)$.

Définition 4.1 *Considérant un nid de boucles, l'ensemble $\mathcal{R}(\mathbf{i}^1, \mathbf{i}^2)$ des données référencées par les deux itérations s'écrit :*

$$\mathcal{R}(\mathbf{i}^1, \mathbf{i}^2) = \mathcal{R}(\mathbf{i}^1) \bigcap \mathcal{R}(\mathbf{i}^2)$$

où $\mathcal{R}(\mathbf{i}^1)$ (resp $\mathcal{R}(\mathbf{i}^2)$) représente la région du tableau de données φ référencée par l'itération $\mathbf{i}^1$ (resp. $\mathbf{i}^2$) avec :

$$\mathcal{R}(\mathbf{i}^1) = \left\{ \varphi \mid \varphi \in \Phi,\ \exists \mathbf{m}^1 \in \mathcal{M}^1,\ \varphi = I^1.\mathbf{i}^1 + M^1.\mathbf{m}^1 \right\}$$

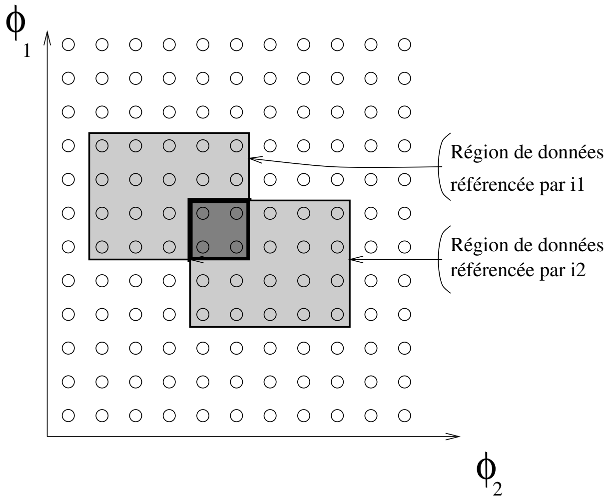


FIG. 5.5. – *Dépendance Lecture/Lecture entre itérations* **i**1 *et* **i**2 *d'un même nid de boucles*

Le volume V de données référencées par $\mathbf{i}^1$ et $\mathbf{i}^2$ est

$$V(\mathbf{i}^1, \mathbf{i}^2) = \text{card } \mathcal{R}(\mathbf{i}^1, \mathbf{i}^2)$$

On dira qu'il y a dépendance lecture-lecture entre $\mathbf{i}^1$ et $\mathbf{i}^2$ si et seulement si :

$$\mathcal{R}(\mathbf{i}^1, \mathbf{i}^2) \neq \varnothing$$

4.2 Simplification dues aux caractéristiques applicatives

On a vu en (chap. 2, § 3.1.4, p. 48) qu'il est possible de donner l'ensemble de points référencés par la totalité des itérations du nid de boucles, par la région : $\mathcal{R} = \{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{i} \in \mathcal{I},\ \exists\ \mathbf{m} \in \mathcal{M},\ \varphi = I.\mathbf{i} + M.\mathbf{m}\}$ où I et M sont des matrices associées à des applications linéaires. Posons alors :

- L'espace $\mathcal{IM}$ est considéré comme étant l'espace $\mathcal{I} \times \mathcal{M}$: $\mathcal{IM} = \{(\mathbf{i};\mathbf{m}) \mid \mathbf{i} \in \mathcal{I} \text{ et } \mathbf{m} \in \mathcal{M}\}$. La dimension de l'espace $\mathcal{IM}$ est $\dim(\mathcal{I}) + \dim(\mathcal{M})$.
- Soit Ω la matrice $\Omega = (I \quad M)$.

La région référencée par la totalité du nid de boucles peut s'écrire :

$$\mathcal{R} = \{\varphi \mid \varphi \in \Phi,\ \exists\ \mathbf{j} \in \mathcal{IM},\ \varphi = \Omega(\mathbf{j})\}$$

Les dépendances Lecture/Lecture correspondent à l'étude du noyau[5] de la fonction d'accès $\Omega = (I \quad M)$. En effet, dire qu'il y a une dépendance Lecture/Lecture signifie qu'il existe pour une donnée φ fixée, au moins deux vecteurs $\mathbf{j}^1$ et $\mathbf{j}^2$ tel que $\varphi = \Omega(\mathbf{j}^1) = \Omega(\mathbf{j}^2)$, ce qui est équivalent (par linéarité de Ω) à $\Omega(\mathbf{j}^1 - \mathbf{j}^2) = \mathbf{0}$.

[5]On appelle noyau d'une application linéaire Ω l'ensemble suivant $\text{Ker}(\Omega) = \{x \mid \Omega(x) = 0\}$.

Proposition 4.2 *Dans le cas des restrictions applicatives décrites précédemment, le noyau de la fonction d'accès Ω est engendré par la famille de vecteurs de $\mathcal{IM} = \mathcal{I} \times \mathcal{M}$ de la forme :*

$$\forall \lambda, \kappa, \quad \left(\underbrace{0, ...0, 1, 0, ..., 0}_{\in \mathcal{I}}, \underbrace{0, ..., 0, \frac{-I_{\lambda,\kappa}}{M_{\lambda,\kappa'}}, 0, ..., 0}_{\in \mathcal{M}} \right)^t \in Ker(\Omega)$$

où toutes les coordonnées sont nulles sauf la κ-ième coordonnée qui vaut 1 et la $\kappa{+}\kappa'$-ième coordonnée qui vaut $\frac{-I_{\lambda,\kappa}}{M_{\lambda,\kappa'}}$.

Grâce aux caractéristiques applicatives, la matrice M est inversible dans $\mathbb{Q}$, car diagonale à une permutation près (chap. 2, § 3.1.5, p. 49).

Démonstration –

$$\mathbf{x} = \mathbf{i} + \mathbf{m} \in \mathrm{Ker}(\Omega) \Leftrightarrow I.\mathbf{i} + M.\mathbf{m} = \mathbf{0} \Leftrightarrow \mathbf{m} = -M^{-1}.I.\mathbf{i}$$

En utilisant les caractéristiques applicatives, on en déduit que les vecteurs engendrant le noyau sont de la forme

$$\forall \lambda, \kappa, \quad \left(\underbrace{0, ..., 0, 1, 0, ..., 0}_{\kappa\text{-ième coordonnée}}, \underbrace{0, ..., 0, \frac{-I_{\lambda,\kappa}}{M_{\lambda,\kappa'}}, 0, ..., 0}_{\kappa+\kappa'\text{-ième coordonnée}} \right)^t$$

□

On remarque qu'une autre base du noyau peut s'écrire sous la forme :

$$\forall \lambda, \quad \left(0, ..., \frac{M_{\lambda,\kappa'}}{\mathrm{pgcd}(I_{\lambda,\kappa}, M_{\lambda,\kappa'})}, 0, ..., 0, 0, ..., , 0, \frac{-I_{\lambda,\kappa}}{\mathrm{pgcd}(I_{\lambda,\kappa}, M_{\lambda,\kappa'})}, 0..., 0 \right)^t \tag{5.6}$$

Cependant, repartons de la définition restreinte des régions de données accédés (chap. 2, § 3.1.5, p. 49) :

$$\mathcal{R}(\mathbf{i}) = \{(\varphi_1, \ldots, \varphi_\lambda, \ldots, \varphi_n) \mid \forall \lambda \;\; \exists(\kappa, \kappa'),\; \exists m_{\kappa'},\; \varphi_\lambda \;=\; I_{\lambda,\kappa}.i_\kappa \;+\; M_{\lambda,\kappa'}.m_{\kappa'}\}$$

Alors $\forall \lambda$, on a : $\exists(\kappa, \kappa'),\; \exists m_{\kappa'},\; \varphi_\lambda = I_{\lambda,\kappa}.i_\kappa \;+\; M_{\lambda,\kappa'}.m_{\kappa'}$.

$$\Leftrightarrow \exists(\kappa, \kappa'),\; \exists m_{\kappa'},\; \varphi_\lambda = \tilde{I_\lambda}.(I'_{\lambda,\kappa}.i_\kappa + M'_{\lambda,\kappa'}.m_{\kappa'}) \tag{5.7}$$

Avec

$$\tilde{I_\lambda} = \mathrm{pgcd}(M_{\lambda,\kappa'}, I_{\lambda,\kappa}), \quad I'_{\lambda,\kappa} = \frac{I_{\lambda,\kappa}}{\mathrm{pgcd}(I_{\lambda,\kappa}, M_{\lambda,\kappa'})},\; M'_{\lambda,\kappa'} = \frac{M_{\lambda,\kappa'}}{\mathrm{pgcd}(I_{\lambda,\kappa}, M_{\lambda,\kappa'})}$$

On en déduit la propriété suivante :

Propriété 4.3 *Si les fonctions d'accès satisfont la contrainte :*

$$(I_{\lambda,\kappa} > M_{\lambda,\kappa'}.up(m_{\kappa'})) \vee (I_{\lambda,\kappa}.up(i_\kappa) < M_{\lambda,\kappa'}) \tag{5.8}$$

alors la fonction d'accès est injective, et il n'y a pas de dépendances Lecture/Lecture.

Démonstration – Il y a dépendances Lecture/Lecture, si :

$$\exists (i_\kappa, i'_\kappa),\ i_\kappa \neq i'_\kappa \text{ et } \exists (m_{\kappa'}, m'_{\kappa'}),\ \text{tel que } I_{\lambda,\kappa}.(i_\kappa - i'_\kappa) + M_{\lambda,\kappa'}.(m_{\kappa'} - m'_{\kappa'}) = 0$$

Il n'y donc a pas de dépendances Lecture/Lecture, si :

$$\forall (i_\kappa, i'_\kappa),\ i_\kappa \neq i'_\kappa \text{ et } \forall (m_{\kappa'}, m'_{\kappa'})\ \text{ tel que } I_{\lambda,\kappa}.(i_\kappa - i'_\kappa) + M_{\lambda,\kappa'}.(m_{\kappa'} - m'_{\kappa'}) \neq 0$$

$$\Longleftarrow (I_{\lambda,\kappa} > M_{\lambda,\kappa'}.up(m_{\kappa'})) \vee (I_{\lambda,\kappa}.up(i_\kappa) < M_{\lambda,\kappa'})$$

□

La méthode d'analyse de dépendances est basée sur le calcul du noyau de la fonction d'accès et ne saurait être appliquée dans le cadre d'accès plus complexes que ceux définis en (chap. 2,§ 3.1.5, p. 49 et § 6.1, p. 67). En effet, lorsque les accès présentent par exemple plusieurs fonctions d'accès dans plusieurs tableaux la représentation pertinente des dépendances est difficile à caractériser. Surapproximer les dépendance lecture-lecture (enveloppe convexe, cône) revient à ajouter artificiellement des accès supplémentaires aux même variables, d'où le risque d'une représentation biaisée.

4.3 Relation entre dépendance de flot et dépendance en lecture

Dans le cas où il y a une dépendance lecture/lecture, la dimension en question, si elle est parallélisée, engendre automatiquement des communications avec les traitements produisant les données, comme nous le démontrons ici. Le seul cas où il n'y a pas de dépendances est le cas où la dimension est entièrement placée sur un seul processeur (sérialisée). Soit la fonction d'accès d'une tâche T^r qui présente des dépendances en lecture sur un ensemble d'itérateurs $\{i^r_n\}$. Soit l'ensemble des itérateurs $\{i^w_m\}$ accédant aux mêmes dimensions de tableaux que $\{i^r_n\}$. Le graphe de dépendances entre $\{i^r_n\}$ et $\{i^w_m\}$ présente une seule et même composante connexe. Soient $i^w_p \in \{i^w_m\}$ et $i^r_q \in \{i^r_n\}$, on peut avoir par exemple :

```
...
T[..,I,...]=...
...
...=T[...,2*I+M,...]
...
```

Soit k la dimension de tableau accédée par ces deux itérateurs.

Proposition 4.4 *Si une dimension i^r_q présente des dépendances lecture/lecture et que $\mathcal{R}^r(i^r_q)$ définit un ensemble d'éléments de tableau convexe, les composantes comportant les itérations i^r_q du graphe de flot de données sont connexes.*

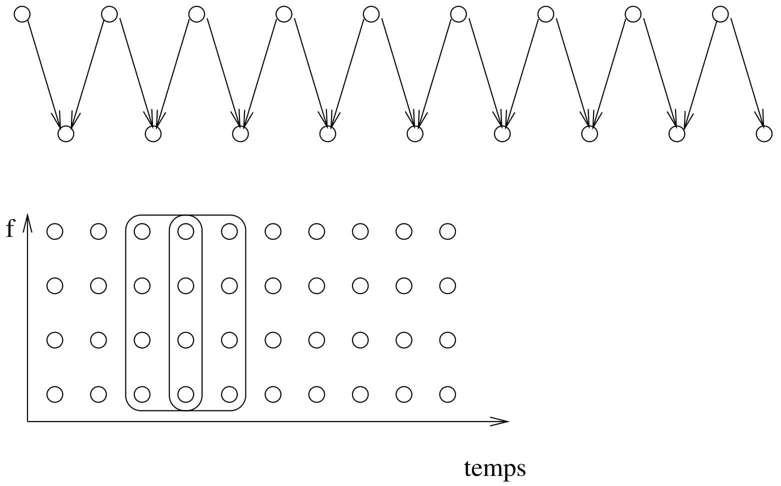
Démonstration –
Soit $\mathcal{R}^w(i^w_p)$, l'ensemble des éléments de tableaux écrits par i^w_p, et $\mathcal{R}^r(i^r_q)$ l'ensemble des éléments de tableaux lus par i^r_q. Si on a une dépendance lecture/lecture, alors par définition :

$$\forall\ i^r_q,\ \exists\ \varphi,\ \exists\ d_q\ \in N \mid \varphi\ \in\ \mathcal{R}^r(i^r_q)\ \wedge\ \varphi\ \in\ \mathcal{R}^r(i^r_q + d_q)$$

or, pour tout élément de tableau Φ, on a

$$\forall \varphi,\ \exists\,!\ i^w_p\ \mid \varphi\ \in\ \mathcal{R}^w(i^w_p)$$

donc



La fonction d'accès dans ce tableau du nid de boucles en lecture est :
$\varphi_t = 2.i_t + m_t,\ 0 \leq m_t < 2$ pour la dimension temporelle,
$\varphi_f = m_f,\ 0 \leq m_f < 4$ pour la dimension finie.
Si un nid de boucles en écriture écrit ne serait-ce qu'un point par itération dans le tableau, les dépendances forment un treillis complètement connexe.

FIG. 5.6. – *Les dépendances présentent un seul ensemble de dépendances connexes*

$$\forall\ i_q^r,\ \exists\ \varphi,\ \exists\ d_q\ \in \mathbb{N} \mid \varphi\ \in\ \mathcal{R}^r(i_q^r)\ \wedge\ \varphi\ \in\ \mathcal{R}^r(i_q^r + d_q)\ \Rightarrow$$
$$\exists\ !\ i_p^w \mid \mathcal{R}_k^w(i_p^w)\ \cap \mathcal{R}_k^r(i_q^r) \neq \emptyset\ \wedge\ \mathcal{R}_k^w(i_p^w)\ \cap \mathcal{R}_k^r(i_q^r + d_q) \neq \emptyset$$

donc

$$\forall\ i_q^r,\ \exists\ \varphi,\ \exists\ d_q\ \in \mathbb{N} \mid \varphi\ \in\ \mathcal{R}^r(i_q^r)\ \wedge\ \varphi\ \in\ \mathcal{R}^r(i_q^r + d_q)\ \Rightarrow\ \exists\ !\ i_p^w \mid D_k(i_p^w, i_q^r)\ \cap\ D_k(i_p^w, i_q^r + d_q) \neq \emptyset$$

et où k est la dimension de tableau accédée.

$\square$

Si la dimension de tableau infinie présente des dépendances lecture-lecture (cf. fig. 5.6), en traitement du signal c'est une fenêtre glissante dans le temps en traitement du signal), ceci à plusieurs significations :

- Si la dimension infinie est parallélisée, en présence de dépendances lecture/lecture (cas d'une fenêtre de calculs qui se recouvre), la dépendance engendrera obligatoirement des communications.
- La durée de vie des variables nécessite l'utilisation d'une allocation cyclique. On ne pourra en effet libérer les variables allouées à une itération i_∞ qu'à la fin de l'exécution de l'itération $i_\infty + d$ où d est la distance de dépendance en lecture (chap. 1, 2.2.2).

5 Conclusion

Nous avons pu, dans ce chapitre, présenter une des difficulté à laquelle l'approche globale est confrontée. Le polyèdre de dépendances inter-procédural entre paires de nids de boucles partitionnés illustre bien cette difficulté de formalisation liée à des représentations dynamiques et multi-dimensionnelles. Néanmoins, les caractéristiques applicatives ainsi que les matrices de partitionnement diagonales nous ont permis dans un premier temps de raisonner dans le plan. Dans un deuxième temps, ces caractéristiques nous autorisent la définition d'une représentation symbolique du polyèdre de dépendances qui sur-approxime l'ensemble des itérations dépendantes. Si cette sur-approximation ne pertube pas la légalité des ordonnancements calculés (comme nous le verrons au chap. (7), elle peut réduire les espaces de solution d'ordonnancement en sur-contraignant ce modèle. Cette remarque est aussi valable pour les autres modèles utilisant la sur-approximation du polytope de dépendances.

La caractérisation des dépendances en lecture est aussi importante pour comprendre les impacts de la réutilisation des données – notamment pour l'allocation mémoire et le coût des communications –. Ces dépendances en lecture seront utilisées en tant qu'heuristiques.

Enfin, comme nous allons le montrer dans la suite de la partie modélisation, la relation de dépendance permet aussi de sur-approximer les coût de communications (chap. 6) ainsi que l'espace mémoire requis pour allouer les données sur chacun des processeurs (chap. 8).

Chapitre 6

Modèle d'Interphase

Les fondements du modèle d'Interphase ont été initiés par Corinne Ancourt, Jean Jourdan *et moi-même. Ils reposent sur la linéarisation des processeurs*[a]*, technique utilisée par les programmeurs du traitement du signal, aussi appelée "projection". Après avoir exprimé les différentes approximations implantables en Programmation par Contraintes, et explicité les liens avec la bande passante et les dépendances, j'ai pu aisément implanter ce modèle en programmation logique avec contraintes.*

[a]Fonction dont j'ai pu prouver la bijectivité

1 Introduction

Le modèle d'interphase permet d'introduire dans la problématique du placement la durée des communications résultant des choix de partitionnement, du type de topologie de la machine et du mode de communications. Chaque nid de boucles de l'application utilise l'architecture d'après un partitionnement propre, définit au chapitre (4). Chaque nid de boucles exploite donc potentiellement les processeurs selon une géométrie particulière . De ce fait, entre deux phases de calculs, la distribution des données peut être complètement différente et redistribuer celles-ci est parfois nécessaire. Il en résulte d'une part des communications effectuées soit en bloc à la fin d'une partition temporelle soit "pipelinées" durant l'exécution des macro-instructions. D'autre part, pour définir finement ces redistributions, une fonction de linéarisation est envisagée permettant la localisation des blocs de calculs emèteurs et récepteurs, c'est l'objet de la section (2). Cependant, la durée des communications obtenue n'est pas uniforme et la présence de nombreux quantificateurs universels nécessite de procéder à des sur-approximations. Celles-ci permettent de manipuler en terme de contraintes non-linéaires les coût de communications afin de choisir des combinaisons de partitionnement pertinentes et autorisent, de surcroît, une génération de code simple (section 3.1).

2 Cas général

Nous avons considéré dans le modèle de partitionnement (chap. 4) que chaque nid de boucles était placé de manière indépendante, en particulier à chaque nid est associée une matrice diagonale P de partionnement, exprimant le nombre de processeurs potentiellement utilisés par le nid. En effet, rappelons que pour un nid de boucles, on associe deux matrices entières (à déterminant non nul) P, L telles qu'il existe un unique triplet (x, y, z) de points de l'espace des itétations $\mathcal{I}$ verifiant :

$$\mathbf{i} = LP\mathbf{x} + L\mathbf{y} + \mathbf{z}$$

$$\text{avec} \begin{cases} \forall\ \mathbf{z},\ \mathbf{0} \leq L^{-1}.\mathbf{z} < \mathbf{1}, \\ \forall\ \mathbf{y},\ \mathbf{0} \leq P^{-1}.\mathbf{y} < \mathbf{1}, \\ \forall\ i \in \mathcal{I},\ \mathbf{0} \leq \mathbf{i} < up(\mathbf{i}) \\ \det(L) \neq 0 \\ \det(P) \neq 0 \text{ et } P \text{ diagonale} \end{cases}$$

où à tout point $i \in \mathcal{I}$ est associé le vecteur $\mathbf{i} = Oi$. Le triplet $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ est une permutation possible de $(\mathbf{c}, \mathbf{p}, \mathbf{l})$ où $\mathbf{c}$ représente le temps local à la boucle, $\mathbf{p}$ permet d'identifier le processeur et $\mathbf{l}$ le motif courant dans le bloc en train d'être calculé. Dans toute la suite du document, nous n'étudierons que la permutation $(\mathbf{c}, \mathbf{p}, \mathbf{l})$.

Dans cette section, nous donnons selon quels critères se définissent théoriquement les volume de données à échanger en § (2.1), comment la linéarisation permet d'associer les blocs de calculs parallèles aux processeurs réels § (2.2). Les sections § (2.3), § (2.4), § (2.5) présentent comment cette fonction de linéarisation, à l'aide des dépendances de flot pourrait être exploitée afin d'évaluer le coût des communications.

2.1 Intersection des ensembles de données référencés

Considérons deux nids de boucles, entre lesquels il y a une dépendance de flot. Nous cherchons à savoir si deux blocs d'itérations – ou partitions – des deux nids de boucles (l'un écrivant, l'autre lisant), placés sur deux processeurs différents référencent les mêmes données. Dans ce cas, l'échange de données entre les deux processeurs concernés est inévitable. Un sous-ensemble des données écrites par la première partition sera alors lu par la deuxième partition. En d'autres termes, il est possible de caractériser ces communications potentielles en calculant le volume de l'intersection des ensembles de données référencées en lecture comme en écriture. Si ce volume est non nul, il en résulte une dépendance de flot de données entre les deux partitions, et outre une communication, les partitions doivent être correctement ordonnancées dans le temps (cf. chapitre 7). Nous devons donc considérer ici l'intersection des ensembles référencés par la partition définie par le couple $(\mathbf{c}, \mathbf{p})$, où $\mathbf{c}$ détermine un événement de calcul lié au nid de boucles, et $\mathbf{p}$ la localisation du calcul (cf. chapitre 4).

Définition 2.1 *Soit T^r et T^w deux nids de boucles et respectivement leurs ensembles de références associés aux partitions en lecture et en écriture : $\mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)})$, $\mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)})$. Le volume de données $\mathcal{V}^{r,w}_{inter}(\mathbf{c}^r, \mathbf{p}^r, \mathbf{c}^w, \mathbf{p}^w)$ à la fois lu par T^r et écrit par T^w correspond à l'égalité suivante :*

$$\mathcal{V}^{r,w}_{inter}(\mathbf{c}^r, \mathbf{p}^r, \mathbf{c}^w, \mathbf{p}^w) \;=\; \text{card}\Big(\ \mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \bigcap \mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)})\ \Big) \tag{6.1}$$

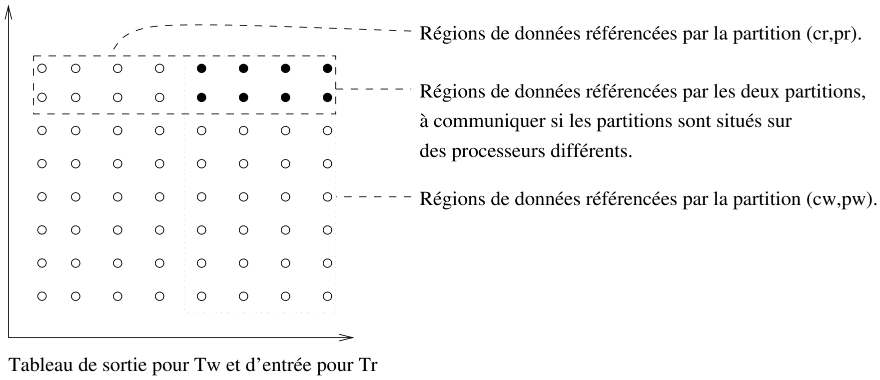


FIG. 6.1. – *Dépendances Lecture/Ecriture entre partitions* $(\mathbf{c}^w, \mathbf{p}^w)$ *et* $(\mathbf{c}^r, \mathbf{p}^r)$ *de deux nids de boucles distincts* T^w *et* T^r

Le volume est formulé en fonction des paramètres dont il dépend. Il en sera de même dans toute la suite de cette section.

Tel que nous avons défini le partitionnement des itérations, il n'est pas possible de savoir si les partitions $\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)}$ et $\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}$ sont situées sur le même processeur physique. En effet, nous n'avons qu'une représentation vectorielle de ces processeurs propre à chaque nid de boucles. Il est donc nécessaire de construire une relation entre les coordonnées vectorieles définies par le vecteur $\mathbf{p}$ et les processeurs physiques de la machine.

2.2 Linéarisation de grille de processeurs

Pour un nid de boucles, la fonction de linéarisation sur les processeurs est une bijection entre l'ensemble des processeurs vectoriels définis par les valeurs du vecteur $\mathbf{p}$ et l'ensemble des processeurs réellement utilisés. En effet, chaque valeur de $\mathbf{p}$ est associée à un seul processeur physique utilisé, comme à chaque processeur physique est associé une seule valeur de $\mathbf{p}$. Si l'application avait été surjective cela aurait signifié que l'on replie des dimensions d'itérations pour les adapter à la dimension de la machine, ce qui est du ressort du partitionnement. Enfin, les processeurs réellement utilisés consitituent un sous-ensemble des processeurs physiques disponibles sur la machine cible. Notre bijection revient donc à définir une application injective de l'ensemble des processeurs vectoriels vers l'ensemble des processeurs physiques disponibles, puisque certains processeurs ne sont pas utilisés. Une telle fonction est associée à chacun des nids de boucle.

2.2.1 Un lemme technique

Nous introduisons ici une forme linéaire permettant détablir une bijection entre l'ensemble des blocs de calculs parallèles et les processeurs de l'architecture physique.

Lemme 2.2 *Soit* $f \;:\; \mathbb{N}^{n+1} \mapsto \mathbb{N}$ *définie par*

$$f(y_0, y_1, ..., y_n) = \sum_{j=0}^{n} \left(\prod_{k=0}^{j} q_k \right) .y_j$$

avec

$$q_0 = 1 \quad et \quad \forall j,\ 1 \leq j \leq n,\ q_j \in \mathbb{N},\ y_{j-1} < q_j$$

Alors f est une application bijective et

$$\forall j, \quad y_j = \left\lfloor \frac{x \bmod \left(\prod_{k=0}^{j+1} q_k\right)}{\left(\prod_{k=0}^{j} q_k\right)} \right\rfloor$$

Démonstration –

1. f est une application car l'addition et la multiplication sont deux lois internes dans $\mathbb{N}$.
2. Montrons donc qu'il existe une application f^{-1} et pour cela donnons sa forme explicite.

   Posons $x = f(y_0, y_1, ..., y_n) = \sum_{j=0}^{n} \left(\prod_{k=0}^{j} q_k \right) .y_j$ avec $q_0 = 1$, et $\forall 1 \leq j \leq n,\ q_j \in \mathbb{N},\ y_{j-1} < q_j$.

$$\begin{aligned} x &= \sum_{j=0}^{n} \left(\prod_{k=0}^{j} q_k \right) .y_j \\ &= y_0 + q_1 y_1 + q_1 q_2 y_2 + \ \dots \ + q_1 q_2 ... q_j y_j + \ \dots \ + q_1 q_2 ... q_j ... q_n y_n \\ &= q_1 q_2 ... q_j \left(q_{j+1} ... q_n y_n + q_{j+1} ... q_{n-1} y_{n-1} + \ \dots \ + q_{j+1} y_{j+1} \right) \\ &\quad + \ \left(q_1 q_2 ... q_{j-1} y_{j-1} + q_1 q_2 ... q_{j-2} y_{j-2} + \ \dots \ + q_1 y_1 + q_0 \right) \end{aligned}$$

   Posons alors $Q_j = \prod_{k=0}^{j} q_k$, $\quad Y_j = y_j + \sum_{k=j+1}^{n} \left(\prod_{l=j+1}^{k} q_l \right) y_k \quad$ et $\quad R_j = \sum_{k=0}^{j-1} \left(\prod_{l=0}^{k} q_l \right) y_k$

   Alors on a :

$$x = Q_j Y_j + R_j$$

   Montrons que l'on a en fait : $x \equiv R_j \bmod Q_j$, c'est-à-dire que $\forall j,\ 0 \leq\ R_j\ <\ Q_j$.

   - On va dons montrer par récurrence que pour tout j, on a :

$$\sum_{k=0}^{j-1} \left(\prod_{l=0}^{k} q_l \right) y_k \ < \ \prod_{k=0}^{j} q_k \tag{6.2}$$

     – Pour $j = 1$, la relation (6.2) est vraie par hypothèse car $y_0 < q_1$.

     – Supposons que l'on ait :

$$\sum_{k=0}^{j-1} \left(\prod_{l=0}^{k} q_l \right) y_k \ < \ \prod_{k=0}^{j} q_k$$

     et démontrons que cela implique que

$$\sum_{k=0}^{j} \left(\prod_{l=0}^{k} q_l \right) y_k \ < \ \prod_{k=0}^{j+1} q_k$$

     Or

$$\sum_{k=0}^{j} \left(\prod_{l=0}^{k} q_l \right) y_k = \sum_{k=0}^{j-1} \left(\prod_{l=0}^{k} q_l \right) y_k \ + \ \left(\prod_{l=0}^{j} q_l \right)$$

     et par hypothèse on a :

$$\left\{\begin{array}{lcl} \displaystyle\sum_{k=0}^{j-1}\left(\prod_{l=0}^{k} q_l\right)y_k & < & \displaystyle\prod_{k=0}^{j} q_k \\ y_j & < & q_{j+1} \end{array}\right. \Leftrightarrow \left\{\begin{array}{lcl} \displaystyle\sum_{k=0}^{j-1}\left(\prod_{l=0}^{k} q_l\right)y_k & \leq & \displaystyle\prod_{k=0}^{j}(q_k-1) \\ (\prod_{k=0}^{j} q_k)y_j & \leq & (\prod_{k=0}^{j} q_k)q_{j+1} - (\prod_{k=0}^{j} q_k) \end{array}\right.$$

Tout cela implique donc :

$$\begin{aligned} \sum_{k=0}^{j}\left(\prod_{l=0}^{k} q_l\right)y_k & \leq \left(\prod_{k=0}^{j} q_k\right) - 1 + \left(\prod_{k=0}^{j+1} q_k\right) - \left(\prod_{k=0}^{j} q_k\right) \\ & \leq \left(\prod_{k=0}^{j+1} q_k\right) - 1 \\ & < \left(\prod_{k=0}^{j+1} q_k\right) \end{aligned}$$

Par récurrence, on a donc bien montré que $(x \equiv R_j) \bmod Q_j$.

- Ceci est donc équivalent à :

$$\left[x \equiv \sum_{k=0}^{j}\left(\prod_{l=0}^{k} q_l\right)y_k\right] \bmod \left(\prod_{k=1}^{j} q_K\right)$$

or

$$\begin{aligned} \sum_{k=0}^{j}\left(\prod_{l=0}^{k} q_l\right)y_k & = q_0q_1...q_{j-1}\left(y_{j-1} + \frac{1}{q_{j-1}}y_{j-1} + \ ... \ + \frac{1}{q_2q_3...q_{j-1}}y_1 + \frac{1}{q_1q_2q_3...q_{j-1}}y_0\right) \\ & = q_0q_1...q_{j-1}\left(y_{j-1} + \sum_{k=0}^{j-2}\left(\prod_{l=k+1}^{j-1}\frac{1}{q_l}\right)y_k\right) \end{aligned}$$

D'où :

$$\frac{x \bmod \left(\prod_{k=1}^{j} q_K\right)}{\prod_{l=0}^{j-1} q_l} = y_{j-1} + \sum_{k=0}^{j-2}\left(\prod_{l=k+1}^{j-1}\frac{1}{q_l}\right)y_k$$

En prenant la partie entière de chaque côté, on a :

$$\lfloor\frac{x \bmod \left(\prod_{k=1}^{j} q_K\right)}{\prod_{l=0}^{j-1} q_l}\rfloor = \lfloor y_{j-1} + \sum_{k=0}^{j-2}\left(\prod_{l=k+1}^{j-1}\frac{1}{q_l}\right)y_k\rfloor$$

On a vu précédemment (voir la relation (6.2) de récurrence) que l'on a :

$$\begin{aligned} \sum_{k=0}^{j-2}\left(\prod_{l=0}^{k} q_l\right)y_k < \prod_{l=0}^{j-1} q_l & \iff \sum_{k=0}^{j-2}\frac{\prod_{l=0}^{k} q_l}{\prod_{l=0}^{j-1} q_l}y_k < 1 \\ & \iff \sum_{k=0}^{j-2}\left(\prod_{l=k+1}^{j-1}\frac{1}{q_l}\right)y_k < 1 \\ & \iff \lfloor y_{j-1} + \sum_{k=0}^{j-2}\left(\prod_{l=k+1}^{j-1}\frac{1}{q_l}\right)y_k\rfloor = y_{j-1} \end{aligned}$$

□

2.2.2 La linéarisation des processeurs

Soit un ensemble de processeurs vectoriels représenté par le vecteur $\mathbf{p}$, alors $0 \leq P^{-1}.\mathbf{p} < \mathbf{1}$ issu du partitionnement. Les composantes sont linéarisés grâce à la bijection définie ci-dessus :

Définition 2.3 *Soit un vecteur de partitionnement* $\mathbf{p}$ *de composantes* $(p_0, p_1, \ldots, p_{n-1})$*, la matrice de partitionnement associée* P *dont les éléments non nuls sont les* P_{kk}*. Soit l'ensemble des processeurs vectoriels* $PV = \{(p_0, \ldots, p_n) \mid \forall\ k,\ p_k < P_{kk}\ \}$ *et soit* x *un processeur (réel) utilisé. Soit* $\mathcal{L}$ *la fonction définie par :*

$$\mathcal{L}(p_0, \ldots, p_{n-1}) = \sum_{j=0}^{n-1} (\prod_{k=0}^{j-1} P_{k,k}).p_j, \qquad \forall\ j,\ 0 \leq j < n,\ 0 \leq p_j < P_{jj} \tag{6.3}$$

alors $\mathcal{L}$ *s'appelle la fonction de* linéarisation des processeurs.

Proposition 2.4 *La fonction* $\mathcal{L}$ *est bijective et admet pour inverse* $(p_0, \ldots, p_n) = \mathcal{L}^{-1}(x)$*, avec :*

$$\forall\ j,\ 0 \leq j < n, \quad p_j = \lfloor \frac{x \mod (P_{j,j} \times P_{j-1,j-1} \ldots \times P_{0,0})}{(P_{j-1,j-1} \times \ldots \times P_{0,0})} \rfloor \tag{6.4}$$

Démonstration – C'est une application directe du lemme (2.2). □

2.3 Volumes de communications entre partitions élémentaires

Il est désormais possible de différencier deux processeurs physiques. Naturellement, à deux nids de boucles différents T^1 et T^2, correspondent deux fonctions de linéarisation distinctes $\mathcal{L}^1$ et $\mathcal{L}^2$. Si on a $\mathcal{L}_1(p_1) \neq \mathcal{L}_2(p_2)$, cela signifie que les processeurs de coordonnées p_1 et p_2 ne sont pas placées sur le même processeur.

Considérons dans cette partie deux nids de boucles T^r et T^w, pour lesquels nous souhaitons caractériser les volumes de communications. Les données référencées par une partition émettrice et une partition réceptrice résidant sur deux processeurs différents doivent être communiquées (fig. 6.2). En utilisant la définition du volume de données approximé $\mathcal{V}^{r,w}_{com}(\mathbf{c}^r, \mathbf{p}^r, \mathbf{c}^w, \mathbf{p}^w)$ à la fois lue par une partition c^r et écrit par c^w (voir éq. (6.1)) et la définition de la fonction de linéarisation (voir éq. (6.3)), on obtient :

Définition 2.5 *Le volume* $\mathcal{V}^{r,w}_{com}(\mathbf{c}^r, \mathbf{p}^r, \mathbf{c}^w, \mathbf{p}^w)$ *de données à communiquer entre deux partitions élémentaires* $\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}$ *et* $\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)}$*, correspond à la cardinalité de l'intersection des ensembles d'éléments de tableaux référencés, lorsque celles-ci sont placées sur des processeurs différents :*

$$\forall(\mathbf{c}^r, \mathbf{p}^r, \mathbf{c}^w, \mathbf{p}^w), \quad \mathcal{L}^r(\mathbf{p}^r) \ \neq\ \mathcal{L}^w(\mathbf{p}^w), \tag{6.5}$$

$$\mathcal{V}^{r,w}_{com}(\mathbf{c}^r, \mathbf{p}^r, \mathbf{c}^w, \mathbf{p}^w) \ = \ \text{card}\ \Big(\ \mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \cap \mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)})\Big) \tag{6.6}$$

La figure (fig. 6.2) exprime le problème de la non uniformité des communications. Dans cet exemple cyclique, n'apparait pas le paramètre L, pour plus de clarté. La fonction de linéarisation est triviale, car l'exemple est mono-dimensionnel. On remarque que le couple (c^w, c^r) joue un rôle important dans les volumes de données engendrés par les accès au tableau `T`.

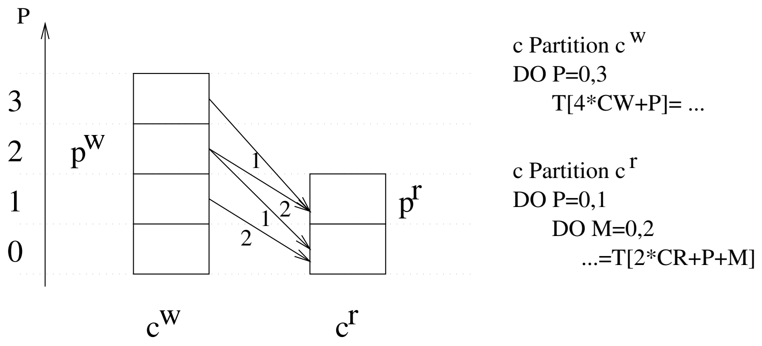


FIG. 6.2. – *Communications non uniformes entre différentes partitions élémentaires*

2.4 Volume de communication élémentaire

La section précédente permet d'obtenir un volume de communications par couple de processeurs pour chaque couple de partitions dépendantes. Si l'on souhaite en plus de cela intégrer le nombre de messages échangés entre les processeurs requis par deux partitions temporelles, on peut sommer sur l'ensemble des couples de différents processeurs possibles à partir de (6.5).

Définition 2.6 *Le volume de communications élémentaires* $\mathcal{V}^{r,w}_{elem}(\mathbf{c}^r,\ \mathbf{c}^w)$ *entre deux partitions temporelles* $\mathcal{P}^r_{(\mathbf{c}^r)}$ *et* $\mathcal{P}^w_{(\mathbf{c}^w)}$*, avec un réseau d'interconnexion totalement connecté est :*

$$\forall(\mathbf{c}^r,\ \mathbf{c}^w),\quad \mathcal{V}^{r,w}_{elem}(\mathbf{c}^r,\ \mathbf{c}^w) = \sum_{\{(\mathbf{p}^r,\mathbf{p}^w)|\mathcal{L}^r(\mathbf{p}^r)\ \neq\ \mathcal{L}^w(\mathbf{p}^w)\}} \text{card}\ \left(\mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \cap \mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)})\right) \tag{6.7}$$

Cette somme regroupe donc le nombre de messages échangés entre deux partitions temporelles dépendantes, ainsi que le volume de communications entre partitions élémentaires. Cette formule est dénommée volume de communication élémentaire. L'inconvénient de ce coût théorique de communications est qu'il n'intègre pas la notion de topologie, il ne fait que calculer le volume global des données à échanger entre deux partitions. La valeur 7 de l'exemple figure 6.3 est en fait la taille **maximale** des données à communiquer (fig. 6.2). En fait, on considère là un réseau d'interconnexion, ou chaque point peut lire et écrire à un autre point sans aucune contrainte. On assimile le réseau à un "crossbar"[1] permettant à deux processeurs de communiquer à un instant donné. Une topologie plus complexe entrainerait un coût de communications comportant d'autres contraintes sur la somme que la seule inéquation :

$$\mathcal{L}^r(\mathbf{p}^r)\ \neq\ \mathcal{L}^w(\mathbf{p}^w) \tag{6.8}$$

2.5 Modèle de communications

La formulation des communications élémentaires permet de définir le modèle de communications. L'objectif de ce modèle est d'évaluer un temps de communication. Si la topologie est pour l'instant

[1]Eventuellement implanté par un réseau à commutation de paquets

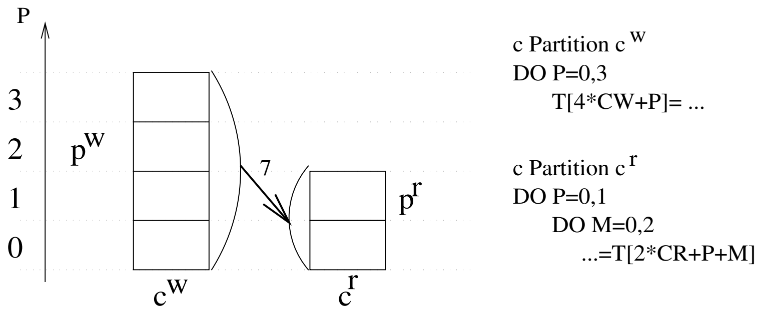


FIG. 6.3. – *Communications entre deux partitions temporelles*

fixée (9, § 2.3.1, p. 153), comme précisée ci-dessus, il reste un certain nombre de choix concernant le mode et le type de communications, exposés ci-dessous. Dans cette section, l'ensemble des éléments permettant de calculer le temps de communication est présenté. En premier lieu, le nombre total de données communiquées dépend des volumes de communications élémentaires calculées en (6.7) et du mode de communications. Il y a deux modes possibles :

Mode en émission où les blocs de données sont communiqués dès qu'ils sont écrits. On considère l'ensemble des données devant être émises.

Mode en réception où les blocs de données sont communiqués dès qu'une partition à besoin de les lire. On considère alors l'ensemble des données devant être reçues.

En second lieu, il faut tenir compte de deux paramètres d'architecture : la bande passante du réseau, et le temps d'accès au réseau. Enfin, le type de communications "pipelinées" ou regroupées par blocs est aussi important pour calculer les coûts de communications :

Mode de communications "pipelinées" : Les communications s'enchainent au fur et à mesure de l'exécution d'un bloc.

Mode de communications regroupées en blocs : Les communications s'exécutent entre l'écriture et la lecture des blocs de données.

A la difficulté de choisir un modèle de communications pertinent s'ajoute la coordination avec les autres modèles. Les équations précédentes, utilisent les variables de partitionnement, établissant ainsi, au niveau de l'implantation des deux modèles une relation mathématique étroite. Outre ces connexions naturelles avec le partitionnement, le lien avec l'ordonnancement événementiel est fondamental. De plus, le modèle mémoire doit faire concorder les communications avec les allocations ou les désallocations. La connexion avec le modèle de la machine est aussi important, car la complexité de l'échange des messages sur le réseau en dépend directement. Ces aspects sont étudiés ultérieurement dans les chapitres des modèles de mémoire (chap. 8), orodonnancement (chap. 7). Mais nous en donnons quelques éléments dans la section suivante (2.5.1).

2.5.1 Communications en émission

Dans cette partie, on cherche à sur-approximer l'ensemble des données qu'un bloc de calculs $\mathcal{P}^w_{\mathbf{c}^w}$ doit émettre. On se situe donc au niveau de la partition temporelle. Pour ce bloc en émission, une communication élémentaire (6.5) pour chaque bloc dépendant *en lecture* (fig. 6.4) est nécessaire. Il

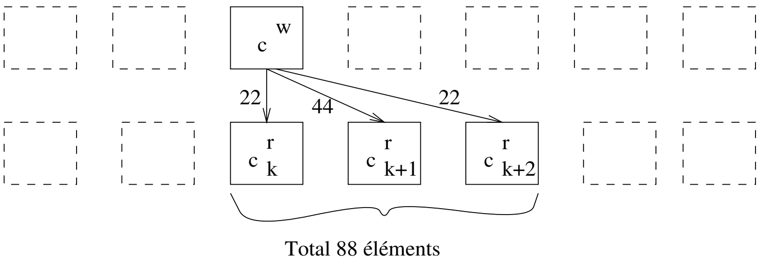


FIG. 6.4. – *Communications entre une partition temporelle en écriture, et les partitions temporelles en lecture, i.e. représentations en émission*

faut en fait tenir compte, pour une partition $\mathcal{P}^w_{\mathbf{c}^w}$, de l'ensemble des arêtes (où $\mathbf{c}^w$ est la source) du graphe de dépendances.

Définition 2.7 *L'ensemble des données à émettre pour une partition $\mathcal{P}_{(c^w)}$ est la somme des volumes des communications élémentaires $\mathcal{V}^w_{emis}(\mathbf{c}^w)$ entre $\mathcal{P}_{(c^w)}$ et les partitions $\mathcal{P}_{(c^r)}$ dépendantes :*

$$\forall c^w,\ \mathcal{V}^w_{emis}(\mathbf{c}^w) \;=\; \sum_{\forall c^r \in \mathcal{D}(c^r, c^w)} \mathcal{V}^{r,w}_{elem}(\mathbf{c}^r, \mathbf{c}^w) \tag{6.9}$$

Ce volume de communications est difficile à manipuler avec des contraintes à cause des nombreux quantificateurs universels et de diséquations (6.8).

Cependant, considèrons que le programmeur, pour certains couples de nids de boucles, notamment ceux présentant des dépendances en lecture, effectue des transpositions de tableaux où l'on communique l'ensemble des données écrites vers un processeur donné. Il est dans ce cas possible de minorer le volume de données communiquées par une contrainte redondante qui ne fait intervenir que le partitionnement. Comme les transpositions sont de lourdes communications, il est interessant de couper la combinatoire associée aux choix de partitionnements à l'aide d'un minorant des communications. Intuitivement, le volume de données à communiquer est supérieur à la différence du volume en écriture total moins le volume de données locales (devant rester sur le processeur) pour une partition donnée.

Proposition 2.8 *Soient deux nids de boucles T^r et T^w, où T^r présentent des dépendances en lecture sur des dimensions d'itérations parallèles. L'approximation uniforme du coût de communications de la transposition de tableau $\mathcal{C}^w(c^w, p^w)$ entre les blocs de calculs c^r et c^w est minorée par :*

$$\forall pe,\quad pe = \mathcal{L}(p^w),\ \mathcal{V}^w_{ecr} \;-\; \mathcal{V}^w_{loc} \;\leq\; card(\mathcal{C}^w(c^w, p^w))$$

Démonstration –

Soit pe un processeur de l'architecture physique réceptionnant toutes les communications. On a, au niveau des ensembles d'éléments de tableaux en écriture :

$$\mathcal{R}^w(c^w, p^w) = \mathcal{R}^w_{pe \neq \mathcal{L}(p^w)}(c^w, p^w) \;\cup\; \mathcal{R}^w(c^w, p^w)_{pe=\mathcal{L}(p^w)}$$

Soit $\mathcal{C}(c^w, p^w)$ l'ensemble des données à communiquer lors d'une transposition. On considère que toutes les données écrites sont utilisées par la suite par un ou plusieurs nœuds du graphe. Dans ce cas, toutes les données écrites sur d'autres processeurs sont à communiquer vers pe :

$$\mathcal{R}^w(c^w,p^w)_{pe\neq\mathcal{L}(p^w)} \;\subset\; \mathcal{C}^w(c^w,p^w)$$

Cette contrainte permet de minorer le volume de communications :

$$card(\mathcal{R}^w(c^w)) \;-\; card\left(\mathcal{R}^w(c^w,p^w)_{pe=\mathcal{L}(p^w)}\right) \;\leq\; card(\mathcal{C}^w(c^w,p^w))$$

On utilise ici la surapproximation du volume de données écrites (cf. chapitre § 8, p. 142, § 2.1.1, p. 144) :

$$\forall c^w \; \mathcal{V}^w_{ecr} \;\geq\; card(\mathcal{R}^w(c^w))$$

De même pour le volume local :
$\forall(c^w,p^w),\; \mathcal{V}^w_{loc} \;\geq\; card(\mathcal{R}^w(c^w,p^w))$
On a donc une surapproximation de la borne inférieure en ce qui concerne le volume de communications lors d'une transposition :
$\forall pe,\;\; pe=\mathcal{L}(p^w),\; \mathcal{V}^w_{ecr} \;-\; \mathcal{V}^w_{loc} \;\leq\; card(\mathcal{C}^w(c^w,p^w))$

□

La surapproximation de la borne inférieure $\mathcal{V}^w_{ecr} - \mathcal{V}^w_{loc}$ peut faire l'objet d'une fonction d'évaluation pour la recherche de couple de partitionnement pertinents et ainsi couper la combinatoire dans la recherche des partitionnements. Cette borne ne peut être utilisée que si l'on considère des transpositions, on considerera un minorant nul sinon.

2.5.2 Communications en réception

On peut considérer le problème de la même façon en définissant l'ensemble des données qu'un bloc de calculs $\mathcal{P}^r_c$ doit recevoir. Là encore, on se situe au niveau de la partition temporelle. Pour ce bloc en réception, une communication élémentaire (6.5) pour chaque bloc dépendant *en écriture* (fig. 6.5) est nécessaire. Il faut en fait tenir compte de l'ensemble des arêtes où $\mathbf{c}^r$ est la destination.

Définition 2.9 *L'ensemble des données à recevoir pour une partition $\mathcal{P}_{(c^r)}$ est la somme des volumes des communications élémentaires entre $\mathcal{P}_{(c^r)}$ et les partitions $\mathcal{P}_{(r^w)}$ dépendantes :*

$$\forall c^r,\; \mathcal{V}^r_{recu}(\mathbf{c}^r) = \sum_{\forall c^w \in \mathcal{D}(c^r,c^w)} \mathcal{V}^{r,w}_{elem}(\mathbf{c}^w,\mathbf{c}^r) \tag{6.10}$$

Encore une fois, la présence de nombreux quantificateurs universels et celle de l'inéquation (6.8) ne permet pas de manipuler directement ces formules. On peut, de la même façon que pour les émissions, minorer le volume de données à communiquer en lecture par $\mathcal{V}^r_{lec} - \mathcal{V}^r_{loc}$. Alors, des surapproximations sont obtenues comme pour le cas des communications en émissions (cf. 2.5.1).

2.5.3 Durée des communications

Le temps d'émission ou de réception d'un ensemble de données relatif à un bloc de calculs est traité dans cette section. D'une manière générale, le coût d'une communication élémentaire dépend du temps élémentaire d'accès au réseau d'interconnexion, de la bande passante du réseau et de la longueur des données à communiquer [GB94]. Soit la fonction $cout(x)$ modélisant ce coût, en nombre de cycles, et $\mathcal{V}$ un volume de données quelconque donnant la taille d'un message :

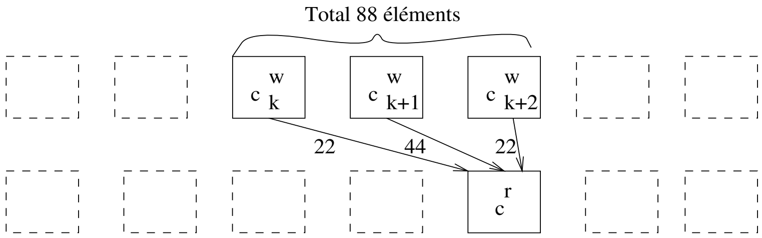


FIG. 6.5. – *Communication entre une partition temporelle en lecture et les partitions temporelles en écriture, i.e. représentation en réception*

$$cout(\mathcal{V}) \;=\; b.\mathcal{V} \;+\; a$$

avec $b,\ a,\ \mathcal{V},\ cout(\mathcal{V}) \ \in\ \mathbb{N}$.

Le paramètre b représente l'inverse de la bande passante (*i.e.* le nombre de cycles requis pour transmettre une donnée élémentaire). Le paramètre a est le temps d'accès au réseau. Le variable $\mathcal{V}$ représente la taille des données à communiquer. Le résultat est le nombre de cycles requis pour exécuter la communication.

Soit un message de $\mathcal{V}$ éléments à communiquer. Les communications peuvent être recouvertes par les calculs. Pour cela on a deux possibilités :

Modèle bloqué : L'une consiste à regrouper les communications en blocs, dans ce cas le recouvrement s'effectue entre blocs de calculs et blocs de communications *indépendants*. Le coût de transmission ne dépend que du volume de données et des paramètres d'architecture.

Définition 2.10 *Le coût de transmission d'un bloc quelquonque de données de taille $\mathcal{V}$ correspond au coup d'un accès au réseau et au temps de communication donné par la bande passante :*

$$cout(\mathcal{V}) \;=\; b.\mathcal{V} \;+\; a$$

Modèle "pipeliné" : La deuxième méthode consiste à recouvrir les blocs de calculs par ses propres communications. Le recouvrement s'effectue entre bloc de calculs et de communications *dépendants*.

Définition 2.11 *Soit le coût de transmission d'une partition de calculs effectuant $det(L)$ opérations élémentaires (cf. chapitre 4, § 3.2, p. 82). Soit le volume $\mathcal{V}'$ à communiquer pour une opération élémentaire, il y a autant d'accès au réseau que d'opérations élémentaires. Le coût de la transmission est alors :*

$$cout(\mathcal{V}') \;=\; det(L).(\mathcal{V}'.b + a)$$

L'ensemble des possibilités de communications est résumé en (fig. 6.6). Les blocs en gris représentent les calculs tandis que les blocs en noir les communications. Les flèches représentent les dépendances entre phases de communications et de calculs. Lorsque l'ensemble des blocs de calculs "dépassent" les blocs de communications, celles-ci sont dites "recouvertes". Ce recouvrement se situe à deux niveaux

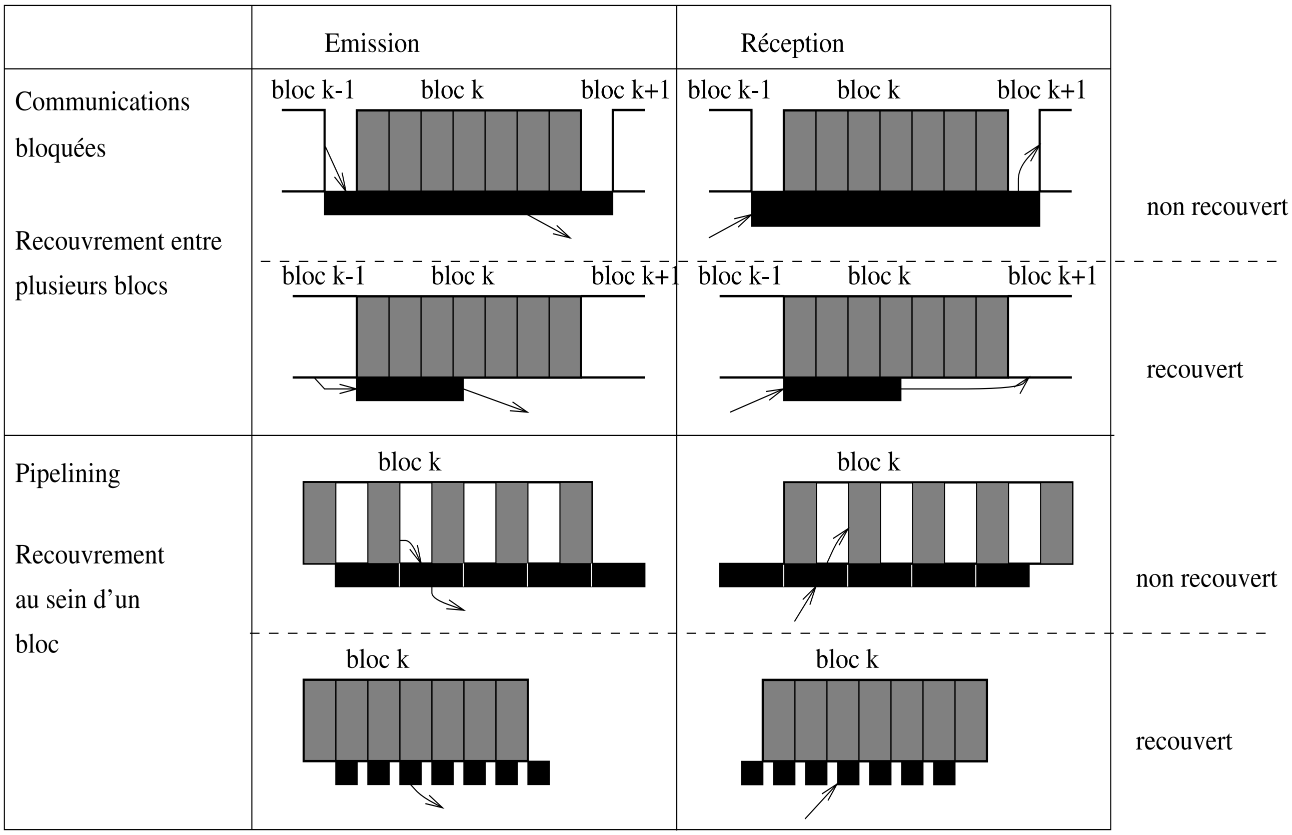


FIG. 6.6. – *Communications "pipelinées" et bloquées*

de granularité, Dans la partie supérieure de la figure, les communications sont exécutées en parallèle des blocs de calculs indexé par $(\mathbf{c^w})$. Dans la partie inférieure les recouvrement se situe aux niveaux des macro-instructions référençées par $(\mathbf{c^w}, \mathbf{p}, \mathbf{l})$.

En fait les deux types de communications ("pipelinées" ou bloquées) répondent à deux problèmes différents. En effet, si le paramètre a d'accès au réseau est très faible par rapport à b, le modèle "pipeliné" doit être considéré car il simplifie la génération de code et les buffers d'entrées ou de sorties associés sont plus faibles. Sinon, il est préférable de regrouper les communications en blocs. En fait, ces deux types de communications sont à deux niveaux de granularité différents.

La contrainte de la durée de communication peut être prise en compte au niveau du volume de communications élémentaires ou bien au niveau du volume global de communications en émision ou en réception. Cela dépend en fait de la topologie. Il faut se reporter au modèle de machine (cf. chapitre 9, § 2.3, p. 153) pour trouver des remarques à ce sujet. Les deux modèles ont été implantés, compte tenu des caractéristiques des architectures ciblées, le type de communication "pipeliné" est retenue dans la maquette et fera l'objet des expérimentations.

3 Simplifications dues aux restrictions applicatives

La prise en compte des accès sur-approximés permet de simplifier les contraintes dont le modèle de communication doit tenir compte.

3.1 Simplification des accès

Rappelons tout d'abord quelques notations :

- Φ (resp. φ) le tableau (resp. l'élément du tableau) accédé en lecture par la tâche r, en écriture par la tâche w,
- $\mathcal{I}^w$ et $\mathcal{M}^w$ (resp. $\mathcal{I}^r$ et $\mathcal{M}^r$) l'espace d'itérations externe et interne de la tâche w (resp. de la tâche r),
- I^w, M^w (resp. I^r et M^r) les matrices d'accès externe et interne de la tâche w (resp. de la tâche r)

On a $\mathrm{rang}(I^r) \leq \dim(\Phi)$ par définition du rang.

Par ailleurs dans chaque colonne de I^r, il y a au plus un élément non nul. Donc I^r est (à une permutation près des lignes) de la forme :

$$\left.\begin{pmatrix} & \backslash & \\ & & \\ & 0 & \end{pmatrix}\ \right\}\dim\ (\Phi)$$

Il en est de même pour I^w. On a de plus, pour la tâche w la restriction applicative suivante : l'application $(\mathbf{i}^w, \mathbf{m}^w) \mapsto I^w\mathbf{i}^w + M^w\mathbf{m}^w$ est injective, car on écrit qu'une seule fois.

L'ensemble des éléments de tableau référencés Φ par un appel de la procédure (voir chap. 2, § 3.1.4, p. 48) est alors

$$\mathcal{R}(\mathbf{i}) = \{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.\mathbf{i} + M.\mathbf{m}\} \tag{6.11}$$

où I et M sont des matrices associées aux applications linéaires de $\mathcal{I} \to \Phi$ (resp. $\mathcal{M} \to \Phi$).

Grâce aux restrictions précédentes (voir chap. 2, § 3.1.5, p. 49), il est possible de représenter les ensembles de références par dimension de tableau ce qui simplifie énormément le problème :

$$\mathcal{R}(\mathbf{i}) = \{(\varphi_1, \ldots, \varphi_\lambda, \ldots, \varphi_n) \mid \forall\lambda\ \ \exists(\kappa, \kappa'),\ \exists m_{\kappa'} \in \mathcal{M},\ \varphi_\lambda\ =\ I_{\lambda,\kappa}.i_\kappa\ +\ M_{\lambda,\kappa'}.m_{\kappa'}\} \tag{6.12}$$

Par substitution de (6.12), la région référencée par une partition (éq. 6.11) s'écrit alors :

$$\mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \{(\varphi_1, ..., \varphi_n) \mid \forall\lambda\ \exists(\kappa, \kappa'),\ \exists m_{\kappa'} \in \mathcal{M},\ \ \varphi_\lambda = I_{\lambda,\kappa}\,(P_\kappa L_\kappa c_\kappa + L_\kappa p_\kappa + l_\kappa) + M_{\lambda,\kappa'} m_{\kappa'}\} \tag{6.13}$$

3.2 Effets sur les modèles de communications

La formule (6.1) et toutes les formules qui en découlent se trouvent dégradées. Soit, pour une dimension de tableau λ l'ensemble des coordonées de tableaux $\mathcal{R}_\lambda(\mathbf{i})$ référencées par un accès :

$$\mathcal{R}_\lambda(\mathbf{i}) = \{\varphi_\lambda \mid \exists(\kappa,\kappa'),\ \exists m_{\kappa'},\ \varphi_\lambda = I_{\lambda,\kappa}.i_\kappa + M_{\lambda,\kappa'}.m_{\kappa'}\} \tag{6.14}$$

On peut aussi définir ces ensembles de coordonnées de tableaux référencés au niveau des partitions :

$$\mathcal{R}_\lambda(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \{\varphi_\lambda \mid \exists(\kappa,\kappa'),\ \exists m_{\kappa'},\ \varphi_\lambda = I_{\lambda,\kappa}\,(P_\kappa L_\kappa c_\kappa + L_\kappa p_\kappa + l_\kappa) + M_{\lambda,\kappa'} m_{\kappa'}\} \tag{6.15}$$

Ce qui permet, comme pour les dépendances (cf. chapitre 5, § 2.3, p. 100) de simplifier énormément la modélisation des communications sous forme de contraintes. C'est l'objet de la propriétés suivante :

Propriété 3.1 *Dans le cas des restrictions applicatives (fonctions d'accès linéairement indépendantes par dimension de tableau), le volume de données $\mathcal{V}^{r,w}_{inter}(\mathbf{c}^w,\mathbf{p}^w,\mathbf{c}^r,\mathbf{p}^r)$ référencées par deux partitions (l'une en écriture, l'une en lecture) peut se définir par un produit de produits cartésiens. Soit λ une dimension de tableau :*

$$\mathcal{V}^{r,w}_{inter}(\mathbf{c}^w,\mathbf{p}^w,\mathbf{c}^r,\mathbf{p}^r) = \prod_\lambda \mathrm{card}\left(\mathcal{R}^r_\lambda(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \bigcap \mathcal{R}^w_\lambda(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)}) \right)$$

<u>Démonstration</u> –
Les accès étant linéairement indépendants, on a :

$$\begin{aligned} &\mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \cap \mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)}) = \bigotimes\nolimits_\lambda(\mathcal{R}^w_\lambda(\mathcal{P}^w_{(\mathbf{c}^w_p,\mathbf{p}^w)_q} \cap \mathcal{R}^r_\lambda(\mathcal{P}^r_{(\mathbf{c}^r_{p'},\mathbf{p}^r_{q'})}) \\ \Rightarrow\ & card(\mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \cap \mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)})) = \prod\nolimits_\lambda card((\mathcal{R}^w_\lambda(\mathcal{P}^w_{(\mathbf{c}^w_p,\mathbf{p}^w_q)} \cap \mathcal{R}^r_\lambda(\mathcal{P}^r_{(\mathbf{c}^r_{p'},\mathbf{p}^r_{q'})})) \end{aligned}$$

Ceci simplifie l'ensemble des contraintes (6.5, 6.7, 6.9), ainsi que (6.10). Cependant cela ne permet toujours pas d'établir l'implantation pratique du modèle de communications à cause des quantificateurs universels ainsi que l'inéquation (6.8).

3.3 Choix d'un modèle de communication

Le choix d'un modèle, spécifiant si celui-ci est en émission ou en réception, comme le mode de communication (pipeliné ou bloqué) entraine d'autres choix sur les autres modèles :

Modèle du signal : Les fonctions permettant de calculer les critères de temps physique ne sont pas les mêmes car la définition du recouvrement n'est pas la même. Les fonctions de recouvrement seront abordées dans le modèle du signal (cf. chapitre 10, § 4.2, p. 168).

Modèle d'ordonnancement : De même, le lien avec l'ordonnancement est traité par le modèle d'ordonnancement (cf. chapitre 7, § 2.4, p. 134).

Modèle mémoire : L'influence du modèle de communications sur le modèle mémoire porte en outre sur la durée de vie des variables en écritures ou en lecture. Le modèle mémoire présenté au chapitre (chap. 8) considère un mode de communications par blocs en émission. La contrainte sur la durée de vie des blocs en écriture (3.2) du modèle mémoire (cf. chapitre 8, § 3.3, p. 148) est à remplacer par :

$$\forall d \quad \sum_s N_w^s(d) \leq 1$$

si le modèle doit être celui de blocs émis en pipeline. De plus, le volume des blocs en écriture, pour leur allocation, est celui du volume de données référencées par une macro-instruction, puisque l'architecture n'attend pas la fin du bloc de calculs pour émettre.

Enfin, le choix du modèle de communication en émission ou réception influence la durée de vie des variables. Dans le cas où on a un modèle en émission, la durée de vie des allocations en écriture est celle d'un événement de temps logique, les variables sont communiquées au top suivant. Il faut donc formuler la durée de vie des allocations après communications, en nombre d'événements (i.e. le temps qu'une donnée doit résider en mémoire). Dans le cas d'un modèle en réception, c'est la situation inverse : il faut mesurer uniquement la durée des allocations en écriture, puisqu'elle sont communiquées au top précedent la lecture.

Deux modèles de communications ont été implantés : le premier est un modèle par blocs en émission, le second un modèle pipeliné, aussi en émission. Compte tenu de l'architecture cible, les expérimentations sont réalisées avec ce dernier modèle.

4 Alignement

L'alignement des données correpond à une distribution particulière des données sur les processeurs. De ce fait, l'alignement des dimensions de tableaux est un problème très étudié en parallélisation automatique. Il se traduit par un ensemble de contraintes linéaires portant sur les dimensions de tableaux[ACIK93][CGSS94]. Résoudre le problème de l'alignement est en général, NP-complet[LC90].

Dans notre contexte, l'alignement des données permet d'éviter les communications. Cela correspond à des couples de partitionnement pour lesquels les données sont alignées. Ils favorisent des codes pour lequel le contrôle est simple, car en plus de l'absence de communications, il n'est pas nécessaire de faire de mouvements de données en mémoire locale. Aligner les données est donc une solution très prisée par les programmeurs de traitement du signal. Dans cette section, on cherche à caractériser les couples de partitions pour lesquels les données sont alignées.

Définition 4.1 *Soient deux tâches T^r et T^w, deux partitions élémentaires $\mathcal{P}_{(\mathbf{c^r})}$ et $\mathcal{P}_{(\mathbf{c^w})}$ sont dites alignées si les données écrites par $\mathcal{P}_{(\mathbf{c^w},\mathbf{p^w})}$ ne sont lues que par une unique partition $\mathcal{P}_{(\mathbf{c^r},\mathbf{p^r})}$. De mêmes, les données lues par une partition $\mathcal{P}_{(\mathbf{c^r},\mathbf{p^r})}$ ne sont écrites que par une seule partition $\mathcal{P}_{(\mathbf{c^w},\mathbf{p^w})}$. Ceci permet d'avoir un alignement des données sur les processeurs et dans le temps entre deux blocs de partitions séquentielles.*

Lorsque deux nids de boucles r et w sont totalement alignés, ces deux assertions donnent les contraintes :

$$\begin{cases} \forall\ (c^r,p^r),\ \exists\ !(c^w,p^w),\ |\ \mathcal{R}^r(c^r,p^r)\ \cap\ \mathcal{R}^w(c^w,p^w) = 0 \\ \forall(c^w,p^w),\ \exists!(c^r,p^r)\ |\ \mathcal{R}^w(c^w,p^w)\ \cap\ \mathcal{R}^r(c^r,p^r) = 0 \end{cases}$$

Compte tenu des quantificateurs universels, il n'est pas possible d'implanter ces deux contraintes. Cependant, ces deux contraintes définissent une bijection entre les blocs (c^w,p^w) en écriture et les blocs (c^r,p^r) en lecture. Par conséquent, une condition suffisante pour définir un alignement est donnée par :

$$\mathcal{R}^r(c^r, p^r) \;=\; \mathcal{R}^w(c^w, p^w)$$

Encore une fois, grâce aux restrictions applicatives (6.15), on peut exprimer ceci dimension par dimension de tableau k :

$$\mathcal{R}^r_k(c^r, p^r) \;=\; \mathcal{R}^w_k(c^w, p^w)$$

Ainsi, la contrainte d'égalité ensembliste se traduit par des égalités d'intervalles pour chaque dimension de tableau :

$$\forall k,\; [I^r_q L^r_q, I^r_q(L^r_q + 1)]_k \;=\; [I^w_p L^w_p, I^w_p(L^w_p + 1)]_k$$

ainsi qu'une égalité sur le nombre de processeurs pour qu'il puisse exister une même fonction de linéarisation :

$$P^r \;=\; P^w$$

La résolution de la contrainte sur les segments, entraine la contrainte linéaire d'alignement suivante :

$$I^r L^r \;=\; I^w L^w$$

Le fait d'avoir une contrainte d'alignement linéaire facilite la résolution et est un bon moyen pour connaître, au cours de la résolution concurrente très rapidement l'ensemble des L^i qui composent les partitions blocs de l'application placée.

La figure (6.7) illustre l'ensemble des notions d'alignement. Il s'agit d'aligner les données en fonction des ensembles de composantes connexes du graphe de dépendances. Si l'on a une fonction non injective sur une dimension de tableau donnée, il est impossible d'aligner des partitions de celles-ci, à moins de considérer un seul et unique bloc sérialisant l'ensemble de la dimension. En revanche, lorsque le graphe de dépendance comporte plusieurs composantes connexes, il est possible de construire des couples de partitionnement incluant ces ensembles. Alors, si les deux nids de boucles sont placés sur le même nombre de processeurs, les dépendances restent locales à chaque processeur.

La partie supérieure de la figure (6.7) illustre un cas de dépendances en lecture (accès sur les dimensions `f` et `temps`), où il est alors impossible d'obtenir un alignement sur les processeurs. La partie inférieure de la figure montre qu'en l'absence de dépendances lecture/lecture, il est possible (malgré une perte de parallélisme) de calculer des partitionnements regroupant les composantes connexes du graphe de dépendances.

Aligner les données tend à augmenter les valeurs pour les familles de matrice de partitionnement (L_i). Or, augmenter ces blocs de partitionnement revient à diminuer le parallélisme (fig. 6.7). La contrainte d'alignement n'est donc pas impérative mais elle est conseillée. Les contraintes d'alignement consituent donc des heuristiques pour la résolution du placement, d'un point de vue global. Nous reviendrons sur ces aspects dans la partie expérimentations de la thèse.

Les contraintes d'alignement obtenues sont semblables à celles définissant les ordonnancements dans le domaine SDF [LM87], puis dans la version étendu MD-SDF de Ptolemy [Che94]. Ceci montre que le problème de l'alignement est un caractère commun à l'ordonnancement des applications de traitement du signal et à la parallélisation automatique.

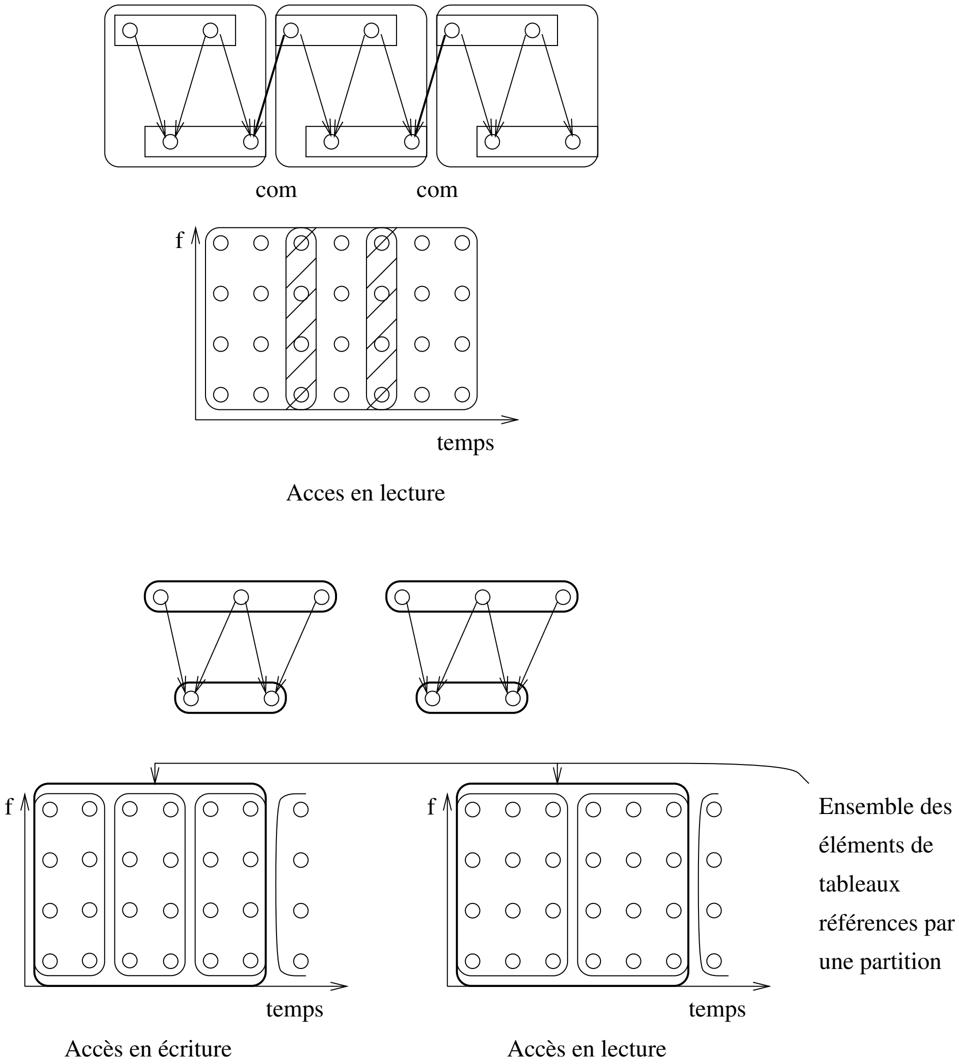


FIG. 6.7. – *Possibilités d'alignement*

5 Conclusion

Nous avons montré dans ce chapitre comment les caractéristiques applicatives permettent de définir des surapproximations de coût de communications entre nids de boucles partitionnés. Si le lien avec le partitionnement est direct – à travers les matrices de partitionnement –, le modèle de communication est conexe avec les modèles de mémoire (chap. 8) et d'ordonnancement (chap. 7). Cela concerne la synchronisation des communications et des calculs d'une part et la synchronisation des allocations et des communications d'autres part. Enfin, nous avons montré comment ces synchronisations permettaient le recouvrement des communications par les calculs pour donner un temps physique de communications selon la bande passante. De ce fait, le modèle d'interphase est connexe avec le modèle du temps physique (chap. 10) et de la machine (chap. 9).

Chapitre 7

L'ordonnancement

Le modèle d'ordonnancement pour le traitement du signal est à l'initiative de Denis Barthou *et* François Irigoin *(CRI).* J. Mattioli *(LCR) l'a repris pour garder le caractère concurrent de la relation avec les dépendances. Après avoir fait plusieurs tentatives d'implantation, j'ai proposé une condition suffisante de croissance qui simplifiait (et allégeait) considérablement à la fois la modélisation, la résolution et l'implantation. Ceci m'a alors permis de construire des distances événementielles simplifiant les modèles connexes comme celui du signal.*

1 Introduction

Nous avons présenté la décomposition de chaque dimension d'un nid de boucles en trois composantes dans le chapitre sur le partitionnement. Chaque vecteur d'itérations est décomposé en trois composantes (en un triplet $(\mathbf{c}, \mathbf{l}, \mathbf{p})$), dont la composante temporelle est représentée par $\mathbf{c}$. Nous étudions ici le séquencement logique (ordre d'apparition des différents événements) relevant de cette composante. Pour cela, le modèle d'ordonnancement événementiel attribue une date logique d'exécution unique à chacun de ces blocs. Ainsi, l'ensemble des calculs présents dans un bloc seront exécutés à la même date événementielle. En l'absence de dépendances lecture/écriture, il n'y a pas d'ordre *à priori* dans l'exécution des blocs représentés par le vecteur $\mathbf{c}$. L'ordonnancement global considère les dates logiques d'exécution de l'ensemble des blocs composant le programme. Pour être légal, l'ordonnancement global doit satisfaire les précédences impliquées par les dépendances de flots entre opérations de différents nids de boucles.

La définition d'un ordonnancement événementiel permet de décoreller le temps réel résultant des choix de partitionnement et d'ordonnancement de la légalité de celui-ci et d'utiliser un modèle capacitif de temps physique. Ainsi, l'ordonnancement événementiel permet de résoudre le problème des dépendance indépendament du temps réel. Cette décomposition correspond à la philosophie de l'approche qui consiste à découper le problème global en sous-problèmes jusqu'au niveau de maîtrise requis pour résoudre le problème.

Ordonnancement et approches transformationnelles

Il est bon de rappeler ici qu'une alternative à la définition d'un ordonnancement événementiel est de considérer une approche transformationnelle basée par exemple sur la fusion de boucles pouvant réordonner l'exécution des instructions du nid de boucles[1]. La motivation du choix pour un ordonnancement événementiel se situe en fait au niveau de l'approche globale :

- La résolution des fusions peut être effectuée localement, lorsque le problème est coupé de tout contexte [KM93, SM96, PW96]. Cependant, la combinatoire propre à cette technique est basée sur des transformations de graphes. Lesquelles ne permettraient pas au modèle de fusion de coopérer efficacement avec les autres modèles. En outre, les modèles caractérisant les performances en terme de mémoire, latence et acquisition pour le couple Architecture-Application sont au coeur du problème. En effet, la fusion regroupe nos modèles d'allocation de communication et d'ordonnancement car elle restructure directement le code. En fait, les solutions pour les autres modèles dépendraient fortement d'une solution quasi-complète de fusion.
- D'un autre coté, la fusion de boucles englobe un ensemble de problèmes fondamentaux pour le placement puisqu'elle permet de regrouper les dimensions de tableaux et réordonne l'exécution des blocs partitionnés. Dans ce sens, la fusion est complémentaire du partitionnement qui a pour rôle d'adapter les traitements à la machine cible. Le couple fusion/partitionnement caractérise l'ensemble des choix de placement

Ordonnancement événementiel affine

Le soucis de conserver l'aspect automatique, incrémental et global du placement en se basant sur une technologie offrant des primitives concurrentes nous a ammené à éclater les disjonctions liées à la fusion ainsi que les impacts sur la mémoire et le temps réel. Un modèle mémoire, un ordonnancement et un modèle de temps réel sont engendrés. De plus, un ordonnancement événementiel peut être représenté par un ordonnancement affine, où les dates d'exécution sont définies par une fonction affine des itérateurs [Fea92a, DR92, DR94a]. Trouver un tel type d'ordonnancement optimal a été prouvé NP-dur [DR94c]. L'ordonnancement affine a été prouvé quasi-optimal [DKR91] lorsque les dépendances sont représentées par un unique vecteur. L'avantage de l'ordonnancement affine est de définir en **intention** la relation d'ordonnancement par une expression algébrique. La taille de la représentation varie seulement en fonction du nombre de boucles et du nombre d'instructions et non en fonction du nombre d'opérations ou des dimensions de l'architecture. En outre, les ordonnancements affines permettent d'exprimer la plupart des ordonnancements utilisés manuellement, mais il faudrait cependant augmenter le nombre de dimension du partitionnement pour les couvrir tous.

Contrairement à l'usage courant de l'ordonnancement affine où les dates d'exécution portent sur des instructions, la date d'exécution porte ici sur un **bloc** d'instructions indicé par $\mathbf{c}$. C'est l'originalité de son utilisation au sein de cette approche concurrente. Cependant, la concurence entre partitionnement et ordonnancement est aussi, comme nous le montrerons, la cause de nombreux problèmes. Enfin,

[1]Le choix entre un ordonnancement événementiel et une approche transformationnelle à fait l'objet d'un long débat passionné. Je résume ici les différents avis ainsi que la décision finale que nous avons prise.

l'ensemble des ordonnancements affines considérés dans un même repère temporel doit représenter un ordonnancement *légal* de l'application globale, c'est à dire qui ne modifie pas les dépendances de flots. Le résultat est en fait l'exécution cyclique d'une séquence de blocs de calculs. Cette séquence correspond à un ensemble de boucles mal imbriqués pouvant être obtenus à partir d'un ensemble de transformations *légales* de programmes dont la fusion de boucles. Afin de limiter la complexité du code ainsi obtenu, nous serons amenés à utiliser des heuristiques du *domaine de la parallélisation* (chap. 13) permettant de cibler les ordonnancements souhaités.

Ce chapitre présente le modèle d'ordonnancement affine. On attribue à chaque nid de boucle partitionné une fonction (affine) en (§ 2, p. 132), définissant ainsi un ordonnancement local. La prise en compte de l'ensemble des fonctions sur le même axe de temps logique nécessite de prendre en compte les contraintes de précedences (§ 3, p. 135). Nous donnons des optimisations possibles pour réduire le nombre de contraintes (§ 4.3, p. 209). Enfin, la section (§ 4, p. 138) soulève le problème des surapproximations par rapport aux solutions pertinentes.

2 Modèle d'ordonnancement

L'adaptation de l'ordonnancement affine à notre contexte nécessite de définir une fonction affine d'ordonnancement par nid de boucles. Chaque fonction affine obtenue définit un ordonnancement vers une même dimension temporelle. La fonction affine porte donc sur le vecteur $\mathbf{c}$, si bien que tous les types d'ordonnancement affine peuvent être envisagés. Cependant, l'ordonnancement affine est restreint à la simple représentation de l'exécution séquentielle des blocs, puisque le parallélisme est déjà exprimé avec le partitionnement. A cette fin, la fonction affine doit être **strictement croissante** par rapport à un ordre lexicographique donné et non pas simplement croissante. Deux blocs de calculs définis par $(\mathbf{c_1}, \mathbf{c_2})$ ne peuvent pas s'exécuter simultanément.

2.1 Fonction d'ordonnancement affine local à un nid de boucles

Pour chaque nid de boucles, on choisit ici une forme affine entière $d(.)$ faisant correspondre à un bloc de calculs, une date événementielle :

$$d(\mathbf{c}) = \alpha.\mathbf{c} + \beta \tag{7.1}$$

où α est un vecteur ligne[2] de même dimension que $\mathbf{c}$, l'opération . représente le produit scalaire usuel et β est un entier. Une solution pour le modèle d'ordonnancement correspond à l'attribution des valeurs aux vecteurs **alpha** et aux scalaires **beta** de l'ensemble des nids de boucles.

2.2 Ordonnancement local

Pour imposer le fait que deux calculs ne s'exécuteront pas au même moment, afin de garantir l'injectivité de la fonction $d(.)$ définie en (7.1), nous allons supposer que **la fonction d(.) est strictement croissante**. Cette croissance est valable grâce aux caractéristiques applicatives des applications définies en (chap. 2, § 3.1.5, p. 49), en particulier, du fait que la matrice d'accès I est positive. Pour cela, on

[2]Prenons un nid de boucles avec un ordonnancement $d(\mathbf{c}) = \alpha.\mathbf{c} + \beta$. Si α est scalaire, c'est-à-dire que le nid ne contient qu'une seule boucle, alors ce dernier va faire un calcul tous les α tops. Entre deux tops de calculs, il y aura au plus $\alpha - 1$ calculs provenant des autres nids de boucles.

munit arbitrairement[3] le domaine $[\mathbf{0}, up(\mathbf{c}) - \mathbf{1}] \cap \mathbb{N}^n$ d'une relation d'ordre total, comme par exemple l'ordre lexicographique défini par : $\forall \mathbf{c} = (c_1, ..., c_n)^t$ et $\forall \mathbf{c}' = (c'_1, ..., c'_n)^t$, on a :

$$\mathbf{c} \ll \mathbf{c}' \Leftrightarrow (\exists j, \quad \forall i \leq j \quad c_i = c'_i \text{ et } c_{j+1} < c'_{j+1}) \tag{7.2}$$

Proposition 2.1 *Soit $d(.)$ une forme affine sur $[\mathbf{0}, up(\mathbf{c}) - \mathbf{1}] \cap \mathbb{N}^n$ définie par $d(\mathbf{c}) = \alpha.\mathbf{c} + \beta$ où $\alpha = (\alpha_1, ..., \alpha_n)$. Alors $d(.)$ est strictement croissante pour l'ordre lexicographique si et seulement si*

$$\alpha_i > \sum_{j>i} \alpha_j \, (up(c_j) - 1) \tag{7.3}$$

$$\alpha_n \geq 1 \tag{7.4}$$

<u>Démonstration</u> – Soit $\ll$ l'ordre lexicographique. Montrons que

$$\forall \mathbf{c}, \mathbf{c}' \quad \mathbf{c}' \ll \mathbf{c} \Longrightarrow d(\mathbf{c}') < d(\mathbf{c})$$

En particulier, si $up(c_i) - 1 > 0$, prenons $\mathbf{c}$ tel que $c_j = 0$ pour $j \neq i$ et $c_i = 1$, et soit $\mathbf{c}'$ tel que $c'_j = 0$ pour $j \leq i$ et $c'_j = up(c'_j) - 1$ pour $j > i$. Alors $\mathbf{c}' \ll \mathbf{c}$, ce qui implique, pour $i < n$, on a :

$$\alpha_i > \sum_{j>i} \alpha_j \, (up(c_j) - 1) \quad \text{ et } \alpha_n \geq 1$$

Si $up(c_i) - 1 = 0$, on peut alors choisir α_i vérifiant les inégalités.

Réciproquement, si les inégalités sur α sont vérifiées et si $\mathbf{c}' \ll \mathbf{c}$, alors on a de manière évidente $d(\mathbf{c}') < d(\mathbf{c})$.
□

Remarque. Ceci implique nécessairement que $\forall i, \ \ \alpha_i > 0$.

Nous donnons ici une seconde hypothèse de croissance, simplifiant la modélisation et les relations avec les autre modèles.

Si $\alpha_n = 1$, on peut montrer que (7.3) est équivalente à

$$\alpha_i \geq \prod_{j>i} up(c_j) \tag{7.5}$$

En effet, posons $\Lambda(i) = \prod_{j>i} up(c_j) - \sum_{j>i} \alpha_j \, (up(c_j) - 1)$. Supposons que $\alpha_n = 1$ alors $\Lambda(n-1) = 1$ et on montre par récurrence que $\Lambda(i) = 1$, d'où

$$(\alpha_n = 1) \Longrightarrow \left(\forall i < n, \quad \prod_{j>i} up(c_j) = \sum_{j>i} \alpha_j \, (up(c_j) - 1) + 1 \right)$$

[3]Si le nid de boucles possède n boucles, alors le domaine $[\mathbf{0}, up(\mathbf{c}) - \mathbf{1}] \cap \mathbb{N}^n$ est un ensemble borné de $\mathbb{N}^n$. On veut alors munir cet ensemble d'un ordre total qui sera nécessairement isomorphe à un ordre lexicographique. Il existe alors $n!$ ordres lexicographiques, qui correspondent chacun à une permutation des axes de $\mathbb{N}^n$. Il faudra donc considérer par la suite toutes les permutations possibles.

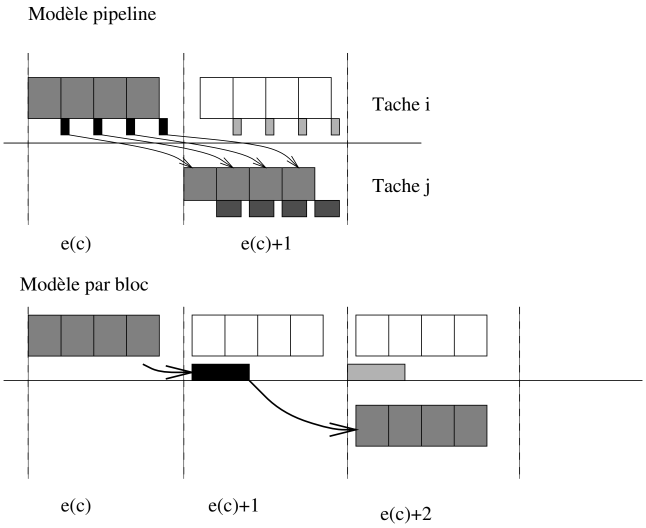


FIG. 7.1. – *Evénements de communications pour les modèles en émission*

Pour chaque nid de boucles, la fonction d'ordonnancement (locale) est une forme affine $d(.)$ strictement croissante qui associe le temps local au temps événementiel, *i.e.*

$$\forall c \in [\mathbf{0}, up(\mathbf{c}) - \mathbf{1}] \cap \mathbb{N}^n, \quad d(\mathbf{c}) = \alpha.\mathbf{c} + \beta \;\text{ où } \alpha = (\alpha_1, \alpha_2, ..., \alpha_n) \in \mathbb{N}^n, \; \beta \in \mathbb{N}$$

$$\text{t.q.} \quad \begin{cases} \alpha_n \geq 1 \\ \alpha_i > \sum_{j>i} \alpha_j \, (up(c_j) - 1) \end{cases}$$

2.3 Dates d'exécution des itérations

De par la sémantique du modèle de partitionnement, l'ensemble des itérations indicées par un vecteur $\mathbf{c}$ seront exécutées à la même date événementielle $\delta(\mathbf{i})$, on a donc :

$$\forall \mathbf{i} \; \in \; \mathcal{P}(\mathbf{c}) \; \Leftrightarrow \; d(\mathbf{c}) \; = \; \delta(\mathbf{i})$$

2.4 Ordonnancement des communications

Nous avons indiqué (cf. chapitre 6, § 2.5, p. 118) toutes les possibilités de modèle de communications. Le type de communications "pipeliné" ou "bloqué" conditionne la position de l'événement de communications par rapport aux calculs dont il dépend (fig. 7.1). Si l'on devait traiter les modèles de communications en réception, on aurait des formulations semblables, la démarche pourrait être reprise selon ce même schéma si besoin etait.

2.4.1 Modèle de communications par blocs

Dans le cas d'un modèle de communication par bloc émis, l'événement porte sur la communication des données produites par la *totalité* du bloc de calculs.

Définition 2.2 *Soit l'événement de communication* $e(\mathbf{c})$ *correspondant à l'émission de l'ensemble des données produite par la totalité d'un bloc :*

$$e(c) = d(c) + 1$$

De la même façon, dans le cas d'un modèle de communication par bloc reçus, l'événement porte sur la communication des données lues par la *totalité* du bloc de calculs.

Définition 2.3 *Soit l'événement de communication* $e(\mathbf{c})$ *correspondant à la reception de l'ensemble des données lues par la totalité d'un bloc :*

$$e(c) = d(c) - 1$$

2.4.2 Modèle de communications "pipeline"

Dans le cas d'un modèle de communication par "pipeline", l'événement porte sur la communication des données lues ou écrites pendant la phase de calcul.

Définition 2.4 *Soit l'événement de communication* $e(\mathbf{c})$ *correspondant à l'émission ou la réception de l'ensemble des données lues ou écrites par la totalité d'un bloc, em mode "pipeliné" :*

$$e(c) = d(c)$$

Dans la suite du document nous considérons le modèle de communications par blocs, nénamoins, pour retrouver les formules d'ordonnancement avec communication en pipeline, il faut systématiquement ôter le $+1$ aux formules d'ordonnancement.

3 Ordonnancement global

L'ordonnancement global légal d'un programme défini par q nids de boucles est donné par une famille de q fonctions $(d^k)_{k=1}^{q}$ affines locales à chacun des nids de boucles, telles que celles-ci respectent les dépendances de flot. On peut donc représenter la fonction d'ordonnancement global en considérant toutes les fonctions d'ordonnancement locaux sur le même axe de temps événementiel sous réserve de préserver les précédences (fig. 7.2).

3.1 Implication des dépendances pour les contraintes de précédences

Les contraintes de précédence portent sur les dates d'exécution des opérations. Elles découlent des dépendances puisqu'il s'agit de trouver une exécution qui ne modifie pas les dépendances de flot, c'est à dire qui reste légale.

$$\forall \mathbf{i}^w, \mathbf{i}^r, \quad D(\mathbf{i}^w, \mathbf{i}^r) \Longrightarrow d^w(\mathbf{i}^w) \leq d^r(\mathbf{i}^r)$$

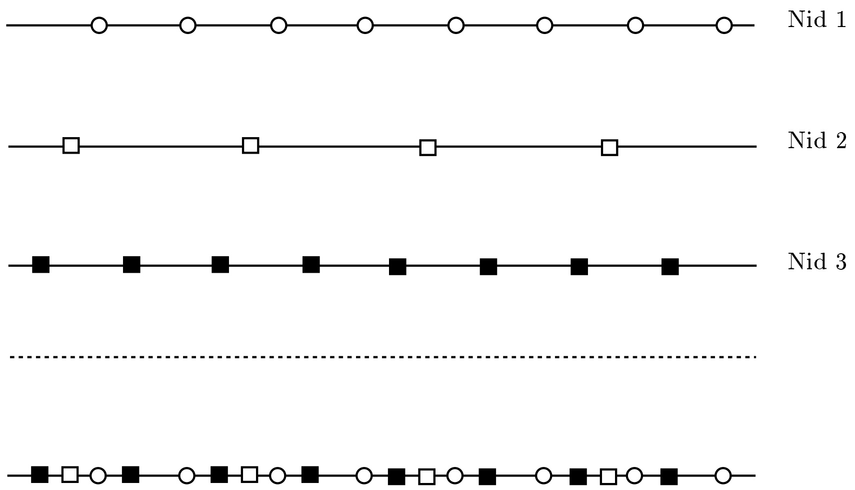


FIG. 7.2. – *L'ordonnancement global est une "projection" sur l'axe temps événementiel de chaque ordonnancement local*

Comme l'ensemble des itérations indicées par un vecteur **c** seront exécutées à la même date événementielle, on peut substituer $d(\mathbf{i})$ par $d(\mathbf{c})$, en prenant en compte l'événement de communications :

$$\forall \mathbf{i}^w, \mathbf{i}^r, \quad D(\mathbf{i}^w,\ \mathbf{i}^r) \Longrightarrow d^w(\mathbf{c}^w)\ <\ e(\mathbf{c}^w)\ <\ d^r(\mathbf{c}^r)$$

d'où

$$\forall \mathbf{i}^w, \mathbf{i}^r, \quad D(\mathbf{i}^w, \mathbf{i}^r) \Longrightarrow d^w(\mathbf{c}^w) + 1 < d^r(\mathbf{c}^r)$$

Nous raisonnons sur les dates d'exécution des blocs. Nous considérons dans toute la suite le modèle de communications par blocs émis. Pour cela, il est nécessaire de rappeler la notion de dépendance de flots entre blocs de calculs. L'intersection des ensembles de références devient alors :

$$\forall \mathbf{c}^w, \mathbf{c}^r, \quad D(\mathbf{c}^w,\ \mathbf{c}^r) \Rightarrow \mathcal{R}^w(c_w)\ \cap\ \mathcal{R}^r(c_r)\ \neq\ \emptyset$$

On peut alors formuler la contrainte de précédence impliquée par les dépendances au niveau des blocs de calculs, lorsque le modèle de communications est par blocs émis :

$$\forall \mathbf{c}^w, \mathbf{c}^r, \quad D(\mathbf{c}^w,\ \mathbf{c}^r) \Longrightarrow d^w(\mathbf{c}^w) + 1\ <\ d^r(\mathbf{c}^r) \tag{7.6}$$

Pour information, il est possible d'avoir le même type de résultat lorsque les précédences avec le modèle de communications est par blocs "pipelinés", l'ordonnancement devient :

$$\forall \mathbf{c}^w, \mathbf{c}^r, \quad D(\mathbf{c}^w,\ \mathbf{c}^r) \Longrightarrow d^w(\mathbf{c}^w)\ <\ d^r(\mathbf{c}^r) \tag{7.7}$$

3.2 Représentation des contraintes de dépendances et de précédences pour la résolution concurrentielle

La contrainte (7.6) présente un inconvénient majeur : l'opérateur $\forall$. Comme il est impossible de raisonner pour chaque couple $(c^w,\ c^r)$, il est nécessaire de raisonner sur une représentation en intention des dépendances. Pour cela, introduisons la manipulation des systèmes d'inéquations et des polyèdres que les dépendances, associés à l'ordonnancement affine engendrent.

Proposition 3.1 *Soit $P(x)$ et $Q(x)$ deux systèmes d'inéquations linéaires de variables x définissant des polyèdres convexes. Supposons que $P(.) = \{x \in \mathbb{Z}^n \mid P(x)\}$ définisse un ensemble borné, alors l'assertion à valeurs entières*

$$[\forall x, \quad P(x) \Rightarrow Q(x)]$$

est équivalente à :

$[Q(y)$ où y prend les valeurs des points extrémaux de l'enveloppe convexe entière de $P(.)]$.

Il est cependant à noter qu'un système d'inéquations linéaires à valeurs entières, défini nécessairement un polydèdre convexe entier (comme intersection de demi-espaces), de plus on utilise l'enveloppe convexe de ce polyèdre à valeurs entières,les points extrémaux sont dans $P(.) = \{x \mid P(x)\}$. La seule hypothèse importante sur l'ensemble induit par le système $P(x)$ est donc qu'il soit borné.

Démonstration – On veut montrer que sous les hypothèses où $P(.)$ défini un sous-ensemble d'un polydèdre convexe borné, on a :

$$[\forall x, \quad P(x) \Rightarrow Q(x)] \iff [\forall y \text{ sommets de } \mathrm{conv}(P(.)), \quad Q(y)]$$

où conv(.) représente l'enveloppe convexe entière.

- L'implication $\Rightarrow$ est immédiate car $\forall y$ sommets de $\mathrm{conv}(P(.))$, y est un point de $P(.)$ (par définition de $\mathrm{conv}(P(.)))$, d'où : $[P(y) \Rightarrow Q(y)] \wedge P(y)$ donc on a en particulier $Q(y)$.
- Démontrons la réciproque, *i.e.* l'implication $\Leftarrow$.
  Comme $P(.)$ est inclus dans $\mathrm{conv}(P(.))$ qui est un polyèdre borné, pour tout point x de $P(.)$, il existe deux sommets s_1 et s_2 de $\mathrm{conv}(P(.))$ et il existe $\lambda \in [0,1]$ tels que $x = \lambda s_1 + (1-\lambda)s_2$ avec $P(s_1)$ et $P(s_2)$. De même, pour tout couple de sommets (s_1, s_2), supposons qu'il existe λ tel que $\lambda s_1 + (1-\lambda)s_2 \in P(.)$, comme $P(.)$ est sous-jacent à un système d'inéquations linéaires, donc

  $$[P(s_1) \wedge P(s_2)] \Rightarrow [\lambda \in [0,1],\ P(\lambda s_1 + (1-\lambda)s_2)]$$

  De plus, on sait que $Q(.)$ est aussi sous-jacent à un système d'inéquations linéaires on a

  $$[Q(s_1) \wedge Q(s_2)] \Rightarrow [\lambda \in [0,1],\ Q(\lambda s_1 + (1-\lambda)s_2)]$$

  Ce qui montre que

$$\left(\begin{array}{c} (\forall x \text{ sommet de } \mathrm{conv}(P(.)), \quad Q(x)) \\ \Longrightarrow \\ (\forall s_1, s_2 \text{ sommets de } \mathrm{conv}(P(.)),\ \forall \lambda \in [0,1],\ \lambda s_1 + (1-\lambda)s_2 \text{ point de } P, \\ Q(\lambda s_1 + (1-\lambda)s_2)) \\ \Longrightarrow \\ (\forall x,\ P(x) \Rightarrow Q(x)) \end{array}\right)$$

□

Proposition 3.2 *La contrainte (7.6) de précédence est équivalente à :*

$$\forall(\mathbf{c}^w, \mathbf{c}^r) \textit{ sommets de } \operatorname{conv}(D), \quad d^w(\mathbf{c}^w(\mathbf{i}^w)) + 1 \ < \ d^r(\mathbf{c}^r(\mathbf{i}^r)) \tag{7.8}$$

où $\operatorname{conv}(D)$ *est l'enveloppe convexe entière de l'ensemble des itérations dépendantes.*

On remarque que $\operatorname{conv}(D)$ est un convexe inclus dans l'espace $\prod_{k=1}^{q} \mathcal{I}^k \times \prod_{k'=1}^{q'} \mathcal{I}'^{k'}$. En fait, dans certains cas, il est distinct de $D(.,.)$ qui peut posséder des trous (du fait qu'il existe des variables libres engendrant les itérations $\mathbf{m}$ intra-procédurales et des fonctions d'accès possédant des modulos).

Démonstration – En fait cette proposition est une réécriture de la proposition (3.1). De manière directe :

- Comme les sommets de $\operatorname{conv}(D)$ sont dans $D(.)$, alors l'implication dans un sens est immédiate.
- Il est clair que s'il n'y a pas de dépendance entre deux blocs séquentiels $\mathbf{c}^w$ et $\mathbf{c}^r$, *i.e.* $\neg D(\mathbf{i}^w, \mathbf{i}^r)$ alors la contrainte (7.6) est toujours satisfaite.
- Tout couple de blocs séquentiels dépendants est un point de $\operatorname{conv}(D)$ car $\operatorname{conv}(D)$ est l'enveloppe convexe associée à $D(.,.)$ donné par l'équation (5.2) (donc contient cet ensemble) et la contrainte est (7.6) satisfaite. De plus, en utilisant les propriétés de l'enveloppe convexe entière, on sait que tout point de $D(.,.)$ s'écrit alors comme combinaison linéaire des sommets de $\operatorname{conv}(D)$

□

4 Problèmes soulevés par l'approximation des sommets

4.1 Cas simple de dépendance

L'approximation des sommets par rapport aux classes d'ordonnancements parcourues engendre un effet pervers. Dans le cas où la forme du polyèdre de dépendances se résume à une droite, l'ordonnancement résultant devrait être très simple car obtenu par deux inéquations de précédence. Cet ordonnancement est très prisé par les ingénieurs lorsqu'ils effectuent un placement manuel car il correspond à une simple fusion de boucles, et ne nécessite pas un code complexe. En effet, si l'on tient compte de l'interprétation graphique du polyèdre de dépendances, un bloc en lecture dépend (par le flot) uniquement d'un bloc en écriture (cf. l'utilisation de la variable S_k du chapitre 6, § 4.6, p. 216) lorsque l'on obtient une droite, il n'y a donc que des communications entre couples de blocs de calculs temporels. On peut faire la même remarque pour la mémoire : il est facile de calculer l'allocation d'un tableau intermédiaire lorsqu'il n'est lu que par une seul bloc de calculs. Malheureusement, les sommets approximés ne permettent pas de représenter cette configuration et donc d'en déduire l'ordonnancement correspondant. Les sommets du polyèdre de dépendances ne permettent pas d'envisager de dépendances bloc à bloc, c'est à dire sous la forme d'une bijection (cf. chapitre 6, § 4, p. 126). Nous sommes dans la situation où les approximations considérées soustraient des solutions opérationnelles très pertinentes (en outre, celles considérant les alignements), c'est le danger des sur-approximations importantes.

La preuve consiste à montrer que les dépendances, telles qu'elles sont définies, ne peuvent être représentées par une droite, pour chacun des polygones constituant le polyèdre. Ainsi, cette définition empêche d'obtenir des solutions "alignées". Pour comprendre la preuve géométriquement, il faut se reférer aux sommets exprimés (cf. chapitre 5, § 2.4, p. 102).

<u>Démonstration</u> – On a $min(x_B, up(x)) > 1$ car la valeur x_B est une valeur absolue supérieure d'un quotient strictement positif. Il y a au moins un calcul dans la boucle concernée, donc $up(x) \geq 1$ Les point B et C ne peuvent être l'origine du repère.

De même pour la valeur $min(y_H, up(y))$ car la valeur y_H est une valeur absolue supérieure d'un quotient strictement positif. De plus, il y a au moins un calcul dans la boucle concernée, donc $up(y) \geq 1$. Les points H et G ne peuvent être l'origine du repère.

En conséquence les points des ensembles $\{H, B\}, \{H, C\}, \{G, B\}, \{G, C\}$ ne peuvent pas être confondus. Par le même type de raisonnement, on démontre qu'au moins un troisième point n'est pas confondus avec H, B, G, C, donc le polyèdre approximé ne peut pas être une droite. □

4.2 Solution proposée

Comme nous l'avons expliqué, le cas de dépendances “bloc à bloc” ou bijectives est très avantageux pour le placement global, tant du point de vue des communications que de l'allocation, car un bloc en lecture dépend d'un bloc en écriture. Interdire ce type de dépendances implique la constuction d'ordonnancements sous-optimaux (qui pénalisent communications, mémoire, et donc latence et temps réel). Un tel cas risque de diminuer les performances du système de placement et d'enlever les solutions préconisées par les experts du domaine. Il faut donc dans ce cas, relaxer les contraintes de sur-approximation tout en préservant la légalité de l'ordonnancement. Considérons la contrainte portant sur les droites D_{inf} et D_{sup} :

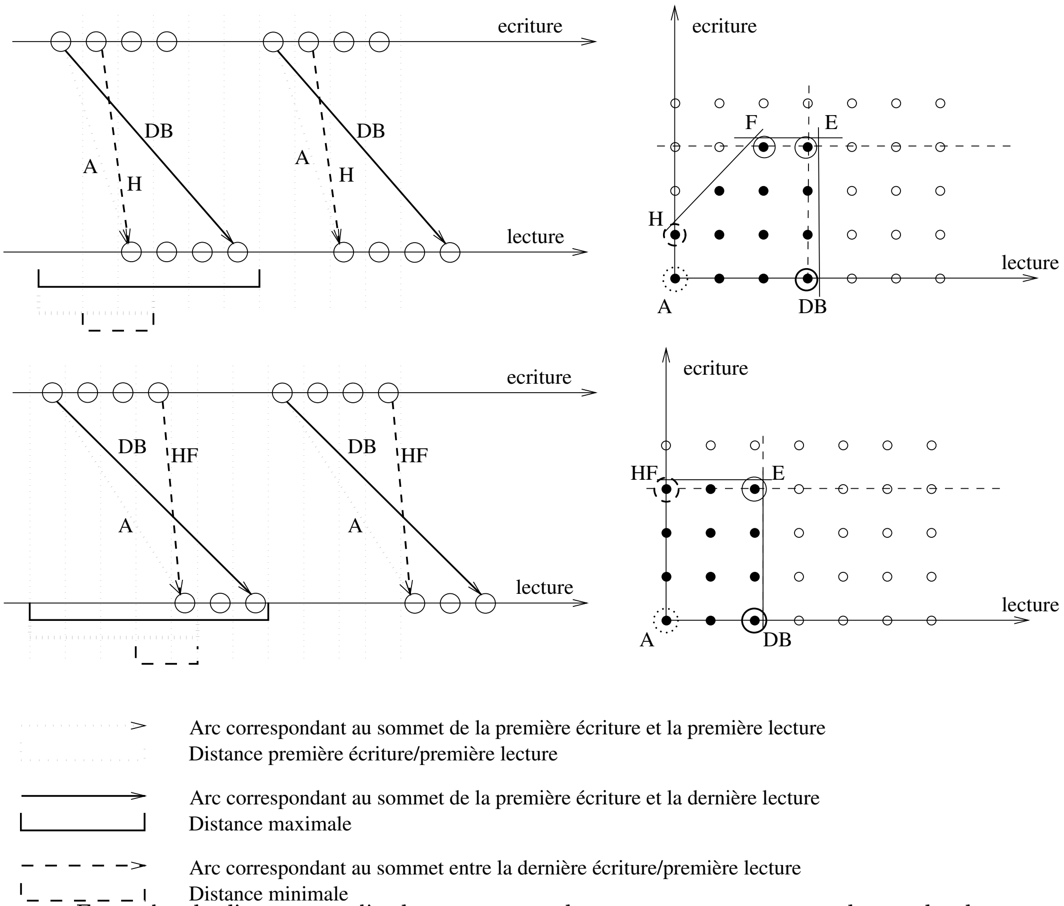
$$confondues(D_{inf}, D_{sup}) \Rightarrow \mathcal{P}_k \;=\; \{A, E\}$$

Le prédicat $confondu$ repose sur les coefficient de l'équation des dépendances (cf. chap. 5, 5.5, (§ 2.3, p. 100). On peut aussi considérer les points $\{F, L\}$ (cf. chap. 5, § 2.4, p. 102). Directement implanté, cette contrainte permet de considérer uniquement les points pertinents lorsque la bande se résume à une droite. La résolution de ce problème prouve ici la souplesse de la PLC lors de la relaxation, dans un cas particulier d'une approximation trop forte. Cependant, ce problème et sa solution prouvent que le nombre de sommets à prendre en compte pour optimiser le placement varie pendant la résolution, même en considérant des approximations. Nous avons ici un exemple de non-monotonie du comportement de certaines contraintes fondamentales pour le placement, ici c'est le cas de la contrainte de précédence.

Nous donnons, dans les chapitres d'expérimentations plusieurs cas dépendant de cette relaxation (chap. 15, p. 276).

5 Distance événementielle

De nombreux travaux ont montré que satisfaire des contraintes temps-réels (cf. chapitre 10), revenait à satisfaire des contraintes temporelles de distance sur les ordonnancements, définissant ainsi des périodes d'activations de tâches ou des deadlines à satisfatire [LL73] [HL89] [CSR89] [HLL95] [XSR+96], même lorsqu'il s'agit d'ordonnancement cycliques [HMR+89] [HLH96]. Pour pouvoir permettre aux autres modèles, et notamment le modèle temps réel du signal (chap. 10, p. 158), d'utiliser ces distances, nous en définissons un ensemble pertinent où chacune est représentée en intention par des contraintes sur des ensembles de sommets.



Exemples de distance sur l'ordonnancement de composantes connexes du graphe de dépendance

FIG. 7.3. – *Interprétation du polyèdre de dépendances*

5.1 Les trois distances utilisées

Les distances événementielles peuvent être définies à partir du moment où le type d'ordonnancement est donné. Ce sont des intervalles de temps donnés en nombre d'événements. Ces distances ne peuvent être calculées statiquement à partir des polygones de dépendances puisque ceux-ci évoluent dynamiquement au cours de la résolution. La programmation avec contraintes, malgré cet inconvénient permet de définir des contraintes de distances, basées sur les sommets de ces polygones.

La figure (7.3) permet de visualiser la correspondance entre sommets et événements ordonnancés. Contrairement à la figure (5.3, p. 104) du chapitre (5), l'ordonnancement et les polygones de dépendance permettent d'établir des distances de dépendances. Ces distances sont permettent d'établir des relations entre modèles. Par exemple, la distance maximale permet de donner la durée de vie des allocations, que nous etudierons au chapitre mémoire (chap. 8).

5.1.1 La distance première écriture/première lecture

Cette distance correspond à l'intervalle de temps logique entre la production de la première donnée par un bloc de calculs en écriture et sa première utilisation par un bloc de calcul en lecture. Comme le sommet A du polyèdre de dépendances est toujours le point de coordonnée $(0,0)$, cette distance est définie par :

$$d^{w,r}_{prem} = d^r(0) - d^w(0)$$

d'où

$$d^{w,r}_{prem} = \beta^r - \beta^w$$

5.1.2 La distance maximale

La distance événementielle maximale de dépendance est l'intervalle de temps maximal séparant l'exécution de deux blocs dépendant au sens du flot :

$$d^{w,r}_{max} \;=\; max\ \{\ d(\mathbf{c^w_s}) - d(\mathbf{c^r_s}),\ \ \mathcal{D}(\mathbf{c^w_s},\ \mathbf{c^r_s})\ \}$$

Cette distance est utilisée pour caractériser la durée de vie des variables écrites et communiquées par un bloc en écriture, seule la durée de vie maximale est considérée. Cela correspond à l'intervalle de temps maximal en événements séparant l'écriture d'une donnée et sa dernière utilisation (cf. chapitre 8, § 3.3, p. 148) :

$$d^{w,r}_{max} \;\geq\; \max_{(w,r)}\left(\ d^r(\mathbf{c}^r(\mathbf{i}^r)) - d^w(\mathbf{c}^w(\mathbf{i}^w))\ \right)$$

La distance maximale est aussi utilisée dans le modèle de calcul de la latence, elle permet de définir le plus long chemin sur les composantes connexes du graphe de dépendances (cf. 10, § 5.2.2, p. 170).

5.1.3 La distance minimale

La distance minimale est utilisée pour exprimer l'heuristique du plus court chemin, c'est le temps minimum en événements entre la production d'une donnée et son éventuelle utilisation.

$$d^{w,r}_{min} \;=\; min\ \{\ d(c^r_s) - d(c^w_s),\ \mathcal{D}(\mathbf{c^w_s},\ \mathbf{c^r_s})\ \}$$

6 Conclusion

Ce chapitre a permis d'établir un ordonnancement affine événementiel, définissant les dates d'exécutions des blocs séquentiels. Cet ordonnancement est représentatif de ceux obtenus manuellement, lorsque l'on considère l'architecture cible à un niveau de repésentation élevé. Le choix de la fonction croissante, comme nous l'étudierons en (chap. 10) est important pour la modélisation, cart elle permet d'établir une contrainte de cardinalité. La résolution d'un ordonnancement légal nécessite l'utilisation des dépendances de flot (chap. 5) et interagit ainsi avec les modèles de mémoire (chap. 8), de temps physique (chap. 10) et de partitionnement (chap. 4). Ces liens sont simplifiés grâce aux notions de distances, pour le modèle mémoire et de temps physique. En revanche, le lien entre partitionnement et ordonnancement, qui est la caractérisation des dépendances (chap. 5) est un bon exemple de l'utilisation des polyèdres paramétrés.

Chapitre 8

Modèle mémoire

À partir d'une première formalisation du modèle d'allocation mémoire faite par Denis Barthou *et* François Irigoin *(CRI),* Bertrand Jeannet *(LCR) a établit le modèle capacitif. En revanche c'est avec l'aide de* Corinne Ancourt *(CRI) que j'ai exprimé le calcul des volumes de données. Le passage à l'implantation a nécessité des raffinements du modèle mémoire, en utilisant les contraintes de distances et la contrainte de cardinalité que je n'aurais jamais su spécifier sans de longues discussions avec* Bertrand Jeannet.

1 Introduction

Dans un contexte de machine à mémoire distribuée, la mémoire disponible sur un processeur peut être assimilée à un tableau permettant de stocker le résultat des calculs en attendant leur utilisation lors de traitements suivants. L'allocation des données d'un programme consiste à attribuer une position à chaque donnée utilisée dans ce tableau. Pour calculer cette allocation, il est bien entendu hors de question de représenter chaque élément de tableau par une variable (c'est à dire en extension).

Une première possibilité consiste à utiliser une fonction d'allocation représentant, là encore, en intention les données du modèle. Le système construirait alors automatiquement la fonction d'allocation. Le partitionnement, d'une part, par l'intermédiaire de la distribution des données a une grande influence sur la taille des blocs de données à allouer en mémoire. D'autre part, l'ordonnancement, de par sa capacité à différer les traitements, fait varier les dates d'allocation et de libération de ces blocs.

Une deuxième possibilité consiste à construire un modèle capacitif, garantissant une allocation calculée en post-traitement. Le modèle de mémoire que nous allons désormais définir comporte deux niveaux de granularité. Le plus fin définit la taille d'un bloc à allouer, tandis que le plus haut niveau détermine les blocs des différents traitements dont le système doit disposer à un instant t. Les

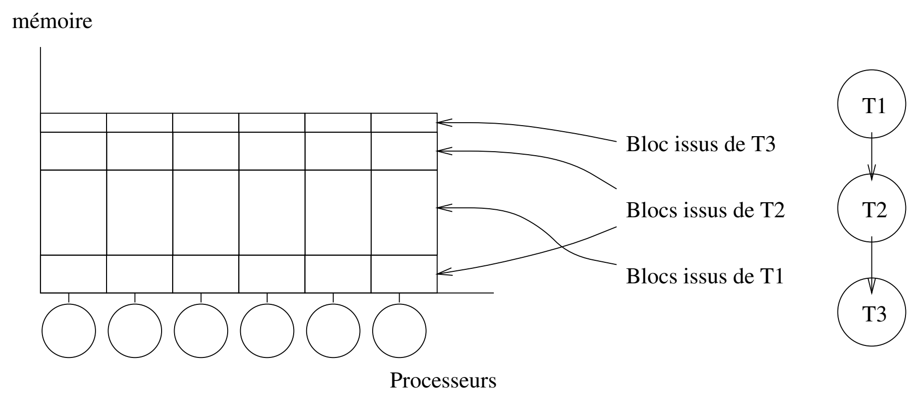


FIG. 8.1. – *Etat de la mémoire à un instant donné pour l'application composée des traitements T1, T2, T3*

contraintes portent alors sur les blocs. Néanmoins, la taille mémoire dépend bien de ces deux niveaux.

Considérons la mémoire de chaque processeur comme un tableau multidimensionnel dans lequel on affecte les blocs de données. Toute cellule de ce tableau est dénommé emplacement mémoire. Pour être correct, le modèle d'allocation doit satisfaire deux types de contraintes. Les contraintes d'anti-dépendances garantissent qu'un emplacement mémoire dont le contenu doit encore être utilisé, ne sera pas modifié. Les contraintes de dépendance en écriture sont aussi à satisfaire, c'est le cas où deux opérations écrivent sur un même emplacement mémoire. L'analyse de ces dépendances a fait l'objet de nombreux travaux [Fea91][MAL93], différentes méthodes ont été conçues pour traiter ce types de dépendances dans les fonctions de parallélisation et placement [PW86][GL95][Wol96].

Ce chapitre est consacré aux contraintes définissant les volumes des ensembles d'éléments de tableaux référencés (§ 2, p. 143) et à la localisation de ces ensembles dans la mémoire distribuée (§ 3, p. 147).

2 Volume de données

Nous distinguons deux types de volume concernant les accès aux tableaux. L'un permet de savoir combien de données sont nécessaires à l'exécution d'un bloc de calculs sur un processeur donné. L'autre serait utile à la définition d'un modèle de gestion de cache. Bien que ce dernier ne soit pas encore défini, il est bon d'en donner une de ses composantes au sein du modèle qui s'en rapproche le plus.

2.1 Volume de données référencées

Le volume de données s'obtient à partir des ensembles de données référençées par un processeur et une partition temporelle. Il s'agit donc de dénombrer les ensembles définis en (4.3.1). Nous considérons dans cette partie, le cas de partitionnement bloc-cyclique (affectation $(\mathbf{c}, \mathbf{p}, \mathbf{l})$), le volume de données sur-approximé $\mathcal{V}_{ref}$ référençé par une partition bloc à l'instant logique $d(\mathbf{c})$ et sur le processeur $\mathbf{p}$:

$$\mathcal{V}_{ref}(\mathbf{c}, \mathbf{p}) \;=\; card(\mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}))$$

$$\Leftrightarrow$$

$$\mathcal{V}_{ref}(\mathbf{c},\mathbf{p}) \;=\; card(\Big\{\varphi \mid \varphi \in \Phi,\ \exists \mathbf{l},\ 0 \leq L^{-1}.\mathbf{l} < 1\ \exists \mathbf{m} \in \mathcal{M},\ \varphi = I.(L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}) + M.\mathbf{m}\Big\}) \tag{8.1}$$

La notation $\hat{\mathcal{V}}_{ref}$ sera utilisée dans d'autres parties de la thèse comme le volume exact, tandis que $\mathcal{V}_{ref}$ est utilisé pour définir les approximations, on a naturellement : $\hat{\mathcal{V}}_{ref} \;<\; \mathcal{V}_{ref}$. Car tout au long de ce chapitre, nous calculons des sur-approximations.

Nous expliquons aussi pourquoi le volume de données calculé dans le cas d'une affectation $(\mathbf{c},\mathbf{l},\mathbf{p})$ entre difficilement dans le cadre de la programmation par contraintes. Enfin, le volume des blocs est calculé spécifiquement dans le cas des restrictions applicatives présentées en (chap. 2, § 3.1.5, p. 49).

2.1.1 Approximation uniforme des blocs de données et de leur volume

Il s'agit alors de trouver une sur-approximation de cet ensemble qui ne dépende ni du vecteur $\mathbf{c}$ ni du vecteur $\mathbf{p}$. En effet, on cherche à allouer un nombre très important de ces blocs et encore une fois, il est impossible d'avoir une représentation non uniforme du volume des blocs comme il est impossible d'associer une variable à chaque bloc de mémoire. Une sur-approximation uniforme du volume de données référencées par bloc est donc nécessaire. Cette approximation ne doit dépendre ni du vecteur $\mathbf{c}$, ni du vecteur $\mathbf{p}$.

En éliminant les quantificateurs $\exists$ de l'ensemble, on obtient :

$$R_{(\mathbf{c},\mathbf{p})} \;\in\; \{\varphi \mid \varphi \in \Phi,\ I.(L.P.\mathbf{c}+L.\mathbf{p}+\mathbf{l}) \leq\; \varphi\; \leq I.(L-1)+M.up(\mathbf{m}-1)+I.(L.P.\mathbf{c}+L.\mathbf{p}+\mathbf{l})\} \tag{8.2}$$

L'ensemble obtenu est une sur-approximation de l'ensemble précédent définissant un polytope convexe invariant par translation. Le volume de ce polytope ne dépend donc pas de cette translation. Cependant, cette représentation comporte un problème majeur : elle considère des points non accédés. En effet, en traitement du signal, les tableaux peuvent être sous-échantillonés entraînant des accès aux tableaux non convexes. De plus, le polytope obtenu n'est pas l'enveloppe convexe du bloc de données. Les accès étant exprimés dimension de tableau par dimension de tableau, cette approximation n'a qu'un faible rôle. En effet, les fonctions d'accès I et M diagonales à une permutation près (chap. 2, § 3.1.5, p. 49), l'enveloppe convexe des blocs de données référencées sont des hyper-parallépipèdes rectangles dont les arêtes sont parallèles aux axes. Ces hyper-parallépipèdes sont proches de la sur-approximation définie par l'équation (8.2) qui est aussi un hyper-parallépipède rectangle dont les arrêtes sont parallèles aux axes. En revanche, pour raffiner l'approximation, on considère les accès locaux aux blocs. L'équation

$$\varphi \;=\; I.\mathbf{l} + M.\mathbf{m}, \quad 0 \leq L^{-1}.\mathbf{l} < 1, \quad \mathbf{m} \in \mathcal{M}$$

définit ces accès, que l'on peut aussi écrire :

$$\varphi \;=\; \begin{pmatrix} I \\ M \end{pmatrix} . \begin{pmatrix} \mathbf{l} \\ \mathbf{m} \end{pmatrix} \; 0 \leq L^{-1}.\mathbf{l} < 1, \quad \mathbf{m} \in \mathcal{M}$$

La technique consiste à utiliser la matrice d'Hermite pour calculer la base de l'image constituée par les composantes diagonales de la matrice $\begin{pmatrix} I \\ M \end{pmatrix}$ [Sch86].

$$\varphi \;=\; P.H.Q.\mathbf{im}$$

où P, Q sont des matrices unimodulaires, P est une matrice de permutation.

$$\varphi \;=\; P.H.\mathbf{im},\;\; \mathbf{im} \;=\; Q.\begin{pmatrix} \mathbf{l} \\ \mathbf{m} \end{pmatrix}$$

Cette décomposition produit une matrice H triangulaire inférieure. Soit $\hat{H}$ la matrice composée de la diagonale de H disposant des éléments diagonaux de la matrice d'Hermite associée à Ω, l'équation définissant une sur approximation de l'accès se réécrit :

$$\varphi \;=\; \hat{H}.\mathbf{im}$$

Ce qui permet à l'approximation (8.2) d'être raffinée en sur-approximant l'équation (8.1) par :

$$R_{(\mathbf{c},\mathbf{p})} \;\subset\; \{\varphi \mid \varphi \in \Phi,\;\; \varphi \;=\; \hat{H}.\begin{pmatrix} \mathbf{l} \\ \mathbf{m} \end{pmatrix}, \tag{8.3}$$

$$I.(L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}) \leq \;\varphi\; \leq I.(L-1) + M.up(\mathbf{m}-1) + I.(L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l})\}$$

L'avantage de cette sur-approximation est que $\hat{H}$ peut être calculée statiquement – dans le cas de l'affectation $(\mathbf{c}, \mathbf{p}, \mathbf{l})$ –, de plus l'approximation est uniforme car elle ne dépend pas du couple de vecteurs $(\mathbf{p}, \mathbf{c})$. Pour approximer le volume de données référencées, on choisit de considérer $\mathbf{p} = 0,\; \mathbf{c} = 0$

$$\forall(\mathbf{c},\; \mathbf{p}),\;\; \mathcal{V}_{ref}(\mathbf{c},\mathbf{p}) \;=\; \mathcal{V}_{ref}(\mathbf{0},\mathbf{0})$$
$$\text{et}$$
$$\mathcal{V}_{ref} \;=\; card(\{\varphi \mid \varphi \in \Phi,\; \varphi \;=\; \hat{H}.\begin{pmatrix} \mathbf{l} \\ \mathbf{m} \end{pmatrix},\; \mathbf{0} \leq \;\varphi\; \leq I.(L-1) + M.up(\mathbf{m}-1)\})$$

Comme $\hat{H}$ est une matrice diagonale, le volume de données référencé est :

$$\mathcal{V}_{ref} \;=\; \frac{\Pi_{i=1}^{n}\; (\mathbf{I.(L-1) + M.up(m-1)})_i}{\det(\|\hat{H}\|)} \tag{8.4}$$

Dans la thèse, les notations $\mathcal{V}_{ecr}^{t}$ et $\mathcal{V}_{lec}^{t}$ représentent respectivement les volumes des ensembles d'éléments de tableaux écrits et lues par les partitions d'un nid de boucle t.

2.1.2 Problème posé par les autres affectations

Le calcul du volume de données dépend naturellement du type de partitionnement déterminé par les choix d'affectation exprimés dans le modèle de partitionnement (cf. chap. 4, § 4.2.2, p. 92). Le deuxième choix d'affectation retenu est $(\mathbf{c}, \mathbf{l}, \mathbf{p})$ (cf. chap. 4, § 4.2.2). Alors, l'ensemble à distribuer sur un processeur est :

$$\mathcal{R}(\mathcal{P}_{(\mathbf{c},\mathbf{p})}) = \left\{\varphi \mid \varphi \in \Phi,\; \exists \mathbf{l},\; 0 \leq P^{-1}.l < 1\; \exists \mathbf{m} \in \mathcal{M},\; \varphi = I.(L.P.\mathbf{c} + L.\mathbf{l} + \mathbf{p}) + M.\mathbf{m}\right\}$$

Alors, la fonction d'accès interne à une partition située sur un processeur sera de la forme :

$$\varphi \;=\; I.L.\mathbf{l} + M.\mathbf{m}, \quad 0 \;\leq\; P^{-1}.\mathbf{l} \;<\; 1, \quad \mathbf{m} \;\in\; \mathcal{M}$$

Contrairement à l'affectation $(\mathbf{c}, \mathbf{p}, \mathbf{l})$ traitée précedement, la fonction d'accès est non linéaire puisqu'elle dépend de la matrice L qui n'est pas connue. La matrice d'Hermite associée ne permet donc plus de calculer statiquement l'approximation voulue. Ceci rend l'implantation de la distribution $(\mathbf{c}, \mathbf{l}, \mathbf{p})$ difficile en Programmation par Contraintes. Nous n'avons pas, aujourd'hui, déterminer de solutions satisfaisantes à ce sujet.

2.1.3 Dégradations dues au domaine applicatif

Le domaine applicatif permet à nouveau de dégrader la cardinalité (§ 8.4) grâce aux fonctions d'accès qui sont linéairement indépendantes sur chacune des dimensions. En effet, dans le cas de l'affectation par blocs $(\mathbf{c}, \mathbf{p}, \mathbf{l})$ chaque composante de la diagonale de cette matrice s'exprime statiquement,[1] :

$$\hat{h_{l,l}} = pgcd(I_{l,c}, M_{l,c'})$$

Le volume de données peut alors être calculé en fonction de ces *pgcd* :

$$\mathcal{V}_{ref} = \left\lceil \frac{\Pi_{l=1}^{n}(\mathbf{I.(L-1)+M.up(m-1)})_l}{\Pi_{l=1}^{n} pgcd(I_{l,c}, M_{l,c'})} \right\rceil \tag{8.5}$$

Malgré sa non-linéarité, cette contrainte de volume peut être implantée dans un langage de contraintes. Par ailleurs, la prise en compte de ces restrictions ne permettent pas le calcul des volumes de données dans le cas de l'affectation $(\mathbf{c}, \mathbf{l}, \mathbf{p})$, puisque les éléments de la diagonale sont la forme :

$$\hat{h_{l,l}} = pgcd(L.I_{l,c}, M_{l,c'})$$

Ceci entraînerait une contrainte de volume non-linéaire faisant intervenir un *pgcd* sous la forme de contraintes.

2.2 Nombre d'accès

Le nombre d'accès à un même tableau est utile pour l'élaboration d'un modèle de mémoire cache, nous le donnons ici à titre indicatif. Il est possible de le calculer pour une macro instruction, une dimension, ou l'ensemble du nid de boucles :

$$Na \;=\; \prod_{\mathbf{m}} up(\mathbf{m})$$

Sachant que l'on a Na accès par macro-instruction, on peut définir un critère équivalent par dimension ou pour l'ensemble du nid de boucles en utilisant les simples volumes de calcul définis dans le modèle du signal (cf. chapitre 10, § 4.1, p. 167).

Donnons un exemple d'utilisation de cette contrainte : à partir du nombre de références, on peut calculer par dimension le nombre de fois qu'une donnée est accédée. Il est alors possible de trier les dimensions en fonction de ce critère, les dimensions présentant le plus d'accès à une même donnée restant dans la mémoire cache.

[1] où l, c, c' sont les indices en lignes et colonnes

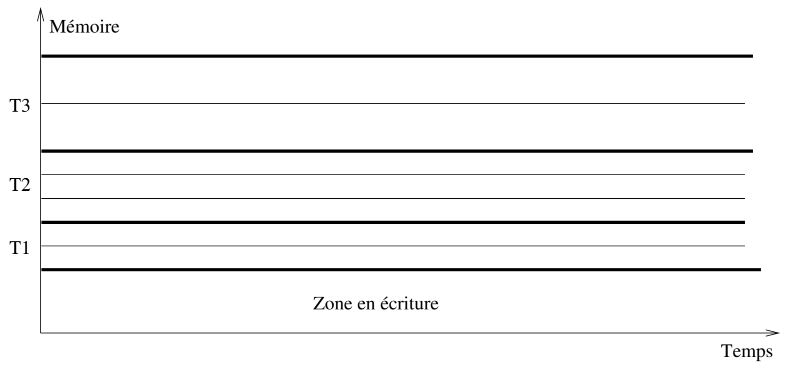


FIG. 8.2. – *Capacité mémoire dans le temps*

3 Modèle capacitif

Un modèle de mémoire capacitif est présenté dans cette partie. Il permet, pour chaque tâche, de considérer l'ensemble des blocs nécessaires en lecture et l'ensemble des blocs écrits. Il est prévu de calculer l'allocation mémoire est calculée en post-traitement. Une zone mémoire privée à chaque tâche est réservée sur chacun des processeurs, ceci ne permet pas la réutilisation de l'espace mémoire entre deux blocs de calculs issus de nids de boucles différents. Le modèle capacitif, compatible avec le modèle de communications, considère des communications par blocs en émission. Mais nous pourrions considérer aussi bien des blocs pipelinés en réception. D'un point de vue PLC, le modèle d'ordonnancement permet de restreindre la complexité des contraintes liées à la mémoire, car il tient compte de contraintes producteur/consommateur.

3.1 Segmentation de la mémoire

On considère un segment mémoire pour les blocs de tableaux en lecture pour chacune des tâches du graphe de flot de données et pour chacun des processeurs. La somme de tous les segments mémoire permet de donner la contrainte mémoire. On considère une capacité uniforme par processeur et dans le temps, si bien que l'on a facilement une contrainte de mémoire globale à partir du nombre de processeurs et de la taille de la mémoire locale. De plus, le modèle considère un segment mémoire particulier sur chaque processeur, auquel on associe les ensembles de données écrites avant que ceux-ci ne soient communiqués. De ce fait, à travers la contrainte de capacité, la réutilisation des segments mémoire se fait tâche par tâche.

La figure (8.2) illustre l'allocation de zones mémoires par nid de boucles. Si à un instant donné, une partie de la mémoire réservée à `T2` est libéré, seuls des blocs de calculs de `T2` pourrons réutiliser cette emplacement et non des blocs de calculs de `T1` ou `T3`.

3.2 Contrainte de mémoire générale

On note respectivement $\mathcal{V}^{\mathbf{s}}_{\mathbf{lec}}$ et $\mathcal{V}^{\mathbf{s}}_{\mathbf{ecr}}$ les volumes sur-approximés en écriture et en lecture d'une tâche s. Si le type de communications est pipeliné, il faut considérer le volume de données en écriture élémentaire à la place de $\mathcal{V}^{\mathbf{s}}_{\mathbf{ecr}}$.

3.2.1 Nombre de blocs de données en lecture actifs à un instant donné

On note $\mathbf{N_r^s}([d^t(\mathbf{c}^t), d^s(\mathbf{c}^s)])$ le nombre de blocs de calculs de la tâche s sur un intervalle d'événements $[d^t(\mathbf{t}), d^s(\mathbf{s})]$ donné qui ont alloué leur tableau en lecture pour recevoir la communication et qui ne se sont pas encore exécutés. C'est donc également le nombre de blocs de données en lecture actifs sur cette intervalle de temps pour l'ensemble des blocs de calculs $\mathbf{c}^s$.

$$N_r^s([d^t(\mathbf{c}^t), d^s(\mathbf{c}^s)]) = \text{Card}\left\{\mathbf{c}'^s \;/\; \left\{\begin{array}{l} D(\mathbf{c}^t, \mathbf{c}^s) \\ d^t(\mathbf{c}^t) + 1 \leq d^s(\mathbf{c}'^s) \leq d^s(\mathbf{c}^s) \end{array}\right.\right\} \tag{8.6}$$

3.2.2 Nombre de blocs de données en écriture actifs à un instant donné

On note $\mathbf{N_w^s}(\mathbf{d^s}(\mathbf{c^s}))$ le nombre de blocs de calculs exécutés sur l'intervalle d'événements de temps séparant l'écriture et la communication sous-jacente au bloc c^t. C'est donc le nombre de blocs de données en écriture actifs pour l'ensemble des tâches s. Leur durée de vie est 2 avec le modèle de communication considéré : ils deviennent actifs lors de l'exécution du bloc de calcul, à l'instant $d^s(\mathbf{c}^s)$, et sont communiqués au temps $d^s(\mathbf{c}^s) + 1$. Ils peuvent donc être réutilisés au temps $d^s(\mathbf{c}^s) + 2$:

$$N_w^s(d(c^s)) = \text{Card}\left\{\mathbf{c}'^s \;/\; d^s(\mathbf{c}^s) \leq d^s(\mathbf{c}'^s) \leq d^s(\mathbf{c}^s) + 1\right\} \tag{8.7}$$

3.2.3 Contrainte capacitive de mémoire générale

Proposition 3.1 *La contrainte de capacité exacte s'écrit avec les notations précédentes :*

$$\sum_{\forall(\mathbf{c}^t, \mathbf{c}^s), D(t,s)} N_r^s([d^t(\mathbf{c}^t), d^s(\mathbf{c}^s)]).\mathcal{V}_{lec}^s + N_w^s(d^s(\mathbf{c}^s))\ \mathcal{V}_{ecr}^s \leq mem \tag{8.8}$$

Le prédicat $D(t, s)$ spécifie l'existence d'une dépendance de flot sur le graphe entre deux tâches.

3.3 Contrainte de mémoire particulière

La contrainte précédente (8.8) n'est pas directement utilisable en PLC, d'une part parce qu'il faut la poser pour chaque instant $d(\mathbf{c})$ et leur nombre atteint facilement des millions (quand il est borné), d'autre part parce qu'il est très difficile d'exprimer à l'aide de contraintes $N_r^s([d^t(\mathbf{c}^t), d^s(\mathbf{c}^s)])$ et $N_w^s(d(\mathbf{c^s}))$ (cf. eq.(8.6) et eq.(8.7)). On va donc chercher une borne supérieure indépendante des couples de partitions pour chacun de ces nombres.

Proposition 3.2 *Par l'hypothèse SIMD et la durée de vie des tableaux en écriture*

$$\forall d \quad \sum_s N_w^s(d) \leq 1$$

Pour $N_r^s(d)$ l'idée est de contraindre l'ordonnancement pour limiter la durée de vie des tableaux en lecture de la tâche s à d_{max}. Nous avons présenté cette distance dans le modèle d'ordonnancement (cf. chapitre 7, § 5.1.2, p. 141). La connaissance du nombre maximal de blocs pouvant s'exécuter dans un intervalle $[d, d + d_{max}[$ permet ensuite d'en déduire le nombre maximum de blocs actifs (fig. 8.3).

Cette contrainte est aussi utilisée pour la latence et est noté $nbc^s(d, d_{max})$; on a $\forall d\ nbc(d, d_{max}) \leq nbc(0, d_{max})$ (cf. chapitre 10, § 3, p. 162), voir la figure (8.3).

En appliquant cette idée on obtient la proposition suivante :

Proposition 3.3 *Considérons une tâche t et associons-lui un entier $d^t_{max} > 0$. Si la fonction d'ordonnancement vérifie la relation*

$$\forall(s, \mathbf{c}^s)\forall\mathbf{c}^t \quad \Big(D(\mathbf{c}^s, \mathbf{c}^t) \Longrightarrow d^t(\mathbf{c}^t) - d^s(\mathbf{c}^s) \leq d^t_{max}\Big) \tag{8.9}$$

alors la durée de vie maximal des blocs est d^t et $N^t_r(d) \leq nbc^t(0, d^t_{max})$.

Démonstration – Supposons que

$$\forall(s, \mathbf{c}^s)\forall\mathbf{c}^t \quad (D(\mathbf{c}^s, \mathbf{c}^t) \Longrightarrow d^t(\mathbf{c}^t) - d^s(\mathbf{c}^s) \leq d^t_{max})$$

Cette condition permet de borner la durée de vie des blocs par d^t_{max}. Soit $N^t = nbc^t(0, d^t_{max})$ le nombre maximal de tâches t pouvant s'exécuter dans un intervalle $[d, d + d^t_{max}[$. Par bloc on entend toujours bloc en lecture de la tâche t.

Plaçons-nous à un temps d. Supposons qu'il y a $N = N^t_r(d)$ blocs actifs. Soit d_k et f_k pour $1 \leq k \leq N$ les dates de début et de fin de blocs, les f_k étant classés par ordre croissant. On a

$$\begin{aligned} \forall k\ d_k &\leq d \leq f_k \\ f_1 &\geq d \\ f_N &\leq d + d^t_{max} - 1 \end{aligned}$$

Or les dates de fin de blocs f_k correspondent au temps d'exécution des tâches qui les utilisent. N est donc inférieur ou égal au nombre $nbc^t(d, d^t_{max})$ de tâches s'exécutant dans l'intervalle $[d, d+d^t_{max}[$, et $nbc^t(d, d^t_{max}) \leq nbc^t(0, d^t_{max})$. □

On déduit de la proposition précédente une contrainte de capacité suffisante :

Proposition 3.4 *Les contraintes suivantes :*

$$\forall t \quad d^t_{max} = \max\Big\{d^t(\mathbf{c}^t) - d^s(\mathbf{c}^s) \mid D(\mathbf{c}^s, \mathbf{c}^t)\Big\} \tag{8.10}$$

$$mem \geq \Big(\sum_s nbc^s(0, d^s_{max})\mathcal{V}^s_{lec}\Big) + 2\max_s \mathcal{V}^s_{ecr} \tag{8.11}$$

impliquent (8.8) et garantissent donc que la consommation mémoire de l'application est inférieure à mem.

La figure (8.3) présente le lien entre l'ordonnancement des blocs de calcul et les allocations. La distance de dépendance (donnée à partir des calculs en gris foncé) permet de statuer quels blocs de calculs (en gris clair) nécessiterons une allocation supplémentaire pour la zone mémoire reservée au nid de boucles T^3.

3.3.1 Mémoire totale

La contrainte de mémoire donne une capacité de mémoire locale par processeur. La mémoire totale est obtenue en multipliant par le nombre de processeurs (cf. chapitre 9, § 2.1, p. 151,§ 2.2, p. 152). Si l'on avait choisi un modèle à mémoire répartie, il suffit de considérer les volumes de données référencés par la totalité d'une partition temporelle indexée par $\mathbf{c}$.

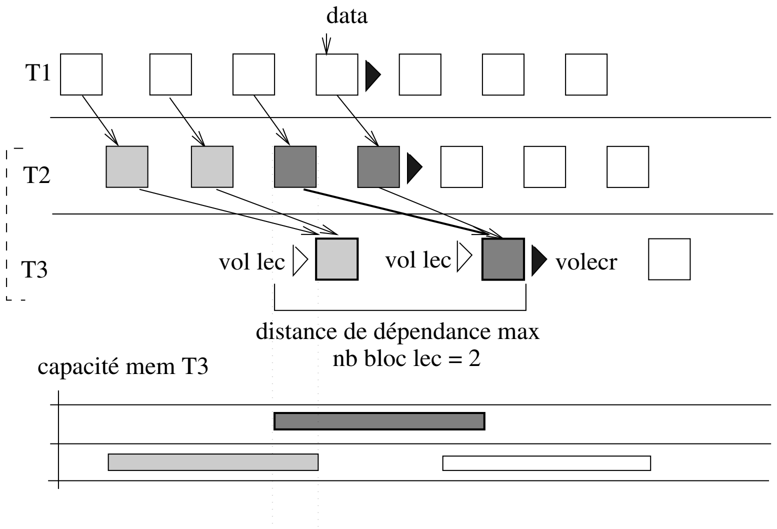


FIG. 8.3. – *Allocation mémoire en fonction de la distance maximale et du nombre d'événements et Interprétation du polytope de dépendances*

4 Conclusion

Ce modèle reflète l'utilisation de la mémoire par les programmeurs de traitement du signal ou les allocations sont définies "par nid de boucles". Il définit cependant une capacité mémoire à deux niveaux de granularité. L'un correspond aux volumes de données référencés, l'autre au niveau de la capacité de mémoire globale requise pour le fonctionnement de l'ensemble de l'application. La granularité que le partitionnement confère à l'ensemble des modèles permet au système de trouver automatiquement le seuil pertinent entre ces deux niveaux. Ce modèle est lié naturellement au modèle de machine, ainsi qu'aux modèles de partitionnement et d'ordonnancement. De ce fait, la contrainte de taille mémoire influe de façon importante sur l'ordonnancement et le partitionnement, du point de vue de la résolution. Enfin, cette contrainte contribue aussi à chercher des solutions qui se révèlent simples d'un point de vue de la génération de code.

Chapitre 9

Modèle Machine

1 Introduction

Dans cette section nous précisons le type d'architectures machines considérées. Dédiés au traitement du signal, elles disposent d'un réseau d'interconnexion haut-débit et de processeurs pouvant être spécialisés. Le modèle d'architecture machine pour le quel nous avons expérimentés l'ensemble des modèles reste général bien qu'il appartienne aux modèles SPMD à mémoire distribuée. Sur cette architecture, les ressources de calculs, mémoire et topologies sont définies uniformément, ce qui a motivé l'approche concernant l'ensemble des modèles. La généralité des modèles de partitionnement et ordonnancement nous permettrait d'envisager d'autres modèles d'architectures machines.

Ce modèle représente l'ensemble des ressources associées à l'architecture physique, qui en fait implicitement définie par les modèles de mémoire, communications et partitionnement. Le coût de la machine motive en fait la cohérence de ce modèle.

Dans ce chapitre, nous donnons les paramètres des machines pris en compte dans la modélisation et les pistes envisageables pour étendre le modèle de machine (§ 2, p. 151). Nous montrons (§ 3, p. 155) comment les contraintes spécifiques à un modèle d'architecture peuvent être simplifiées.

2 Architecture machine

Le modèle de l'architecture machine se traduit par trois paramètres fondamentaux décrivant les trois types de ressources de la machine : processeurs, mémoire et topologie. D'un point de vue modélisation, cela se traduit par des contraintes impératives (nombre de processeurs maximal, bande passante maximale ou mémoire maximale), que le placement doit respecter. Enfin, cette modélisation permet d'optimiser des fonctions de coût sur la machine.

2.1 Nombre de Processeurs

De par le modèle de partitionnement, on ne dispose seulement que du nombre de processeurs utilisés pour placer chacun des nids de boucles t. Pour chaque nid de boucles, cette valeur s'obtient grâce aux inéquations définissant l'ensemble des processeurs nécessaires au placement d'un nid de boucles t.

2.1.1 Modèle de partitionnement $(\mathbf{c}, \mathbf{p}, \mathbf{l})$

D'après le modèle du partitionnement, l'ensemble des processeurs est donné par la matrice P, pour un nid de boucles donné :

$$\forall t, \{\mathbf{p} \mid 0 \ \leq \ P^{t^{-1}}.\mathbf{p}^t \ < \ 1\}$$

Le théorème sur les volumes de parallépipèdes (3.5) du chapitre de partitionnement (cf. chapitre 4, § 3.2, p. 82), donne directement le nombre de processeurs :

$$npe^t \ = \ det(P^t)$$

en prenant en compte les restrictions applicatives (chap. 2, § 3.1.5, p. 49), on obtient :

$$npe^t \ = \ \prod_k^n P_{kk}$$

2.1.2 Modèle de partitionnement $(\mathbf{c}, \mathbf{l}, \mathbf{p})$

Dans ce cas d'affectation, c'est la matrice L qui permet de définir l'ensemble des processeurs, pour un nid de boucles donné :

$$\forall t, \{\mathbf{p} \mid 0 \ \leq \ L^{t^{-1}}.\mathbf{p}^t \ < \ 1\}$$

Le nombre de processeurs utilise le même théorème :

$$npe^t \ = \ det(L^t)$$

Mais , comme nous l'avons ennoncé dans différents chapitre, ce type de partitionnement n'est pas implanté.

2.1.3 Nombre de processeurs effectifs

Dans un deuxième temps, pour définir le nombre npe de processeurs total requis pour placer l'application, il faut considérer le nombre maximal de processeurs utilisés sur l'ensemble des nids de boucles, on obtient alors :

$$npe \ \geq \ max_t(npe^t)$$

2.2 Taille Mémoire globale

La taille mémoire définie par le modèle mémoire (cf. chapitre 8, § 2.1.3, p. 146) donne une capacité maximale uniforme par processeur. Pour prendre en compte la totalité de la mémoire, il faut considérer la contrainte :

$$maxmem \ = npe.mem$$

2.3 Topologie

Le choix d'une topologie particulière influence la nature même des autres modèles. Aussi ces modèles, comme le partitionnement et l'ordonnancement, ont-ils été définis de façon génériques pour éviter que la formalisation du placement soit dédié à une machine précise. On considère une topologie entièrement connectée à haut débit où la durée de communication entre deux processeurs ne dépend de leurs positions sur le réseau. Le paramètre de la bande passante est alors déterminant pour connaître la taille maximale des messages circulant sur le réseau (cf. chapitre 6). Néanmoins, nous pourrions très bien considérer d'autres topologies.

2.3.1 Caractéristiques de la topologie

Les caractéristiques topologiques considérées pour l'implantation permettent de mieux situer le type de machines ciblées.

Réseau de communication uniforme : La durée physique d'une communication ne dépend ni de la position de l'émetteur, ni de la position du récepteur sur le réseau de communications. Il ne peut y avoir plus d'une communication simultanément sur le réseau (fig. 9.1-1). On s'interdit donc la parallélisation des communications sur un anneau ou sur une topologie plus complexe. L'ensemble de ces restrictions permet de définir une sur-approximation uniforme de la durée des communications.

Recouvrements des calculs par les communications : Un processeur peut calculer et communiquer simultanément. Ceci permet, de tenter de recouvrir (cf. chapitre 10, § 4.2, p. 168) les communications par les calculs . Ceci permet de définir une durée physique uniforme d'exécution des partitions, c'est à dire de prendre en compte à la fois la durée des communications et la durée des calculs.

Utilisation statique : Les architectures ciblées ne disposent pas de systèmes d'exploitation et sont spécialisées de façon à privilégier une utilisation statique (allocation, partitionnement). De ce fait, le temps d'accès au réseau est infime, comparé à la bande passante, le coefficient a de la section § 2.5.3, p. 121 du chapitre d'interphase (cf. chapitre 6) peut être négligé. On note que le modèle de communication peut donc être étendu à d'autres types d'architectures, en prenant en compte un coefficient plus important. Ce coefficient n'est pas négligé dans l'implantation du modèle de communications mais n'est pas exploité expérimentalement compte tenu des caractéristiques des architectures ciblées.

2.3.2 Extension à d'autres topologies

Les architectures parallèles spécialisées pour le traitement du signal sont variées (cf. figure 9.1) . Le niveau de représentation actuel est celui de la figure (9.1-a). Considérer les architectures à un autre niveau revient à enrichir de contraintes les formules sur les volumes et durées de communications exposées dans le modèle d'interphase (cf. chapitre 6, § 2.5, p. 118,§ 2.5.3, p. 121). Ce raffinement nécessiterait, dans le cadre de notre approche, la conservation d'une représentation uniforme de l'architecture. Ceci afin que l'ensemble des paramètres du placement global puisse être exprimé toujours en intension.

Une brève description des principales architectures spécialisées "traitement du signal" pour lesquelles il est possible de raffiner le modèle de communications et ses liens avec les autres modèles (notamment, celui de l'interphase et du signal) est donnée ci-dessous :

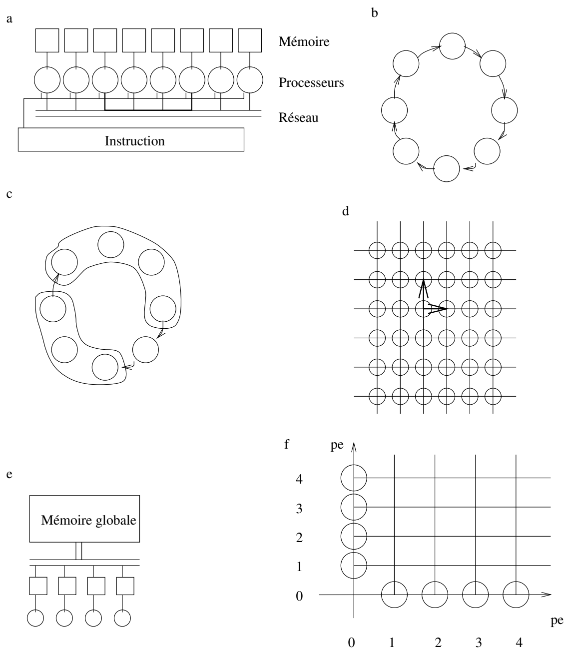


FIG. 9.1. – *Architectures spécialisées*

- La figure (9.1-b) est une topologie en anneau.
- La figure (9.1-c) est une topologie en anneau dont les processeurs sont configurés en segments, aussi appelés étages de pipe-line. On ne peut communiquer qu'entre segments de processeurs. La machine peut être configurée selon une application spécifique [JLG$^+$95][1].
- La figure (9.1-d) représente une grille de processeurs, c'est l'exemple classique de cible en matière de parallélisation automatique.
- La figure (9.1-e) est une machine à mémoire globale pour laquelle nous avons expliqué comment le modèle mémoire pourrait s'adapter (cf. chapitre 8, § 3.3.1, p. 149).
- La figure (9.1-f) est une machine disposant d'une topologie dite "crossbar" pour laquelle le temps de communication est indépendante des positions des émetteurs et récepteurs. D'ailleurs, la topologie (9.1-d) peut aussi être considérée comme un "crossbar" grâce aux méthodes de communications dites "direct packet switch".

2.4 Coût de l'architecture

La principale motivation de ce modèle est en fait son coût,qu'il est en fait possible de minimiser. Le coût est une fonction affine liant les trois variables qui caractérisent le modèle d'architecture :

$$Cout = cout(pe).npe + cout(mem).mem + cout(bp).bp$$

[1]Ce sujet à fait l'objet de la publication [JLG$^+$95], le système de placement actuel permettrait également de prendre en compte d'autres architectures.

Le paramêtre npe est le nombre de processeurs utilisés, mem représente la taille mémoire, tandis que bp, la bande passante représente le paramètre a du chapitre (6).

Les constantes $cout(pe), cout(mem), cout(bp)$ sont les coûts unitaire associés aux processeurs, mémoires et bande passante.

3 Contraintes SIMD/SPMD pour l'ordonnancement global

Les modèles sont plutôt orientés vers des machines de type SPMD, où l'ensemble des processeurs effectuent la même instruction au même moment. Ceci résulte en fait d'un choix à priori de l'architecture. Néanmoins, une certaine généricité est conservée dans l'ensemble de ces modèles. Seules les contraintes réellement SIMD sont exposées dans cette section.

3.1 Contrainte générale SIMD

La contrainte générale SIMD impose que deux blocs de calculs appartenant à des nids de boucles différents ne peuvent pas être exécutés simultanément. Etant donné la nature de l'ordonnancement affine, il faut établir une contrainte portant sur les familles de (α_k) et (β_k) pour satisfaire la contrainte SIMD.

Proposition 3.1 *Considérons un programme* `P` *ayant q nids de boucles parfaitement imbriquées. Soit $(d^k)_{k=1}^q$ la famille de fonctions affines liées à l'ordonnancement global de* `P` *(i.e. $d^k(\mathbf{c}^k) = \alpha^k.\mathbf{c}^k + \beta^k$).*

Alors les deux assertions suivantes :

1. $\forall r \neq s, \quad \beta^r \neq \beta^s$
2. $\forall r \neq s$, *il existe un commun diviseur différent de ± 1 à $\alpha_1^r, , ..., \alpha_i^r, ..., \alpha_n^r, \alpha_1^s, ..., \alpha_j^s, ..., \alpha_{n'}^s$ qui ne divise pas $(\beta^s - \beta^r)$,*

impliquent $\forall r \neq s, \quad d^s(\mathbf{c}^s) \neq d^r(\mathbf{c}^r)$.

Démonstration – On va ici encore montrer la contraposition de cette implication.

Supposons donc que $d^s(\mathbf{c}^s) = d^r(\mathbf{c}^r)$, *i.e.* $\alpha^r.\mathbf{c}^r + \beta^r = \alpha^s.\mathbf{c}^s + \beta^s$

On a vu, proposition (3.1) que cela impliquait que soit $\beta^r = \beta^s$, soit $\mathrm{pgcd}(\alpha^r, \alpha^s)$ divise $(\beta^s - \beta^r)$. Or tous les diviseurs communs (non nécessairement le plus grand) qui divisent $\alpha_1^r, ..., \alpha_n^r, \alpha_1^s, ..., \alpha_{n'}^s$ divisent aussi $\mathrm{pgcd}(\alpha^r, \alpha^s)$ donc divisent aussi $(\beta^s - \beta^r)$.

Pour cela, on utilise le lemme suivant :
Lemme technique : *Pour qu'un élément de $\mathbb{Z}$ divise simultanément $x_1, ..., x_n$ il faut et il suffit qu'il divise* $\mathrm{pgcd}(x_1, ..., x_n)$.

D'où par contraposition on a :

$$\forall r \neq s \quad \left\{ \begin{array}{l} \beta^r \neq \beta^s \\ \\ \exists \lambda \neq 1,\ \lambda \in \mathrm{cd}(\alpha^r, \alpha^s) \text{ qui ne divise pas } (\beta^s - \beta^r) \end{array} \right. \Longrightarrow d^s(\mathbf{c}^s) \neq d^r(\mathbf{c}^r)$$

où $\mathrm{cd}(\alpha^r, \alpha^s)$ est l'ensemble de tous les diviseurs communs à $\alpha_1^r, ..., \alpha_i^r, ..., \alpha_n^r, \alpha_1^s, ..., \alpha_j^s, ..., \alpha_{n'}^s$. □

3.2 Simplification de la contrainte d'architecture

3.2.1 Problèmes liés au modèle initial

Le modèle initial de l'ordonnancement $d^k(c^k) = \alpha^k.\mathbf{c}^k + \beta^k$ (modèle affine strictement croissant satisfaisant les contraintes SIMD/SPMD) a été implémenté en PLC. Après quelques expérimentations sur la première version de l'implantation, les contraintes ne présentaient pas de bonnes propriétés de propagation et étaient beaucoup trop lentes.

La formalisation mathématique de la contrainte SIMD/SPMD mettant en jeu le pgcd des α, et donc portant sur les diviseurs communs des α (voir proposition (3.1) génère en PLC un nombre très important de contraintes (de l'ordre de 300 pour la première application test : Veille Bande Large).

3.2.2 Modèle simplifié

Si on a N tâches à ordonnancer, chaque tâche reçoit arbitrairement un slot entre 0 et $N-1$. Tous les α et les β sont multipliés par N. La tâche k ($k \leq N$) est donc ordonnancée par :

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

où les applications $\mathbf{c}^k \mapsto \alpha^k.\mathbf{c}^k + \beta^k$ sont injectives.

Afin de garantir l'injectivité de la fonction $d^k(.)$ définie précédemment, c'est-à-dire imposer que deux calculs de la même tâche ne s'exécutent pas au même moment, nous allons supposer que la fonction $\mathbf{c}^k \mapsto \alpha^k.\mathbf{c}^k + \beta^k$ est strictement croissante. On obtient donc les mêmes hypothèses que celles de la proposition (2.1), c'est-à-dire un ordre lexicographique défini sur les variables $\mathbf{c}^k$:

$$\begin{aligned} \alpha_i &> \sum_{j>i} \alpha_j \, (up(c_j) - 1) \\ \alpha_n &\geq 1 \end{aligned}$$

Il faut naturellement (comme dans le cas du premier modèle) considérer tous les ordres lexicographiques (*i.e.* toutes les permutations de boucles internes à un même nid distinctes de la boucle temporelle infinie).

Par nature, ce modèle satisfaisait l'hypothèse SIMD/SPMD, c'est-à-dire que l'on a :

$$\forall r \neq s, \ \forall \mathbf{c}^s, \mathbf{c}^r, \qquad d^s(\mathbf{c}^s) \neq d^r(\mathbf{c}^r)$$

<u>Démonstration</u> – Supposons qu'il existe r, s tel que $d^s(\mathbf{c}^s(\mathbf{i}^s)) = d^r(\mathbf{c}^r(\mathbf{i}^r))$. Alors on a :

$$\begin{aligned} d^s(\mathbf{c}^s) = d^r(\mathbf{c}^r) &\Rightarrow N(\alpha^r.\mathbf{c}^r + \beta^r) + r = N(\alpha^s.\mathbf{c}^s + \beta^s) + s \\ &\Leftrightarrow N(\alpha^r.\mathbf{c}^r - \alpha^s.\mathbf{c}^s) = s - r + N\beta^s - N\beta^r \\ &\Rightarrow N \text{ divise } (s - r + N\beta^s - N\beta^r) \\ &\Leftrightarrow N \text{ divise } (s - r) \end{aligned}$$

Or on a $0 < r < N$ et $0 < s < N$, soit $-N \leq s - r \leq N$. Donc soit $r = s$, ($r = 0$ et $s = N$), soit ($s = 0$ et $r = N$), ce qui est impossible.

On a donc montré que s'il existe r, s et $\mathbf{c}^r, \mathbf{c}^s$ tels que $d^s(\mathbf{c}^s) = d^r(\mathbf{c}^r)$ alors $r = s$, et par la proposition (2.1), on a $\mathbf{c}^r = \mathbf{c}^s$. Par contraposition, on en déduit ce que la contrainte d'architecture SIMD/SPMD est naturellement vérifiée. □

3.2.3 Transformation de la contrainte de précédence avec une architecture SIMD/SPMD

Naturellement, on a : $(d^w(\mathbf{c}^w) < d^r(\mathbf{c}^r)) \Longrightarrow (d^w(\mathbf{c}^w) \leq d^r(\mathbf{c}^r))$. Montrons que s'il n'y a pas de dépendances dans un nid de boucles, et si la machine cible est SIMD/SPMD[2], on a la réciproque : *i.e.* $(d^w(\mathbf{c}^w) \leq d^r(\mathbf{c}^r)) \Longrightarrow (d^w(\mathbf{c}^w) < d^r(\mathbf{c}^r))$. En effet, la contrainte machine SIMD/SPMD implique de manière naturelle que s'il n'y a pas de dépendances dans un nid de boucles, alors on a $d^w(\mathbf{c}^w) \neq d^r(\mathbf{c}^r)$, d'où : $(d^w(\mathbf{c}^w) \leq d^r(\mathbf{c}^r)) \Longrightarrow (d^w(\mathbf{c}^w) < d^r(\mathbf{c}^r))$. On peut donc transformer le modèle associé aux contraintes de précédence en prenant en compte les événements de communication. Dans le cas où les communications sont émises par blocs :

$$\forall \mathbf{i}^w, \mathbf{i}^r, r \neq w \quad D(\mathbf{i}^w, \mathbf{i}^r) \Longrightarrow (d^w(\mathbf{c}^w) + 1 \leq d^r(\mathbf{c}^r))$$

Dans le cas où les communications sont pipelinées :

$$\forall \mathbf{i}^w, \mathbf{i}^r, r \neq w \quad D(\mathbf{i}^w, \mathbf{i}^r) \Longrightarrow (d^w(\mathbf{c}^w) \leq d^r(\mathbf{c}^r))$$

4 Conclusion

Au niveau de l'implantation, ce modèle est une liste de variables contraintes par les ressources maximales (nombre de processeurs, taille mémoire, capacité de la bande passante). Un coût de machine est une contrainte permettant de lier ces trois propriétés au sein de ce modèle. Nous avons dressé une liste représentative des architectures de machines spécialisées. Le modèle d'architecture résulte principalement d'un choix à priori et l'ensemble des modèles, malgrè une certaine généricité, sont orientés vers la programmation en mode SPMD. Les propriétés de cette architectures sont implicites au modèle de partitionnement (chap. 4) et au modèle d'interphase (chap. 6), eb ce qui concerne la fonction de coût de communications. Néanmoins, le mode de programmation SPMD s'adapte aussi aux machines de type MIMD, pour lesquelles le partitionnement 3D permet de maintenir l'équilibrage des charges. Notre approche n'est donc pas limitative.

[2]Deux calculs de différentes natures ne peuvent pas s'exécuter simultanément

Chapitre 10

Le modèle du signal temps-réel

Les bases de la formalisation de la latence ont été posées par Denis Barthou. *À partir de la condition suffisante de l'hypothèse de croissance,* Bertrand Jeannet *(LCR) et moi-même, avions suggéré une contrainte de cardinalité. Cette dernière est restée pendant plus de deux mois, à l'état de conjecture et c'est finalement* Juliette Mattioli *(LCR) qui en a fait la preuve. Cette contrainte générique a permis un raffinement du modèle mémoire. Enfin, les contraintes de distances, que j'avais introduites dans le modèle d'ordonnancement événementiel, ont considérablement simplifier le modèle signal-temps réel.*

1 Introduction

Les contraintes temps réel liées au signal permettent de garantir les dates physiques de durée de vie de l'information. Que cela concerne l'acquisition ou la restitution des résultats, il est important que la solution garantisse une spécification opérationnelle. Cette spécification prend en compte deux problèmes : l'un concerne le temps imparti entre deux exécutions, que l'on associe à l'entrée ou à la sortie d'un ou plusieurs nœuds du DFG. **La récurrence d'entrée** permet de modéliser la cadence d'aquisition du signal. C'est en fait la fréquence à laquelle les échantillons sont reçus. **La récurrence de sortie** modélise la cadence à laquelle les résultats sont produits. Le second problème est la contrainte de **latence** est le temps de calcul écoulé entre la production d'un résultat et la plus ancienne donnée acquise ayant contribuée au calcul de ce résultat.

Définir les contraintes pour le temps physique nécessite, pour de nombreux modèles, d'établir le lien avec le temps-réel. Par exemple, en ce qui concerne le partitionnement, il s'agit de donner les volumes de calculs pour connaître la durée d'exécution des partitions. Pour le modèle d'interphase, il faut associer volume de calculs et recouvrement des communications. Quand au modèle d'ordonnancement, cela nécessite l'étude détaillée d'une contrainte de cardinalité.

L'utilisation des distances dans un contexte temps réel a fait l'objet de nombreux travaux et se matérialise généralement par des deadlines à satisfaire [LL73], ou permettent de représenter un problème cyclique [HL89] [CSR89] [HLL95] [XSR$^+$96]. Les contraintes de distances temporelles portent sur des intervalles de temps matérialisant les durées de vie des entrées et sorties. Pour des ordonnancements cycliques de tâches, ces intervalles permettent aussi de définir cycliquement des intervalles de précédences [LL73][HMR+89][HLH96] entre plusieurs tâches périodiques de même nature. Dans notre contexte, les distances calculées à partir du polyèdre de dépendances et sont utilisées aussi pour définir des intervalles de temps. Cependant, ces distances établissent des liens entre plusieurs modèles concurrents (comme les dépendances et l'ordonnancement ou les dépendances et la mémoire), compliquant ainsi la résolution du problème global. Dans le cadre des algorithmes temps-réel du signal, une méthode d'ordonnancement a été proposée dans Ptolemy[LM87][Che94]. Dans ces travaux, la prise en compte des calculs en extension dans un cycle permet alors un calcul plus fin des contraintes de récurrences et de latence [SL93] [BML95] [BSL96].

Nous donnons dans ce chapitre les propriétés mathématiques des ordonnancements (§ 2, p. 159), ce qui permet d'établir la contrainte approximant un nombre d'évenéments dans un intervalle événementiel donné (§ 3, p. 162). Nous définissons alors l'ensemble des volumes de calculs (§ 4, p. 167) pour pouvoir donner les contraintes de latence (§ 5, p. 169) et d'entrées-sorties (§ 6, p. 172).

2 Propriétés mathématiques des fonctions d'ordonnancement

Pour pouvoir traiter la contrainte de dénombrement des événements, il est nécessaire d'introduire les propriétés mathématiques des fonctions d'ordonnancement, et de savoir quel type de fonction croissante il faut considérer afin de pouvoir établir une contrainte de cardinalité permettant de dénombrer les événements écoulés.

2.1 L'espace d'itérations associé à un nid de boucles

Soit $\mathcal{I}$ l'espace d'itérations associé à un nid de boucles, alors

$$\mathcal{I} = \mathbb{N} \times \{0, ..., up(c_2) - 1\} \times \{0, ..., up(c_3) - 1\} \times ... \times \times \{0, ..., up(c_n) - 1\}$$

On munit $\mathcal{I}$ de l'ordre lexicographique défini par : $\forall \mathbf{c} = (c_1, ..., c_n)$ et $\forall \mathbf{c}' = (c'_1, ..., c'_n)$, on a :

$$\mathbf{c} \ll \mathbf{c}' \Leftrightarrow (\exists j, \quad \forall i \leq j \quad c_i = c'_i \text{ et } c_{j+1} \leq c'_{j+1}) \tag{10.1}$$

Proposition 2.1 *L'ensemble $(\mathcal{I}, \ll)$ est noetherien, i.e. pour tout $\mathbf{c} \in \mathcal{I}$, toute chaîne décroissante d'origine $\mathbf{c}$ est finie. De plus,*

$$\text{Card}(\ \{\mathbf{c}' \mid \mathbf{c}' \ll \mathbf{c}\}\ ) \ = \ 1 + \sum_{i=1}^{n} c_i \left(\prod_{k>i}^{n+1} up(c_k) \right)$$

avec par convention $up(c_n) = 1$.

Démonstration – L'ensemble $(\mathcal{I}, \ll)$ est un ensemble totalement ordonné qui admet $\mathbf{0}$ comme borne inférieure. Montrons à présent qu'il est dénombrable, *i.e.* il existe une bijection de $\mathcal{I}$ vers $\mathbb{N}$. En effet, soit Υ l'application définie par :

$$\Upsilon \ : \ \begin{cases} \mathcal{I} & \mapsto \quad \mathbb{N} \\ \mathbf{c} & \to \quad \Upsilon(\mathbf{c}) = \sum_{i=1}^{n} c_i \left(\prod_{k>i}^{n+1} up(c_k) \right) \end{cases}$$

avec par convention $up(c_{n+1}) = 1$.

- Cette fonction est trivialement une application.
- Exprimons sa réciproque. Soit $p = \Upsilon(\mathbf{c})$, calculons $c = \Upsilon^{-1}(p)$.

$$\begin{aligned} p &= \sum_{i=1}^{n} c_i \left(\prod_{k>i}^{n+1} up(c_k) \right) \\ &= c_n + c_{n-1} up(c_n) + c_{n-1} up(c_n) up(c_{n_1}) + ... + c_1 up(c_n) up(c_{n_1}) ... up(c_2) \end{aligned}$$

On en déduit alors que :

$$c_n = p \bmod up(c_n) \quad \text{et} \quad \lfloor \frac{p}{up(c_n)} \rfloor = c_{n-1} + c_{n-1} up(c_{n_1}) + ... + c_1 up(c_{n_1}) ... up(c_2)$$

Par récurrence, on a alors :

$$\forall k > 1, \quad c_k = \lfloor \frac{p}{\prod_{i>k} up(c_i)} \rfloor \bmod up(c_k) \quad \text{et} \quad c_1 = \lfloor \frac{p}{\prod_{i>1} up(c_i)} \rfloor$$

La fonction $\Upsilon : \mathcal{I} \mapsto \mathbb{N}$ est donc bien bijective, ce qui implique que $\mathcal{I}$ est un ensemble dénombrable totalement ordonné ayant une borne inférieure, donc il est noetherien. On en déduit alors de manière triviale que :

$$\text{Card}(\ \{\mathbf{c}' \mid \mathbf{c}' \ll \mathbf{c}\} \ ) = 1 + \Upsilon(\mathbf{c}) = 1 + \sum_{i=1}^{n} c_i \left(\prod_{k>i}^{n+1} up(c_k) \right)$$

□

Dans toute la suite nous notons $nb_{\ll}(\mathbf{c})$ ce nombre :

$$nb_{\ll}(\mathbf{c}) = 1 + \sum_{i=1}^{n} c_i \left(\prod_{k>i}^{n+1} up(c_k) \right) \tag{10.2}$$

<u>Nota-Bene</u> : Cette bijection est semblable à celle utilisée pour la linérisation des processeurs en chapitre (chap. 6, § 2.2, p. 114).

2.2 Propriétés mathématiques d'une fonction d'ordonnancement

Pour chaque nid de boucles, on choisit ici une forme affine entière $d(.)$ faisant correspondre au temps local, le temps événementiel :

$$d(\mathbf{c}) = \alpha.\mathbf{c} + \beta \tag{10.3}$$

où α est un vecteur ligne de même dimension que $\mathbf{c}$, l'opération . représente le produit scalaire usuel et β est un entier.

2.2.1 Les deux différentes hypothèses de croissance

Dans toute la suite, nous considèrons deux hypothèses distinctes sur la valeur de $\alpha = (\alpha_1, ..., \alpha_n)$. La première est le cas général mais de programmation difficile alors que la seconde est un cas particulier plus facile à implanter.

Hypothèse 2.2 (Hypothèse générale de croissance)

$$\begin{aligned} \alpha_i &> \sum_{j>i} \alpha_j \left(up(c_j) - 1 \right) \\ \alpha_n &\geq 1 \end{aligned} \tag{10.4}$$

Hypothèse 2.3 (Contrainte suffisante)

$$\begin{aligned} \alpha_i &\geq \alpha_{i+1}.up(c_{i+1}) \\ \alpha_n &\geq 1 \end{aligned} \tag{10.5}$$

On a vu que sous l'hypothèse 2.2, la fonction $d(.)$ est strictement croissante. Montrons qu'il en est de même avec l'hypothèse 2.3.

Proposition 2.4 *Sous l'hypothèse (2.3), la fonction $d(.)$ est strictement croissante.*

Démonstration –
On a vu (chap. 7, § 2.2, p. 132) qu'une condition nécessaire et suffisante pour qu'une fonction affine $d(.)$ vérifiant (10.3), soit strictement croissante pour l'ordre lexicographique était donné par l'hypothèse (2.2, p.2.2). Montrons donc que $\forall i < n$ on a :

$$\alpha_{i+1}.up(c_{i+1}) > \sum_{j>i} \alpha_j \,(up(c_j) - 1)$$

Ceci est donc équivalent à montrer que l'hypothèse (2.3) est une condition suffisante à l'hypothèse (2.2). L'hypothèse (2.3) est équivalente à

$$\forall i < n, \quad \left\{ \begin{array}{rcl} \alpha_n & \geq & 1 \\ \alpha_{n-1} & \geq & \alpha_n.up(c_n) \\ \alpha_{n-2} & \geq & \alpha_{n-1}.up(c_{n-1}) \\ & \ldots & \\ \alpha_i & \geq & \alpha_{i+1}.up(c_{i+1}) \end{array} \right.$$

Ce qui implique que

$$\begin{aligned} \alpha_i + \sum_{j>i} \alpha_j &\geq 1 + \sum_{j>i} \alpha_j.up(c_j) \\ \Longleftrightarrow \alpha_i &\geq 1 + \sum_{j>i} \alpha_j \,(up(c_j) - 1) \\ \Longleftrightarrow \alpha_i &> \sum_{j>i} \alpha_j \,(up(c_j) - 1) \;\Leftrightarrow\; (10.4) \end{aligned}$$

□

2.2.2 Étude de la fonction trace événementielle associée à un nid de boucles

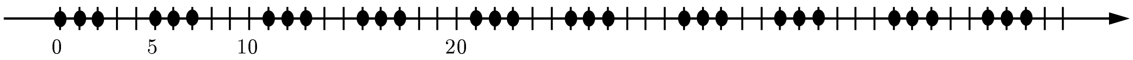
Soit $d \,:\, \mathcal{I} \mapsto \mathbb{N}$ la fonction d'ordonnancement associée à un nid de boucles, et définie par

$$\forall \mathbf{c} \in \mathcal{I}, \quad d(\mathbf{c}) = \alpha.\mathbf{c} + \beta$$

Pour simplifier les calculs et les démonstrations, on peut, dans ce paragraphe, supposer sans perdre aucune généralité que $\beta = 0$. D'où : $\forall \mathbf{c} \in \mathcal{I}, \quad d(\mathbf{c}) = \alpha_1.c_1 + \alpha_2.c_2 + ... + \alpha_n.c_n$. Soit Θ_d la fonction définie par

$$\Theta_d \;:\; \left\{ \begin{array}{lcl} \mathbb{N} & \mapsto & \{\bullet\} \\ x & \rightarrow & \bullet \text{ si } \exists \mathbf{c} \in \mathcal{I} \text{ t.q. } x = d(\mathbf{c}) \end{array} \right.$$

Cette fonction Θ_d sera appelée trace événementielle associée à la fonction d'ordonnancement d (voir un exemple de trace événementielle dans la figure (10.1).



- $\mathcal{I} = \mathbb{N} \times \{0, 1\} \times \{0, 1, 2\}$,
- $\alpha = (11, 5, 1)$

Alors $d(c) = 11c_1 + 5c_2 + c_3$.

FIG. 10.1. – *Un exemple de fonction trace associée à une fonction d'ordonnancement sur un espace d'itérations de dimension 3.*

Proposition 2.5 *Soit Θ_d la fonction trace événementielle associée à une fonction d'ordonnancement $d\ :\ \mathbf{c} \rightarrow \sum_{i=1}^{n} \alpha_i.c_i$. Alors α_1 est une période de Θ_d.*

<u>Démonstration</u> – Montrons que $\forall x \in \mathbb{N}$, on a $\Theta_d(x + \alpha_1) = \Theta_d(x)$.

- Supposons que $\Theta_d(x) = \bullet$, alors il existe $\mathbf{c} \in \mathcal{I}$ tel que $x = \sum_{i=1}^{n} \alpha_i.c_i$ et $\mathbf{c} = (c_1, c_2, ..., c_n)$, ce qui implique :

$$\exists \mathbf{c} = (c_1, c_2, ..., c_n) \quad \text{t.q. } x + \alpha_1 = \alpha_1.(c_1 + 1) + \alpha_2.c_2 + ... + \alpha_n.c_n$$

$$\Longleftrightarrow \exists \mathbf{c}' = (c_1 + 1, c_2, ..., c_n) \quad \text{t.q. } x + \alpha_1 = d(\mathbf{c}') \Longleftrightarrow \Theta_d(x + \alpha_1) = \bullet$$

  On a donc démontré que $\Theta_d(x) = \bullet \Rightarrow \Theta_d(x + \alpha_1) = \bullet$.

- Supposons à présent que $\Theta_d(x) \neq \bullet$ et montrons que l'on a alors $\Theta_d(x + \alpha_1) \neq \bullet$. Par contraposition, on a :

$$\Theta_d(x + \alpha_1) = \bullet \Leftrightarrow \exists \mathbf{c} \ \text{ t.q. } x + \alpha_1 = \alpha_1.c_1 + \alpha_2.c_2 + ... + \alpha_n.c_n$$

$$\Leftrightarrow x = \alpha_1.(c_1 - 1) + \alpha_2.c_2 + ... + \alpha_n.c_n$$

  D'on il existe $\mathbf{c}' = (c_1 - 1, c_2, ..., c_n)$ tel que $x = \alpha_1.(c_1 - 1) + \alpha_2.c_2 + ... + \alpha_n.c_n$, car c_1 est nécessairement supérieur ou égal à 1. En effet, sous l'hypothèse 2.2 (et donc *a forciori* sous l'hypothèse 2.3), on a $\alpha_1 > \sum_{j>1} \alpha_j\,(up(c_j) - 1)$, ce qui implique que l'on ait $x + \alpha_1 > \alpha_2.c_2 + ... + \alpha_n.c_n$, d'où $c_1 \geq 1$. Et donc $\Theta_d(x) = \bullet$.

On a donc $\forall x \in \mathbb{N}$, on a $\Theta_d(x + \alpha_1) = \Theta_d(x)$, d'où α_1 est une période de Θ_d. □

3 Approximation du nombre d'événements

Une fois l'ensemble des caractéristiques des fonctions d'ordonnancement enoncé, nous abordons le problème crucial de savoir dénombrer l'ensemble des évenements (en fait, l'ensemble des partitions exécutées) dans un intervalle de temps donné. Nous construisons ici, à cet effet, une **contrainte de cardinalité** surapproximant ce nombre dévénements. Outre l'application de cette contrainte à la latence, nous l'exploitons aussi pour dénombrer les événements au sein du modèle capacitif de mémoire (cf. chapitre 8, § 3.3, p. 148), ainsi que pour les contraintes d'entrées-sorties (cf. § 6, p. 172).

3.1 Nombre d'événements pour un nid de boucles

Définition 3.1 *Le* nombre d'événements *produits dans un nid de boucles (i.e. par une tâche) entre les dates θ et $\theta + \ell$ est donné par :*

$$nbc(\theta, \ell) = \text{Card}\left(\{\mathbf{c} \mid \theta \leq d(\mathbf{c}) \leq \theta + \ell\}\right) \tag{10.6}$$

En fait, on a de manière évidente : $nbc(\theta, \ell) = \text{Card}\left(\{x \mid \Theta_d(x) = \bullet \text{ et } \theta \leq x \leq \theta + \ell\}\right)$.

Lemme 3.2 $\forall\theta,\ \forall\ell, \quad nbc(\theta, \ell) = nbc(\theta \bmod \alpha_1, \ell)$.

Démonstration – On a $nbc(\theta, \ell) = \text{Card}\left(\{x \mid \Theta_d(x) = \bullet \text{ et } \theta \leq x \leq \theta + \ell\}\right)$. Comme la fonction Θ_d est périodique de période α_1 (2.5), on en déduit que :

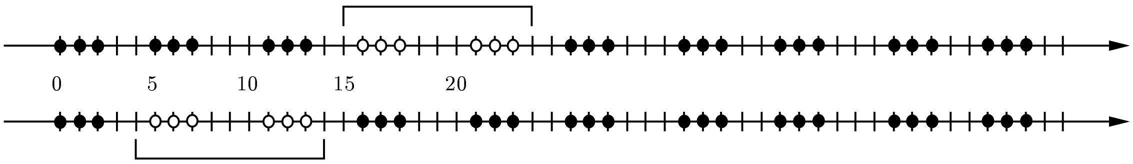
$$nbc(\theta, \ell) = \text{Card}\left(\{x \mid \Theta_d(x) = \bullet \text{ et } \gamma \leq x \leq \gamma + \ell \text{ avec } \gamma = \theta \bmod \alpha_1\}\right)$$

□

Exemple 3.3 *Reprenons l'exemple donné dans la figure (10.1).*

- $\mathcal{I} = \mathbb{N} \times \{0, 1\} \times \{0, 1, 2\}$,
- $\alpha = (11, 5, 1)$.

On a alors $nbc(15, 10) = nbc(4, 10)$ *(car* $4 = 15 \bmod 11$*).*



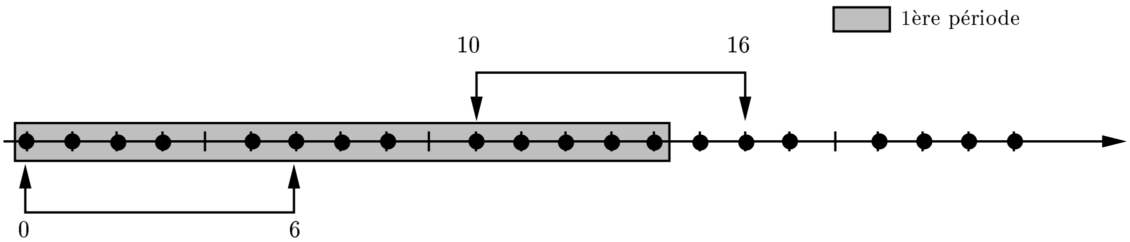
Théorème 3.4 *Soit* $d : \mathcal{I} \mapsto \mathbb{N}$ *une fonction d'ordonnancement associée à un nid de boucles de dimension* n *(i.e.* $\mathcal{I} = \mathbb{N} \times \{0, ..., up(c_2) - 1\} \times ... \times \{0, ..., up(c_n) - 1\}$ *et* $d(\mathbf{c}) = \sum_{i=1}^{n} \alpha_i.c_i$*). Supposons que* d *satisfasse l'hypothèse 2.3, i.e. :* $\alpha_i \geq \alpha_{i+1}.up(c_{i+1})$ *et* $\alpha_n \geq 1$*. Alors :*

$$\begin{aligned} &\forall\gamma \in \mathbb{N},\ \forall\ell > 0, \quad nbc(\gamma, \ell) \leq nbc(0, \ell) \\ \Leftrightarrow\ &\text{Card}\left(\{\mathbf{c} \mid \gamma \leq d(\mathbf{c}) \leq \gamma + \ell\}\right) \ \leq\ \text{Card}\left(\{\mathbf{c} \mid 0 \leq d(\mathbf{c}) \leq \ell\}\right) \end{aligned} \tag{10.7}$$

Remarque 3.5 *Ce théorème peut-être faux si la fonction d'ordonnancement* $d(.)$ *ne satisfait que l'hypothèse (2.2), i.e.* $\alpha_i > \sum_{j>i} \alpha_j \left(up(c_j) - 1\right)$ *et* $\alpha_n \geq 1$.

Contre-exemple : Supposons que $\mathcal{I} = \mathbb{N} \times \{0, 1, 2\} \times \{0, 1, 2, 3\}$ et que $d(\mathbf{c}) = 14c_1 + 5c_2 + c_3$. Alors d ne satisfait pas l'hypothèse (2.3), car α_1 ne vérifie pas $\alpha_1 \geq 5 \times 3$. On montre alors $nbc(0, 6) = 6$ et $nbc(10, 6) = 7$. En effet :

$$nbc(0, 6) = \text{Card}\{\mathbf{c} | 0 \leq d(\mathbf{c}) \leq 6\} = \text{Card}\{(0,0,0); (0,0,1); (0,0,2); (0,0,3); (0,1,0); (0,1,1)\}$$
$$nbc(10, 6) = \text{Card}\{\mathbf{c} | 10 \leq d(\mathbf{c}) \leq 16\} = \text{Card}\{(0,2,0); (0,2,1); (0,2,2); (0,2,3); (1,0,0); (1,0,1); (1,0,2)\}$$



Afin de rendre plus lisible la démonstration du théorème, nous allons dans un premier temps définir quelques notions utilisées dans la suite.

Hypothèses : Soit $d : \mathcal{I} \mapsto \mathbb{N}$ une fonction d'ordonnancement associée à un nid de boucles de dimension n (*i.e.* $\mathcal{I} = \mathbb{N} \times \{0, ..., up(c_2) - 1\} \times ... \times \{0, ..., up(c_n) - 1\}$) et $d(\mathbf{c}) = \sum_{i=1}^{n} \alpha_i . c_i$. La fonction d satisfait l'hypothèse (2.3), *i.e.* : $\alpha_i \geq \alpha_{i+1}.up(c_{i+1})$ et $\alpha_n \geq 1$.

On s'intéresse alors aux propriétés géométriques du graphe de la fonction trace événementielle Θ_d associée à d. On appelle :

Motif de degré 0 : un point $(x, \bullet)$ tel que $(x, \bullet) \in \mathrm{Graph}(\Theta_d)$, *i.e.* $\exists \mathbf{c}$ tel que $d(\mathbf{c}) = x$,

Trou de degré 0 : un point $(x, \emptyset)$ tel que il n'existe pas de $\mathbf{c}$ avec $d(\mathbf{c}) = x$,

Motif de degré 1 : un ensemble de points suivant :

$$\{(x, \bullet) \mid \exists \mathbf{c} \text{ t.q. } d(\mathbf{c}) = x \text{ où } \mathbf{c} \in \{c_1\} \times \{c_2\} \times ... \{c_{n_1}\} \times \{0, ..., up(c_n) - 1\}\}$$

Trou de degré 1 : un point $(x, \emptyset)$ qui sépare deux motifs de degré 1,

...

Motif de degré k : l'ensemble de points suivant :

$$\left\{ \begin{array}{ll} (x, \bullet) \mid & \exists \mathbf{c} \text{ t.q. } d(\mathbf{c}) = x \text{ où} \\ & \mathbf{c} \in \{c_1\} \times ... \times \{c_{n_k}\} \times \times \{0, ..., up(c_{k_n+1}) - 1\} \times ... \times \{0, ..., up(c_n) - 1\} \end{array} \right\}$$

Trou de degré k : un point $(x, \emptyset)$ qui sépare deux motifs de degré k.

La figure (10.2) illustre toutes ces définitions pour l'exemple suivant : $\mathcal{I} = \mathbb{N} \times \{0, 1\} \times \{0, 1, 2\} \times \{0, 1, 2\}$ et $d(\mathbf{c}) = 26c_1 + 13c_2 + 4c_3 + c_4$.

Démonstration – du théorème (2.5, p. 162).

Motif de degré 1 :

Longueur d'un motif de degré 1 : La longueur d'un motif de degré 1 est égale à :

$$\begin{array}{rcl} \nu(1) & = & 1 + d((c_1, ..., c_{n-1}, up(c_n) - 1)) - d((c_1, ..., c_{n-1}, 0) \\ & = & 1 + \alpha_n(up(c_n) - 1) \end{array}$$

Nombre de motifs de degré 0 : le nombre total de motifs de degré 0 dans un motif de degré 1 (*i.e.* nombre de points constituant un motif élémentaire) est égal à $up(c_n)$,

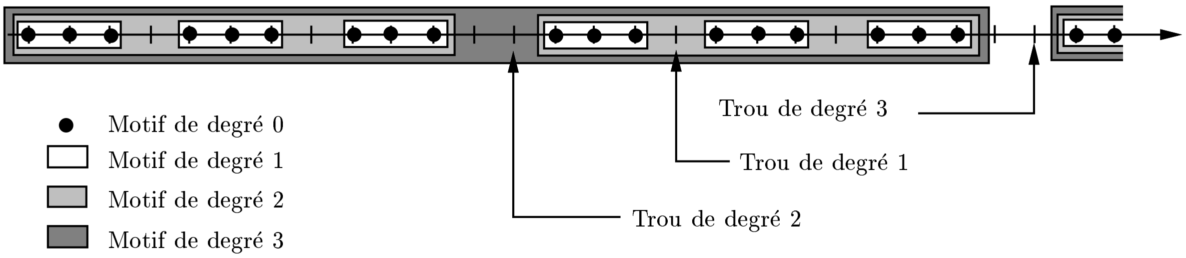


FIG. 10.2. – *Illustration des différents degrés de motifs pour le graphe de la trace événementielle associée à* $d(\mathbf{c}) = 26c_1 + 13c_2 + 4c_3 + c_4$ *et* $up(\mathbf{c}) = (\infty, 2, 3, 3)$

Nombre de trous de degré 0 : le nombre total de trous dans un motif élémentaire est alors égal à :

$$\begin{aligned}\tau(0) &= 1 + \alpha_n(up(c_n) - 1) - up(c_n)\\ &= (\alpha_n - 1)(up(c_n) - 1)\end{aligned}$$

On remarque donc que sous l'hypothèse (2.2) comme sous l'hypothèse (2.3), si $\alpha_n = 1$ alors il n'y a pas de trous (*i.e.* de non événements) dans un motif de degré 1.

Motif de degré 2 :

Longueur d'un motif de degré 2 : La longueur d'un motif de degré 2 est égale à :

$$\begin{aligned}\nu(2) &= 1 + d((c_1, ..., c_{n-2}, up(c_{n-1} - 1), up(c_n) - 1)) - d((c_1, ..., c_{n-2}, 0, 0)\\ &= 1 + \alpha_{n-1}(up(c_{n-1}) - 1) + \alpha_n(up(c_n) - 1)\end{aligned}$$

Nombre de motifs de degré 1 : le nombre total de motifs de degré 1 dans un motif de degré 2 est égal à $up(c_{n-1})$,

Nombre de trous de degré 1 : le nombre total de trous de degré 1 dans un motif de degré 2 (*i.e.* le nombre total de trous séparant deux motifs de degré 1 dans un motif de degré 2) est alors égal à :

$$\begin{aligned}\tau(1) &= 1 + \alpha_{n-1}(up(c_{n-1}) - 1) + \alpha_n(up(c_n) - 1) - up(c_{n-1})\,(1 + \alpha_n(up(c_n) - 1))\\ &= (up(c_{n-1}) - 1)(\alpha_{n-1} - 1 - \alpha_n(up(c_n) - 1))\end{aligned}$$

On en déduit alors que le nombre de trous de degré 1 séparant deux motifs consécutifs de degré 1 $\tau^\sharp(1)$ est $\tau^\sharp(1) = (\alpha_{n-1} - 1 - \alpha_n(up(c_n) - 1))$.

- Sous l'hypothèse (2.2), on sait que $\alpha_{n-1} > \alpha_n(up(c_n) - 1)$, on en déduit donc seulement que le nombre de trous de degré 1 séparant deux motifs consécutifs de degré 1 est positif ou nul, donc en particulier, on peut (dans certain cas) le choisir inférieur au nombre de trous de degré 0 séparant deux motifs consécutifs de degré 1,*i.e.* $\tau^\sharp(1) \leq \tau^\sharp(0)$ ce qui rend l'assertion du théorème faux (voir le contre-exemple de la page 163).
- Sous l'hypothèse (2.3), on sait que $\alpha_{n-1} \geq \alpha_n up(c_n)$, on en déduit donc que le nombre de trous de degré 1 séparant deux motifs consécutifs de degré 1 est supérieur ou égal à $\alpha_n - 1$. En particulier $\tau^\sharp(1)$ est supérieur ou égal au nombre de trous de degré 0 séparant deux motifs consécutifs de degré 1, *i.e.* $\tau^\sharp(1) \geq \tau^\sharp(0)$.

On montre alors que $\forall k < n$:

Motif de degré k+1 :

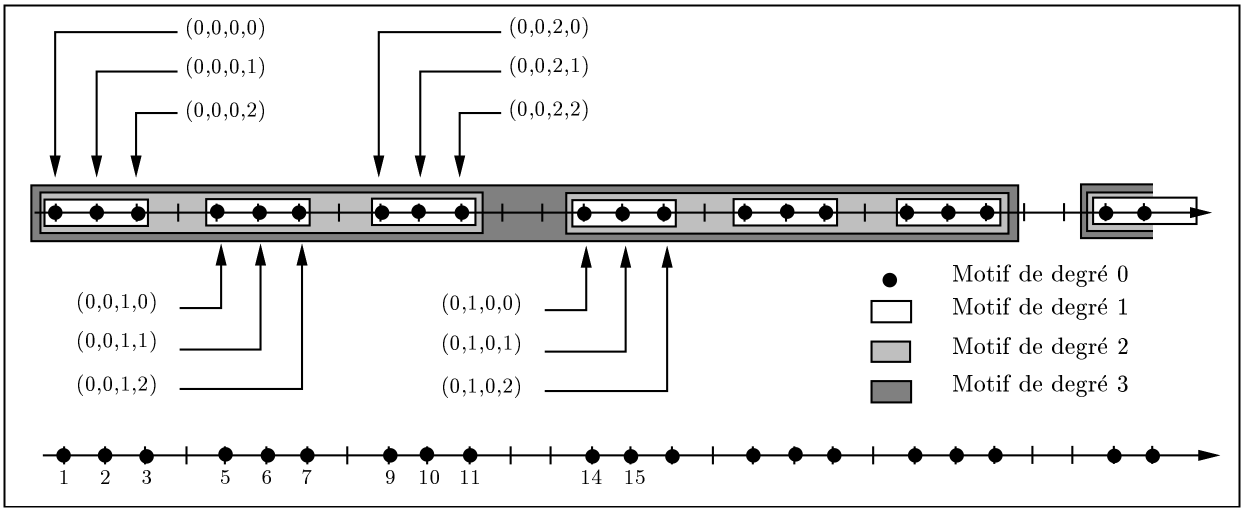


FIG. 10.3. – *Valeur de la distance événementielle associée aux différents événenements pour* $d(\mathbf{c}) = 26c_1 + 13c_2 + 4c_3 + c_4$ *et* $up(\mathbf{c}) = (\infty, 2, 3, 3)$

Longueur d'un motif de degré k+1 : La longueur d'un motif de degré $k+1$ est égale à :

$$\begin{aligned}\nu(k) &= 1 + d((c_1, ..., c_{n-k-1}, up(c_{n-k}-1), ..., up(c_n)-1)) - d((c_1, ..., c_{n-k-1}, 0, .., 0)) \\ &= 1 + \sum_{j=n-k}^{n} \alpha_j (up(c_j) - 1)\end{aligned}$$

Nombre de motifs de degré k : le nombre total de motifs de degré k dans un motif de degré $k+1$ est égal à $up(c_{n-k})$,

Nombre de trous de degré k : le nombre total de trous de degré k dans un motif de degré $k+1$ est alors égal à :

$$\begin{aligned}\tau(k) &= 1 + \sum_{j=n-k}^{n} \alpha_j (up(c_j) - 1) - up(c_{n-k}) \left(\sum_{j=n-k+1}^{n} \alpha_j (up(c_j) - 1) \right) \\ &= (up(c_{n-k}) - 1) \left(\alpha_{n-k} - 1 - \sum_{j=n-k+1}^{n} \alpha_j (up(c_j) - 1) \right)\end{aligned}$$

On en déduit alors que le nombre de trous de degré k séparant deux motifs consécutifs de degré k, $\tau^\sharp(k)$ est

$$\tau^\sharp(k) = \alpha_{n-k} - 1 - \sum_{j=n-k+1}^{n} \alpha_j (up(c_j) - 1)$$

Sous l'hypothèse (2.3), on sait que $\alpha_{n-k} \geq \alpha_{n-k+1} up(c_{n-k+1})$, on en déduit donc que le nombre de trous de degré k séparant deux motifs consécutifs de degré k est supérieur ou égal à $\alpha_{n-k+1} - 1$. En particulier $\tau^\sharp(k)$ est supérieur ou égal au nombre de trous de degré $k-1$ séparant deux motifs consécutifs de degré $k-1$, *i.e.* $\tau^\sharp(k) \geq \tau^\sharp(k-1)$.

On en déduit donc que pour tout $k < n$, le nombre de motifs de degré k séparant deux motifs consécutifs de degré k est supérieur ou égal au nombre de trous de degré $k-1$ séparant deux motifs consécutifs de degré $k-1$.

Ceci implique que le nombre de points présents dans un intervalle de longueur ℓ sera inférieur ou égal au nombre de points présents dans l'intervalle initial (*i.e.* d'origine 0) de même longueur, *i.e.* $\forall \gamma,\ \ nbc(\gamma, \ell) \leq nbc(0, \ell)$. En fait, d'après ce que l'on vient de montrer on ajoute (ou non) des trous à chaque degré lorsque l'on passe de l'intervalle initial à un autre intervalle de même longueur. □

En utilisant à présent le fait que la fonction d'ordonnancement est croissante par rapport à l'ordre lexicographique, et en notant $\mathbf{c}^{(\ell)}$ le vecteur tel que $\mathbf{c}^{(\ell)} = \max_{\ll}\{\mathbf{c} \mid d(\mathbf{c}) \leq \ell\}$, on en déduit que :

$$nbc(0,\ell) \ = \ nb_{\ll}\left(\max_{\ll}\{\mathbf{c} \mid d(\mathbf{c}) \leq \ell\}\right) = nb_{\ll}\left(\mathbf{c}^{(\ell)}\right) \tag{10.8}$$

4 Temps physiques d'exécution des partitions élémentaires

Le temps physique lié aux partitions est la deuxième brique permettant de définir les contraintes temps-réel. Il faut tenir compte du nombre de cycles impliqués par l'exécution d'une partition, mais aussi du type de communication et de recouvrement qu'elle engendre. Ceci afin de donner la contrainte globale du temps d'exécution communication/calcul associée à une partition, et ce pour chacune des tâches.

4.1 Volume de calculs des partitions

Le partitionnement permet d'établir les différents volumes de calcul d'une partition á partir du vecteur exprimant la localité $\mathbf{l}$. Si l'on considère d'autres partitionnements possibles comme $(\mathbf{c}, \mathbf{l}, \mathbf{p})$, les volumes de calculs ne sont plus obtenus de la même façon. De plus, cette remarque est aussi vraie pour le calcul du nombre de processeurs (cf. chapitre 9, § 2.1, p. 151).

4.1.1 Rappel des hypothèses du partitionnement

Rappelons les hypothèses mathématiques du partionnement (chap. 4,§ 4.1, p. 90):

$$\mathbf{i} = LP.\mathbf{x} + L.\mathbf{y} + \mathbf{z} \quad \text{avec} \left\{ \begin{array}{l} \forall\ \mathbf{z},\ \mathbf{0} \leq L^{-1}.\mathbf{z} < \mathbf{1}, \\ \forall\ \mathbf{y},\ \mathbf{0} \leq P^{-1}.\mathbf{y} < \mathbf{1}, \\ \forall\ i \in \mathcal{I},\ \mathbf{0} \leq \mathbf{i} < up(\mathbf{i}) \\ \det(L) \neq 0 \text{ et diagonale} \\ \det(P) \neq 0 \text{ et diagonale} \end{array} \right. \quad \text{où} \quad (\mathbf{x}, \mathbf{y}, \mathbf{z}) \in \{(\mathbf{c}, \mathbf{l}, \mathbf{p}), (\mathbf{c}, \mathbf{p}, \mathbf{l})\} \tag{10.9}$$

Sous ces hypothèses, nous avons éliminé les calculs redondants.

4.1.2 Volume de calculs d'une partition effectuée en un événement donné et sur un processeur

Définissons tout d'abord pour un temps $\mathbf{c}$ donné, et un processeur $\mathbf{p}$ fixé, le volume de calculs effectués par une tâche.

Modèle de partitionnement $(\mathbf{c}, \mathbf{p}, \mathbf{l})$: Pour un couple $(\mathbf{c},\ \mathbf{p})$ donné, la partition associée à une tâche s'écrit :

$$\mathcal{P}_{(\mathbf{c},\mathbf{p})} = \{\mathbf{i} \mid \exists\ \mathbf{l} < L.\mathbf{1},\ \ \mathbf{i} = L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}\}$$

On en déduit que le volume de calculs sur $\mathcal{P}_{(\mathbf{c},\mathbf{p})}$ est égal à

$$\mathcal{V}_{calcul}(\mathbf{c},\mathbf{p}) = \text{Card}\{\mathbf{l} \mid \mathbf{0} \leq \mathbf{l} < L.\mathbf{1}\}$$

En utilisant le calcul d'un volume d'un parallélépipède (entier) défini par une application linéaire, on montre [Ber90, Str86] alors que : $\mathcal{V}_{calcul}(\mathbf{c},\mathbf{p}) = \det(L)$.

Modèle de partitionnement $(\mathbf{c},\mathbf{l},\mathbf{p})$: Pour un couple $(\mathbf{c},\mathbf{p})$ donné, la partition associée à une tâche s'écrit :

$$\mathcal{P}_{(\mathbf{c},\mathbf{p})} = \{\mathbf{i} \mid \exists\, \mathbf{l} < P.\mathbf{1},\quad \mathbf{i} = L.P.\mathbf{c} + L.\mathbf{l} + \mathbf{p}\}$$

On en déduit donc que le volume de calculs sur $\mathcal{P}_{(\mathbf{c},\mathbf{p})}$ est égal à

$$\mathcal{V}_{calcul}(\mathbf{c},\mathbf{p}) = \text{Card}\{\mathbf{l} \mid \mathbf{0} \leq \mathbf{l} < P.\mathbf{1}\}$$

On montre de même que : $\mathcal{V}_{calcul}(\mathbf{c},\mathbf{p}) = \det(P)$.

4.2 Recouvrement des calculs et des communications

Le recouvrement des communications par les calculs est une notion très importante en parallélisme. Ainsi, le temps perdu en communications est amorti, en fait recouvert, par les temps de calculs et il y a plusieurs techniques pour mettre en oeuvre celà [Dio96]. Que ce soit en mode réception ou en mode émission les formules de recouvrement calculs/communications dépendent uniquement du mode pipeliné ou bloqué. Nous utilisons ici les notations V_{te}^t et V^t associées aux volumes de communications qui ont été données au chapitre 6, § 4.6, p. 213, § 4.6, p. 216. Le paramètre V_{te}^t est le volume de communication établi selon la technique exposée en (cf. chapitre 6, § 4.6, p. 211), mais restreint au niveau de la macro-instruction (celle qui écrit pour le mode émission, celle qui lit pour le mode réception), puisqu'il s'agit du volume émis (resp, reçu par une partition).

Mode pipeliné : Dans ce mode, le recouvrement se fait au niveau du traitement élémentaire compris dans une partition. Il faut d'abord exécuter la première macro-instruction (resp. communication) en ammorçage de pipeline pour le mode émission (resp. réception). Comme l'approximation du volume de communication est constante, en régime stationnaire, il faut exécuter la $k^{\text{ème}}$, $k < det(L)$ macro-instruction pendant que l'on communique les résultats issues (resp. destinées à) de la $k-1^{\text{ème}}$ (resp. $k+1^{\text{ème}}$). En fin de pipeline, il faut exécuter la dernière communication (resp. le dernier calcul). D'où la contrainte de recouvrement communications/calculs (cf. chapitres 6 et 7). On peut tenir compte du modèle de partitionnement :

Modèle $(\mathbf{c},\mathbf{p},\mathbf{l})$

$$\forall t,\ ct(t) \ = \ cout_{elem}^t + (\det(L)-1).max(cout(V_{te}^t), cout_{elem}^t) + cout(V_{te}^t) \qquad (10.10)$$

Modèle $(\mathbf{c},\mathbf{l},\mathbf{p})$

$$\forall t,\ ct(t) \ = \ cout_{elem}^t + (\det(P)-1).max(cout(V_{te}^t), cout_{elem}^t) + cout(V_{te}^t) \qquad (10.11)$$

où le paramètre $cout_{elem}^t$ est constant. Il s'agit du nombre de cycles requis pour l'exécution d'une macro-instruction. Ce paramètre est une donnée d'entrée (chap. 2, § 3, p. 46). Le terme $ct(t)$ représente la durée totale des calculs/communications pour un bloc indexé par $\mathbf{c}^t$ pour la tache t.

Mode bloqué : Dans ce mode, le recouvrement se fait au niveau des événements de l'ordonnancement. A la date $d(\mathbf{c})$, alors que l'on exécute la partition associée à $\mathbf{c}$, on communique en émission (resp. réception) les données calculées à la date $d(\mathbf{c}'), \mathbf{c}' << \mathbf{c}$ (resp. $\mathbf{c}' >> \mathbf{c}$), où $\mathbf{c}'$ est le prédécesseur (resp. successeur) lexico-graphique de $\mathbf{c}$. Dans tous les cas, il faut un événement supplémentaire pour exécuter la dernière communication (resp. dernier calcul).

Modèle $(\mathbf{c}, \mathbf{p}, \mathbf{l})$

$$\forall t,\ ct(t) = max(cout(V^t), \det(L).cout^t_{elem}) \tag{10.12}$$

Modèle $(\mathbf{c}, \mathbf{l}, \mathbf{p})$

$$\forall t,\ ct(t) = max(cout(V^w), \det(P).cout^t_{elem}) \tag{10.13}$$

5 Contraintes de latence

La latence d'une application de traitement de signal est le maximum de l'âge réel de tout signal de sortie, *i.e.* c'est le *maximum de la différence des dates de sortie d'un signal avec les dates des signaux d'entrée qui ont contribué à son élaboration.* La contrainte de latence d'une application exprime le fait qu'elle doit être inférieure ou égale à la latence opérationnelle donnée par l'utilisateur. Pour la satisfaire, nous devons donc évaluer la latence d'une application. Celle-ci utilise le calcul du plus long chemin sur une composante fortement connexe maximale du graphe de dépendances.

5.1 Distance maximale de dépendances pour un graphe acyclique

Cette section permet d'établir les distances nécessaires pour connaître l'intervalle événementiel requis pour l'exécution de la totalité de l'application. On obtient alors l'expression d'un plus long chemin sur un graphe de dépendances, donnée dans un deuxième temps. Enfin, basée sur les distances et la contrainte de cardinalité portant sur les événements, nous donnons la contrainte globale de latence.

5.2 Expressions des plus long chemins

Nous définissons ici l'expression formelle du plus long chemin sur le graphe de dépendance, et expliquons comment cette expression peut être implantée grâce à la notion de distance maximale entre couple de tâches. Ceci permet de réduire la complexité de l'expression.

5.2.1 Cas général du graphe acyclique

Dans ce paragraphe, on considère que le programme est modélisé par un graphe de traitements T^1,..., T^p tels que le graphe de dépendances associé soit acyclique (cf. chapitre 5). Il faut définir le nombre de tops maximal Λ entre une entrée et une sortie, pour lesquels il existe un chemin dans le graphe de dépendances. Celà correspond à la distance du plus long chemin (en nombre de tops événementiel) sur les composantes fortement connexes du graphe de dépendances (fig. 10.4-a, p. 170). On peut alors le calculer par :

$$\bigwedge D(\mathbf{i}^w, \mathbf{i}^r) \Longrightarrow \Lambda = \max\left(\sum_w \left(\max_r \left(\ d^r(\mathbf{c}^r(\mathbf{i}^r)) - d^w(\mathbf{c}^w(\mathbf{i}^w))\ \right)\right)\right) \tag{10.14}$$

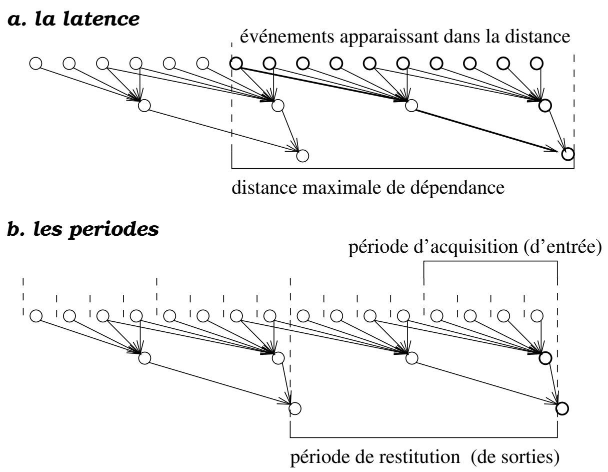


FIG. 10.4. – *Latence et périodes dans le graphe dépendances*

Démonstration – Cette contrainte est basée sur la distance maximale de dépendance entre deux nids de boucles w, r:

$$\max_{(w,r)} (\ d^r(\mathbf{c}^r(\mathbf{i}^r)) - d^w(\mathbf{c}^w(\mathbf{i}^w))\ )$$

Considérant l'ensemble du chemin de dépendances, on a donc :

$$\begin{aligned} \bigwedge D(\mathbf{i}^w, \mathbf{i}^r) &\implies \Lambda = \sum_w \left(\max_r (\max (\ d^r(\mathbf{c}^r(\mathbf{i}^r)) - d^w(\mathbf{c}^w(\mathbf{i}^w))\ )) \right) \\ &\implies \Lambda = \max \left(\sum_w \left(\max_r (\ d^r(\mathbf{c}^r(\mathbf{i}^r)) - d^w(\mathbf{c}^w(\mathbf{i}^w))\ ) \right) \right) \end{aligned}$$

□

5.2.2 Utilisation des sommets du polytope

Le problème de cette expression réside dans la contrainte

$$\max_r (\ d^r(\mathbf{c}^r(\mathbf{i}^r)) - d^w(\mathbf{c}^w(\mathbf{i}^w))\ )$$

qui correspond à l'intervalle de temps maximum séparant deux partitions dépendantes, concernant un couple de tâches, ce maximum peut porter sur un nombre important de couples. Nous utilisons ici, la contrainte de distance portant sur les sommets du polytope (cf. chapitre 7, § 5.1.2, p. 141) afin de pouvoir dériver une contrainte réaliste et implantable. La contrainte (10.14) devient donc :

$$\bigwedge D(\mathbf{i}^w, \mathbf{i}^r) \Longrightarrow \Lambda = \sum_w \left(\ \max_r (d^{w,r}_{max})\ \right) \tag{10.15}$$

5.2.3 Borne de la distance maximale et dimension infinie

Dans le cas où chaque nid de boucles présente une dimension infinie, nous donnons une condition suffisante que le nombre Λ soit borné. Il est cependant toujours fini si on se place dans le cas d'un domaine fini d'itérations.

Proposition 5.1 *Soit une application composée de* p *tâches notées* $T^1, ..., T^p$ *dont les domaines d'itérations sont respectivement* $\mathcal{I}(T^1)$, ... , $\mathcal{I}(T^p)$*, avec :* $\forall j = 1,..,p,\quad \mathcal{I}(T^j) = \mathbb{N} \times [0, up(c_1^j)[... \times [0, up(c_{n_j}^j)[$*. Alors, pour chaque couple de tâches dépendantes* (T^r, T^w)*, on a :*

$$\alpha^r.I^w.L^w.P^w = \alpha^w.I^r.L^r.P^r$$

<u>Démonstration</u> –

Si le prédicat $\mathcal{D}(\mathbf{c}^w, \mathbf{c}^r)$ est vrai alors on doit avoir l'expression $d^w(\mathbf{c}^w) - d^r(\mathbf{c}^r)$ bornée. Ceci implique que l'expression : $\boldsymbol{\alpha}^w.\mathbf{c}^w - \boldsymbol{\alpha}^r.\mathbf{c}^r$ est bornée[1]. Comme une seule dimension, que ce soit pour T et S est infinie c'est la différence des termes comportant les itérateurs infinis qu'il faut borner. Or, si l'on considère la composante d'itérations infinie :

$$\forall\ c_\infty,\ c_\infty = \frac{i_\infty}{L_q.P_q}$$

Le problème revient donc à borner l'expression :

$$\boldsymbol{\alpha}^w.\frac{i_\infty^w}{I_p^w.L_q^w.P_q^w} - \boldsymbol{\alpha}^r.\frac{i_\infty^r}{I_p^r.L_q^r.P_q^r}\ ssi\ \mathcal{D}(i_\infty^w, i_\infty^r)$$

or

$$\mathcal{D}(i_\infty^w, i_\infty^r)\ \Leftrightarrow\ \mathcal{R}^w(i^w) \cap \mathcal{R}^r(i^r)$$

En considérant les restrictions applicatives et en simplifiant cette expression des intersections entre ensembles d'éléments de tableaux, on calcule formellement un majorant :

$$\mathcal{D}(i_\infty^w, i_\infty^r)\ \Rightarrow\ I_p^w.\mathbf{i}_\infty^w\ -\ I_q^r.\mathbf{i}_\infty^r\ \leq\ M^r.(up(m^r) - 1)$$

exprimant ainsi une condition suffisante pour que le couple (i_∞^w, i_∞^r) soit dépendant. Ceci permet de borner notre expression grâce à la contrainte :

$$\frac{\boldsymbol{\alpha}_\infty^r}{I_p^w.L_q^w.P_q^w}\ =\ \frac{\boldsymbol{\alpha}_\infty^w}{I_p^r.L_q^r.P_q^r}$$

□

Cette contrainte, exprimant une condition suffisante, est très importante. Elle permet de sur-contraindre les vecteurs d'ordonnancement $\boldsymbol{\alpha}$ et permet d'instancier rapidement ces vecteurs le long du DFG.

[1]On a la même propriété en ce qui concerne la mémoire, la même contrainte garantie une mémoire bornée

5.3 Contrainte de latence

On peut calculer le nombre maximal de tops séparant une entrée d'une sortie dépendante de cette entrée, puis calculer pour chaque tâche le nombre maximum de calculs et de communications pouvant avoir lieu dans cet intervalle. Il suffit ensuite de faire la somme pour toutes les tâches de ce nombre multiplié par une estimation du temps réel de calculs/communications pris par cette même tâche. Le modèle de calcul de la latence est basé sur le calcul de la distance maximale de dépendance entre la première et la dernière tâche (§ 5.1, p. 169), et le calcul du nombre de tâche dans cet intervalle de temps. Étant donné une tâche s, le nombre d'événements ayant lieu dans un intervalle de longueur Λ est donné par $\max_{x<\alpha_1^s} nbc^s(x,\Lambda) = nbc^s(0,\Lambda)$.

Dans ce paragraphe, on considère que le programme est modélisé par une suite de tâches T^1,..., T^p telles que le graphe de dépendances associé soit acycliqe.

En utilisant le coût des calculs/communications $ct()$ définie en (§ 4.2, p. 168) , on en déduit donc, une majoration du temps de latence :

$$\sum_{k=1}^{p} nbc^k(0,\Lambda).ct(k)$$

où $nbc^k(0,\Lambda)$ est le nombre d'événements concernant la tache k, ayant lieu dans l'intervalle de longueur Λ (avec Λ le nombre de tops maximal entre une entrée et une sortie). La valeur $ct(k)$ (cf. § 5.1, p. 169) est la durée physique de l'exécution d'une partition intégrant sa communication. Une condition suffisante pour satisfaire une latence applicative, est donc :

$$\sum_{k=1}^{p} nbc^k(0,\Lambda).ct(k) \leq \text{latence} \tag{10.16}$$

6 Contraintes d'entrées-sorties

Les contraintes de périodicité permettent de donner les cadences de lecture et d'écriture des éléments de tableaux pour une tâche donnée. En fait, on s'intéresse surtout à ces cadences pour les traitements d'entrée comme pour les traitements de sortie de l'ensemble du graphe de flot de données.

6.1 Contraintes de récurrence d'entrée

Plus communément appelé *récurrence de l'application*, elle porte sur le tableau d'acquisition de l'application, *i.e.* c'est la durée réelle qui sépare les écritures successives du capteur dans le tableau d'acquisition lu par l'application. Cela correspond à la période d'acquisition de toutes les données entre t et $t+1$ et c'est en fait l'inverse de la fréquence d'échantillonnage (en Hertz).

6.2 Contraintes de récurrence de sortie

La récurrence de sortie est liée au tableau de sortie, elle donne la durée séparant l'écriture de deux résultats successifs dans ce tableau pour l'ensemble de l'application. C'est en fait la période de sortie des données, inverse de la fréquence de sortie (en Hertz).

6.3 Calcul d'une contrainte de récurrence

Que ce soit pour les sorties ou pour les entrées, on utilise la même formalisation pour les contraintes d'entrées-sorties. En extrapolant, cette formalisation peut s'adapter à n'importe quel nid de boucles si cela est demandé. On procède de la même façon que pour la contrainte de latence, sauf que l'on associe à la distance en question un volume de donnée aquis (resp. produit). Les programmeurs calculent généralement un seule type de récurence, par exemple en radar les contraintes de sorties se dérivent en fonction des entrées. En programmation multimédia, o'u l'on a un signal video à restituer, les contraintes d'entrées se calculent en fonction des sorties.

6.3.1 Distance temporelle entre acquisitions (resp. productions)

La distance considérée ici est celle qui sépare l'exécution de deux itérations sur la dimension infinie. Soit $\mathbf{1}_\infty$ le vecteur permettant d'augmenter d'un uniquement la composante infinie associée à $\mathbf{c}$. La distance séparant le début de deux groupes d'acquisition (resp. production) est :

$$\gamma \ = \ \| \ d(\mathbf{c}) - d(\mathbf{c} + \mathbf{1}_\infty) \ \|$$

Trivialement on a : $\gamma \ = \alpha_\infty$.

6.3.2 Volume de données aquis (resp. produits)

Le volume de données V_{acq} (resp. V_{rec}) correspond aux volume d'éléments de tableaux référencés en lecture (resp. écriture) entre deux tops d'acquisition pour l'itérateur infini. Il faut calculer les volumes de la même façon que dans chapitre 8, § 2, p. 143 :

$$V_{acq} \ = \ card\left(\left(\bigcup_{\mathbf{c}+\mathbf{1}_\infty} \mathcal{R}^r(\mathcal{P}_{(\mathbf{c}+\mathbf{1}_\infty)}) \right) \setminus \left(\bigcup_{\mathbf{c}} \mathcal{R}^r(\mathcal{P}_{(\mathbf{c})}) \right) \right)$$

En considérant, comme pour le modèle mémoire (cf. chapitre 8, § 2.1.3, p. 146), les restrictions d'accès aux tableaux, on peut exprimer ces cardinalités uniquement par rapport à l'itérateur infini :

$$\forall c_\infty, \ V_{acq} \ = \ card\left(\bigcup_{\mathbf{0} \le \mathbf{c} < up(\mathbf{c})} \mathcal{R}^r(\mathcal{P}_c) \right)$$

On peut procéder comme pour le modèle mémoire pour définir une surapproximation uniforme de ce volume d'acquisition (resp. de production), voir chapitre 8,§ 2, p.143. Soit $\hat{V}_{acq}$ ce volume de données.

6.3.3 Contrainte physique de récurrence

La spécification de la récurrence est le rapport de volume de données acquis (resp. produit) sur un temps **constant** donné. C'est la contrainte de récurrence constante. La contrainte physique de périodicité permet de garantir qu'un volume de données supérieur ou égal à celui de la spécification sera acquis (resp. produit) pour un même temps constant. Or, il est possible de calculer le temps physique tp^k écoulé (fig. 10.4-b, p. 170) dans une distance γ, pour laquelle V_{acq} données ont été aquises (resp produites) :

$$tp^k = \sum_{k=1}^{p} nbc^k(0,\gamma).ct(k)$$

Enfin, sachant que pour le nid de boucle k, $det(P^k)$ processeurs sont utilisés (cf. chap. 9, § 2.1, p. 151), la contrainte de récurrence se traduit par :

$$recurence \geq \frac{tp^k}{det(P^k)}$$

où *recurence* est une contrainte spécifée par l'utilisateur.

6.3.4 Vers une contrainte de récurrence multi-dimensionnelle

La prise en compte d'une récurrence seulement définie sur la dimension infinie ne suffit pas pour permettre de modéliser les entrées ou/et sorties de certaines applications. En effet, il est parfois nécessaire d'avoir une récurrence multi-dimensionnelle, (i.e. ne portant pas uniquement sur la dimension infinie). On peut donc avoir une contrainte du type (fig. 10.4-b, p. 170) pour chacune des dimension d'itérations afin de pouvoir garantir une spécification plus fine. Ce type de récurrence multi-dimensionnelle porte le nom de contrainte de récurrence d'acquisition en rafale. Enfin, l'avantage de traiter des récurrences multi-dimensionnelles est de permettre de sélectionner directement des ordres lexico-graphiques parmi l'ensemble des choix d'ordres lexicos disponibles.

7 Conclusion

Ce modèle traite les deux aspects du temps réel, c'est-à-dire des contraintes d'acquisition et de restitution des résultats ainsi que la latence. La modélisation soulève des problèmes théoriques résolus dans le cas particulier de la contrainte de croissante suffisante. Les contraintes de cardinalité et distances permettent ainsi de modéliser la latence applicative. Les entrées-sorties, quand à elle, se traduisent par des contraintes de cardinalité qui sur-contraignent l'ordonnancement local sous-jacent aux traitements de sorties ou d'entrées. Comme pour le modèle mémoire, les modèles de latence et temps réel sont définis à deux niveaux de granularité. Le niveau global considère les blocs de traitements issus de l'ensemble des nids de boucles, tandis qu'au niveau local, c'est le volume des blocs qui est considéré. Ici encore, la granularité que le partitionnement confère à l'ensemble des modèles permet au système de placement de trouver automatiquement le seuil pertinent entre ces deux niveaux.

Conclusion

Nous avons définit l'ensemble des modèles nécessaires au placement automatique en tenant compte de l'ensemble des aspects de la problématique. Nous avons montré quelles relations permettaient de maintenir la cohérence des solutions locales à un modèle par rapport aux autres. Parmis ces relations, les dépendances de flot ont fait l'objet d'un chapitre. Les autres relations sont implicites à la présentation de la modélisation. Beaucoup de modèles, notamment celui de la mémoire et du temps réel disposent de plusieurs niveaux de granularité grâce aux propriétés du modèle de partitionnement. Ceci est en fait possible qu'au prix de contraintes non-linéaires reliant les différents modèles.

Etant donné l'importance des relations pour la compréhension de la structure du problème, nous explicitons celles-ci globalement à partir des variables et formulation définies dans les modèles.

Ordonnancement-Partitionnement : Ces deux modèles sont naturellement liés par la contrainte de croissance qui s'établit à partir des vecteurs $up(\mathbf{c})$ (chap. 7, § 2.2, p. 2.2).
Ils sont aussi et surtout liés par les dépendances qui font l'objet d'un chapitre entier, à cause de la complexité de leur représentation (chap. 5, p. 5). Celles-ci engendrent des contraintes de précédences (chap. 7, § 3.1, p. 3.1) et permettent de calculer les distances maximales (chap. 7, § 5.1.2, p. 5.1.2).

Temps Physique-Interphase : Ce même calcul de coût physique est aussi basé sur le coût des communications formulées dans le modèle d'interphase. (resp. chap. 10, § 4.2, p. 4.2 et chap. 6, § 4.6, p. 4.6).

Partitionnement-Temps Physique : Le nombre de blocs exécutés séquentiellement sur un processeur dans une période, formulé à l'aide de la contrainte de cardinalité, donne le coût physique associé à un événement (chap. 10, § 4, p. 4). Le nombre de processeurs sur lesquels ont effectués une partie des acquisitions est aussi un lien (chap. 10, § 6.3.3, p. 6.3.3).

Ordonnancement-Temps Physique : Un même type de contrainte est utilisé au niveau de cette relation puisqu'elle donne le nombre d'événements apparus dans la distance de dépendance totale. En effet, elle utilise aussi une contrainte de plus long chemin sur le graphe de flot de données résultant du partitionnement (chap. 10, § 5, p. 5).

Interphase-Partitionnement : Le calcul du nombre de messages à échanger requiert l'utilisation des matrices de partionnement pour déterminer les zones de tableau accédées et en déduire une sur-approximation des échanges de données. Ces deux modèles sont aussi liés par les dépendances qui permettent de déterminer le nombre partitions potentielement réceptrices (chap. 6, § 4.6, p. 4.6).

Mémoire-Partitionnement : Les volumes de données accédées permettent d'établir le modèle de mémoire capacitif. Ces volumes sont connus à partir du partitionnement et des fonctions d'accès aux données (chap. 8, § 2.1.1, p. 2.1.1).

Ordonnancement-Mémoire-Architecture : La contrainte permettant de comptabiliser les blocs de données d'un même nid de boucles partitionnés devant être alloués dans la distance de dépendance. Cette contrainte repose sur une distance définie à partir de l'ordonnancement et sur la contrainte de cardinalité. C'est en fait la durée de vie des blocs de données alloués. La mémoire totale est la contrainte permettant de lier les modèles d'architecture et de mémoire. Le nombre de processeurs et le volume capacitif de données par processeur permettent d'établir la mémoire nécessaire (chap. 8, § 3.3, p. 3.3).

Interphase-Architecture : Les valeurs de bande passante et de temps d'accès au réseau rendent ces deux modèles connexes (chap. 6, § 2.5.3, p. 2.5.3).

Architecture-Partitionnement : La contrainte du nombre de processeurs est le lien entre l'architecture et le modèle de partitionnement (chap. 9, § 2.1, p. 2.1).

Voici, en conservant une vision par modèle, un résumé des principales contraintes développées dans cette partie, dont certaines sont bien entendu des liens explicités ci-dessus :

Modèle	**Interprétation**	**Contraintes**
PARTITIONNEMENT	Dimension (temps,processeur,calcul) = $(\mathbf{c}, \mathbf{p}, \mathbf{l})$ Dimensions des itérations Partitionnement uniforme parallèle aux axes Il y a au moins 1 processeur Il y a au moins 1 une itération en mémoire Non redondance des calculs dans le temps Non redondance des calculs sur les processeurs	$\mathbf{i} = L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}.$ $\forall \mathbf{i},\ \mathbf{0} \leq \mathbf{i} < up(\mathbf{i})$ P et L diagonales $\det(P) \neq 0$ $\det(L) \neq 0$ $\forall \mathbf{p},\ \mathbf{0} \leq \mathbf{p} < P.\mathbf{1}$ $\forall \mathbf{l},\ \mathbf{0} \leq \mathbf{l} < L.\mathbf{1}$
ORDONNANCEMENT	Deux blocs issus de deux nids de boucles distincts ne s'exécutent pas en même temps. Modèle SIMD ou SPMD implicite. Deux blocs issus d'un même nid de boucle ne s'exécutent pas en même temps	$N =$ Nbr. de T.E., $\forall k = 1, ..., N$ $d^k = N(\alpha^k \mathbf{c}^k + \beta^k) + k$ $\alpha_n^k \geq 1$ et $\alpha_i^k \geq \alpha_{i+1}^k up(c_{i+1}^k)$
DÉPENDANCES	Une donnée ne peut être lue avant d'avoir été écrite $D(\mathbf{i}^w, \mathbf{i}^r) \Rightarrow d^w(\mathbf{c}^w) + 1 \leq d^r(\mathbf{c}^r)$	$\forall(\mathbf{i}^w, \mathbf{i}^r)$ sommets du polytope engendré par $D(\mathbf{i}^w, \mathbf{i}^r)$ on a, $d^w(\mathbf{c}^w) + 1 \leq d^r(\mathbf{c}^r)$

Modèle	Interprétation	Contraintes
Mémoire	Approximation de la capacité mémoire	$2 \max_{k=1..N}(VolEcr^k) + \sum_{k=1..N}(VolLec^k.NbBlocLec^k)$
Signal	Calcul du nombre d'événements apparus avant $\mathbf{c}$ Distance maximale événementielle (dans une chaîne) Coût calculs-communications Sur-approximation de la latence physique Sur-approximation de l'acquistion physique	$nbc(0, d(\mathbf{c})) = \mathrm{Card}\{\mathbf{c}' \vert \mathbf{c}' \ll \mathbf{c}\}$ $\Lambda = \max(d^N(\mathbf{c}^N) - d^1(\mathbf{c}^1))$ $cout^k_{elem} + (\det(L)-1).\max(cout(V^k_{te}), cout^k_{elem}) + cout(V^k_{te})$ $\sum_{k=1..N} nbc(0, \Lambda).ct(k) \leq Latence.$ $tp(k) = \sum_{k=1..N} nbc(0, \alpha^1_\infty).ct(k)$
Interphase	Linéarisation des processeurs Nombre de récepteurs Volumes élémentaires Coût de communications	$\forall\ j,\ 0 \leq j < n,\ 0 \leq p_j < P_{jj},$ $\mathcal{L}(p_0, \ldots, p_{n-1}) = \sum_{j=0}^{n-1} (\prod_{k=0}^{j-1} P_{k,k}).p_j,$ $\forall c^w,\ \mathcal{V}^w_{emis}(\mathbf{c}^w) = \sum_{\forall c^r \in \mathcal{D}(c^r, c^w)} \mathcal{V}^{r,w}_{elem}(\mathbf{c}^r, \mathbf{c}^w)$ $\forall (\mathbf{c}^r,\ \mathbf{c}^w),\ \mathcal{V}^{r,w}_{elem}(\mathbf{c}^r,\ \mathbf{c}^w) =$ $\sum_{\{(\mathbf{p}^r, \mathbf{p}^w) \vert \mathcal{L}^r(\mathbf{p}^r)\ \neq\ \mathcal{L}^w(\mathbf{p}^w)\}} \mathrm{card}\ \left(\mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r, \mathbf{p}^r)}) \cap \mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w, \mathbf{p}^w)})\right)$ $cout(\mathcal{V})\ =\ b.\mathcal{V}\ +\ a$
Machine	Nombre de processeurs Type SIMD Type SPMD	$\det(P^k)$ $up(\mathbf{c}^k) = P^{-1}_{kk} L^{-1}_{kk} up(i_k)$ $up(\mathbf{c}^k) = \lceil P^{-1}_{kk} L^{-1}_{kk} up(i_k) \rceil$

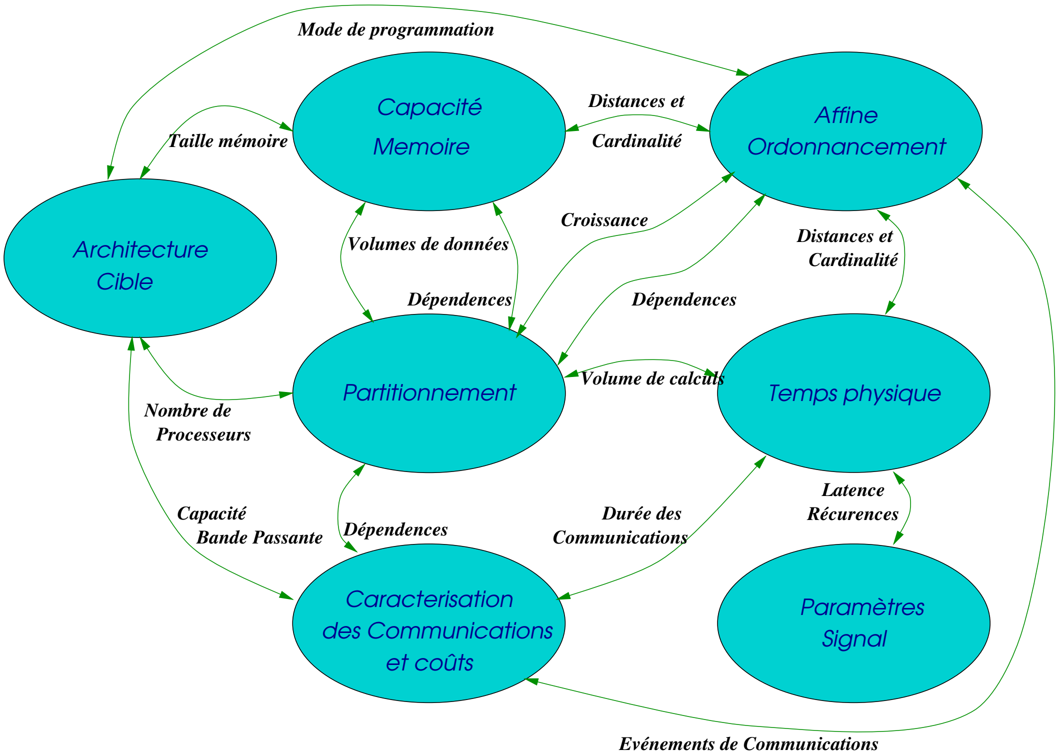


FIG. 10.5. – *Représentations en modèles concurrents relationnels du problème du placement*

Les liens entre modèles et relations peuvent aussi être résumés par la figure (10.5).

Nous avons établis l'ensemble de la modélisation mathématique du placement global, la partie suivante comment il est résolus.

Quatrième partie

Système de résolution pour le placement automatique en programmation logique avec contraintes

Introduction

La partie précédente a présentée comment la modélisation était construite. Celle-ci est munit d'entités et de relations décrivant des caractéristiques et fonctionnalités physiques. Chaque sous-problème du placement global est NP-difficile comme nous l'avons souligné à plusieurs reprises en (chap. 1). En outre, la modélisation présente des contraintes complexes et non-linéaires. Ceci n'est pas surprenant, puisque Paul Feautrier montre dans [Fea94a] que résoudre un ordonnancement à grain fin en tenant compte de plusieurs contraintes de ressources nécessite la résolution de contraintes non-linéaires. Selon lui, la question est : comment combiner la programmation en nombre entier et la recherche de solutions ?

Nous choisissons justement de suivre une approche qui associe la résolution de contraintes et le non-déterminisme. Nous allons montrer dans cette partie que cela nécessite d'associer à la modélisation une approche de résolution concurrente. Cette partie est entièrement consacrée à la résolution des modèles et relations. Nous présentons l'approche par modèles concurrents (chap. 11), qui conduit à une richesse de description du problème ainsi qu'un contrôle de la résolution sans précédent. Pour cela, les modèles concurrents exploitent les propriétés de la programmation logique avec contraintes et font l'objet d'une implantation.

De plus, cette approche confère au concepteur une grande liberté de développement. Les modèles et relations peuvent être construits indépendament (chap. 12). Aussi, nous choisirons l'implantation la plus naturelle, fidèle à la physique du problème tout en suivant la modélisation mathématique. Mais les avantages de cette liberté ne s'arrètent pas à une meilleure compréhension de la structure du problème. En effet, la possibilité de manipuler les modèles, tant sur les plans syntaxiques que sémantiques, garantit de surcroît la maintenance et la réutilisabilité des modèles. De même, l'utilisation quasi-automatisée du système de placement, en imposant des solutions partielles permet en outre sa mise au point.

Néanmoins, la difficulté globale nécessite l'introduction d'heuristiques (chap. 13, § 2) pour pouvoir converger automatiquement vers des solutions intéressantes. On utilise alors des heuristiques de domaine, comme favoriser le parallélisme, ou le recouvrement des communications. Ces heuristiques interviennent sur le choix des valeurs en utilisant des contraintes particulières (recouvrement des communications, alignement, ...) ou en choisissant un ordre d'énumération (croissant, décroissant ...). De plus, la richesse de description du problème permet la réalisation d'heuristiques sophistiquées. Enfin, les stratégies dédiées aux critères de résolution (chap. 13, § 4) régissent les heuristiques de même que le choix des variables à énumérer.

Le chapitre (11) présente les langages de Programmation avec Contraintes et l'approche de résolution concurrente. L'implantation est décrite dans le chapitre (12) tandis que le chapitre (13) explique la nature et la fonction des heuristiques et des stratégies de résolutions.

Chapitre 11

Méthode de résolution par modélisation concurrente

Introduction

Les méthodes de résolution jusqu'à présent éprouvées pour considérer le problème du placement sont essentiellement basée sur la Recherche Opérationnelle (RO) et la Programmation Linéaire en Nombre Entier (PLNE) (chap. 3). La RO permet d'aborder chaque sous-problème du placement à l'aide d'un algorithme de résolution, généralement basé sur une heuristique. Cette approche aboutit à plusieurs algorithmes difficiles à combiner pour adresser le problème global. L'approche basée sur la PLNE est intéressante car elle constitue un premier pas vers la modélisation. Cependant, celle-ci nécessite l'ajout de variables nécessaires à l'expression du contrôle de la résolution. Or, cette dernière ne distingue pas la fonction des variables si bien que la combinatoire du problème n'est pas contrôlable. De plus, à l'echelle du placement global, le contrôle de la combinatoire est crucial. C'est pourquoi nous choisissons une approche basée sur la Programmation Logique avec Contraintes concurrentes qui permet la désarticulation de la modélisation avec le contrôle de la résolution. Contrairement aux résolveurs classiques, qui considèrent la résolution comme une boîte noire, ces langages sont basés sur une résolution incomplète. Cette approche utilise donc des opérateurs de contrôle permettant de définir de nouvelles contraintes. Cependant, rester au niveaux des contraintes atomiques ne permet pas de manipuler efficacement le problème. Comprendre et résoudre le problème nécessite de le considérer globalement. Pour cela, il est nécessaire de se situer au niveau du modèle qui représente une fonctionalité du problème. Aussi, nous introduisons la méthodologie de modélisation par contraintes concurentes qui permet l'implantation d'environnements d'optimisations.

Nous donnons justement comment, en (§ 1) cette désarticulation est rendue possible dans les langages de programmation logique avec contraintes concurrentes. Issues de cette notion, les propriétés de ces langages – compositionalité, généricité – conduisent à l'expression et la résolution des différents modèles de placement (§ 2).

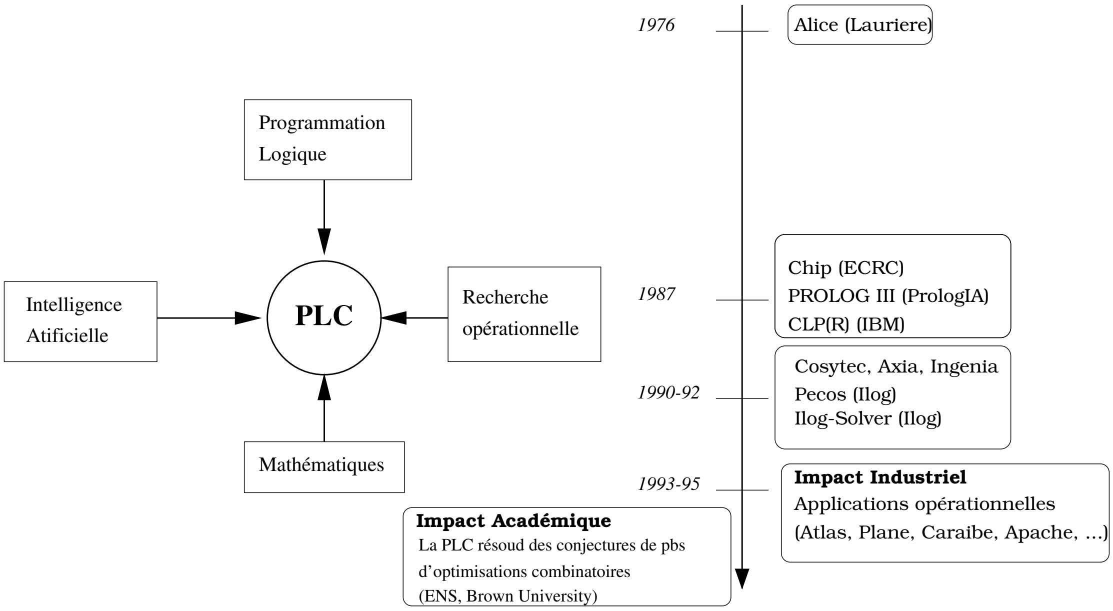


FIG. 11.1. – *PLC : Historique*

1 La Programmation Logique avec Contraintes

La Programmation Logique avec Contraintes (PLC), de par ses capacités de base à résoudre les problèmes d'optmisation combinatoire apporte un ensemble de propriétés nécessaires pour appréhender de manière global le problème du placement.

1.1 Historique

La PLC est issue du domaine de l'Intelligence Artificielle. Cependant ses fondement reposent sur les mathématiques et la programmation logique (voir figure 11.1).

De ce fait, ces langages permettent naturellement d'exprimer des concepts à l'aide de formalismes[1].

D'un point de vue résolution, la PLC héritent aussi des techniques de recherches de solutions développées dans le cadre de la recherche opérationnelle (RO). Dès 94, les réalisations PLC rivalisent avec des algorithmes spécialisés de recherche opérationnelle pour des problèmes d'ordonnancement [CL94].

Cependant, la réelle efficacité des réalisations PLC dépend souvent de trois notions : les contraintes redondantes efficaces, les heuristiques et les stratégies d'énumérations spécifiques parfois difficiles à établir. La portée de celles-ci est d'autant plus étroite que le système est complexe et les contextes d'utilisations variés. L'environnement de programmation que représente aujourd'hui la PLC permet aussi d'introduire d'autres techniques de résolution.

Liberté d'expression et intégration : La PLC permet donc d'accueuillir des structures mathématiques dans un cadre formel basé sur la logique. La résolution fait appel à des algorithmes de RO, de programmation linéaire ou d'Intelligence Artificielle. De nombreuses contraintes

[1]La formalisation mathématique des éléments et des fonctions du problème étudié représente à la fois le point fort et le point faible de l'approche "*modélisation concurrente par contraintes*". En effet, l'axiomatisation ne peut pas être faite sans contraintes car il faut garantir une certaine efficacité. Cette efficacité est essentiellement basée sur une axiomatisation ne sortant pas d'une sémantique mathématique suffisamment simple pour être efficacement implémentable.

génériques ou spécialisées peuvent alors être implémentées. De surcroît, grâce aux connecteurs logiques, il est possible de composer un large ensemble de contraintes, c'est l'aspect déclaratif de la composition.

Puissance de résolution : Dans un système de PLC(X), la notion de solution est aussi étendue, on ne cherche plus à instancier les variables d'un but. Une solution en PLC est en fait un système de contraintes définissant l'ensemble des solutions pour lesquelles le but est satisfait.

Dans toutes les approches logicielles, l'état d'un système est caratérisé par le contenu de la mémoire à un instant donné. Les opérations élémentaires sont la lecture et l'écriture sur, ou à partir de la mémoire. L'état d'un système n'est alors caractérisé que par l'ensemble des valeurs des cases mémoires associées aux variables qui le composent. La différence fondamentale entre l'approche PLC et les autres approches logicielles est la représentation de cette mémoire. Dans le cas de la PLC, la mémoire n'est pas réduite à un ensemble de cases mémoires mais constitue en elle-même une contrainte. Cette dernière est capable de fournir de l'**information partielle** sur l'ensemble des variables qui composent le système.

Les langages de PLC acceptent donc en paramètres le type de domaine numérique X et sont notés PLC(X) [JL87]. On associe au paramètre X, une structure mathématique S définie à partir des contraintes élémentaires comme $\{=, <, \neq\}$ (voir figure 11.2). Cette structure fixe l'interprétation des prédicats contraintes. Les opérateurs de ces structures arithmétiques correspondent à des primitives de résolution et de satisfaction de contraintes, permettant d'étendre la simple unification. La logique du premier ordre confère à tous programme de PLC une véritable sémantique logique [Fag94]. Le dernier stade d'évolution expose des opérateurs de contrôle nous permettant d'introduire la programmation par modèles concurents.

1.2 Programmation Logique avec contraintes concurrentes

Dans les langages concurrents de contraintes, les méchanismes de résolution cèdent la place aux primitives `Ask` et `Tell`, permettant de statuer une contrainte, suspendre un but et synchroniser la résolution. Le `Tell` permet de statuer une contrainte. Le `Ask` permet de tester l'implication d'une contrainte pour permettre de déclencher un but[2].

- L'opération qui permet de fournir une nouvelle information partielle au système est usuellement appelée opération **Tell**.
- L'opération qui permet d'obtenir de l'information partielle à partir de la mémoire est usuellement appelée opération **Ask**.

Les primitives `Ask` et `Tell` forment un puissant paradigme de programmation[Mah87, Sar87b], permettant d'articuler la résolution d'un problème combinatoire. Dans ce cadre, opérateurs et contraintes peuvent facilement être définis, avec les primitives `Ask` et `Tell` introduites en [Sar87b]. Elles permettent d'implanter de nouvelles contraintes en définissant des règles de propagation, de suspension et de synchronisation.

Par composition, ce paradigme permet de synchroniser et coordonner les étapes de résolution de plusieurs processus concurrents en définissant des règles de propagation de contraintes. De puissantes stratégies de résolution combinant différents heuristiques sont exprimables sur ces bases, c'est l'aspect opérationnel de la composition.

[2] Ceci est équivalent à suspendre l'exécution d'un but tant que la contraintes n'est pas impliquée.

1.3 Propriétés de la Programmation Logique avec Contraintes

L'ensemble des systèmes de Programmation avec Contraintes permet de composer les contraintes à l'aide d'opérateurs définit par leur sémantique déclarative. C'est cette propriété de compositionalité qui apporte la facilité de statuer les contraintes décrivant un problème. C'est aussi cette propriété qui permet de créer de nouvelles contraintes basées sur des opérateurs atomiques ou/et des contraintes déjà existantes. Néanmoins, exprimer un problème comme une composition de contraintes ne garantie pas l'efficacité de la résolution. Aussi, à chaque opérateur, est associée une sémantique opérationnelle permettant de contrôler et coordonner la résolution.

1.3.1 La $\wedge$-composition

Plusieurs opérateurs ont été définis selon les structures pour pouvoir composer des contraintes. La $\wedge$-composition déclare une conjonction de contraintes. Opérationnellement, elle est la base des méthodes de résolution déclarative de la plupart des systèmes de PLC. Elle permet de décomposer récursivement un problèmes en un ensemble de sous-problèmes résolus séparemment. Cela signifie que le résultat d'une conjonction de contraintes peut être calculé à partir du résultat de chaque contrainte[FLMP89].

1.3.2 Extension de la composition en Programmation Logique avec Contraintes concurrentes

Le passage au niveau langage de Programmation avec Contraintes concurentes étend la notion de composition grâce aux opérateurs `Ask` et `Tell`. Ces deux opérateurs permettent de définir les suspensions et activations des modifications portant sur les domaines de variables. Lors de la définition d'une nouvelle contrainte, il faut établir ces suspensions, activations et modifications pour chaque variable. La difficulté réside d'une part dans l'analyse des invariants permettant de spécifier les suspensions et activations. D'autre part, il faut programmer la nature des modifications portant sur les variables. Une nouvelle contrainte pourra alors se composer avec d'autres contraintes pour représenter les différents problèmes ou sous-problèmes de la réalité.

1.3.3 Efficacité dans les domaines finis

La structure des domaines finies permet la construction d'opérateurs pour lesquels la sémantique opérationnelle est reconnue efficace. Nous citerons en exemple l'implication[VSD92c], la disjonction constructive[JS93] ou la cardinalité[Van93]. Nous utilisons cette structure dans cette thèse.

Opérationnelement, l'ensemble de ces opérateurs propagent de l'information sur les bornes des variables contraintes grâce au paradigmes `Ask` et `Tell`. Ainsi, on peut composer autant de contraintes que l'on souhaite tant que l'on est capable de maintenir la cohérence des contraintes. Nous donnons, à titre informatif, comment le maintient de la cohérence est implanté de façon sous jacente au langage de programmation avec contrainte. Il s'agit d'**algorithmes génériques** dit "arc-consistency" reposant sur un graphe de contraintes. Ce genre d'algorithme s'apparente à la description suivante :

- *Pour une variable $\mathcal{V}$ dont le domaine est modifié,*
- *Selectionner l'ensemble des contraintes impliquant $\mathcal{V}$, modifier si nécessaire tous les domaines des variables V_i impliquées dans ces contraintes.*
- *Si un domaine est vide alors la cohérence n'est plus assurée. Arrêt de l'algorithme.*
- *Si toutes les variables sont instanciées par une valeur, arrêt de l'algorithme.*
- *Pour chaque V_i modifiée, reprendre au début.*
- *S'il n'y a plus de variable à modifier, arrêt de l'algorithme.*

Lors d'un arrêt sur un échec du maintiens de la cohérence, le systéme doit pouvoir provoquer un événement de retour-arrière ("backtrack"). Une procédure de résolution basée sur un tel algorithme explore alors un espace de recherche délimité par les événements provoqués par cet algorithme.

1.3.4 La généricité

L'avantage de pouvoir programmer en logique est considérable pour pouvoir exprimer des combinaisons de contraintes. Il est alors possible d'utiliser des formules de logique dans un langage de premier ordre. Ce concept est un des pilliers de la modélisation concurrente.

Pour illustrer ceci, considérons désormais la définition d'une nouvelle contrainte à partir de contraintes existantes. Par la compositionalité, l'expression syntaxique de cette contrainte suffit à garantir la cohérence globale de l'ensemble des méchanisme de contrôle évoqués ci-dessus.

Nous donnons à titre d'exemple l'opérateur d'implication [VSD92c] utilisé par beaucoup de contraintes de cette thèse :

$$(c \Rightarrow G) \leftarrow (c \rightarrow G) \wedge (\neg c \rightarrow true)$$

Où G est un but et c une contrainte, le test d'implication est noté $\rightarrow$ et $\Rightarrow$ l'opérateur d'implication.

De nombreuses autres contraintes ont ainsi été définies, comme la propagation conditionnelle, l'équivalence entr contraintes, la disjonction constructive ou la cardinalité **??**. Disposer d'un ensemble de contraintes de haut niveau pour résoudre différentes classes de problèmes démontre la généricité de la PLC.

Cependant, il est envisageable de construire des opérateurs propres au domaine de la parallélisation/placement automatique. Un opérateur de *minimum* lexical a d'ailleurs été développé sur ces bases. Il n'a pas été utilisé, car la modélisation actuelle du problème du placement occulte cette contrainte. Ce type d'opérateur est l'objet de recherches futures. Générique au problème du placement, de tels opérateurs permettraient de résoudre plusieurs instances du problème de placement.

1.4 Optimisation de fonctions de coûts

L'optimisation combinatoire se traduit par la minimisation (resp. maximisation) de critères. Les systèmes de programmation avec contraintes mettent en oeuvre une technique de résolution semblable au "Branch and Bound"[Van95] pour converger vers la solution minimale (resp. maximale) entière. Dans le cas des domaines finis, cette phase de minimisation (resp. maximisation) peut facilement se combiner avec la phase de recherche de solutions. Soit $c = f(v_1, \ldots, v_n)$ un coût à minimiser (resp. maximiser), où $v_1, \ldots, v_n$ sont des variables du problème :

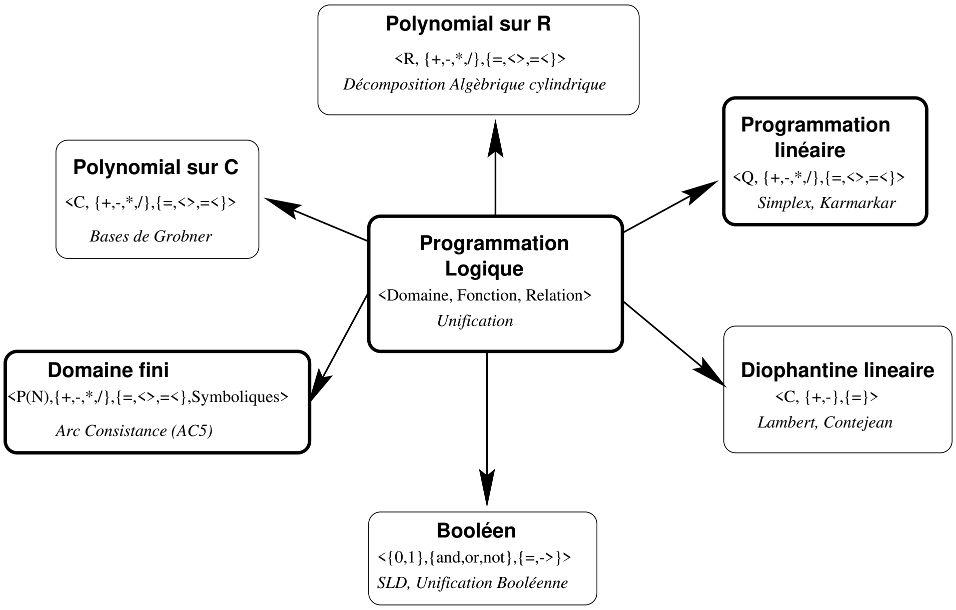


FIG. 11.2. – *Les différents domaines couverts par la PLC*

- *Instantier chaque variable par un parcours d'espace de recherches en maintenant la cohérence du graphe de contraintes.*
- *S'il n'y a pas cohérence lorsque l'on a parcouru l'espace de recherche, soit il n'y a pas de solution, soit le coût à prendre en compte est c_{i-1}. Arrêt de l'algorithme.*
- *Sinon, on obtient le coût c_i, considérer la contrainte associée au nouveau coût : $c_{i+1} < c_i$ et reprendre au début.*

1.5 Les domaines et techniques de résolution

Comme nous l'avons déjà expliqué, les systèmes de programmation logique avec contraintes acceptent en paramètre un domaine de résolution comme les entiers, rationnels ou réels. Selon ces domaines, différentes techniques de résolution sont à considérer. Dans tout les cas, la programmation logique acceuille un algèbre sur un domaine associé à la technique de résolution dédiée.

Les structures mathématiques à même de considérer les problèmes de parallélisation associées aux principes de résolutions sont présentées dans cette section.

Les Rationels :

La résolution sur les rationels utilise directement les résultats de la programmation linéaire. La structure mathématique est $< \mathbb{Q}, \{+, -, *, /\}, \{=, \neq, \leq\} >$ et permet de prendre en compte des contraintes linéaires du type :

$$A.\mathbf{x} \leq \mathbf{b}$$

Ce système décrit un polytope dans $\mathbb{Q}$, L'algorithme de résolution la plus courante dans les systèmes de PLC est le Simplex, utilisant une representation basé sur les sommets du polytope.

Les Booléens :
La structure est l'algèbre de Boole $< \{(0,1), \{\vee, \wedge, \neg\}, \{=, \Rightarrow\} >$, qui permet de poser des contraintes booléennes comme :

$$\neg x \vee y \Leftarrow z$$

Les domaines finis :
L'utilité de l'algèbre sur les domaines finis en particulier sur les problèmes d'ordonnancement n'est plus à démontrer, et de nombreux systèmes de résolution ont vu le jour dans diverses entités industrielles ou universitaires. Les systèmes de résolution CHIP de Cosytech et ILOG SOLVER d'Ilog sont basés sur cette algèbre. Le système PLC($\mathbb{R}$) d'IBM travaillant dans les réels était quand à lui, basé sur un Simplex. La structure de calcul dite des domaines finis[3] a fait l'objet de nombreux travaux. La facilité de manipulation des représentations font de ce domaine un champs d'application privilégié pour l'implantation des opérateurs `Ask` et `Tell`.

1.6 Les systèmes de programmation avec contrainte

Plusieurs systèmes de programmation avec contraintes ont vus le jour dans les années 80 (voir figure 11.1), certains de ces sytèmes travaillent sur plusieurs domaines de calcul, et combinent ainsi plusieurs méthode de résolution. Nous en donnons une liste non exhaustive et présentons le système de programmation logique avec contraintes utilisé.

1.6.1 Exemple de systèmes

Voici un ensemble de langages définis à partir des clauses de Horn et admettant plusieurs domaines de calcul :

PLC($\mathbb{R}$) : Ce système [JM87] dispose d'une arithmétique linéaire sur les réels, les contraintes non-linéaires ne sont pas traitées sauf lorsqu'elles deviennent linéaires.

CHIP : Initialement développé par l'ECRC (*European Computer Industry Research Center*) est actuellement commercialisé. Il permet de calculer sur les booléens et traite les contraintes linéaires sur les rationels. Il dispose d'une arithmétique sur les entiers bornés, aussi appelés domaines finis[DHS+88]. Le système intègre plusieurs méchanismes de résolution, basés sur les méthodes de Programmation Linéaires en Nombres Entier et sur les domaines finis.

Eclypse : Dans la lignée de CHIP, ce langage intègre plusieurs domaines de calcul. La particularité d'Eclypse est de pouvoir combiner un système de résolution par propagation de contraintes assoxiés à des outils de recherches locales de solutions, comme les recherches par réparation.

PrologIII-IV : Le langage PrologIII [Col87], successeur de PrologII, possède booléens et rationnels comme domaines de calcul. Sur les rationels, ce langage traite les contraintes linéaires, et base son méchanisme de résolution sur un algorithme Simplex [Col90]. L'intégration d'un Simplex dans le cadre de la programmation logique a donc fait l'objet de nombreux travaux. Cependant, la complétude du résolveur ne permet aucune souplesse au niveau de la résolution. Son successeur, PrologIV intègre plusieurs méthodes de résolution.

D'autres langages associant Programmation Logique et Contraintes sur différents domaines de calculs font encore l'objet de nombreux travaux :

[3] Une variable de domaine est définie dans une partie finie de $\mathbb{N}$

CC(FD) : Est un environnement permettant de construire des contraintes concurrentes sur les domaines finis, à l'aide des opérateurs `Ask` et `Tell`. [VSD92b]

Oz : Est un langage utilisant la notion d'objet développé par le DFKI qui privilégie à la manière de CC(FD) la résolution concurrente des contraintes. Ce langage est la base d'un modèle plus général de programmation utilisant le concept d'agents concurrant [Smo95].

Ilog-Solver : Est un produit commercial de Programmation avec Contraintes basé sur les domaines finis et les rationnels [ILO92]. La résolution combine plusieurs méthodes.

Claire : Est un langage permettant de définir contraintes et méthodes de résolution. Celles-ci peuvent être spécialisées ou génériques [LC96]. Le langage permet de construire facilement des arbres de recherches en utilisant la notion d'objet et la transmission d'événements.

Il va sans dire que la coexistence de plusieurs structures dans un même système pose de sérieux problèmes de résolution. C'est encore un problème ouvert qui fait l'objet de nombreux travaux. Le système utilisé [CFJ$^+$92] dans le cadre de cette thèse et qui a servit à implanter les différents modèles est Meta(F). Ce langage Meta(F) a été developpé par Thomson-CSF/LCR, d'une part pour mener des recherches sur la résolution de contraintes, sur le backtrack intelligent et l'incrémentalité, d'autre part pour expérimenter la résolution d'applications réelles. Il est très orienté optimisation combinatoire et est basé sur les méchanismes de résolution concurrente (paradigme `Ask` & `Tell` sur la structure des domaines finis $< \mathcal{P}(\mathbb{N}), \{+,-,*\}, \{<,=\} >$. Il utilise un algorithme de maintient de la consistence similaire à celui décrit en (1.3.3). C'est une implantation complète de PLC($\mathcal{FD}$) au dessus de Prolog (Sicstus) et C.

1.6.2 Le système utilisé : Meta(F)

Meta(F) est donc un résolveur dit incomplet. Ce n'est pas un inconvénient de perdre en complétude si le méchanisme permet la transmission d'informations partielles et la coordination de la résolution. De nombreuses contraintes supplémentaires ansi que différents opérateurs ont été ajoutées.

Ces principales caractéristiques sont :

Extensibilité : Meta(F) offre plusieurs niveaux de meta programmation qui permettent de définir de nouvelles contraintes.

Performances : Les schémas de meta programmation utilisés limitent le surcoût à un ordre de magnitude.

Portabilité : L'indépendance entre le moteur logique et le résolveur de contraintes assure une large portabilité.

Voici la liste des contraintes spécifiques utilisées pour exprimer les modèles de placement :

Produits non-linéaires : $\Pi_{i=1}^{n} X_i \leq \Pi_{j=1}^{m} Y_j$
La division euclidienne : $n = d.q + r,\ 0 \leq r\ <\ d$
Minimums et maximums d'un ensemble de variables.

et l'ensemble des opérateurs logique (c_1 et c_2 sont des contraintes) :

L'implication logique : $c_1 \Rightarrow c_2$
L'équivalence logique : $c_1 \Leftrightarrow c_2$
La disjonction : $c_1 \vee c_2$

La Programmation Logique avec Contraintes permet d'exploiter un ensemble de propriétés très importantes pour la formulation et la description des problèmes de placement. Les propriétés de compositionalités et de généricité permettrons d'enrichir et d'articuler la formulation du problème de placement. Néanmoins, les dimensions du problème de placement en terme de fonctionnalités à représenter (ordonnancements, communications, distributions ...) ainsi que le nombre de contraintes à résoudre ne permet pas de directement programmer le problème en Programmation Logique avec Contraintes. Nous montrons dans le chapitre suivant comment une approche basée sur les modèles enrichit la description de ces fonctionnalités afin de résoudre le problème global.

2 Méthodologie de modélisation par contraintes concurrentes

La maîtrise de la complexité d'un problème de grande échelle, en terme de résolution comme en terme de compréhension nécessite de décomposer le problème général en différentes parties. Néanmoins, pour que ces sous-problèmes soient cohérents vis à vis du problème général il faut pouvoir conserver les liens entre les différentes entités qui composent le problème. Dans cette section, nous présentons les fondements de la programmation par modèles, aussi appelé "Concurrent Constraints Model Based Computing" (CCMBC) [Jou95]. Enfin, nous expliquons comment les propriétés de cette approche permettent d'une part de procéder à la décomposition du problème général en sous problèmes et d'autres part, de considérer l'ensemble des caractéristiques d'un large problème d'optimisation combinatoire.

2.1 Modèle conceptuel

La programmation logique avec contraintes (PLC), de par ses propriétés de compositionalité, de déclarativité reposant sur des méthodes formelles, est un cadre privélégié pour développer une approche basée sur les modèles. Ces langages de PLC offrent un haut niveau d'expression grâce aux propriétés de compositionalité et de généricité qu'apporte la logique du premier ordre associée à l'algèbre de contraintes. Chaque point de vue ou fonction d'un problème peut alors être décrit par un véritable modèle mathématique construit à l'aide de contraintes et de variables permettant ainsi de décrire un espace de solution local au modèle. Cette facilité d'expression permet aussi de définir les liens entre les modèles par des contraintes. Les modèles et les liens correspondent respectivement aux entités et aux relations du modèle conceptuel.

Modèle Conceptuel = Modèles + Relations

En fin de compte, le langage de PLC permet d'exprimer la sémantique de chaque modèle et de chaque relation sans avoir à déformer la réalité. Tout modèle conceptuel s'articule donc autour de la compositionalité et de la déclarativité. Mais l'exploitation de ces deux propriétés se situe à deux niveaux. Microscopiquement, elles permettent d'établir les fondements des modèles à partir d'opérateurs de base ou de contraintes spécifiques. Macroscopiquement, elles permet de lier les entités par des relations. La modélisation globale résultante comporte alors :

Des modèles complémentaires, décrivant des sous parties distinctes du problème à considérer.

Des modèles redondant, décrivant une même partie du problème global.

Des modèles hierarchique, puisqu'un modèle peut être décomposé en sous-modèles.

2.2 L'héritage des capacités de la Programmation Logique avec Contraintes

Hélas, la résolution d'applications industrielles ne se cantonne pas à une problématique bien définie, mais intègre la combinaison de plusieurs modèles. Nous avons souvent à résoudre des problèmes d'optimisation combinatoire sur des problèmes multi-composants, multi-fonctions dans lesquels les contraintes sont très hétérogènes et où il faut, en fonction du problème à résoudre, considérer les différents éléments à différents niveaux de granularité. La PLC supporte **la coexistance de modèles multiples**.

La composition des modèles relationnels est tout simplement la conjonction logique des relations qui constituent le modèle. Ceci implique une sémantique simple de la compositionalité. L'ensemble des solutions d'un modèle composite est tout simplement l'intersection des solutions des modèles.

Outre la facilité d'expression, la possibilité de décrire un problème sous l'ensemble de ses facettes autorise une grande flexibilité au niveau de la résolution. Les variables qui composent les différents modèles peuvent être selectionnées en fonction de leur sémantique et de leur importance à une étape de résolution donnée. Cette facilité autorise la construction de procédures de recherche puissantes et permet de mieux synchroniser la résolution sur les différents modèles. Cela correspond à choisir les variables pertinentes selon l'état de la résolution courante pour guider les étapes suivantes. Si les relations entre modèles permettent de facilement décomposer le problème, d'un point de vue de la résolution, ces relations permettent la transmission d'informations partielles entre les modèles. Cet échange d'informations repose donc sur le paradigme de résolution `Ask` & `Tell` puisque ce sont des contraintes qui lient les modèles. Ceci permet à un modèle d'informer un autre modèle de son état sans avoir à attendre que l'ensemble de ces variables soient instanciées. Ainsi, le transfert d'informations partielles et la synchronisation des processus de résolution n'est pas orientée, mais déclarative et est assuré par la combinaison de contraintes. Ceci permet de réduire les domaines des variables qui composent les modèles. L'approche multi-modèle hérite donc des propriétés du langage de Programmation avec contraintes, tant sur le plan de la résolution, que sur le plan de la formulation des contraintes à satisfaire.

Une alternative pour les problèmes multi-composants, multi-fonctions : La spécification de modèles pour chaque composant et pour chaque fonction qui se combinent ensuite lors de la résolution d'un but, permet d'apporter une alternative à la résolution de problèmes de systèmes. De plus, souvent les modèles sont très hétérogènes. Certains, peuvent s'exprimer exclusivement à l'aide de contraintes linéaires, d'autres peuvent nécessiter des contraintes symboliques ou booléennes.

La PLC permettra, et ce, indépendamment de l'hétérogénéité des contraintes, par de simples interactions locales, de garantir une coordination globale du système.

Une alternative pour les problèmes d'optimisation combinatoire : La résolution de problèmes hautement combinatoires se résume rarement à une seule formalisation mathématique. Souvent, pour ne pas dire toujours, des formalisations complémentaires sont nécessaires. Un exemple pourrait être les formalisations redondantes qui tirent avantage des propriétés des solutions partielles, un autre exemple serait la prise en compte des symétries du problème, etc.

Le problème qui se pose alors est d'utiliser en même temps toutes ces informations. Dans le contexte d'approches plus classiques, comme la RO ou la programmation en nombres entiers, cette étape s'avère toujours très délicate. En effet, les programmes écrits dans un langage impératif nécessitent un temps de développement non négligeable et sont souvent difficiles à étendre et à modifier.

La PLC apparaît pour ce type de problème non seulement comme **un bon outil d'expérimentation** mais permet, au moment de la résolution, l'**utilisation concurrente de tous les modèles redondants**.

Une alternative pour un raisonnement à plusieurs niveaux de granularité : L'efficacité de la réalisation finale dépend de trois choses. Tout d'abord, elle dépend crucialement de l'efficacité du système de contraintes utilisé, du contrôle que l'on a sur la recherche d'une solution mais aussi, considérablement, de la granularité considérée dans la modélisation. L'expérience montre qu'il est indispensable de considérer différents niveaux de granularité et de pouvoir raisonner sur les différents niveaux. Ici encore, la PLC offre une alternative en permettant la description, la coexistence et la coordination de modèles à différents niveaux de granularité. Ce dernier point n'a pas encore, à notre connaissance, été vraiment exploité dans des applications. Il reste donc assez prospectif mais ouvre des horizons très intéressants.

2.3 Propriétés de la modélisation concurrente

Mais ce ne sont pas là les uniques propriétés de CCMBC. La richesse de l'information, à travers une représentation multi-modèles, apporte de nombreuses propriétés supplémentaires par rapport à la notion de langage de programmation. Cette richesse d'information est requise par rapport à la taille des problèmes à résoudre. En effet, la résolution des problèmes combinatoires d'échelle industrielle nécessite l'usage d'une telle approche, tout d'abord car les dimensions du problème sont importantes et se composent de nombreuses fonctions combinatoires hétérogènes à résoudre, il faut de plus considérer des heuristiques souvent males définies. Pourtant, pour pouvoir les résoudre, ces problèmes nécessitent d'avoir une vue aussi bien globale que détaillé. Comparée à la simple notion de langage, seule la richesse de la description qu'offre une approche CCMBC permet de comprendre et exploiter le rôle partiel de chaque variable. L'approche permet donc non seulement une meilleure compréhension du problème mais offre encore plus de liberté quand à l'utilisation des variables des modèles pour résoudre le problème global. Cette utilisation se traduit par l'échange d'informations partielles entre les modèles mise en oeuvre par une stratégie de résolution.

CCMBC = Modèle Conceptuel + Communication d'informations partielles

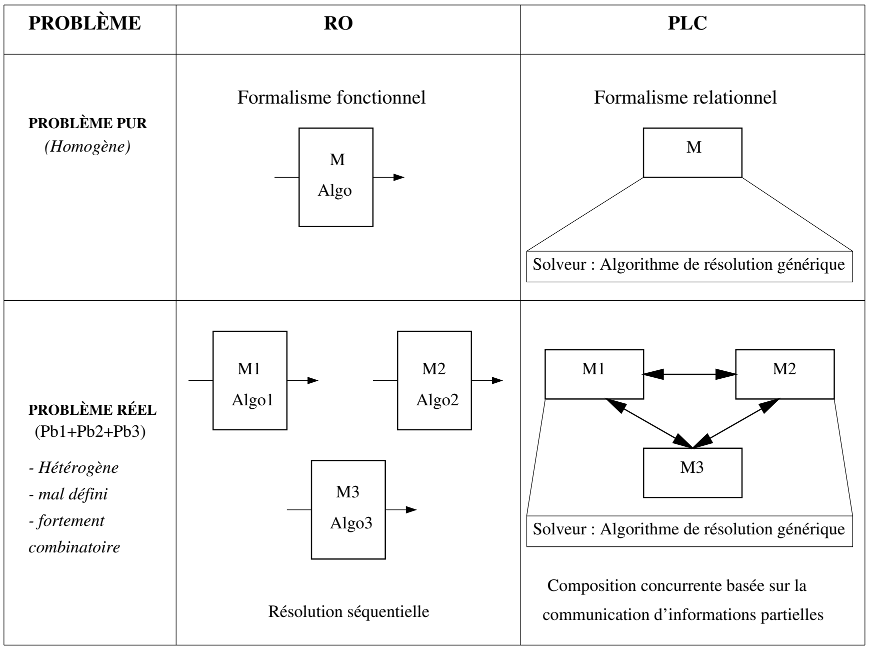
Notre approche consiste à représenter le problème réel par une modélisation formelle de ces composants en utilisant CCMBC et en exploitant l'ensemble de ses propriétés pour résoudre[Jou95].

Déclarativité : L'implantation des modèles et des relations est déclarative, la résolution est adirectionnelle. Les variables des modèles peuvent être instanciée ou non instanciées selon la requette que l'on pose au système. On peut ainsi définir plusieurs objectifs à partir d'une même modélisation.

Généricité : La généricité des modèles permet leur réutilisation dans d'autres contextes. Il suffit pour cela de redéfinir les relations du modèles avec les autres. On peut ainsi capitaliser une expérience.

Modularité : Tout modèle peut être retiré, sans modifier l'interface avec les autres modèles. Cela constitue une simplification du modèle conceptuel global. De même, de nouveaux modèles peuvent être ajoutés. Il n'est pas besoin dans les deux cas de changer les modèles existants et uniquement les relations sont à redéfinir.

Evolutivité : Les modèles peuvent être localement spécialisés ou/et réorientés selon l'évolutions de certaines fonctions du domaine applicatif. Là encore il est possible de changer uniquement les modèles affectés par les modifications.



La recherche opérationnelle (RO) et l'I.A. ont comme préocupation commune, la résolution de problème à forte combinatoire. Cependant, la RO semble plus adapter à l'étude de problèmes spécifiques, souvent simplifiés par rapport aux problèmes réels et pour lesquels il s'agit de trouver l'algorithme aussi efficace que possible. La PLC semble plus adapter à résoudre des problèmes hétérogènes par le biais d'une formalisation "fine" des différents modèles et de leurs relations.

FIG. 11.3. – *La PLC versus la R.O.*

2.3.1 Les problèmes considérés

Ces larges problèmes combinatoires comportent des affectations de ressources, des ordonnancements, des contraintes producteurs-consommateurs à satisfaire. La cardinalité des ensembles à prendre en compte est très importantes. Les applications présentent par exemple beaucoup de tâches et de ressources. Pour cela, il est n'est pas envisageable de représenter chaque élément explicitement. Il est préférable de construire un modèle représentant en intention ces ensembles d'éléments. La capacité d'exprimer et de résoudre sur ces modèles est donc fondamentale.

Granularité des modèles Si les modèles doivent être représentés en intention, on doit avoir recours à une formulation algébrique pour conserver une finesse de description. L'utilisation d'un algèbre de contraintes permet de donner de telles formulations. De façon générique, les ensembles de données $\{d_i\}$ caractérisant un modèle sont définies par une formulation logique du premier ordre A dont certains prédicat sont interprétés selon l'algèbre X :

$$\{\ d_i \mid A(d_i)\}$$

Le problème consiste alors à trouver les paramètres de A qui constituent une solution du modèle. Bien entendu, A peut comporter des quantificateurs universels qui peuvent rendre la recherche difficile. Il faut alors procéder à des approximations, ou trouver des contraintes suffisantes. On doit alors veiller à la perte de complétude de l'espace de solutions et donc à ne pas oter des solutions pertinentes. C'est là toute la difficulté de la modélisation.

Les ressources capacitives Les ressources capacitives sont un bon exemple de quantification universelle. Elles traduisent l'utilisation dans le temps d'une ressource à ne pas dépasser. Le paramètre temporel intervient donc dans la modélisation :

$$\forall t,\ f(t)\ <\ r$$

La variable r représente la ressource à ne pas dépasser, tandis que la fonction f du temps donne le nombre de ressources utilisées. La modélisation consiste à éliminer le quantificateur universel, et à procéder à des sur-approximations si nécessaire.

Les ressources disjonctives Ces ressources ne varient pas dans le temps mais traduisent un couplage entre deux ensembles d'éléments du problème. Cela peut être représenté par exemple par une bijection R entre les deux ensembles :

$$\{(d_i, d_j) \mid R(d_i, d_j)\}$$

Le problème consiste alors à trouver la bijection qui satisfait l'ensemble des contraintes. Généralement les ressources disjonctives font l'objet d'un modèle à part entière devant satisfaire l'ensemble des relations avec les autres modèles.

Les critères d'optimisation Enfin, l'optimisation combinatoire se traduit par l'optimisation de fonctions de coûts pouvant être non-linéaires. Ces fonctions de coûts peuvent porter sur toute partie des variables du modèle conceptuel. Si $\mathcal{D}$ est l'ensemble des variables du modèle conceptuel cela peut se traduire par :

$$min(cout(\mathcal{P}(\mathcal{D})))$$

où *cout* est une fonction quelquonque et $\mathcal{P}$ une partie des variables du problèmes. Ainsi, la modélisation peut être utiliser à plusieurs fins, à différents types d'optimisation combinatoire en conservant le même modèle conceptuel.

Contraintes globales L'utilisation de contraintes globales se traduit dans l'approche CCMBC par des contraintes fortement connexes avec plusieurs modèles. Ce sont donc des relations liant différentes parties de la modélisation, et définissent généralement, comme dans cette thèse, la structure du problème à résoudre.

2.3.2 Efficacité de la résolution

La résolution nécessite la transmission d'informations partielles ainsi que la coordination de la résolution, il est pour cela nécessaire d'utiliser un résolveur incomplet[VSD92b]. Cette perte de complétude se traduit par des preuves d'existence de solutions, d'optimalité. Pour maîtriser la complexité globale du problème, nous tirons partie de la richesse de description du problème et de l'utilisation d'heuristiques, le tout grâce aux opérateurs de contrôle de la Programmation Logique avec Contraintes concurentes.

Les modèles redondants Si le modèle conceptuel reflète la composition de modèles complémentaires définissant le problème général, il permet aussi de disposer de plusieurs modèles pour décrire une même fonctionnalité à résoudre. Ces modèles sont dits redondants. Pour une étape de résolution, il est donc possible de favoriser l'un ou l'autre des modèles selon l'état des différentes variables du problème[AB92][DHS90].

Les heuristiques Les heuristiques de domaines, nécessaires à la résolution de problèmes combinatoires peuvent être introduit. Bien souvent, pour un problème combinatoire aussi large que celui considéré dans cette thèse, il n'existe pas d'heuristique globale. On dispose généralement d'un ensemble d'heuristiques locales aux différentes parties du modèle conceptuel. Là encore, CCMBC permet de combiner l'utilisation des heuristiques.

Stratégies de résolution Enfin, l'ensemble de la résolution peut être coordonnée par une stratégie globale de recherche. Par la connaissance de l'état des différents modèles, ce type de stratégie est capable de combiner le contrôle des heuristiques et de coordonner la résolution sur les différents modèles [FF91][JL95].

2.4 La modélisation concurrente pour résoudre le problème du placement

La recherche réalisée par la communauté de la parallélisation automatique est principalement centrée sur la conception de nouveaux algorithmes et techniques pour la résolution séparée de chacune des fonctions nécessaires au placement d'applications sur machines parallèles. Peu de travaux ont été consacrés à essayer de traiter simultanément et de façon concurrente les quatre fonctions de partitionnement, d'alignement, de distribution et de séquencement.

Dans notre approche, chaque solution partielle doit être conforme aux solutions des autres sous-problèmes pour que l'ensemble puisse constituer une solution globale. Aussi, tant pour les applications de traitement du signal que pour la résolution du placement, l'approche est globale. La difficulté est donc de considérer un ensemble de représentations hétérogènes alors que par définition, la solution est une instantiation de ces représentations par des valeurs cohérentes. Le fait de prendre en compte la globalité du problème de placement nécessite implicitement des solutions à un niveau de granularité très fins. Aussi, le nombre de contraintes à satisfaire et de variables à considérer interdit d'aborder le problème sans méthodologie. La modélisation par contraintes concurrentes fixe le bon niveau d'abstraction reflettant de surcroît la réalité physique. Il est alors plus facile de manipuler la complexité globale et de comprendre les interactions entre les grandes fonctions du système de placement.

L'approche originale de notre méthode réside dans l'association :

- d'une modélisation par contraintes linéaires et non linéaires du placement caractérisant à la fois les contraintes applicatives (taille des tâches élémentaires spécifiques au traitement du signal systématique) mais aussi l'architecture (nb de processeurs, capacité mémoire, topologie du réseau et débit), et l'exécution (ordonnancement affine, recouvrement communications/calculs)
- de la programmation logique par contraintes qui permet d'aborder de manière concurrente la résolution de plusieurs modèles de contraintes.

Les langages de spécifications d'applications de signal, les polyèdres en nombres entiers et les applications affines permettent de modéliser précisément les fonctionnalités du placement. L'algèbre linéaire permet de construire les différents modèles au niveau de granularité requis par la complexité du problème général. Ces modèles sont issus de l'"*état de l'art*" du placement automatique et on fait ici l'objet d'une adapation au contexte applicatif du signal. Ils sont exprimés à l'aide de contraintes linéaires et non linéaires intégrable au domaine de la PLC (Programmation Logique avec Contraintes).

La PLC s'impose alors d'une part par sa capacité à traiter l'algèbres propre aux différents modèles et d'autre part par sa dimension langage qui facilite l'expression et la composition d'heuristiques et de contraintes spécifiques à un domaine, et/ou de stratégies complexes permettant le contrôle des étapes de résolutions.

Les données manipulées par le système de placement sont donc des éléments de tableaux, des macro-instructions, dont il faut nécessairement modéliser le comportement selon les fonctions à résoudre. L'ensemble des modèles que nous allons décrire sont entièrement formalisés à partir de l'algèbre linéaire, permettant le contrôle de la granularité pour chaque modèle. Chaque modèle définit une composante de la compilation d'une application de traitement du signal sur machine parallèle.

- La distribution des calculs et des données sur les processeurs, est en fait un problème de ressources dijonctives.
- L'ordonnancement des traitements en tenant compte des ressources mémoire et communications, représentées par des contraintes capacitives.

Le but est d'optimiser différents critères comme la latence de l'application ou le coût de l'architecture cible. De plus, de nombreux modèles spécifiques au problème de placement ont été développés et viennent compléter la description du problème comme les communications ou le temps physique. Compte tenu du nombre de tâches et du nombre de données à considérer lors du placment, chaque modèle est définit en intention plutôt qu'en extension. La possibilité de travailler à plusieurs niveaux de granularité s'avère fondamentale pour le problème de placement, et l'on utilise pour cela une formulation algébrique du partitionnement. Celui-ci fixe la granularité des autres modèles, il entretient donc de nombreuses relations au sein même du modèle conceptuel. De plus, les contraintes de dépendances relient à de nombreuses reprises plusieurs modèles, ce sont donc des contraintes globales et la clef de voûte du problème à résoudre. Enfin, on dispose d'heuristiques locales à un ou plusieurs modèles. Cependant, on ne connait pas d'heuristique globale. Le cœur de notre méthode utilise l'approche

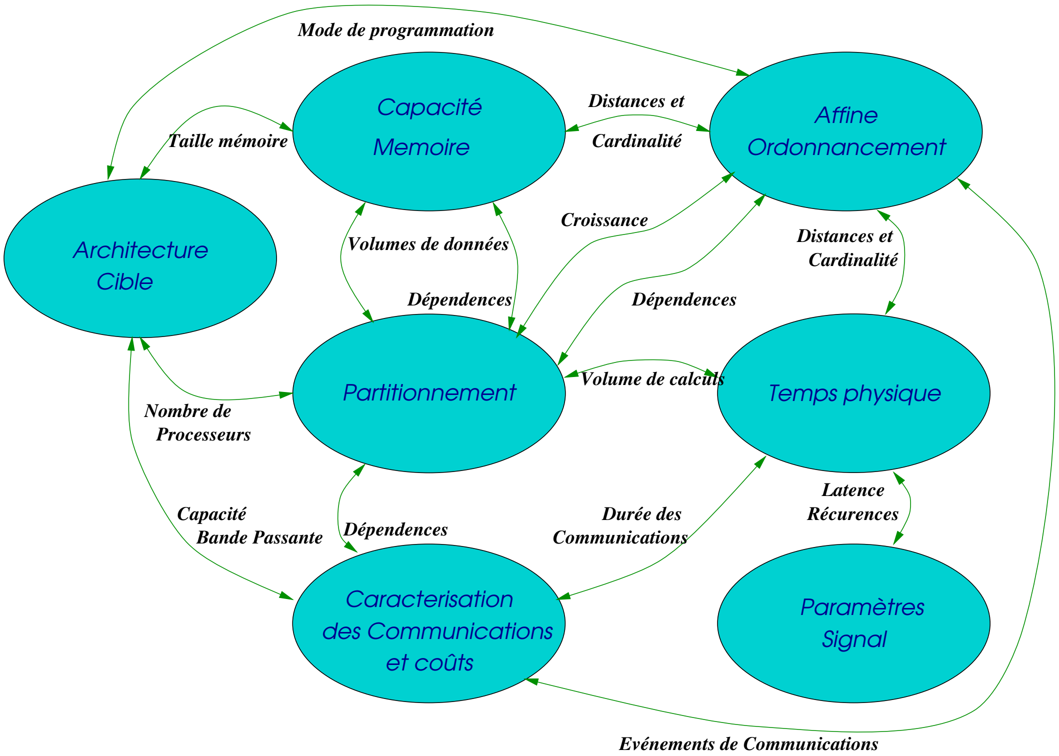


FIG. 11.4. – *Représentations en modèles concurrents relationnels du problème du placement*

multi-modèles par contraintes concurrentes qui nous a permis d'appréhender le problème du placement automatique de manière globale. Voici donc un survol (fig. 11.4) de ces modèles et de leurs relations.

Dans les chapitres précédant cette partie, nous détaillons chacun de ces modèles. Le nombre de relations et la structure de la modélisation globale ne permet pas une présentation linéaire. Dans les chapitres suivant, nous résumons l'implantation du système de placement.

Conclusion

Notre approche est une bonne alternative aux méthodes de placement existantes (cf. chap. 1) et dont les avantages majeurs sont :

Modularité de l'architecture cible : Une modélisation générique représentative d'une classe d'architectures qui peut se spécialiser par identification de certains paramètres ou de contraintes supplémentaires à chaque type de machine, permet d'avoir un outil compatible avec une grande classe d'ordinateurs parallèles.

Optimisation globale : Le système développé traite de manière globale la fonction de placement. Ceci implique donc que le critère optimisé par l'utilisateur (comme par exemple temps réel, place mémoire, coût financier...) le sera sous les seules contraintes définies initialement. Contrairement à certains systèmes existants, l'outil ne donne pas par exemple, le meilleur placement en taille mémoire pour un séquencement donné, mais retourne le meilleur placement en taille mémoire pour tous les séquencements possibles (*i.e.* séquencements qui respectent les contraintes applicatives et matérielles).

Plusieurs niveaux de granularité : La PLC offre une alternative en permettant la description, la coexistence et la coordination de modèles à différents niveaux de granularité (le degré de granularité est par exemple lié à la taille des blocs lors du découpage des données et des calculs).

Comparée aux méthodes classiques utilisées dans la résolution de problèmes complexes, l'approche choisie offre des avantages clairs incluant une plus grande **fléxibilité**, **plus d'interactions** avec l'utilisateur, sans pour autant sacrifier **l'efficacité**. De plus, cette technique offre des solutions plus génériques et plus robustes qui ont un impact important sur le **coût d'évolution et de mise à niveau** du logiciel.

Toutefois, il est à noter que plus problème est complexe, comme c'est le cas pour le placement d'applications sur machine parallèle, plus une collaboration entre spécialistes du domaine applicatif et spécialistes des techniques de programmation par contraines est nécessaire. En effet, le succès d'une telle approche réside dans le degré de modélisation des différents composants du problème et de leur spécialisation pour être compatible avec le cadre arithmétique de la PLC.

La modélisation (décrite dans les chapitres précédants) comporte donc tout les problèmes que CCMBC permet d'aborder. En particulier, comme nous allons le montrer dans la partie suivante, on aboutit dans de nombreux cas à des contraintes complexes et non-linéaires qui sont soit des liens, soit des éléments des modèles. Résoudre ces contraintes non-linéaires dans les entiers en aussi importante quantité est un pari considérable pour l'optimisation combinatoire. Pour cela, seule l'approche CCMBC permet de combiner recherche et résolution **au niveau d'abstraction du modèle** tout en contrôlant la complexité globale. De plus, ces modèles résultant d'une description de la réalité, l'approche permet de mieux maîtriser la compréhension et la maintenance du système automatique.

La méthodologie décrite ci-dessus trouve dans le placement automatique un domaine d'investigation sans précédent.

Chapitre 12

Système de placement de traitement de signal sur machine parallèle de type SIMD/SPMD

1 Introduction

L'implantation exploite les propriétés de l'approche concurrente (définie au chap. 11). Les modèles de cette partie peuvent donc être spécifiés, implantés pour résoudre le problème du placement automatiquement. La spécification est directement issue de la modélisation mathématique. Si certaines contraintes peuvent être exprimées directement, plusieurs d'entre elles ont nécessité parfois d'être reformulées afin d'être implantées dans un langage de programmation avec contraintes. Malgré cela, et grâce aux propriétés de l'approche concurrente, la réalisation reflète néanmoins la modélisation mathématique. Aussi, les modèles implantés sont des composants informatiques tandis que les relations des connexions entre ces composants. Ceci est particulièrement adapté à la modularité du problème de placement. Cependant, résoudre efficacement le problème de placement global sur des exemples de taille réelle nécessite l'introduction de stratégies et d'heuristiques. Celles-ci sont encore une fois, grâce à l'approche de modélisation concurrente, des composantes informatiques.

La réalisation est **un système automatique de placement d'algorithmes de traitement de signal admettant le dimensionnement et la configuration de machines parallèles de type SPMD/SIMD**. En outre ce système d'aide au placement libère le programmeur des tâches ingrates comme la génération de solutions multiples, la prise en compte des ressources de la machine, le respect des contraintes temps réel et d'architecture, la visualisation de l'ordonnancement et de la distribution des calculs. De plus, il permet à l'expert d'analyser un plus grand nombre de solutions en étudiant, par simple interaction avec le système, l'influence de ses choix d'optimisation sur une solution finale. De cette façon, le système peut être la composante d'un "atelier de développement" qui fournit rapidement à l'ingénieur des informations dont il a besoin pour prendre des décisions relatives à la conception de la machine cible ou des algorithmes, aux différents coûts, et au temps réel.

C'est donc une nouvelle méthode de placement, proche du **prototypage virtuel** que nous décrivons dans ce chapitre. Le système fournit à l'ingénieur des moyens d'analyse relatifs à la conception de la machine cible (modularité et codesign) sur laquelle une application sera placée ainsi qu'aux différents coûts liés au placement (optimisation globale). Il fournit alors aux ingénieurs des informations dont ils ont besoin pour prendre des décisions, à un moment où les modifications sont encore faciles à apporter et peu coûteuses (phase de conception de l'architecture par exemple).

Nous donnons en (§ 2) la structure de l'architecture du système de placement. En section (§ 3), nous donnons l'ensemble des fonctions que le système peut ou pourrait résoudre. Enfin, en (§ 4), nous donnons l'ensemble des spécifications qui ont permies l'implantation.

2 Architecture et composants du système de placement

Le système résoud le problème **du placement automatique** d'applications traitement de signal sur machines parallèles. A partir d'une description formelle de l'application, une solution consiste en un ensemble de directives de placement à partir desquelles il est facile de générer du code. Chaque modèle peut alors hériter des propriétés de la résolution concurrente. Aussi l'architecture informatique se déduit-elle de la formalisation mathématique du problème du placement dont nous rappelons le résultat.

2.1 Présentation des composants

L'architecture du système de placement et ses propriétés sont importantes tant d'un point de vue résolution que d'exploitation de l'information présente. On distingue cinq types de composants.

2.1.1 Les modèles

A chaque modèle on associe un composant logiciel. Celui-ci doit pouvoir être rafiné ou particularisé. Il est par exemple possible de spécialiser des ordonnancements et/ou partitionnements en fonction du type de code à générer. Il suffit simplement d'ajouter les contraintes caractérisant les solutions souhaitées au différents modèles. Chaque composant logiciel du système doit pouvoir être désactivé ou échangé avec un nouveau composant garantissant ainsi l'évolutivité et la maintenabilité du système de placement.

2.1.2 Les relations

Chaque relation peut être vu comme un composant informatique particulier puisqu'il lie la consistence de deux modèles. Chacune des relations peut être modifiée ou supprimée. Elles permettent non-seulement la résolution du problème global mais représentent en partie la sémantique du placement global pour les algorithmes de traitement de signal.

2.1.3 Stratégies de résolution

Grâce à l'approche de modélisation concurrente, les fonctions de coût et stratégies de résolution sont indépendantes de la modélisation et constituent des composants logiciels à part entière. La fonction de coût n'est en général pas distinguée de la stratégie car celle-ci est le résultat d'une spécialisation. Par exemple, une des stratégies la mieux adaptée consiste à énumérer les variables selon la topologie du graphe de flot de données (chap. 13). En revanche, les procédures de résolution efficaces utilisent des heuristiques qui sont encore d'autres composants.

2.1.4 Les heuristiques

Les différents heuristiques sont définis indépendament de tout autre autre composant. Ils peuvent porter sur un ou plusieurs modèles, et sont exploités au niveau de la résolution. Ceux-ci sont issus de constations sur la modélisation et sont formulés mathématiquement. L'heuristique du plus court chemin en est un bon exemple (chap. 13).

2.1.5 Contraintes élémentaires

Les contraintes élémentaires portent sur l'algèbre linèaire. Le produit scalaire a par exemple été définit en tant que contrainte élémentaire. Afin d'optimiser au mieux la résolution du problème de placement, il existe deux formes de ce produit scalaire :

- Les deux vecteurs du produit scalaires sont inconnus.

  Le système de programmation avec contraintes utilise alors la multiplication comme opérateur. La contrainte résultante est donc non-linéaire.
- Un des vecteurs est connu. Il s'agit en fait là d'une contrainte linéaire.

Il existe bien d'autres contraintes élémentaires comme le produit des composantes d'un vecteur ou l'addition vectorielle. L'ensemble de ces contraintes résultant de l'algèbre linéaire réside dans une librairie particulière auquelle l'ensemble des composants logiciels accède.

2.2 Résumé des composants issus des modèles et relations

Les deux tableaux suivants résument les résultats décrits dans la partie III.

Modèle	**Interprétation**	**Contraintes**
PARTITIONNEMENT	Dimension (temps,processeur,calcul) = $(\mathbf{c}, \mathbf{p}, \mathbf{l})$ Dimensions des itérations Partitionnement uniforme parallèle aux axes Il y a au moins 1 processeur Il y a au moins 1 une itération en mémoire Non redondance des calculs dans le temps Non redondance des calculs sur les processeurs	$\mathbf{i} = L.P.\mathbf{c} + L.\mathbf{p} + \mathbf{l}$. $\forall \mathbf{i},\ \mathbf{0} \leq \mathbf{i} < up(\mathbf{i})$ P et L diagonales $\det(P) \neq 0$ $\det(L) \neq 0$ $\forall \mathbf{p},\ \mathbf{0} \leq \mathbf{p} < P.\mathbf{1}$ $\forall \mathbf{l},\ \mathbf{0} \leq \mathbf{l} < L.\mathbf{1}$
ORDONNANCEMENT	Deux blocs issus de deux nids de boucles distincts ne s'exécutent pas en même temps. Modèle SIMD ou SPMD implicite. Deux blocs issus d'un même nid de boucle ne s'exécutent pas en même temps	$N =$ Nbr. de T.E., $\forall k = 1, ..., N$ $d^k = N(\alpha^k \mathbf{c}^k + \beta^k) + k$ $\alpha_n^k \geq 1$ et $\alpha_i^k \geq \alpha_{i+1}^k up(c_{i+1}^k)$
DÉPENDANCES	Une donnée ne peut être lue avant d'avoir été écrite $D(\mathbf{i}^w, \mathbf{i}^r) \Rightarrow d^w(\mathbf{c}^w) + 1 \leq d^r(\mathbf{c}^r)$	$\forall(\mathbf{i}^w, \mathbf{i}^r)$ sommets du polytope engendré par $D(\mathbf{i}^w, \mathbf{i}^r)$ on a, $d^w(\mathbf{c}^w) + 1 \leq d^r(\mathbf{c}^r)$

Modèle	**Interprétation**	**Contraintes**
MÉMOIRE	Approximation de la capacité mémoire	$2 \max_{k=1..N} (VolEcr^k) + \sum_{k=1..N} (VolLec^k.NbBlocLec^k)$
MODÈLE SIGNAL	Calcul du nombre d'événements apparus avant $\mathbf{c}$ Distance maximale événementielle (dans une chaîne) coût calculs-communications Sur-approximation de la latence physique Sur-approximation de l'acquistion physique	$nbc(0, d(\mathbf{c})) = \mathrm{Card}\{\mathbf{c}' \mid \mathbf{c}' \ll \mathbf{c}\} = \sum_i c_i \left(\prod_{j>i} up(c_j)\right)$ $\Lambda = \max(d^N(\mathbf{c}^N) - d^1(\mathbf{c}^1))$ $ct(k) = cout^k_{elem} + (\det(L) - 1).max(cout(V^k_{te}), cout^k_{elem}) + cout(V^k_{te})$ $\sum_{k=1..N} nbc(0, \Lambda).ct(k) \leq Latence.$ $tp(k) = \sum_{k=1..N} nbc(0, \alpha^1_\infty).ct(k)$
INTERPHASE	Linéarisation des processeurs Nombre de recepteurs Volumes élémentaires Coût de communications	$\mathcal{L}(p_0, \ldots, p_{n-1}) = \sum_{j=0}^{n-1} (\prod_{k=0}^{j-1} P_{k,k}).p_j,$ $\forall\ j,\ 0 \leq j < n,\ 0 \leq p_j < P_{jj}$ $\forall c^w,\ \mathcal{V}^w_{emis}(\mathbf{c}^w) = \sum_{\forall c^r \in \mathcal{D}(c^r, c^w)} \mathcal{V}^{r,w}_{elem}(\mathbf{c}^r, \mathbf{c}^w)\ \forall(\mathbf{c}^r,\ \mathbf{c}^w),\ \mathcal{V}^{r,w}_{elem}(\mathbf{c}^r,\ \mathbf{c}^w) =$ $\sum_{\{(\mathbf{p}^r, \mathbf{p}^w) \mid \mathcal{L}^r(\mathbf{p}^r) \neq \mathcal{L}^w(\mathbf{p}^w)\}} \mathrm{card}\ \left(\ \mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r, \mathbf{p}^r)}) \cap \mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w, \mathbf{p}^w)})\right)$ $cout(\mathcal{V})\ =\ b.\mathcal{V}\ +\ a$
MACHINE	Nombre de processeurs Type SIMD Type SPMD	$\det(P^k)$ $up(\mathbf{c}^k) = P^{-1}_{kk} L^{-1}_{kk} up(i_k)$ $up(\mathbf{c}^k) = \lceil P^{-1}_{kk} L^{-1}_{kk} up(i_k) \rceil$

Ces modèles mathématiques résument le problème du placement. On ne dispose pas ici de l'ensemble des éléments nécessaires à la réalisation. En effet, d'une part différents modèles ne sont pas directement implantable dans un langage de programmation par contraintes comme le modèle des dépendances ou celui de l'interphase. Il faut donc dans la plupart des cas soit adapter le modèle en procédant à des surapproximations, soit revoir la modélisation. D'autre part, ces modèles mathématiques ne constituent pas la résolution d'une fonction de coût, ou bien même le simple calcul d'une solution, puisque la résolution est incomplète. Pour compléter celle-ci, il est nécessaire de construire une stratégie de résolution explorant un espace de recherche spécialisé selon une fonction de coût. Pour que l'on puisse envisager celle-ci il est nécessaire, au préalable de décrire les fonctionnalités du système de placement.

3 Fonctionnalité du système de placement

Le système automatisé de placement admet le prototypage de diverses facettes du couple architecture/applications, en tenant compte des contraintes du signal. Certaines de ces fonctionnalités ont été implantées, d'autres pourraient l'être.

3.1 Recherche automatique de solutions

Le système doit traiter de manière globale la fonction de placement. Cela consiste à trouver une solution au problème formulé par l'utilisateur. Lorsque le système est capable de calculer une solution, on peut envisager de programmer l'optimisation de fonctions de coût, comme une ressource d'architecture ou un paramètre du signal.

3.1.1 Composition d'une solution

Une solution se compose de l'instantiation de l'ensemble des paramètres des modèles par des valeurs satisfiant l'ensemble des contraintes. Un ensemble minimal de ces paramètres définissant totalement une solution se compose de toutes les variables sous-jacentes à l'ordonnancement $\left\{\boldsymbol{\alpha}^k, \beta^k\right\}$ et de toutes les matrices de partitionnement $\left(P^k, L^k\right)$ associées aux nids de boucles.

3.1.2 Optimisation automatique

Ceci implique donc que le critère optimisé par l'utilisateur (temps réel, place mémoire, coût financier...) le sera sous les seules contraintes définies initialement, qui peuvent bien sûr être différentes du critère à optimiser. Il est par exemple possible de minimiser la mémoire avec des contraintes de latence fortes ou vice-versa. Contrairement à certains systèmes existants, le système ne donne pas par exemple, le meilleur placement en taille mémoire pour un séquencement donné, mais retourne le meilleur placement en taille mémoire pour tous les séquencements possibles (*i.e.* séquencements qui respectent les contraintes applicatives et matérielles).

Voici les critères qui ont été implantés :

- **Latence** : le système permet de trouver le (ou les) placement(s) qui minimise(nt) le temps d'exécution de l'application sur une machine cible prédéfinie par l'utilisateur.
- **Mémoire** : le système permet de trouver le (ou les) placement(s) qui minimise(nt) la mémoire consommée par l'application sur une machine cible prédéfinie par l'utilisateur.

- **Efficacité** : le système permet de maximiser le parallélisme du (ou des) placement(s) de l'application sur une machine cible prédéfinie par l'utilisateur.

Voici quelques exemples de critères pouvant être implantés :

- **Dimension machine** : le système admettrait de configurer une machine minimale pour une application donnée. L'utilisateur choisirait par exemple, de configuer une machine avec un nombre minimal de processeurs.
- **Coût** : En intégrant un coût (financier par exemple) sur chacun des composants matériels, le système trouverait le (ou les) placement(s) de l'application qui minimise(nt) ce coût.
- **Temps d'occupation de la machine** : le système fournirait le (ou les) placement(s) qui minimise(nt) le temps d'occupation de la machine cible prédéfinie par l'utilisateur afin de pouvoir éventuellement placer une seconde application.
- **Récurrence d'entrée ou de sortie** : le système donnerait le (ou les) placement(s) qui minimise(nt) la cadence d'entrée et/ou de sortie des résultats produits par un ensemble de traitements. Ces contraintes sont souvent imposées dans le cadre des signaux vidéo, acoustiques et/ou hyperfréquences.

3.2 Paramètres d'optimisation

Plusieurs paramètres sont critiques pour le développement des applications de traitement du signal, dont certains font l'objet de fonctions de coût. D'autres font simplement l'objet de contraintes à satisfaire. On peut alors composer les contraintes sur ces paramètres : optimiser l'efficacité avec des contraintes de mémoire et d'acquisition restreintes. Ces paramètres induisent alors des réactions au niveau de l'optimisation qui sont différentes de celles obtenues si l'on optimise un critère sans autres contraintes. Nous aurons l'occasion de revenir sur cet aspect dans le chapitre sur les expérimentations (V).

3.2.1 Dimensionner la machine

Une modélisation générique représentative d'une classe d'architectures que l'on peut particulariser par identification de certains paramètres ou de contraintes supplémentaires pour chaque type de machine, permet d'avoir un système compatible avec une grande classe d'ordinateurs parallèles. Le système permet non seulement de concevoir une ou plusieurs composantes de l'architecture cible en donnant leurs dimensions et configurations adéquates. Ceci couvre donc la mémoire, la bande passante, les processeurs en respectant les ressources maximales disponibles. Le système permet de spécifier les ressources nécessaires pour placer une application particulière sur un type de machine donnée sans violer les contraintes applicatives. Cela conciste à prendre en compte les dimensions et le nombre de chaque composant matériel (Hardware). Les composants matériels considérés sont ceux du modèle de machine définit en (chap. 9) :

- Nombre et puissance des processeurs.
- Performance du réseau d'interconnections.
- Taille de mémoire.

3.2.2 Les paramètres du signal

La latence, la récurrence d'entrée pour une machine et une application données peuvent être contraint par l'utilisateur. On peut par exemple encadrer le paramètre d'acquisition par des bornes maximales et minimales, pratique courante en traitement du signal radar.

3.3 Manipulation de solutions partielles

Le programmeur saisit une solution partielle de placement et le système poursuit sa recherche en complétant la solution de placement.
Par exemple :

- l'utilisateur a configuré sa machine avec un nombre insuffisant de processeurs pour le type de placement qu'il désire. Le système permet de trouver le nombre minimal de processeurs nécessaires au placement parmis l'ensemble des processeurs disponibles.
- l'utilisateur peut imposer un séquencement des calculs. Le système va alors trouver les partitionnements (*i.e.* distributions des données et des calculs en mémoire et sur les processeurs) adéquates.
- de même, l'utilisateur peut imposer un partitionnement initial, le système va trouver les ordonnancements compatibles.

L'utilisateur peut aussi exprimer une borne minimale ou maximale pour les paramètres d'architecture. Il peut alors valider les compromis entre les paramètres sensibles du placement en phase de conception de l'application ou de l'architecture, compris entre :

- Nombre de processeurs/Bande passante.
- Mémoire/temps réel.
- Nombre de processeurs/temps réel.
- Bande passante/temps réel.
- Mémoire/Bande passante.

Manipuler les solutions partielles s'est révélé être d'un grand interêt. Cela permet d'identifier les insufisances au niveau de la modélisation. Ou bien lorsqu'il s'agit de concevoir une stratégie d'optimisation, cela permet de simuler la recherche de solutions locales à un modèle.

3.4 Heuristiques

La présence d'un ensemble d'heuristiques permet de soulager la recherche d'une solution, en l'orientant.

- Ordonnancement des calculs au plus tôt, ou au plus tard.
- Maximiser le parallélisme.
- Minimiser les communications.
- Recouvrir les communications par les calculs.
- Maximiser la localité des données.
- Aligner les données pour éviter les communications.

Nous décrirons ces heuristiques au chapitre 13.

3.5 Les stratégies de résolution

Les stratégies de résolution permettent le contrôle de la recherche. Certaines sont statiques, d'autres dynamiques, elles dépendent fortement du critère à optimiser et sont précisées au chapitre 13.

4 Spécification et Implantation des modèles et relations

Cette section est consacrée à l'implantation des différents modèles qui composent l'approche de placement global. Cette implantation reflète naturellement la structure abstraite de la modélisation. Ceci confère aux différents composantes informatiques des propriétés de modularité, de maintenabilité et d'évolutivité. Nous donnons dans cette section la structure générale du système de placement automatique, suivi de la spécification des modèles et relations. Certains modèles posent des problèmes d'implantation liés à la nature des contraintes. Nous étudions spécifiquement les problèmes rencontrés.

4.1 Structure générale

Voici la structure générale de la réalisation du système de placement (fig. 12.1). Au niveau de l'implantation, les relations sont associées aux modèles, les variables permettant de construire ces relations sont donc les arguments des modèles. Il s'agit là d'une implantation parmis d'autres possibles, du reste proche de la réalité physique du problème.

Dans les grandes lignes, le modèle de partitionnement définit les listes de matrices `LMatP, LMatL` (resp. $\left\{P^k\right\}_{k=1}^n, \left\{L^k\right\}_{k=1}^n$), les vecteurs `LVecUpC` (en fait $\left\{\mathbf{up(c)^k}\right\}_{k=1}^n$) et prend en compte le type d'architecture `ArchType`).

Le modèle machine construit la contrainte sur le nombre de processeurs et définit la variable mémoire.

L'analyse des dépendances construit le polytope de dépendances `LDep`, dont la structure est une liste de sommets par couple de nid de boucles dépendant, qui sera utilisé par le modèle d'ordonnancement. Ceci permet, avec le partitionnement de poser les contraintes de précédences et de distances (variables `LDist`, i.e. $\left\{d^k_{max}\right\}_{k=1}^n$). Ce modèle engendre naturellement les fonctions d'ordonnancement `LDFonc` (c'est à dire $\left\{(\boldsymbol{\alpha}^{\mathbf{k}}, \beta^k)\right\}_{k=1}^n$). C'est les modèles de mémoire et de signal qui construisent alors les contraintes capacitives basées sur ces distances.

Le modèle mémoire définit le nombre de blocs maximal alloués en lecture `LNBlocs`. Il définit de même les volumes de données écrits et lus `LVolEcr, LVolLec`.

Le modèle de signal utilisant les communications définies par le modèle d'interphase, établit les contraintes de latence `Latence` et temps physique `InPeriod` ainsi que la distance maximale du plus long chemin `DistMax`.

Le prédicat `latence` est la procédure de résolution sur la variable `Latence` qui permet d'optimiser la latence de l'application.

Voici donc la spécification des modèles, elle établit le lien entre la modélisation mathématique et l'implantation. Les variables n, k, w, r représentent respectivement le nombre de nids de boucles, un nid de boucles quelquonque, un nid de boucles en écriture, un nid de boucles en lecture.

```
placement(AppName,LMatP,LMatL,LVecUpC,LDFonc,LDistMax,MaxMem,Latence,InPeriod,ArchType):-
        lecture_spec_application(AppName,_,_,_,_,_),
        partitionne(LMatP,LMatL,LVecUpC,ArchType),
        machine(ArchType,LMatP,LMatL,MaxMem,NPe),
        analyse_dependances(LDep,LMatP,LMatL,LVecUpC),
        ordonne(LVecUpC,LMatP,LMatL,LDep,LDFonc,LDistMax),
        memoire(LMatP,LMatL,LVecUpC,LVolLec,LVolEcr,LDistMax,LNBlocs,MaxMem),
        interphase(LCom,LMatP,LMatL,LVecUpC),
        signal_tr(LDFonc,LMatL,LMatP,LVecUpC,LDistMax,LCom,DistMax,Latence,InPeriod,ArchType),
        minimise(
                (latence(LVecUpC,LMatP,LMatL,LDFonc,LDistMax),LVProd,min_lat)), Latence)

        ).
```

FIG. 12.1. – *Structure générale*

4.2 Partitionnement

Les contraintes et variables permettant de construire le partitionnement sont des listes de matrices de partitionnements. Il en résulte aussi les listes de vecteurs $\{up(\mathbf{c}^k)^k\}_{k=1}^n$ (chap. 4). La complexité des contraintes de ce modèle est réduite, principalement car les matrices de partionnement sont diagonales. Chaque ligne de la matrice faisant donc l'objet d'une contrainte.

Partitionnement			
Descriptif	*Variables*	*Contrainte*	*Dimensions*
Parallélisme	$\{L^k\}_{k=1}^n$	$\mathbf{0} \leq \mathbf{p}_l^k < P_{ll}^k.\mathbf{1}$	$\sum_{k=1}^n dim(\mathcal{I}^k)$
Localité	$\{P^k\}_{k=1}^n$	$\mathbf{0} \leq \mathbf{l}_l^k < L_{ll}^k.\mathbf{1}$	idem
Type SIMD	$\{up(\mathbf{c}^k)\}_{k=1}^n$	$up(\mathbf{c}^k) = P_{ll}^{-1} L_{ll}^{-1} up(i_l)$	idem
Type SPMD	$\{up(\mathbf{c}^k)^k\}_{k=1}^n$	$up(\mathbf{c}^k) = \lceil P_{kk}^{-1} L_{kk}^{-1} up(i_k) \rceil$	idem

4.3 Ordonnancement affine

Ce modèle définit la fonction d'ordonnancement, la contrainte de croissance étant un lien avec le partitionnement. La distance de dépendance résulte d'une contrainte permettant la simplification de l'implantation de plusieurs modèle basés sur le lien de dépendance. Les premières réalisations du système de placement ne faisant pas figurer cette distance (chap. 7).

Ordonnancement affine			
Descriptif	*Variables*	*Contrainte*	*Dimensions*
Distance de dépendance max. par couple	$\{d_{max}^{w,r}\}$		
Ordonnancement	$\{(\boldsymbol{\alpha}^k, \beta^k)\}_{k=1}^n$	$\alpha_n^k \geq 1$	n
Type SPMD/SIMD			

4.4 Mémoire

La structure principale du modèle mémoire sont les nombre de blocs alloués dans la distance maximale de dépendances, ainsi que les volumes de références en écritures et lectures (chap. 8). Le modèle mémoire était initialement basé sur le plus petit α du nid de boucle pour sur-approximer le nombre de blocs. Cette contrainte introduisait un biais important dans la sélection des solutions pertinentes. La contrainte de distance associée à la contrainte de cardinalité ont permis de simplifier la modélisation, d'obtenir de meilleurs solutions et de rendre l'implantation plus simple.

Mémoire			
Descriptif	*Variables*	*Contrainte*	*Dimensions*
Capacité	$\{VolEcr^k, VolLec^k\}_{k=1}^n$		
Nombre de blocs alloués en lecture	$\{NbBlocLec^k\}_{k=1}^n$		

4.5 Temps physique

Les variables temps réel sont la latence et l'acquisition définies toutes à partir de la contrainte de cardinalité. Les distances utilisées sont celles du plus long chemin sur le graphe de dépendances ainsi que la période du premier nid de boucles (créant ainsi une nouvelle relation avec le partitionnement, cf. chap. 10). Le plus long chemin, qui aurait pus être irréaliste par le nombre est en fait une contrainte maximale par couple de nids de boucles dépendants.

Signal, Temps physique			
Descriptif	*Variables*	*Contrainte*	*Dimensions*
Plus long chemin	Λ		
Latence	$\{ct(k)\}_{k=1}^{n}$	$\sum_{k=1..N} nbc(0,\Lambda).ct(k) \leq Latence$	1
Acquisition	$\{tp(k)\}_{k=1}^{n}$		
Récurence	*Recurence*		
Latence	*Latence*		

4.6 Interphase

Les contraintes sous-jacentes au modèle d'Interphase font l'objet dans cette section d'une simplification. La complexité de la modélisation comme les difficultés d'implantation nécessitent ici de construire des surapproximations. Nous rappelons (cf. chap. 6, § 2.4, p. 118, équations 6.7 et 6.8, § 2.5.1, p. 119, équation 6.9) les contraintes à considérer :

$$\forall(\mathbf{c}^r,\ \mathbf{c}^w),\ \ \mathcal{V}^{r,w}_{elem}(\mathbf{c}^r,\ \mathbf{c}^w) = \sum_{\{(\mathbf{p}^r,\mathbf{p}^w)|\mathcal{L}^r(\mathbf{p}^r)\ \neq\ \mathcal{L}^w(\mathbf{p}^w)\}} \text{card}\ \Big(\mathcal{R}^r(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \cap \mathcal{R}^w(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)})\Big)$$

$$\forall c^w,\ \mathcal{V}^w_{emis}(\mathbf{c}^w)\ =\ \sum_{\forall c^r \in \mathcal{D}(c^r,c^w)} \mathcal{V}^{r,w}_{elem}(\mathbf{c}^r,\mathbf{c}^w)$$

Les structures de données liées à l'interphase permettent de construire une sur-approximation du coût des communications (chap. 6). La première simplification consiste en une représentation encore plus simple des régions référencées que celles couramment utilisées dans la thèse (voir chap. 2, § 3.1.5, p. 49) dont voici l'expression :

$$\mathcal{R}(\mathbf{i}) = \{(\varphi_1,\ldots,\varphi_\lambda,\ldots,\varphi_n) \mid \forall\lambda, \exists(\kappa,\kappa')\ \ I_{\lambda,\kappa}.i_\kappa < \varphi_\lambda \leq I_{\lambda,\kappa}.i_\kappa + M_{\lambda,\kappa'}.up(m_{\kappa'})\} \tag{12.1}$$

où le tableau considéré possède n dimensions. Cette définition correspond à une surapproximation convexe des ensembles d'éléments de tableaux référencés. Elle est obtenue par élimination des quantificateurs existentiels portant sur $\mathbf{m}_\kappa$ (cf. chap 6, p. 112 les formules référencées).

L'expression (12.1) est bien une simplification de (6.12). En effet, montrons que (6.12) $\Rightarrow$ (12.1).

$$\begin{array}{rcl} (6.12)\ :\ (\varphi_1,\ldots,\varphi_\lambda,\ldots,\varphi_n) \in \mathcal{R}(\mathbf{i}) & \Leftrightarrow & \forall\lambda\ \ \exists(\kappa,\kappa'),\ \exists m_{\kappa'},\ \varphi_\lambda\ =\ I_{\lambda,\kappa}.i_\kappa\ +\ M_{\lambda,\kappa'}.m_{\kappa'} \\ \text{or } 0 \leq m_{\kappa'} < up(m_{\kappa'}) & \Rightarrow & I_{\lambda,\kappa}.i_\kappa \leq \varphi_\lambda < I_{\lambda,\kappa}.i_\kappa + M_{\lambda,\kappa'}.up(m_{\kappa'}) \text{ d'où (12.1)} \quad \square \end{array}$$

Il est possible d'instancier les formules définies précédemment sur cette définition de région soit en Lecture, soit en Ecriture. A l'instar de l'ensemble définit en (6.15), la région référencée par une partition s'écrit alors :

$$\mathcal{R}(\mathcal{P}_{(c,p)}) = \left\{ \begin{array}{l} (\varphi_1, \ldots, \varphi_\lambda, \ldots, \varphi_n) \mid \ \forall\lambda\ \exists(\kappa, \kappa') \\ I_{\lambda,\kappa}(P_\kappa L_\kappa c_\kappa + L_\kappa p_\kappa) \leq \varphi_\lambda \leq I_{\lambda,\kappa}(P_\kappa L_\kappa c_\kappa + L_\kappa(p_\kappa + 1) - 1) + M_{\lambda,\kappa'} up(m_{\kappa'}) \end{array} \right\} \quad (12.2)$$

Les formules effectives des volumes de communications (chap. 6, équations 6.5, p. 117, 6.7, p. 118, 6.9, p. 120), même avec cette représentation de région (12.2) est difficilement implantable en Programmation Logique sous Contrainte. En effet, il n'est pas réaliste de construire une contrainte pour chaque instantiation des variables liées aux $\forall$ et il faut de surcroît tenir compte de la diséquation (6.8). Il s'agit donc d'établir une sur-approximation du volume de communications autorisant une implantation. D'autre part, les modèles de communication présentent des volumes de données non uniformes, que ce soit entre processeurs ou entre partitions temporelles, ce qu'illustrent les figures (fig. 6.2, p. 118, fig. 6.4, p. 120 et fig. 6.5 p. 122). Ceci est due à l'inéquation (6.8, p. 118) et aux quantificateurs universels portant sur les partitions ($\forall(\mathbf{c}^r, \mathbf{c}^w)$ par exemple dans la formule 6.5, p. 118). Tenir compte de cette non-uniformité au niveau des communications constitue un véritable problème car cela s'étend au modèle de temps réel, où la non uniformité des volumes de communications entrainent une non-uniformité des recouvrements calculs/comms. On cherche à établir dans cette section une sur-approximation uniforme du volume de communications, en fait un volume de communications “au pire des cas” ne dépendant pas des vecteurs **c** et **p**, mais uniquement des matrices de partitionnement et d'accès.

La sur-approximation des communications s'établie sur deux notions. Il faut, dans un premier temps, définir l'intersection maximisant le volume. En tenant compte de la proriété (3.1), on ramène le problème à l'intersection d'ensembles de coordonnées sur des dimensions de tableaux, et on obtient une majoration de $\mathcal{V}^{r,w}_{elem}$:

$$\mathcal{V}^{r,w}_{elem}(\mathbf{c}^w, \mathbf{c}^r) \ \leq \ \prod_\lambda (max_{(\mathbf{p}^r,\mathbf{p}^w)} \Big(\text{ card } \Big(\ \mathcal{R}^r_\lambda(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \bigcap \mathcal{R}^w_\lambda(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)}) \ \Big) \ \Big)$$

Ceci consiste à avoir une sur-approximation de (6.5). En remplaçant les ensembles de références par leur définition (12.2), on a :

$$\begin{array}{c} \mathcal{R}^r_\lambda(\mathcal{P}^r_{(\mathbf{c}^r,\mathbf{p}^r)}) \bigcap \mathcal{R}^w_\lambda(\mathcal{P}^w_{(\mathbf{c}^w,\mathbf{p}^w)}) \ \subset \\ \Big(\{\varphi \mid \ \exists\kappa, \ I^w_{\lambda,\kappa} P^w_\kappa L^w_\kappa c^w_\kappa \ \leq \ \varphi_\lambda \ < \ I^w_{\lambda,\kappa}(P^w_\kappa L_\kappa c^w_\kappa + L^w_\kappa - 1)\}\Big) \ \bigcap \\ \Big(\{\varphi \mid \ \exists\kappa'', \ I^r_{\lambda,\kappa''} P^r_{\kappa''} L^r_{\kappa''} c^r_{\kappa''} \ \leq \ \varphi_\lambda \ < \ I^r_{\lambda,\kappa''}(P^r_{\kappa''} L_{\kappa''} c^r_{\kappa''} + L^r_{\kappa''} - 1) + M_{\lambda,\kappa'''}.up(m)_{\kappa'''}\}\Big) \end{array}$$

Ceci permet de borner le volume $V^{r,w}_{elem}$ par la cardinalité de l'intersection des deux ensembles. Cela revient à borner par la valeur $\mu^{w,r}$, pour chaque dimension de tableau, la taille de l'intersection de deux segments :

$$\begin{array}{r} E_1 = \{\varphi \mid \ \exists\kappa, \ I^w_{\lambda,\kappa} P^w_\kappa L^w_\kappa c^w_\kappa \ \leq \ \varphi_\lambda \ < \ I^w_{\lambda,\kappa}(P^w_\kappa L^w_\kappa c^w_\kappa + L^w_\kappa - 1)\} \\ E_2 = \{\varphi \mid \ \exists\kappa'', \ I^r_{\lambda,\kappa''} P^r_{\kappa''} L^r_{\kappa''} c^r_{\kappa''} \ \leq \ \varphi_\lambda \ < \ I^r_{\lambda,\kappa''}(P^r_{\kappa''} L^r_{\kappa''} c^r_{\kappa''} + L^r_{\kappa''} - 1) + M_{\lambda,\kappa'''}.up(m)_{\kappa'''}\} \end{array}$$

$$card(E_1) \bigcap card(E_2) \leq \mu^{w,r}$$

avec

$$\mu_\lambda^{w,r} = min\left(I^w_{\lambda,\kappa}\left(L^w_\kappa(P_\kappa+1)-1\right), I^r_{\lambda,\kappa''}\left(L^r_{\kappa''}(P^r_{\kappa''}+1)-1+M_{\lambda,\kappa'''}.up(m)_{\kappa'''}\right)\right) \quad (12.3)$$

L'avantage de cette sur-approximation est qu'elle ne dépend pas du quantificateur universel sur les partitions $\mathbf{c}_w$ et $\mathbf{c}_r$, ni de diséquation. Le volume de données référencées par une partition en lecture et une autre en écriture est donc borné par :

$$\mu^{w,r} = \prod_\lambda \mu_\lambda^{w,r} \quad (12.4)$$

Comme dans le cas général (formule 6.7, p. 118), il faut désormais évaluer l'ensemble des communications entre deux partitions temporelles. Nous considérons dans ce modèle simplifié que chaque processeur communique à la totalité des autres processeurs $\mu^{w,r}$ éléments de tableaux.

Soit npe^r et npe^w le nombre de processeurs respectifs pour T^r et T^w, le nombre maximal de processeurs impliqués dans la communication sera[1] :

$$npe^{wr} = max(npe^r, npe^w) \quad (12.5)$$

Nous ne comptabilisons pas la communication locale. Si la communication implique npe^{wr} processeurs[2], on obtient :

$$V^{w,r}_{elem} = \mu^{w,r}.npe^{wr}.(npe^{wr}-1) \quad (12.6)$$

Même si cette contrainte est non-linéaire, elle est d'une part plus simple que la contrainte générale (6.7, p. 118), d'autre part, elle correspond bien à la réalité, car si l'on augmente le nombre de processeurs les communications augmentent aussi. En revanche, elle sur-approxime de façon drastique les échanges entre processeurs, il suffit de comparer les figures (6.2, p. 118) et (12.2, p. 214) pour s'en rendre compte. Néanmoins, dans beaucoup de cas où les partitionnement sont vraiment différents, la formulation (6.7, p. 118) aurait conduit à un nombre d'échanges de messages très important.

Enfin, le dernier avantage de cette dégradation est de pouvoir prendre en compte les contraintes de topologie. En effet la topologie considérée permettant de tout communiquer simultanément, l'algorithme de communications consiste à tout communiquer en parallèle. Si une autre topologie imposait des contraintes supplémentaires sur les communications (lecture/écriture non-simultanées, un seul mouvement possible simultanément,...), il faudrait remplacer la formule (12.6, p. 213), par le nombre de messages échangés pour l'algorithme de communication associée à cette autre topologie.

- Modèle uniforme en émission

Propriété 4.1 *Le volume de communication en émission* $\mathcal{V}^w_{emis}(\mathbf{c^w})$ *peut être approximé par* V^w_{emis} *:*

$$V^w_{emis} = max_{c^w}(card\ (\{c^r \in \mathcal{D}(c^r, c^w)\}).V^{r,w}_{elem}) \quad (12.7)$$

[1]Pour le calcul du nombre de processeurs pour une tâche, se référer au modèle de machine (cf. chapitre 9, § 2.1, p. 151)

[2]Se reporter au modèle de la machine (cf. chapitre 9, § 2.1, p. 151) pour la contrainte du nombre de processeurs

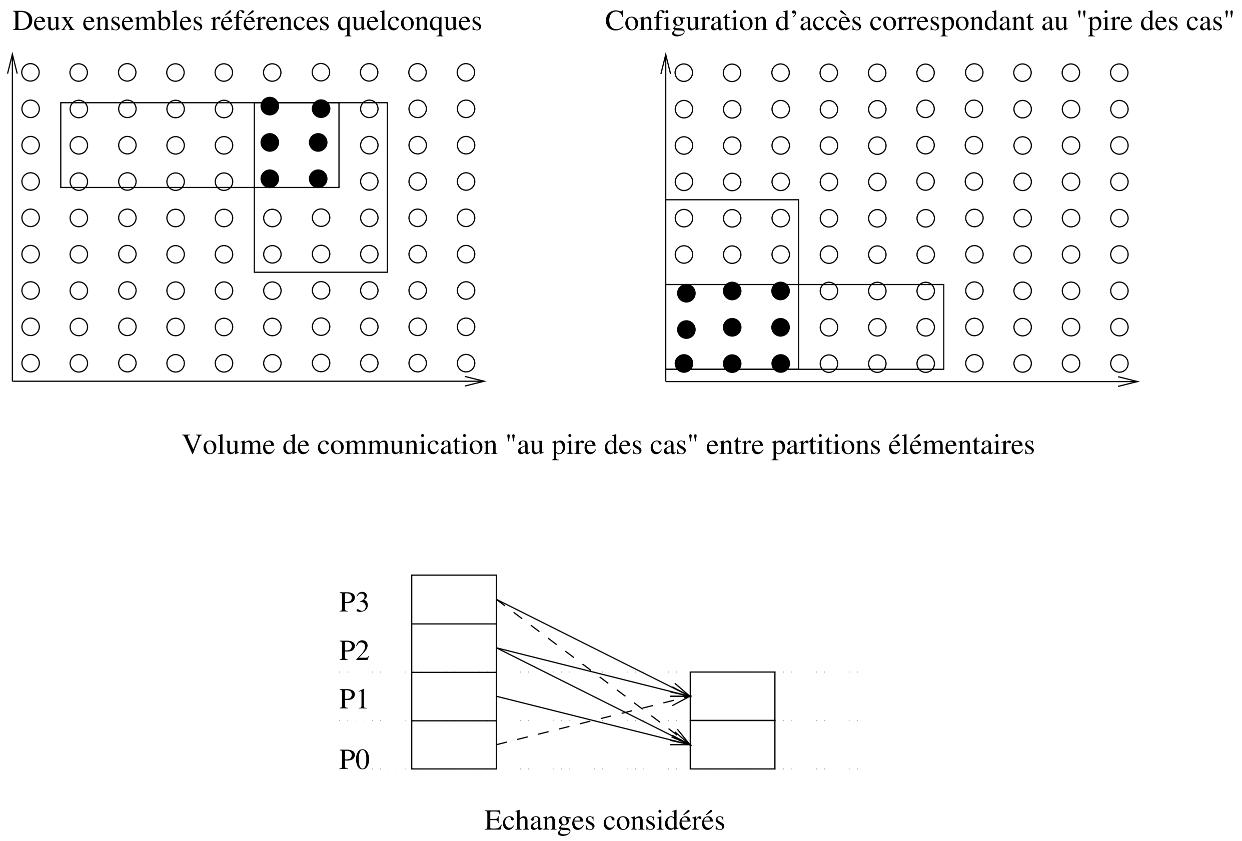


FIG. 12.2. – *Approximation des communications élémentaires*

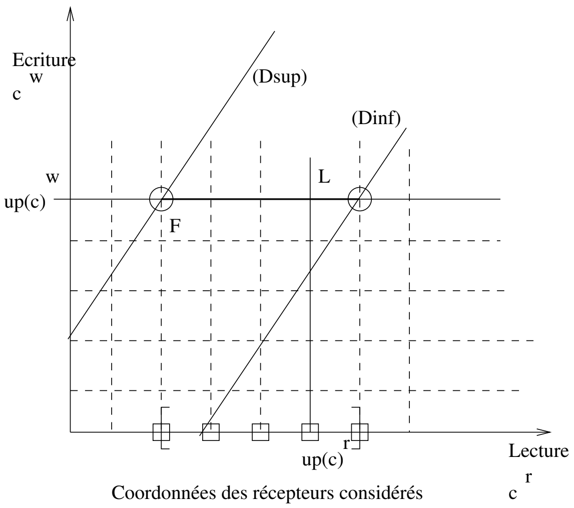


FIG. 12.3. – *Cardinalité de l'ensemble des coordonnées permettant de définir les vecteurs recepteurs*

Démonstration –
Le volume de communications élémentaire sur-approximé ne dépendant pas des vecteurs $\mathbf{c}$, pour la formule (6.9, p. 120), on ne considère plus qu'une somme sur une constante :

$$\forall c^w,\ \mathcal{V}_{emis}^{w}(\mathbf{c}^w)\ \leq \sum_{c^r \in \mathcal{D}(c^r,c^w)} V_{emis}^{r,w} \tag{12.8}$$

ce qui donne :

$$\forall c^w,\ \mathcal{V}_{emis}^{w}(\mathbf{c}^w)\ \leq\ card(\{c^r\ \in\ \mathcal{D}(c^r,c^w)\}).V_{emis}^{r,w} \tag{12.9}$$

Pour borner l'ensemble des c^w, il suffit de prendre le maximum de la cardinalité et l'on obtient (12.7, p. 213).

□

Grâce à la propriété (2.1, § 2, p. 97), la sur-approximation (12.7, p. 213) peut se réécrire par dimension de tableaux k :

$$V_{emis}^{w}\ =\ \prod_k (max_{c^w}(card\ \{c_{p'}^r \in \mathcal{D}_k(c_p^r, c_q^w)\ \}).V_{elem}^{r,w}) \tag{12.10}$$

Ceci revient à considérer la relation de dépendance dans les couples de plans formés par les composantes de $\mathbf{c}^r$ et $\mathbf{c}^w$, comme pour les dépendances (cf. chapitre 5, § 2, p. 97). On tient alors compte du polyèdre convexe de dépendances et des droites parallèles (Dsup) et (Dinf), alors le problème devient géométrique (fig. 12.3, p. 215). L'ensemble des coordonnées cherchées, pour un couple de plan et pour un vecteur $\mathbf{c}^w$ correspond à l'intersection de la droite $y = c^w$ et de la bande $[(Dsup),(Dinf)]$.
On peut calculer statiquement la taille du segment S_k qui représente cette intersection, nous passons ce calcul long et fastidieux, le résultat correspond à $S_k^w\ =\ \|\ x_L\ -\ x_F\ \|$. Ces points sont définis au niveau des dépendances (cf. chapitre 5, § 2.3, p. 100). Ceci permet de calculer la sur-approximation V^w du volume de communications en émission :

$$V^w \;=\; V^w_{elem}.\prod_k \; S^w_k \tag{12.11}$$

- Modèle uniforme en réception

  On peut procéder au même raisonnement concernant le modèle en réception, les points à considérer sur le polyèdre de dépendances sont alors $S^r_k \;=\; \parallel x_K - x_D \parallel$:

$$V^r \;=\; V^r_{recu}.\prod_k \; S^r_k \tag{12.12}$$

Seuls les modèles simplifiés par blocs émis et reçus ont été traités dans les deux sections précédentes. Cette formulation reste valable pour les modèles pipelinés en émission (resp. en reception). En effet, comme les volumes émis (resp. reçus) sont élémentaires, il suffit de fixer $L^w = 1$ (resp. L^r) pour obtenir les bons volumes.

Un autre type de sur-approximation pour modéliser les émission (resp. les réceptions) est envisageable. Il s'agit de considérer l'ensemble des volumes en écriture (resp. en lecture) et de les communiquer (resp. recevoir) à l'ensemble des processeurs. Soit V^w_{ecr} (resp. V^r_{lec}), le volume de données écrit (resp. lu), voir pour cela le chapitre (cf. chapitre 8, § 2.1.1, p. 144). Selon la même topologie que ci-dessus, dans le cas des émissions, nous avons vu (§ 4.6, p. 212) que le nombre total de données échangées (fig. 12.4-2, p. 217) est :

$$\mathcal{V}^w = V^w_{ecr} * npe^{wr}(npe^{wr} - 1)$$

Si l'on compare la qualité de cette sur-approximation avec celle définie en (12.7, p. 213), on calcule le rapport :

$$Q \;=\; \mathcal{V}^w / V^w \;=\; V^w_{ecr} / (\prod_k .S^r_k.\mu^{w,r})$$

Comme il est possible que $S^r_k = 1$, dans le cas où le polyèdre de dépendances se résume à une droite, et que $V^w_{ecr} \;\geq\; \mu^{w,r}$ (car μ repose sur un produit de min), alors on peut avoir $Q \;\geq\; 1$ (fig. 12.4, p. 217).

Dans tout les cas, la contrainte :

$$\forall \mathbf{c^w},\; \mathcal{V}^w_{emis}(\mathbf{c^w}) \;\leq\; \mathcal{V}^w$$

est une contrainte redondante supplémentaire dont il faut tenir compte, car elle ne dépend pas du polyèdre de dépendances mais d'un seul nid de boucles partitionnés.

La figure (12.4, p. 217) montre la même distribution d'un tableau et les communications effectives correspondant à une transposition, selon les deux approximations. Une communication correspond à un bloc de données circonscrit par les segments (pleins et pointillés) associé à une liste de destinataire. Nous donnons une communication élémentaire représentée par les flèches. Celles en pointillés matérialisent les communications inutiles. Voici le code parallèle simplifié (par rapport aux vecteurs $\mathbf{c}, \mathbf{p}, \mathbf{l}$) correspondant nécessitant ce type de transposition :

```
DO P=0,3
```

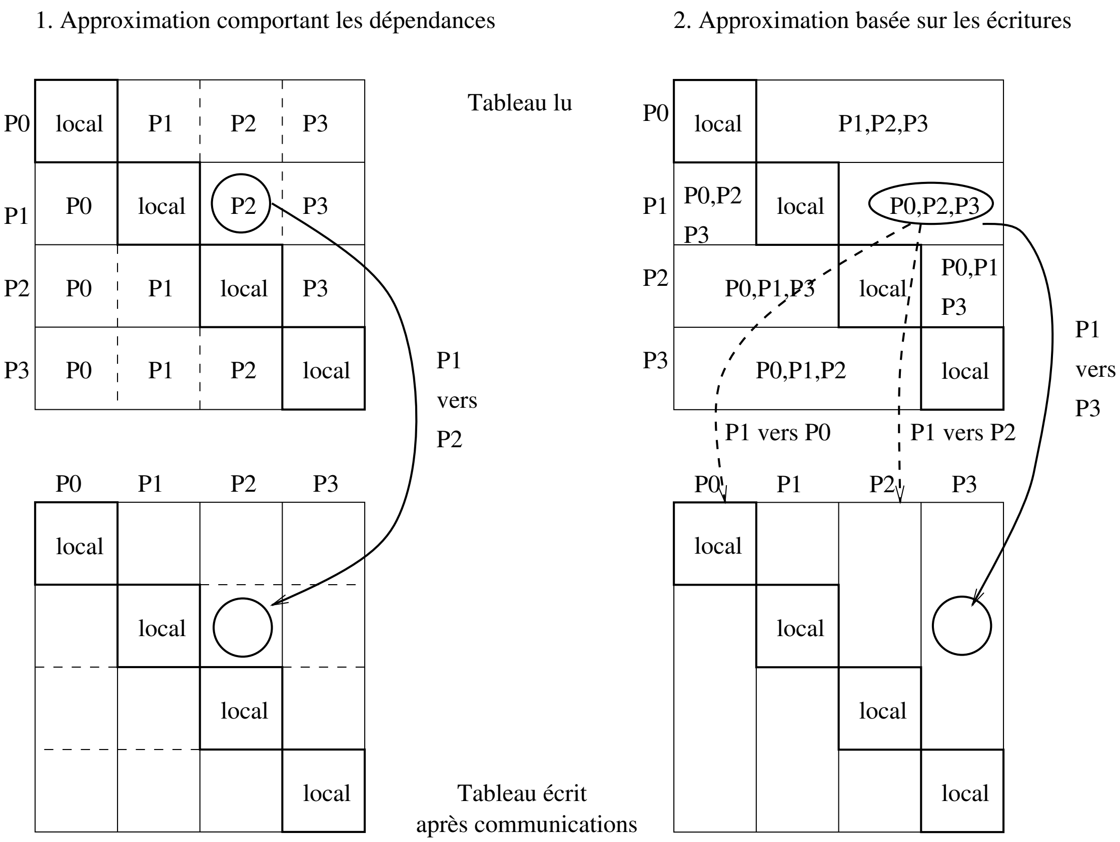


FIG. 12.4. – *Deux approximations*

```
  DO L1=0,3
    DO L2=0,10
      T[P,L1,L2]=...
    ENDDO
  ENDDO
ENDDO

DO P=0,3
  DO L=0,3
    DO M=0,10
      ...=T[L,P,M]
    ENDDO
  ENDDO
ENDDO
```

Voici donc un résumé des variables et contraintes locales aux modèle d'Interphase :

Interphase			
Descriptif	*Variables*	*Contrainte*	*Dimensions*
Echange maximal	$\{V_{elem}^{w,r}\}$		
Intersection élémentaire	$\{\mu^{w,r}\}$		
Nombre de recepteurs	$\{S^w\}_{w=1}^n$		
Coût de communications pour une macro-instruction	$\{V_{te}^w\}$	$V_{te}^w = V_{elem}^{w,r}.S^w$	autant que d'arc (r, w)

4.7 Architecture cible

Les données de l'architecture sont spécifiées par l'utilisateur. Il est aussi possible d'associer une fonction de coût à la machine ciblée (chap. 9).

Architecture cible			
Descriptif	*Variables*	*Contrainte*	*Dimensions*
Nombre de processeurs	npe	utilisateur	1
Mémoire	mem	utilisateur	1
Bande passante	a	utilisateur	1
Accès au réseau	b	utilisateur	1

4.8 Contraintes de relations

Dans la méthodologie que nous suivons dans cette thèse, un ensemble de variables, composantes d'un modèle dècrivent une fonctionnalité particulière. Il n'est donc pas nécéssaire de définir une même variable dans deux modèles différents. De ce fait, les variables utilisées par un modèle sont soit internes, soit externes. Les contraintes d'un modèle exploitant une variable externe engendrent une relation, entre celui-ci et le modèle où est déclarée la variable. Les relations se traduisent par des contraintes, souvent non-linéaires, nécessitant de combiner plusieurs contraintes élémentaires. Cela nécessite uniquement l'introduction de certaines variables intermédiaires que nous passons sous silence.

4.8.1 La croissance

La relation entre l'ordonnancement et le partitionnement est constitué de la contrainte de croissance (chap. 7, § ordonnancement.local, p. 2.2) ainsi que des dépendances. Si cette contrainte garantit au niveau du modèle conceptuel l'intégrité de l'ordonnancement, son implantation se traduit par un lien entre le partitionnement et l'ordonnancement à travers le vecteur $up(\mathbf{c})$, qui fixe les bornes multidimensionnelles des boucles partitionnées. En effet, ce vecteur peut être aussi bie vu comme un résultat du partitionnement ou comme le terme de la contrainte de croissance.

Ordonnancement, Partitionnement		
Descriptif	*Contrainte*	*Dimensions*
Ordonnancement des blocs affine strictement croissant	$\alpha_i^k \geq \alpha_{i+1}^k up(c_{i+1}^k)$	$\sum_{k=1}^{n}(dim(\mathcal{I}^k) - 1)$

4.8.2 Partitionnement et Ordonnancement : les dépendances

Les dépendances entre nids de boucles sont calculées à partir des fonctions d'accès et du partitionnement et se résument à un ensemble de sommets par couples de nids de boucles dépendant dont nous ne donnons pas la structure dans cette section. Les dépendances pourraient être assimilées à un modèle. Elles permettent aussi, associées à l'ordonnancement, de définir la distance maximale de dépendances (chap. 7, § 3.1, p. 135 et § 5.1.2, p. 141).

Selon les combinaisons de partitionnement des nids de boucles en lecture et en écriture, certains sommets peuvent être confondus. De ce fait, leur nombre varie au cours de la résolution globale du problème. Ce dernier point rend l'implantation en Programmation par Contraintes délicate car des contraintes et des variables peuvent disparaitre ou apparaitre dynamiquement lors de la résolution, pour lesquelles il faudrait maintenir l'ensemble des structures de données. Mais la plus grosse difficulté de la variation dynamique des polygones réside dans la structure du problème qui évolue, sans réelle prédiction possible, lorsque l'on "backtrack" sur des choix de partitionnements différents ou lorsque l'on avance dans les étapes de résolution. L'approche considérée dans cette thèse consiste à définir, grâce aux caractéristiques applicatives et aux matrices de partitionnement diagonales, un sur-ensemble de sommets, dont certains peuvent devenir confondus lors de la résolution.

La somme de contraintes engendrées par le nombre important de sommets du polytope de dépendances peuvent ralentir les étapes de propagations. Si une optimisation de nombre de contraintes n'avantage en rien la résolution, elle peut être pénalisantes pour les performances du système de placement. Une première optimisation consiste à prendre en compte uniquement les sommets pertinents du polytope de dépendance pour les contraintes de précédences.

- Suppressions des sommets superflus suivant les contraintes

  En raison de la forme particulière des contraintes posées sur le polyèdre des dépendances $\mathcal{P}$ (contraintes de type $d^r(\mathbf{c}^r) - d^w(\mathbf{c}^w) \leq$ ou $\geq m$), poser les contraintes sur tous les sommets engendre des redondances, ce qui est inutile et néanmoins coûteux; il faut en effet avoir à l'esprit qu'en PLC lorsqu'on pose des contraintes avec des produits scalaires n-dimensionnels, on génère n variables intermédiaires par produit scalaire. On allège donc de manière non négligeable le nombre de variables de domaine sur lesquels il faut propager les contraintes. Le fait que le polytope de dépendance est construit à partir d'un produit cartésien simplifie considérablement la tâche : on peut analyser la situation et éliminer les points au niveau des polygones $\mathcal{P}_k$ (cf. chap. 5, § 2, p. 97), en se plaçant dans le plan (c_p^w, c_q^r). En effet on a les propriétés suivantes qui permettent d'éliminer certains points $(\mathbf{c'}^w, \mathbf{c'}^r)$:

  Proposition 4.2

$$\forall k\ \forall (\mathbf{c}^w, \mathbf{c}^r) \in \mathcal{P}_k \mid d^r(\mathbf{c}^r) - d^w(\mathbf{c}^w) \geq min\ , \begin{cases} c'^w_p \leq c^w_p \\ c'^r_q \geq c^r_q \\ \mathbf{c'} = \mathbf{c} \\ \textit{sur les autres dimensions} \end{cases} \Longrightarrow d^r(\mathbf{c'}^r) - d^w(\mathbf{c'}^w) \geq min \tag{12.13}$$

$$\forall k\ \forall (\mathbf{c}^w, \mathbf{c}^r) \in \mathcal{P}_k \mid d^r(\mathbf{c}^r) - d^w(\mathbf{c}^w) \leq max\ , \begin{cases} c'^w_p \geq c^w_p \\ c'^r_q \leq c^r_q \\ \mathbf{c'} = \mathbf{c} \\ \textit{sur les autres dimensions} \end{cases} \Longrightarrow d^r(\mathbf{c'}^r) - d^w(\mathbf{c'}^w) \leq max \tag{12.14}$$

  <u>Démonstration</u> – Notons :

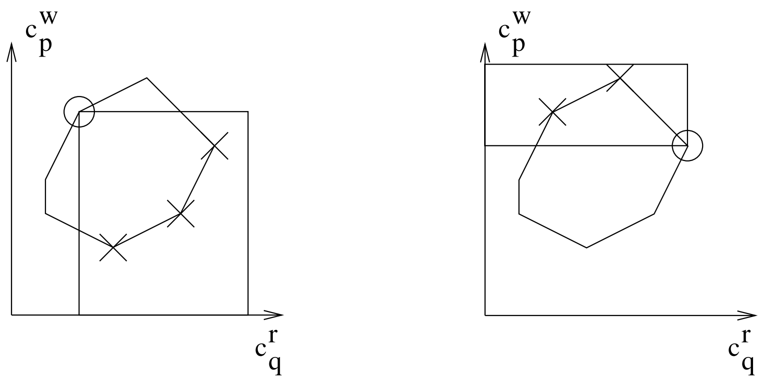


FIG. 12.5. – *Suppression de points pour les contraintes de type $\geq min$ et $\leq max$*

– $\mathbf{c'}_s = (c_1, ..., c_{s-1}, c_{s+1}, ..., c_n)$
– $\gamma_k(\mathbf{c'}^w_p, \mathbf{c'}^r_q) = d^r(\mathbf{c}^r) - d^w(\mathbf{c}^w) - (\alpha^r_q c^r_q - \alpha^w_p c^w_p)$
– Γ_k l'ensemble des valeurs prises par γ_k sur $\otimes_{l \neq k} \mathcal{P}_l$.

Alors

$$(\mathbf{c}^w, \mathbf{c}^r) \in \mathcal{P} \Rightarrow d^r(\mathbf{c}^r) - d^w(\mathbf{c}^w) \geq min \iff \left\{ \begin{array}{l} (c^w_p, c^r_q) \in \mathcal{P}_k \\ \gamma_k \in \Gamma_k \end{array} \right. \Rightarrow \alpha^r_q c^r_q - \alpha^w_p c^w_p + \gamma_k \geq min$$

et de même pour (12.14). à partir de là le résultat est trivial, en utilisant le fait que $\forall k \; \alpha_k > 0$. □

En examinant les expressions des points de $\mathcal{P}_k = \{A^k, BC^k, CDL^k, EKL^k, FGK^k, GH^k\}$ donnés dans la proposition (2.2, chap. 5, § 2.3, p. 100) et les différents cas, on conclut que deux points suffisent pour chaque type de contrainte :

Proposition 4.3 *Poser les contraintes du type*

$$d^r(\mathbf{c'}^r) - d^w(\mathbf{c'}^w) \geq min \tag{12.15}$$

sur les sommets de $\mathcal{P} = \bigotimes_k \mathcal{P}_k$ est équivalent à poser ces contraintes sur $\mathcal{P}^o = \bigotimes_k \mathcal{P}^o_k$ avec $\mathcal{P}^o_k = \{GH^k, FGK^k\}$. Poser les contraintes du type

$$d^r(\mathbf{c'}^r) - d^w(\mathbf{c'}^w) \leq max \tag{12.16}$$

sur les sommets de $\mathcal{P} = \bigotimes_k \mathcal{P}_k$ est équivalent à poser ces contraintes sur $\mathcal{P}^m = \bigotimes_k \mathcal{P}^m_k$ avec $\mathcal{P}^m_k = \{BC^k, EKL^k\}$.

<u>Démonstration</u> – On a les relations suivantes entre les coordonnées de A, BC, CDL, EKL, FGK et GH :

(M_1, M_2)	x	y
(A, BC)	$\leq$	$= (\geq)$
(A, GH)	$= (\geq)$	$\leq$
(BC, GH)	$\geq$	$\leq$
(CDL, EKL)	$= (\leq)$	$\geq$
(CDL, FGK)	$\geq$	$\leq$
(EKL, FGK)	$\geq$	$= (\leq)$

qu'on obtient en examinant les formules des coordonnées des points et en raisonnant suivant les cas lorsqu'on doit comparer des minimums ou des maximums. Examinons le cas des contraintes (12.15) posées par l'ordonnancement en utilisant dans la proposition (4.2) et en passant la revue les lignes du tableau ci-dessus :

1. on garde A, on élimine BC
2. on garde GH, on élimine A
3. (-)
4. on garde CDL, on élimine EKL
5. on garde FGK, on élimine CDL
6. (-)

Il ne reste plus que GH et FGK. On procède de même pour $\mathcal{P}^m$. □

On divise donc le nombre de points par plan par 3. Après produit cartésien des $\mathcal{P}_k$ pour obtenir $\mathcal{P}$, et si par exemple les itérateurs ont tous les deux n dimensions, on divise par 3^n le nombre de points du polyèdre complet sur lesquels il faut poser les contraintes, et ce pour chaque nid de boucles. Si $n = 3$, on divise par 27 ce nombre, si $n = 4$, cas encore fréquent, on passe de 1296 points à 16.

En fait, la suppression des points superflus permet de mettre en lumière la sémantique des droites D_{sup} et D_{inf}. En effet, les sommets gardés pour la contrainte de précédence appartiennent à la droite ("premier lu", "derniere écrite"). La formulation en français de la contrainte correspond donc à : **soit deux ensembles de blocs, l'un en lecture, l'autre en écriture inter-dépendant, la date d'exécution du premier bloc en lecture doit être supérieure à la date d'exécution du dernier bloc en écriture**.

La deuxième optimisation correspond à ne considérer que la droite ("première écrite","dernière lue"). Cette droite caractérise la durée de vie des données : **une donnée doit conserver son allocation impérativement entre le moment où le premier bloc en écriture est exécuté et le moment où le dernier bloc en lecture est exécuté**.

- Optimisation du produit cartésien

  Décomposons le produit cartésien $\mathcal{P}^o = \mathcal{P}^o_k \otimes \mathcal{P'}^o_k = \{BC^k, EKL^k\} \otimes \mathcal{P'}^o_k$. Il apparaît que Card$\mathcal{P'}^o_k$ points dans $\mathcal{P}$ ont 2 coordonnées identiques. Lorsqu'on pose

$$d^r(\mathbf{c}^r) - d^w(\mathbf{c}^w) = N(\sum_{i=1}^{n^r} \alpha_i^r c_i^r + \beta^r - \sum_{i=1}^{n^w} \alpha_i^w c_i^w - \beta^w) + k^r - k^w$$

  sur ces points on pose donc Card$\mathcal{P'}^o_k$ fois les 2 produits intermédiaires $z_k^w = \alpha_p^w c_p^w$ et $z_k^r = \alpha_q^r c_q^w$. On génère donc $2 * (\text{Card}\mathcal{P'}^o_k - 1)$ variables inutiles pour chaque plan. Le phénomène est identique pour $\mathcal{P}^m$. Il convient donc de poser les produits intermédiaires sur les polygones et non sur le polyèdre produit. Si les deux itérateurs sont de dimension n, la pose des contraintes d'ordonnancement (entre deux tâches) sur $\mathcal{P}^o$ n'introduit plus que $2(2 * n) = 4n$ variables de domaine intermédiaires au lieu de $2n * 2^n = 4n * 2^{n-1}$.

L'élimination des contraintes redondantes et cette optimisation font passer le nombre de variables intermédiaires de $(2n\ 6^n)$ à $4n$. Le tableau suivant résume les améliorations :
p : nombre de contraintes de produit = nombre de variables intermédiaires
l : nombre de contraintes d'inégalités = nombre de produits scalaires divisé par 2

Dimension	Nombre de contraintes							
n	sans optimisation		supp. de points		optimisation PLC		les deux	
	p	l	p	l	p	l	p	l
1	12	6	4	2	12	6	4	2
2	72	36	16	4	24	36	8	4
3	1296	216	48	8	36	216	12	8
4	10368	1296	128	16	48	1296	16	16

Il faut noter qu'en appliquant uniquement l'optimisation PLC, on a peu de variables intermédiaires mais on garde un nombre de contraintes linéaires identique au cas sans optimisation. **Soulignons enfin que cette optimisation n'améliore pas la résolution mais accélère la propagation**.

Partitionnement et Ordonnancement : Dépendances		
Descriptif	*Contrainte*	*Dimensions*
Dépendances de flot	$\forall(\mathbf{c}_S^w, \mathbf{c}_S^r) \in \mathcal{D}(\mathbf{c}_S^w, \mathbf{c}_S^r) \Rightarrow d^w(\mathbf{c}_S^w) + 1 \leq d^r(\mathbf{c}_S^r)$	autant que d'arrêtes (r, w)
Distance maximale entre partitions	$d_{max}^{w,r} = \max_{\mathcal{D}(\mathbf{c}_S^w, \mathbf{c}_S^r)} (d^w(\mathbf{c}_S^w) - d^r(\mathbf{c}_S^r))$	autant que d'arrêtes (r, w)

4.8.3 Partitionnement, Temps physique et Interphase

Ce lien permet d'associer un coût physique à un événement. Il prend en compte le nombre de macro-instructions séquentielles dans un bloc et le coût des communications (resp. chap. 10, § 4.2, p. 168 et chap. 6, § 4.6, p. 211).

Partitionnement, Temps physique et Interphase		
Descriptif	*Contrainte*	*Dimensions*
Calcul-Comms	$ct(k) = cout_{elem}^k + (\det(L) - 1).max(cout(V_{te}^k), cout_{elem}^k) + cout(V_{te}^k)$	n

4.8.4 Partitionnement, Temps physique

La relation suivante permet d'obtenir une contrainte de récurence globale (chap. 10, § 6.3.3, p. 173).

Partitionnement, Temps physique		
Descriptif	*Contrainte*	*Dimensions*
Recurence	$tp(k).det(P^k)^{-1} \leq Recurence$	1

4.8.5 Ordonnancement, Temps physique

Le même type de relation, en plus complexe car utilisant le plus long chemin, permet le calcul de la latence (chap. 10, § 5, p. 169).

Ordonnancement, Temps Physiques : Dépendances et périodes		
Descriptif	*Contrainte*	*Dimensions*
Plus long chemin	$\Lambda = pluslong(d_{max}^{w,r})$	1
Temps réel	$tp(k) = \sum_{k=1..N} nbc(0, \alpha_{\infty}^{1}).ct(k)$	1

4.8.6 Interphase, Partitionnement

Ces contraintes complexes permettent de sur-approximer le coût des communications à partir du partitionnement et des fonctions d'accès (chap. 6, § 4.6, p. 211).

Interphase, Partitionnement		
Descriptif	*Contrainte*	*Dimensions*
Intersection élémentaire	$\mu^{w,r} = card(\mathcal{R}^w(\mathcal{P}^w(0,0)) \cup \mathcal{R}^r(\mathcal{P}^r(0,0)))$	nombre d'arc (r,w)
Echange maximal	$V_{elem}^{w,r} = \mu^{w,r}.max(det(P)^r, det(P)^w)(max(det(P)^r, det(P)^w) - 1)$	nombre d'arc (r,w)

4.8.7 Interphase, Partitionnement : Dépendances

Les dépendances permettent de caractériser le nombre de récepteurs. Ceci nécessite l'utilisation des sommets du polytope de dépendances x_L, x_F par plan de dépendance p (§ 4.6, p. 211).

Interphase, Partitionnement : Dépendances		
Descriptif	*Contrainte*	*Dimensions*
Nombre de recepteurs	$S^w = \prod_p (\parallel x_L - x_F \parallel))_p$	nombre d'arc (r,w)

4.8.8 Mémoire, Partitionnement

Cette relation correspond aux volumes de données référencées, qui prend en compte les "trous" dans les accès et les "recouvrements" (chap. 8, § 2.1.1, p. 144).

Mémoire, Partitionnement		
Descriptif	*Contrainte*	*Dimensions*
Volumes de données entre partitions	$\left\lceil \frac{\Pi_{l=1}^{n}(\mathbf{I.(L-1)+M.up(m-1)})_l}{\Pi_{l=1}^{n} pgcd(I_{l,c}, M_{l,c'})} \right\rceil$	autant que d'arrêtes (r, w)

4.8.9 Architecture, Mémoire, Ordonnancement : Dépendances

La contrainte de mémoire utilise les distances, la contrainte de cardinalité sur cette distance ainsi que les volumes de données écrits et lus (chap. 8, § 3.3, p. 148).

Architecture, Mémoire, Ordonnancement : Dépendances		
Descriptif	*Contrainte*	*Dimensions*
Approximation	$\left(\sum_s nbc^s(0, d^s_{max})\mathcal{V}^s_{lec}\right) + 2\max_s \mathcal{V}^s_{ecr} \leq Memoire$	1

4.8.10 Interphase, Architecture

La fonction coût prend en argument les volumes de données échangées (chap. 6, § 2.5.3, p. 121).

Interphase, Architecture		
Descriptif	*Contrainte*	*Dimensions*
Bande passante	$cout(V^k_{te}) = a.V^k_{te} + b$	n

4.8.11 Architecture, Partitionnement

Le nombre de processeurs lie le partitionnement et l'architecture (chap. 9, § 2.1, p. 151).

Architecture, Partitionnement		
Descriptif	*Contrainte*	*Dimensions*
Nombre de processeur	$\max_{k=[1..n]}(det(P^k))$	1

```
ordonne(LVecUpC,LMatP,LMatL,LDep,LDFonc,LDistMax):-
        cons_ordonne_local(LDFonc),
        affine_croissant(LVecUpC,LDFonc),
        dependances_ordonnancement(LVecUpC,LDFonc,LDep,LDistMax,LMatP,LMatL).
```

FIG. 12.6. – *Modèle et relations*

```
[...]
        distance_polytope(LSommets,(LAlphaW,BetaW),(LAlphaR,BetaR),LDiffDFonc),
        maxlist(DistMax,LDiffDFonc),
[...]
```

FIG. 12.7. – *Distance maximale*

4.9 Structure de modèles et de relations

De manière générique, un modèle se compose de définition de variables sur lesquelles reposent les contraintes propres aux modèles. Ces variables décrivent un espace de solutions local au modèle. Les relations permettent de sélectionner les solutions pertinentes par rapport aux autres modèles. Au niveau de l'implantation des modèles, les variables doivent impérativement être définies avant d'être contraintes. C'est la seule règle à respecter sur l'ordre des instructions qui compose l'implantation. L'utilisation des variables par de multiples contraintes est assurée par l'unification du système de programmation logique avec contraintes. Nous donnons, en figure (12.1) un ordre arbitraire de définition des modèles et des relations, suivi du prédicat d'optimisation.

Nous raffinons dans cette section notre présentation de l'implantation en focalisant sur un modèle comportant certaines des relations avec les autres modèles (fig. 12.6). Le prédicat `cons_ordonne_local` construit les variables d'ordonnancement `LDFonc`. Le prédicat `affine_croissant` définit la fonction croissante, établissant ainsi une première relation avec le partitionnement. Enfin, `dependances_ordonnancement` permet de définir les sommets des dépendances à partir du partitionnement et d'en déduire la distance maximale.

Voici justement, figure (12.7), l'implantation de la contrainte de distance maximale utilisant les sommets du polytope de dépendances. Dans cette partie de clause, le prédicat `distance_polytope` permet de lier les sommets du polytope avec les fonctions d'ordonnancement associées aux blocs en écriture et aux blocs en lecture (resp. `(LAlphaW,BetaW),(LAlphaR,BetaR)`). Le résultat est une liste de différences `LDiffDFonc`, pour laquelle on choisira la différence maximale grâce au prédicat `maxlist`.

5 Visualisation des solutions complexes

Le programmeur de traitement du signal doit pouvoir disposer d'une interface intuitive visualisant les choix de parallélisme, de localité, les différents effets sur les capacité mémoires... En fait chaque modèle est concerné par une interface particulière.

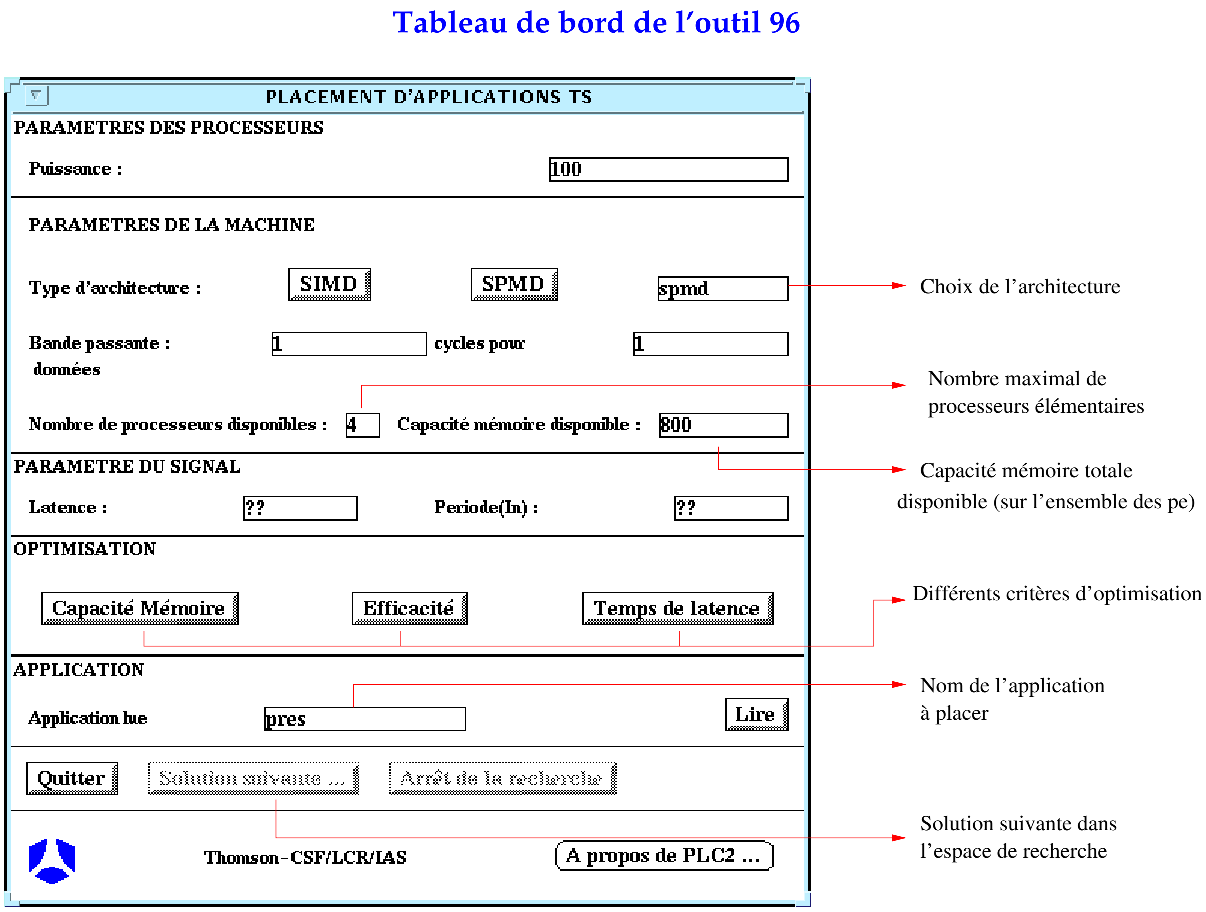


FIG. 12.8. – *Tableau de bord du système de placement*

5.1 Interface générale

L'interface générale permet de poser les contraintes sur les paramètres clefs pour l'expert: le nombre de processeurs, la taille mémoire, la bande passante caractérisant l'architecture ainsi que la récurence d'entrée et la latence pour les paramètres du signal (cf. fig 12.8).

5.2 Résultats des optimisations

Une interface générale résume l'optimisation selon un paramètres critère de placement et permet de visualiser paramètres globalement l'impact sur les autres paramètres du problème. Ce résumé produit de la même façon un fichier au format ASCII, utile pour les tests de non-régression du système.

5.3 Partitionnement

L'interface de ce modèle adopte une technique très utilisée par les experts en traitement du signal numérique de Thomson-CSF. Il s'agit de donner un axe séquentiel (correspondant au vecteur **l**, l'axe vertical sur la figure 12.9) et un axe parallèle (correspondant au vecteur **p**, l'axe horizontal sur la

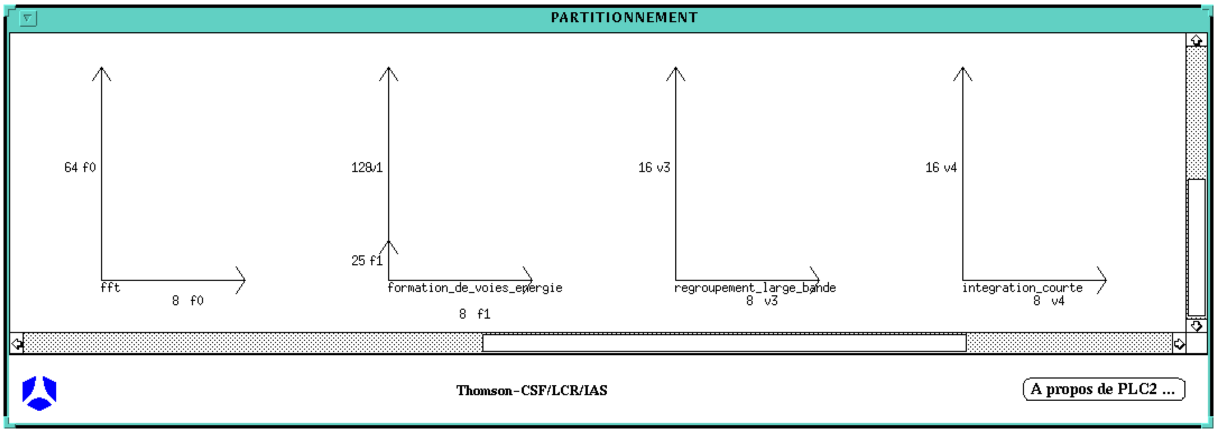


FIG. 12.9. – *Le partitionnement caractérisant le placement par nid de boucles*

figure 12.9) pour décrire le partitionnement par rapport à ces dimensions. Il en résulte aussi le degré de parallélisme et les volumes de calculs.

5.4 Communications

Cette interface présente le graphe de flot de données dont les arcs sont de différentes couleurs représentant les communications, les alignements, les communications recouvertes (fig. 12.10).

5.5 Ordonnancement

L'ordonnancement événementiel présente par nid de boucles, dans le temps et par processeur les exécution des blocs. Cette interface **est obtenue par simulation événementielle de l'exécution** des solutions de placement (fig. 12.11). Chaque carré de couleur représente un bloc (vecteur **c**) comportant plusieurs sous-blocs parallèles (vecteur **p**) ainsi que les itérations séquentielles (vecteur **l**).

L'ordonnancement physique (fig. 12.12) est une **simulation de l'exécution réelle** où l'axe négatif des ordonnées présente la durée des communications non recouvertes par les calculs et l'axe positif la durée des calculs (sur une échelle logarithmique).

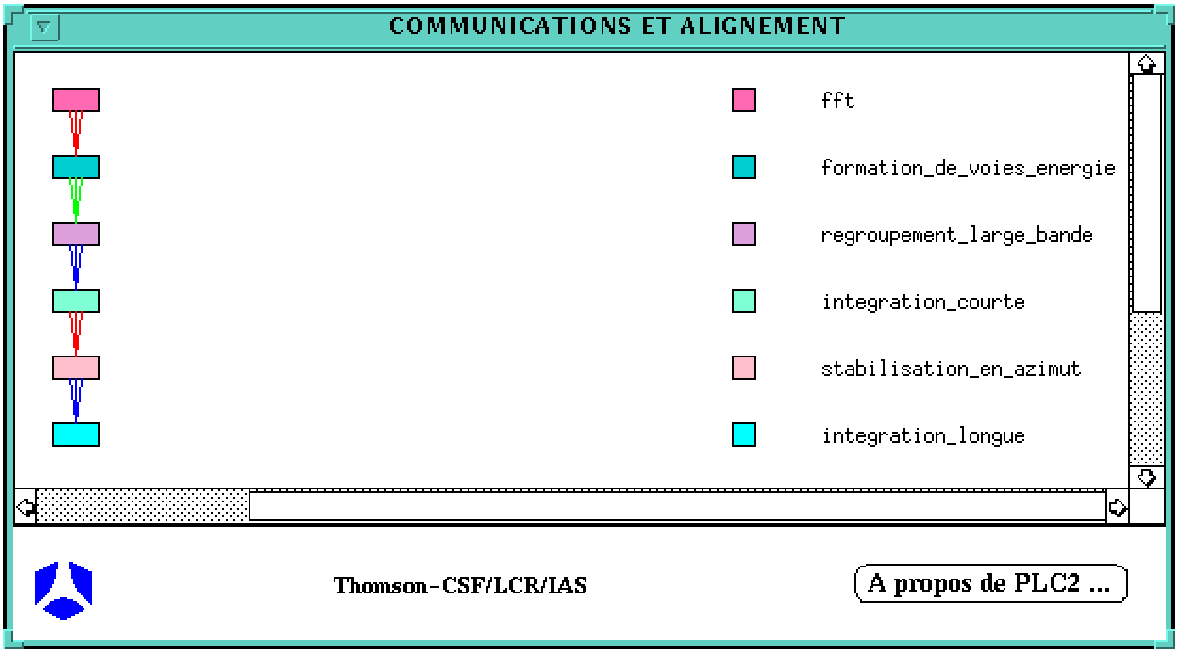


FIG. 12.10. – *Les communications*

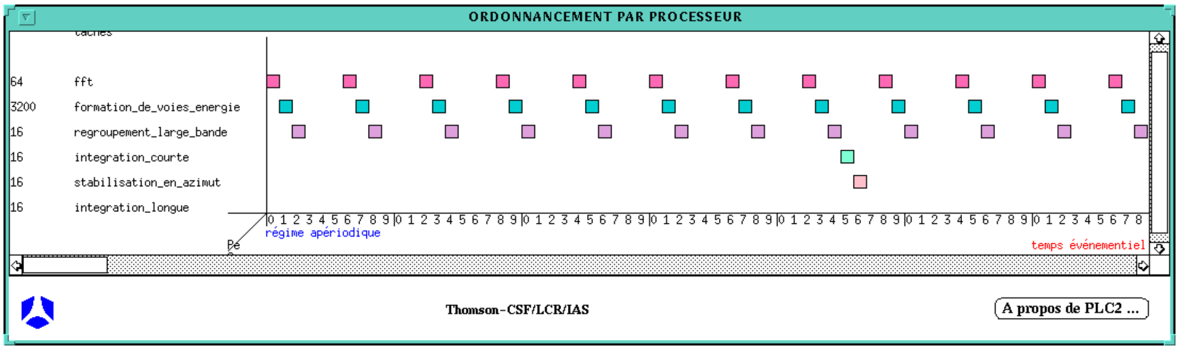


FIG. 12.11. – *Ordonnancement événementiel*

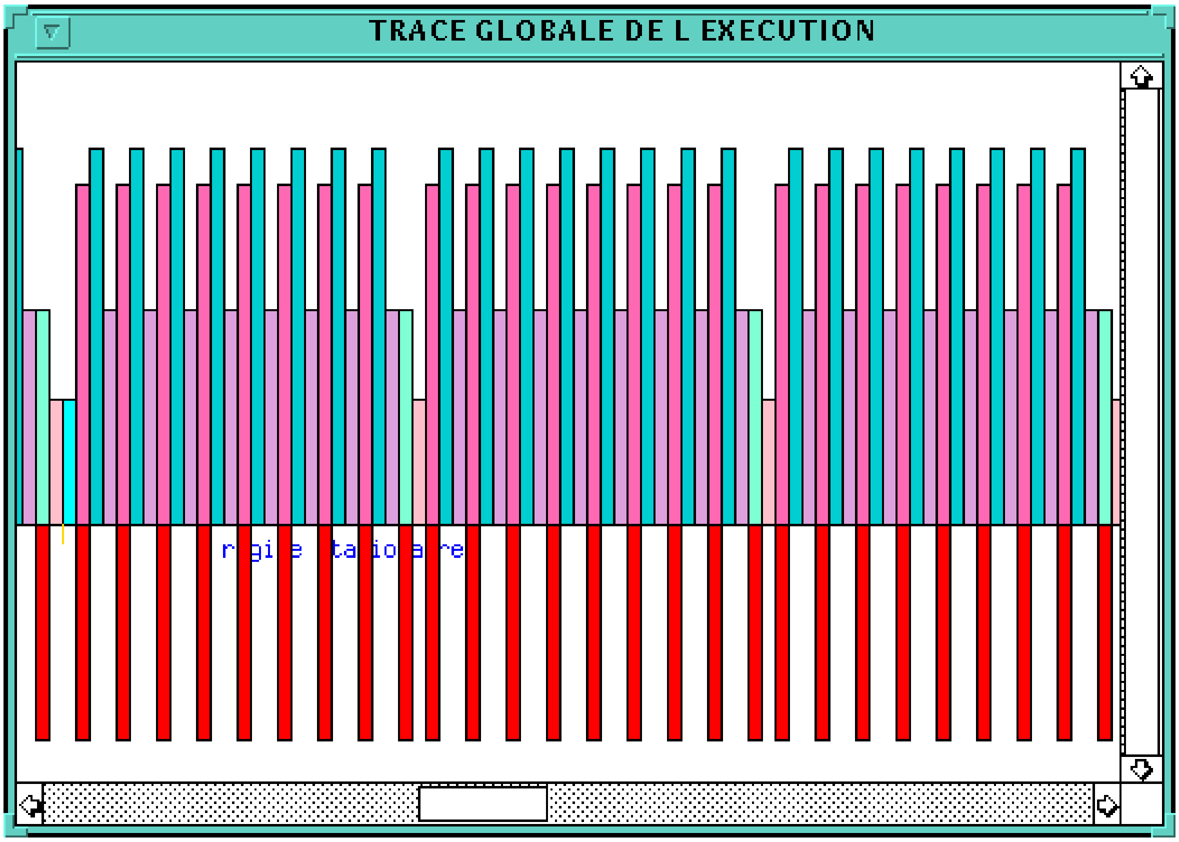


FIG. 12.12. – *Trace physique de l'ordonnancement*

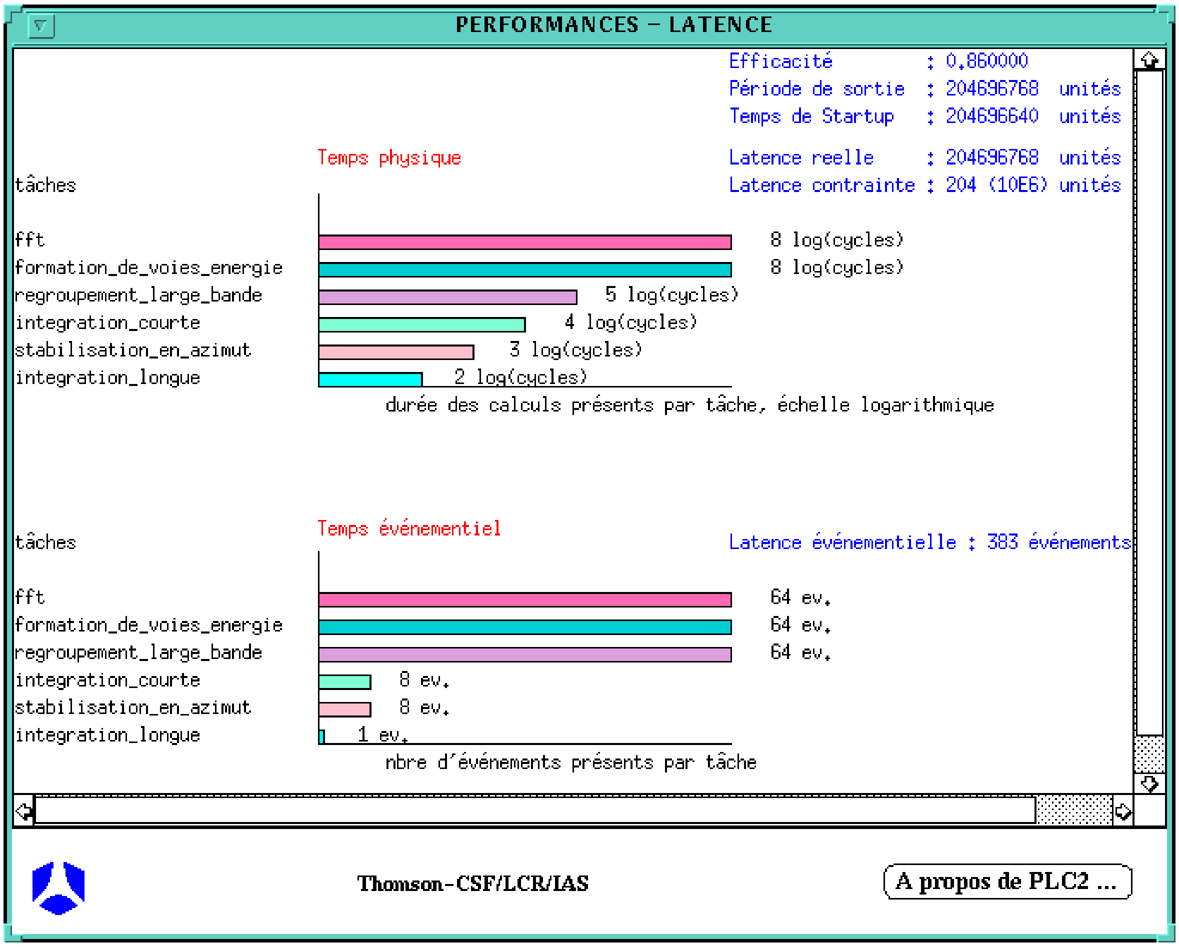


FIG. 12.13. – *Interprétation de la latence*

5.6 Signal

Le signal se compose de deux interfaces, l'une pour la latence, l'autre pour l'acquisition. Les paramètres caractéristiques sont les mêmes : elles permettent de visualiser le nombre d'évenements apparus dans la distance de dépendance (resp. période de l'aquisition) et de vérifier la pondération avec les coûts associés aux événements élémentaires.

5.7 Allocations

La représentation graphique des allocations (fig. 12.14) résulte de l'ordonnancement événementiel : elle présente, par nid de boucles, dans le temps et par processeur la durée des allocations, le nombre de blocs à allouer et le volume des blocs (sur une échelle logarithmique). Cette interface **est obtenue par simulation événementielle des allocations** correspondant aux solutions de placement.

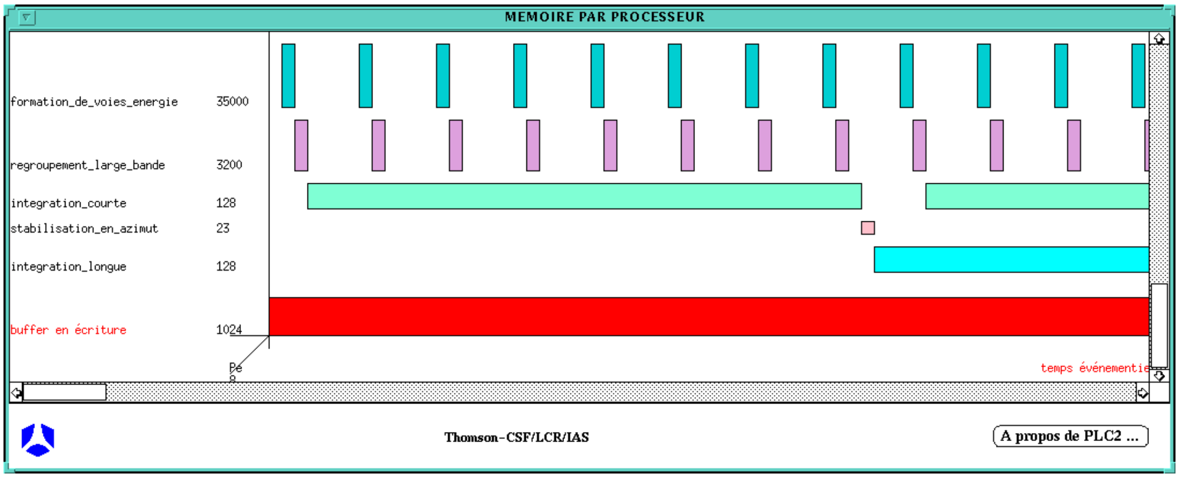


FIG. 12.14. – *Interprétation des allocations mémoires*

6 Conclusion

Composée à l'origine de différentes parties hétérogènes, le problème de placement est intégré, lors de l'implantantion, dans une structure homogène qu'est CCMBC (chap. 11). Les modèles et relations spécifiés sont exactement représentés par des parties de programme que l'on peut manipuler indépendamment. Aussi, il est intéressant de souligner que la réalisation reste homogène lorsque l'on spécialise les modèles, ou lorsque ceux-ci sont inhibés. De plus, la possibilité de construire facilement des critères d'optimisation sans avoir à redéfinir les modèles marque la différence avec les approches utilisant des algorithmes spécialisés.

Ces avantages nous ont permis d'établir de nombreuses propriétés intéressantes comme la maintenabilité et l'évolutivité du système de placement même. D'un autre point de vue, la manipulation des modèles et relations pourrait être dévolues au programmeur de traitement du signal. Ceci lui apporterait une meilleure compréhension du problème ainsi qu'une assistance non négligeable. Nous avons là aussi expliqué comment une telle utilisation se rapproche en fait du prototypage virtuel.

En conclusion, ce chapitre nous a permis de comprendre une conrétisation de l'approche CCMBC. Il expose la déclinaison du programme à partir d'une spécification multi-modèle d'une part et souligne les nombreuses propriétés fonctionnelles qu'offre l'approche globale d'autre part.

Chapitre 13

Méthodes avancées de résolution

1 Introduction

La technologie sous-jacente à l'approche concurrente est basée sur une résolution incomplète. La prise en compte de solutions partielles aux modèles permet en effet d'orienter très tôt la résolution du placement. La complétude est alors assurée par une stratégie de recherche de solutions qui influence l'échange d'informations partielles entre les modèles. Celle-ci s'appuie aussi bien sur la connaissance du domaine de placement que sur la structure du problème.

La première série d'expérimentations, portant sur des cas d'écoles (chap. 14), met en évidence que sur des exemples de faibles importances, le système de placement donne de bons résultats sans avoir à définir une recherche de solutions sophistiquée. Cependant, sur des exemples de taille importante, la profondeur et les bornes de boucles, ainsi que le nombre de nids impliquent une combinatoire importante. Elle nécessite donc l'introduction d'heuristiques pour diminuer le nombre d'étapes de résolution. La richesse de description de la modélisation concurrente nous ouvre de nombreuses possibilités pour exprimer des heuristiques de domaine. En effet, on dispose d'un choix important de variables dont le rôle est parfaitement défini. Parmi l'ensemble des heuristiques de domaine utilisées, on distingue deux groupes. L'un utilise le partitionnement et traite du parallélisme et des communications, et l'autre l'ordonnancement, l'allocation et le temps réel.

Les techniques spécialisées de placement, présentées au niveau du chapitre 1, (§ 4.1, p. 38, et § 4.2, p. 38), se concrétisent ici sous la forme d'heuristiques. En effet, les approches classiquement utilisées orientent les choix de placement ou de parallélisation selon des critères comme la localité ou le parallélisme sans réellement chercher à les composer. Or c'est bien une telle composition d'heuristiques qui facilite l'instanciation des variables de placement par des valeurs intéressantes.

Si le choix des valeurs, au travers des heuristiques de domaine, est fondamental pour obtenir des solutions lors de la phase de recherche, le choix des variables à explorer est aussi important. Les stratégies de résolution consistent justement à construire l'ordre dans lequel les variables seront explorées. Cette construction peut être statique, lorsque cet ordre est définit à l'avance par exemple à la suite d'analyses statiques du programme de traitement du signal. Elle peut aussi être dynamique, lorsque l'ordre est redéfini en cours de résolution. Les choix concernant les heuristiques peuvent aussi être controlés par la stratégie. Nous étudierons comment cela se traduit au niveau de l'implantation afin de conserver une structure homogène de résolution.

Quelques soient les stratégies et les heuristiques choisis, la recherche de solutions demeure exhaustive. L'arbre de recherche est entièrement parcouru et si des nœuds ne sont pas explorés cela résulte de la propagation sur les variables du problème.

En résumé, la fonction de la stratégie est de compléter la résolution du système de placement. La structure de l'arbre de recherche est basée sur l'ordre des variables à parcourir. Les heuristiques permettent de choisir les branches de l'arbre en orientant le choix des valeurs.

La section (§ 2) présente les familles d'heuristiques qui ont été testées sur des exemples d'échelle réelle, la section (§ 3) un ensemble potentiel de contraintes redondantes tandis que la section (§ 4) illustre le type de stratégie se révélant efficace sur des exemples de taille réelle.

2 Heuristiques de domaine

Si l'on classe les heuristiques mises en œuvre lors de la réalisation d'un système d'échelle réelle, nous retrouvons les différents thèmes abordés dans le domaine du placement chapitre 1 (§ 4.1, p. 38 et § 4.2, p. 38). Certaines heuristiques considèrent le choix des valeurs pour l'instanciation des variables du problème de placement. Un choix correspond à une étape de résolution que le système peut remettre en question par "*backtrack*". D'autres heuristiques se composent de contraintes réduisant les domaines des variables. Pour contrôler ceux-ci, nous associons une variable booléenne à l'activation de l'heuristique.

2.1 Parallélisme et localité

Les heuristiques décrites dans cette section sont relatives aux matrices de partitionnement. Celles-ci permettent déjà de spécifier des heuristiques portant sur la localité et le parallélisme, entraînant de ce fait des optimisations sur la mémoire et les communications.

Le vocabulaire utilisé dans cette section est définit dans les chapitres sur les modèles de partitionnement (chap. 4, § 4.2.2, p. 92), de l'interphase (chap. 6, § 4, p. 126) et du signal temps-réel (chap. 10, § 4, p. 167).

2.1.1 Dépendances en lecture

Une façon de maximiser la localité consiste à analyser statiquement les réutilisations d'informations grâce aux dépendances lecture/lecture définies au chapitre 5, § 4.2, p. 107. Pour éviter les communications, on minimise le parallélisme sur les dimensions concernées. Pour permettre la réutilisation des données et optimiser ainsi la capacité mémoire, on favorise la *sérialisation* (mise en série) de ces même dimensions sur un processeur. Voici comment sont instanciées les variables correspondantes :

Parallélisme : Cela consiste à considérer les valeurs d'un élément de matrice P_{ll}^{k} correspondant à la dimension k qui comporte les dépendances. On procède à l'instanciation de la variable par des valeurs du domaine selon l'ordre de la borne minimale à la borne maximale.

Mémoire : On considère les valeurs d'un élément de matrice L_{ll}^{k} correspondant à une dimension k qui comporte les dépendances. On procède à l'instanciation de la variable par des valeurs du domaine selon l'ordre de la borne maximale à la borne minimale.

Compte tenu du lien du partitionnement avec les dépendances de flot (chap. 5, § 4.3, p. 109), il est clair que ces deux heuristiques ont aussi un effet sur l'ordonnancement que nous ne chercherons pas à analyser dans cette section.

2.1.2 Maximiser le parallélisme

Exploiter complètement les ressources de l'architecture cible nécessite aussi de favoriser les solutions pour lesquelles le parallélisme est maximal.

Nous proposons une méthode simple pour **maximiser le parallélisme**, il suffit pour cela d'instancier les variables de partitionnement P_{ll}^{k} par des valeurs du domaine selon l'ordre de la borne maximale à la borne minimale.

Si cette heuristique favorise les solutions disposant d'un parallélisme maximal, elles peuvent défavoriser les communications rendues complexes par le nombre de processeurs. Il est donc conseillé de combiner cette heuristique avec d'autres qui permettent d'éviter les communications (cf. § 2.1.1 et § 2.1.3).

2.1.3 Aligner les données

Une autre façon de **maximiser la localité** et de favoriser les couples de partitionnement pour lesquels les données écrites sur un processeur seront réutilisées sur ce même processeur. Cela correspond en fait à satisfaire les contraintes d'alignement définies en (chap. 6, § 4, p. 126).

$$\begin{aligned} P^r &= P^w \\ I^r L^r &= I^w L^w \end{aligned}$$

où w, r sont des sommets dépendants du (DFG).

Si le système ne parvient pas à trouver des solutions à l'aide de ces contraintes, il faut évidement pouvoir relacher ce système de contraintes. C'est pourquoi, l'heuristique d'alignement nécessite l'introduction d'un point de choix. Pour plus de souplesse dans la conception de la stratégie, ce point de choix est implanté grâce à une variable booléenne décorant les arêtes du Graphe de Flot de Données :

$$\left(\begin{array}{rcl} P^r &=& P^w \\ I^r L^r &=& I^w L^w \end{array} \right) \Leftrightarrow Alignement(r, w)$$

Cette méthode homogénéise les stratégies de recherches car on dispose des paramètres qui controlent l'activation des heuristiques. De plus, cette technique aligne les données uniquement sur quelques arêtes du graphe.

Cette heuristique tend à réduire le parallélisme sur les différents nœuds du graphe de flot de données. Expérimentalement, cet effet se propage sur l'ensemble du Graphe de Flot de Données. Il faut donc intuitivement combiner cette heuristique avec une maximisation du parallélisme.

2.1.4 Recouvrement des communications

Nous donnons dans cette section une heuristique qui tend à **minimiser le coût des communications**. Elle consiste à favoriser les solutions pour lesquelles les communications sont recouvertes par

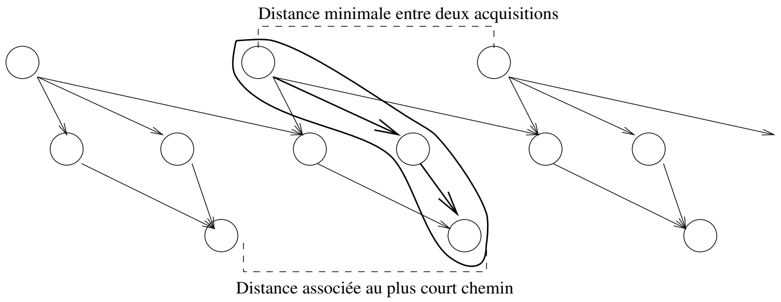


FIG. 13.1. – *Plus court chemin et distance dans un ordonnancement*

les calculs. Il suffit donc de s'appuyer sur la modélisation (chap. 10, § 4.2, p. 168) pour implanter cette heuristique :

$$cout(V_{te}^t) < cout_{elem}^t \Leftrightarrow revouvert(t)$$

Nous utilisons ici la même méthode qu'en (§ 2.1.3) afin de pouvoir contrôler le recouvrement d'un traitement ou non.

2.2 Ordonnancement

Cette section utilise la définition de l'ordonnancement événementiel (chap. 7, § 2, p. 2). Cette heuristique évite de calculer et allouer des données à l'avance en orientant la recherche de solutions vers un **ordonnancement "au plus tôt"**. Cet ordonnancement permet de surcroît une génération de code simple car il autorise la réutilisation d'un unique tampon en lecture pour chaque nid de boucle.

L'heuristique d'ordonnancement utilise la distance minimale de calcul d'un résultat sur l'ensemble du graphe de flot de données. Ceci correspond à la distance minimale entre l'exécution d'un traitement d'entrée et d'un traitement résultat. Celà signifie que cette heuristique est basée sur la distance minimale de dépendance et sur la contrainte du plus court chemin sur l'ensemble des blocs de calculs dépendant du graphe (cf. 13.1).

Pour établir cette heuristique globale (il concerne en effet une sous-chaîne du DFG), on doit donc se baser sur la distance de dépendance minimale (chap. 7, § 5.1.3, p. 141), ainsi que sur un calcul de plus court chemin sur un graphe de flot de données dont les arêtes sont valuées par les distances minimales et les sommets ont les macro-instructions des nids de boucles. En procédant à la même démarche que pour le plus long chemin (chap. 10, § 5.1, p. 169), on obtient le même type de contrainte implantable :

$$\bigwedge D(\mathbf{i}^w, \mathbf{i}^r) \Longrightarrow \underline{\Lambda} = \sum_w \Big(\min_r (d_{min}^{w,r}) \Big) \tag{13.1}$$

On rappelle ici la contrainte de distance minimale (p. 141) :

$$d_{min}^{w,r} \;=\; min\ \{\ d(c_s^r) - d(c_s^w),\ \mathcal{D}(\mathbf{c_s^w},\ \mathbf{c_s^r})\ \} \tag{13.2}$$

L'heuristique de l'ordonnancement au plus tôt s'exprime alors par :

Heuristiques	**Conséquences**	**Contraintes**
Maximiser le parallélisme	pénalise les communications, augmente la puissance de calcul	Intanciation des variables P_{ll}^k
Alignement des données	maximise la localité, diminue le parallélisme	$P^r = P^w$ et $I^r L^r = I^w L^w$
Recouvrement des communications	minimise le côût des communications	$cout(V_{te}^t) < cout_{elem}^t \Leftrightarrow revouvert(t)$
Ordonnancement “au plus tôt”	permet une génération de code simple, autorise la réutilisation d'un unique tampon en lecture pour chaque nid de boucle, améliore la latence	$\alpha_1^0 \geq \underline{\Lambda} \Leftrightarrow pluscourt$

FIG. 13.2. – *Les différentes heuristiques et leurs conséquences sur le type de solutions recherchées*

$$\alpha_1^0 \geq \underline{\Lambda} \Leftrightarrow pluscourt \tag{13.3}$$

où α_1^0 correspond à la boucle la plus externe du premier nid de la sous chaîne concernée. De cette manière, le système favorise les solutions pour lesquelles uniquement les calculs nécessaires apparaissent dans l'intervalle minimal de dépendance. La version implantée de cette heuristique considère la chaîne linéaire dans son intégralité.

3 Contraintes redondantes

Les contraintes redondantes, ou modèles redondants sont d'autres aspects de la modélisation. Grâce à ces derniers, la résolution peut jouer sur plusieurs modèles représentant un même concept. La modélisation actuelle ne fait pas explicitement apparaître ce type de modèle, puisque nous avons un seul modèle par fonctionnalité (partitionnement, ordonnancement, *etc...*). Cependant, il y a plusieurs voies d'investigation. Il est prévu d'implanter plusieurs de ces contraintes et modèles redondants, en voici deux exemples :

Bornes inférieures des communications : Cette contrainte redondante à été définie lors du chapitre sur l'interphase (chap. 6, § 2.5.1, p. 119).

Modèle redondant Temps-réel : Le modèle redondant temps réel résulte d'une expression des experts de Thomson-CSF : Il s'agit d'exprimer la contrainte de latence directement en fonction des contraintes d'acquisition. Ceci n'est pas le cas dans la modélisation actuelle puisque la latence est exprimée à l'aide de la contrainte de cardinalité. Nous obtiendrions ainsi un meilleur couplage entre les contraintes d'entrées-sorties et de temps réel.

Les contraintes redondantes, à définir sur la modélisation concurrente, sont l'objet de travaux futurs.

4 Stratégies globales

Les stratégies, comme nous l'avons déjà expliqué, définissent l'ordre dans lequel les variables du problème seront examinées. Cet ordre est important pour converger rapidement vers une solution. En effet, certaines variables sont plus contraintes que d'autres et leur instanciation entraîne une propagation importante sur les autres variables. C'est par exemple l'objectif de la stratégie de premier échec (connu en anglais sous le nom de "*firstfail*"). Dans notre cas, les stratégies peuvent être raffinées grâce à la connaissance du domaine du placement automatique. Cette connaissance autorise la maîtrise des variables-clefs du problème lorsque celui-ci est partiellement résolu.

4.1 Orientation des stratégies

Les stratégies globales sont tout d'abord orientées selon le type d'optimisation et doivent être mises en œuvre de façon à favoriser les solutions correspondantes au critère choisi. Le problème réside donc dans le choix des heuristiques ainsi que dans l'ordre dans lequel les variables sont à examiner. Ces décisions sont difficiles à prendre. Il existe plusieurs moyens de combiner les heuristiques. De plus, le nombre d'ordres possibles correspond au nombre de permutations de l'ensemble des variables du problème. De surcroît, la sélection des variables pertinentes est aussi déterminante. Enfin, la stratégie de recherche peut être dynamique, lorsque l'ordre des variables à examiner et l'activation des heuristiques dépendent de l'étape de résolution courante.

La connaissance du domaine, ne serait-ce que par la modélisation ainsi que les analyses d'expérimentations permettent de construire ces stratégies. Celles-ci reposent alors sur les trois piliers suivant :

Le placement automatique : La connaissance du domaine fournit des heuristiques *a priori*, qui seront invalidés ou validés par les expérimentations. Dans les heuristiques données précédement (§ 2), seules l'alignement des données, le recouvrement des communications et les dépendances en lecture ont été définis *a priori*. Les autres ont été déduites des expérimentations à grande échelle et sur les cas d'école.

La modélisation concurrente : En raisonnant sur les modèles, il est possible de donner un ordre de variable à examiner "par modèle". Cependant, il est préférable de conserver la concurrence entre les modèles en préférant un ordre entrelaçant les variables.

Les expérimentations : Celles-ci valident l'efficacité des heuristiques, permettent d'en déduire de nouvelles et éclairent le rôle de chaque variable du problème. On procède pour cela à des recherches de solutions partielles, ou certaines variables du problème sont fixées ou artificiellement contraintes. Les variables les plus discriminantes devront être examinées en premier selon le critère d'optimisation.

Nous donnons dans la section suivante une stratégie élaborée à partir des conclusions sur les cas d'écoles et sur des exemples de tailles réelles. Celle-ci illustre parfaitement le processus itératif de la définition des stratégies et donc le rôle des trois points abordés ci-dessus.

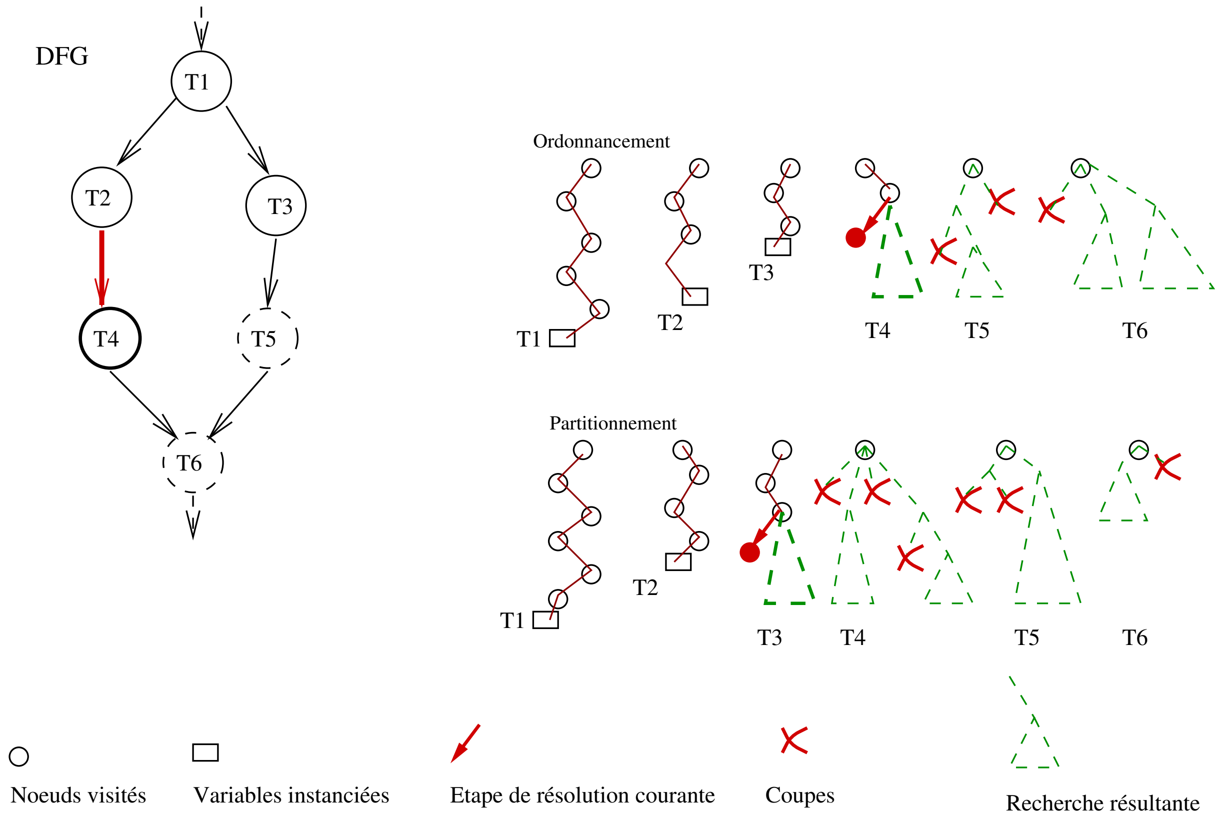


FIG. 13.3. – *Ordre des variables calqué sur la topologie du Dataflow Graph*

4.2 Description d'une stratégie statique basée sur la topologie du graphe de flot de données

Nous présentons dans cette section une stratégie statique dont l'ordre des variables à examiner est dérivé la topologie du graphe de flot de données. Cela permet d'entrelacer l'examen des variables des modèles de partitionnement et d'ordonnancement suivant un ordre qui n'est pas anodin vis-à-vis du problème de placement. En effet, une décision de placement sur un nœud du graphe engendre des contraintes sur les nœuds connexes (à travers la mémoire et les communications) (fig. 13.3), ce sont donc ces variables qu'il est alors préférable d'examiner. De plus, l'ordre de parcours des valeurs correspond à maximiser le parallélisme.

Une macro-étape de résolution correspond donc à un arc du graphe de flot de données w (représenté en gras sur la figure 13.3) et consiste en l'instanciation des variables P^w, L^w et α^w du nœud source, les variables du nœud r destination en lecture $up(\mathbf{c}^\mathbf{r})$ et β^r sont alors examinées (fig. 13.3). Cette stratégie propage des contraintes très tôt sur les nœuds finaux du DFG et permet d'éliminer un grand nombre de branches inutiles dans l'arbre de recherche. De plus, elle considère les variables déterminantes en premier, puisqu'il s'agit des plus gros traitements du signal qui se situe au début du graphe.

Une première série d'expérimentations a permis au système de trouver des solutions. L'inconvénient est que celles-ci n'étaient pas satisfaisantes vis-à-vis de la latence. L'heuristique du plus court chemin s'est révélé nécessaire pour trouver des solutions intéressantes selon les critères d'efficacité et de latence. De plus, ces solutions sont proches de celles données par les experts de Thomson-CSF. Cette heuristique est activée ou désactivée grâce au booléen introduit précédement (§ 2.2). Il suffit donc d'instancier ce paramètre à `vrai` ou `faux` pour permettre le contrôle de l'heuristique. Cette méthode rend la structure de la stratégie lisible et maintenable. En effet, cette structure est constamment sollicitée lors de la mise au point. Si les stratégies statiques sont facilement implantables en Programmation Logique avec Contraintes, les stratégies dynamiques demandent un effort de programmation conséquent car il faut pouvoir remmettre en question dynamiquement des branches de l'arbre de recherche. Ceci fera l'objet d'expérimentations supplémentaires et de travaux futurs.

5 Conclusion

Ce chapitre a mis en évidence la facilité avec laquelle il est possible de combiner les heuristiques et la modélisation au sein d'une stratégie de résolution globales. Pratiquement, les heuristiques sont établies à partir d'expérimentations. Néanmoins, la facilité de manipulation du système de placement, permet de raffiner itérativement les heuristiques afin d'accélérer la résolution. Les stratégies de résolution permettent la coordination et le contrôle de l'ensemble des composantes de la résolution.

Conclusion

L'implantation se présente donc en deux phases : l'une est la stricte représentation des modèles à l'aide de variables et de contraintes, l'autre est la réalisation de fonctions d'optimisation à l'aide de stratégies dédiées.

Cette présentation permet de comprendre à quel point l'implantation en programmation avec contraintes reflète les modèles et relations, lorsque ceux-ci sont définis en intention. Comme il n'existe pas de modélisation unique du placement global, il n'existe pas non plus d'implantation unique. Aussi, des choix doivent être parfois pris : un ensemble de variables peut être définit dans un modèle plutôt qu'un autre. Mais ces choix, au même titre que l'ordre dans lequel les contraintes sont posées, n'influent pratiquement pas la résolution. Nous avons donc choisi l'implantation la plus naturelle car la plus proche de la réalité physique en suivant la modélisation mathématique.

La programmation des fonctions d'optimisation est quand à elle fondée sur les stratégies et heuristiques. Les stratégies permettent de contrôler le choix des variables à examiner lors d'une étape de résolution, et définissent ainsi l'arbre de recherche. Les heuristiques permettent de sélectionner les valeurs pertinentes afin de maximiser la propagation dans l'arbre de recherche. Les stratégies statiques sont aisées à programmer tandis que les dynamiques, remettant en cause dynamiquement l'arbre de recherche, nécessitent des étapes de coordination difficiles à implanter dans un langage de programmation logique où l'on est prisonnié de la pile de retour-arrière.

Enfin, d'un point de vue utilisation, et grâce aux temps de réponse corrects, le système de placement automatisé permet non seulement de trouver des solutions optimisées mais aussi de manipuler des solutions partielles. Dans ce sens, il se rapproche d'un outil de prototypage virtuel. Cet aspect est enrichit par la modularité de l'approche qui autorise la spécialisation, la réutilisation, la maintenabilité et l'évolution du système.

Cinquième partie

Expérimentation et Validation

Introduction

La phase d'implantation a nécessitée de nombreux tests et essais permettant de valider ou invalider les modèles et relations. Diverses contraintes, ou anomalies ont d'ailleurs été détectées à l'aide de ce que l'on appelle les cas d'école. Ceux-ci se composent à une échelle réduite de différents problèmes caractéristiques rencontrés en traitement du signal (recouvrements en lecture, accès modulo,...). De ce fait, ils sont utilisés pour vérifier les solutions et tester les réactions des modèles (chap. 14), sans avoir recours aux heuristiques. Ces cas d'école ont ici une fonction pédagogique, facilitant l'explication des solutions et de leur profil, comme les réactions du système lors de l'optimisation. Avec cet ensemble d'essais sur les cas d'école s'achève en fait la partie réalisation, tout en connaissant déjà certaines conséquences liées au comportement des modèles. Cette démarche expérimentale a permis en fait la réalisation de stratégies et l'utilisation à bon escient des heuristiques.

Il est donc nécessaire de valider notre approche sur plusieurs exemples d'échelle réelle (chap. 15), présentant une combinatoire bien plus importante. Nous avons selectionné des jeux d'essais représentatifs de la couverture applicative dans le cadre des restrictions exposées (chap. 2, 3.1.5). Si l'approche globale est solide, cette série d'essais permet d'affiner les stratégies, contraintes redondantes et heuristiques utilisés afin que l'actuel prototype de système de placement devienne un outil fiable.

La partie sur les exemples industriels fait encore l'objet d'expérimentations et sera remise ultérieurement

Chapitre 14

Étude et analyse du comportement du système de placement sur des cas d'école

1 Introduction

Afin de faciliter la compréhension et la vérification manuelle des résultats produit par chacun des modèles comme la satisfaction des dépendances, l'allocation mémoire ou les communications de données, nous introduisons un jeu de "*cas d'école*". Ils nous ont aussi permis de comprendre et d'analyser la réaction des procédures de résolution lors de l'optimisation sur des fonctions de coûts variées ainsi que les diverses interactions entre les modèles.

Dans ce chapitre, les trois cas d'écoles considérés sont : `pres`(§ 2), `pres_rec`(§ 3), `pres_mod`(§ 4). Le premier est très simple et permet d'identifier le comportement de l'optimisation. Les deux autres présentent des particularités applicatives. En effet, `pres_rec` permet de savoir si les effets recherchés au niveau du modèle mémoire sont obtenus, et `pres_mod` permet de savoir si les accès modulo sont effectivement pris en compte et quels en sont les conséquences sur les modèles lors de l'optimisation.

Les résultats donnés dans ce chapitre sont issus du système de placement fonctionnant sur SUN-SPARC10. Ces résultats sont obtenus SANS AUCUNE HEURISTIQUE. Lorsque les données sont alignées ou les communications recouvertes par les calculs, c'est UNIQUEMENT le fruit de l'algorithme de "*Branch & Bound*" et de la modélisation.

Nous présentons dans cette partie les résultats obtenus à l'aide de trois différentes fonctions d'optimisation. Nous rappelons que si un paramètre n'est pas un argument de la fonction d'optimisation, il reste nénamoins présent en tant que contrainte. Dans un premier temps, nous considérerons des contraintes initiales "lâches" (affaiblies) afin de comprendre le comportement du système vis à vis de la fonction d'optimisation, sans interférence avec d'autres paramètres. Dans un deuxième temps, nous donnons des résultats d'optimisation avec des contraintes initiales fortes.

Voici les critères d'optimisation considérés :

La latence

La latence est le temps maximal séparant un traitement d'acquisition et un traitement de sortie dépendants (chap. 10, p. 10). C'est donc un paramètre primordial de l'aspect temps réel du placement.

La taille mémoire

L'optimisation d'une ressource du système comme la mémoire est caractéristique des objectifs des programmeurs de traitement du signal (chap. 8, p. 8).

L'efficacité

L'efficacité de la solution est un critère opérationnel classique mesurant la qualité de la solution. C'est en fait le rapport du nombre de cycles utilisés sur l'architecture parallèle par rapport au nombre de cycles nécessaires sur un seul processeur. Ce rapport est très proche du "*speed-up*" dénotant du gain obtenu par le parallélisme. En pratique, dans notre problème, cela correspond au nombre de cycles perdus à communiquer entre processeurs. Le nombre de cycles peut être celui nécessaire à l'exécution d'une période de traitement, le terme est alors l'efficacité par rapport à la période. Nous choisissons ici de définir l'efficacité par rapport à la latence, car ce dernier paramètre est essentiel dans les applications temps-réel. Dans ce cas, il faut considérer le nombre de cycles nécessaires entre la production d'un résultat et l'acquisition la plus ancienne dépendante (chap. 10).

Plan du chapitre

Les sections (§ 2,3,4) présentent les cas d'école dans un contexte d'optimisation sans d'autres paramètres fortement contraints. En revanche, nous décrivons ce type d'optimisation en (§ 5).

2 Optimisation de la latence sur un cas simple (`pres`)

Ce premier cas d'école est une application simple composée de trois tâches mais ayant une dimension temporelle infinie. Nous allons tout d'abord en donner une description (§ 2.1) avec son code associé en Pseudo-Fortran. Nous donnons (§ 2.2.1) la requête initiale, puis nous décrirons (§ 2.2.2, ..., 2.2.5) les différentes solutions trouvées. Enfin, la dernière section (§ 2.3) présente l'analyse du comportement du système de placement selon ce cas d'école.

2.1 Description de l'application `pres`

La première tâche calcule le carré de chaque élément du tableau `TAB1` qui possède une dimension infinie et une dimension de 8 éléments. Le résultat est écrit dans le tableau `TAB12`. La seconde tâche calcule la différence entre deux éléments successifs (selon la dimension infinie, et sans dépendance lecture-lecture) du tableau `TAB12` et écrit le résultat dans le tableau `TAB23`. La dernière tâche effectue une somme (intégration) sur 4 éléments consécutifs (selon la dimension finie) du tableau `TAB23` et écrit le résultat dans le tableau `TAB3`. Tous les éléments de tableaux sont ici des entiers codés sur deux mots (fig. 14.1).

```
DO T=0,INFINI
  DO I=0,3
    TAB12[T,I]=TAB1[T,I]*TAB1[T,I]
  ENDDO
ENDDO

DO T=0,INFINI
  DO I=0,3
    TAB23[T,I]=TAB12[2*T,I]-TAB12[2*T+1,I]
  ENDDO
ENDDO

DO T=0,INFINI
c PARTIE INTRA PROCEDURALE
  S=0
  DO MI=0,3
    S=S+TAB23[T,MI]
  ENDDO
  TAB3[T]=S
c FIN PARTIE INTRA PROCEDURALE
ENDDO
```

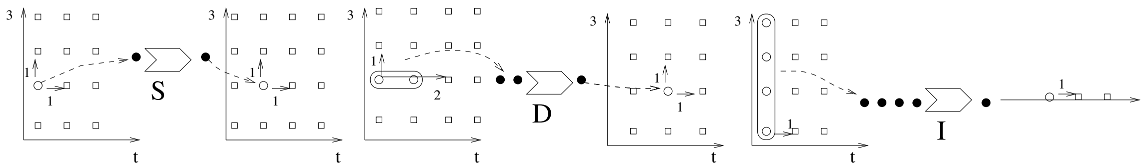


FIG. 14.1. – *Code Pseudo-Fortran de l'application* `pres`

2.2 Optimisation de la latence

La requête consiste ici à minimiser la latence de l'application sous les contraintes spécifiées dans la requête. Le système optimise ce critère en parcourant l'espace de solution. Le résultat est donc une suite de solutions pour lesquelles le critère de latence est décroissant.

2.2.1 Description de la requête : minimisation de la latence avec 4 processeurs disponibles

Critère à optimiser	MINIMISATION DE LA LATENCE
Architecture type	SPMD
Nombre de processeurs disponibles	4
Capacité mémoire disponible	800 (éléments)
Latence applicative	60 cycles

2.2.2 Solution détaillée No 1

Dans cette solution, le système a choisi d'aligner les dimensions finies de tableaux afin d'éviter les communications. On remarque le mauvais taux d'efficacité de la solution. Celui-ci est du à la réduction sur un seul processeur du dernier traitement. L'ordonnancement obtenu ne favorise ni la latence ni la mémoire car entre un traitement d'entrée et un traitement de sortie, beaucoup de calculs inutiles sont présents. Ils allongent le temps de latence et nécessite une allocation.

Partitionnement À chaque nid de boucles est associées deux matrices L et P, dénotant respectivement de la localité et du parallélisme. Dans cette première solution, le système utilise un parallélisme maximal sur les dimensions finies, pour chacun des nids de boucles. Le calul s'achève sur un seul processeur.

NOMBRE DE PROCESSEURS NÉCESSAIRES : 4

	integ	diff	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel Ce tableau donne les solutions d'ordonnancement événementiel pour chaque nid de boucles (fig. 14.2).

	integ	diff	carre
$\beta^t =$	0	2	3
$\alpha^t =$	${}^t(1, 1)$	${}^t(2, 1)$	${}^t(2)$

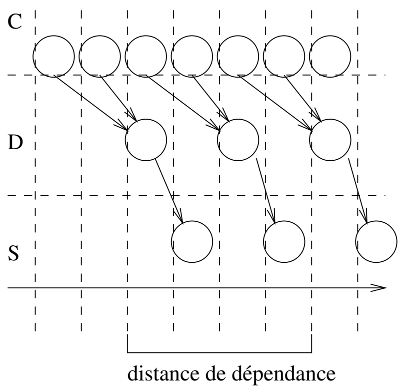


FIG. 14.2. – *Ordonnancement par processeur*

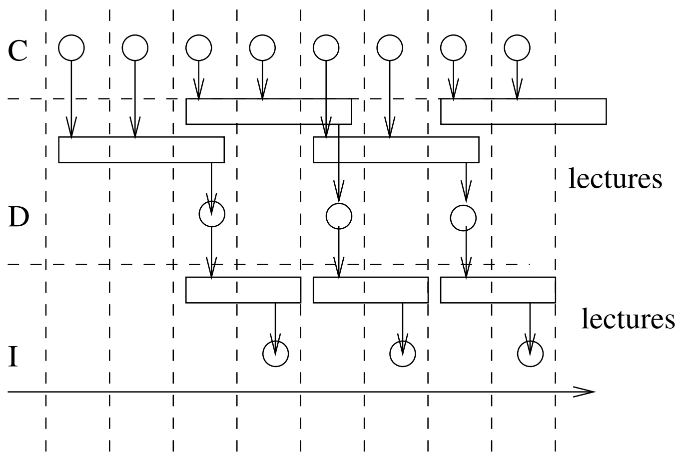


FIG. 14.3. – *Mémoire par proceseur*

Capacité mémoire On considère ici les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application. (La première tâche écrit, la seconde lit). Ce tableau donne la distance de dépendance permettant de comptabiliser le nombre d'événements. Le partitionnement engendre des blocs de tailles respectives 4 et 8 éléments[1], pour les 2 derniers nids de boucles. Les blocs d'acquisition (i.e. issus du premier nid de boucles) ne sont en effet pas considérés par le modèle mémoire. Or, l'ordonnancement choisit nécessite d'allouer en mémoire 2 blocs en lecture pour le nid de boucles `diff` et un bloc pour le nid `somme` (fig. 14.3), ainsi qu'un tampon en écriture de taille 2×2 (chap. 8), le tout pour chaque processeur. Si l'on applique la contrainte de mémoire, on obtient :

$$(2 \times 4 + 1 \times 8 + 2 \times 2) \times 4 = 80$$

Ainsi, le système considère l'allocation de blocs de données dont l'utilisation est différée.

Capacité mémoire totale utilisée 80 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre → diff)	2	2	2
(diff → integ)	1	1	1

Dans les expérimentations qui suivent, nous ne donnerons plus la taille des blocs engendrés par le partitionnement. En revanche, nous spécifierons si ce modèle influence la taille des blocs.

Caractérisation des communications On considère ici encore les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application (La première tâche écrit, la seconde lit). Les partitionnements des deux premiers nids de boucles entraînnent un alignement, il n'y a donc pas de communication sur cet arc. En revanche, il faut regrouper toutes les données présentes sur les trois autres processeurs pour effectuer la somme sur quatre éléments. Les alignements sont décrits par des pointillés, les communications par des traits pleins sur la figure (14.4).

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre → diff)	0	1	12	0
(diff → integ)	2	1	3	6

Signal Le tableau ci-dessous résume à la fois la distance événementielle maximale de dépendance (fig. 14.2), selon laquelle la contrainte de cardinalité évalue un nombre d'événements nécessaire pour chaque nid de boucle. Ceci permet de calculer la latence.

Temps apériodique calculé = 11 (cycles)
Temps périodique = 10 (cycles)
Longueur du chemin critique = 3
Efficacité de la solution (selon la latence): = 0,16
Latence obtenue = 20 (cycles)

[1]Les éléments de tableaux sont des complexes, chacun d'entre eux sont codés sur généralement sur des "mots" mémoires (usuellement 2×32 bits)

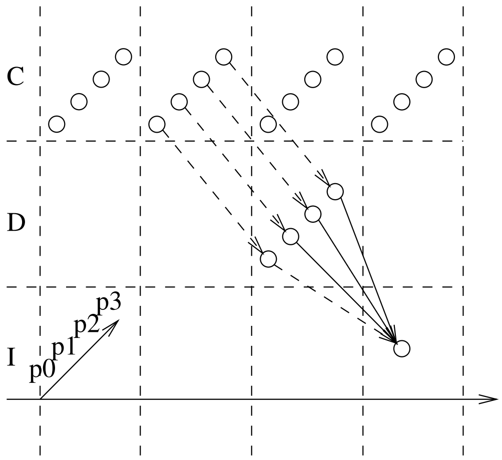


FIG. 14.4. – *Communications entre processeurs*

	integ	diff	carre
coût élémentaire	1	1	1
volume de calculs	1	7	1
nombre d'événements	4	2	2
temps physique	4	14	2

2.2.3 Solution détaillée No 2

Dans cette solution, le système améliore l'ordonnancement en augmentant les valeurs des α_i, ce qui a pour effet de faire disparaître les traitements inutiles entre calculs dépendants. C ela permet de réduire la latence ainsi que le volume mémoire. En revanche, les communications ne sont pas affectées.

Partitionnement Le partitionnement des nids de boucles est ici inchangé.

Ordonnancement événementiel Cet ordonnancement permet de réduire les nombre d'événements dans la distance de dépendance, pour le modèle du signal.

	integ	diff	carre
$\beta^t =$	0	3	4
$\alpha^t =$	${}^t(2,1)$	${}^t(4,1)$	${}^t(4)$

Capacité mémoire On considère ici les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application. (La première tâche écrit, la seconde lit).

CAPACITÉ MÉMOIRE TOTALE UTILISÉE 64 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre $\rightarrow$ diff)	3	1	1
(diff $\rightarrow$ integ)	1	1	1

Caractérisation des communications Les communications sont inchangées.

Signal Le nombre d'événements chutant dans la distance de dépendance : la latence est optimisée.

Temps apériodique calculé = 10 (cycles)
Temps périodique = 10 (cycles)
Longueur du chemin critique = 4
Efficacité de la solution (selon la latence): = 0,17
Latence obtenue = 19 (cycles)

	integ	diff	carre
coût élémentaire	1	1	1
volume de calculs	1	7	1
nombre d'événements	3	2	2
temps physique	3	14	2

2.2.4 Solution détaillée No 3

Dans cette solution, tous les paramètres changent, car il n'y a plus que 2 processeurs. Le fait de considérer deux processeurs permet de communiquer beaucoup moins de données et ainsi de gagner en latence. Comme il le sera confirmé en (§ 4), cela diminue aussi la mémoire allouée, mais ce n'est pas le critère considéré.

Partitionnement Parallélisme et localité sont exploités sur la dimension finie. Le nombre de processeurs chute à 2.

Nombre de processeurs nécessaires : 2

	integ	diff	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel La solution d'ordonnancement est inchangée.

Capacité mémoire Capacité mémoire totale utilisée 64 (mots mémoire)

La capacité mémoire est changée uniquement à cause du partitionnement. C'est donc les volumes de données qui sont affectés et non le nombre de blocs issus de l'ordonnancement.

Caractérisation des communications Comme, il n'y a plus que deux processeurs, il n'y a plus qu'un message.

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre $\rightarrow$ diff)	0	1	2	0
(diff $\rightarrow$ integ)	2	1	1	2

Signal Les paramètres de ce modèle ont changés à cause des volumes de calcul modifiés.

Temps apériodique calculé $= 11$ (cycles)
Temps périodique $= 10$ (cycles)
Longueur du chemin critique $= 4$
Efficacité de la solution (selon la latence): $= 0,36$
Latence obtenue $= 18$ (cycles)

	integ	diff	carre
coût élémentaire	1	1	1
volume de calculs	2	5	1
nombre d'événements	3	2	2
temps physique	6	10	2

2.2.5 Solution détaillée No 4

Dans cette solution optimale, l'énumérateur trouve à nouveau 4 processeurs, en effet l'ordonnancement optimal (où les calculs sont effectués dès que les données sont présentes) permet de tirer parti des 4 processeurs.

Partitionnement On retrouve les mêmes partitionnements qu'au début de l'optimisation.
Nombre de processeurs nécessaires : 4

	integ	diff	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel L'ordonnancement ne laisse plus apparaître que les calculs nécessaires et suffisant dans la distance de dépendance (fig. 14.5), optimisant ainsi le modèle du signal.

	integ	diff	carre
$\beta^t =$	0	4	5
$\alpha^t =$	${}^t(3,1)$	${}^t(6,1)$	${}^t(6)$

Nous donnons en (fig. 14.6) la structure du code associée à la solution. L'alignement entre le traitement `carre` et le traitement `diff` a permis de paralléliser d'avoir une seule boucle parallèle englobant les deux traitements. Le résultat de `carre` est communiqué dès que possible du processeur `I` au processeur `0`. Le reste des traitements est séquentiel. De plus les tableaux `TAB12(2,4), TAB1(2,4)` ne comportent plus que des dimensions finies et `TAB23(4)` est en fait un tableau de 4 éléments.

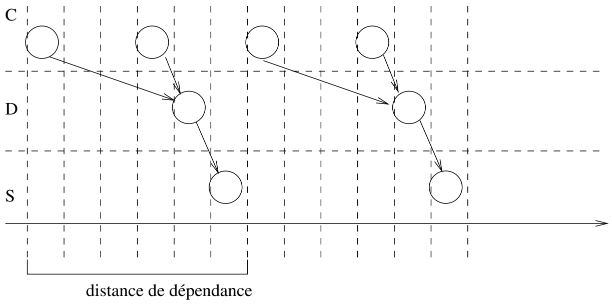


FIG. 14.5. – *Ordonnancement par processeur*

```
DO T=0,INFINI
  DOALL I=0,3
    DO T1=0,1
      TAB12[T1,I]=TAB1[T1,I]*TAB1[T1,I]
    ENDDO
    TAB23[I]=TAB12[0,I]-TAB12[1,I]
    COMMUNICATIONS(I,0,TAB23[I])
  ENDDO
  S=0
  DO MI=0,3
    S=S+TAB23[MI]
  ENDDO
  RESULTAT=S
ENDDO
```

FIG. 14.6. – *Structure du code*

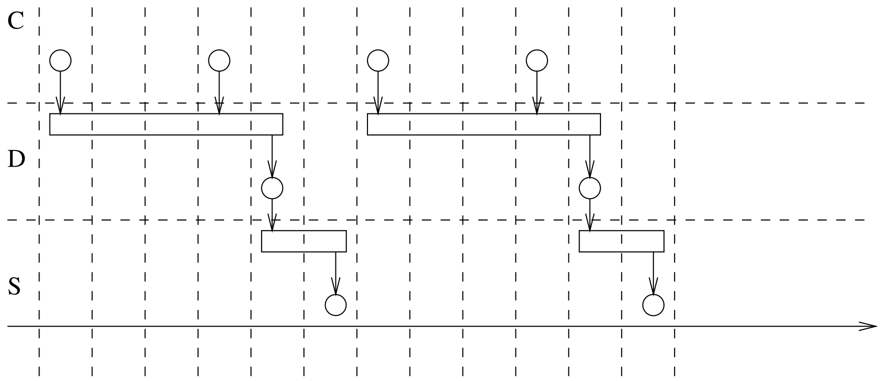


FIG. 14.7. – *Mémoire par processeur*

Capacité mémoire Comme il n'y a plus que les calculs nécessaires et suffisant dans la distance de dépendance, ce n'est plus la peine d'allouer des données dont l'utilisation est retardée (14.7).

CAPACITÉ MÉMOIRE TOTALE UTILISÉE 64 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre → diff)	4	1	1
(diff → integ)	1	1	1

Caractérisation des communications On considère ici les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application. (La première tâche écrit, la seconde lit).

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre → diff)	0	1	12	0
(diff → integ)	2	1	3	6

Signal Ce modèle est optimisé pour les mêmes raisons que la mémoire : uniquement les calculs nécessaires et suffisant sont présent dans la distance de dépendance.

TEMPS APÉRIODIQUE CALCULÉ = 9 (cycles)
TEMPS PÉRIODIQUE = 10 (cycles)
Longueur du chemin critique = 5
EFFICACITÉ DE LA SOLUTION (SELON LA LATENCE): $= 0,32$
LATENCE OBTENUE = 10 (cycles)

	integ	diff	carre
coût élémentaire	1	1	1
volume de calculs	1	7	1
nombre d'événements	2	1	1
temps physique	2	7	1

2.3 Synthèse et analyse de l'optimisation de la latence

Le récapitulatif donne la synthèse des résultats obtenussous forme de tableau, on procède alors à une analyse globale de l'optimisation.

2.3.1 Récapitulatif

Le tableau suivant résume les solutions trouvées par le système. La solution No. 0 correspond aux contraintes initiales portant sur les paramètres de la recherche. On remarque alors que certaines valeurs sont notée nc, car ne peuvent pas être calculées lors de l'initialisation. La solution No. 1 correspond à la première solution trouvée, les solutions No. 2,3 sont intermédiaires, alors que la solution No. 4 correspond à la solution optimale.

Tableau de synthèse							
Critère à optimiser : minimisation de la latence							
no	Nb proc.	Mémoire éléments	Chemin critique évts	Latence cycles	Temps Apériodique cycles	Temps Périodique cycles	Efficacité %
0	4	800	nc	60	nc	nc	nc
1	4	80	3	20	11	10	16
2	4	64	4	19	10	10	17
3	2	40	4	18	11	10	36
4	4	64	5	10	9	10	32

2.3.2 Analyse globale

Dans cette expérience :

- Le partitionnement reste identique, sauf pour la solution No 3 qui présente moins de communications,
- L'ordonnancement a une influence tant sur la latence que sur la mémoire. En effet, l'ordonnancement optimal est ici un ordonnancement où chaque partition est exécutée dès que les données requises sont disponibles, c'est à dire *au plus tôt*, ce qui privilégie la localité. On a donc bien dans cette expérience minimisé la latence, avec comme effet de bord une minimisation de la capacité mémoire.
- La latence optimale est égale à la période. Ceci est possible parce que le graphe de dépendance est non connexe. Durant une période on ne traite qu'une composante connexe du graphe : la latence doit être inférieure ou égale à la période. De plus, la latence correspond au nombre d'événements présents dans le plus long chemin d'une composante connexe. Au moins tous les éléments dépendants sont présents puisque les dépendances sont satisfaites. La latence doit donc être supérieure ou égale à la période. Nous en concluons donc que la latence égale la période dans le cas optimal sur cet exemple.
- On remarque aussi l'influence des communications sur les partitionnements : entre la solution No 2 et No 3, le système joue sur le partitionnement pour réduire les communications. En

revanche, on note les mauvais "*scores*" obtenus pour l'efficacité. Ceci est dû au traitement `somme` s'effectuant sur un seul processeur.

3 Optimisation de la latence sur un cas comportant un accès modulo (`pres_mod`)

Ce second cas d'école est une adaptation de `pres` pour introduire les dépendances lecture-lecture et les modulos. La structure de cette section est identique à la précédente.

3.1 Description de l'application `pres_mod`

La première tâche calcule le carré de chaque élément du tableau `TAB1` qui possède une dimension infinie et une dimension de 8 éléments. Le résultat est écrit dans le tableau `TAB12`. La seconde tâche cluster calcule la différence entre deux éléments successifs (au sens de la dimension infinie) du tableau `TAB12` tout en faisant la somme des termes calculés sur une fenêtre tournante de dimension 4 (dans le sens de la dimension finie). Il y a donc un accès modulo en lecture dans cette seconde tâche. Finalement le résultat est écrit dans le tableau `TAB23`. La dernière tâche effectue une somme (intégration) sur 4 éléments consécutifs (selon la dimension finie) du tableau `TAB23` et écrit le résultat dans le tableau `TAB3` (fig. 14.8). Tous les éléments de tableaux sont ici se composent de deux mots.

3.2 Optimisation de la latence

La requête consiste ici à minimiser la latence de l'application sous les contraintes spécifiées dans la requête. La contrainte de latence initiale est étroite, permettant ainsi une partie de la recherche redondante avec la partie précédente.

3.2.1 Description de la requête : minimisation de la latence avec 4 processeurs disponibles

Critère à optimiser	MINIMISATION DE LA LATENCE
Architecture type	SPMD
Nombre de processeurs disponibles	4
Capacité mémoire disponible	200 (éléments)
Latence applicative	12 cycles

3.2.2 Solution détaillée No 1

Le fait d'utiliser un maximum de processeurs engendre, à cause de l'accès modulo 4, de nombreuses communications entre les nids `carre` et `cluster`. L'optimisation consistera à diminuer ces communications. La contrainte initiale de 12 cycles permet de trouver directement un ordonnancement optimal vis-à-vis de la latence et de la mémoire.

Partitionnement On décrit ici tâche par tâche, tous les paramètres associés au partitionnement. Le système utilise encore le parallélisme sur la dimension finie.
NOMBRE DE PROCESSEURS NÉCESSAIRES : 4

```
c CARRE

DO T=0,INFINI
  DO I=0,7
    TAB12[T,I]=TAB1[T,I]*TAB1[T,I]}
  ENDDO
ENDDO

c DIFFERENCE

DO T=0,INFINI
  DO I=0,3
  c PARTIE INTRA PROCEDURALE
  S=0
  DO MI=0,3
    S=S+TAB12[2*T,I+MI MOD 4]-TAB12[2*T+1,I+MI MOD 4]
  ENDDO
  TAB23[T,I]=S
  c FIN PARTIE INTRA PROCEDURALE
  ENDDO
ENDDO

c SOMME

DO T=0,INFINI
  c PARTIE INTRA PROCEDURALE
  S=0
  DO MI=0,3
    S=S+TAB23[T,MI]
  ENDDO
  TAB3[T]=S
  c FIN PARTIE INTRA PROCEDURALE
ENDDO
```

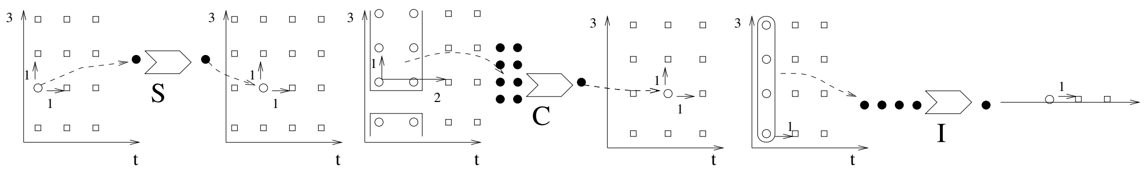


FIG. 14.8. – *Code Pseudo-Fortran de l'application* `pres_mod`

	integ	cluster	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel On décrit ici tâche par tâche, tous les paramètres associés à l'ordonnancement événementiel. Etant donné la contrainte initiale de latence, l'ordonnancement trouvé est directement "au plus tôt".

	integ	cluster	carre
$\beta^t =$	0	4	5
$\alpha^t =$	${}^t(3, 1)$	${}^t(6, 1)$	${}^t(6)$

Capacité mémoire On considère ici les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application (La première tâche écrit, la seconde lit).

Capacité mémoire totale utilisée 112 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre $\rightarrow$ cluster)	4	1	1
(cluster $\rightarrow$ integ)	1	1	1

Caractérisation des communications On considère ici les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application (La première tâche écrit, la seconde lit). On note ici l'effet du modulo sur les communications (fig. 14.9), par rapport aux résultats du premier cas d'école `pres`.

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre $\rightarrow$ cluster)	2	1	12	24
(cluster $\rightarrow$ integ)	2	1	3	6

Signal Temps apériodique calculé = 10 (cycles)
Temps périodique = 11 (cycles)
Longueur du chemin critique = 5
Efficacité de la solution (selon la latence): $= 0,30$
Latence obtenue = 11 (cycles)

	integ	cluster	carre
coût élémentaire	1	1	1
volume de calculs	4	2	1
nombre d'événements	2	1	1
temps physique	8	2	1

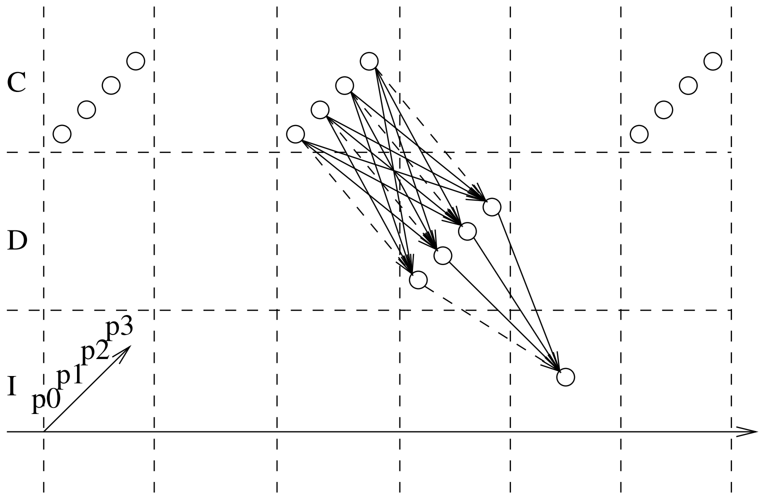


FIG. 14.9. – *Communications entre processeurs*

3.2.3 Solution détaillée No 2

Le partitionnement de cette solution à changé de manière à éviter au maximum les communications impliquées par l'accès modulo. Ainsi, le traitement intermédiaire n'est plus placé que sur deux processeurs.

Partitionnement Le système a préféré réduire les communications en diminuant le parallélisme afin d'optimiser la latence.
NOMBRE DE PROCESSEURS NÉCESSAIRES : 4

	integ	cluster	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel L'ordonnancement de cette solution est inchangé.

Capacité mémoire La mémoire de cette solution change à cause du partitionnement qui est différent, en revanche le nombre de blocs est inchangé, au même titre que l'ordonnancement.
CAPACITÉ MÉMOIRE TOTALE UTILISÉE 128 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre $\rightarrow$ cluster)	4	1	1
(cluster $\rightarrow$ integ)	1	1	1

Caractérisation des communications Le fait de réduire les communications entre les deux premiers nids de boucles permet de diminuer considérablement les communications dues aux accès modulos.

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre $\rightarrow$ cluster)	2	1	6	12
(cluster $\rightarrow$ integ)	2	1	1	2

Signal La diminution des communications permet d'améliorer le signal puisqu'il y a moins de communications non recouvertes.
TEMPS APÉRIODIQUE CALCULÉ = 9 (cycles)
TEMPS PÉRIODIQUE = 10 (cycles)
Longueur du chemin critique = 5
EFFICACITÉ DE LA SOLUTION (SELON LA LATENCE): $= 0,32$
LATENCE OBTENUE = 10 (cycles)

	integ	cluster	carre
coût élémentaire	1	1	1
volume de calculs	3	3	1
nombre d'événements	2	1	1
temps physique	6	3	1

3.2.4 Solution détaillée No 3

On assiste ici à une réduction du parallélisme à nouveau sur les deux derniers traitements. C'est la solution que choisi le système pour aligner les données des deux derniers traitements sur un seul processeur. De cette façon, il n'y a plus de réduction pour le dernier traitement.

Partitionnement La suppression du parallélisme sur le deuxième nid de boucle permet au système d'oter les communications entre les deux derniers traitement et de réduire considérablement la communication entre les deux premiers.
NOMBRE DE PROCESSEURS NÉCESSAIRES : 4

	integ	cluster	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel La solution d'ordonnancement n'a pas changée.

Capacité mémoire La mémoire augmente à cause du changement de partitionnement.
CAPACITÉ MÉMOIRE TOTALE UTILISÉE 160 (mots mémoire)

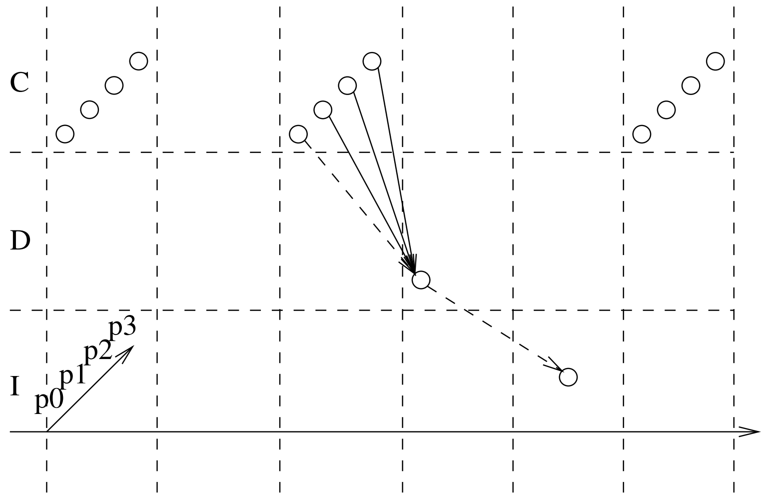


FIG. 14.10. – *Communications entre processeurs*

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre → cluster)	4	1	1
(cluster → integ)	1	1	1

Caractérisation des communications On considère ici les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application (La première tâche écrit, la seconde lit). Les communications sont réduites d'une part grâce à l'alignement entre les deux derniers traitement. D'autre part, la communication entre les deux premiers se résume à une réduction (fig. 14.10).

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre → cluster)	2	1	3	6
(cluster → integ)	0	1	0	0

Signal TEMPS APÉRIODIQUE CALCULÉ = 8 (cycles)
TEMPS PÉRIODIQUE = 9 (cycles)
Longueur du chemin critique = 5
EFFICACITÉ DE LA SOLUTION (SELON LA LATENCE): $= 0,36$
LATENCE OBTENUE = 9 (cycles)

	integ	cluster	carre
coût élémentaire	1	1	1
volume de calculs	2	4	1
nombre d'événements	2	1	1
temps physique	4	4	1

3.3 Synthèse et analyse de l'optimisation la latence

Nous présentons un tableau résumant l'ensemble des solutions obtenues ainsi que l'analyse des résultats sur les effets obtenus lors de l'optimisation de la latence.

3.3.1 Récapitulatif

Le tableau suivant résume les solutions trouvées. La solution No. 0 correspond aux contraintes initiales, c'est à dire à la requête spécifiant les contraintes. On remarque alors que certaines valeurs sont notée nc, car ne peuvent pas être calculées lors de l'initialisation. La solution No. 1 correspond à la première solution trouvée par le système, la solution No. 2 est une solution intermédiaire, alors que la solution No. 3 correspond à la solution optimale.

Tableau de synthèse							
Critère à optimiser : minimisation de la latence							
no	Nb proc.	Mémoire éléments	Chemin critique évts	Latence cycles	Temps Apériodique cycles	Temps Périodique cycles	Efficacité %
0	4	200	nc	12	nc	nc	nc
1	4	112	5	11	10	11	30
2	4	128	5	10	9	10	32
3	4	160	5	9	8	9	36

3.3.2 Analyse globale

Dans cette expérience, on remarque que le partitionnement change à cause des communications qui sont importantes au début de l'optimisation. Elles sont en fait dues à l'accès modulo. Le système cherche donc à réduire ce critères en diminuant le parallélisme. Là encore, l'ordonnancement a une influence tant sur la latence que sur la mémoire, mais à cause des contraintes initiales, il est directement optimal et ne change pas lors de l'optimisation. En effet, l'ordonnancement optimal est ici un ordonnancement qui privilégie la localité des données comme pour le cas de `pres` (§ 2.3), malgré un volume mémoire plus important. On a donc bien, dans cette expérience, minimisé la latence avec comme effet de bord une optimisaiton des communications.

4 Optimisation de la mémoire sur quatre nids de boucles (`pres_rec`)

Il s'agit d'une adaptation du premier cas d'école `pres` pour disposer des dépendances lecture-lecture sans accès modulo. Nous allons tout d'abord en donner une description (§ 4.1) ainsi que son code associé en Pseudo-Fortran. Nous allons ensuite présenter (§ 4.2.1) la requête initiale, puis nous décrirons (§ 4.2.2, ...,4.2.6) différentes solutions trouvées par la maquette dans sa recherche d'une solution optimale. Enfin, la dernière section (§ 4.3) analyse le comportement de l'optimisation sur cet exemple.

4.1 Description de l'application `pres_rec`

La première tâche calcule le carré de chaque élément du tableau `TAB1` qui possède une dimension infinie et une dimension de 8 éléments. Le résultat est écrit dans le tableau `TAB12`. La seconde tâche filtre (`fil`) calcule la somme des quatre éléments consécutifs sur la dimension finie du tableau `TAB12`, cette fenêtre se déplace en se recouvrant (dépendances lecture/lecture) suivant cette dimension. Le résultat est alors écrit dans le tableau `TAB23`. La troisième tâche `diff` calcule la différence entre deux éléments successifs (selon la dimension infinie) du tableau `TAB23` et écrit le résultat dans le tableau `TAB34` (fig. 14.11). La dernière tâche effectue une somme (intégration) sur 4 éléments consécutifs (selon la dimension finie) du tableau `TAB34` et écrit le résultat dans le tableau `TAB4`.

4.2 Optimisation de la mémoire

La minimisation de la mémoire illustre l'optimisation d'un paramètre d'architecture. La requête consiste ici à minimiser la mémoire de l'applicatiion sous les contraintes spécifiées dans la requête. Dans cette expérimentation, la latence est volontairement relachée pour connaıtre uniquement les effets de l'optimisation mémoire.

4.2.1 Description de la requête : minimisation globale de la mémoire avec 4 processeurs disponibles

Critère à optimiser	MINIMISATION GLOBALE DE LA MÉMOIRE
Architecture type	SPMD
Nombre de processeurs disponibles	4
Capacité mémoire disponible	800 (éléments)
Latence applicative	74 cycles

4.2.2 Solution détaillée No 1

Le système calcule un alignement entre deux traitements successifs. Là encore, le dernier traitement nécessite une réduction sur un processeur.

Partitionnement On décrit ici tâche par tâche, tous les paramètres associés au partitionnement. Les nids de boucles `diff` et `fil` sont parallèles sur la dimension finie.

NOMBRE DE PROCESSEURS NÉCESSAIRES : 4

	integ	diff	fil	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

```
c CARRE

DO T=0,INFINI
  DO I=0,7
    TAB12[T,I]=TAB1[T,I]*TAB1[T,I]
  ENDDO
ENDDO

c FILTRE

DO T=0,INFINI
  DO I=0,3
  c PARTIE INTRA PROCEDURALE
  S=0
  DO MI=0,3
    S=S+TAB12[T,I+MI]
  ENDDO
  TAB23[T,I]=S
  c FIN PARTIE INTRA PROCEDURALE
  ENDDO
ENDDO

c DIFF

DO T=0,INFINI
  DO I=0,3
    TAB34[T,I]=TAB23[2*T,I]-TAB23[2*T+1,I]
  ENDDO
ENDDO

c SOMME

DO T=0,INFINI
  c PARTIE INTRA PROCEDURALE
  S=0
  DO MI=0,3
    S=S+TAB34[T,MI]
  ENDDO
  TAB4[T]=S
  c FIN PARTIE INTRA PROCEDURALE
ENDDO
```

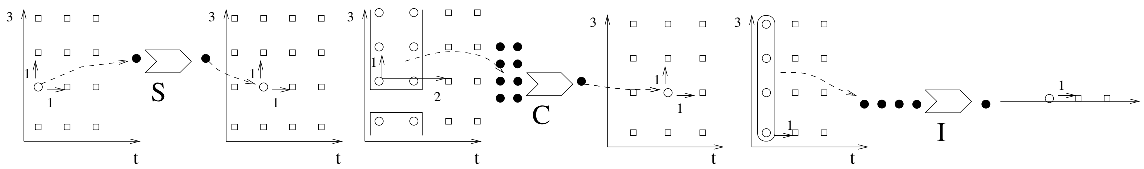


Fig. 14.11. – *Code Pseudo-Fortran de l'application* `pres_rec`

Ordonnancement événementiel On décrit ici tâche par tâche, tous les paramètres associés à l'ordonnancement événementiel. Par rapport aux constatations sur les autres optimisations, cet ordonnancement alloue plus d'informations que nécessaire.

	integ	diff	fil	carre
$\beta^t =$	0	1	3	4
$\alpha^t =$	${}^t(1,1)$	${}^t(1,1)$	${}^t(2,1)$	${}^t(2)$

Capacité mémoire On considère ici les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application (La première tâche écrit, la seconde lit). Le nombre de blocs de cette solution sont important, à cause d'un ordonnancement qui ne favorise pas l'utilisation des résultats intermédiaire au plus tôt.

Capacité mémoire totale utilisée 144 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre $\rightarrow$ fil)	1	2	2
(fil $\rightarrow$ diff)	2	2	2
(diff $\rightarrow$ integ)	1	1	1

Caractérisation des communications On considère ici les couples de tâches dépendantes, *i.e.* une arête du graphe acyclique décrivant l'application (La première tâche écrit, la seconde lit). Le fait de ne pas paralléliser le premier nid de boucle évite d'avoir une communication trop complexe entre les deux premiers nids de boucles.

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre $\rightarrow$ fil)	2	1	3	6
(fil $\rightarrow$ diff)	0	1	12	0
(diff $\rightarrow$ integ)	2	1	3	6

Signal Comme nous l'avons vu, la latence ne souffre pas beaucoup des communications mais surtout d'un ordonnancement qui est sous-optimal.

Temps apériodique calculé = 46 (cycles)

Temps périodique calculé = 21 (cycles)

Longueur du chemin critique = 4

Efficacité de la solution (selon la latence): = 0,12

Latence obtenue = 54 (cycles)

	integ	diff	fil	carre
coût élémentaire	1	1	1	1
volume de calculs	8	1	2	1
nombre d'événements	5	5	3	3
temps physique	40	5	6	3

4.2.3 Solution détaillée No 2

Afin de réduire l'allocation, le système diminue le parallélisme, puisque cela permet de réduire les zones mémoires en écriture. Noter la conservation de l'alignement entre les deux traitements élémentaires. En revanche, l'ordonnancement n'est pas optiisé.

Partitionnement Cette solution diminue le parallélisme sur les deux nids de boucles intermédiaires pour économiser la mémoire.
NOMBRE DE PROCESSEURS NÉCESSAIRES : 2

	integ	diff	fil	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel La solution d'ordonnancement est inchangée.

Capacité mémoire L'économie de la mémoire est effective grâce au nombre de tampons en écriture qui est diminué d'uine part et d'autre part grâce aux duplications de données qui sont réduites sur deux processeurs en ce qui concerne le nid de boucle `diff`. Celui-ci présente en effet de nombreuses dépendances en lecture sur la dimension finie qui nécessitent des duplications si elle est parallèle.
CAPACITÉ MÉMOIRE TOTALE UTILISÉE 96 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre → fil)	1	2	2
(fil → diff)	2	2	2
(diff → integ)	1	1	1

Caractérisation des communications L'alignement entre les deux nids de boucles intermédiaire permet de diminuer les communications, néanmoins ce n'est pas le sujet de l'optimisation.

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre → fil)	2	1	1	2
(fil → diff)	0	1	2	0
(diff → integ)	2	1	1	2

Signal La diminution du parallélisme pénalise la latence mais ce n'est pas le sujet de l'optimisation.
TEMPS APÉRIODIQUE CALCULÉ = 51 (cycles)
TEMPS PÉRIODIQUE CALCULÉ = 24 (cycles)
Longueur du chemin critique = 4
EFFICACITÉ DE LA SOLUTION (SELON LA LATENCE): = 0,22
LATENCE OBTENUE = 62 (cycles)

	integ	diff	fil	carre
coût élémentaire	1	1	1	1
volume de calculs	8	2	3	1
nombre d'événements	5	5	3	3
temps physique	40	10	9	3

4.2.4 Solution détaillée No 3

Le système n'a toujours pas optimisé l'ordonancement mais choisi de réduire le parallélisme, il n'y a plus qu'un seul processeur et les communications sont toutes évitées.

Partitionnement Le système place l'ensemble des traitements sur un processeur, afin de minimiser les duplications.
NOMBRE DE PROCESSEURS NÉCESSAIRES : 1

	integ	diff	fil	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel La solution d'ordonnancement est inchangée.

Capacité mémoire Le nombre de tampon en écriture et de duplications ayant diminués, la mémoire a pu diminuer.
CAPACITÉ MÉMOIRE TOTALE UTILISÉE 72 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre → fil)	1	2	2
(fil → diff)	2	2	2
(diff → integ)	1	1	1

Caractérisation des communications Il n'y a plus de communication !

Signal La diminution du parallélisme pénalise malgrè tout la latence, bien qu'il n'y ait plus de communication.
TEMPS APÉRIODIQUE CALCULÉ = 55 (cycles)
TEMPS PÉRIODIQUE CALCULÉ = 27 (cycles)
Longueur du chemin critique = 4
EFFICACITÉ DE LA SOLUTION (SELON LA LATENCE): = 0,39
LATENCE OBTENUE = 70 (cycles)

	integ	diff	fil	carre
coût élémentaire	1	1	1	1
volume de calculs	7	4	4	1
nombre d'événements	5	5	3	3
temps physique	35	20	12	3

4.2.5 Solution détaillée No 4

Dans cette solution, le système considère que le parallélisme peut se être deux processeurs si l'on optimise l'ordonnancement.

Partitionnement Le parallélisme est de nouveau de deux processeurs. Ceci prouve que l'optimisation de l'ordonnancement est parfois préférable à la diminution du nombre de processeurs.
Nombre de processeurs nécessaires : 2

	integ	diff	fil	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel L'ordonnancement a optimisé la mémoire en diminuant l'allocation de blocs intermédiaires. L'utilisation des résultats intermédiaire se fait plus tôt avec ce type d'ordonnancement.

	integ	diff	fil	carre
$\beta^t =$	0	1	4	5
$\alpha^t =$	${}^t(2, 1)$	${}^t(2, 1)$	${}^t(4, 1)$	${}^t(4)$

Capacité mémoire Ce tableau démontre que l'utilisation des blocs de résultats intermédiaires est très importante si l'on souhaite optimiser la mémoire.
Capacité mémoire totale utilisée 60 (mots mémoire)

arêtes	distance événementielle	cardinalité des événements	nombre blocs actifs en lecture
(carre $\rightarrow$ fil)	1	1	1
(fil $\rightarrow$ diff)	3	1	1
(diff $\rightarrow$ integ)	1	1	1

Caractérisation des communications L'alignement des données entre les deux traitements intermédiaires permet d'obtenir de faibles communications.

arêtes	Communications élémentaires	Blocs récepteurs	Echanges	Durée
(carre $\rightarrow$ fil)	2	1	1	2
(fil $\rightarrow$ diff)	0	1	2	0
(diff $\rightarrow$ integ)	2	1	1	2

Signal Les faibles communications associées à l'utilisation de deux processeurs font que cette solution est bonne vis-à-vis de la latence.
Temps apériodique calculé = 33 (cycles)
Temps périodique calculé = 24 (cycles)
Longueur du chemin critique = 5
Efficacité de la solution (selon la latence): = 0, 36
Latence obtenue = 38 (cycles)

	integ	diff	fil	carre
coût élémentaire	1	1	1	1
volume de calculs	8	2	3	1
nombre d'événements	3	3	2	2
temps physique	24	6	6	2

4.2.6 Solution détaillée No 5

Cette solution montre qu'il est aussi important d'optimiser l'ordonnancement que d'optimiser le nombre de processeurs pour avoir une bonne allocation. L'ordonnancement est optimisé et le parallélisme n'est plus que d'un seul processeur.

Partitionnement La système place l'ensemble des traitements sur un seul processeur, réduisant, comme à l'habitude le nombre de tampons en écriture ainsi que les duplications de données.
Nombre de processeurs nécessaires : 1

	integ	diff	fil	carre
$P^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$L^t =$	$\begin{pmatrix} 1 & 0 \\ 0 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$
$up(\mathbf{c})$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty, 1)$	${}^t(\infty)$

Ordonnancement événementiel La solution d'ordonnancement est inchangée.

Capacité mémoire L'optimisation de la mémoire est uniquement du au changement de partitionnement.
Capacité mémoire totale utilisée 42 (mots mémoire)

Caractérisation des communications Il n'y a plus de communication !

Signal Malgrè l'absence de communication, la diminution du parallélisme restreint les performances de l'application `pres_rec`.
Temps apériodique calculé = 37 (cycles)
Temps périodique calculé = 27 (cycles)
Longueur du chemin critique = 5

Efficacité de la solution (selon la latence): $= 0,63$
Latence obtenue $= 43$ (cycles)

	integ	diff	fil	carre
coût élémentaire	1	1	1	1
volume de calculs	7	4	4	1
nombre d'événements	3	3	2	2
temps physique	21	12	8	2

4.3 Synthèse et analyse de l'optimisation de la mémoire

Tableau de synthèse							
Critère à optimiser : minimisation de la latence							
no	Nb proc.	Mémoire éléments	Chemin critique évts	Latence cycles	Temps Apériodique cycles	Temps Périodique cycles	Efficacité %
0	4	800	nc	74	nc	nc	nc
1	4	144	4	54	46	21	12
2	2	96	4	62	51	24	22
3	1	72	4	70	55	27	39
4	2	60	5	38	33	24	36
5	1	42	5	43	37	27	63

Cette application illustre le cas où le recouvrement influence le partitionnement. En effet, au cours de la recherche de la solution optimale pour minimiser la capacité mémoire, le partitionnenement et l'ordonnancement changent. En effet, les premières solutions sont mappées sur 4 processeurs, puis 2 et finalement on s'aperçoit que pour satisfaire la requête initiale, un seul processeur est utile. Le système, une fois encore, privilégie de lui-même un ordonnancement des tâches où seules les partitions connexes sont exécutées entre la partition de la tâche d'entrée et la partition de la tâche de sortie. Le système optimise l'ordonnancement jusqu'à ce que le nombre de tampons à allouer en lecture soit unique pour chaque nid de boucles.

5 Optimisations avec contraintes sur paramètres initiaux

Nous donnons ici quelques exemples caractéristiques d'optimisation avec des contraintes fortes sur les paramètres qui ne sont pas dans la fonction de coût. L'explication des solutions obtenues peut être parfois ambigue à cause de nombreuses contraintes reposant sur les paramètres applicatifs. Les cas d'écoles `pres` et `pres_rec` sont utilisés pour procéder à ces expérimentations. Nous donnons ici des résultats sur des optimisations antagonistes avec des paramètres initiaux comme par exemple : minimisation de la latence avec contrainte de mémoire. Les caractéristiques des solutions étant semblables à celles trouvées précédement nous donnons donc uniquement les récapitulatifs.

5.1 Optimisation de l'efficacité avec contrainte de latence (`pres`)

On optimise ici l'efficacité de l'application `pres`, en utilisant une contrainte très forte pour la latence.

5.1.1 Description de la requête : maximise efficacité avec 4 processeurs disponibles

Critère à optimiser	MAXIMISE EFFICACITE
Architecture type	SPMD
Nombre de processeurs disponibles	4
Capacité mémoire disponible	800 (éléments)
Latence applicative	12 cycles

5.1.2 Récapitulatif

Tableau de synthèse							
Critère à optimiser : minimisation de la latence							
no	Nb proc.	Mémoire éléments	Chemin critique évts	Latence cycles	Temps Apériodique cycles	Temps Périodique cycles	Efficacité %
0	4	800	nc	12	nc	nc	nc
1	4	64	5	10	9	10	32
2	2	40	5	10	9	10	65

5.1.3 Analyse du résultat

La dernière solution, optimale avec un processeur illustre bien le poids des communications dans le problème du placement. Le dernier traitement de `pres` s'effectuant forcément sur un processeur, le système doit nécessairement considérer une communication entre les deux derniers traitements. Ceci pénalise donc l'efficacité des solutions parallèles. Si l'on maximise l'efficacité, le système devra réduire le nombre de processeurs. Néanmoins, la contrainte de latence impose d'utiliser du parallélisme. C'est pourquoi l'optimisation s'achève sur un compromis de 2 processeurs.

5.2 Optimisation de la mémoire avec contrainte de latence (`pres_rec`)

Cette section donne l'optimisation d'un paramètre d'architecture, comme la mémoire, sous une contrainte de latence forte pour l'application `pres_rec`.

5.2.1 Description de la requête : minimisation globale de la mémoire avec 4 processeurs disponibles

Critère à optimiser	MINIMISATION GLOBALE DE LA MÉMOIRE
Architecture type	SPMD
Nombre de processeurs disponibles	4

Capacité mémoire disponible 800

Latence applicative 40 cycles

5.2.2 Récapitulatif

Tableau de synthèse							
Critère à optimiser : minimisation de la latence							
no	Nb proc.	Mémoire éléments	Chemin critique évts	Latence cycles	Temps Apériodique cycles	Temps Périodique cycles	Efficacité %
0	4	800	nc	40	nc	nc	nc
1	4	96	5	33	29	21	20
2	2	60	5	38	33	24	36

5.2.3 Analyse du résultat

L'optimisation de la mémoire a tendance à faire baisser le nombre de processeurs puisque la solution utilisera moins de tampons en écriture. Cependant, la contrainte de latence de 40 cycles exige d'utiliser du parallélisme pour gagner en performances. Le système calcule une première solution avec une mémoire de 96 éléments exploitant les 4 processeurs. Cependant, il amméliore la capacité mémoire en réduisant le nombre de processeurs à 2. Ceci ne pénalise pas beaucoup la latence car la solution expose moins de communications. En revanche, il ne peut y avoir un unique processeur compte tenu de la latence. La dernière solution est donc optimale.

5.3 Optimisation de la latence avec contrainte de mémoire (`pres`)

Cette section présente l'optimisation de la latence avec une contrainte mémoire sur le cas d'école `pres`.

5.3.1 Description de la requête : minimisation de la latence avec 4 processeurs disponibles

Critère à optimiser MINIMISATION DE LA LATENCE

Architecture type SPMD

Nombre de processeurs disponibles 4

Capacité mémoire disponible 70 (éléments, 8,16,32 ou 64 bits)

Latence applicative 60 cycles

5.3.2 Récapitulatif

Tableau de synthèse							
Critère à optimiser : minimisation de la latence							
no	Nb proc.	Mémoire éléments	Chemin critique évts	Latence cycles	Temps Apériodique cycles	Temps Périodique cycles	Efficacité %
0	4	70	nc	60	nc	nc	nc
1	2	56	3	20	13	10	32
2	4	64	4	19	10	10	17
3	2	40	4	18	11	10	36
4	4	64	5	10	9	10	32

5.3.3 Analyse du résultat

La contrainte de mémoire de 70 éléments exige un bon ordonnancement des traitements. Or, les deux premières solutions trouvées ne considèrent pas d'ordonnancement au plut tôt, c'est pourquoi ces solutions peuvent être optimisées pour la mémoire ou la latence. Changer l'ordonnancement permet en solution n^o 3, d'améliorer ces deux paramètres. Avec la solution n^o 4, le système exploite les 4 processeurs disponibles pour optimiser la latence tout en satisfaisant la contrainte de mémoire.

6 Conclusion

Ce chapitre éclaire le type de résultats obtenus au niveau de la solution élémentaire. Cependant il ne permet pas de comprendre à quel point ces cas d'école ont permis la mise au point du système de placement. Nous avons eu en fait recours à un nombre plus élevé de cas d'écoles, chacun d'entre eux disposant d'une particularité. Ceci font donc l'objet de tests de non-régression. Nous avons choisi ici de donner les plus représentatifs. Ces expérimentations, conduites très tôt lors du développement du système ont permies la mise au point de contraintes spécifiques, entraînant parfois un raffinement de la modélisation, comme pour la contrainte de cardinalité ou les contraintes de dépendances.

Outre la validation des solutions obtenues, ces expériences de petites échelles ont permies, à partir des constations, de concevoir de nouvelles heuristiques. De cette manière, le nombre d'événements apparaissant dans la distance de dépendance s'est révélé crucial pour le choix des ordonnancements ainsi que des paramètres du modèle mémoire. Dans le même sens, cela a aussi mis en évidence l'intérêt des dépendances en lecture. De ce fait, les transformations issues d'une étape d'optimisation sur ce type de paramètres nous ont paru fondamentales à décrire, notament à cause de la structure non-linéaire du problème. Il est pour cela important d'étudier ces transformations vis-à-vis d'un unique paramètre.

Enfin, et plus généralement, le développement des stratégies et heuristiques s'est beaucoup appuyé sur ces expérimentations. Notamment lorsque celles-ci ont été poussées jusqu'à l'étude d'optimisations avec des paramètres très contraints, réputés antagonistes comme la mémoire et la latence. Il est à noter que dans de telles situations, le système effectue les bons compromis et atteind rapidement l'optimalité.

Chapitre 15

Expérimentations sur applications réelles

Introduction

Nous donnons dans cette section les résultats obtenus avec le système de placement automatique. Les exemples utilisés dans cette partie sont des applications réelles utilisées dans des sytèmes de traitement du signal embarqués. Elles se composent de traitements classiques du signal (cf. chap. 2, p. 41). De ce fait, elles présentent les difficultés connues en traitement du signal : dépendances en lecture (dues aux fenêtres de calculs), transpositions, dépendances de flots ... Ces applications, de diverses natures, présentent des complexités variables cependant bien supérieures à celles des cas d'école. Chacune d'entre elles comporte une structure et des caractéristiques particulières, si bien que l'application influence beaucoup la structure du problème de placement.

Néanmoins toutes les applications ont une caractéristique globale : les traitements impliquant le plus de données et le plus de calculs sont au début du graphe de flot de données. De plus, toutes les spécifications, mis à part le "Multi-Mode-Radar" où un seul mode est considéré, vérifient les propriétés applicatives exposées en (chap. 6.1, p. 67).

Trois paramètres du placement ont été optimisés lors des expérimentations (cf. chap. 12, p. 200) :

- le coût de l'architecture,
- la latence,
- et l'efficacité.

Deux types d'optimisations permettent d'évaluer le système de placement développé. L'une est l'optimisation d'un paramètre en contraignant faiblement les autres. L'autre est l'optimisation d'un paramètre, avec un paramètre antinomique fortement contraint. Dans ce dernier cas, l'optimisation est compliquée par la prise en compte d'une contrainte qui ne priviligie pas le critère à optimiser. Le système de placement doit donc procéder à des compromis. Ceux-ci sont de plusieurs natures, comme la diminution du parallélisme, le changement de dimension à paralléliser ou bien le changement de granularité grâce au bloc-cyclique. Trois types de stratégies d'énumérations sont sollicitées, couplées avec différents heuristiques (cf. chap. 13, p. 233). Ceci conduit à quatres types de stratégies globales :

1. Stratégie d'énumération des variables selon un ordre de modèles.
2. Stratégie d'énumération selon des variables selon le Graphe de Flot de Données

3. Combinaison de la stratégie précédante avec l'heuristique de plus court chemin, dit ASAP (As Soon As Possible).
4. Combinaison de la stratégie précédente avec les heuristiques d'alignements et de recouvrement de communications. Ceci est difficile à implanter, et nécessite d'associer à chaque arrête du graphe de flot de données deux propriétés :

   Alignement satisfait

   Recouvrement satisfait

   Ce concept permet à tout moment de tester l'alignement et le recouvrement. Cela permet d'introduire une combinatoire pour envisager en priorité des solutions partiellement alignées ou pour lesquelles les communications sont partiellement recouvertes.

Dans la quasi-totalité des expérimentations, l'architecture se compose de 8 ou 16 processeurs, avec un temps d'accès au réseau nul et une bande passante de 4 cycles pour une donnée communiquée. De ce fait, nous utilisons le modèle de communications "pipeline" revenant à faire un partitionnement bloc-cyclique de manière implicite.

Nous analysons donc les résultats d'un point de vue optimisation combinatoire comme d'un point de vue placement automatique.

Cette section est organisée selon les exemples d'applications de traitement du signal. La veille bande large (§ 1, p. 277), la veille bande large modifiée (§ 2, p. 282), le "flank array" (§ 5, p. 302) sont des applications sonar. Le multi-mode-radar (§ 4, p. 295) et le radar3d (§ 3, p. 288) sont des applications de radar.

Chaque section est divisée en trois parties. La première permet de caractériser les difficultées présentées par l'application. La seconde donne les invariants du comportement du système dans les diverses situations rencontrées. Enfin, l'étude des temps de résolution et du nombre de "retours-arrières" renforce l'analyse globale.

Ces sections sont illustrées de schémas représentant les solutions (dans le même esprits que ceux utilisés par les experts en Traitement du signal numérique). Des courbes d'optimisations permettent de visualiser l'optimisation du critère ainsi que l'évolution des autres paramètres. Enfin, les tableaux résument les expérimentations : ils donnent le nombre de "retours-arrières"pour chaque stratégie d'énumération et par modèle[1].

1 Veille bande large

L'application de "veille bande large" a déjà été présentée en (chap. 4.3, p. 59).

1.1 Caractérisation de l'application

Elle comporte le traitement de *FFT* suivi de la *Formation des Voies* avec un accès modulo au tableau, selon l'axe des capteurs. Ceci engendre de nombreuses dépendances en lecture, et donc des communications et allocations potentiellement importantes entre les deux traitements. De plus, l'application enchaine sur un *Regroupement dans les Bandes de Fréquences* ce qui nécessite de transposer les données. La difficulté de cette exemple réside dans le placement de ces traitements : si on place le

[1] Ces nombres sont donnés dans l'ordre : modèle de partitionnement, modèle d'ordonnancement, heuristiques (lorsqu'il s'agit de la dernière stratégie d'énumération). La somme de ces valeurs donne le nombre total de "retours-arrières".

parallélisme sur les capteurs, on fusionne les boucles des traitements suivants facilement, mais en ayant une communication importante entre les deux premiers traitements. En revanche, si on place le parallélisme sur les fréquences, cela entraine deux communications successives moins importantes.

Cet exemple nécessite aussi la résolution des dépendances de flot sur le temps. En effet, l'application enchaîne plusieurs intégrations sur le temps en fin de traitements.

1.2 Comportement du système

Les optimisations se font avec huit processeurs au maximum. Quelque soit le type d'optimisation, avec la stratégie appropriée, le système détecte le problème de l'enchainement des traitements : *FFT, Formation des Voies, Regroupement Large Bandes.* Aussi, au début de l'optimisation, le parallélisme est placé sur les voies et sur les fréquences en fin d'optimisation comme le montre la figure 15.1, page 279). Cette figure donne le placement de l'application de la même façon que les experts en traitement du signal le représente : pour un traitement, le repère central indique les axes de calculs, le repère de gauche indique les axes du tableau de flot d'entrée tandis que le repère de droite, les axes du tableau de flot de sorties. Les dimensions de calcul en gras indiquent les axes parallélisés.

En fait, le début de l'optimisation implique des communications et allocations bien plus importantes qu'en fin d'optimisation. Au cours de l'optimisation, le système bascule le parallélisme depuis les voies vers les fréquences, en testant les solutions intermédiaires. Ce comportement est global aux trois critères d'optimisation. Il est combiné avec la réduction du parallélisme lorque l'on optimise le coût de l'architecture ou bien l'efficacité.

L'ordonnancement est le même lors de l'optimisation lorsque l'on utilise l'heurisique ASAP, et seul le partitionnement varie au cours de l'optimisation. Si on ôte cette heuristique, l'ordonnancement converge lentement vers l'ordonnancement de type ASAP.

La figure (fig. 15.2, p. 280) illustre bien cela : Les trois premières solutions sont obtenues grâce au changement progressif du parallélisme sur les axes de calculs, diminuant ainsi les communications. Puis, le système déduit qu'il peut placer l'application sur 4 processeurs compte tenu de la contrainte de latence (solution no. 4). Néanmoins, un mauvais placement sur 4 processeurs est moins efficace – malgré la baisse de parallélisme – qu'un bon placement sur 5, c'est pourquoi le système considère subitement les 5 processeurs (solution no. 5). Enfin, le système calcule un bon partitionnement sur 4 processeurs (solution no. 6).

1.3 Optimisation combinatoire

Les trois types d'optimisations – Latence, Efficacité, Architecture – ont de nombreux points communs pour cet exemple puisqu'un même type d'optimisation améliore la mémoire comme les communications.

Quelques chiffres montrant la complexité du problème posé :

- Nombre de points de choix pour l'ordre lexicographique : 4
- Nombre de variables à instancier pour résoudre le problème : 45
- Nombre de contraintes à résoudre : 374

L'optimisation de la latence (table 15.1, p. 283) traduit l'efficacité de l'heuristique d'ordonnancement ASAP et donc de la deuxième stratégie utilisant le Graphe de Flot de Données. Le fait d'imposer la mémoire serrée met en évidence la coupe de la recherche qui se traduit par un gain en nombre de

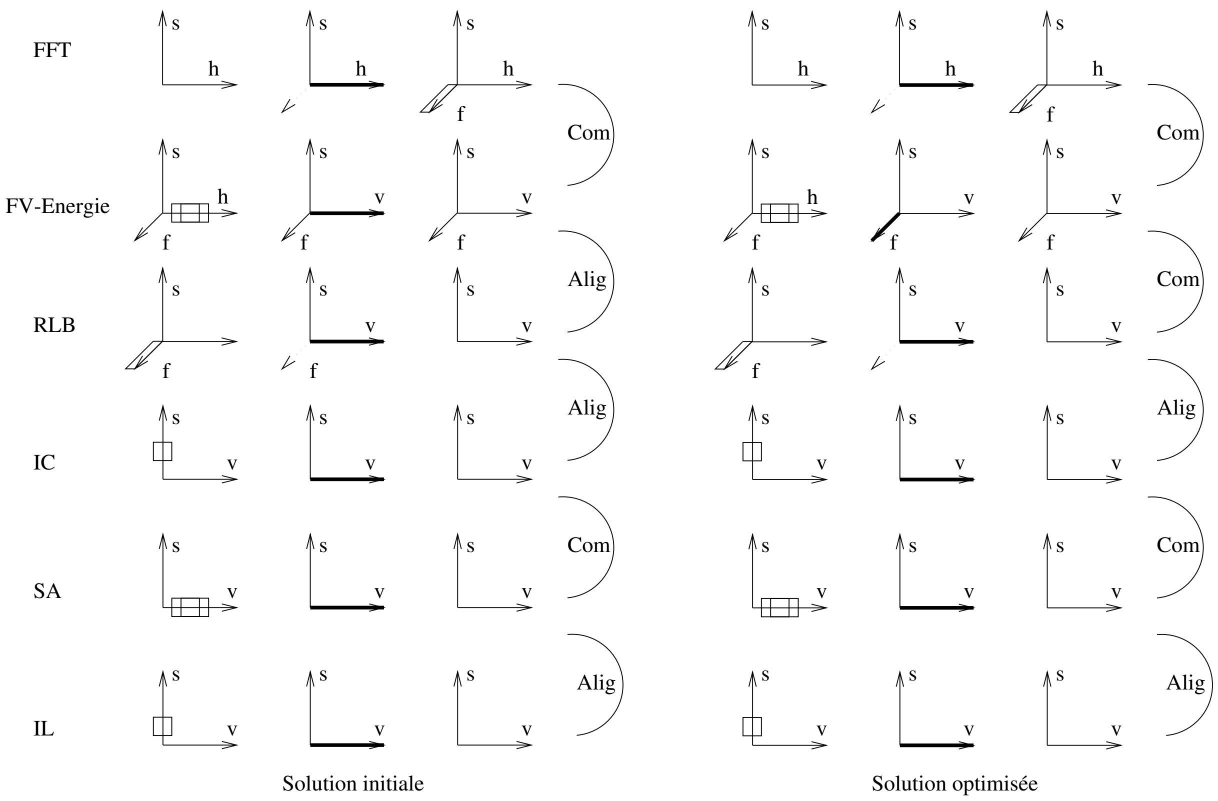


FIG. 15.1. – *Solutions globales pour la Veille Bande Large*

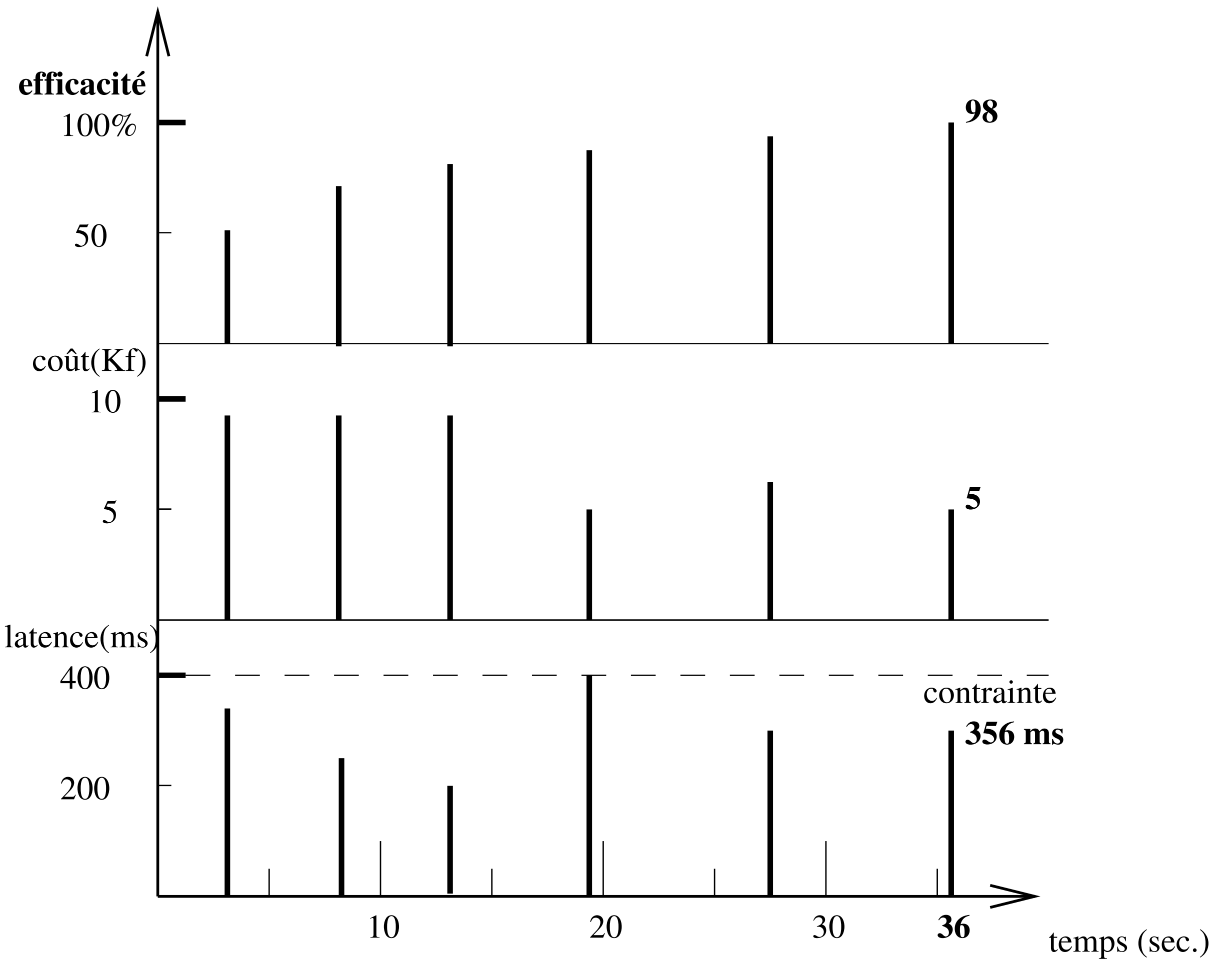


FIG. 15.2. – *Optimisation de la "Veille Bande Large" selon l'efficacité avec une contrainte de latence, une stratégie "DataFlow" et un heuristique de plus court chemin*

"retours-arrières". En revanche, la dernière stratégie nécessite une recherche bien plus conséquente pour un résultat un peu moins bon (205 ms au lieu de 204 ms). Le record de 204 ms n'a pu être battu, même en effectuant des recherches manuelles.

L'optimisation de l'architecture (table 15.2, p. 284) avec une latence importante présente peu d'intérêt puisque le système trouve la solution triviale, un processeur, rapidement. Cependant, le fait d'imposer une latence serrée engendre peu de recherche supplémentaire : 2 "retours-arrières" supplémentaires malgré des solutions très différentes. Encore une fois, la faiblesse de la dernière stratégie d'énumération est mise en évidence par le nombre important de "retours-arrière".

Sans l'heuristique de plus court chemin, le système rencontre des difficultés à optimiser l'efficacité (table 15.3, p. 285). En effet, il cherche à optimiser les partitionnements et ordonnancement plutôt que de diminuer le parallélisme. Dans le cas où l'heuristique ASAP est utilisée, le système résiste à l'introduction d'une contrainte de latence serrée (fig. 15.2, p. 280). On conserve une efficacité de 98% malgré l'utilisation de 4 processeurs.

2 Veille bande large modifiée

Cette application est une variante de la "Veille Bande Large". Elle se compose des mêmes traitements, mais les accès aux tableaux comportent plusieurs propriétés supplémentaires, détaillées ci-dessous.

2.1 Caractérisation de l'application

Le traitement de Stabilisation Azimutale a été enrichi d'une région en écriture, et le temps de calcul de ce traitement est augmenté. Cette propriété peut entrainer des communications entre les deux derniers traitements.

Le traitement de la Transformée de Fourier est exécuté avec des dépendances en lecture sur le temps. Cette modification nécessite d'allouer et de communiquer plus de données sur le chemin critique de l'application. En revanche, comme la Transformée de Fourier est le premier traitement de la chaîne, les dépendances de flots sont inchangées.

2.2 Comportement du système

Les optimisations se font avec 16 processeurs au maximum. Concernant le début de l'application, le système se comporte de la même façon que pour l'exemple de veille bande large : le parallélisme bascule progressivement des voies sur les fréquences pour obtenir deux petites communications au lieu d'une très importante. On assiste aussi à la même combinaison de réduction de parallélisme et changement de partitionnement pour la formation des voies comme l'indique la figure 15.3, page 286 : la dimension h de la FFT expose moins de parallélisme, de même que la dimension v de l'Intégration Longue IL. En ce qui concerne l'ordonnancement, on note le même comportement que pour la "Veille Bande Large".

Les traitements modifiés de fin de chaîne nécessitent l'optimisation du placement des trois derniers traitements (*Intégration courte, Stabilisation Azimut, Intégration Longue*), afin de réduire les communications. Ceci ajoute une combinatoire supplémentaire à l'ensemble de l'optimisation et augmente le nombre de "retours-arrière" par rapport à la "Veille Bande Large" pour trouver des solutions optimisées. Ceci est illustré par la figure 15.4, page 286. Les nombreuses communications impliquées par cet exemple pénalise l'efficacité malgré l'optimisation de la latence.

2.3 Optimisation combinatoire

Quelques chiffres montrant la complexité du problème :

- Nombre de points de choix pour l'ordre lexicographique : 4
- Nombre de variables à instancier pour résoudre le problème : 45
- Nombre de contraintes à résoudre : 374

L'optimisation de la latence (table 15.4, p. 289) illustre, comme pour "la Veille Bande Large", la coupe de l'espace de recherche que le système effectue lorsque l'on contraint la mémoire. On a 107 "retours-arrières" de partitionnement lorsqu'elle n'est pas contrainte contre 8 lorsque la capacité mémoire est serrée. Ceci confirme donc le résultat obtenue sur l'exemple précédent. La solution obtenue est la même dans les deux cas.

Application : VBL, Optimisation : Latence (ms, a 100Mhz)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de mémoire	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique Modèle	Maximiser (P)	importante	> 455160 ms	> 141, > 0 bk	ms	, bk		ms	, bk
	Maximiser le (L)	serrée	> 184920 ms	> 63, > 0 bk	ms	, bk		ms	, bk
Statique Flot	Maximiser (P)	importante	2919 ms	0, 0 bk	45060 ms	102, 4 bk	220	> 121420 ms	> 316, > 5 bk
	Maximiser le (L)	serrée	11520 ms	7, 1 bk	353250 ms	746, 637 bk	211	> 789690 ms	> 1932, > 1019 bk
Statique Flot	Maximiser (P)	importante	2870 ms	0, 0 bk	16600 ms	6, 0 bk	204	> 159726 ms	> 1075, > 61 bk
	Maximiser le (L)	serrée	3070 ms	0, 0 bk	7120 ms	1, 0 bk	204	> 97300 ms	> 487, > 61 bk
	Ordo. ASAP								
Statique Modèle	Maximiser (P)	importante	3780 ms	6,0,0 bk	1137830 ms	3055,0,162 bk	205	> 1137830 ms	> 3055, > 0, > 162 bk
	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.1. –

Application : VBL, Optimisation : Architecture (Kf)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique	Maximiser (P)	importante	> 182150 ms	$> 63,\ > 0$ bk	ms	, bk		ms	, bk
Modèle	Maximiser le (L)	serrée	> 155520 ms	$> 136,\ > 0$ bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	5240 ms	0, 0 bk	20400 ms	0, 0 bk	2	22440 ms	0, 0 bk
Flot	Maximiser le (L)	serrée	5200 ms	4, 0 bk	13810 ms	16, 0 bk	5	> 672660 ms	$> 2511,\ > 17$ bk
Statique	Maximiser (P)	importante	3340 ms	1, 0 bk	20400 ms	0, 0 bk	2	22230 ms	0, 0 bk
Flot	Maximiser le (L)	serrée	3340 ms	1, 0 bk	8860 ms	2, 0 bk	5	> 132200 ms	$> 55,\ > 147$ bk
	Ordo. ASAP								
Statique	Maximiser (P)	serrée	93880 ms	$250, 0, 26$ bk	1137830 ms	$3055, 0, 162$ bk	5	1137830 ms	$> 3055, > 0, > 162$ bk
Modèle	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.2. –

Application : VBL, Optimisation : Pourcentage efficacité									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique Modèle	Maximiser (P)	importante	> 455160 ms	> 141, > 0 bk	ms	, bk		ms	, bk
	Maximiser le (L)	serrée	> 149490 ms	, bk	ms	, bk		ms	, bk
Statique Flot	Maximiser (P)	importante	3000 ms	0, 0 bk	198180 ms	397, 4 bk	93	>1632980 ms	>4928, > 14 bk
	Maximiser le (L)	serrée	5150 ms	4, 0 bk	103350 ms	207, 4 bk	91	> 364900 ms	> 1028, > 7 bk
Statique Flot Modèle	Maximiser (P)	importante	2970 ms	0, 0 bk	57270 ms	34, 0 bk	100	59970 ms	34, 0 bk
	Maximiser le (L)	serrée	3400 ms	1, 0 bk	36410 ms	26, 0 bk	98	> 96330 ms	> 102, > 28 bk
	Ordo. ASAP								
Statique	Maximiser (P)	serrée	94950 ms	250, 0, 26 bk	363630 ms	999, 0, 133 bk	91	> 363630 ms	> 999, > 0, > 133 bk
	Maximiser le (L) Ordo. ASAP alig., rec. com.								

Table 15.3. –

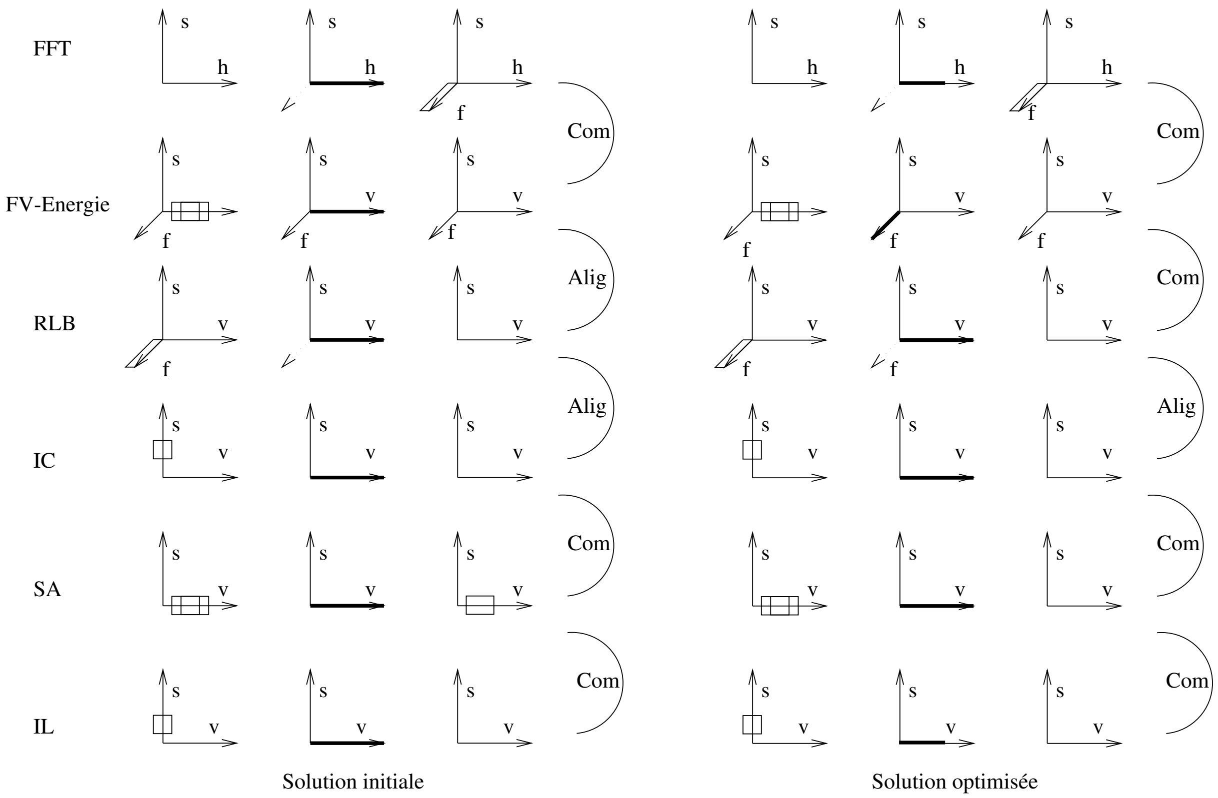


FIG. 15.3. – *Solutions globales pour la Veille Bande Large Modifiée*

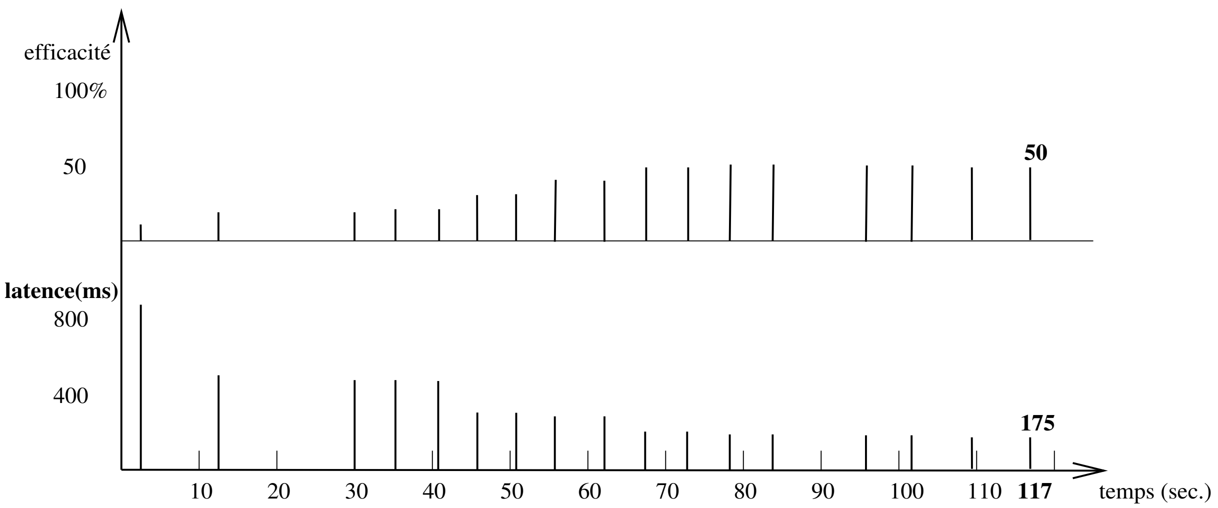


FIG. 15.4. – *Optimisation de la "Veille Bande Large Modifiée" selon la latence avec une stratégie "DataFlow" et un heuristique de plus court chemin.*

L'optimisation du coût de l'architecture (table 15.5, p. 290) avec une contrainte de latence serrée ne nécessite pas de beaucoup de "retours arrières" (seulement 9 de plus). En effet, l'optimisation de l'architecture sans contrainte de latence est triviale.

De même que pour l'optimisation de la latence, l'optimisation de l'efficacité (table 15.6, p. 291) avec une contrainte de latence supprime une combinatoire inutile : 219 "retours arrières" contre 7 lorsque ce paramètre est contraint.

La dernière stratégie expérimentée affiche encore de mauvais résultats par rapport aux autres, le nombre de "retours-arrières" est bien plus important, et les résultats obtenus sont moins bons.

3 Radar 3D

Ce traitement de radar se compose de trois axes de calculs : les capteurs, la distance et le temps (ou bien axe échantillon). L'axe distance est sollicité en premier lieu pour effectuer une compression d'impulsion[2]. L'axe des capteurs permet d'effectuer la formation de faisceaux (ce qui correspond en fait au calcul des directions), l'axe temps permet ensuite d'effectuer un filtrage Doppler pour calculer une vitesse. Enfin, l'axe échantillon est sollicité par une intégration.

3.1 Caractérisation de l'application

Cet exemple d'application ne dispose pas de dimension infinie. Le traitement est effectué sur un intervalle de temps donné à l'avance.

L'exemple est déséquilibré, les dimensions de tableau "distance" engendrent des boucles avec des bornes élevées par rapport aux dimensions "échantillons" et "capteurs" (rapport de 1 sur 100).

La notion de motif ou région est utilisée à un seul endroit, pour spécifier la compression d'impulsions. Il engendre des dépendances en lecture justement sur la dimension distance. Si le parallélisme est placé selon cette dimension de calcul, d'importants blocs de données sont à communiquer entre les *deux premiers traitements* (entre DAP et Comp. d'impulsions).

Or, si l'on considère les autres dimensions de tableaux : "capteurs" et "échantillons", elles sont toutes deux l'objets de traitements séquentiels, la première sur la formation des faisceaux et la seconde sur le filtre Doppler. Ceci signifie qu'une ou plusieurs redistributions sont indispensables.

Néanmoins la réduction liée à l'intégration (réduction selon l'axe des récurence) peut se révéler être une communication très importante entre les *deux derniers traitements* (i.e. entre Module et Intégration).

Comme nous pouvons le remarquer, la granularité de cette spécification est au niveau de la dimension de tableau puisque la plupart des "motifs" d'accès concernent toute un axe de tableau. Aussi, le placement consiste surtout à déterminer quels sont les dimensions devant être séquentielles et quels sont celles devant être parallèles, en évitant des communications trop importantes.

3.2 Comportement du système

L'optimisation s'effectue avec 16 processeurs au maximum. Le comportement du système diffère en fonction du type d'optimisation, contrairement aux exemples précédents. Néanmoins, dans tous les cas de figure, il aligne les deux premiers et deux derniers traitements afin d'éviter des communications lourdes de conséquences. L'ordonnancement est ici beaucoup moins important, car il n'y a pas de dimensions infinies et la somme des calculs spécifiés doit être exécutée.

Latence : Le système calcule d'emblée un profil de solution invariant lors de l'optimisation : le parallélisme est situé sur les mêmes dimensions tout le long de l'optimisation. Le système agence les partitionements en trouvant des divisions "entières" de dimension, évitant ainsi la prise en compte de calcul issus du reste d'une division.

Architecture : L'optimisation joue aussi bien sur la diminution du parallélisme globalement (passage de 16 à 10 processeurs) que sur une répartition du parallélisme sur plusieurs axes de calcul, lorsque la latence est contrainte (cf. fig. 15.5, p. 292). Néanmoins, le système conserve là encore l'alignement initial entre les deux premiers et deux derniers traitements (cf. fig. 15.6, p. 293).

[2]Ce traitement est un filtre comparant le signal émis avec le signal reçu

Application : VBLM, Optimisation : Latence (ms, a 100Mhz)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de mémoire	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique	Maximiser (P)	importante	> 211880 ms	> 94, > 0 bk	ms	, bk		ms	, bk
	Maximiser le (L)	serrée	> 157070 ms	> 63, > 0 bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	3290 ms	0, 0 bk	169190 ms	226, 1 bk	190	> 323150 ms	> 588, > 16 bk
	Maximiser le (L)	serrée	> 334990 ms	> 1397, > 16 bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	2730 ms	0, 0 bk	117090 ms	107, 0 bk	174	> 224950 ms	> 316, > 69 bk
	Maximiser le (L)	serrée	5530 ms	2, 0 bk	27080 ms	8, 0 bk	174	> 161300 ms	> 182, > 47 bk
	Ordo. ASAP								
Statique	Maximiser (P)	importante	4140 ms	$4, 0, 0$ bk	914080 ms	$1831, 0, 93$ bk	194	> 914080 ms	$> 1831, > 0, > 93$ bk
	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.4. –

Application : VBLM, Optimisation : Architecture (Kf)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique	Maximiser (P)	importante	> 210800 ms	$> 93, > 0$ bk	ms	, bk		ms	, bk
	Maximiser le (L)	serrée	> 158120 ms	$> 94, > 0$ bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	3150 ms	$0, 0$ bk	28000 ms	$0, 0$ bk	2	30140 ms	$0, 0$ bk
	Maximiser le (L)	serrée	> 95490 ms	$> 226, > 1$ bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	2860 ms	$0, 0$ bk	26220 ms	$0, 0$ bk	2	28420 ms	$0, 0$ bk
	Maximiser le (L)	serrée	4300 ms	$4, 0$ bk	11410 ms	$7, 0$ bk	11	112120 ms	$> 132, > 62$ bk
	Ordo. ASAP								
Statique	Maximiser (P)	serrée	2485500 ms	$4551, 0, 77$ bk	3109850 ms	$5781, 0, 130$ bk	11	> 3109850 ms	$> 5781, > 0, > 130$ bk
	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.5. –

Application : VBLM, Optimisation : Pourcentage efficacité									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique	Maximiser (P)	importante	> 210800 ms	$> 93, > 0$ bk	ms	, bk		ms	, bk
	Maximiser le (L)	serrée	> 21660 ms	$> 94, > 0$ bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	3290 ms	0, 0 bk	168720 ms	226, 1 bk	46	> 284090 ms	$> 462, > 15$ bk
	Maximiser le (L)	serrée	> 96820 ms	$> 229, > 1$ bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	2880 ms	0, 0 bk	288960 ms	219, 0 bk	100	299930 ms	219, 0 bk
	Maximiser le (L)	serrée	4300 ms	4, 0 bk	11410 ms	7, 0 bk	81	> 115420 ms	$> 134, > 30$ bk
	Ordo. ASAP								
Statique	Maximiser (P)	serrée	> 890140 ms	$> 1732, > 0, > 43$ bk	ms	, bk		ms	, bk
	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.6. –

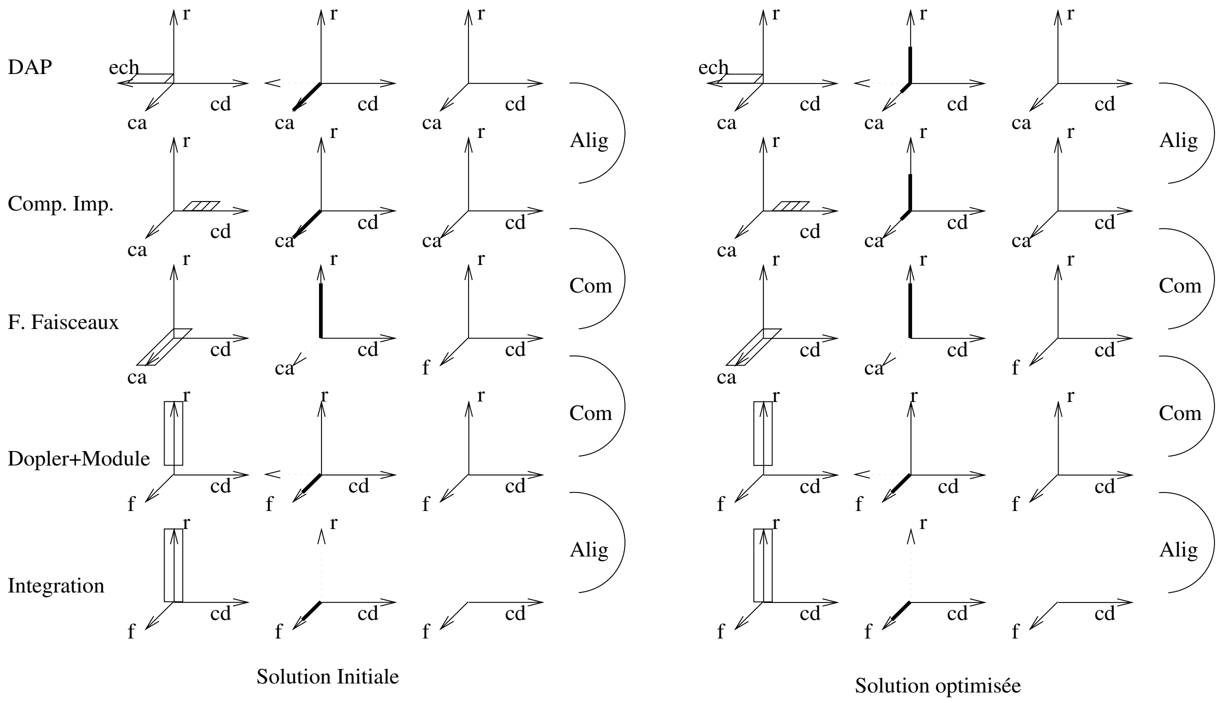


FIG. 15.5. – *Solutions architecture pour le Radar 3D*

Efficacité : Le système étant contraint par les deux premiers et derniers traitements, il n'a d'autre choix que de réduire le parallélisme.

Globalement, le système privilégie deux petites communications et des alignements qu'il calcule initialement et qu'il conserve durant l'optimisation, plutôt que de considérer une importante communication.

3.3 Optimisation combinatoire

Quelques chiffres montrant la complexité du problème :

- Nombre de points de choix pour l'ordre lexicographique :32
- Nombre de variables à instancier pour résoudre le problème : 44
- Nombre de contraintes à résoudre : 406

La seconde stratégie d'énumération est la plus pertinente pour cet exemple : elle obtient des résultats constants sur les trois critères, aussi bien du point de vue qualitatif qu'en nombre de "retour arrières". L'heuristique ASAP semble donc mal indiqué en ce qui concerne cet exemple.

Ceci est particulièrement vrai sur l'optimisation de la latence (table 15.7, p. 296), où le système calcule d'emblèe une "bonne" solution.

L'optimisation de l'architecture avec contrainte de latence (table 15.8, p. 297) n'engendre que peu de "retour arrières" supplémentaires, – comme pour les deux exemples d'application précédents –

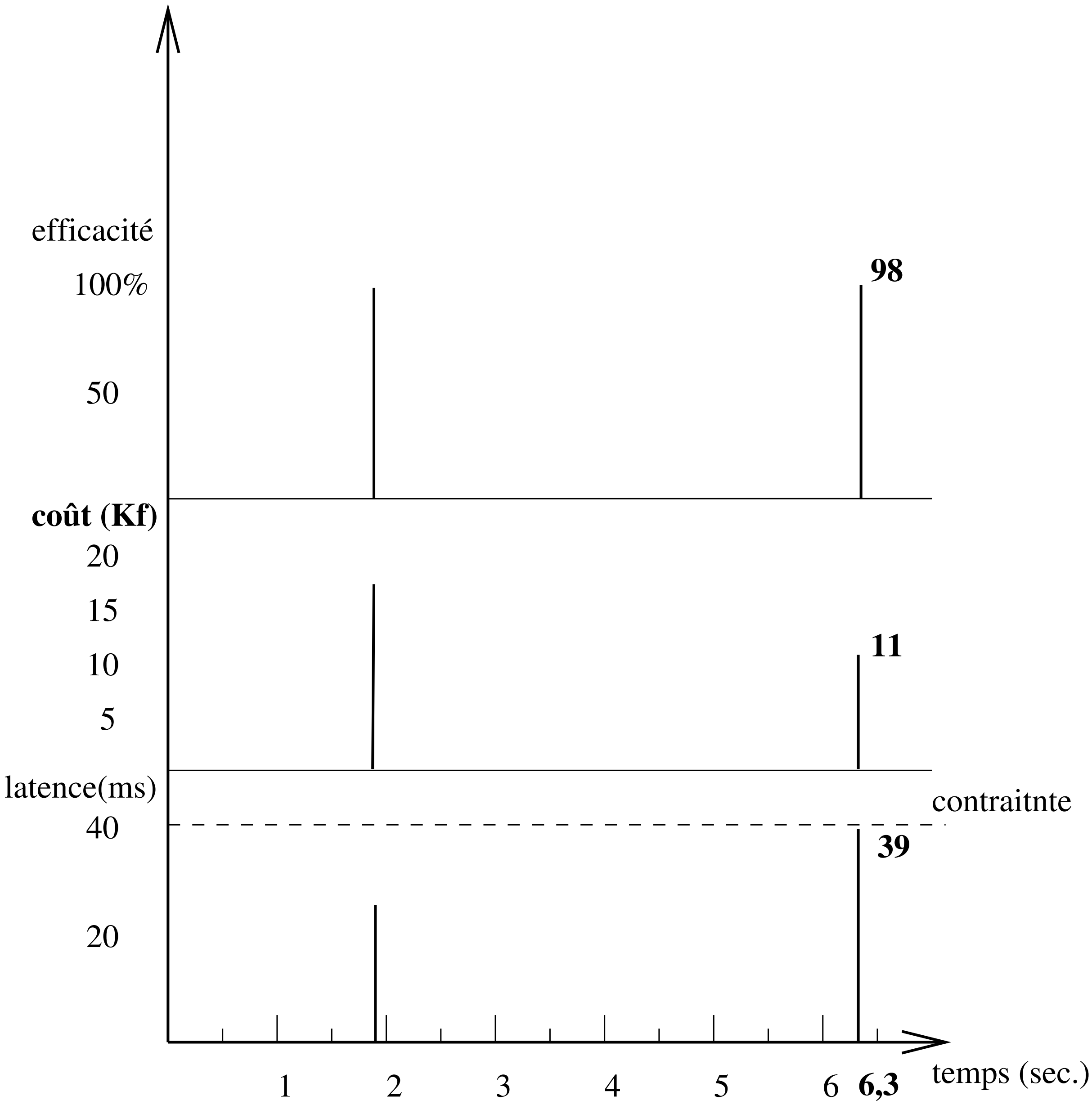


FIG. 15.6. – *Optimisation du "Radar 3D" selon le coût de l'architecture avec une stratégie "DataFlow" et une contrainte de latence*

L'optimisation de l'efficacité avec contrainte de latence (table 15.9, p. 298) pose plus de problèmes au système de placement, même avec la seconde stratégie d'optimisation. En revanche, la dernière stratégie d'énumération est plus performante de 4 "retour arrières" ($9 + 4$ pour la première solution contre 17 en deuxième solution). Le système obtient d'ailleurs les mêmes solutions dans les deux cas, qui est de 98%.

4 Multi-Mode Radar

L'application de Multi-Mode-Radar comporte beaucoup de nids de boucles, chaque nid est quadri-dimensionnel. De ce fait, l'application est bien plus complexe que celles exposées précédement. Initialiement, cet exemple d'application dispose de plusieurs mode de fonctionnement, modifiant les bornes de boucles. Dans le cadre des modèles étudiés, nous considérons dans cette exemple un seul mode radar.

4.1 Caractérisation de l'application

Comme pour l'exemple précédent, cette application ne comporte pas de dimension infinie. L'exemple enchaine des traitements complexes : filtres Doppler, “Fast Fourrier Transform” (FFT) dont l'explication dépasse le cadre de ce document. Néanmoins, cette application dispose d'une dimension de tableau “indépendante”, pour laquelle il n'y a jamais de motifs ou régions, ni de dépendance en lecture.

Les autres dimensions sont toutes sollicitées par des dimensions séquentielles portant généralement sur toute la dimension de tableau. C'est en fait ce qui caractérise les traitements comme le Doppler et les FFT. Une nouvelle fois, la granularité de la spécification est donc celle de la dimension de tableau. Le système doit alors considérer quelles sont les dimensions séquentielles pour chaque traitement afin d'éviter des transpositions trop conséquentes et maximiser le parallélisme sur la dimension indépendante. Le système de placement doit donc effectuer des compromis entre alignements et transpositions, lorsqu'il doit placer les autres dimensions.

4.2 Comportement du système

L'optimisation se fait avec 16 processeurs au plus. La durée de l'optimisation est bien plus importante que celle des exemples précédents, à cause de la complexité de l'application. L'optimisation est complexe et nous ne chercherons pas à l'expliquer en détail. Le système, comme pour l'exemple de “Radar3d”, choisit des combinaisons de petites communications et d'alignements plutôt qu'une ou deux grosses communications dans l'ensemble de l'application. Ceci confirme encore une fois le comportement du système de placement sur les exemples précédents.

Généralement, le système calcule longtemps pour trouver la dimension indépendante (1653 “retours-arrière”) qui est r, (cf. fig. 15.7,p. 299). Ceci est du à l'énumérateur des matrices de partitionnement qui considère cette possibilité très tard dans l'arbre de recherche, alors que d'autres faibles optimisations intermédiaires peuvent être conduites. Le système envisage plusieurs alignements possibles et réduit le parallélisme sur les dimensions non indépendantes quasi-systématiquement. Là encore, l'ordonnancement n'est pas prépondérant pour l'optimisation des différents paramètres.

Lorsque l'on optimise l'efficacité avec une contrainte de latence, le système conserve le parallélisme et rencontre des difficultés à résorber les communications. Pour cela, il diminue le parallélisme en privilégiant les dimensions qui engendrent peu de communications.

4.3 Optimisation combinatoire

Quelques chiffres montrant la complexité du problème :

- Nombre de points de choix pour l'ordre lexicographique : 2000
- Nombre de variables à instancier pour résoudre le problème : 91

Application : RADAR 3D, Optimisation : Latence (ms, a 100Mhz)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de mémoire	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique Modèle	Maximiser (P)	importante	2500 ms	2, 0 bk	$>$ 87000 ms	$>$ 4, $>$ 1767 bk		ms	, bk
	Maximiser le (L)	serrée	$>$ 87620 ms	$>$ 79, $>$ 0 bk	ms	, bk		ms	, bk
Statique Flot	Maximiser (P)	importante	1800 ms	0, 0 bk	$>$ 217130 ms	$>$ 340, $>$ 34 bk	25	ms	, bk
	Maximiser le (L)	serrée	3360 ms	3, 0 bk	14420 ms	19, 0 bk	39	$>$ 300080 ms	$>$ 319, $>$ 0 bk
Statique Flot	Maximiser (P)	importante	3020 ms	0, 8 bk	27000 ms	20, 32 bk	29	$>$ 108980 ms	$>$ 234, $>$ 106 bk
	Maximiser le (L)	serrée	5920 ms	3, 8 bk	64380 ms	47, 40 bk	57	$>$ 143340 ms	$>$ 143, $>$ 53 bk
	Ordo. ASAP								
Statique Modèle	Maximiser (P)	serrée	2210 ms	13, 0, 0 bk	57490 ms	258, 0, 24 bk	25	$>$ 57490 ms	$>$ 258, $>$ 0, $>$ 24 bk
	Maximiser le (L) alig., rec. com.								

Table 15.7. –

Application : RADAR 3D, Optimisation : Architecture (Kf)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique Modèle	Maximiser (P)	importante	2500 ms	$2, 0$ bk	> 85420 ms	$> 4, > 1009$ bk		ms	, bk
	Maximiser le (L)	serrée	> 55500 ms	$> 2, > 0$ bk	ms	, bk		ms	, bk
Statique Flot	Maximiser (P)	importante	1840 ms	$0, 0$ bk	23060 ms	$0, 0$ bk	2	25310 ms	$0, 0$ bk
	Maximiser le (L)	serrée	1830 ms	$0, 0$ bk	6340 ms	$2, 0$ bk	11	> 85170 ms	$> 3, > 447$ bk
Statique Flot	Maximiser (P)	importante	3050 ms	$0, 8$ bk	27620 ms	$0, 16$ bk	2	29930 ms	$0, 16$ bk
	Maximiser le (L)	serrée	3750 ms	$0, 8$ bk	> 58270 ms	$> 37, > 154$ bk	17	ms	, bk
	Ordo. ASAP								
Statique Modèle	Maximiser (P)	serrée	3970 ms	$9, 0, 4$ bk	> 1547020 ms	$> 10, > 9942, > 4$ bk	11	ms	, bk
	Maximiser le (L) Ordo. ASAP alig., rec. com.								

Table 15.8. –

Application : RADAR 3D, Optimisation : Pourcentage efficacité									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique	Maximiser (P)	importante	2490 ms	2, 0 bk	> 86670 ms	> 4, > 1767 bk		ms	, bk
Modèle	Maximiser le (L)	serrée	> 54630 ms	> 2, > 0 bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	1990 ms	0, 0 bk	56500 ms	58, 0 bk	100	59230 ms	58, 0 bk
Flot	Maximiser le (L)	serrée	1900 ms	0, 0 bk	12150 ms	17, 0 bk	98	> 155880 ms	> 202, > 14 bk
Statique	Maximiser (P)	importante	3140 ms	0, 8 bk	74380 ms	95, 56 bk	85	> 74380 ms	> 395, > 56 bk
Flot	Maximiser le (L)	serrée	3770 ms	0, 8 bk	29930 ms	20, 40 bk	83	> 43510 ms	> 46, > 49 bk
	Ordo. ASAP								
Statique	Maximiser (P)	serrée	3990 ms	9, 0, 4 bk	> 443930 ms	> 604, > 0, > 30 bk	98	ms	, bk
Modèle	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.9. –

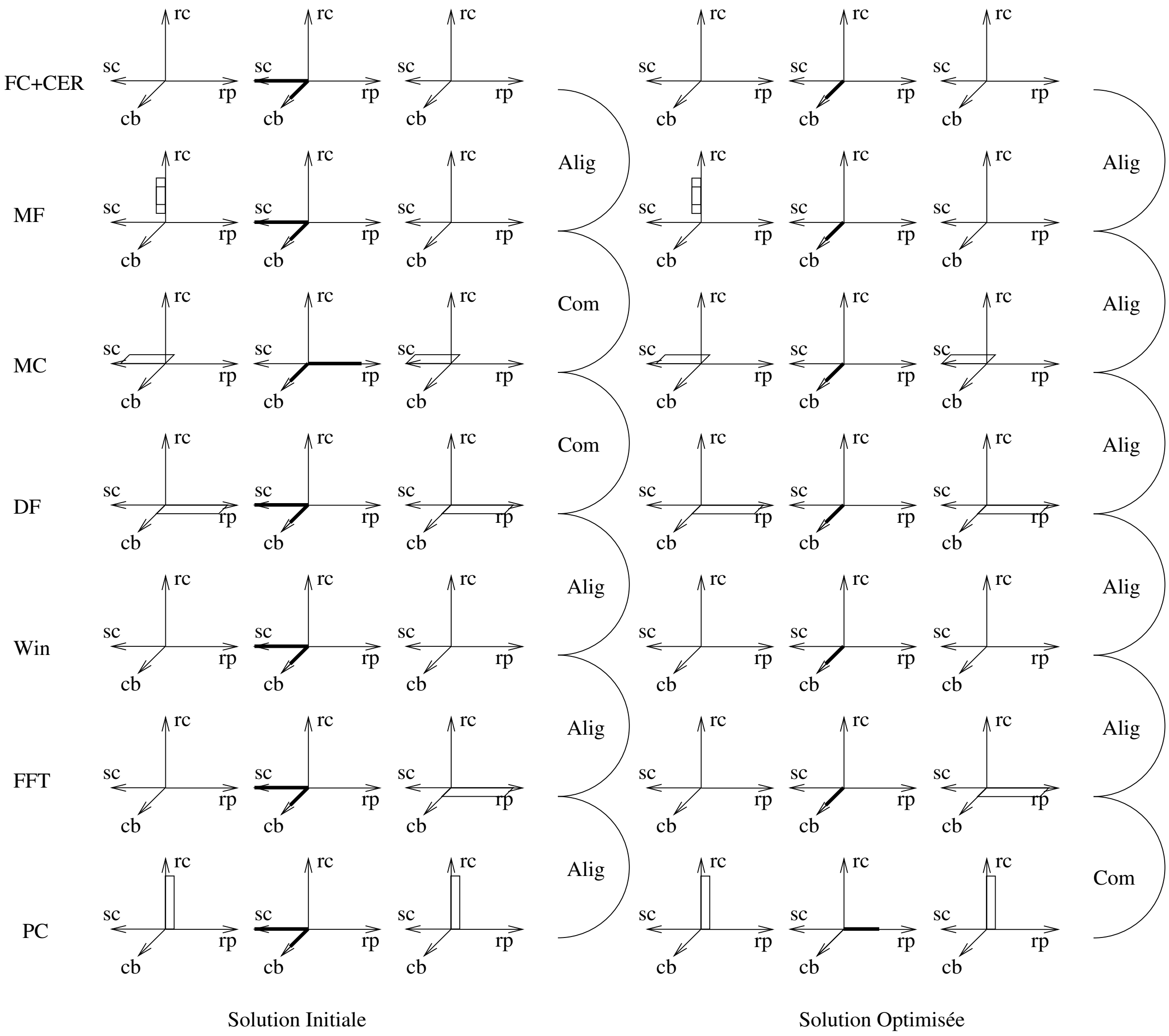


FIG. 15.7. – *Solutions efficacité pour le Multi-Mode Radar*

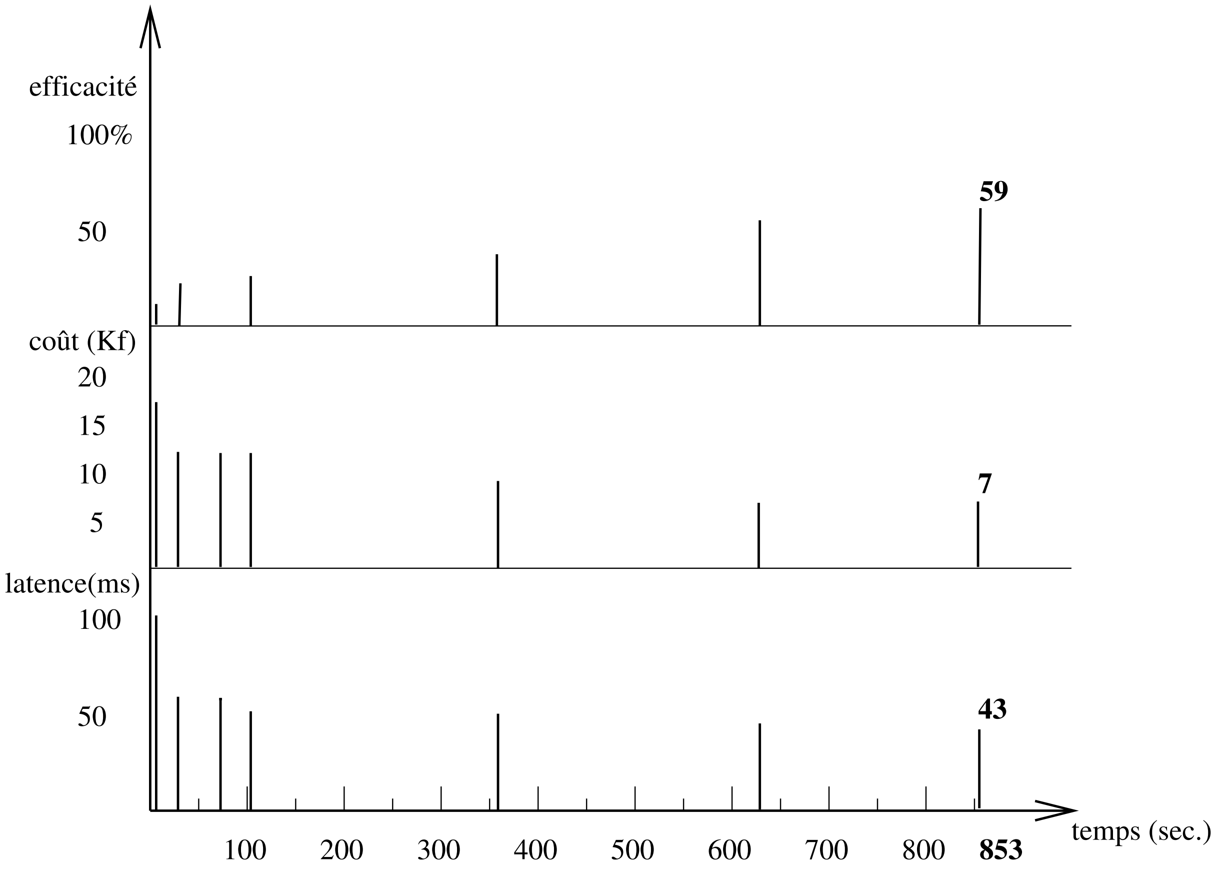


FIG. 15.8. – *Optimisation du "Multi-Mode Radar" selon la latence, avec une contrainte mémoire, selon une stratégie "DataFlow"*

- Nombre de contraintes à résoudre : 1250

On retrouve les caractéristiques cités dans les exemples précédents, mais le système rencontre plus de difficulté à optimiser les différents critères. En effet que ce soit pour l'optimisation de la latence (table 15.10, p. 303) ou de l'efficacité (table 15.12, p. 305), le système utilise beaucoup de "retours-arrière" pour optimiser les solutions. Les temps d'optimisation en jeu sont plus importants. Dans ce dernier cas d'optimisation, le système ne résiste pas à l'ajout d'une contrainte de mémoire serrée, lorsque l'heuristique ASAP est activé. Cela provient du nombre de contraintes – plus conséquent – à propager lorsque l'on considère une étape de résolution.

Coupe de la combinatoire lorsque l'on contraint la mémoire pour optimiser la latence (cf. fig. 15.8, p. 300), cela correspond dans le tableau à 2080 "retours-arrières" contre 1145 (table 15.10, p. 303).

Optimiser le coût lorsque la latence est serrée ne nécessite pas beacoup de "retours-arrières" supplémentaires : seulement 311 supplémentaires, ce qui est peu par rapport à la complexité de l'application (table 15.11, p. 304).

La dernière stratégie n'est pas très efficace sur cet exemple d'application. En effet, elle trouve un seul résultat pour la latence – moins bon – et utilise beaucoup plus de "retours-arrières" et de temps pour y parvenir. L'optimisation a donc été intérompue.

5 Flank Array

L'application de "Flank Array" est utilisée pour les sonars et constitue un exemple de taille importante. Elle comporte 8 nids de boucles pour la plupart quadri-dimensionnels. La complexité de cette application est donc importante.

5.1 Caractérisation de l'application

La dimension r, qui est en fait la dimension temps est non contrainte, il n'y a pas de dépendances lecture-lecture ni de dépendance de flot.

La granularité de la spécification est au niveau de la dimension de tableau. En effet, là encore les "motifs" d'accès concernent généaralement toute une dimension. Ceci à pour effet de supprimer le parallélisme sur les autres dimensions que r, comme pour le "Radar 3D" et le "MMR". Néanmoins, quelques traitements présentent des motifs d'accès en lecture. Il s'agit d'une part du traitement "Cluster", où le motif d'accès est multidimensionnel. D'autres parts, les traitements succédant l'acquisition ainsi que le traitement "Staz" comportent des accès avec motifs. Le système dispose donc d'une importante liberté pour partitionner ces dimensions de calculs, mais il doit aussi faire des compromis entre alignement et transpositions, comme pour les exemples précédents.

5.2 Comportement du système

La durée de l'optimisation sur cet exemple est excessivement longue, à cause de la complexité de l'application – en fait, du même ordre que le Multi-Mode Radar –. La dimension r est bien entendu fusionnée sur tout l'exemple, par le calcul d'un ordonnancement trivial.

L'optimisation de la latence peut durer plusieurs jours et même dans ce cas, n'est pas très probante sauf si l'on utilise la dernière stratégie. Comme le montre la figure 15.10 page 307, on arrive rapidement à une solution intéressante. En revanche, le temps d'optimisation, le nombre de "retours-arrière" sont tout de même très importants pour atteindre une solution quasi-optimale.

Lors de la maximisation de l'efficacité comme de la latence, le système tente différentes solutions d'alignements, jusqu'à trouver une solution quasi-optimale, caractérisée par un grand nombre d'alignements entre les traitements. Dans ces modes d'optimisation, une seule communication, inévitable si l'on veut exploiter le parallélisme ailleurs que sur r (ce qui ne présente pas beaucoup d'intérêt), est placée avant ou après le traitement de *Cluster*. La quasi-totalité des communications est recouverte, ce qui engendre une solution quasi-optimale équivalente à celle réellement implantée par les experts de Tho mson-CSF. Contrairement aux autres exemples, le système cherche à minimiser les communications entre les différents nœuds que le système trouve initialement (cf. fig. 15.9, p. 15.9).

L'optimisation de l'architecture consiste principalement à réduire le parallélisme sur les différents traitements. Pour cette optimisation, la dernière stratégie tend à réduire le parallélisme en premier lieu, puisqu'elle cherche à recouvrir les communications et/ou priviliégier les alignements. De ce fait elle est plus efficace que les autres.

5.3 Optimisation combinatoire

Quelques chiffres montrant la complexité du problème :

- Nombre de points de choix pour l'ordre lexicographique : 70

Application : MMR, Optimisation : Latence (ms, a 100Mhz)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de mémoire	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique	Maximiser (P)	importante	7820 ms	0, 0 bk	> 296190 ms	> 0, > 3963 bk		ms	, bk
Modèle	Maximiser le (L)	serrée	> 175150 ms	> 0, > 318 bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	5500 ms	0, 0 bk	880480 ms	2080, 0 bk	15	> 1244230 ms	> 3111, > 0 bk
Flot	Maximiser le (L)	serrée	5880 ms	0, 0 bk	853460 ms	1145, 0 bk	43	> 942150 ms	> 1235, > 0 bk
Statique	Maximiser (P)	importante	11550 ms	14, 0 bk	61502470 ms	151189, 882 bk	50	> 61502470 ms	> 151189, > 882 bk
Flot	Maximiser le (L)	serrée	11880 ms	14, 0 bk	> 211120 ms	> 330, > 28 bk	392	ms	, bk
	Ordo. ASAP								
Statique	Maximiser (P)	serrée	103000 ms	$11, 14, 0$ bk	1605890 ms	$165, 210, 0$ bk	151	> 1605890 ms	$> 165, > 210, > 0$ bk
Modèle	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.10. –

Application : MMR, Optimisation : Architecture (Kf)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique Modèle	Maximiser (P)	importante	7900 ms	0, 0 bk	> 195230 ms	> 0, > 1607 bk		ms	, bk
	Maximiser le (L)	serrée	> 56680 ms	> 0, > 1 bk	ms	, bk		ms	, bk
Statique Flot	Maximiser (P)	importante	5670 ms	0, 0 bk	14560 ms	0, 0 bk	12	> 5433730 ms	> 7079, > 0 bk
	Maximiser le (L)	serrée	7100 ms	9, 0 bk	183670 ms	311, 0 bk	5	> 200680 ms	> 327, > 0 bk
Statique Flot	Maximiser (P)	importante	11670 ms	0, 14 bk	> 283980 ms	> 203, > 28 bk	17	ms	, bk
	Maximiser le (L)	serrée	> 217710 ms	> 0, > 14 bk	ms	, bk		ms	, bk
	Ordo. ASAP								
Statique Modèle	Maximiser (P)	importante	> 4827840 ms	> 8681, > 14, > 18 bk	ms	, bk		ms	, bk
	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.11. –

Application : MMR, Optimisation : Pourcentage efficacité									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique Modèle	Maximiser (P)	importante	> 7900 ms	0, 0 bk	> 120820 ms	> 0,> 1074 bk		ms	, bk
	Maximiser le (L)	serrée	> 56740 ms	> 0, > 1 bk	ms	, bk		ms	, bk
Statique Flot	Maximiser (P)	importante	5590 ms	0, 0 bk	1588550 ms	2684, 0 bk	100	1588550 ms	2684, 0 bk
	Maximiser le (L)	serrée	7130 ms	9, 0 bk	75450 ms	133, 0 bk	50	> 213460 ms	> 391, > 0 bk
Statique Flot	Maximiser (P)	importante	11260 ms	0,14 bk	55006690ms	92100,154 bk	74	> 55006690ms	> 92100,> 154 bk
	Maximiser le (L)	serrée	> 498000 ms	> 1031, > 14 bk	ms	, bk		ms	, bk
	Ordo. ASAP								
Statique Modèle	Maximiser (P)	serrée	> 2086150ms	> 4623, > 14, > 17bk	ms	, bk		ms	, bk
	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.12. –

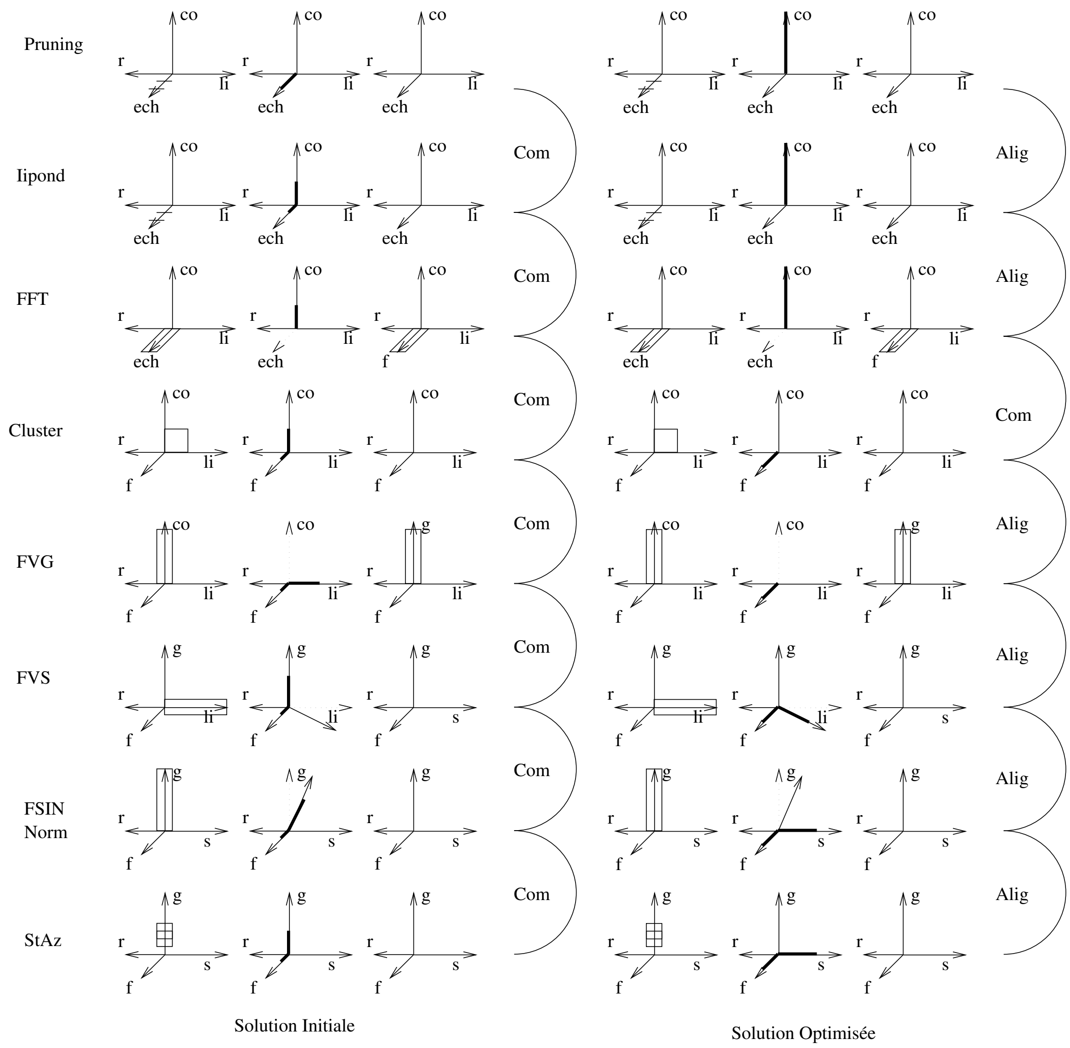


FIG. 15.9. – *Solutions quasi-optimale pour l'efficacité du Flank Array*

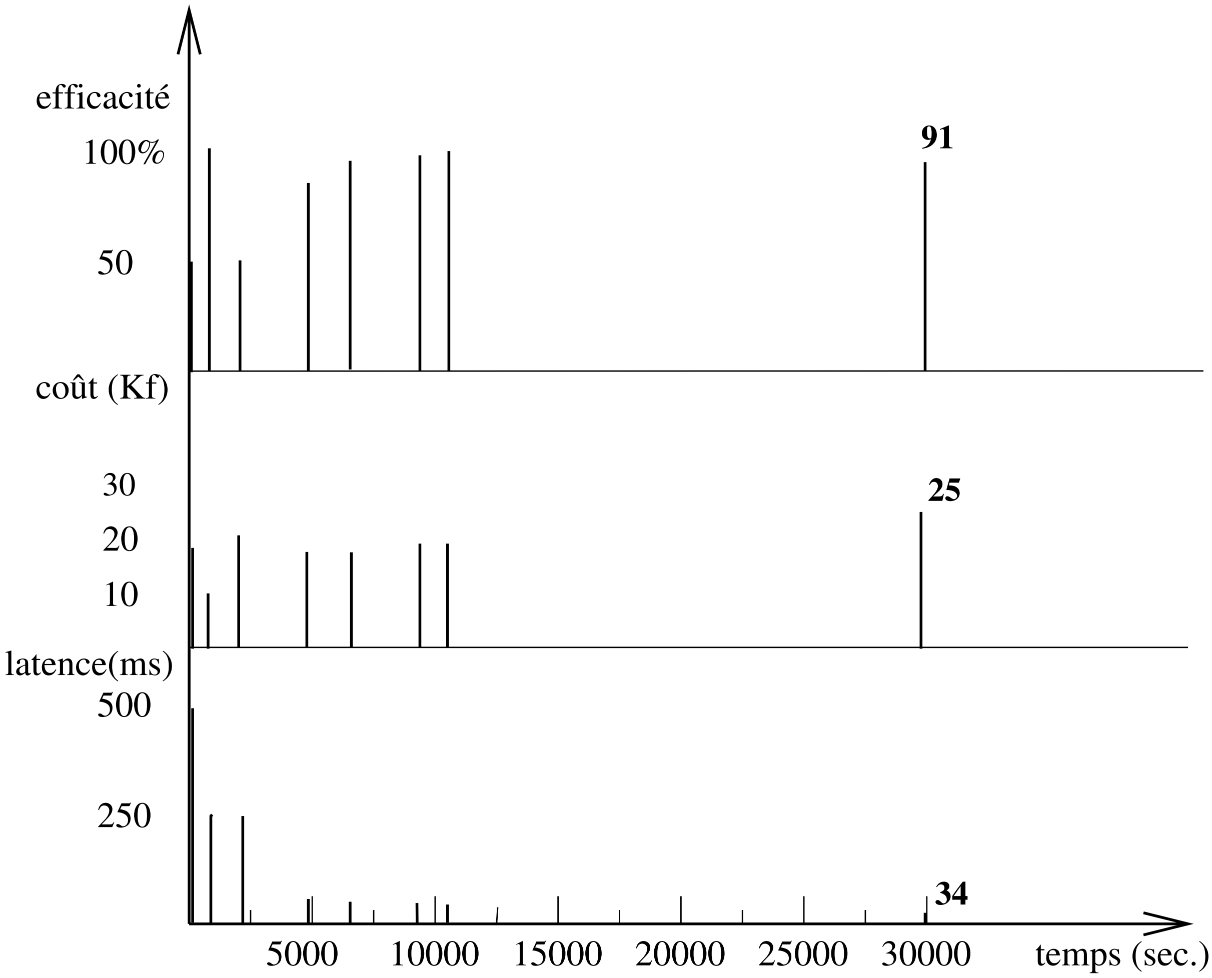


FIG. 15.10. – *Optimisation du "Flank Array" selon la latence avec une stratégie "DataFlow", les heuristiques de plus court chemin, recouvrement des communications et alignement*

- Nombre de variables à instancier pour résoudre le problème : 98
- Nombre de contraintes à résoudre : 1316

La durée de l'optimisation pénalise la réalisation des expérimentations sur cet exemple. Néanmoins, comme nous l'avons déjà expliqué, le système trouve rapidement de bonnes solutions en minimisant la latence (table 15.13, p. 310)avec la dernière stratégie d'énumération : les solutions sont à 5000 secondes. En fait, une stratégie globale d'alignements est plus indiquée sur cet exemple.

En plus de cela un nouvel effet est constaté. Lorsque l'on ajoute simplement l'heuristique au plus tôt, on remarque un impact : les partitionnements initiaux sont différents. Ceci prouve qu'il y a une propagation du modèle d'ordonnancement sur le modèle de partitionnement.

Les trois optimisations : latence (table 15.13, p. 310), efficacité (table 15.15, p. 312) et architecture(table 15.14, p. 311), laissent apparaitre des temps d'optimisation très importants. Il faut noter le très bon score de la dernière stratégie lorsque l'on optimise l'efficacité avec une contrainte de latence serrée. Toujours à l'avantage de la dernière stratégie, la latence est facilement optimisée mais la meilleure solution s'obtient en un temps très important. De même pour l'architecture, où les alignements combinés avec un faible parallélisme permettent de trouver rapidement de bons compromis (78 retours arrières).

Conclusion

Ces expériences nous ont permis de confronter cinq applications réelles au prototype de placement. Sauf en ce qui concerne le Multi-Mode-Radar, où la comparaison n'a pu avoir lieu, le système trouve des solutions semblables à celles implantées par les spécialistes en Traitement du Signal Numérique.

Selon les différents critères expérimentés, le système réalise bien l'optimisation avec des paramètres antinomiques contraints, grâce à des coupes effectuées dans la combinatoire. Le système effectue ainsi des compromis de façon aussi pertinente que l'expert. En voici les principaux :

- Basculement du parallélisme d'une dimension sur l'autre pour économiser la mémoire ou réduire les communications.
- Recouvrement des communications afin de diminuer la latence.
- Réduction du parallélisme pour économiser les communications ou l'espace mémoire (par l'intermédiaire des tampons en écriture).
- Alignement de dimensions de calculs critiques afin d'éviter les communications importantes.
- Ordonnancement "au plus tôt" afin d'éviter des calculs et communications dans le chemin critique de l'application placée.

Bien que la plupart de ces phénomènes soient mis en évidence sur des cas d'écoles, il est intéressant de les observer sur des exemples d'applications réelles.

Dans les dernières expérimentations effectuées, mais non rédigées, nous avons sélectionné le mode de communication par blocs. Le système calcule automatiquement des partitionnements blocs-cycliques avec la troisième stratégie d'énumération. De ce fait, l'ordonnancement évolue lors de l'optimisation, en concurrence du partitionnement.

D'un point de vue optimisation combinatoire, nous avons observé à plusieurs reprises des phénomènes de propagation d'un modèle sur un autre. Bien souvent, le fait d'imposer une contrainte sur la mémoire coupe la combinatoire du partitionnement, lors de l'optimisation de la latence. De même, l'heuristique ASAP influençant le choix des partitionnements a aussi été observé. De surcroît, maximiser l'efficacité avec une contrainte de latence ôte une partie de la combinatoire du partitionnement. Ceci signifie que de nombreux modèles interagissent, ces observations montrent que le système propage principalement de l'information partielle entre les couples de modèles suivants :

- Signal-Partitionnement
- Mémoire-Partitionnement
- Ordonnancement-Partitionnement

Le système doit aussi effectuer de la propagation entre le modèle de communications et le modèle du signal puisque la latence intervient fortement dans ces interactions. De même, il existe certainement une propagation entre d'autres couples de modèles, puisque la cohérence de la solution globale est constament vérifiée. Si on obtient ici la preuve que la propagation entre modèles est effective, celle-ci devrait être analysée dans le détail. Cependant, les schémas de propagation semblent être très complexes et les étudier demande une très bonne connaissance de la structure des problèmes et donc des applications de Traitement du Signal.

Du point de vue de la recherche, ces expérimentations nous ont permis d'identifier des règles de déclenchement de sous-espaces de recherches locaux :

Application : FLANK ARRAY, Optimisation : Latence (ms, a 100Mhz)								
Stratégie globale		**Expérimentations**						
Stratégie de contrôle	Heuristiques utilisées	Contrainte de mémoire	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique Modèle	Maximiser (P)	importante	> 49120 ms	$> 0, > 0$ bk	ms	, bk		ms	, bk
	Maximiser le (L)	serrée	> 49780 ms	$> 0, > 0$ bk	ms	, bk		ms	, bk
Statique Flot	Maximiser (P)	importante	374900 ms	666, 0 bk	1748730 ms	2781, 0 bk	466	> 1933020 ms	$> 3043, > 0$ bk
	Maximiser le (L)	serrée	417480 ms	564, 0 bk	> 600730 ms	$> 761, > 0$ bk	473	ms	, bk
Statique Flot	Maximiser (P)	importante	25020 ms	31, 0 bk	1743380 ms	2314, 0 bk	1400	> 1743380 ms	$> 2314, > 0$ bk
	Maximiser le (L)	serrée	34860 ms	54, 0 bk	112810 ms	171, 0 bk	2074	> 141160 ms	$> 200, > 0$ bk
	Ordo. ASAP								
Statique Modèle	Maximiser (P)	importante	20290 ms	$32, 0, 0$ bk	29959990 ms	$93939, 0, 163$ bk	34	> 29959990 ms	$> 93939, > 0, > 163$ bk
	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.13. –

Application : FLANK ARRAY, Optimisation : Architecture (Kf)									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique Modèle	Maximiser (P)	importante	> 49120 ms	$> 0, > 0$ bk	ms	, bk		ms	, bk
	Maximiser le (L)	serrée	> 49320 ms	$> 0, > 0$ bk	ms	, bk		ms	, bk
Statique Flot	Maximiser (P)	importante	376380 ms	$666, 0$ bk	3061890 ms	$3657, 0$ bk	23	> 3456640 ms	$> 3978, > 0$ bk
	Maximiser le (L)	serrée	341370 ms	$548, 0$ bk	> 744860 ms	$> 1030, > 0$ bk	33	ms	, bk
Statique Flot	Maximiser (P)	importante	25210 ms	$31, 0$ bk	1217000 ms	$1083, 0$ bk	9	> 3630660 ms	$> 2195, > 579$ bk
	Maximiser le (L)	serrée	380750 ms	$660, 0$ bk	1188620 ms	$1859, 0$ bk	24	> 585050 ms	$> 965, > 0$ bk
	Ordo. ASAP								
Statique Flot	Maximiser (P)	serrée	19381 ms	$32, 0, 0$ bk	46610 ms	$78, 0, 0$ bk	9	> 46610 ms	$> 78, > 0, > 0$ bk
	Maximiser le (L)								
	Ordo. ASAP								
	alig., rec. com.								

Table 15.14. –

Application : FLANK ARRAY, Optimisation : Pourcentage efficacité									
Stratégie globale		**Expérimentations**							
Stratégie de contrôle	Heuristiques utilisées	Contrainte de latence	Première Solution		Meilleure Solution			Preuve Optimalité	
Statique	Maximiser (P)	importante	> 50520 ms	> 0, > 0 bk	ms	, bk		ms	, bk
Modèle	Maximiser le (L)	serrée	> 50780 ms	> 0, > 0 bk	ms	, bk		ms	, bk
Statique	Maximiser (P)	importante	380750 ms	666, 0 bk	1188620 ms	1859, 0 bk	7	> 1473110 ms	> 2276, > 0 bk
Flot	Maximiser le (L)	serrée	340920 ms	548, 0 bk	2132140 ms	3311, 0 bk	7	> 2476430 ms	> 3840,> 0 bk
Statique	Maximiser (P)	importante	25700 ms	31, 0 bk	17613090 ms	27368, 0 bk	100	17613090 ms	27368, 0 bk
Flot	Maximiser le (L)	serrée	96710 ms	239, 0 bk	2857680 ms	5158, 0 bk	33	> 2857680 ms	> 5158, > 0 bk
	Ordo. ASAP								
Statique	Maximiser (P)	serrée	19880 ms	32, 0, 0 bk	72130 ms	108, 0, 0 bk	100	72130 ms	108, 0, 0 bk
	Maximiser le (L)								
Modèle	Ordo. ASAP								
	alig., rec. com.								

Table 15.15. –

1. L'heuristique ASAP doit être systématiquement appliqué aux applications qui présentent une dimension infinie, en revanche cette heuristique n'a pas d'effets sur les applications à dimensions finies.
2. Les traitements les plus couteux doivent être parallélisés en priorité.
3. En présence de l'heuristique ASAP, l'ordonnancement doit être remis en question uniquement si l'un des vecteurs $up(c_i)$ est modifié. En effet, l'heuristique couvre expérimentalement la quasi-optimalité de l'ordonnancement affine sur des chaînes linéaires.
4. Les dimensions comportant des dépendances lecture-lecture doivent être parallélisées en dernier lieu (pour la mémoire ou les communications). On pourrait imaginer un tri selon la taille du recouvrement et la somme des données en jeu.
5. Si le partitionnement bloc-cyclique s'avère nécessaire et lorsqu'un traitement expose des communications (i.e. en l'absence d'alignements), il doit être partitionné dichotomiquement. En effet, selon les dernières expériences, le système partitionne automatiquement les traitements en bloc-cyclique afin de recouvrir les communications. Or, le temps d'accès au réseau nécessite plutôt de partitionner uniquement en bloc. Seul un bon compromis de bloc-cyclique permet de résoudre le problème.
6. Si une contrainte de mémoire ne peut être satisfaite, il faut partitionner en bloc-cyclique le traitement allouant le plus de données. Cela a pour effet de diminuer la taille des blocs produits et consommés. Ces effets se combinent avec ceux de la règle précédente.

En ce qui concerne les stratégies globales de résolution, plusieurs conclusions sont à faire. La stratégie combinant l'heuristique de plus court chemin et l'énumération selon le graphe de flot de données s'est révélée être la plus pertinente sur l'ensemble des exemples d'applications. Néanmoins, dans certains cas, il faut ôter l'heuristique ASAP (ce qui constitue une simplification, comme pour le Radar3d), en d'autres il est préférable de combiner plusieurs heuristiques (recouvrement des communications, alignements, *etc...*). Ceci nous conduit en fait à une double conclusion :

D'une part, on réalise donc avec ces expérimentations que la stratégie de recherche dépend de la structure du problème posé. C'est à dire que le type de critère à optimiser et surtout la structure de l'application à placer influencent énormément la recherche de solutions. De ce fait, l'*introduction de connaissances statiques issues de l'analyse de l'application* dans la stratégie de recherche s'avère fondamentale. Ceci est techniquement aisé à mettre en œuvre et conduirait rapidement à de meilleurs résultats. De plus, ces analyses permettraient de construire automatiquement les stratégies de résolutions.

D'autre part, l'augmentation de la complexité de l'enumération est plus efficace que la combinaison de plusieurs heuristiques à considérer simultanément. En outre, l'efficacité de la stratégie selon le graphe de flot de données a conduit à *un meilleur maintien de la concurrence entre les différents modèles au niveau de la recherche de solutions.* Cependant, le système `prolog` supporte difficilement la complexification à outrance des stratégies de recherches, comme le montre la dernière stratégie implantée qui combine plusieurs heuristiques. Or, les futures stratégies seront basées sur des graphes de flots de données et des règles de déclenchement (comme ci-dessus), ce qui n'est pas simple à manipuler. Pour cela, il faut être capable de se déplacer facilement dans l'arbre de recherche global pour pouvoir activer des recherches locales.

Sixième partie

Conclusion

Conclusions

Le travail de la thèse nous a permis la réalisation d'un prototype de système de placement automatique basé sur la Programmation Logique avec Contraintes (PLC) dédié aux applications de traitement du signal (cf. chap. 2). Les résultats présentés illustrent l'intérêt de l'approche basée sur la modélisation concurrente. Le système permet de trouver des solutions satisfaisant les contraintes de placement sur des applications réelles ainsi que d'optimiser une fonction de coût choisie par l'utilisateur.

Lors de la modélisation (part. [III]) du placement, nous avons cherché à conserver la concurrence entre les différents modèles qui composent le placement globale. Ceci n'est possible qu'au prix d'approximations portant sur les modèles. L'expert procèdant de même manuellement. Ceci nous a conduit à une spécification (cf. chap. 12) qui se compose de modèles formels et de relations implantables dans un langage de Programmation Logique avec Contraintes. Les modèles utilisés sont dédiés au traitement du signal et sont obtenus à partir de techniques de parallélisation et de placement classiques comme ceux exposés en [ACIK93]. Chacun d'entre eux est exprimé à différents niveaux de granularité, dont le plus fin est celui de la macro-instruction (cf. chap. 2). Les relations, expression de la concurrence entre les modèles, sont souvent des contraintes complexes pouvant être non-linéaires.

Comme le système de PLC utilisé met en œuvre une résolution incomplète (cf. chap. 11), il favorise l'expression des contraintes complexes fondamentales à la modélisation du problème. Cela permet de **considérer de manière concurrente la complexité globale du problème** plutôt que d'étudier hors de tout contexte, la complexité de chacun des sous-problèmes. De ce fait, l'approche ne dessert en rien l'efficacité de la résolution mais au contraire l'accélére. La **modélisation par contraintes concurrente** du problème permet d'articuler la résolution du placement global en exploitant avantageusement chaque modèle. Chacun d'entre eux présente alors un ensemble de solutions locales. Celles-ci seront validées ou invalidées par les relations qui propagent l'information partielle. Cette utilisation de la PLC, au travers de la résolution concurrente et de la recherche de solution constitue une réponse à la question soulevée par Paul Feautrier en [Fea94a].

Les solutions alors obtenues sont optimales sur les cas d'écoles sans tenir compte d'heuristique (chap. 14). Dans ce cas, le système sélectionne dans les espaces de recherches locaux aux modèles les solutions locales cohérentes. Les résultats sur les cas d'écoles – décrivant une optimisation complexe où les variables des différents modèles sont indirectement corrélées – démontrent la difficulté de considérer simultanément la totalité des contraintes afin de résoudre le problème de placement global, que ce soit manuellement ou automatiquement. Lors de la résolution, le système doit donc effectuer sans cesse des compromis entre la séquentialité et le parallélisme de l'application selon le degré de liberté dont il dispose. Dans toutes les solutions de cas d'écoles, ces compromis étaient corrects.

Les résultats obtenus à petite comme à grande échelle permettent de constater les interactions entre les modèles. C'est l'observation de ces interactions qui a permis de construire itérativement les heuristiques spécifiques au domaine ainsi que les stratégies (chap. 13) de résolution. L'objet des stratégies est la gestion des compromis globaux comme la granularité, le parallélisme, et la séquentialité en ayant recours, si besoin est, aux heuristiques. Elle influence ainsi par propagation les degrés de liberté pour chacun des modèles. De ce fait, le système converge plus rapidement vers des solutions intéressantes. Finalement, sur des exemples de taille réelle comme la `Veille Bande Large` décrite en (chap. 2), les résultats donnés sont équivalents à ceux calculés manuellement. L'adjonction des heuristiques s'est révélée nécessaire pour converger rapidement vers ces solutions. En revanche, les stratégies statiques sont suffisantes pour trouver des solutions équivalentes à celles des experts en traitement du signal.

Sur le plan du **traitement du signal numérique**, nous avons mis en évidence l'interêt d'automatiser le placement des applications à un niveau fin de granularité. Mis à part le compilateur FX [OWS97], cette conclusion est en opposition avec la plupart des approches automatiques sur le placement. Ce constat conforte ainsi les conclusions établies par les experts de Thomson-CSF en la matière. Le niveau fin de granularité, associé à la modularité et à la généricité de l'approche conduit naturellement au prototypage virtuel du couple architecture-applications.

Quelque soit la taille des exemples que nous avons donnés, le trait commun au comportement du **système de placement automatique** est l'adaptation automatique du niveau de granularité aux composantes l'architecture cible. Nous avons observé ceci sur les cas d'écoles ainsi que sur des cas réels. Pour des raisons de confidentialité, ces expérimentations ne sont pas données dans la thèse. Pour bien comprendre pourquoi on arrive à ce résultat, il faut tout d'abord considérer la granularité du placement automatique. Dans la plupart des modèles traitant de ressources (partitionnement, mémoire, communications), nous avons délibérément donné une souplesse suffisante par rapport à la granularité de l'architecture comme par rapport à celle de l'application. Cette souplesse est due au fait que de nombreux modèles sont décomposés en deux niveaux.

Le niveau local au nid de boucle inclut les volumes de données accédées (modèle mémoire), le partitionnement, les volumes de calculs, les contraintes de cardinalités (modèle de temps physique) et l'ordonnancement local.

Le niveau global inclue la légalité de l'ordonnancement (respect des dépendances de flot), les communications, les contraintes capacitives globales de mémoire et de temps physique.

La limite entre ces deux niveaux est laissée au libre arbitre du système. C'est le système, par l'intermédiaire du partitionnement, qui décide à quelle granularité les blocs de calcul sont ordonnancés dans le temps et distribués sur les processeurs. Ainsi, chaque modèle est libre de choisir son niveau de granularité vis-à-vis de l'application (cela à travers le graphe de dépendances) comme de l'architecture cible tant qu'il reste cohérent avec les autres modèles. De même, le système réalise automatiquement les compromis entre séquentialité et localité.

Cela n'est possible que grâce à la stratégie de résolution qui contrôle l'échange d'informations partielles entre les modèles concurrents, les contraintes de relations décident de la validité des solutions locales à un ou plusieurs modèles. Lorsqu'une solution partielle est invalidée, la stratégie éprouve de nouveaux paramètres de placement pertinents. Les relations entre modèles assurent de cette façon la cohérence de la solution globale.

Cette adéquation entre l'architecture, l'application et la modélisation est réalisable car les propriétés du langage de Programmation Logique avec Contraintes utilisé (compositionalité, modularité et généricité) se sont répercutées sur les modèles et les relations. A cause de la non-linéarité du choix de la granularité et finalement de la complexité du problème de placement global, il n'est pas possible de raisonner au niveau le plus fin mais nécessairement au niveau d'un modèle. Cette remarque est valable aussi bien pour l'homme que pour la machine. Dans ce sens, les propriétés de l'approche de modélisation concurrente permettent non seulement la résolution du placement mais apportent aussi une grande maintenabilité où l'homme peut intervenir directement sur les modèles. Ajouter, modifier, supprimer des modèles, des relations et des fonctions de coûts aussi facilement que l'on manipule des contraintes garantit l'évolutivité et la modularité du système de placement global. Ceci permet de considérer celui-ci comme un véritable environnement de placement et d'optimisations d'applications de traitement du signal.

Les résultats ouvrent plusieurs voies dans les domaines abordés par la thèse.

En **Traitement du Signal**, les paramètres applicatifs comme la résolution ou la taille des sur--échantillonages conditionnent aussi bien la qualité de l'algorithme que la puissance de calcul requise. Pouvoir optimiser ces paramètres, à l'aide d'un modèle d'application, est désormais envisageable. Dans la même ligne de conduite, certains traitements numériques du signal sont fonctionnellement équivalents mais pas parallélisables de la même façon. Le choix de la bonne combinaison de traitements pourrait être automatisé de la même façon, selon un **modèle d'applications**. De même, les organes d'acquisition peuvent être modélisés à l'aide d'un lien avec le temps physique. Un **modèle de capteur** décrivant ces organes est envisageable et enrichirait la description des entrées/sorties. Composant coûteux, ils pourraient faire l'objet d'optimisations et de configurations automatiques (taille ou géométrie d'une antenne radar).

Au niveau du **Placement Automatique** proprement dit, la génération de code est envisageable à partir des solutions obtenues. De ce fait, on pourrait imaginer des **modèles de génération de code** appropriés aux différentes architectures disponibles. D'ailleurs, le raffinement du **modèle d'architecture** est envisageable, autorisant ainsi le design d'une ou plusieurs de ses composantes. En particulier, **le modèle de mémoire cache** reste à définir. Cependant, l'élargissement de la couverture applicative reste une préocupation importante. Cela nécessite de prendre en compte les tests, les indirections dans les tableaux, les bornes de boucles variables. Un certain nombre de solutions a déjà été envisagé. Le partitionnement cyclique, qui favorise l'équilibrage de charge, semble être déjà une solution aux problèmes de bornes de boucles. De même, il faut pouvoir construire automatiquement selon l'application de traitement du signal, les polytopes paramétrés. Tout ceci nécessite l'ajout d'informations sur les arcs et les nœuds du graphe de flot de données.

Enfin, l'approche de **Modélisation par Contraintes Concurrentes** proprement dite trouve dans le domaine du placement automatique un champs d'expérimentations sans précédent ébauché en [Jou95], [JFRD94]. L'ensemble des extensions proposées ci-dessus n'est envisageable que grâce à cette approche. Celle-ci exploite les fonctionnalités et propriétés d'un langage de contraintes non seulement adaptés à l'optimisation combinatoire sur de tels problèmes, mais aussi à la nature des formalismes de placement automatique. Néanmoins, la résolution pourrait être accéléré par l'introduction de **contraintes redondantes**. De plus, des opérateurs dédiés au placement automatique pourrait être conçus afin d'augmenter la modularité de l'approche.

Ainsi, sur ces bases, le développement d'un outil industriel dépasserait largement le cadre du placement automatique en ouvrant des portes dans le design automatique, la synthèse d'architectures, le dimensionnement et la configuration de systèmes de traitement du signal.

La robustesse de ce prototype nécessite néanmoins de nombreuses investigations au niveau des techniques de base de recherche de solutions. L'agébrisation de la recherche a été identifiée par le laboratoire ATS comme une voie de recherche qui se concrétise par la bibliothèque ECLAIR. De même, un tel outil nécessite bien entendu de pouvoir adjoindre d'autres stratégies de résolution performantes, comme par exemple les méthodes de recherche locale. Parmi celles-ci, la transformation locale de solutions partielles semble être adaptée à l'optimisation. En effet, dans plusieurs cas, lors de l'optimisation de solutions, le changement de quelques variables du problème optimise une solution. Combiner une telle approche locale avec la résolution globale existante conduirait à traiter des problèmes plus conséquent avec autant d'efficacité.

Bibliographie

[AB92] A. Aggoun and N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling and Placement Problems. In *Tregastel 92*, 1992.

[ABM+92] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagner. *Fortran 90 Handbook.* Mc Graw Hill, 1992.

[AC94] B. Apvrille-Creusillet. Calcul de régions de tableaux exactes. In *The proceedings of 6èmes Rencontres francophones du parallélisme, RenPar'6*, June 1994.

[ACIK93] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static HPF code distribution. In *The proceedings of the Fourth Workshop on Compilers for Parallel Computers, CPC'93*, November 1993.

[aG75] L. R. Rabiner andernard Gold. *Theory and Application of Digital Signal Processing.* Artech House Boston, London, 1975.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. of ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50, June 1991.

[AK87] J. R. Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM TOPLAS*, 9(4):491–542, October 1987.

[AL93a] S. P. Amarasinghe and Monica S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.

[AL93b] J.M. Anderson and M.S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *SIGPLAN Conf. on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993. ACM Press.

[AYI94] C. Ancourt, Y. Yang, and F. Irigoin. Minimal data dependence abstrations for loop transformations. In *The Seventh Annual Workshop on Languages and Compilers for Parallel Computing, LCPC'94*, August 1994.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston / Dordrecht / London, 1988.

[Ban89] U. Banerjee. A Theory of Loop Permutations. In *2nd Workshop on Languages and Compilers for parallel computing*, 1989.

[Ban90] U. Banerjee. Unimodular Transformations of DO Loops. Technical Report CSRD Rpt. No. 1036, University of Illinois, August 1990.

[BC86] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *SIGPLAN'86 Symposium on Compiler Construction*, pages 162–175, June 1986.

[Ber66] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Trans. on El. Computers*, EC-15, 1966.

[Ber90] M. Berger. *Géométrie*, volume (Tome 1). Nathan, Paris, 1990.

[BFCM94] F. Balasa, F. H. M. Franssen, F. V. M. Catthor, and H. J. De Man. Transformation of Nested Loops with Modulo indexing to affine recurrences. *Parallel Processing Letters*, 4(3):271–280, 1994.

[BGJ89] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. Technical Report 459, IRISA, Rennes, France, 1989.

[BKK94] E. Bixby, K. Kennedy, and U. Kremer. Automatic Data Layout Using 0-1 Integer Programming. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, August 1994.

[BKPS94] D. Bau, I. Kodukula, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In *Proc. of the seventh Annual Workshop on Languages and Compilers for Parallelism*, pages 4.1–4.15, August 1994.

[BML95] S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Renesting Single Appearance Schedules to Minimize Buffer Memory. Technical Report UCB/ERL M95/43, University of California, Berkeley, CA 94720, 1995.

[Bou] M. Bouvet. Traitements des Signaux Pour les Systèmes Sonars.

[BSL96] S. S. Bhattacharyya, S. Sriram, and E. A. Lee. Latency-Constrained Resynchronisation For Multiprocessor DSP Implementation. In *Proceedings of ASAP'96*, 1996.

[Cal87] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, March 1987.

[CFJ+92] P. Codognet, F. Fages, J.Jourdan, R. Lissajoux, and T. Sola. On the design of `meta(f)` and its application to air traffic control. In *Proc. ICLP'92*, Washington DC, USA, 1992.

[CGSS94] S. Chatterjee, J.R. Gilbert, R. Scheiber, and T.J. Sheffler. Array Distribution in Data-Parallel Programs. *Journal of Programming Languages*, 1994.

[Cha95] S. Chatterjee. Locality, Communication, and Code Generation for Array-Parallel Languages. In *Seventh SIAM conference on Parallel Processing for Scientific Computing*, pages 656–660, February 1995.

[Che65] N. V. Chernikova. Algorithm for finding a general formula for the non-negative of a system of inequalities. *U.S.S.R. Comput. Math. Phys.*, 5:228–233, 1965.

[Che94] M. J. Chen. Developing a multidimensional synchronous dataflow domain in ptolemy. ERL Technical Report UCB/ERL 94/16, University of California, Berkeley, Berkeley, CA 94720, May 1994.

[Chv83] V. Chvátal. *Linear Programming.* Freeman, 1983.

[CI96] B. Creusillet and F. Irigoin. Exact vs. Approximate Array Region Analyses. In *Ninth International Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[CK88] D. Callahan and K. Kennedy. Analysis of Interprocedural Side Effects in a Parallel Programming Environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[CL94] Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. In *ICLP 94*, Pise, Italie, 1994.

[Coe94] F. Coelho. Experiments with HPF compilation for a network of workstations. In *The proceedings of International Conference and Exhibition, HPCN Europe '94*, volume II, pages 423–428, April 1994.

[Col87] A. Colmerauer. Opening the Prologiii Universe. *Byte magazine*, august 1987.

[Col90] A. Colmerauer. An introduction to Prologiii. *Communications of the ACM CACM*, 33:69–90, 1990.

[Cre96] B Creusillet. *Array Region Analyses and Applications.* PhD thesis, École des Mines de Paris, December 1996.

[CSR89] S.-C. Cheng, J.A. Stankovic, and K. Ramamritham. Scheduling Algorithms for Hard Real-Time Systems. *Tutorial Hard-Real Time Systems, IEEE*, pages 150–173, 1989.

[Dar93] A. Darte. *Techniques de parallélisation automatique de nids de boucles.* PhD thesis, LIP, 1993.

[DHS+88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T.Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *International Conference on Fifth Generation Computer System*, Tokyo, Japan, December 1988.

[DHS90] M. Dincbas, P. Van Hentenryck, and H. Simonis. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8:75–93, 1990.

[Dio96] M. Dion. *Alignement et distribution en parallélisation automatique.* Thèse informatique, ENS,LYON, 1996. 136 P.

[DKR91] A. Darte, L. Khachiyan, and Y. Robert. Linear Scheduling is nearly optimal. In *Parallel Processing Letters*, pages 73–81, 1991.

[DLB+95] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J-C. Dufourd, and J-L. Marro. Array-OL : proposition d'un formalisme tableau pour le Traitement du Signal Multi-Dimensionnel. In *CRETSI*, Juan les Pins France, Jun 1995.

[DR92] A. Darte and Y. Robert. Affine-by-statement Scheduling of Uniform Loop Nests over parametric domains. Technical Report 92-16, LIP-IMAG, April 1992.

[DR94a] A. Darte and Y. Robert. Constructive methods for scheduling uniform loop nests. *IEEE Transactions on parallel and Distributed Systems*, 5(8):814, August 1994.

[DR94b] A. Darte and Y. Robert. Mapping uniform loop nests onto distributed memory architectures. *Parallel Processing Letters*, 20:679–710, 1994.

[DR94c] A. Darte and Y. Robert. Mapping Uniform Loop Nests onto Distributed Memory Architectures. *Parallel Computing*, 20:679–710, 1994.

[Fag94] F. Fages. Programmation logique. Cours de majeure d'informatique fondamentale et applications, Ecole Polytechnique, France, 1994.

[Far92] A. Farina. *Antenna-Based Signal Processing Technique For Radar Systems.* Artech House Boston, London, 1992.

[Fea88] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, Sep 1988.

[Fea91] P. Feautrier. Dataflow Analysis of Scalar and Array References. *IJPP*, 20(1):23–53, February 1991.

[Fea92a] P. Feautrier. Some Efficient Solution to the Affine Scheduling Problem, Part II, Multidimensional Time. *Int. J. of Parallel Programming*, 21(6), December 1992.

[Fea92b] P. Feautrier. Some Efficient Solutions to the Affine Scheduling Problem, Part I, One Dimensional Time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea94a] P. Feautrier. Fine-Grain Scheduling under Resource Constraints. In *7th Workshop on Language and Compiler for Parallel Computer"*, 1994.

[Fea94b] P. Feautrier. Toward Automatic Distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[FF91] B. Falkenhainer and K. D. Forbus. Compositional modeling: Finding the right model for the job. *Artificial Intelligence Journal*, 51(1-3):95–143, 1991.

[FGMS92] J. Ferrante, G.R. Gao, S. Midkiff, and E. Schonberg. A critical Survey of Automatic Data Partitionning. Technical Report IBM RC 18443, IBM Watson Research Center, Octobre 1992.

[FLMP89] M. Falashi, G. Levi, M. Martelli, and C. Plamidessi. Declarative modeling of the operational behavior of logic programming. *Theorical Computer Scinece*, 3(69):289–318, 1989.

[Fou27] J.B.J. Fourier. Histoire de l'Académie Royale des Sciences de l'Institut de France pendant l'année 1823, partie mathématiques, 1827.

[Gaz89a] C. Gazanhes. Cours d'accoustique Sous-Marine. Glm, CNRS, 1989.

[Gaz89b] C. Gazanhes. Instrumentation en Acoustique Sous-Marine. Issn 0750-7356, CNRS, 1989.

[GB92] Manish Gupta and Prith Banerjee. Demonstration of Automatic Data Partitionning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel And Distributed Systems*, 3(2):179–193, March 1992.

[GB94] Manish Gupta and Prith Banerjee. Compile-Time Estimation of Communication Costs of Programs. *Journal of Programming Languages*, 1994.

[GL95] Junjie Gu and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing*, 1995.

[Gom63] R. E. Gomory. *Recent Advances in Math. Programming*, chapter An algorithm for integer solutions to linear programs, pages 269–302. ed. R. L. Graves and P. Wolfe, Mac-Graw Hill, New-York, 1963. chap. 34.

[HC90] H.Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, ACM, 1990.

[HCF97] K. Högstedt, L. Carter, and J. Ferrante. Determining the Idle Time of a Tiling. In *POPL 97*, Paris, France, 1997.

[HK90] P. Havlak and K. Kennedy. Experience with Interprocedural Analysis of Array Side Effects. Technical Report CRPC-TR90-124, Center for Research on Parallel Computation, Rice University, Houston, TX, USA, 1990.

[HL89] C.-C. Han and K.-J. Lin. Scheduling Jobs with Temporal Distance Constraints. In *Sixth IEEE Workshop Real-Time Operating Systems and Software*, pages 18–23, Pittsburgh, May 1989.

[HLH96] C. Han, K.-J. Lin, and C.-J. Hou. Distance Constrained Scheduling and its Applications to Real-Time Systems. *IEEE Transactions On Computers*, 45(7):814–825, Jul 1996.

[HLL95] C.-C Han, K.-J. Lin, and J.W.-S. Liu. Scheduling Jobs with Temporal Distance Constraints. *SIAM J. Computing*, 24:1104–1121, Oct 1995.

[HMR+89] R. Holte, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. The Pinwheel: A Real-Time Scheduling Problem. In *22nd Hawaii Int'l Conf. System Science*, pages 693–702, Jan 1989.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical Interprocedural Parallelization: an overview of the PIPS project. In *ACM International Conference on Supercomputing, ICS'91*, Cologne, Allemagne, June 1991.

[ILO92] ILOG SA, 12, avenue Raspail, Gentilly. *ILOG-RULES : Manuel de références et d'utilisation*, 1992.

[Iri87] F. Irigoin. *Partitionnement de boucles imbriquées, une technique d'optimisation pour les programmes scientifiques.* PhD thesis, Université Pierre et Marie Curie, juin 1987.

[IT87] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[IT88a] F. Irigoin and R. Triolet. Dependence Approximation and Global Parallel Code Generation for Nested Loops. In North-Holland, editor, *International Workshop Parallel and Distributed Algorithms*, pages 297–308, Bonas, France, Oct 1988.

[IT88b] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proc. 15th POPL*, pages 319–328, San Diego, Cal., January 1988.

[JFRD94] J. Jourdan, F. Fages, D. Rozzonelli, and A. Demeure. Data alignment and task scheduling on parallel machines using concurrent model-based programming. In *Proc. ILPS 94, Poster paper*, Ithaca, November 1994.

[JL87] J. Jaffar and J-L Lassez. Constraint Logic Programming. In *POPL*, Munich, Jan 1987.

[JL95] J. Jourdan and R. Lissajoux. Plc et séquencement des vols à l'arrivée. In *Proc. Transportation and Constraint Programming*, Montpellier, France, 1995.

[JLG+95] J. Jourdan, R. Lissajoux, C. Guettier, D. Rozzonelli, and A. Demeure. Global optimisation of task scheduling and data mapping on parallel machine. In *Proc. HPCS 95*, Montréal, November 1995.

[JM87] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *ICLP*, May 1987.

[Jou95] J. Jourdan. *Concurrence et coopération de modèles multiples dans les langages de contraintes CLP et CC : Vers une méthodologie de programmation par modélisation.* PhD thesis, Université Denis Diderot, Paris VII, février 1995.

[JS93] J. Jourdan and T. Sola. The versatility of handling disjunctions as constraints. In *5th International Symposium on Programming Language Implementation and Logic Programming, PLILP'93*, Tallinn, Estonia, Aug 1993.

[JVP90] J. Jourdan and R.E. Valdés-Pérez. Constraint logic programming applied to hypothetical reasoning in chemistry. In Saumya Debray and Manuel Hermenegildo, editors, *NACLP'90: Proceedings North American Conference on Logic Programming*, pages 154–172, Austin, November 1990. ALP, MIT Press.

[KK93] K. Kennedy and U. Kremer. Initial framework for automatic data layout in fortran-D: A short update on a case study. Technical Report CRPC-TR93-324, Center for Research on Parallel Computation, Rice University, Houston, TX, USA, July 1993.

[KKB92] K.G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines. In *International Conference on Supercomputing*, pages 82–92, July 1992.

[KLS90] K. Knobe, J. D. Lukas, and G. L. Steele. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *J. of Parallel and Distributed Computing*, 8, 1990.

[KLS+94] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr. Zosel, and M. Zosel. *The High Performance Fortran Handbook.* The MIT Press, Cambridge, MA, 1994.

[KM93] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, Portland, Or., August 1993.

[Kre93a] U. Kremer. NP–completeness of Dynamic Remapping. In *Workshop on Compilers for Parallel Computers, Delft*, pages 135–141, December 1993.

[Kre93b] U. Kremer. NP-Completeness of dynamic remapping. In *Proceedings of the fourth Workshop on Compilers for Parallel Computers, Delft, Netherlands*, December 1993.

[Kre95] Ulrich Kremer. *Automatic Data Layout for Distributed Memory Machines.* PhD thesis, Rice University, Houston, Texas, October 1995. Available as CRPC-TR95-559-S.

[Lam74] L. Lamport. The Parallel Execution of DO Loops. *CACM*, 17:83–93, February 1974.

[LC91] J. Li and M. Chen. The data alignment phase in compiling programs for distributed memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

[LC96] F. Laburthe and Y. Caseau. Introduction to the CLAIRE programming language. Technical Report LIENS-96-15, Ecole Normale Supérieure, Paris, October 1996.

[LC90] J. Li and M. Chen. Index Domain Alignment: Minimizing Cost of Cross-Referencing Between Distributed Arrays. In *Proc. Third Symp. on the Frontiers of Massively Parallel Computation*, pages 424–433. IEEE, October 90.

[LD56] Ford L.R. and Fulkerson D.R. Maximal Flow through a network, 1956.

[LL73] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *ACM*, 20:44–61, jan 1973.

[LL94] A. W. Lim and M. S. Lam. Communication-free parallelization via affine transformations. In *Procs of the 7^{th} Languages and Compilers for Parallel Computing, LNCS (to appear)*, August 1994.

[LL97] A.-W. Lim and M. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Synmposium on Principles of Programming Languages*, Paris, France, Jan 1997.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous Dataflow. In *Proceedings of the IEEE*, September 1987.

[LS92] C. Lavarenne and Y. Sorel. Spécification, optimisation de performance et génération d'exécutif pour application temps-réel embarquée multi-processeur avec SynDEx. In *Real-Time Embedded Processing for Space Applications CNES International Symposium*, Les Saintes-Maries-de-la-Mer, Nov 1992.

[LS93] C. Lavarenne and Y. Sorel. Performance Optimization of Multiprocessor Real-Time Applications by Graphs Transformations. In *ParCo'93*, Grenoble, Sep 1993.

[LYZ89] Z. Li, P-C. Yew, and C-Q Zhu. Data Dependence Analysis on Multidimensional Array References. In *Proceedings of the 3rd ACM Int. Conf. on Supercomputing*, June 1989.

[Mah87] M.J. Maher. Logic Semantics for a class of Committed-Choice Programs. In Jean-Louis Lassez, editor, *ICLP'87: Proceedings 4th International Conference on Logic Programming*, pages 858–876, Melbourne, 1987. MIT.

[Mai94] R. J. Mailloux. *Phased Array Antenna Handbook.* Artech House Boston, London, 1994.

[MAL93] D. E. Maydan, Sama P. Amarasinghe, and Monica Lam. Array data-flow analysis and its use in array privatization. In *Principles of Programming Languages*, 1993.

[MBL96] P. Murthy, S. S. Bhattacharyya, and E. A. Lee. Minimising Memory Requirements for Chain-Structured Synchronous Dataflow Programs. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing, April 1994.*, 1996.

[NS96] C. J. Newburn and J. P. Shen. Automatic Partitionning of Signal Processing Programs for Symmetric Multiprocessors. In *PACT'96*, 1996.

[OWS97] D.R. O'Hallaron, J. Webb, and J. Subhlok. Performance Issues in High Performance Fortran Implementation of sensor-base applications. *Scientific Programming*, 6:59–72, 1997.

[Pla95] A. Platonoff. *Contribution à la Distribution Automatique des Données pour Machines Massivement Parallèles.* PhD thesis, Ecole Nationale Supérieure des Mines de Paris, 1995.

[Pug92] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8, 1992.

[PW86] D. J. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputer. *Communications of the ACM*, 12(29):1184–1201, December 1986.

[PW96] T. A. Proebsting and S. A. Watterson. Filter Fusion. In *Symposium on Principles of Programming Language*, 1996.

[RS90] J. Ramanujam and P. Sadayappan. Tiling of iteration spaces for multiconputers. In *Proc. International Conference on Parallel Processing*, volume 2, pages 179–186, aug 1990.

[Sar87a] V. Saraswat. CP as a general-purpose constraint language. In *The national conference AAAI87*, Seattle, July 1987. American Association for Artificial Intelligence.

[Sar87b] V. Saraswat. The Concurrent Logic Programming Language CP: Denotational and Operational Semantics. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 49–62, January 1987.

[Sch86] A. Schriver. *Theory of Linear and Integer Programming*. Jonh Wiley & Sons, 1986.

[SD90] R. Shreiber and J.J. Dongarra. Automatic Blocking of Nested Loops. Technical Report TN 37996-13014, University of Tenessee, 1990.

[SL93] Gilbert C. Sih and Edward A. Lee. Declustering: A New Multiprocessor Scheduling Technique. *IEEE Transaction on Parallel and Distributed Systems*, 4(6):625–637, June 1993.

[SM96] S. Singhai and K. McKinley. Loop Fusion for Data Locality and Parallelism. In *Proceedings of the Mid-Atlantic Student Workshop on Programming Languages and Systems*, New Paltz, April 1996.

[Smo95] Gert Smolka. The oz programming model. *Computer Science Today*, 1000:324–344, 1995.

[Sor94] Y. Sorel. Massively Parallel Computing Systems with Real Time Constraints The "Algorithm Architecture Adequation" Methodology. In *Massively Parallel Computing Systems Conference*, Italy, 1994.

[Str86] G. Strang. *Linear Algebra and its applications*. Harcourt Brac Jovanish, 1986.

[SV96] J. Subhlok and G. Vondran. Optimal latency-troughput tradeoffs for data parallel pipelines. In *Proc. SPAA'96*, Padua, Italy, 1996.

[Tol96] S. Toledo. On the Communication Complexity of the Discrete Fourier Transform. *IEEE Signal Processing Letters*, 3(6):271–272, june 1996.

[Tri84] R. Triolet. *Contribution à la Parallélisation Automatique de Programme Fortran Comportant des appels de Procédures*. PhD thesis, Université Paris VI, 1984.

[Van93] P. Van Hentenryck. Scheduling and packing in the constraint language cc(FD). Technical Report CS-93-02, Department of Computer Science, Brown University, January 1993.

[Van95] P. Van Hentenryck. Constraint Solving for Combinatorial Search Problems: A Tutorial. In *Proceedings of the First International Confernce, CP'95*, pages 564–587, September 1995.

[VSD92a] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint logic programming over finite domain: The design, implementation and application of cc(fd). Technical report, Brown university, Providence, 1992.

[VSD92b] P. Van Hentenryck, V. Saraswat, and Y. Deville. Constraint Processing in CC(FD). Technical report, Brown University, 1992.

[VSD92c] P. Van Hentenryck, H. Simonis, and M. Dincbas. Constraint Satisfaction Using Constraint Logic Programming. *Artificial Intelligence Journal*, 58:113–159, 1992.

[Wil94] D. Wilde. The Alpha Language. Technical Report 827, Irisa, may 1994.

[Wol82] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[Wol89] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman and The MIT Press, 1989.

[Wol96] M. Wolfe. *High Performance Compilers For Parallel.* Addison-Wesley Publishing Company, 1996.

[XSR$^+$96] M. Xiong, J. A. Stankovic, K. Ramamritham, D. Towsley, and R. Sivasankaran. Maintaining Temporal Consistency: Issues and Algorithms. In *1st Int. workshop on Real-Time Databases*, California, 1996.

[YAI94] T-Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations. In K. Pingali et al., editor, *Proc. of* 7^{th} *Int. Workshop on Langauges and Compilers for Parallel Computing*, volume 892 of *LNCS*, pages 201–216, Ithaca, NY, August 1994.

Table des Figures