| | |
|---|---|
| **Titre:** Title: | Évaluation des performances de l'imagerie thermique par fluorescence pour l'analyse de défaillance des flip chips |
| **Auteur:** Author: | Nicolas Boyer |
| **Date:** | 1999 |
| **Type:** | Mémoire ou thèse / Dissertation or Thesis |
| **Référence:** Citation: | Boyer, N. (1999). Évaluation des performances de l'imagerie thermique par fluorescence pour l'analyse de défaillance des flip chips [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8577/ |



| | |
|---|---|
| **URL de PolyPublie:** PolyPublie URL: | https://publications.polymtl.ca/8577/ |
| **Directeurs de recherche:** Advisors: | Michel Meunier |
| **Programme:** Program: | Non spécifié |

# INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

**The quality of this reproduction is dependent upon the quality of the copy submitted.** Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI®
800-521-0600

UNIVERSITÉ DE MONTRÉAL

# MÉTHODES DE CONCEPTION ET DE PARTITIONNEMENT LOGICIEL POUR DES ARCHITECTURES PARALLÈLES SIMD LINÉAIRES

NICOLAS CONTANDRIOPOULOS
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ET DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
AUTOMNE 1998

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*

*Our file Notre référence*



0-612-42901-6

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

# MÉTHODES DE CONCEPTION ET DE PARTITIONNEMENT LOGICIEL POUR DES ARCHITECTURES PARALLÈLES SIMD LINÉAIRES

présenté par: Nicolas Contandriopoulos

en vue de l'obtention du diplôme de: Maîtrise ès Sciences Appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BOIS Guy, Ph.D., président

M. SAVARIA Yvon, Ph.D., membre et directeur de recherche

M. BLAQUIÈRE Yves, Ph.D., membre et codirecteur de recherche

M. SAWAN Mohamad, Ph.D., membre

# REMERCIEMENTS

Même si le titre de maître n'est décerné qu'à un seul, même si cette gratification ne semble récompenser qu'un travail individuel, il est évident que jamais celui-ci n'aurait pu aboutir sans l'aide d'un grand nombre de personnes qui m'ont soutenu scientifiquement, administrativement et affectivement. À moins d'avoir confiance absolue en soi, le travail de chercheur est une perpétuelle remise en question, une succession de périodes d'enthousiasme et de périodes de doutes, de succès et de découragements aggravés par les tracas administratifs quotidiens. Ainsi ce n'est qu'avec le soutien de chercheurs aguerris, dans le cadre de travail purgé de contraintes non scientifiques et avec le soutien moral de mes proches que j'ai pu conclure mes études.

Mes premiers remerciements vont tout naturellement à M. Yvon Savaria qui m'a fait partager sa vision scientifique pour aborder les embûches rencontrées lors de mes recherches. Son appui et sa compréhension tout au long de mon programme m'ont permis de me sentir à l'aise pour discuter ouvertement de tout problème. Je tiens aussi à exprimer ma gratitude envers M. Yves Blaquière pour son appui et sa disponibilité malgré l'éloignement géographique. Je le remercie tout particulièrement pour sa grande contribution dans la rédaction et la mise au point de l'article réalisé durant mes études.

Je dois beaucoup également aux chercheurs de l'équipe PULSE avec qui j'ai travaillé. Ce sont entre autre et par ordre d'apparition (si mes souvenirs sont exacts) Qunshan Gu, Paul Marriott, Zahir Boukari et Ivan C. Kraljic, pour ne pas tous les citer.

Il m'est impossible de remercier tous les amis qui m'ont accompagné dans mon travail tant ils sont nombreux.

Enfin, je remercie ma famille qui a toujours été très présente même si l'éloignement géographique m'en a séparé.

# RÉSUMÉ

La demande toujours croissante d'intégration, de sophistication, de fiabilité, et surtout de performance pousse à concevoir des systèmes multiprocesseurs de plus en plus complexes ce qui rend leur conception et leur maîtrise plus difficile. De nouvelles méthodes d'aide à l'optimisation logiciel sont recherchées pour permettre d'accroître les performances, la fiabilité, et la portabilité des applications exécutées sur de tels systèmes.

Le présent mémoire s'inscrit dans ce contexte de recherche. Une revue de littérature des méthodes pertinentes est réalisée. L'une d'entre elles est identifiée selon nos besoins et adaptée pour être utilisée dans le cadre du projet PULSE (Parallel Ultra Large Scale Engine). Ce projet vise à développer et concevoir un multiprocesseur SIMD (Single Instruction Multiple Data) complexe dédié au traitement temps réel de signaux numériques de type vidéo. Une série d'exemples d'applications types sont optimisées suivant la méthode retenue. Les performances obtenues sont analysées et comparées avec des mises en œuvre manuelles, pour ainsi évaluer les gains obtenus et les limites de la méthode.

Un outil de simulation du système PULSE a été développé pour permettre l'analyse et l'estimation de performances dynamiques. De plus, cet outil permet de visualiser les états d'une simulation pas à pas. Utilisé par de nombreux membres de l'équipe de développement du projet PULSE, l'outil de simulation a permis la mise au point de nombreuses applications et facilité l'apprentissage du maniement du multiprocesseur PULSE.

Ces travaux ont permis d'adapter des méthodes d'optimisation manuelles, d'élaborer des heuristiques d'estimation de performances, et de développer un environnement de simulation.

# ABSTRACT

The growing demand for integration, sophistication, reliability and particularly performance, pushes to design microelectronics systems of increasing complexity which makes the development and the use of these systems more difficult. This motivates the need for new optimization methods allowing to increase the performances, reliability and applications portability on these systems. The search for such methods is the object of this thesis.

This thesis begins with a literature review of relevant methods. A suitable method is selected and adapted to be used in the framework of the PULSE project. This project was aimed at developing a SIMD multiprocessor dedicated to real time processing of digital signals, including video signals. A series of examples representative of typical applications are optimized according to the selected method. The obtained performances are analyzed and compared with manual practical applications, evaluating obtained gains and setting the method limits.

A tool for simulating the PULSE system was developed to allow the analysis and the estimation of dynamic performances. This tool also allows to visualize step by step, the simulation states. This characteristic was very much used by a wide number of development team members on the PULSE project. Which permitted the debugging of many applications and facilitated the training of users for the PULSE multiprocessor.

This thesis is a step toward the development of methods for optimizing and estimating performances. In spite of significant progress, the goal of developing an automatic parallelising compiler for PULSE remained elusive. Such a compiler, whose development appears feasible, but beyond the scope of this project, could handle the hard task of extracting and exploiting parallelism available in the applications.

# TABLE DES MATIÈRES

page

page

# LISTE DES TABLEAUX

# LISTE DES FIGURES

page

# INTRODUCTION

La demande croissante de fonctionnalités et de performances dans les applications des systèmes électroniques modernes pousse les concepteurs à réaliser des prouesses en termes de puissance de calcul et de rapidité d'exécution. Pour illustrer cette observation et ainsi justifier le fait qu'il faut trouver des méthodes pour accroître les performances des calculateurs, utilisons un domaine très en vogue, le multimédia, et plus précisément le traitement d'images.

Le traitement numérique d'images est un domaine particulièrement vaste. Les applications sont nombreuses et variées. Elles vont de la gestion de bases de données d'images, à la compression d'images fixes ou animées en vue de leur mémorisation ou de leur transmission, pour aller jusqu'à l'analyse et la compréhension d'images. Toute manipulation d'images met en jeu un volume considérable de données. Celui-ci est lié à la dimension des images manipulées par l'application et croît lorsque, pour fournir un élément d'interprétation, il s'agit d'avoir accès à plusieurs images consécutives d'une séquence.

Généralement, chaque image passe par plusieurs niveaux de traitement. D'une manière schématisée, ceux-ci sont habituellement qualifiés de bas niveau, de niveau intermédiaire et de haut niveau. Les traitements de bas niveau les plus classiques sont les opérations au niveau du pixel qui créent de nouvelles images (algorithmes de seuillage, égalisation d'histogrammes, etc.), ainsi que les opérations de voisinage (filtrages) et les opérations construisant une nouvelle image à partir de deux images initiales (addition, soustraction, produit d'images, opération logique, etc.). Les traitements de niveau intermédiaire partent d'images et aboutissent à des primitives, c'est-à-dire à des représentations condensées de l'information. Les traitements de haut

niveau exploitent les primitives résultant du niveau intermédiaire pour produire de nouvelles primitives et visent à fournir une interprétation de l'image. Il convient cependant de noter que la frontière entre traitement de bas niveau et traitement de niveau intermédiaire tend de plus en plus à disparaître. Dans les algorithmes d'analyse d'images, le terme traitement de bas niveau englobe toutes transformations d'une matrice de pixels en une matrice de primitives.

À chacun de ces niveaux et plus particulièrement au niveau le plus bas, le traitement est dispendieux en calculs. Aussi, le volume de calculs inhérents à de nombreuses applications de traitement d'images excède la capacité de traitement de toute machine séquentielle. Le problème est encore plus critique dans un contexte temps réel où une séquence d'images doit être traitée dans un intervalle de temps très court.

La structure de données couramment manipulée en traitement d'image est le tableau bi-dimensionnel qui représente la distribution spatiale des valeurs de luminance (niveaux de gris ou couleur). Dans de nombreux algorithmes, le traitement à réaliser consiste en l'application d'un ensemble d'opérations en chaque point de la structure de données image. Ces opérations peuvent aussi être réalisées en parallèle pour chaque élément du tableau.

Les systèmes matériels utilisés pour réaliser ce type d'opérations s'étendent de la simple carte (type carte PC) aux énormes super-calculateurs. Ils tirent parti d'une vaste gamme de composants allant du processeur commercial aux circuits intégrés VLSI (Very Large Scale Integration) spécifiques. En conséquence, les machines parallèles couramment employées pour le traitement des images sont aussi bien des architectures à usage général que des architectures dédiées. Les architectures parallèles à usage général sont intéressantes pour leur flexibilité ; elles supportent une large gamme d'algorithmes avec des performances intéressantes qui permettent de rencontrer les spécifications de l'application considérée. Quant aux architectures spécialisées ou dédiées, elles offrent généralement des performances supérieures en terme de vitesse de traitement, avec souvent comme objectif un traitement en temps réel, et elles disposent de mécanismes

d'entrées/sorties dédiées à leur environnement cible. Le champ d'application de ces machines reste dédié et leur flexibilité généralement limitée.

Plusieurs raisons militent en faveur d'une approche "architecture parallèle spécialisée" où le système est "taillé sur mesure" pour une classe de problèmes ; ce sont les caractéristiques du problème qui déterminent la structure et le contrôle du système.

Les principales raisons sont les suivantes :

- Il est souvent nécessaire d'atteindre une grande rapidité de traitement (système temps réel).
- Le système doit souvent être compact (matériel embarqué). Il est dans ce cas primordial d'optimiser l'architecture pour une classe de problèmes, afin d'en minimiser l'encombrement.
- Les progrès de la technologie des circuits intégrés VLSI ont rendu possible la réalisation de systèmes ayant un grand nombre de processeurs, où chaque processeur intègre des fonctionnalités de traitements minimales ou réalise une fonction particulière. Il est ainsi plus facile de dimensionner le système.

Une solution naturelle conduit à la réalisation de multiprocesseur approchant les performances des super-ordinateurs, tout en ayant le coût des monoprocesseurs. Les composants disponibles commercialement ne permettent généralement pas d'atteindre ce rapport prix/performance. Grâce à la conception de circuits intégrés dédiés (ASIC, *Application Specific Integrated Circuit*) et les nouvelles technologies, plusieurs processeurs peuvent être intégrés dans une puce pour augmenter la capacité de traitement. Parmi les architectures existantes, la structure SIMD[1] (Single Instruction stream, Multiple Data stream) est bien adapté aux problèmes du traitement d'images. Les traitements de bas et de moyen niveaux, qui sont caractérisés par l'application d'opérations identiques à tous les pixels, sont bien adaptés à cette classe d'architecture. Un parallélisme très fin peut-être exploité par ce type de configuration.

---

[1] D'après la classification de Flynn (1972).

Par ailleurs, l'idée de base qui consiste à associer un processeur élémentaire (PE) par pixel de l'image peut s'avérer trop coûteuse, surtout dans le cas de réseaux 2D. Le nombre de processeurs est aussi généralement inférieur à la dimension de l'image, ce qui implique un découpage en blocs et rend la mise en œuvre de l'application plus complexe. Dans ce cas, la parallélisation des applications est dirigée par les données.
C'est en suivant ces considérations que le projet PULSE (Parallel Ultra Large Scale Integration) a émergé (P. Mariott, 1998). Ce projet se propose de réaliser un système permettant des opérations en temps réel pour certaine classes d'applications de traitement des signaux numériques. Ce système possède une structure SIMD linéaire basée sur des processeurs complets, possédant des jeux d'instructions complets et des chemins de données complexes et divers. Ce projet va constituer une base de référence et d'expérimentation pour les sujets traités dans ce mémoire.
Cette architecture parallèle, bien que spécialisée, est programmable. En effet, la programmabilité permet une évolution rapide des fonctionnalités par le développement de nouvelles applications. Mais il est important que la maîtrise de cette programmation ait été prise en compte dès la conception de l'architecture, afin d'éviter que l'utilisation d'un langage évolué s'avère difficile et peut conduire à des mises en œuvre inefficaces.
L'objectif des recherches actuelles est donc, d'une part, d'élaborer un moyen d'exploiter efficacement ces machines, c'est-à-dire de se rapprocher, pour chaque application, de la puissance théorique maximale offerte par la machine, et d'autre part de pouvoir concevoir, vérifier et maintenir les programmes le plus facilement possible. Ces problèmes ont été étudiés dans le cadre du traitement séquentiel et, il est nécessaire d'y apporter également une réponse pour la programmation parallèle.

C'est dans cette optique qu'une adaptation des programmes doit être réalisée pour exploiter au maximum les performances potentielles de l'architecture. Cette *optimisation* des programmes est basée sur deux aspects : le problème du placement des données dans le temps (problème temporel, problème d'ordonnancement) ; et le problème du placement des données au sein des processeurs (problème spatial, problème

d'allocation). Une étude approfondie de ces deux aspects est nécessaire pour la mise en œuvre d'outils de parallélisation automatiques et c'est précisément l'objet de ce mémoire. Compte tenu de la complexité du problème, ce mémoire se limite à présenter une méthode de parallélisation manuelle systématique adaptée à l'architecture PULSE. Cette méthode pourra par la suite être automatisée lors d'une intégration dans un outil de compilation de haut niveau. Pour permettre de comparer, et d'analyser finement les performances obtenues par les méthodes de parcellisation un outil d'analyse a été développé. Cet outil a en plus subit de nombreuses évolutions qui lui ont permit d'accroître son champ d'action et ainsi devenir un outil de simulation et d'estimation de performances dans un milieu de conception de système VLSI.

Ce mémoire est composé de trois chapitres. Le premier (chapitre I) décrit la méthode de parallélisation de programmes appliquées au système PULSE. Il est lui-même découpé en deux parties : la première est une revue de littérature qui définit les concepts, l'état de l'art, et tante de différencier les approches existantes ; la seconde partie détaille une des approches, la mieux adaptée au système PULSE. Elle passe en revue les règles d'optimisation de cette méthode, appelées les stratégies de transformation. L'objectif de cette partie est de présenter quelques-unes de ces stratégies, en précisant les hypothèses sous-jacentes à leur utilisation.

Une fois les méthodes de parallélisation de programme assimilées, le chapitre suivant (chapitre II) traite plusieurs cas réels en insistant sur les performances atteintes avec le système PULSE. Chacun des cas étant unique, une comparaison peut-être réalisée entre eux et avec des applications réelles. Ce chapitre se termine sur une synthèse des performances obtenues et des améliorations à apporter aux méthodes de parallélisation définies au chapitre précédent.

Puisqu'il est question de performance, le dernier des trois chapitres (chapitre III) décrit la manière avec laquelle les performances ont été extraites. Un outil d'analyse a été développé à cette fin dans le cadre de ce mémoire. Cet outil permet de visualiser le fonctionnement des programmes pas a pas et d'estimer les performances dynamiques du

système PULSE. Deux versions de cet outil ont été réalisées. Ce chapitre décrit leur fonctionnement et leur utilisation.

Enfin, une conclusion clôture ce mémoire. Elle replace ce projet dans le contexte de la boucle d'optimisation globale de système imaginée lors de mon entrée à l'École Polytechnique. Nous y proposons une manière de fermer cette boucle qui reste ouverte jusqu'à présent.

# CHAPITRE I

# À LA RECHERCHE DE PERFORMANCES

C'est maintenant un lieu commun que de constater que les processeurs modernes font et vont faire de plus en plus appel au parallélisme pour améliorer leurs performances. Ceci est vrai dans toute la gamme des puissances, des monoprocesseurs jusqu'aux super-ordinateurs. Dans le cas des monoprocesseurs, le parallélisme exploitable est limité. Il se trouve essentiellement à l'intérieur d'une instruction ou d'une séquence de quelques dizaines d'instructions, si bien qu'il peut être caché au programmeur. Cela n'est pas suffisant pour les super-ordinateurs, surtout dans leurs variantes massivement parallèles. En apparence, ceci nécessite la collaboration de l'utilisateur, dû au fait que les interactions entre les traitements ne sont pas faciles à mettre en évidence, alors que le programmeur en a généralement une connaissance au moins intuitive.

La programmation d'une machine parallèle peut ainsi se faire à bas niveau, par exemple en ajoutant à un langage classique les instructions représentant les *opérations* caractéristiques de la programmation parallèle (i.e. création de tâches, synchronisation, communication) et en supprimant celles qui n'ont pas lieu d'exister dans ce contexte. Dans le cas du projet PULSE, ce niveau de programmation se situe dans le langage appelé Cpulse (M. Achim, C. Bonello et V. Van Dongen, 1997). Cette approche a le grand mérite de la transparence : le programmeur contrôle dans le détail l'implantation de son algorithme. Il s'agit donc de la méthode de choix pour le développement de nouveaux algorithmes nécessitant des mécanismes particuliers et précis.

En revanche, en tant que méthode de production générale d'applications, elle possède des défauts, tel que la difficulté de mise au point des applications, due essentiellement au caractère non-déterministe de nombreuses architectures parallèles. De plus, il y a des

cas où la manière d'exploiter le parallélisme présent dans l'application n'est perçue clairement que par le programmeur. Un grand nombre de chercheurs se sont préoccupés, depuis les travaux pionniers de David J. Kuck (1972), de développer des outils d'aide à la programmation parallèle.

Nous pouvons imaginer qu'un compilateur, plus sophistiqué que les compilateurs usuels (appelé *compilateur-parallèliseur*), se charge de rechercher dans les programmes le parallélisme implicite et de l'exploiter sur la machine cible. Ainsi, le programmeur pourrait définir son application en utilisant des langages sources sans séquencement (langage fonctionnel, à flot de données, logique, etc.) ou des langages de haut niveau beaucoup plus simples à utiliser, tel que HPCP dans le cas du projet PULSE (N. Belanger. 1997). Cette approche correspond à ce que nous appelons la parallélisation automatique au sens le plus courant du terme. La parallélisation automatique se donne ainsi trois objectifs :

- faciliter le développement de nouvelles applications,
- rechercher un optimum de performance pour une application sur une architecture parallèle cible,
- assurer la portabilité des programmes entre les architectures parallèles.

Au près des groupes de recherche les deux premiers objectifs sont très en vogue. Il est très tentant de développer un algorithme en version séquentielle, avec tous les moyens et environnements existants, et de ne passer à la version parallèle que lorsque l'essentiel de la mise au point a été fait, afin de laisser un compilateur-parallèliseur rechercher le meilleur partitionnement atteignant les performances souhaitées. De plus, ces objectifs permettent de rendre explicite des aspects parallèles d'un programme qui n'ont pas été décelés par le programmeur.

Bien qu'éloigné de nos objectifs immédiats, l'importance du troisième objectif croit avec l'apparition à cadence rapide de nouvelles architectures. Les ordinateurs séquentiels se ressemblent tous, au moins du point de vue conceptuel, et il est tout à fait concevable de mettre au point un algorithme sur un IBM-PC pour l'exécuter sur un IBM-3090. En

revanche, la portabilité d'un programme entre des architectures parallèles est très délicate et complexe.

Les techniques de parallélisation sont en pleine évolution[2]. Cependant, les méthodes utilisées ont un air de famille. Le travail se fait toujours en deux étapes principales. Une phase d'analyse permet de rassembler des informations globales sur la façon dont les diverses instructions interagissent entre elles. Nous parlons usuellement d'*analyse sémantique*, pour montrer que cette phase s'intéresse aux traitements exécutés par le programme et non pas seulement à la façon dont il est écrit.

La deuxième étape consiste à exploiter les résultats obtenus dans la phase d'analyse pour guider la génération du programme parallèle. Elle peut être vue comme l'identification dans le programme source des formes caractéristiques des opérations possédant un parallélisme implicite. Naturellement, la reconnaissance de ces formes est d'autant plus facile (i.e. il y a d'autant moins de modèles à considérer) que la phase d'analyse a fourni une représentation plus synthétique du programme original.

La suite du chapitre décrit des méthodes de parallélisation de programme. Une de ces méthodes est détaillée en profondeur en vue d'être appliquée dans le cadre du projet PULSE. Ce chapitre se répartit en quatre sections. La première (section 1.1) définit le principe traditionnel de la parallélisation de programmes et propose un diagramme de flot de ce processus, suivi d'une comparaison des différentes techniques. Les deux sections suivantes (sections 1.2 et 1.3) détaillent les différentes étapes de ce processus de parallélisation. Décrivant, pour la première, les mécanismes d'analyse des dépendances, et pour la seconde, la technique de transformation de boucles. Enfin, une conclusion clôture cette présentation des méthodes de parallélisation. Tout au long de ce chapitre, l'architecture soutenue par le projet PULSE guide nos critères de sélection.

---

[2] La parallélisation automatique est actuellement étudiée par de nombreux groupes de recherche et plusieurs outils de parallélisation ont été écrits: SUIF à l'université de Stanford en Californie, PIPS à l'École Nationale Supérieure des Mines de Paris, la bibliothèque Omega à l'université du Maryland, PooPo à l'université de Passau en Allemagne, le compilateur Paradigm à l'université de l'Illinois, PAF à l'université de Versailles, pour n'en citer que quelques-uns.

## 1.1. Le principe traditionnel de la parallélisation

L'analyse et l'interprétation d'un programme en vue de son exécution parallèle ne peut pas s'appuyer sur la notion d'instruction, parce que c'est entre les différentes répétitions d'une même instruction que nous espérons trouver le plus de parallélisme. Chaque exécution d'une instruction doit être considérée comme une entité distincte, une *opération*. Un programme doit être vu comme un ensemble d'opérations donné *a priori*, au moins conceptuellement. Mais un programme ne se réduit pas à l'ensemble de ses opérations, pas plus qu'une sonate ne peut se jouer en frappant simultanément l'ensemble des notes de la partition. Ici, comme en musique, l'ordre d'exécution des opérations est primordial. Un programme doit donc être représenté comme un ensemble ordonné d'opérations. L'ordre d'exécution est caractérisé par leur séquence caractérisée par des successions et imbrications de boucles et de conditions.

C'est pour cette raison que les emboîtements de boucles sont au cœur de la stratégie de parallélisation des compilateurs-paralléliseurs des ordinateurs parallèles contemporains. Leur importance, en termes d'applications, est claire : pour de nombreux programmes scientifiques, le temps passé dans quelques boucles constitue une grande fraction du temps d'exécution total, et le parallélisme potentiel en est souvent considérable (P. Boulet, A. Darte, G-A. Silber et F. Viven, 1997). D'une part, la restriction de l'analyse aux emboîtements de boucles ne constitue pas une limitation trop pénalisante. Cela permet de traiter une vaste classe d'applications (qui contient notamment la plupart des applications de traitement du signal ou d'algèbre linéaire numérique). La parallélisation des boucles a fait l'objet de nombreuses recherches (J.R. Allan et K. Kennedy, 1984 ; U. Banerjee, Fév. 1993 ; U. Banerjee, 1993, R. V. Hanxleden et K. Kennedy, 1992).

D'autre part, la structure régulière et répétitive des emboîtements de boucles facilite la mise en œuvre de techniques d'analyse des dépendances et la recherche de fonctions d'ordonnancement et d'allocation. Le problème général de l'ordonnancement optimal d'un système de tâches sur une machine parallèle est connu comme un problème difficile, du essentiellement aux communications, même s'il est supposé disposer d'un

nombre illimité de processeurs. Pourtant, dans le cas des boucles imbriquées, il est possible de définir des algorithmes d'ordonnancement efficaces et d'établir leur optimalité en utilisant des outils mathématiques. Les caractéristiques principales qui différencient les techniques d'ordonnancement des emboîtements de boucles par opposition aux systèmes généraux de tâches sont :

- *La cyclicité* : La structure régulière et répétitive d'un emboîtement de boucles permet de ne considérer que les dépendances pour la recherche d'un ordonnancement optimal.
- *La généricité* : Il est possible de trouver des ordonnancements génériques, valables pour toutes les valeurs des paramètres.

Par déduction, l'étude de la parallélisation des emboîtements de boucles attirera particulièrement notre attention dans ce mémoire.

### 1.1.1. Les fondations de la parallélisation

La figure 1.1 présente le processus de parallélisation d'un programme séquentiel effectué par un compilateur-paralléliseur contemporain. Ce processus peut-être décomposé comme suit, en plusieurs étapes interdépendantes :

- L'*analyse des dépendances* consiste a créer un graphe, appelé *domaine d'itérations*, représentant les contraintes sur l'ordre d'exécution des opérations déduites à partir des relations entre les données utilisées par chaque opération.
- L'étape d'*ordonnancement* utilise le domaine d'itérations pour construire un *graphe de communication* qui associe un temps d'exécution à chaque instance de chaque opération. Ainsi un moment d'exécution est assigné à chaque opération. Le but de cette étape est de minimiser la latence globale, qui se trouve être la concaténation des tout les temps d'exécutions de chacune des opérations ayant des interdépendances.

- L'étape d'*allocation et placement* cherche à répartir les données et calculs sur une *grille virtuelle de processeurs* de taille et de dimension infinie. Cette recherche se base sur le graphe de communication et le domaine d'itérations pour trouver une répartition où les communications inter-processeurs sont minimales.
- Le *partitionnement* a pour but de découper la grille virtuelle de processeurs et de répartir les morceaux sur la *grille physique de processeurs*. Cette tâche est effectuée en répartissant la charge de travail de chaque processeur le plus équitablement possible, et en respectant les travaux effectués lors des phases d'ordonnancement et d'allocation précédentes.
- La dernière étape est la *génération de code*. Cette dernière a pour but de réécrire, dans un langage de programmation donné, la nouvelle représentation des boucles et tableaux du programme initial pour chacun des processeurs physiques avec un parallélisme explicite.

Les étapes d'ordonnancement et d'allocation ne sont pas indépendantes et c'est le problème majeur que toute technique de compilation a à affronter. Deux tâches ordonnancées au même moment ne peuvent pas être allouées au même processeur sans perte de parallélisme.

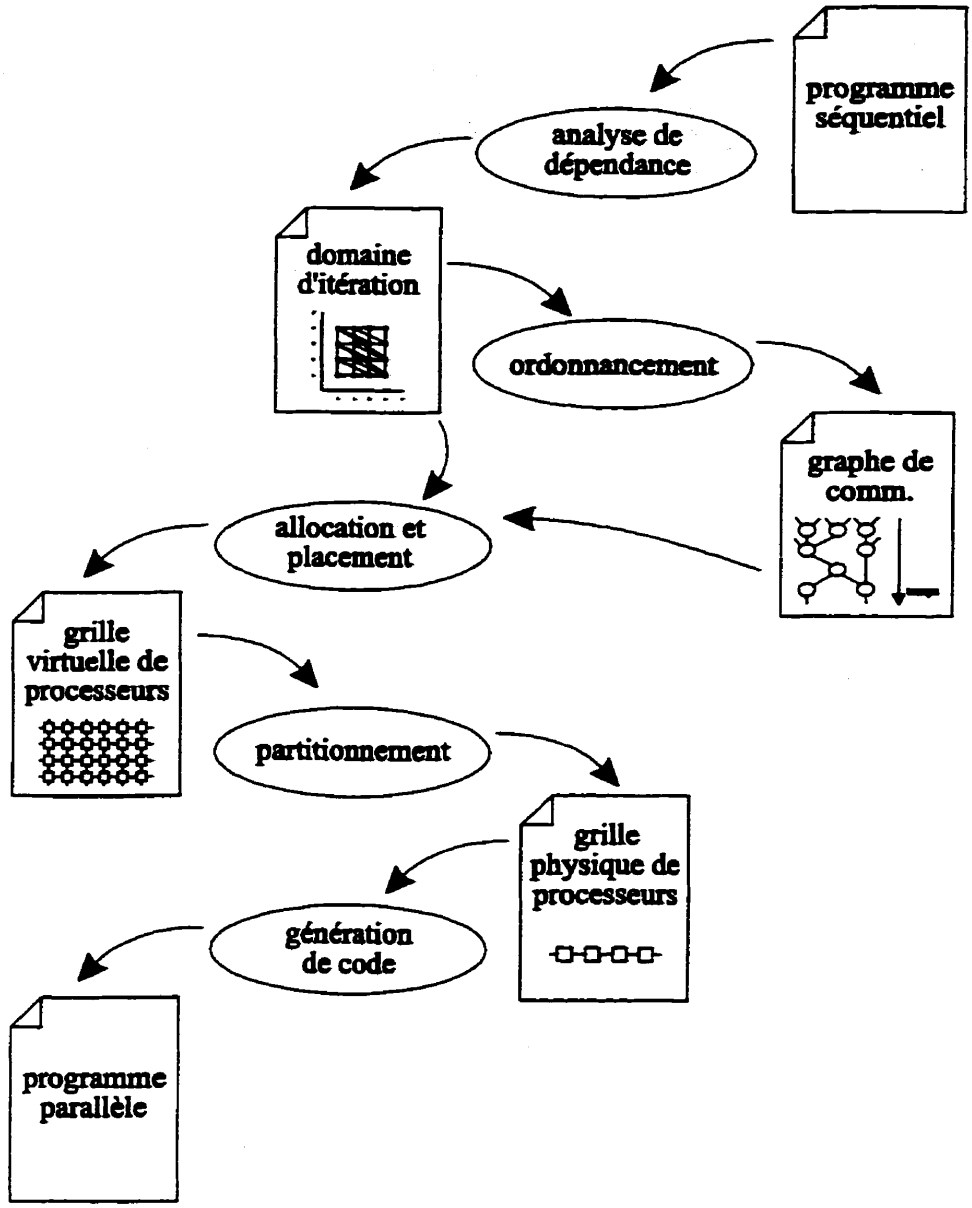


*Figure 1.1: Principe de la parallélisation*

Il est à noter que la figure 1.1 n'est pas entièrement exacte. Dans un cas réel, des optimisations de code avant la première étape et après la dernière peuvent être réalisées. Ces optimisations sont de bas niveau et servent à éliminer le code mort, à normaliser les constantes et indices, et à supprimer les fausses dépendances.

### 1.1.2. Notations et terminologies

Avant de se lancer dans le vif du sujet, une identification et une définition des principaux termes et règles de notation utilisées est réalisée. Il est à noter que les restrictions énoncés lors des définitions ci-dessous ont pour but de restreindre le champ d'investigation et ainsi de permettre de proposer une méthode et d'exposer sont déroulement dans le cadre d'un mémoire de maîtrise.

**Définition 1.1 - Type de données :** À tout moment, sauf si spécifié, lorsqu'il est question de données, il est sous-entendu que les données se trouvent sous une représentation de scalaires ou de tableaux multidimensionnels de scalaires. L'allocation de mémoire est supposée faite de façon unique (sans surnommage ou *aliasing*), c'est-à-dire que deux éléments d'un même tableau correspondent à la même adresse mémoire si et seulement si ce sont les éléments d'un même tableau et que les indices sont identiques.

**Définition 1.2 - Paramètres de structure :** Le programme peut dépendre de paramètres, définis une et une seule fois dans le programme par une instruction d'entrée/sortie ou par une relation avec d'autres paramètres de structure déjà définis.

**Définition 1.3 - Instructions :** Les instructions sont les opérations d'affectation portant sur un scalaire ou un élément d'un tableau et celles supportées implicitement par l'architecture matérielle cible. De la même façon que pour les compteurs de boucle, les fonctions d'accès aux tableaux sont restreintes aux fonctions affines composées par des paramètres de structure et les indices des boucles englobantes.

**Définition 1.4 - Boucle :** La notion de boucle est construite à partir des opérateurs du type `for` avec un compteur explicite $i$, dont les bornes inférieures $L$ et supérieures $U$ sont-elles aussi explicites. Le *corps* de la boucle, noter $H$, contient toutes les instructions effectuées par la boucle. Si le corps ne contient pas de branchement hors de la boucle et si le pas du compteur est 1, alors les bornes du compteur sont atteintes.

**Définition 1.5 - Nid de boucles :** Lorsque le corps d'une boucle contient lui-même une autre boucle, cet emboîtement est appelé un *nid*. Notons que cet emboîtement est nécessairement fini. Si le corps de toute boucle est une boucle ou une séquence d'assignations[3], le nid est dit *parfait*. Un exemple de la structure d'un nid parfait est codé ci-dessous. Cette structure est reprise de celle utilisée par Uptal Banerjee (1988). Il est à noter que les bornes des boucles ne sont pas constantes dans ce modèle. Cette non-constance des bornes et des indices peut amener à avoir des *vecteurs de distance de dépendances* (voir définition 1.10 en page 29) non constants, ce qui rend le problème de la recherche de parallélisme dans les boucles NP-complet (M.L. Dowling, 1990). La *profondeur* d'un nid de boucles est le nombre de boucles emboîtées. De façon analogue, la profondeur d'une instruction est le nombre de boucles englobantes. Par abus de notation, les termes nid et nid de boucles seront utilisés indifféremment.

```
for i1=n1, N1 {
   for i2=n2(i1), N2(i1) {
      ...
         for in=nn(i1, i2,...in-1), Nn(i1, i2,...in-1) {
            H(i1, i2,...in)
         }
      ...
   }
}
```

*Figure 1.2: Le nid de boucles parfait*

---

[3] Les assignations portent sur des scalaires ou des tableaux multidimensionnels de scalaires uniquement (définition 1.1).

**Définition 1.6 - Vecteur d'itération :** Le vecteur d'itération d'un nid est le vecteur formé par les compteurs englobants, du compteur de la boucle externe jusqu'au compteur de la boucle interne. Celui du nid précédent est donc $i=(i_1,i_2)$. Il est clair que la dimension du vecteur d'itération est égale à la profondeur du nid de boucles correspondant.

**Définition 1.7 - Domaine d'itération :** Nous appelons domaine d'itération (voir figure 1.3a) $G$ d'un nid, l'ensemble des valeurs prises par le vecteur d'itération au cours de l'exécution, tel que

$$G=\{(i_1,...,i_n) \mid L_1 \le i_1 \le U_1,...,L_n(i_1,...,i_{n-1}) \le i_n \le U_n(i_1,...,i_{n-1})\} \quad (1.1)$$

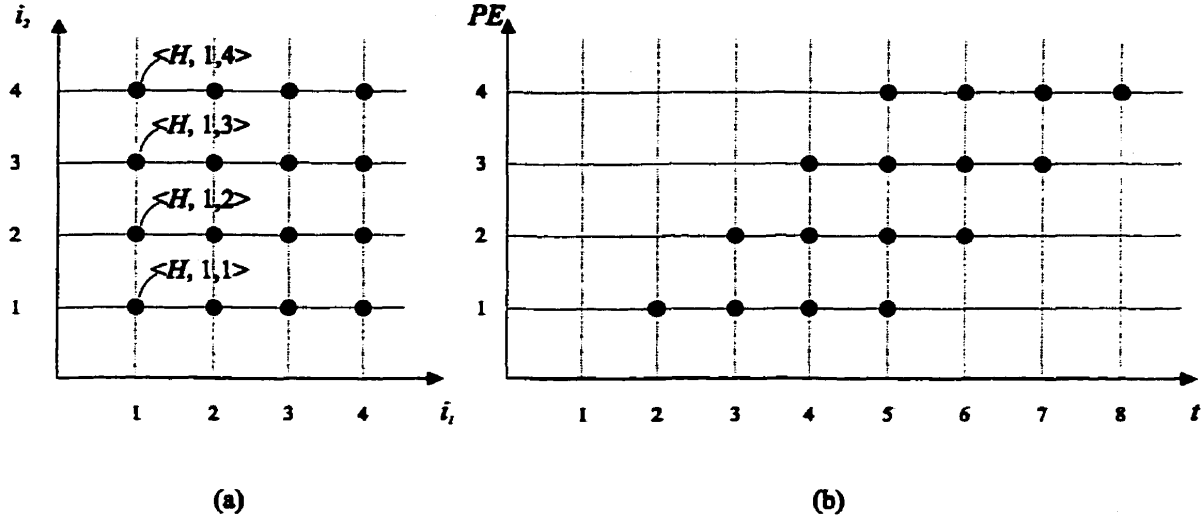


*Figure 1.3: Domaine d'itération vs. allocation et placement des données sur des processeurs pour un domaine à deux dimensions* $\langle H, i_1, i_2 \rangle$

L'exécution d'un nid donnera lieu à autant d'*instances* de $H$, ou d'opérations, qu'il y a de valeurs possibles pour le vecteur d'itération. Ainsi, chaque point du domaine d'itération peut être identifié avec une instance de $H$, et $G(H)$ représente le domaine d'itération associé au corps de boucle $H$. Une opération peut alors être référencée sans ambiguïté

par le nom de l'instruction et par la valeur du vecteur d'itération ; la notation $\langle H,i\rangle$ est utilisée pour désigner une instance de $H$ pour une valeur $i$ du vecteur d'itération. Par abus de notation, $\langle H,i\rangle$ et $\langle H,i_1,...,i_n\rangle$ seront utilisés indifféremment.

D'après la définition, le domaine d'itération est caractérisé par un ensemble d'inégalités linéaires et affines correspondant aux bornes des vecteurs d'itérations. Étant affines, ces équations peuvent être représentées sous une forme matricielle par les deux relations suivantes : $S_L I \geq l$ et $S_U I \leq u$, où $S_L$ et $S_U$ sont des matrices triangulaires de taille $n$ par $n$ ; $I$ est un vecteur d'itération de taille $n$ par 1 ; $l$ et $u$ sont des vecteurs d'entiers de taille 1 par $n$. En les combinant, nous obtenons un système qui régit les limites haute et basse des vecteurs d'itération d'un nid de boucle

$$(S,c)=\left(\begin{bmatrix}\frac{S_L}{-S_U}\end{bmatrix},\begin{bmatrix}l\\-u\end{bmatrix}\right) \tag{1.2}$$

où $(S,c)$ est la représentation complète des bornes du domaine d'itération.

**Définition 1.8 - Ordre lexicographique** (noté $\prec$) : Les couples $i=(i_1,\cdots,i_n)$ et $j=(j_1,\cdots,j_n)$ satisfont $i \prec j$ si et seulement si il existe un entier $k$, tel que $1 \leq k \leq n$ et que $i_1 = j_1,\cdots,i_{k-1} = j_{k-1}$ et $i_k < j_k$.

**Définition 1.9 - Matrice de transformation** : Matrice qui permet de réaliser des transformations de boucles et les instructions contenues dans un nid de boucles. La section 1.3. y est entièrement consacrée.

### 1.1.3. Histoire d'algorithmes

Cette section passe en revue les principaux algorithmes de détection du parallélisme : ceux de Lamport, d'Allen et Kennedy, de Wolf et Lam, de Feautrier, de Darte et Vivien, et de Lim et Lam. Cette liste permet de choisir la (ou les) méthode qui se rapproche le plus possible de nos préoccupation. Il est bien entendu que cette liste n'est pas exhaustive, seul les noms des personnes ayant apportés une contribution significative dans le domaine son cité. Ensuite une synthèse sous forme d'un tableau de comparaison est présenté en guise de conclusion de cette sous-section.

**L'algorithme de Lamport** (1974) : Leslie Lamport est connu en parallélisation pour sa *méthode de l'hyperplan*. Cette méthode considère des nids de boucles parfaits avec des vecteurs de distances constantes. Il crée un hyperplan qui lui permet de constituer sa matrice unimodulaire de transformation. Cette méthode, qui repose sur la lexico-positivité des vecteurs de distance, est en fait un algorithme de construction d'ordonnancements linéaires représenté sous forme d'hyperplan fortement séparateur. Elle considère des nids dont les dépendances sont représentées par des vecteurs dirigés dont seul la première composante peut ne pas être constante. Il a également présenté une méthode généralisée qui s'appelle : la *méthode du plan* qui permet d'avoir des vecteurs de direction constants.
Leslie Lamport a remarqué que ses ordonnancements est un problème de programmation linéaire, mais il ne connaît pas de méthode pratique permettant de le résoudre dans un cas général. Son algorithme est optimal pour des nids de boucles dont les dépendances sont uniformes et dont les graphes et dépendances sont fortement connexes.

**L'algorithme de Allen et Kennedy** (1987) : Cet algorithme a été originellement proposé pour la vectorisation de boucles. Il a été ultérieurement étendu pour maximiser le nombre de boucles parallèles générées et pour minimiser le nombre de

synchronisations requises. Cet algorithme prend en entrée un *graphe de dépendances réduit par niveaux* (GDRN). Il repose sur les deux propriétés suivantes :

- Une boucle est parallèle si elle ne porte pas de dépendance, c'est-à-dire s'il n'existe pas de dépendance dont le niveau soit égal à la profondeur de cette boucle et concernant une instruction étant à l'intérieur de cette dernière.
- Toutes les itérations du corps de boucles $H$ peuvent être exécutées avant que ne soit évaluée une seule instance de l'instruction $H^{:}$, si le graphe de dépendance réduit ne contient pas de dépendances de $H^{:}$ vers $H$.

La première propriété permet de marquer les boucles : séquentielles ou parallèles. La seconde suggère que la détection du parallélisme puisse être menée indépendamment dans les différentes composantes du GDRN. L'extraction du parallélisme se fait au moyen d'une simple distribution de boucles déroulées. Cet algorithme détecte autant de boucles englobantes parallèles que possible pour chaque instruction du code original. Ceci le rend optimal pour générer du code parallélisé dans lequel les instances d'une instruction quelconque sont énumérées avec les mêmes boucles que dans le code initial. Autrement dit, aucun algorithme ne peut trouver plus de parallélisme dans un graphe de dépendance réduit par niveaux que n'en trouve cet algorithme.

**L'algorithme de Wolf et Lam** (1991) : Michael Wolf et Monica Lam ont proposé un algorithme qui accepte en entrée une approximation des dépendances par des vecteurs dirigés. Leurs travaux unifient les algorithmes antérieurs basés sur des opérations matricielles élémentaires, telles que des torsions (ang. *skew*), l'inversion (ang. *interchange*) et le renversement (ang. *reversal*) de boucles, en se plaçant dans le cadre unique des transformations unimodulaires valides. L'algorithme proposé généralise la méthode du plan de Lamport.

L'algorithme de Wolf et Lam a pour objectif de construire des ensembles de boucles totalement permutables. Ces boucles totalement permutables sont à la base de toutes les techniques de partitionnement (ang. *tiling*). Le partitionnement est utilisé pour exposer du parallélisme à grain moyen. Cependant, un ensemble de $n$ boucles imbriquées

totalement permutables peut toujours être transformé en un ensemble de $n$-1 boucles parallèles imbriquées entourées d'une boucle séquentielle. Les boucles permutables sont donc aussi liées à la parallélisation à grain fin.
Wolf et Lam cherchent, au moyen de transformations unimodulaires, le plus grand[4] ensemble de boucles externes totalement permutables[5]. Ils considèrent récursivement les dimensions restantes et les vecteurs non satisfaits par ces boucles. Pour que les boucles soient permutables, il faut qu'elles soient imbriquées. Wolf et Lam ne considèrent donc que des nids parfaitement imbriqués. Cet algorithme trouve un maximum de parallélisme à gros grain.

**L'algorithme de Feautrier** (1992) : Contrairement aux algorithmes précédants, Paul Feautrier n'a pas essayé de trouver du parallélisme dans une représentation approchée des dépendances : il ne s'occupe implicitement que de dépendances exactes. Dans le cadre des programmes à contrôle statique, il a d'abord décrit un algorithme de construction d'ordonnancements affines unidimensionnels, puis multidimensionnels. Ces algorithmes s'appuient sur une représentation des dépendances par des polyèdres. Cette représentation lui permet de paralléliser des nids parfaits et des nids débalancés contenant des dépendances affines, ce qui rend cet algorithme plus performant pour la recherche de dépendances exactes dans des nids de boucles. Les seuls résultats d'optimalité communs pour cet algorithme concernent les nids de boucles uniformes à une ou plusieurs instructions.

**L'algorithme de Darte et Vivien** (1996) : Alain Darte et Frédéric Vivien proposent une simplification de l'algorithme de Feautrier au niveau de la représentation des dépendances, ce qui leur permettent de réaliser des transformations affines sur les boucles. Elles possèdent toutefois des restrictions sur leurs formes. Cet algorithme est

---

[4] Nous pouvons compter sans risque le nombre de boucles du nid, car les transformations unimodulaires ne modifient pas ce nombre.
[5] Dans le cadre de boucles dont tous les incréments sont positifs, les $k$ boucles externes sont totalement permutables si les $k$ premières composantes de tous les vecteurs de directions sont positives.

optimal pour détecter un maximum de parallélisme si les dépendances peuvent être approximées par des polyèdres. Ceci résulte en un algorithme plus simple et plus performant que celui de Feautrier pour la permutation de boucles, la minimisation des synchronisations entre les boucles, et la génération de code. Des restrictions au niveau des boucles à paralléliser impliquent que cet algorithme est globalement moins exact sur la recherche de parallélisme.

**L'algorithme de Lim et Lam** (1997) : Les algorithmes précédemment décrits cherchaient tous à exhiber le maximum de parallélisme. Celui de Amy Lim et Monica Lam recherche plutôt le maximum de parallélisme exposable sans dépasser un volume donné de synchronisations. Cet algorithme travaille dans les mêmes conditions que celui de Feautrier et peut donc être considéré comme une extension : il s'intéresse aux dépendances exactes des nids à contrôle statique dont les fonctions d'accès sont affines. Il produit un placement des calculs et un ordonnancement de ceux-ci au moyen de fonctions affines. Plutôt que de résoudre des programmes linéaires comme l'algorithme présenté par Feautrier, celui de Lim et Lam calcule des noyaux de matrices. Cet algorithme recherche le parallélisme qui peut-être obtenu avec une qualité croissante de synchronisation : aucune synchronisation, $O(1)$ synchronisations, $O(N)$ synchronisations, etc., où $N$ est le paramètre de taille du domaine d'itération. Lim et Lam affirment que leur algorithme explore le parallélisme de façon optimale : « L'algorithme trouve tous les degrés de parallélisme du programme étudié, tout le parallélisme étant à grain aussi gros que possible ». Il faut toutefois rajouter la condition suivante pour que cette affirmation soit valide : « parmi les algorithmes produisant des transformations affines ».

*Tableau 1.1: Comparaison entre les différents algorithmes de parallélisation, adapté de P. Boulet (1997)*

| Algorithme | dépendance | transformation | parallélisme | synchro-nisation | génération de code | permu-tation |
|---|---|---|---|---|---|---|
| **Lamport** | simple, imparfait | unimodulaire | optimal | oui | très simple | oui |
| **Allen-Kennedy** | simple, imparfait | distribution | optimal | oui | très simple | non |
| **Wolf-Lam** | vecteur de direction, parfait | unimodulaire | optimal | non | simple | oui |
| **Feautrier** | affine (exact), imparfait | affine | sub-optimal | non | compliqué | non |
| **Darte-Viven** | polyèdre, parfait | (shift linear) | optimal | partiel | un peut compliqué | oui |
| **Lim-Lam** | affine (exacte), imparfait | affine | sub-optimal | | | Oui |
| **Uptal Banerjee** | simple, imparfait | unimodulaire | optimal | | simple | Oui |

Le tableau 1.1 expose les différences majeures entre les six algorithmes discutés plus celui choisi dans le cadre de ce mémoire. Nous avons d'un côté des algorithmes utilisant des techniques puissantes sur une représentation exacte des dépendances mais sans garantie sur leur efficacité. De l'autre côté, nous avons des algorithmes plus simples travaillant sur une représentation approchée des dépendances et dont l'(in)efficacité est bien définie. Schématiquement, dans le contexte de ce mémoire, nous recherchons un algorithme travaillant sur une représentation approchée des dépendances afin de pouvoir traiter des programmes plus généraux que ceux étudiés par l'algorithme de Feautrier. Nous voudrions utiliser des techniques suffisamment puissantes pour extraire tout le parallélisme contenu dans la représentation des dépendances choisie. Cependant, nous

aimerions générer des codes aussi simples que faire se peut. Nous aimerions que cet algorithme idyllique trouve du parallélisme à chaque fois que Allen et Kennedy ou Wolf et Lam en trouvent. Pour ce faire et pour ne pas entrer dans des considérations mathématiques aussi complexes que celles qui soutiennent les algorithmes de Feautrier, Darte et Vivien, ou Lim et Lam, dans ce mémoire, la technique de Banerjee (1988), décrite en détail dans la prochaine section, va être utilisée pour l'analyse des dépendances. L'algorithme d'extraction du parallélisme jusqu'à la réécriture du code va être dérivé d'une symbiose des algorithmes de Wolf et Lam et de Banerjee (1993).

## 1.2. Analyse des dépendances

La connaissance des instructions et des domaines d'itération n'est pas suffisante pour définir complètement la sémantique d'un nid de boucles. Tel qu'énoncé dans la section précédente, l'analyse des dépendances est une part importante de la recherche de parallélisme d'une application (1993). Cette section décrit les mécanismes et processus qui régissent cette analyse.

L'exécution séquentielle attribue un ordre total aux opérations, appelé ordre lexicographique. Par contre, une description séquentielle est souvent inutilement rigide, et peut être partiellement désordonnancée. En fait, deux opérations peuvent être effectuées dans n'importe quel ordre, y compris simultanément, si elles sont *indépendantes*. Une dépendance apparaît lorsque les deux opérations cherchent à utiliser une même ressource ou à modifier un même résultat.

Pour ce mémoire, nous nous restreindrons à un type bien particulier de conflits d'accès : les dépendances liées à l'utilisation de la mémoire et des unités de calcul. Ces dépendances sont représentatives de l'ensemble des facteurs spécifiques qui limitent l'exploitation du parallélisme dans l'architecture multiprocesseurs SIMD qu'est PULSE. En effet, le présent travail se concentre sur les noyaux[6] des programmes, ainsi que sur

[6] Ces noyaux sont essentiellement des nids de boucles.

les entrées/sorties aux périphériques, qui sont supposées avoir lieu avant et/ou après ces noyaux.

Ces dépendances décrivent parfaitement les liens qui existent entre les différents calculs. Une analyse exacte des dépendances est souvent impossible car trop d'information sur les zones mémoire affectées est analysée d'une manière statique. Une attitude pessimiste est généralement adoptée, consistant à dire qu'il existe une dépendance dans les cas où l'indépendance ne peut pas être certifiée. Ceci permet d'éviter d'autoriser des transformations qui pourraient mener à une perte de la sémantique du programme initial. Des algorithmes ont été proposés, comme le test de Banerjee, le *GCD test*, le *Lambda test*, le *Power test* et le *Omega test* pour trouver une solution satisfaisante à ce problème NP-complet (P. Feautrier, 1991).

Quels sont donc les conflits d'accès à la mémoire ? Si deux opérations successives doivent écrire dans une même cellule de la mémoire, il est clair qu'inverser l'ordre d'exécution de ces opérations aboutirait à un état différent de la mémoire. De même, si une des opérations lit et l'autre écrit dans une même case, la valeur lue par la première ne peut être préservée que si l'ordre initial est respecté. Trois conditions suffisantes ont été identifiées pour que deux opérations $o_1$ et $o_2$, prescrites dans un ordre particulier dans le programme séquentiel, puissent être exécutées dans un ordre quelconque (U. Banerjee, 1988). Soient $L(o_1), M(o_1)$ et $L(o_2), M(o_2)$, les ensembles des cellules mémoires lues et mises à jour par $o_1$ et $o_2$ respectivement. Les opérations $o_1$ et $o_2$ sont indépendantes si et seulement si

- C1: $M(o_1) \cap L(o_2) = 0$, $o_1 \delta^t o_2$
- C2: $L(o_1) \cap M(o_2) = 0$, $o_1 \overline{\delta} o_2$
- C3: $M(o_1) \cap M(o_2) = 0$, $o_1 \delta^o o_2$

Le non-respect de chacune de ces conditions provoque une *dépendance de données* :

- **Non-respect de C1 :** Le résultat d'une opération est sauvegardé en mémoire et utilisé ensuite par une autre opération. Cette dépendance est appelée

indifféremment *vraie dépendance* ou *dépendance producteur-consommateur* (PC). Nous noterons cette dépendance $o_1 \delta^t o_2$. Par exemple, soit les opérations $\langle S,1,1 \rangle$ et $\langle S,2,1 \rangle$ de l'exemple 1.1 où $\langle S,a,b \rangle$ est l'opération réalisée lorsque les indices valent $i=a$ et $j=b$. Ces opérations sont dépendantes suivant une dépendance PC, puisque la première écrit dans la cellule mémoire `a(2)` qui est lue par la seconde. L'ensemble des dépendances PC du programme `exemple`, pour $n$=4, sont représentées par des arcs dans le graphe de dépendance de la figure 1.4a.

- **Non-respect de C2 :** Le non-respect de C2 est une *anti-dépendance* ou une *dépendance consommateur-producteur* (CP), notée $o_1 \bar{\delta} o_2$. Une dépendance CP lie $\langle S,1,3 \rangle$ à $\langle S,2,1 \rangle$ puisque $\langle S,1,3 \rangle$, compte tenu de l'ordre lexicographique du programme, doit pouvoir lire dans `a(3)` avant que le contenu de cette case mémoire ne soit remplacé par $\langle S,2,1 \rangle$.
- **Non-respect de C3 :** De la même façon, le non-respect de C3 provoque une *dépendance de sortie* ou *dépendance producteur-producteur* (PP), notée $o_1 \delta^o o_2$. Ainsi les opérations $\langle S,1,2 \rangle$ et $\langle S,2,1 \rangle$ de l'exemple sont en dépendance de sortie puisque toutes deux écrivent dans `a(3)`.

**Exemple 1.1.**

```
for i=1, n {
   for j=1, n{
S:    a(i+j) = a(i+j-1);
   }
}
```

Lorsqu'une dépendance de donnée PP, PC ou CP existe entre deux opérations $o_1$ et $o_2$, nous écrirons $o_1 \delta o_2$. L'ensemble de toutes les dépendances de données de l'exemple 1.1, pour $n$=4, est représenté sur la figure 1.4b. Une dépendance d'une opération $o_1$ à une

opération $o_2$ est dite *respectée* si $o_1$ est exécutée avant $o_2$. Une dépendance est donc une contrainte de précédence. La relation d'ordre $\delta$ est un ordre inclus dans $\prec$, l'ensemble des relation d'ordre lexicographique ; si $\delta$ est partielle, le programme *contient du parallélisme*. Le but de la parallélisation est donc d'exhiber un ordre partiel $\prec_{//}$ tel que

$$\delta \subseteq \prec_{//} \subseteq \prec \tag{1.3}$$

et que $\prec_{//}$ soit évidemment le plus *proche* possible de $\delta$.

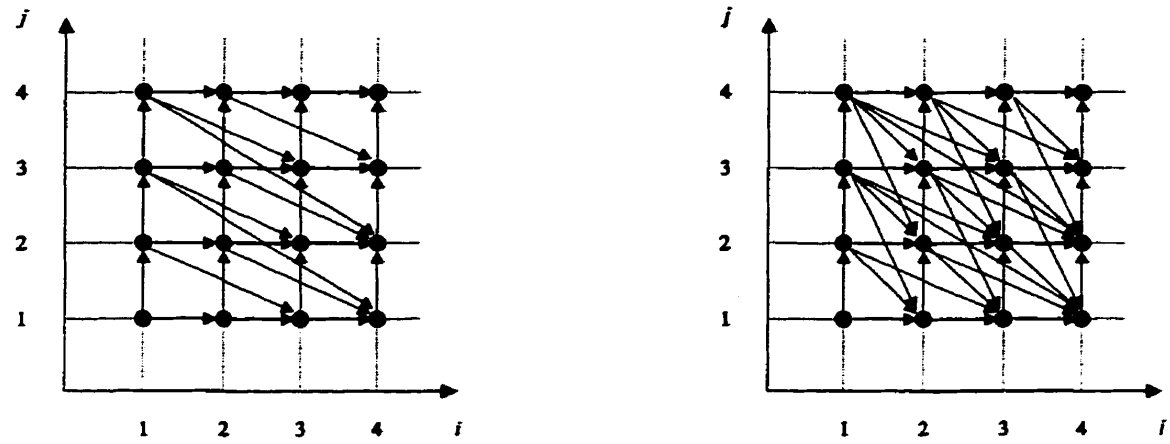


*Figure 1.4: Domaine d'itération de l'exemple 1.1 :*
*(a) dépendances PC, (b) ensemble des dépendances PC, CP et PP*

Une remarque peut ainsi être faite : de nombreuses dépendances sont redondantes car elle sont obtenues par transitivité.

- Deux dépendances PP sur une même cellule mémoire entraînent l'existence d'une troisième dépendance PP ; ces dépendances PP obtenues par transitivité ne sont pas montrées sur la figure 1.4a.

- Deux dépendances PP $o_1\delta^o o_2$ et PC $o_2\delta^t o_3$ donnent par transitivité une dépendance PC $o_1\delta^t o_3$. Cette nouvelle dépendance est artificielle, car si la cellule mémoire lue par $o_3$ a été modifiée par $o_1$ et $o_2$, l'effet de bord de $o_1$ a été oblitéré (ang. *killed*) par l'effet de $o_2$. En fait, la valeur réellement lue par $o_3$ est celle produite par $o_2$. Ce type particulier de dépendance PC, qui concerne une opération de lecture dans une cellule et la *dernière opération écrivant dans cette cellule*, est appelé *flot de donnée*. De façon imagée, la donnée *coule* de sa *source* (l'opération productrice $o_2$) à son *puits* (l'opération $o_3$). Cette dépendance sera notée $o_1\Gamma o_3$.
- Pour trois opérations $o_1$, $o_2$ et $o_3$ exécutées dans cet ordre, deux dépendances PC $o_1\delta^t o_2$ et PP $o_1\delta^o o_3$ impliquent l'existence d'une dépendance CP $o_1\bar{\delta} o_3$. En général, une dépendance CP est due à la succession d'une dépendance PC et d'une dépendance PP[7].

Nous arrivons donc à la conclusion que supprimer tous les arcs PP entraîne la suppression de la plupart des arcs PC et CP dans le graphe de dépendances. En fait, les dépendances PP, CP, et PC autres que celles dues au flot de données sont des dépendances dues à une réutilisation de la mémoire (ang. *memory-based dependencies*). Par opposition, les dépendances dues au flot de données transportent les valeurs au cours de l'exécution et sont donc inhérentes à l'algorithme, et non plus au programme et/ou au langage de programmation.

De plus, la coupure de ces arcs diminue le nombre de contraintes, donc nous pouvons accroître le parallélisme. Ce qui ce trouve être le but de la recherche du parallélisme maximal. Le graphe des flots de données dans l'exemple 1.1 apparaît en figure 1.5. En

---

[7] Ce n'est pas vrai si la lecture n'a pas de source dans le programme, par exemple si cette lecture est la première opération du programme.

terme d'ordre, $\Gamma \subseteq \delta \subseteq \prec$, ce qui nous incite à essayer de prendre en compte $\Gamma$ au lieu de $\delta$ , et de construire un ordre parallèle $\prec_{//}$ tel que

$$\Gamma \subseteq \prec_{//} \subseteq \prec \tag{1.4}$$

et qui soit évidemment le plus *proche* possible de $\Gamma$. Comme nous venons de le discuter, ceci nécessite la coupure des arcs PP.

À partir de cette constatation, un mécanisme de parallélisation pourrait être *la mise en assignation unique*. Pour couper tous les arcs PP, il faut que toutes les opérations écrivent dans une cellule mémoire qui lui soit propre. Si toute cellule mémoire n'est écrite que par au plus une opération, alors le programme est en assignation unique. Deux objections viennent à l'esprit : un programme en assignation unique est extrêmement gourmand en espace mémoire, puisque le nombre de cellules est égal au nombre d'assignations. Deuxièmement, les programmes réels n'ont pas cette propriété. Mais, étant donné le parallélisme potentiel des programmes en assignation unique, un mécanisme de conversion automatique en assignation unique a été proposé par Paul Feautrier (1988). Ce mécanisme d'optimisation peut-être intéressant dans des cas bien spécifiques. Dans le cadre du projet PULSE, ce type de programme composé d'aucune dépendance PP est rare et trop lourd pour les petites mémoires de PULSE, donc pas vraiment intéressant.

Pour réaliser nos transformations de boucles telles que décrites dans la section suivante (section 1.3), nous n'avons pas besoin d'aller si loin dans la recherche car nous nous basons sur le *vecteur de distance de dépendance* (ang. *dependence distance vector*) et ce pour toutes les paires d'itérations.

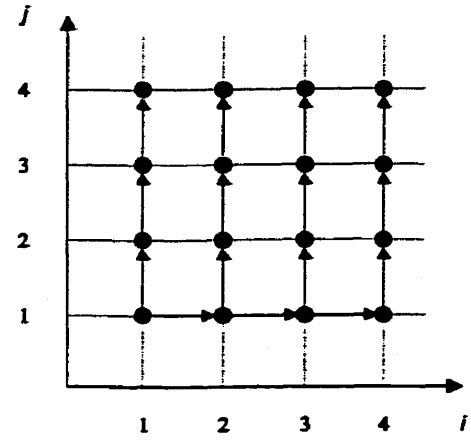


*Figure 1.5: Domaine d'itération de l'exemple 1.1 possédant un parallélisme accrut*

**Définition 1.10 - Vecteur de distance de dépendance :** Pour une paire d'itérations $i=(i_1,\cdots,i_n)$ et $j=(j_1,\cdots,j_n)$ tel que $i\delta j$, le vecteur de distance de dépendance est $j-i=(j_1-i_1,\cdots,j_n-i_n)$.

Dorénavant, à moins d'exceptions clairement identifiées, les dépendances seront exprimées par un *vecteur de distance de dépendance.*

**Définition 1.11 - Direction de dépendance :** Pour une paire d'itération $i=(i_1,\cdots,i_n)$ et $j=(j_1,\cdots,j_n)$, la direction de dépendance est :

$sign(j-i)=(sign(j_1-i_1),\cdots,sign(j_n-i_n))$, où $sign(a)\in\{1,0,-1\}$.

Un vecteur de dépendance $(d_1,\cdots,d_n)$ est dit *positif* si le premier élément non nul $d_k$ est positif. Dans ce cas, nous parlons de dépendance portée par la $k^{\text{ième}}$ boucle. La sémantique séquentielle des boucles suggère que le flux des dépendances est en tout temps positif[8]. La matrice de dépendance $D$ est de taille $n$ par $m$, où $n$ est la profondeur

[8] Cette propriété détermine la validité des transformations effectuée sur un nid de boucles.

du nid de boucle et $m$ est le nombre de dépendances non nulles que possède $H(i_1,\cdots,i_n)$.

## 1.3. Transformations de boucles

Une fois l'analyse des dépendances effectuée, nous nous retrouvons avec un ordonnancement des opérations en fonction du temps contenu dans un graphe de communication guidé par le temps d'exécution. À partir de là, le temps total du programme est connu.

L'étape suivante a pour objectif de réordonnancer les opérations pour minimiser le temps total d'exécution en exploitant le parallélisme implicite du programme source (A. Darte, 1993). Cette étape est réalisée en essayant de supprimer le maximum de dépendances infligées par l'aspect séquentiel du programme source. Ceci permet par la suite d'attribuer les opérations et les données aux éléments d'une grille virtuelle de processeurs au cours de l'étape d'allocation. Cette grille est, dans cette étape, de taille et de dimension non bornée. Le but de cette allocation est de réaliser une répartition qui propose une minimisation du nombre de communications inter-processeurs, pour ainsi réduire la latence engendrée. Cette répartition doit prendre comme référence de départ le graphe de communications qui supporte l'ordonnancement total des opérations. D'une manière générale, la distribution des calculs est dirigée par celle des données suivant la règle du *owner computes* (U. Banerjee, 1993) (certains aspects de cette règle sont détaillés à la section 1.3.3). Cette règle stipule que le processus sur lequel est alloué la donnée résultante d'une opération exécute cette opération.

Une fois que sont connues les tâches que doivent réaliser chaque processeur virtuel, il suffit de réécrire le programme contenant : des boucles explicitement parallèles et une répartition des données qui est, elle aussi, explicite. Ce travail est le rôle de la génération de code, la dernière étape d'un compilateur-parallèliseur (voir figure 1.1). Cela peut paraître simpliste, mais un vaste travail reste cependant à effectuer pour produire une méthode de réécriture de boucles prenant en compte tous les décalages

d'indices entre les différentes instructions, tant du point de vue spatial que du point de vue temporel : une telle méthode devra permettre de réécrire automatiquement les nids de boucles en gérant les problèmes liées aux effet de bords, aux réenroulement des boucles, et à l'introduction de pas d'itération (ang. *step*) différents de 1. Conceptuellement, il n'y a pas de difficulté majeure à la complète automatisation de la réécriture et la génération d'un code correct. Par contre, la génération d'un code efficace dans lequel les expressions des bornes et les structures de contrôle sont simplifiées nécessite encore un travail important et difficile (A. Darte, 1993). Mais, ceci n'est qu'une partie du problème de la génération de code. Il faut également être capable de générer tous les transferts de données impliqués par une telle répartition : communications externes des données, gestion de la mémoire et réutilisation de données. Cette section se donne pour objectif de démystifier ces étapes d'ordonnancement d'allocation et de génération de code en proposant un algorithme basé sur la transformation de boucles. Le but de cet algorithme est de reconstruire un nouveau nid de boucles pouvant exploiter le parallélisme proposé par l'architecture matérielle ciblée, tout en générant les mêmes résultats que le nid original. Les objectifs de cette transformation sont d'accroître le parallélisme explicite du nid à un niveau donné, de minimiser la taille de la boucle séquentielle dominante, d'égaliser la charge de travail de chacune des unités de calculs, de réduire les distances entre les données et les unités de calculs, et enfin de regrouper les données réutilisées. Il existe plusieurs types de transformation applicables aux boucles :

- La *permutation* effectue une permutation de deux ou plusieurs boucles d'un même nid. Cette transformation peut-être utilisée dans le cas où nous souhaitons amener une boucle parallélisable à un niveau donné.
- L'*inversion* change le sens de l'incrément d'une boucle donnée. Tout en gardant un incrément positif et égal à 1 en accord avec la méthode.
- Le *front d'onde* (ang. *wavefront*) identifie un nombre d'itérations possédant des dépendances et restructure le nid de boucle pour les effectuer séquentiellement.

- Le *regroupement* combine des blocs d'itérations dans les boucles internes. Ce regroupement réduit la taille des boucles par un facteur donné. Plus la taille des boucles les plus internes est petite, plus les données ont la chance de tenir dans la mémoire locale du processeur ce qui réduit ainsi les communications externes.

Une transformation unimodulaire a l'effet de réaliser un changement de repère dans le domaine d'itération du nid source de boucles. Réaliser des transformations de boucle implique l'utilisation de règles strictes lors du changement de repère et des limites des bornes de boucle. Ces règles ont pour unique but de s'assurer que le nid de transformations résultant réalise le même traitement que le nid source. Pour ce faire (dans le cas de notre analyse), seules les transformations unimodulaires sont valides (M.L. Dowling, 1990). Pour exploiter au maximum la structure des architectures matérielles cibles, il est généralement nécessaire d'appliquer plusieurs transformations successives.

Dans la sous-section suivante, l'algorithme qui réalise des transformations unimodulaires proposé par U. Banerjee (1993) est développé. Ce ne sera pas avant la sous-section 1.3.2. que nous chercherons à obtenir une transformation efficace.

## 1.3.1. Technique générale

Une transformation unimodulaire est définie par une matrice unimodulaire[9] de nombres entiers $U$ de taille $n$ par $n$. $U$ transforme un vecteur d'itération $I$ en un nouveau vecteur d'itération $K$, tel que

$$UI = K \tag{1.5}$$

et ainsi transforme les dépendances du nid source $D$ en de nouvelles $D'$, tel que

$$D' = UD \tag{1.6}$$

---

[9] Qui possède un déterminant égale à ±1.

Cette transformation est légale si est seulement si $D'$ est lexicographiquement positif. La positivité des dépendances est une condition de la validité de la transformation. Il existe des cas où nous pouvons recourir à des valeurs négatives (M.L. Dowling, 1990), mais dans ce mémoire (pour ne pas surcharger la méthode de transformation), nous nous astreindrons à satisfaire ce critère de positivité.

Pour réaliser cette transformation, il faut substituer les indices de boucles (le vecteur d'itération) du nid source par les nouveaux, et les bornes des boucles par les nouvelles.

Dans un premier cas, la substitution des indices de boucles et des références dans les instructions par les nouveaux est supportée par l'équation suivante

$$U^{-1}K = I \tag{1.7}$$

En revanche, déterminer les nouvelles bornes des boucles est moins évident. Partant de

$$SI \geq c \tag{1.8}$$

nous obtenons

$$SU^{-1}UI \geq c \tag{1.9}$$

d'où

$$SU^{-1}K \geq c \tag{1.10}$$

Cette inégalité décrit un polyèdre convexe, qui implique que les nouvelles bornes du nid de boucles soient des fonctions affines.

En partant de

$$S' = SU^{-1} \tag{1.11}$$

les nouvelles bornes du nid peuvent être directement obtenues à partir de la rangée de $S'$. En général, la technique d'élimination de variable proposée par Fourier-Motzkin en 1986 se base sur $S'$ pour obtenir ces nouvelles bornes. Supposons que $\alpha$ et $\beta$ sont des

expressions linéaires dans $k_1,\cdots,k_{n-1}$, et $a$ et $b$ sont des constantes. Les inégalités $\beta \le bk_n$ et $ak_n \le \alpha$ proviennent de la borne supérieure de $k_n$, $\max\left\lfloor \alpha/a \right\rfloor$ et de la borne inférieure $\min\left\lfloor \beta/b \right\rfloor$. En éliminant $k_n$, cela nous donne l'inégalité $a\beta \le b\alpha$, où $k_n$ a disparu. En réarrangant ces nouvelles inégalités, nous obtenons $k_{n-1}$ en fonction de $(k_1,\cdots,k_{n-2})$. Si nous continuons de la même manière pour toutes les autres variables, nous arrivons à obtenir $k_1$ constant. L'exemple suivant illustre ce concept.

Considérons le nid de boucles

```
for i1=n1, N1 {
   for i2=n2, N2 {
      H(i1, i2)
   }
}
```

où $(i_1,i_2)^T$ est le vecteur d'itération. Supposons la matrice unimodulaire suivante

$$U=\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \tag{1.12}$$

Le nid résultant de la transformation a la forme suivante

```
for k1=m1, M1 {
   for k2=m2(k1), M2(k1) {
      H(f(k1, k2), g(k1, k2))
   }
}
```

où $(k_1,k_2)^T$ est le nouveau vecteur d'itération, tel que

$$(k_1,k_2)^T = U(i_1,i_2)^T = (u_{11}i_1+u_{12}i_2, u_{21}i_1+u_{22}i_2)^T \tag{1.13}$$

et $m_1$ et $M_1$ sont constant, $m_2()$ et $M_2()$ sont des fonctions de $k_1$, et $f()$ et $g()$ sont des fonctions de $k_1$ et $k_2$.

Il est à se rappeler que nous devons respecter l'unimodularitée de la matrice de transformation et donc nous avons : $\Delta = \det(U) = \pm 1$. Si nous notons $\Delta u_{ij}$ équivalant à $\det(U) * u_{ij}$, alors nous pouvons écrire

$$U^{-1} = \begin{bmatrix} \Delta u_{22} & -\Delta u_{12} \\ -\Delta u_{21} & \Delta u_{11} \end{bmatrix} \tag{1.14}$$

Ce qui donne

$$(i_1, i_2)^T = U^{-1}(k_1, k_2)^T = (\Delta u_{22} k_1 - \Delta u_{12} k_2, -\Delta u_{21} k_1 + \Delta u_{11} k_2)^T \tag{1.15}$$

Les bornes de $k_1$ et $k_2$ peuvent être obtenues comme suit. Les bornes de $k_1$ sont constantes et dérivées de

$$k_1 = u_{11} i_1 + u_{12} i_2 \tag{1.16}$$

Ce qui donne

$$m_1 = \min(u_{11} i_1 + u_{12} i_2, n_1 \le i_1 \le N_1, n_2 \le i_2 \le N_2)$$
$$M_1 = \max(u_{11} i_1 + u_{12} i_2, n_1 \le i_1 \le N_1, n_2 \le i_2 \le N_2)$$

Pour alléger les notations nous introduisons la notation suivante : $a^+ = \max(a, 0)$ et $a^- = \max(-a, 0)$. Ce qui permet d'écrire $m_1$ et $M_1$ tel que

$$m_1 = u_{11}^+ n_1 - u_{11}^- N_1 + u_{12}^+ n_2 - u_{12}^- N_2 \tag{1.17}$$

$$M_1 = u_{11}^+ N_1 - u_{11}^- n_1 + u_{12}^+ N_2 - u_{12}^- n_2 \tag{1.18}$$

Les bornes de $k_2$ ne sont pas constantes et plus complexes à obtenir, dû au fait que $n_1 \le i_1 \le N_1$ et $n_2 \le i_2 \le N_2$ ce qui se transforment en

$$n_1 \le \Delta u_{22} k_1 - \Delta u_{12} k_2 \le N_1$$
$$n_2 \le -\Delta u_{21} k_1 + \Delta u_{11} k_2 \le N_2$$

en substituant $i_1$ et $i_2$, nous obtenons

$$\frac{n_1 - \Delta u_{22} k_1}{-\Delta u_{12}} \leq k_2 \leq \frac{N_1 - \Delta u_{22} k_1}{-\Delta u_{12}}$$

$$\frac{n_2 + \Delta u_{21} k_1}{\Delta u_{11}} \leq k_2 \leq \frac{N_2 + \Delta u_{21} k_1}{\Delta u_{11}}$$

qui nous donne les bornes hautes $hb_1$ et $hb_2$, et les bornes basse $lb_1$ et $lb_2$ tel que

$$m_2(k_1) = \lceil \max(lb_1, lb_2) \rceil \tag{1.19}$$

$$M_2(k_1) = \lfloor \min(hb_1, hb_2) \rfloor \tag{1.20}$$

Pour ne pas se perdre dans la théorie un exemple plus concret est présenté.

**Exemple 1.2.**

Soit le nid de boucles suivant

```
for i1=0, 10 {
   for i2=0, 10{
      A(i1, i2)= A(i1-1, i2)+ A(i1, i2-1)+ A(i1-2, i2+1)
   }
}
```

En réalisant une analyse de dépendances telle que décrite dans la section 1.2, les dépendances pour cet exemple se réduisent à trois dépendances du type PC. Elles sont décrites par le vecteur de dépendances $D$ tel que

$$D = \{(1,0), (0,1), (2,-1)\}$$

Soit $U$, une matrice de transformation possible pour cet exemple

$$U = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \text{ et } U^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

Nous obtenons

$$(i_1, i_2)^T = (k_1 - k_2, k_2)^T$$

et

$$D' = \{(1,0), (1,1), (1,-1)\}$$

Remarquons que toutes les dépendances sont lexicographiquement positives, donc la transformation est légale. Les bornes de $k_1$ sont

$$\begin{aligned} m_1 &= u_{11}^+ n_1 - u_{11}^- N_1 + u_{12}^+ n_2 - u_{12}^- N_2 \\ &= 1*0 - 0*10 + 1*0 - 0*10 \\ &= 0 \end{aligned}$$

$$\begin{aligned} M_1 &= u_{11}^+ N_1 - u_{11}^- n_1 + u_{12}^+ N_2 - u_{12}^- n_2 \\ &= 1*10 - 0*0 + 1*10 - 0*0 \\ &= 20 \end{aligned}$$

Les bornes de $k_2$ sont

$$hb_1 = \frac{n_1 - \Delta u_{22} k_1}{-\Delta u_{12}} = \frac{0 - k_1}{-1} = k_1$$

$$hb_2 = \frac{N_2 + \Delta u_{21} k_1}{\Delta u_{11}} = \frac{10}{1} = 10$$

$$lb_1 = \frac{N_1 - \Delta u_{22} k_1}{-\Delta u_{12}} = \frac{10 - k_1}{-1} = k_1 - 10$$

$$lb_2 = \frac{n_2 + \Delta u_{21} k_1}{\Delta u_{11}} = \frac{0 + 0}{1} = 0$$

Ce qui permet de réécrire le nid de boucles source sous la forme suivante

```
for k1=0, 20 {
   for k2=max(0, k1-10), min(10, k1) {
      A(k1-k2, k2)= A(k1-k2-1, k2)+ A(k1-k2, k2-1)
           + A(k1-k2-2, k2+1)
   }
}
```

La figure 1.6 présente le domaine d'itération avec les deux formes : avant et après transformation. En observant cette figure, nous pouvons constater que le long des axes $k_1$ et $k_2$, il y a des coupures d'arcs de dépendance. Ceci suggère que les opérations ne peuvent pas être exécutées en parallèle sans communications.

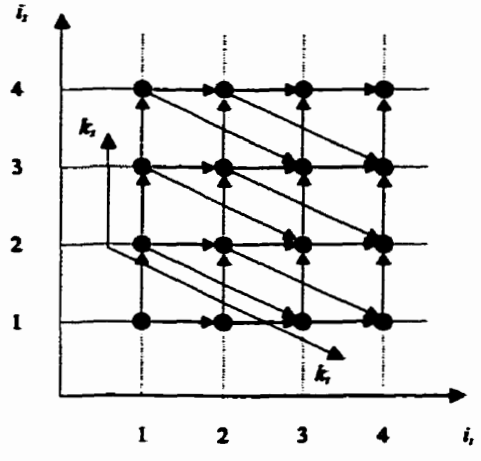


*Figure 1.6: Domaine d'itération de l'exemple 1.2*

Une transformation unimodulaire peut-être une transformation primaire telle qu'une permutation par exemple, mais elle peut aussi être une combinaison de transformations. La figure 1.7 présente un éventail non exhaustif de matrices de transformation primaire pour des nids d'une profondeur de deux. La figure 1.7a présente une matrice qui réalise une inversion de la boucle externe. En (b), elle réalise l'opposé. En (c), l'inversion s'applique sur les deux boucles. En (d), la matrice permute des boucles entre elles. En

(e) et (f) une torsion d'un facteur $p$ est réalisée. Pour sa part, (g) réalise un front d'onde.

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

(a) (b) (c)

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ p & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

(d) (e) (f) (g)

*Figure 1.7: Dictionaire de transformation primaire*

Une combinaison de matrices est le résultat de multiplications de matrices primaires appliquées dans l'ordre inverse. Mais dans bien des cas, nous ne connaissons pas d'avance le type de transformation que nous souhaitons effectuer sur un nid. La sous-section suivante montre comment créer des transformations sur mesure en fonction de nos objectifs.

### 1.3.2. La transformation sur mesure

Jusqu'à présent nous nous satisfaisons de récupérer des matrices de transformation existantes et de les appliquer aux nids de boucles étudiés. Dans cette sous-section, nous allons rechercher une ou plusieurs matrices de transformation adaptées à nos besoins, à partir des dépendances existantes et de l'architecture matérielle cible. Supposons que nous cherchons à paralléliser la boucle externe d'un nid de boucle. Pour ce faire, il nous faut appliquer une transformation qui supprime (ou diminue) les dépendances qui existent sur ce niveau du nid.

**Exemple 1.3.**

Soit le nid suivant

```
for i1=0, n {
   for i2=0, n {
      A(i1, i2)= A(i1-1, i2-1)
   }
}
```

Possédant la dépendance suivante

$$D=\left\{\begin{pmatrix}1\\1\end{pmatrix}\right\}=\left\{\begin{pmatrix}d_1\\d_2\end{pmatrix}\right\}$$

et nous souhaitons obtenir la dépendance suivante

$$D'=UD \Leftrightarrow (d'_1, d'_2)^T=\begin{bmatrix}u_{11} & u_{12}\\ u_{21} & u_{22}\end{bmatrix}D$$

où $d'_1=0$ et $d'_2\geq 0$. Ce qui donne

$$\begin{cases}u_{11}d_1+u_{12}d_2=0\\ u_{21}d_1+u_{22}d_2\geq 0\end{cases}$$

Avec la règle sur le déterminant, ce système possède trois équations pour quatre inconnues. Après résolution, une des solutions possibles est

$$U=\begin{bmatrix}1 & -1\\ 0 & 1\end{bmatrix} \text{ avec } \Delta=+1 \text{ et } D'=\left\{\begin{pmatrix}0\\1\end{pmatrix}\right\}$$

Cette matrice permet de reconstituer la boucle suivant la méthode décrite dans la sous-section précédente

```
for k1=-n, n {
   for k2=max(0, -k1), min(n, n-k1) {
      A(k1+k2, k2)= A(k1+k2-1, k2-1)
   }
}
```

En observant la figure 1.8, nous remarquons aisément qu'il n'y a plus d'arc coupé par l'axe de la boucle externe $k_1$. Cette boucle peut alors être entièrement distribuée sur des processeurs indépendants sans qu'il y ait de communication entre eux.

Cette technique de recherche de matrice de transformation unimodulaire permet ainsi, sans connaître le flux des données d'un algorithme, de trouver la ou les matrices de transformation unimodulaire permettant d'atteindre des objectifs sur les contraintes posées sur l'algorithme lui-même, sur l'architecture ou sur les deux.

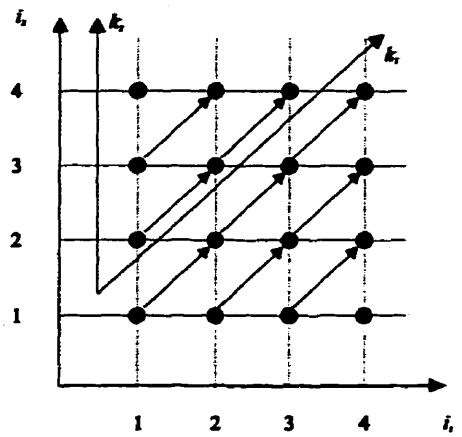


*Figure 1.8: Domaine d'iteration de l'exemple 1.3*

## 1.3.3. Remarques sur la règle du "owner compute"

La plupart des compilateurs-paralléliseurs utilisent la règle de l'*owner compute* et placent les calculs d'une instruction au sein du processeur possédant la donnée à modifier, ceci pour simplifier le travail du compilateur (A. Darte, 1993). Mais cela entraîne une perte

d'optimalité. Cette perte possible d'optimalité est à considérer dans le cadre du modèle que nous nous sommes fixés. En particulier, la possibilité de réutiliser les données déjà transmises pour un calcul antérieur n'est pas prise en compte. Il existe des exemples pour lesquels une allocation suivant la règle du *owner compute* avec une réutilisation intelligente des données fournit une allocation pour laquelle moins de communications sont nécessaires. Ici, la duplication des données n'est pas considérée.

## 1.4. Conclusion

Nous avons montré dans ce chapitre, en nous basant sur la littérature, comment déceler les dépendances de données d'un algorithme et ainsi construire un domaine d'itération. Nous avons montré aussi comment ce dernier permet de déceler un ordonnancement statique du flux de ces données. C'est à partir de l'usage de fonctions linéaires ou même affines que la réécriture du nid de départ sous une forme parallélisable se fait de façon complètement automatique et revient à un simple changement d'indices de description par des matrices unimodulaires.

Il est bien entendu qu'un langage universel, indépendant de la machine cible, ne pourra pas être compilé et offrir des résultats aussi performants que ceux d'un code écrit dans un langage spécifique à la machine cible et exploitant à fond ses capacités. Il y aura toujours des programmeurs pour se passionner pour les optimisations de bas niveau, à la recherche du code le plus efficace et prêts aux optimisations les plus scandaleuses du point de vue conceptuel.

Peut-on espérer une programmation à haut niveau des machines parallèles? La conclusion de ce chapitre est de répondre par l'affirmative à cette question. Un travail énorme reste à fournir, mais nous voyons déjà que le fossé menant des nids de boucles à un langage de haut niveau est en voie d'être comblé. Le nombre croissant de chercheurs et d'articles publiés chaque année prouvent que ce domaine est en pleine effervescence. Si nous parvenions à générer automatiquement un code même deux fois moins bons

qu'un code écrit à la main, nous pourrions estimer le pari gagné, tant il est pénible aux non spécialistes de programmer ne serait-ce qu'un code parallèle correct.

Le chapitre suivant présente une succession d'applications des règles et méthodes développées tout au long de ce dernier. Son but est de démontrer comment les mettre en application et obtenir un gain par rapport aux méthodes manuelles dans le cadre de l'architecture multiprocesseur PULSE.

# CHAPITRE II

# DE LA THÉORIE À LA PRATIQUE

Sans applications pratiques une théorie reste une curiosité universitaire. Après avoir exposé les mécanismes de la parallélisation de nids de boucles tout au long du chapitre précédent, celui-ci s'attache plus particulièrement à donner des exemples et des démonstrations de parallélisation sur des algorithmes concrets et ciblés pour l'architecture matérielle PULSE. Pour concrétiser cette démarche, nous nous attarderons à comparer les résultats de performance de partitionnement et de génération de code avec des résultats réels[10]. Puis nous conclurons sur une synthèse des performances obtenues et des améliorations à apporter sur l'algorithme de parallélisation étudié dans ce mémoire.

Quatre sous-sections composent ce chapitre, les deux premières analysent un algorithme et proposent une parallélisation possible pour les filtres IIR 1D et FIR 2D respectivement. Ces algorithmes ont été choisis pour servir d'étude de cas et ne constituent pas des solutions clefs en main. La section suivante (2.3) vise à appliquer la même démarche à des algorithmes hostiles à la méthode de parallélisation soutenue par ce mémoire. Ces algorithmes peuvent-être considérés comme des contres exemples (filtres de détection de contours et filtre médian). Des discussions, comparaisons, améliorations, et conclusions sont présentées à la section 2.6.

---

[10] S'ils sont existants et disponibles.

Notons que l'architecture, les mécanismes de contrôle des mémoires, et le jeu d'instruction de PULSE (P. Mariott, I.C. Kraljic et Y. Savaria, 1998) guident nos choix lors de l'application de la méthode de parallélisation des algorithmes.

## 2.1. Le filtre IIR 1D

Le premier algorithme proposé est bien connu dans le domaine des filtres linaires numériques : le filtre IIR 1D (ang. *Infinite Impulse Response*). Il est présenté en premier car son implantation logicielle possède les particularités d'être organisé selon un nid de boucles de profondeur de deux et avec des variables à une dimension. Cette particularité en fait un algorithme intéressant pour commencer avec une parallélisation facile.
Sa fonction de transfert en Z s'écrit sous la forme

$$H(z)=\frac{\sum_{j=0}^{N} a_j z^{-j}}{\sum_{i=1}^{N} b_i z^{-i}} \tag{2.1}$$

qui peut-être réécrit dans le domaine temporel comme suit

$$y_m = \sum_{j=0}^{N} a_j x_{m-j} + \sum_{i=1}^{N} b_i y_{m-i} \tag{2.2}$$

où $x_m$ et $y_m$ sont respectivement les signaux entrant et sortant, $a_n$ et $b_n$ les coefficients, et $N$ représente l'ordre du filtre.

Il est intéressant de constater que la majorité des opérations effectuées par ce filtre sont des multiplications-accumulations et des déplacements de données. De plus, il est à noter qu'une erreur a volontairement été insérée dans ce modèle logiciel. Cette modification est imposée par les contraintes de l'algorithme de parallélisation qui n'est

valide uniquement sur des nids de boucle parfait. Ce qui nous impose de ne pas calculer le premier terme de l'équation ci-dessous équivalente à la précédente

$$y_n = a_0 x_n + \sum_{j=1}^{N} a_j x_{n-j} + b_j y_{n-j} \tag{2.3}$$

Voici l'implantation logicielle de la portion de ce filtre décrite par la sommation dans l'équation 2.3

```
L1: for i1=0, M-1{
L2:   for i2=1, N {
H:      y(i1)=y(i1)+a(i2)*x(i1-i2)+b(i2)*y(i1-i2);
      }
    }
```

où M est la longueur de la trame des données entrantes. Un soin particulier a été apporté pour que ce nid soit écrit sous une forme *parfaite*. Il est à noter que pour cette implantation logicielle et les suivantes présentées dans ce chapitre, il est supposé que toutes les variables sont initialisées à la valeur zéro avant leur utilisation.

Une fois l'implantation logicielle fixée, une analyse des dépendances est réalisée pour obtenir le domaine d'itération sur lequel l'ordonnancement et le placement seront réalisés. En ce référant à la définition 1.10 et ce pour l'opération $H(i_1, i_2)$ de ce filtre, nous obtenons le vecteur de distance de dépendance suivant

$$D = \left\{ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} i_2 - i_1 \\ i_2 - i_1 \end{pmatrix} \begin{pmatrix} i_2 \\ i_2 \end{pmatrix} \begin{pmatrix} i_2 - i_1 \\ i_2 - i_1 \end{pmatrix} \begin{pmatrix} i_2 \\ i_2 \end{pmatrix} \right\} \tag{2.4}$$

Pour des fins de simplification, nous supposons que les vecteurs de coefficients $a_n$ et $b_n$ peuvent être contenues dans les mémoires de chacun des processeurs ou passé en paramètre par le mot d'instruction. Cette hypothèse permet de faire abstraction des dépendances relatives à ces vecteurs, ainsi le vecteur de distance de dépendance devient

$$D=\left\{\begin{pmatrix}0\\1\end{pmatrix},\begin{pmatrix}i_2\\i_2\end{pmatrix},\begin{pmatrix}i_2\\i_2\end{pmatrix}\right\} \tag{2.5}$$

Il faut remarquer que dans cette analyse des dépendances, les données et les résultats sont traités au même titre. Cette remarque provient du fait que l'allocation se réalise suivant la règle du *owner compute* tel que discutée dans le chapitre précédent. Cela implique qu'il faut regrouper le résultat temporaire d'une opération le plus proche possible de sa prochaine destination. En appliquant cette règle, le raisonnement précédent peut-être appliqué aux données et aux coefficients utilisés dans les opérations. Ces remarques sont valides pour tous les autres filtres et algorithmes traités dans ce chapitre.

Il est à noter que les deux derniers vecteurs de dépendances (équation 2.5) ont des valeurs variables dépendant de l'indice de boucle $i_2$ ce qui équivaut à dire que nous nous retrouvons avec $2N^2+1$ vecteurs de dépendances.

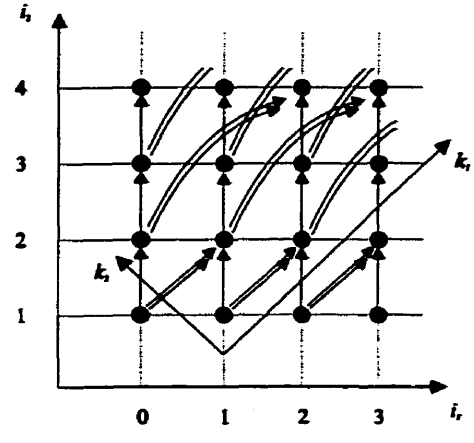


*Figure 2.1: Domaine d'itération du filtre IIR 1D*

En observant le domaine d'itération de *D* illustré par la figure 2.1, nous remarquons aisément qu'il n'est pas possible de réaliser un partitionnement suivant les axes $i_1$ ou $i_2$ sans couper des arcs de dépendance. Il faut donc réordonnancer ce domaine pour

obtenir un partitionnement sans (ou avec un minimum) coupure d'arc. Pour cela, nous nous définissions un vecteur de distance de dépendance objectif

$$D'=\left\{\begin{pmatrix} d'_{11} \\ d'_{12} \end{pmatrix}\begin{pmatrix} 0 \\ d'_{22} \end{pmatrix}\begin{pmatrix} 0 \\ d'_{32} \end{pmatrix}\right\} \tag{2.6}$$

où $\forall([d'_{11}, d'_{12}, d'_{22}, d'_{32}] \in Z) \wedge ([d'_{11}, d'_{12}, d'_{22}, d'_{32}] \geq 0))$ (2.7)

Cet objectif a pour particularité de supprimer les dépendances qui se trouvent au niveau de la boucle externe, et de répartir cette boucle sur différents processeurs pour une exécution parallèle sans communication. Il est aussi important de rechercher des valeurs minimales pour $d'_{11}$, $d'_{12}$, $d'_{22}$, et $d'_{32}$, afin d'avoir des communications inter-processeurs les plus courtes possible. Ceci permet de perdre le moins de temps possible dans les communications.

En appliquant les concepts vus lors du dernier chapitre pour l'équation 1.6, nous avons

$$D'=UD \Leftrightarrow D'=\begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} D$$

où $U$ est la matrice unimodulaire de transformation recherchée. Le système précédent peut se réécrire tel que :

$$\begin{cases} d'_{11} = u_{12} \\ d'_{12} = u_{22} \\ 0 = u_{11} i_2 + u_{12} i_2 \\ d'_{22} = u_{21} i_2 + u_{22} i_2 \\ 0 = u_{11} i_2 + u_{12} i_2 \\ d'_{32} = u_{21} i_1 + u_{22} i_2 \end{cases}$$

Dû au fait que la matrice $U$ est unimodulaire, elle se doit d'avoir un déterminant égal à $\pm 1$. Ce qui donne

$$\begin{cases} d'_{11} = u_{12} \\ d'_{12} = u_{22} \\ 0 = u_{11} i_2 + u_{12} i_2 \\ d'_{22} = u_{21} i_2 + u_{22} i_2 \\ d'_{32} = u_{21} i_1 + u_{22} i_2 \\ u_{11} u_{22} - u_{21} u_{12} = \pm 1 \end{cases} \Leftrightarrow \begin{cases} d'_{11} = u_{12} \\ d'_{12} = u_{22} \\ u_{11} = -u_{12} \\ u_{21} = -u_{22} + d'_{22} / i_2 \\ d'_{32} = d'_{22} \\ u_{11} u_{22} - u_{21} u_{12} = \pm 1 \end{cases}$$

Le système comporte huit inconnues pour seulement six équations. Ce qui permet d'avoir plusieurs solutions. Une solution possible est

$$U = \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix} \text{ avec } \Delta = -1 \tag{2.8}$$

ce qui donne

$$U^{-1} = \begin{bmatrix} -1 & 1 \\ 0 & 1 \end{bmatrix} \text{ et } D' = \left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ i_2 \end{pmatrix}, \begin{pmatrix} 0 \\ i_2 \end{pmatrix} \right\} \tag{2.9}$$

Toutes les nouvelles dépendances sont légales puisqu'elles sont lexicographiquement positives, car $i_2$ varie de 1 à N, ce qui lui assure d'être toujours positif. L'étape suivante se trouve être l'extraction des nouveaux indices des boucles : $k_1$ et $k_2$.

$$(i_1, i_2)^T = U^{-1}(k_1, k_2)^T = (k_2 - k_1, k_2)^T \tag{2.10}$$

$D'$ rencontre les exigences déterminées par l'objectif fixé au départ (équation 2.6). La figure 2.1 illustre les nouveaux axes $k_1$ et $k_2$. En les observant, nous constatons que ces axes créent un nouveau domaine d'itération qui permet de réaliser un partitionnement suivant l'axe des $k_1$ avec une diminution notable des communications entre les partitions. Nous pouvons en déduire que les transformations effectuées sur ce

nid de boucles sont satisfaisantes. Il nous suffit de générer le code de ce nouveau nid pour compléter sa parallélisation.

Les bornes de $k_1$ et $k_2$ sont extraites à partir de

$$(0,1) \leq (k_2 - k_1, k_2) \leq (M-1, N) \tag{2.11}$$

Pour extraire les nouveaux indices, nous utilisons la technique d'élimination de variables proposée par Fourier-Motzkin qui, pour $k_2$, se traduit par

$$\begin{cases} 0 \leq k_2 - k_1 \\ 1 \leq k_2 \\ k_2 \leq N \\ k_2 - k_1 \leq M-1 \end{cases}$$

ce qui donne

$$\lceil \max(1, k_1) \rceil \leq k_2 \leq \lfloor \min(N, M-1+k_1) \rfloor \tag{2.12}$$

et pour $k_1$, nous avons

$$\begin{cases} 1 \leq M-1+k_1 \\ k_1 \leq N \end{cases}$$

Ce qui donne

$$2 - M \leq k_1 \leq N \tag{2.13}$$

De là, le nouveau nid peut-être réécrit

```
for k1=2-M, N {
   for k2=max(k1,1), min(M-1+k1,N) {
H:    y(k2-k1)=y(k2-k1)+a(k2)*x(-k1)+b(k2)*y(-k1);
   }
}
```

En étudiant les deux formes de ce même algorithme nous pouvons proposer des estimations de performance sur l'architecture PULSE, et ainsi tirer des conclusions sur la manière dont il a été parallélisé. Les résultats sont estimés[11] dans le cas où la boucle interne a été distribuée sur des processeurs différents et ces résultats sont donnés pour un `y(i)` calculé.

En observant le tableau 2.1, nous pouvons constater que la forme parallèle se distingue de la forme primaire par un nombre de communications inter-processeurs inférieur de moitié. Ce gain a des répercussions directes sur le temps d'exécution de l'algorithme. De plus, un élargissement du parallélisme est observé, ce qui donne à l'algorithme le potentiel d'utiliser une architecture parallèle plus large et ainsi distribuer plus efficacement les calculs.

*Tableau 2.1: Filtre IIR 1D*

| | **communications (distance)** | **parallélisme maximal** | **taille de la mémoire** | **calcul** |
|---|---|---|---|---|
| **Forme primaire** | 1 entrée (1)<br>1 sortie (1)<br>2(N-1) inter PE (1+i) | N | 2N | N *macc*<br>2N *mult*<br>2(N-1) *comm.* |
| **Forme parallèle** | 1 entrée (1)<br>1 sortie (1)<br>N-1 inter PE (1) | N+*i*<br>$\forall(2-M \le i \le N)$ | 2N | N *macc*<br>2N *mult*<br>N-1 *comm.* |

## 2.2. Le filtre FIR 2D

Le filtre suivant est le FIR 2D. Il est de la famille des filtres manipulant des données sur deux dimensions. Cette famille s'attaque aux problèmes d'imagerie en particuliers et aux calculs matriciels à deux dimensions en général. Ces filtres sont généralement

[11] Cette étude et analyse des performances de l'algorithme sur le système PULSE est réalise par l'environnement SimPULSE décrit dans le chapitre suivant.

utilisés dans le cadre du traitement d'images pour réaliser des tâches du type détection de contours, réduction de bruit, ou rehaussement de couleurs.

La fonction de transfert d'un filtre FIR 2D est

$$H(z_1, z_2) = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} w_{i,j} \cdot z_1^{-i} \cdot z_2^{-j} \tag{2.14}$$

Cette équation peut-être directement réécrite dans le domaine temporel suivant une convolution 2D

$$y_{m,n} = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} w_{i,j} \cdot x_{m-i,n-j} \tag{2.15}$$

où $x_{m,n}$ et $y_{m,n}$ sont respectivement les données entrantes et sortantes, et $w_{k,k}$ représente les coefficients.

Tel que mentionné ci-dessus, une très large classe de traitements d'images peut-être réalisée en utilisant ce modèle de filtre de convolution 2D. Pour une taille de fenêtre de convolution donnée seul les coefficients $w_{i,j}$ différencient les filtres les uns des autres. Leur valeur peut-être constante, variable ou fonction d'une entrée du signal.

Pour des applications de traitement d'images classiques, la fenêtre de convolution (délimitée par les coefficients $w_{i,j}$) varie entre 3x3 et 9x9. Cela nous indique que dans le meilleur des cas il faut réaliser 9 multiplications-accumulations et 81 dans le pire des cas pour chaque nouveau résultat produit. Parallèlement, de 3 à 9 lignes d'image doivent être mémorisées pour les calculs ultérieurs[12] lorsque les données sont traitées au fur et a mesure de leur arrivée.

---

[12] Ce chiffre peut monter jusqu'à 18 dans le cas où $w_{i,j}$ serait en fonction des données d'entrée.

L'implantation logicielle de ce filtre peut-être exprimée comme suit

```
L1: for i1=0, N-1 {
L2:   for i2=0, M-1 {
L3:     for i3=0, K-1 {
L4:       for i4=0, K-1 {
H:          y(i1,i2)=y(i1,i2)+w(i3,i4)*x(i1-i3,i2-i4);
          }
        }
      }
    }
```

où N, M, et K représentent respectivement la largeur, la hauteur de l'image à traiter, et la taille de la fenêtre de convolution. Du point de vue de l'implantation, il n'y a pas beaucoup de différences avec le filtre IIR 1D présenté à la sous-section précédente. Un soin particulier a été apporté pour que ce nid soit écrit sous une forme *parfaite*. Pour une exécution réelle, il est supposé que toutes les variables sont initialisées à la valeur zéro avant leur utilisation.

La réalisation d'une parallélisation orientée vers l'architecture PULSE commence par une analyse des dépendances. Soit *D*, le résultat de cette analyse

$$D=\left\{\begin{pmatrix} i_1-i_3 \\ i_2-i_4 \end{pmatrix}\begin{pmatrix} i_3 \\ i_4 \end{pmatrix}\right\} \tag{2.16}$$

Le premier vecteur reflète les dépendances qui existent entre $w(i_3,i_4)$ et $y(i_1,i_2)$. Ce vecteur ne nous intéresse pas car nous supposons que les coefficients sont distribués localement sur tous les processeurs ou transmis par les instructions elles-mêmes. En revanche, le second vecteur de dépendances est plus intéressant, car il reflète de réelles dépendances entre $x(i_1-i_3,i_2-i_4)$ et $y(i_1,i_2)$. Il est à noter que ce vecteur n'est pas constant et dépend des indices de boucles $i_3$ et $i_4$, ce qui équivaut à dire que nous nous

trouvons en présence de $K^2$ vecteurs de dépendances. Le vecteur de dépendance et son domaine d'itération[13] sont représentés ci-dessous

$$D = \left\{ \begin{pmatrix} i_3 \\ i_4 \end{pmatrix} \right\} \tag{2.17}$$

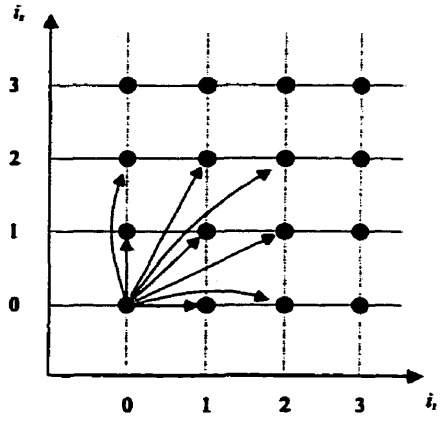


*Figure 2.2: Domaine d'itération du filtre FIR 2D*

Pour palier au problème de différence entre la dimension des tableaux du filtre (`y(i)` et `x(i)`) et la largeur du vecteur d'itération du nid de boucles (largeur de 4 dans ce cas), une nouvelle instruction `H'` est crée. Cette instruction `H'` regroupe `H` et les boucles `L3` et `L4`.

L'objectif est de n'avoir aucune dépendances dans la boucle externe, ce qui donne comme effet de pouvoir la paralléliser. Ce découpage du parallélisme permet de diviser les images à traiter par bandes, et ainsi faire traiter chaque bande par un processeur. Ce qui donne comme avantage que plus il y a de processeurs disponibles, plus les bandes seront petites et ainsi plus le balayage d'une image sera rapide.

---

[13] Pour des raisons de clarté, seules les dépendances de l'opération (0,0) ont été exposées.

Cet objectif pousse à avoir des dépendances résultantes qui s'écrivent sous la forme

$$D' = \left\{ \begin{pmatrix} d'_1 \\ d'_2 \end{pmatrix} \right\} \tag{2.18}$$

où $\forall(([d'_1, d'_2] \in Z) \wedge (d'_1 \approx 0) \wedge ([d'_1, d'_2] \geq 0))$ (voir note [14]) (2.19)

Il faut maintenant trouver une matrice unimodulaire qui satisfasse cet objectif. Pour cela il faut résoudre le système d'équations suivant

$$\begin{cases} 0 = u_{11} i_3 + u_{12} i_4 \\ 0 \leq u_{21} i_3 + u_{22} i_4 \\ u_{11} u_{22} - u_{12} u_{21} = \pm 1 \end{cases} \Rightarrow \begin{cases} u_{11} = -i_4 / d'_2 \\ u_{12} = i_3 / d'_2 \\ u_{21} = 0 \\ u_{22} = d'_2 / i_4 \end{cases}$$

Si $d'_2 = i_4$, la matrice $U$ obtenue est la suivante

$$U = \begin{bmatrix} -1 & i_3 / i_4 \\ 0 & 1 \end{bmatrix} \text{ avec } \Delta = -1 \tag{2.20}$$

Les dépendances deviennent

$$D' = \left\{ \begin{pmatrix} 0 \\ i_4 \end{pmatrix} \right\} \tag{2.21}$$

Remarquons que cette dépendance est légale car elle est lexicographiquement positive, puisque $i_4$ varie de 0 à k-1 ce qui lui assure d'être toujours positif. De plus, $D'$ rencontre les exigences déterminées par l'objectif fixé au départ. Nous pouvons en déduire que les transformations effectuées sur ce nid de boucles sont satisfaisantes.

---

[14] A partir de ce point et pour tout le reste de ce mémoire, le signe $\approx 0$ est utilisé pour indiquer : le plus petit possible.

L'étape suivante consiste à extraire de nouveaux indices de boucle : $k_1$ et $k_2$.

$$(i_1,i_2)^T=\begin{bmatrix}1 & i_3/i_4\\ 0 & -1\end{bmatrix}(k_1,k_2)^T=(k_1+k_2\frac{i_3}{i_4},-k_2)^T \tag{2.22}$$

Les nouveaux axes $k_1$ et $k_2$ sont difficilement représentables sur un domaine d'itération à deux dimensions dû au fait qu'ils dépendent de variables.
Le code de ce nouveau nid doit être généré pour obtenir le filtre exprimé avec un parallélisme explicite. Les bornes de $k_1$ et $k_2$ sont extraites à partir de l'inéquation suivante

$$(0,0)\le\left(k_1+k_2\frac{i_3}{i_4},-k_2\right)\le(N-1,M-1) \tag{2.23}$$

Pour extraire les nouveaux indices, la technique d'élimination de variable proposée par Fourier-Motzkin est utilisé pour $k_2$ et donne

$$\begin{cases}0\le k_1+k_2\frac{i_3}{i_4}\\ 0\le -k_2\\ k_1+k_2\frac{i_3}{i_4}\le N-1\\ -k_2\le M-1\end{cases}$$

soit,

$$\left\lceil\max\left(-k_1\frac{i_4}{i_3},1-M\right)\right\rceil\le k_2\le\left\lfloor\min\left((N-1-k_1)\frac{i_4}{i_3},0\right)\right\rfloor \tag{2.24}$$

Pour extraire $k_1$ la procédure suivante est utilisée

$$\begin{cases}k_1\le 0\\ \frac{i_3}{i_4}\left(1-M-\frac{i_4}{i_3}(N-1)\right)\ge k_1\end{cases}$$

soit,

$$0 \le k_1 \le N-1+\frac{i_3}{i_4}(M-1) \tag{2.25}$$

Ainsi le nid de boucles peut-être réécrit sous une forme parallèle. Remarquons que les boucles `L3` et `L4` étaient considérées jusqu'à présent parties intégrantes de l'instruction `H'`. Dorénavant, les boucles `L1` et `L2` sont fonction des indices des boucles internes. L'indice d'une boucle externe ne peut pas être fonction d'un indice d'une boucle plus interne à elle-même. Ceci nous pousse à réaliser une permutation des deux boucles externes avec les boucles internes, ce qui donne après réécriture

```
L3: for i3=0, K-1 {
L4:   for i4=0, K-1 {
L1:     for k1=0, N-1+i3/i4(M-1) {
L2:       for k2=max(-k1(i3/i4),1-M), min((N-1-k1)i4/i3,0) {
H:         y(k1+k2(i3/i4),-k2)+=w(i3,i4)*x(k1+k2(i3/i4)-i3,-k2-i4);
          }
        }
      }
    }
```

En étudiant les deux formes de ce même algorithme, nous pouvons proposer des estimations de performance sur l'architecture PULSE, et ainsi tirer des conclusions sur la manière dont il a été parallélisé. Les résultats sont estimés dans le cas où la boucle interne a été distribuée sur des processeurs différents[15] et sont donnés pour un `y(i)` produit et sortie de la structure de processeurs.

De plus, une comparaison avec des résultats de performances réelles a été réalisée pour comparer cette méthode de parallélisation automatique avec les solutions intuitives. Les performances réelles sont obtenues avec une allocation dite par *bloc* (voir figure 2.4a et b) suivant la terminologie de Kees van Reeuwijk (1996). La méthode soutenue par ce

---

[15] Pour garder une structure de processeurs linéaires.

mémoire est une allocation dite *cyclique* (voir figure 2.4c). Cette remarque est à prendre en considération lors des comparaisons[16].

En étudiant le tableau 2.2, nous pouvons constater que la forme parallèle se distingue de la forme primaire par un nombre de communication inter-processeurs d'un ordre inférieur. Cette diminution a des répercussions directes sur le temps d'exécution de l'algorithme. Il est à noter que plus la fenêtre de convolution est grande, plus ce gain est important, puisqu'il dépend de la taille K. De plus, un élargissement du parallélisme est observé, ce qui donne à l'algorithme le potentiel d'utiliser une architecture parallèle plus large et ainsi de distribuer plus efficacement les calculs. Cet élargissement permet de soulager le travail effectué par chaque processeur et de gagner sur le temps de calcul total.

*Tableau 2.2: Filtre FIR 2D*

| | **communications (distance)** | **parallélisme maximal** | **taille de la mémoire[17]** | **calculs** |
|---|---|---|---|---|
| **Forme primaire** | 1 entrée (1)<br>1 sortie (1)<br>$K^2$/ inter PE (max. de 2) | N-1 | (K-1)N/P<br>$+K^2+K-1$ | $K^2$ *macc*<br>$K^2$ *comm.* |
| **Forme parallèle** | 1 entrée (1)<br>1 sortie (1)<br>3 inter PE (max. de 2) | $(N-1-k_1)i_1/i_2$<br>$\forall(0 \le (i_1,i_2) \le K-1)$ | (K-1)N/P<br>$+K^2+K-1$ | $K^2$ *macc*<br>*K comm.* |
| **Solution manuelle** | 1 entrée (1)<br>1 sortie (1)<br>1 inter PE (1) | $K^2$ | (K-1)N/P<br>$+K^2+K-1$ | $K^2$ *macc*<br>*1 comm.* |

[16] Une discussion plus détaillée est effectuée à la section 2.6.

[17] Dans ce cas nous supposons que les coefficients $w_{ij}$ sont constants.

## 2.3. Les vilains petits canards

Les *vilains petits canards* ne sont pas de nouveaux filtres pour le traitement du signal, mais tout simplement un pseudonyme utilisé par l'auteur, sous lequel se cache une catégorie de filtres qui possèdent des dépendances tellement restrictives qu'il n'est pas possible de trouver une matrice unimodulaire de transformation qui diminue le nombre de dépendances. Deux filtres sont présentés à titre d'exemple : le filtre de détection de contours à trois pixels, et le filtre médian 3x3.

## 2.4. Le filtre de détection de contours

Le filtre de détection de contours est une réduction du filtre de convolution. L'avantage que propose cette approche est la diminution du nombre d'opération et la suppression de coefficients. Il perd toutefois de la précision puisque la fenêtre est plus petite. Il peut être exprimé ainsi

$$y_{i,j} = \max\left(\left|x_{i,j} - x_{i-1,j}\right|, \left|x_{i,j} - x_{i,j-1}\right|\right) \tag{2.26}$$

où $x_m$ et $y_m$ sont respectivement les signaux entrant et sortant. Son implantation logicielle s'écrit comme suit

```
L1: for i1=0, N-1 {
L2:   for i2=0, M-1 {
H1:     a = abs(x(i1,i2)-x(i1-1,i2));
H2:     b = abs(x(i1,i2)-x(i1,i2-1));
H3:     y(i1,i2)=max(a,b);
      }
    }
```

où N et M représentent respectivement la largeur et la hauteur de l'image à traiter. Un soin particulier a été apporté pour que ce nid soit écrit sous une forme *parfaite*.

Soit $D$, le résultat de l'analyse de dépendances réalisée sur (2.26)

$$D=\left\{\begin{pmatrix}0\\1\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix}\right\} \tag{2.27}$$

Pour cet algorithme, comme pour les précédents, notre objectif est de n'avoir aucune dépendance dans la boucle externe pour permettre de la paralléliser, et ainsi de réaliser un traitement par bande pour augmenter le nombre de processeurs et réduire la charge individuelle des processeurs. Cet objectif nous pousse à avoir des dépendances résultantes qui s'écrivent sous la forme

$$D'=\left\{\begin{pmatrix}d'_{11}\\d'_{12}\end{pmatrix}\begin{pmatrix}d'_{21}\\d'_{22}\end{pmatrix}\right\} \tag{2.28}$$

où $\forall(([d'_{11},d'_{12},d'_{21},d'_{22}]\in Z)\wedge([d'_{11},d'_{21}]=0)\wedge([d'_{11},d'_{12},d'_{21},d'_{22}]\geq 0))$ (2.29)

Il nous suffit de trouver une matrice unimodulaire de transformation qui satisfait les objectifs pour ainsi permettre la réécriture du nid de boucles.
Soit $U$, la matrice unimodulaire de transformation recherchée, telle que

$$\begin{cases} d'_{11}=u_{11} \\ d'_{12}=u_{21} \\ d'_{21}=u_{12} \\ d'_{22}=u_{22} \\ u_{11}u_{22}-u_{12}u_{21}=\pm 1 \end{cases} \tag{2.30}$$

En cherchant à résoudre ce système, nous constatons qu'il n'y a pas de solution pouvant satisfaire les objectifs avec moins de dépendances que n'en possède la forme originale. Cet algorithme possède des dépendances contenant trop de contraintes pour pouvoir trouver une forme plus performante.

## 2.5. Le filtre médian

La figure 2.3 illustre un filtre médian 3x3 sur lequel nous allons tenter d'extraire le parallélisme implicite.

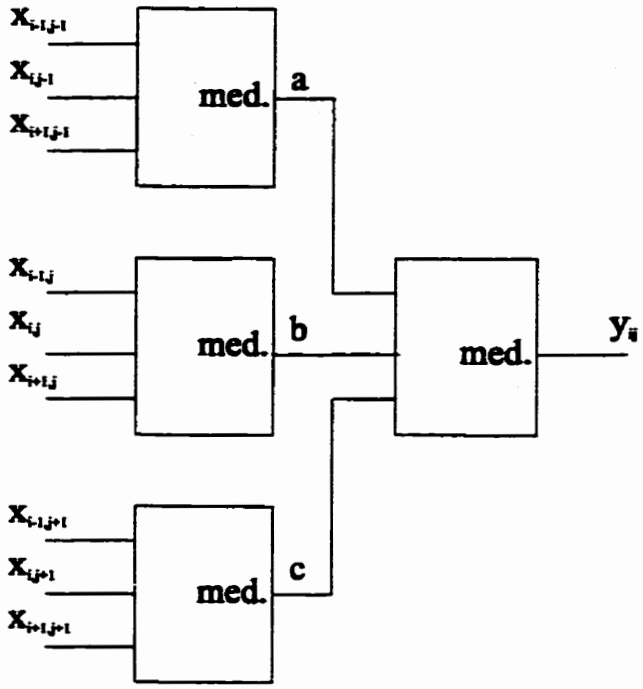


*Figure 2.3: Filtre médian 3x3*

L'implantation logiciel de ce filtre peut-être exprimé tel que suit

```
L1: for i1=0, N-1 {
L2:   for i2=0, M-1 {
H1:     a = med(x(i1-1,i2-1), x(i1,i2-1), x(i1+1,i2-1));
H2:     b = med(x(i1-1,i2),   x(i1,i2),   x(i1+1,i2));
H3:     c = med(x(i1-1,i2+1), x(i1,i2+1), x(i1+1,i2+1));
H4:     y(i1,i2)=med(a,b,c);
      }
    }
```

où N et M représentent respectivement la largeur et la hauteur de l'image à traiter.

Soit $D$,

$$D = \left\{ \begin{pmatrix} -1 \\ -1 \end{pmatrix} \begin{pmatrix} -1 \\ 0 \end{pmatrix} \begin{pmatrix} -1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\} \tag{2.31}$$

En observant $D$, il est remarquable que les vecteurs de dépendances couvrent toute la gamme de solutions possibles. Ce qui pousse à conclure qu'il est impossible de trouver une matrice de transformation unimodulaire qui puisse diminuer le nombre de dépendances ou leurs distances.

## 2.6. Conclusion

D'une manière globale pour tous les algorithmes traités dans ce chapitre, excepté pour certains cas pathologiques, nous observons après transformation un élargissement du parallélisme et une diminution des communications inter-processeurs. Cette méthode permet d'utiliser une architecture parallèle plus large et ainsi de distribuer plus efficacement les calculs. Ceci permet de soulager le travail effectué par chaque processeur, ce qui réduit le temps de calcul car chacun possède une charge moins importante.

En comparant nos résultats avec ceux obtenus par partitionnements manuels, nous constatons des différences notables. Elles proviennent en grande partie de la manière dont l'allocation a été réalisée, et particulièrement de l'optimisation locale due aux subtilités de l'architecture cible.
Ces optimisations locales sont difficiles à intégrer dans une méthode de partitionnement systématique pour obtenir des résultats probants. En revanche, une allocation non-optimisée est plus facile à analyser et à corriger. Nous allons donc nous y attarder. Il est à noter que notre méthode recherche une allocation avec un parallélisme maximal suivie d'une recherche d'un partitionnement vers l'architecture cible. Contrairement à la méthode manuelle qui permet de faire ces étapes simultanément.

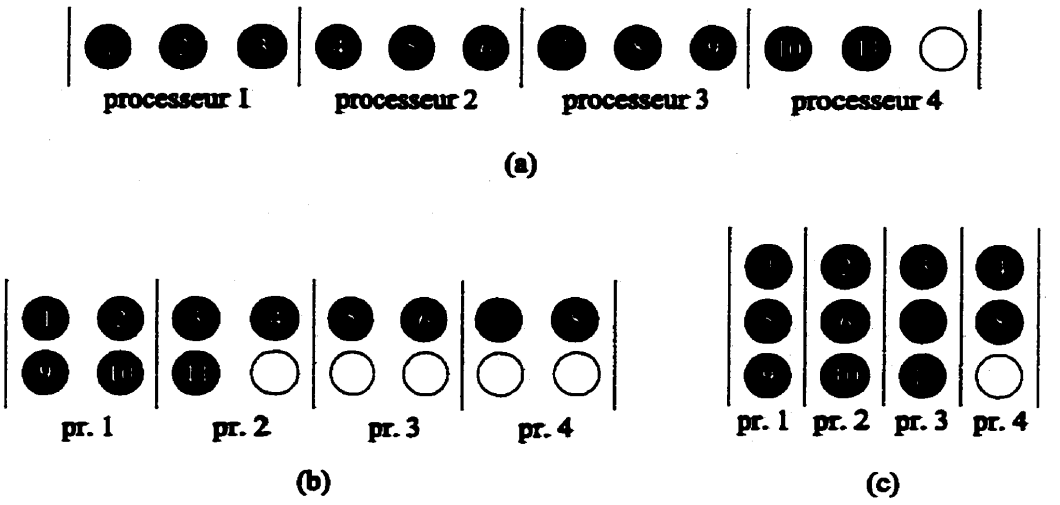


*Figure 2.4: Différents types d'allocation: (a) allocation par bloc, (b) allocation bloc/cyclique, (c) allocation cyclique*

Tel que mentionné plus tôt, le partitionnement manuel réalise une allocation dite par *bloc*. Cette allocation permet de regrouper des opérations qui possèdent des données et résultats communs sur un même processeur (voir figure 2.4a (K. Van Keeuwijk, W. Denissen, H.J. Sips et E.M.R.M. Paalvast, 1996)). Cette technique permet aux données et résultats générés par des opérations voisines de se trouver sur le même processeur et ainsi faire des échanges, si besoin est, sans réaliser des communications inter-processeurs. La figure 2.4b est une extension de cette méthode d'allocation appelé *bloc/cyclique*.

Ces méthodes d'allocation par bloc, cyclique ou bloc/cyclique ont un côté très séduisant qui pousse à vouloir regrouper sur le même processeur toutes les opérations qui communiquent ensemble. Cette possibilité, observée récemment, ne sera pas analysée dans la suite de ce mémoire faute de manque de temps.

Le chapitre suivant présente un outil qui permet de réaliser des estimations de performances d'algorithmes, de partitionnements et de choix architecturaux d'une manière semi-automatique.

# CHAPITRE III

# LE SIMULATEUR SIMPULSE

Dans les chapitres précédents nous parlions de résultats de performance, mais performances de quoi exactement ?

Lorsqu'il est question de système, en l'occurrence d'un système basé sur un ou plusieurs processeurs, il est sous-entendu que nous parlons d'un regroupement de trois parties distinctes qui le composent : la partie applicative, la partie logicielle et la partie matérielle (représentée à la figure 3.1). La première partie regroupe l'ensemble des applications et librairies associées. La partie logicielle est composée des outils logiciels qui traduisent les applications en langage interprétable par le matériel. Cette partie est essentiellement composée des compilateurs de différents niveaux. Quant à la troisième partie, elle associe les composants matériels représentés par : le ou les processeurs, et les périphériques adjacents (tel que les mémoires, interfaces et autres processeurs connexes).

Dans un tel système, il est clair que les performances finales reflètent la cohérence et les performances locale de chacune des trois parties. Si à un moment donné les prouesses de l'une s'écroulent, ce sont les performances du système au complet qui s'en trouvent touchées. C'est pour cette raison qu'il faut à tout moment prendre en compte les trois parties pour réaliser nos estimations de performance.

Pour faciliter la tâche d'estimation de performance, nous avons réalisé un outil développé sous l'aspect d'une interface. Cet outil a été conçu pour réaliser des simulations au niveau système, c'est-à-dire simuler le comportement système le plus proche possible de ce qu'il va être lors de sa réalisation. Grâce à son interface, cet outil

regroupe les différentes équipes travaillant sur le même projet autour d'un même environnement de simulation. Cela permet d'instaurer une cohésion entre elles. Cette cohésion découle de l'utilisation d'un même modèle de référence au travers d'une interface commune. Cet outil permet aux équipes de développer des sections du projet, chacunes de leurs côtés, puis de les intégrer dans un modèle commun de simulation, de manière transparente pour les autres équipes. En d'autres mots, lorsqu'une équipe réalise ou met à jour une section du projet, pour la rendre publique, elle doit simplement l'insérer dans le modèle commun. Ainsi à chaque utilisation de cet environnement, toutes ces sections sont mises à contribution pour mener à bien la simulation. De plus, ce partage des sections permet aux équipes de peaufiner l'interaction entre elles et d'éviter des chevauchements.

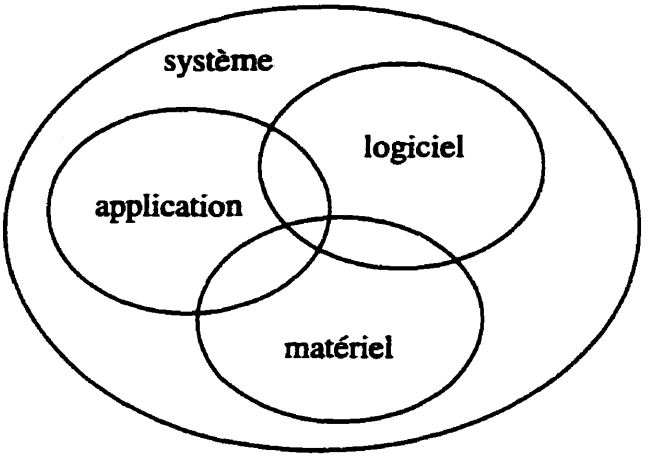


*Figure 3.1: Représentation d'un système complet*

Un article[18] inclut à l'annexe I décrit cette méthode de simulation haut niveau à partir d'un modèle système utilisé et développé par plusieurs équipes de travail. Le contenu de cet article est complémentaire à la discussion présentée dans ce chapitre. Le traitement

[18] Référence : N. Contandriopoulos, Y. Blaquière and Y. Savaria, *Simulation Based Methodology for the Development of Complex VLSI Systems*, soumis pour publication.

dans l'article est plus conceptuel et ne présente l'outil qu'avec une vue de très haut niveau.

La suite de ce chapitre décrit les objectifs visés et les méthodes misent en œuvre pour les atteindre. Trois sections composent le corps de ce chapitre. La section 3.1 décrit sommairement la pertinence et les objectifs visés par cet outil. La section suivante (section 3.2) explique les deux versions de l'outil développés, elle présente leur fonctionnement, et insiste sur les différences. La dernière section (section 3.3) décrit les méthodes et les manières avec lesquelles les performances du modèle simulé sont estimées.

## 3.1. Pertinence et objectifs

Ils existent plusieurs environnements de simulation dans le domaine de l'aide à la conception et à la vérification de système (L. Maliniak, 1994). Les principaux facteurs menant au développement d'un nouvel environnement adapté aux projet PULSE sont décrit dans cette section. Nous allons essayer de répondre à cette question en énumérant les principaux points qui justifient la pertinence de ce nouvel environnement dans le cadre du projet PULSE.

Une solution, largement adoptée lors de la réalisation de systèmes, est de concevoir une machine virtuelle qui simule le comportement escompté. De là, des mouchards sont implantés dans le modèle pour permettre d'analyser son comportement et d'extraire des estimés de performance. La méthode classique consiste à développer un modèle spécifique pour la simulation, séparé du modèle du système en cours de développement. Ainsi, à chaque mise à jour, soit du modèle, soit de la machine virtuelle, des réajustements doivent être effectuées. Ce dédoublement peut apporter des problèmes de cohérence.

De plus, la machine virtuelle ne peut pas suivre le modèle du système à travers toutes les phases de conception, où de nombreuses adaptations doivent souvent être apportées. Pour éviter ce type de problème, nous proposons que la machine virtuelle soit représentée par le modèle en cours de développement : le modèle VHDL (VHSIC Hardware Description Language). Ainsi, les problèmes de cohérence sont grandement réduits puisqu'il n'y a qu'un modèle unique pour la conception, la vérification, et l'analyse.

## 3.2. Description sommaire de l'environnement de simulation

L'environnement proposé pour la simulation de bas niveau s'appuie sur une base constituée de :

- la description en langage de description de matériel des différentes configurations du système projeté (modèle VHDL),
- son environnement de programmation,
- des outils de simulation, d'analyse et d'estimation de performances.

Une coquille recouvre cette base. Cette dernière permet de rehausser le niveau d'abstraction des simulateurs VHDL classique et des outils de programmation, très détaillés, complexes et lourds d'utilisation, au niveau instructions machine. Elle offre un accès à tous les membres du projet la possibilité d'évaluer, d'analyser et d'estimer les performances des différents modèles virtuels du système projeté.

Cette description VHDL du système matériel, permet de reproduire virtuellement une vaste gamme d'environnements auxquels le système sera confronté. Les simulations commandées permettent d'obtenir des détails relatifs à tous les signaux composants réellement les circuits du système (ce qui n'est pas possible, ou complexe à réaliser, à partir de composants finis ou de machines virtuelles). En analysant les signaux

résultants de la simulation, l'utilisateur peut analyser le fonctionnement du système d'une manière dynamique et très proche de la réalité[19].

Pour tirer le maximum d'information des simulations, l'environnement propose deux méthodes d'analyse. La première, appelée la méthode *monitoring*, permet de passer en revue de façon détaillée les résultats d'une simulation. Elle permet d'étudier le contenu des différents registres et mémoires du modèle pour chaque cycle de la simulation d'une application. Son rôle est de permettre à l'utilisateur de visualiser finement l'état du système, cycle après cycle. Un outil de navigation (voir un exemple à la figure 3.5) permet de se promener à travers les cycles de simulations en réalisant un déroulement pas à pas ou par sauts en utilisant des points d'arrêts (*break points*). Une des utilisations possible de cette méthode est la validation de la fonctionnalité. L'utilisateur cherche alors à analyser les résultats produits par les unités fonctionnelles. Cette méthode est surtout utilisée par les personnes qui développent des outils de programmation ou des applications logicielles. Dans ce cas, c'est le déroulement d'une application et les interactions avec le monde extérieur qui sont observés.

La seconde méthode est appelée *analyse* (voir figure 3.4(d) pour voir un exemple de compte rendu d'analyse réalisé). Cette méthode permet d'aller chercher des informations plus globales sur l'état et le comportement du système durant une simulation et de réaliser des traitements et analyses sur ces dernières. C'est à partir de ce regroupement d'informations que des estimés de coûts et de performances du système sont réalisés. Ces estimés peuvent s'effectuer sur une partie restreinte ou globale de la simulation réalisée. C'est grâce à cette méthode que sont comparés les performances découlant des choix architecturaux et la capacité des outils de programmation à interpréter les applications pour une configuration matérielle spécifique. En utilisant cette méthode d'analyse, ce ne sont plus les valeurs des résultats qui importent, mais plutôt l'interprétation et l'analyses des performances des contrôleurs, des ports

---

[19] Puisque c'est à partir de cette description VHDL que le circuit est synthétisé pour la conception.

d'entrées/sorties, et des communications entre les différentes unités fonctionnelles. Les types de résultats obtenus sont principalement des analyses de remplissage des mémoires et registres, le degré de parallélisme entre les unités, le taux d'utilisation des unités ou le taux de transfert d'information entre les unités. De plus, des fonctions ont été développées pour comparer des applications ou des configurations entre elles, mais aussi pour comparer le système en développement avec des systèmes concurrents via des mesures de performances globales (*i.e.* MIPS, MOPS, CPI,...). La section 3.3 est entièrement consacrée à la description de ces mesures.

Une des forces de cette approche, basée sur l'environnement de simulation, est le fait que lors d'une simulation pour une analyse ou une estimation de coût d'application logicielle sur une configuration matérielle, les mesures de performance portent sur l'ensemble du système. Ces mesures nous renseignent non seulement sur la structure modèle, mais aussi sur la manière avec laquelle les outils de programmation interprètent l'application et l'adaptent efficacement à la configuration matérielle du modèle. Il est à noter que les résultats interprétés pour l'analyse et l'estimation proviennent directement de la simulation du modèle détaillé. La même remarque peut être faite pour les outils de programmation logiciels : les outils utilisés pour la simulation sont ceux développés spécifiquement pour la programmation du système réel.

## 3.3. L'interface de l'environnement de simulation

Dans cette section, l'outil qui réalise l'environnement de simulation et son interface avec l'utilisateur est décrit. Dans un premier temps, l'implantation logicielle de cette interface et la manière avec laquelle elle interagit avec l'utilisateur est décrit succinctement. Dans un second temps, nous allons discuter des différences entre les deux versions qui ont été développées (sous-sections 3.2.1 et 3.2.2).

Cet outil procède par trois étapes distinctes : la saisie des données ; la simulation ; l'interprétation et l'analyse.

1. - **La saisie des données :** Cette saisie s'effectue via une interface graphique. Le rôle de cette interface est de collecter les paramètres de configuration, les données et les applications que souhaitent simuler l'utilisateur. De plus, cette interface doit être conviviale et adaptée à tous les profils d'utilisateur. Il est à noter que les utilisateurs appartiennent à des domaines de compétence variés et ont des attentes très diverses envers l'interface. Certains utilisateurs désirent voir le détail de l'exécution d'une application. D'autre cherchent à extraire les performances globales du système. Enfin, une troisième catégorie d'utilisateurs est intéressée par la vérification de la fonctionnalité d'un module donné. Cette interface doit pouvoir évoluer et s'adapter rapidement aux changements et modifications du système.
2. - **La simulation :** Le rôle de cette étape est de simuler le comportement du système avec un des modèles, selon la précision désirée. Les résultats de simulation sont ensuite traites pour extraire les paramètres pertinents sous une forme compréhensible par les outils d'interprétation et d'analyse. Une bibliothèque de modèles simulables du système existe pour présenter à l'utilisateur divers cas de figure dans lesquels le système se trouvera lors de son utilisation réelle.
3. - **L'interprétation et analyse :** Une fois la simulation complétée et les résultats compilés, reste l'étape d'interprétation et d'analyse. Le but est d'interpréter les résultats de simulation afin de les formater pour permettre leurs visualisations ou de les analyser pour extraire le plus fidèlement possible des estimés de performance, ou des statistiques. Cette étape procède en fonction des souhaits de l'utilisateur exprimés lors de la saisie des données. Enfin, une mise en forme est réalisée pour permettre à l'utilisateur une lecture et une interprétation claire et concise des résultats de simulation.

En créant cet environnement de simulation, nous avons créé un regroupement de logiciels et une bibliothèque de modèles de systèmes, qui permettent de passer à travers

les trois étapes décrites ci-dessus. La figure 3.2 représente les interactions qui existent au sein de cette confédération d'outils. Les deux parties externes représentent les interactions de l'environnement de simulation avec l'utilisateur. Ce sont les *interfaces usager*. La partie centrale représente le corps de l'environnement de simulation. Ce dernier regroupe les outils de programmation, simulation, interprétation et analyse.

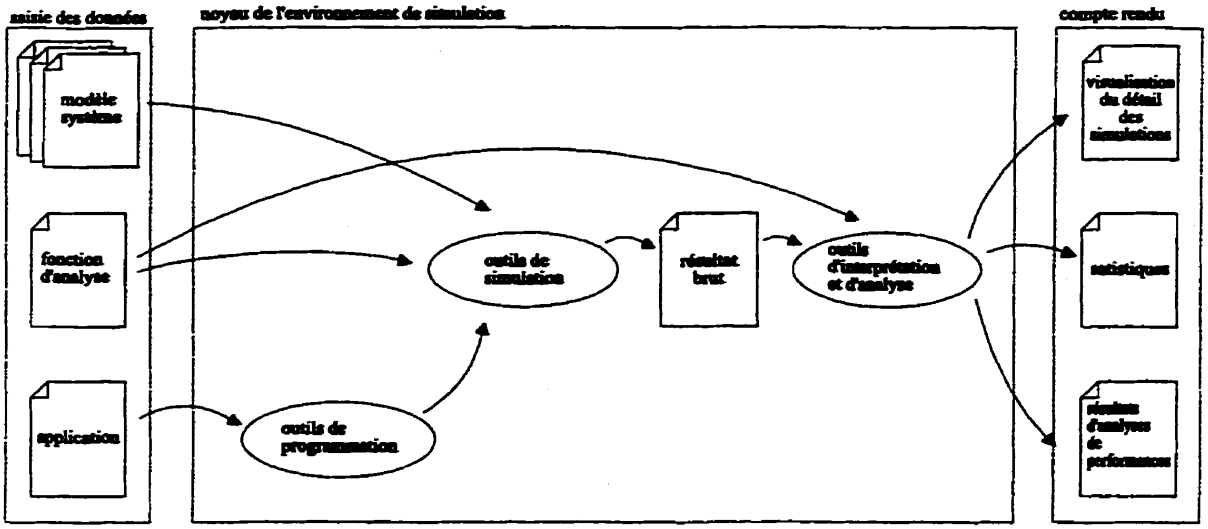


*Figure 3.2: Flux d'information à travers l'environnement de simulation*

Deux versions de cet outil ont été développées d'une manière séquentielle. La première, appelée *SimPULSE-CGI*, a été développée durant le projet PULSE et son évolution a été guidée par les besoins des utilisateurs. La seconde version, appelée *SimPULSE-Java*, a été développée avec la technologie au goût du jour. Elle a bénéficié de l'expérience acquise lors du développement de la première et des observations des utilisateurs. Cette seconde version vise à approcher des outils de qualité industrielle et son utilisation est plus conviviale. De plus, elle propose un mode d'utilisation plus flexible et présente un accroissement de la fonctionnalité. Ce sont ces points que nous allons discuter tout au long des sous-sections suivantes.

### 3.3.1. La première version : SimPULSE-CGI

Dans cette première version, l'attention a été portée sur le corps de l'environnement de simulation et l'aspect graphique de l'interface a été simplifié. Effectivement, une interface basée sur un navigateur HTML (HyperText Mark-up Language) a été utilisée avec un corps constitué par un regroupement de scripts CGI (Common Gateway Interface).

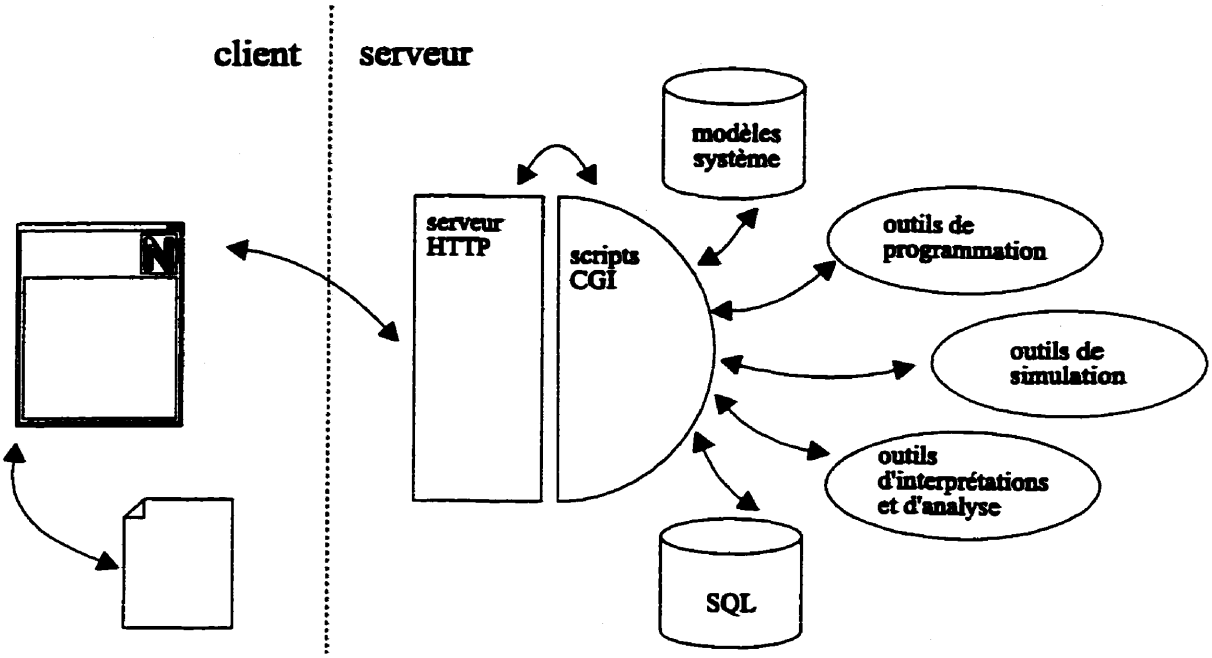


*Figure 3.3: Modèle de fonctionnement du SimPULSE-CGI*

Cet environnement utilise quatre acteurs : le navigateur HTML ; le serveur HTTP (HyperText Transfer Protocol) ; le regroupement de scripts CGI ; et les outils logiciels de programmation, simulation, interprétation, et analyse. Leurs interactions sont représentées à la figure 3.3.

- Le *navigateur HTML* est l'interface usagers. Il réalise la saisie des commandes de l'utilisateur et lui restitue les résultats attendus.

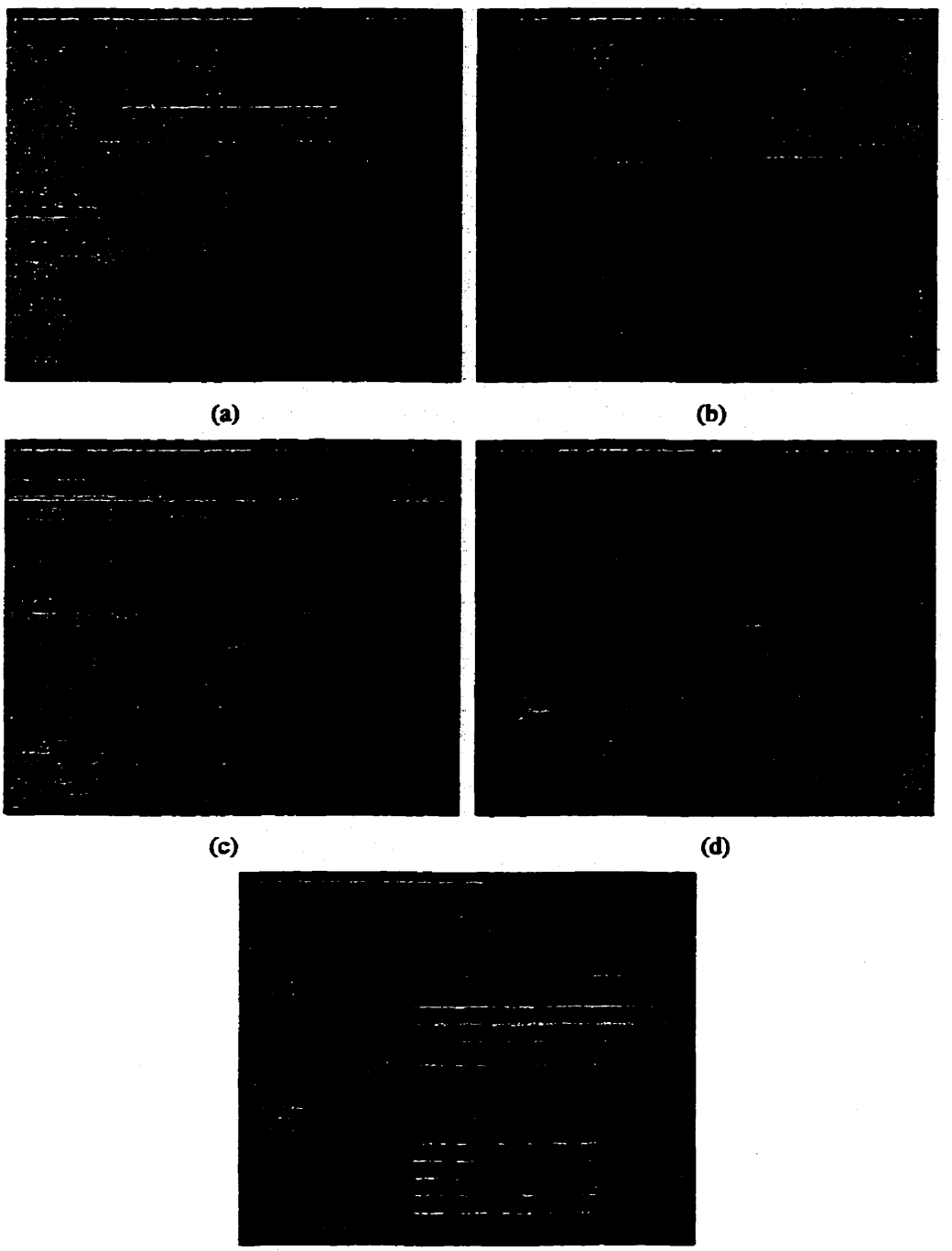

*Figure 3.4: L'interface SimPULSE-CGI*

- Le *serveur HTTP* gère les communications entre les navigateurs HTML des différents utilisateurs et l'environnement de simulation.
- Le *regroupement de scripts CGI* compose le noyau de l'environnement de simulation. Il s'occupe de la gestion des différents utilisateurs, des différents outils logiciels, et des différents modèles de système. De plus, il s'occupe de réaliser la mise en page de l'interface usager et la gestion des résultats de simulation de chaque utilisateur, par l'entremise d'une base de donnée SQL (Server Query Language).
- Les *outils logiciels*, tels que les outils de programmation, simulation, interprétation et analyse, interagissent uniquement avec le noyau de l'environnement. C'est le noyau qui leur indique les tâches qu'ils doivent accomplir et de quelle manière.

Du point de vue de l'utilisateur, SimPULSE-CGI ressemble à un navigateur HTML. Plusieurs pages constituent l'interface usager. La figure 3.4 présente quelque exemples. La page (a) est un exemple où l'utilisateur sélectionne le modèle du système qu'il souhaite simuler. Celle en (b) sert à la saisie des paramètres de configuration, des données et des applications que l'utilisateur souhaite simuler. La page (c) fournit un exemple de page affichant les résultats de simulation. C'est sous cette forme que l'utilisateur visualise, en fonction du temps, les états et le contenu des mémoires et registres du système simulé. La figure 3.5 illustre la barre d'outils qui permet de réaliser cette navigation au travers des résultats d'une simulation. La figure 3.4(d) expose un tableau de résultats d'estimation de performances. Enfin (e) illustre une page qui permet à l'administrateur de l'environnement de simulation d'effectuer la maintenance de l'interface.

*Figure 3.5: Le moteur de recherche de SimPULSE-CGI*

Cette interface fonctionne bien mais elle est exigeante envers le serveur où se trouve logé le noyau de l'environnement de simulation. Le problème apparaît de façon aiguë quand le nombre d'utilisateurs augmente. Pour soulager le serveur et créer de nouvelles fonctionnalités, une seconde version de cette interface a été créée.

### 3.3.2. La seconde version : SimPULSE-Java

Une année après avoir fini de développer la version SimPULSE-CGI, la structure de cette nouvelle version (SimPULSE-Java) était élaborée. Cette version bénéficie de l'expérience acquise et des nouvelles technologies client/serveur promues par le langage Java[20]. Cette solution permet une meilleure cohésion entre le *client Java* (le remplaçant du navigateur HTML) et le groupe : serveur HTTP et les scripts CGI. Le client Java a été entièrement recréé pour cette application. Il répond exactement aux attentes de l'environnement et de l'utilisateur. Le serveur HTTP et les scripts CGI ont été fusionnés en un *serveur Java* dédié à l'environnement de simulation. Il forme ainsi un noyau plus solidaire et plus cohérent, tout en sollicitant moins de ressources au niveau du serveur.

Pour son fonctionnement, cet environnement met à contribution trois acteurs : un client Java ; un serveur Java ; et les outils logiciels de programmation, simulation, interprétation, et analyse. La figure 3.6 illustre les relations qui existent entre ces composants du nouvel environnement de simulation.

- Le *client Java* est l'interface usager. Il est le lien de communication entre le serveur Java et l'utilisateur. Il s'occupe de la gestion graphique, de saisir les informations provenant de l'utilisateur, de les empaqueter et de les envoyer vers le serveur Java. De plus, le client Java gère les résultats de simulation reçus par l'entremise d'une base de données privée et locale. Cette dernière permet de

soulager la charge de travail du serveur et la quantité des communications entre eux.

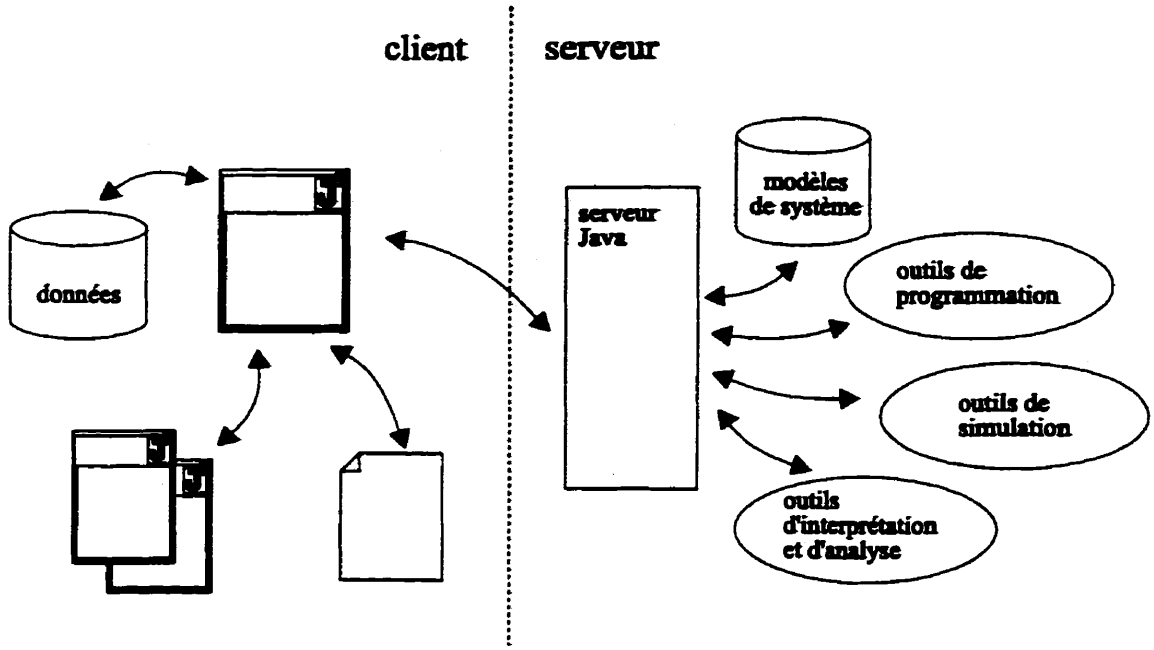


*Figure 3.6: Modèle de fonctionnement du SimPULSE-Java*

- Le *serveur Java* est le noyau de l'environnement de simulation et le gestionnaire de communication avec les différents clients Java. Il s'occupe de la gestion des utilisateurs, des outils logiciels, et des modèles de systèmes. De plus, le serveur Java possède une gestion des préférences des utilisateurs. Comme cela, il peut mieux prédire leurs attentes lors de futures simulations.
- Sous le nom *outils logiciels* sont regroupés : les outils de programmation, de simulation, d'interprétation et d'analyse. Ils interagissent uniquement avec le serveur Java. C'est le serveur Java qui leur indique les tâches qu'ils doivent accomplir et de quelles manières ils doivent les réaliser.

---

[20] Java est une marque de commerce déposée, de Sun Microsystems Inc, d'un langage de programmation orientée objet.

Cette seconde version de l'environnement de simulation propose de nouvelles caractéristiques. Ces caractéristiques sont pour la plupart des améliorations par rapport à la version précédente. Huit d'entre elles sont prédominantes :

- *Gestion des processus* : Une meilleure gestion des processus permet aux utilisateurs de réaliser plusieurs simulations simultanées, de pouvoir visualiser leur état et le cas échéant de les suspendre.
- *Élimination de la base de donnée SQL sur le serveur* : Les résultats de simulation sont transformés par le serveur Java dans un langage (humainement compréhensible) et un format supprimant la redondance d'information. Cette diminution de la quantité d'information permet d'envoyer les résultats aux clients et d'éliminer la charge de travail sur le serveur. C'est le client qui devient gestionnaire des données et c'est lui qui se charge de les trier, de les arranger, et de les mettre en forme pour l'affichage.
- *Enregistrement local des simulations* : Cette caractéristique permet à l'utilisateur, après une simulation, d'enregistrer localement les résultats pour une analyse ultérieure ou pour une comparaison avec une autre.
- *Gestion des communications* : Grâce à cette nouvelle gestion des processus, l'utilisateur, après avoir invoqué la simulation, peut à tout moment couper la communication avec le serveur et y revenir à son gré. Le serveur Java va conserver les données de la simulation en attente d'une nouvelle connexion de l'utilisateur pour les lui distribuer.
- *Gestion des préférences rendant l'interface personnalisable* : Cette caractéristique permet de personnaliser l'outil pour les besoins de l'utilisateur sous forme de configuration.
- *Indépendance du système d'exploitation* : L'interface étant entièrement réalisée avec le langage de programmation Java, la portabilité est assurée sur toutes les plates formes où Java est disponible.
- *Plusieurs modes de fonctionnement* : Cette caractéristique permet aux membres du projet de travailler sur le même modèle à partir de plusieurs lieux

géographiques (caractéristique que possède aussi la version précédente). Ce qui est nouveau, c'est qu'un mode de simulation autonome est rendu possible pour les utilisateurs qui utilisent la machine virtuelle de PULSE (développée en Java) et qui ne désirent pas se connecter sur Internet vers le serveur Java. Ce mode a le mérite de permettre la distribution de cette seconde version sur un support informatique tel qu'un CD-ROM.

- *Plus de ségrégation entre les fureteurs* : L'interface usager est réalisée en Java avec ainsi un mode de fenêtrage qui est indépendant du navigateur HTML. Il est donc indépendant des versions et des compagnies qui développent différents navigateurs.

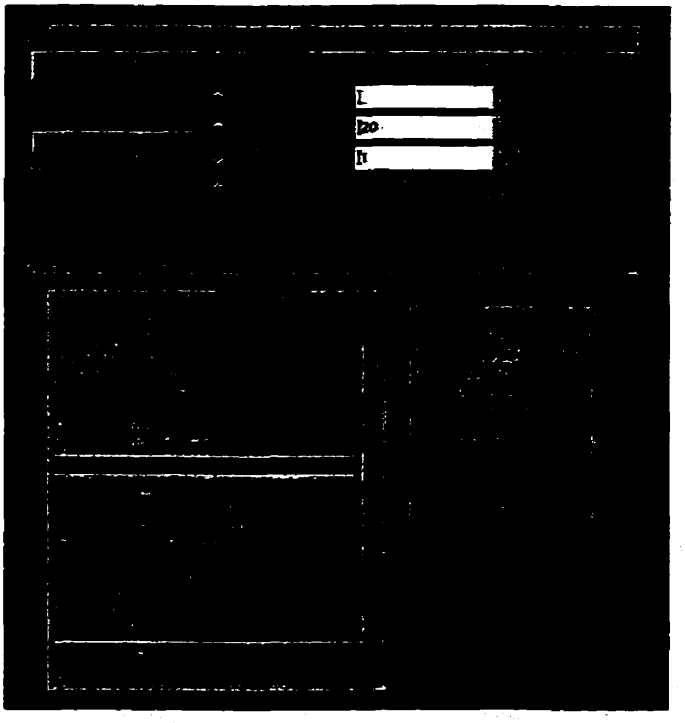

*Figure 3.7 : L'interface SimPULSE-Java: panneau de contrôle et moniteurs*

La nouvelle version de l'environnement de simulation SimPULSE-Java possède aussi des améliorations au niveau de l'interface usager. Cette dernière a été entièrement remodelée en prenant en compte les propositions reçues des utilisateurs de l'ancienne

version. L'amélioration la plus visuelle est le nouveau mode d'affichage multi-fenêtré (voir des exemples sur les figure 3.7, 3.8 et 3.9).
La figure 3.7 présente la fenêtre principale du client Java : le *panneau de contrôle*. De plus, des exemples de *moniteurs* sont illustrés, tel que le moniteur d'une mémoire de programme et le moniteur contenant des états des registres du système PULSE.

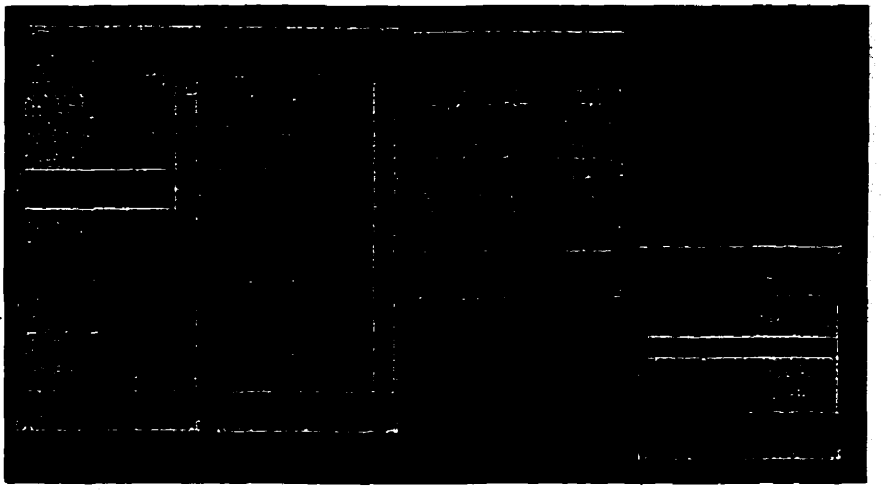

*Figure 3.8: Le contenu d'un processeur PULSE visualisé par SimPULSE-Java*

La fonctionnalité du panneau de contrôle est rendue plus intuitive et permet une plus grande diversité de navigation dans les résultats de simulation. La figure 3.8 illustre ce concept de multi-fenêtrage. Cela permet à l'utilisateur de n'afficher que les moniteurs qu'il souhaite visualiser et de les regrouper à sa guise. Différents moniteurs contenant diverses informations sont représentés.

Dans la version précédente, seuls les données brutes étaient présentées par les moniteurs. Cette nouvelle version apporte des nouvelles fenêtres qui permettent de visualiser le contenu des moniteurs (tel qu'une mémoire) sous une forme d'image et de visualiser son évolution au cours de la simulation (voir figure 3.9).

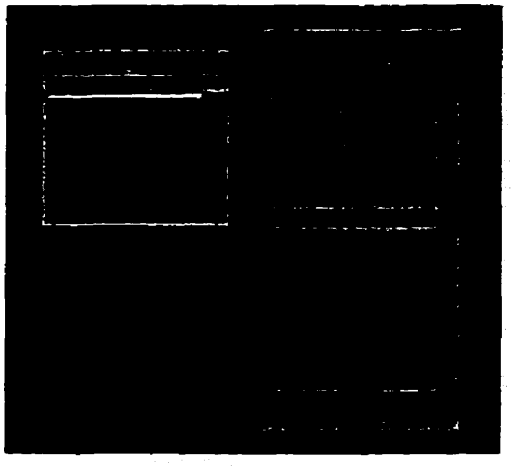

*Figure 3.9: Le contenu d'une mémoire externe visualisé par SimPULSE-Java*

En définitive, cette seconde version a bénéficiée de l'expérience d'une année d'utilisation acquise avec la première et son développement a été réalisé sur un produit où les modifications majeures étaient finies. Ses deux principales évolutions sont, pour l'une, que le serveur et le client ont été conçus spécialement pour cette application, ce qui leur permet de répondre plus efficacement aux attentes des utilisateurs. La seconde grande amélioration est que le client Java peut réaliser des simulations en mode autonome sur une machine virtuelle, sans nécessitées de connexion avec le serveur Java.

## 3.4. Les estimations de performance

Jusqu'à présent, le fonctionnement de notre environnement de simulation et de ses diverses interfaces a été décrit. Une contribution additionnelle importante de cet outil réside dans les outils d'analyse et d'estimation de performance qu'il rend disponible. Ce sont ces fonctionnalités que nous allons détailler dans cette section.

Dans le cadre de la recherche de performance lors d'une exécution d'une application sur un système[21], l'information la plus importante est, dans la majorité des cas, la durée

[21] Dans le contexte de notre environnement, les simulation permettent d'effectuer une analyse dynamique du système.

d'exécution. Cette valeur dépend des performances de l'ensemble du système (voir la figure 3.1). En d'autres mots, elle dépend de : la manière avec laquelle le programmeur a codé l'application qu'il souhaite exécuter ; de l'habilité des compilateurs à interpréter, à comprendre l'architecture cible, et à générer du code efficace dans les conditions prescrites par le programmeur ; et de l'agencement et de la rapidité de la structure matérielle du système pour exécuter l'application.

Ainsi, pour aider l'utilisateur à cerner le ou les goulots d'étranglement du système, l'environnement de simulation effectue plusieurs mesures en divers point lors d'une simulation. C'est à partir de ces mesures que les analyses sont effectuées, et les estimés de performances produits.

Ces mesures sont basées sur trois points :

- l'achalandage des unités (temps et ressources),
- les transferts de données entre les unités,
- le type de tâche accomplie par les unités.

Le tableau 3.1 présente la liste des informations que les outils d'analyse et d'estimation de performances tirent à partir des résultats de simulation.

C'est grâce à l'outil SimPULSE que nous avons pu caractériser les différents partitionement des applications énoncés lors des études au chapitre 3. La première version, SimPULSE-CGI, a été très utilisée par l'équipe de développement d'applications. Par l'utilisation intensive de cet outil, beaucoup d'informations sur la fonctionnalité du système PULSE dans son ensemble ont été générées. Ceci a permi aux équipes de conception de réaliser des modifications et des ajustements basés sur l'expérience. Dans un second temps, en utilisant cette interface, l'équipe de développement d'applications s'est mise à produire des applications qui seront exécutées une fois le système réalisé.

*Tableau 3.1: Estimateurs de performance disponibles*

| Nom | Unité | Valeur mesurée |
|---|---|---|
| nombre de cycles | cycles/algorithme | nombre de cycles atteint jusqu'au terme de la simulation |
| temps d'exécution | unité de temps (ns, ms) | nombre de cycles divisé par la période d'horloge spécifiée pour la simulation |
| cycles par instruction[22] | cycles/instruction | nombre de cycles divisé par le nombre d'instructions |
| MIPS (méga instruction par seconde) | méga instructions/seconde | nombre d'instructions divisé par le temps d'exécution |
| facteur d'accélération[22] | cycles/cycle/algorithme | rapport entre les nombres d'instructions en utilisant ou pas des instructions parallèles |
| taux de transfert | méga octets/seconde | taux de données transférées entre les unités |
| taux de parallélisme | 1) cycles/algorithme<br>2) largeur du parallélisme | nombre de fois que plusieurs chemins de données sont utilisés parallèlement |
| taux d'utilisation des bus | cycles/algorithme | pourcentage en fonction du nombre de fois qu'un bus distinct est utilisé par rapport au nombre de cycles |
| taux d'utilisation d'instruction | cycles/algorithme | comptage des instructions |
| taux d'utilisation des mémoires | 1) pourcentage d'utilisation<br>2) nombre d'accès | occupation de l'espace mémoire |
| imbrication des nids de boucle | cycles/boucle | comptage des appels de boucles |
| type d'instruction | cycles/algorithme | pourcentage des types d'instruction (calcul, contrôle, communication, attente) suivant le nombre de cycles |

La seconde version, SimPULSE-Java, est fonctionnelle (mai 1998). Une machine virtuelle du système PULSE-II en Java a été insérée dans l'environnement de simulation pour permettre son utilisation en mode autonome. Une pénurie relative d'utilisateurs empêche de la certifier *fonctionnelle et stable* et ainsi de la déclarée finie. Pourtant,

[22] Information analysée avec et sans tenir compte des instructions d'attente (*nop*).

cette nouvelle interface a le grand mérite d'être intuitive et conviviale. Cette propriété permet une très grande facilité d'apprentissage de l'utilisation du système PULSE par de nouveaux membres de l'équipe.

# CONCLUSION

La contribution de ce mémoire porte essentiellement sur deux points. Le premier clarifie les mécanismes de transformation de boucles utilisés pour la parallélisation d'applications représentées sous la forme de nids de boucles parfaits. Ces restrictions sont imposées par l'architecture soutenu par le projet PULSE et les applications types exécutées sur cette machine. Ces mécanismes de transformation se traduisent par une réécriture du nid original où le changement de représentation peut-être exprimé en terme matriciel. Cette formulation permet de fixer un cadre d'étude commun à ces manipulations de boucles qui permet de mieux comprendre leur impact sur l'efficacité du code résultant et de faciliter leur introduction dans un outil de transformation automatique. La seconde contribution apportée par ce mémoire est la réalisation et la mise en œuvre de l'interface de simulation qui a permit de quantifier les performances obtenue et d'aider les membres du groupe de développement logiciel PULSE de développer des applications.

La parallélisation automatique est un problème très complexe qui n'est pas encore résolu (NP-Complet). Résumons cependant ceux commençant à être bien compris, en partie grâce aux apports de ce mémoire, et les questions restant en suspend pour chacun d'eux. Ces questions sont toujours liées au problème majeur qui, actuellement, freine l'avancement des compilateurs-parallèliseurs : le problème de la modélisation des communications. Les questions pertinentes se rapportent à l'ordonnancement, l'allocation, et la génération de code.

La minimisation de la latence comme critère de qualité d'un ordonnancement n'est évidemment qu'une heuristique qui revient à trouver un ordre sur les calculs minimisant le nombre de barrières de synchronisation. Le modèle d'exécution sous-jacent ne prend absolument pas en compte les optimisations concernant les communications, la réutilisation des données et la granularité du programme, c'est-à-dire le rapport entre les temps de calcul et les temps de communication. Tout ces problèmes devront être étudiés dans l'avenir et traduits si possible comme des contraintes dans la recherche d'ordonnancements performants.

Le problème de l'allocation est certainement le plus mal compris. Nous avons vu comment dans le cas de nids de boucles simples et parfaitement uniformes le problème de la minimisation des communications pouvait être exprimé en modélisant le coût d'une communication externe par un coût non-nul et par un coût nul pour une communication interne. Le tout se résume à trouver une allocation optimale en résolvant un système d'équations dans le but de rechercher une annulation ou une diminution des coûts attribués aux communications.

La génération de code correspondant à la répartition des données et des calculs ne présente pas trop de difficulté, puisque cette tâche peut-être résolue par la réécriture de boucles en prenant en compte les nouveaux indices et les bornes des boucles. Un vaste travail de programmation reste cependant à effectuer pour produire un outil de transformation de boucles prenant en compte tous les décalages d'indices entre les différentes instructions tant du point de vue spatial que du point de vue temporel : un tel programme permettrait de réécrire automatiquement les nids de boucles en gérant les problèmes liés aux bornes, au ré-enroulement de boucles, et à l'utilisation de pas d'itérations non-unitaire. Conceptuellement, il n'y a pas de difficulté majeure à la complète automatisation de la réécriture et la génération d'un code correct. Par contre, la génération d'un code efficace, dans lequel les expressions des bornes et les structures de contrôle sont simplifiées, nécessite encore un travail important et difficile.

Ceci n'est cependant qu'une partie du problème de la génération de code. Il faut également être capable de générer tous les transferts de données impliqués par une telle répartition : communications externes et vectorisation de messages, gestion de la mémoire et réutilisation de données dans le cas d'allocation de plusieurs opérations sur un même processeur physique.

Nous croyons qu'un compilateur-paralléliseur pour traiter les cas que le programmeur ne veut pas (ou ne sait pas) écrire directement en parallèle est une aide efficace à la programmation des machines parallèles. Il resterait encore à construire un langage de haut niveau agréable à utiliser. High Performance Fortran (D.B. Loveman, 1993) est un bon point de départ. Le système des directives qui sont des conseils que l'utilisateur donne au compilateur est intéressant car un bon compilateur-paralléliseur peut éventuellement déroger aux directives lorsque des choix autres que ceux spécifiés permettent d'atteindre de meilleures performances. Un tel outil pourrait profiter du meilleur des deux mondes : la connaissance de l'algorithme par le programmeur et des méthodes automatique performantes, pour obtenir le programme parallèle le plus performant possible.

Jusqu'à présent la vue que nous venons de développer se borne à une architecture matérielle fixe. Cette vue est efficace pour un cas d'optimisation de système fini et en cours d'utilisation. Cependant dans un cadre de recherche, une vue plus large peut-être envisagée.

Imaginons être en possession d'un outil d'analyse d'algorithme qui estime les performances sur la base d'une analyse statique. Ce dernier fournit les performances maximales que l'application peut espérer atteindre sur une architecture matérielle idéalisée pour cette application. De là, un objectif est créé. Cet objectif est comparé à des résultats d'analyse dynamique basés sur des simulation détaillées. La comparaison produit des différences et à partir de ces différences, des études peuvent être réalisées pour connaître une manière de les supprimer. Deux solutions existent. La première réalise des optimisations au niveau du codage de l'application elle-même. Cette solution

produit des directives de modification aux programmeurs et aux outils de compilation. Ceci permet de modifier la manière dont l'application a été codée pour l'adapter plus finement à l'architecture cible. Cette solution fait appel à des méthodes d'optimisation logicielle telle que celles étudiées tout au long de ce mémoire.

En revanche, la seconde solution s'intéresse à la recherche architecturale afin d'améliorer la structure matérielle. À partir des directives d'optimisation, l'architecture matérielle est prise en cible. En fixant les performances de l'application, l'architecture peut être remodelée pour permettre de les atteindre. Les deux options décrites ci-dessus sont illustrées par la figure ci-dessous.

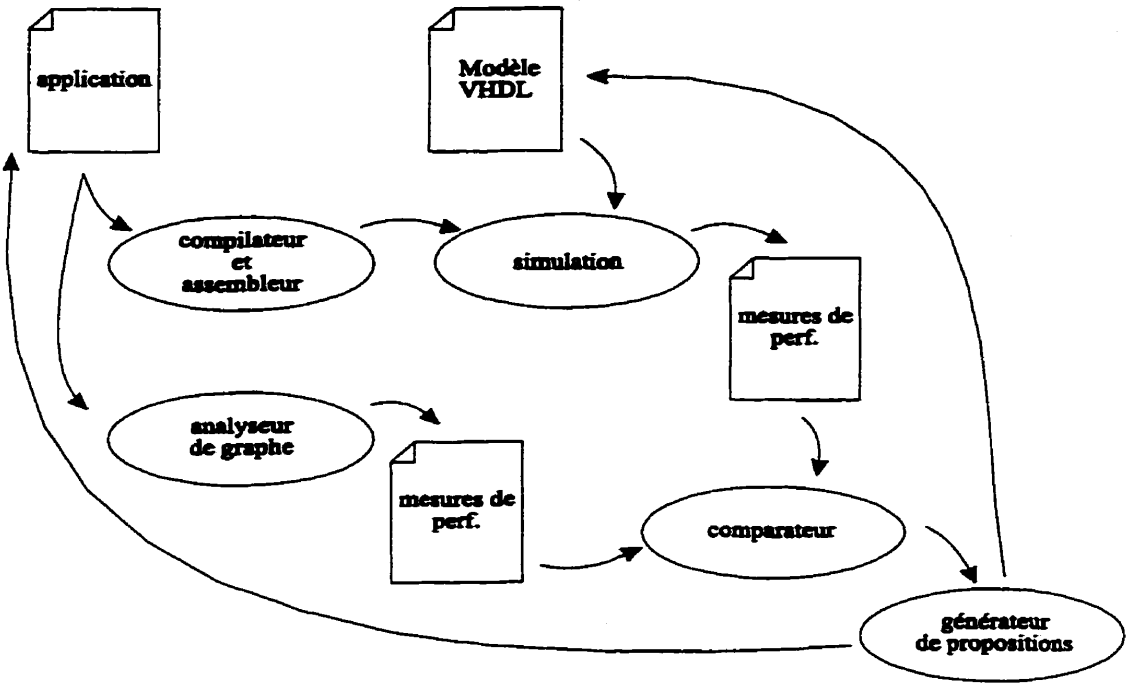


*Figure c.1 : La boucle générale d'optimisation*

Un tel outil est une utopie à l'heure actuelle, mais de nombreuses recherches vont vers cette voie qui permettrait une symbiose plus fine entre le matériel et le logiciel.

# RÉFÉRENCES

M. ACHIM, C. BONELLO et V. VAN DONGEN (Jul. 1997), C-Pulse – A Language for Parallel DSP Systems, 11th Annual International Symposium on High Performance Computing (HPCS'97).

JOHN R. ALLEN et KEN KENNEDY (June 1984), Automatic Loop Interchange, Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices, Vol.19, No.6.

JOHN R. ALLEN et KEN KENNEDY (Oct. 1987), Automatic Translation of FORTRAN Programs to Vector Form, ACM Transactions on Programming Languages and Systems, Vol.9, No.4, p.491-542.

UPTAL BANERJEE (1988), An Introduction to a Formal Theory of Dependence Analysis, The Journal of Supercomputing, Vol.2, p.133-149.

UPTAL BANERJEE (Feb. 1993), Automatic Program Parallelization, Proceeding of the IEEE, Vol.81, No.2.

UPTAL BANERJEE (1993), Loop Transformations for Restructuring Compilers, Kluwer Academic Publishers.

P. BANNERJEE et al. (Oct. 1995), The Paradigme Compiler for Distributed-memory Multicomputers, IEEE Computers, p.37-47.

N. BELANGER (1997), Référence sur le langage HPCP, Rapport technique interne, École Polytechnique de Montréal.

L. BOUGÉ et J. L. LEVAIRE (1992), Control structures for data-parallel SIMD-languages: Semantics and implementation, Semantics and implementation, Future Generation Computer Systems, Vol.8, p.363-378.

PIERRE BOULET, ALAIN DARTE, GEORGES-ANDRÉ SILBER et FRÉDÉRIC VIVIEN (June1997), Loop Pparallelization Algorithms: from Parallelism Extraction to Code Generation, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, Research Report N°97-11.

ALAIN DARTE (21 Avril 1993), Techniques de parallélisation automatique de nids de boucles, Thèse de doctorat présentée à l'École Normale Supérieure de Lyon, Laboratoire de l'Informatique du Parallélisme.

ALAIN DARTE et FRÉDÉRIC VIVIEN (Oct. 1996), Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs, In Proceedings of PACT'96, Boston, MA, IEEE Computer Society Press.

M.L. DOWLING (1990), Optimum Code Parallelisation Using Unimodular Transformations, Parallel Computing, Vol.16, p.155-171.

P. FEAUTRIER (1988), Array Expansion, ACM Int. Conf. on Supercomputing, p.429-441.

P. FEAUTRIER (Oct. 1992), Some Efficient Solutions to the Affine Scheduling Problem, Part I: One-Dimensional Time, Int. Journal Parallel Programming, Vol.21, No.5, p.313-348.

P. FEAUTRIER (Dec. 1992), Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multi-Dimensional Time, Int. Journal Parallel Programming, Vol.21, No.6, p.389-420.

P. FEAUTRIER (1996), Distribution automatique des données et des calculs, PRiSM, Université de Versailles.

M. J. FLYNN (1972), Some Computer Organisations and their Effectiveness, IEEE Trans. on Computers, Vol.21, No.9, p.948-960.

REINHARD V. HANXLEDEN et KEN KENNEDY (1992), Relaxing SIMD Control Flow Constraints using Loop Transformations, SIGPLAN Notices, Vol.27, No.7.

High Performance Fortran Forum (Nov.1994), High Performance Fortran Language Specification, ver.1.1.

KEE VAN REEUWIJK, WILL DENISSEN, HENK J. SIPS, et EDWIN M.R.M. PAALVAST (Sept. 1996), An Implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems, IEEE Transactions on Parallel and Distributed Systems, Vol.7 , No.9.

KATHLEEN KNOBE, JOAN D. LUKAS et GUY L. STEELE (1990), Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines, Journal of Parallel and Distributed Computing, Vol.8, p.102-118.

KATHLEEN KNOBE et VENKATARAMAN NATARAJAN (1993), Automatic Data Allocation to Minimize Communication on SIMD Machines, The Journal of Supercompiting, Vol.7, p.387-415.

DAVID J. KUCK, TOICHI MURAOKA et SHYH-CHING CHEN (Dec. 1972), On the Number of Operations Simultaneously Executable in Fortran-Like Programs and their Resulting Speedup, IEEE Trans. on Computers, C-12, p.1293-1310.

LESLIE LAMPORT (Feb. 1974), The Parallel Execution of DO Loop, Communication of the ACM, Vol.17, No.2, p.83-93.

Groupe du Pr. Lengauer, The Loopo Project, World Wide Web document, URL http://brahms.fmi.uni-passau.de/cl/loopo.

DAVID J. LILJA (Feb. 1994), Exploiting the Parallelism Available in Loops, COMPUTER, Vol.27.

AMY W. LIM et MONICA S. LAM (Jan. 1997), Maximizing Parallelism and Minimizing Synchronization with Affine Transforms, In Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

DAVID B. LOVEMAN (1993), High Performance Fortran, IEEE Parallel & Distributed Technology, Vol.1.

NELSON LUIZ PASSOS et EDWIN HSING-MEAN SHA (November 1996), Achieving Full Parallelism Using Multidimensional Retiming, IEEE Transaction on Parallel and Distributed Systems, Vol.7, No.11.

NELSON LUIZ PASSOS, EDWIN HSING-MEAN SHA et LIANG-FANG CHAO (February 1997), Multidimensional Interleaving for Synchronous Circuit Design Optimization, IEEE Transaction on Parallel and Distributed Systems, Vol.16, No.2.

R. H. PERROTT et T. F. LUNNEY (March 1991), A Syntax-Directed Integrated Programming Environment for Developing SIMD Supercomputer Software, Software Practice and Experience, Vol.21, No.3, p.269-286.

PAUL MARRIOTT, IVAN C. KRALJIC et YVON SAVARIA (Oct.1998), Parallel Ultra Large Scale Engine SIMD Architecture For Real-time Digital Signal Processing Applications, accepter pour publication lors d'ICCD 98, 4-7.

L. MALINIAK (Jan. 1994), Process Management will Tap EDA Productivity, Electronic Design, Vol.42, p.80-84.

Équipe PIPS, Pips (Interprocedural Parallelizer for Scientific Programs), World Wide Web document, URL http://www.cri.ensmp.fr/~pips.

Groupe PRiSM SCPDP, Systematic Construction of Parallel and Distributed Programs, World Wide Web document, URL http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html.

WILLIAM PUGH et al., The Omega Project, World Wide Web document, URL http://www.cs.umd.edu/projects/omega.

Stanford Compiler Group, Suif Compiler System, World Wide Web document, URL http://suif.stanford.edu/suif/suif.html.

MICHEL E. WOLFE et MONICA S. LAM (Oct. 1991), A Loop Transformation Theory and an Algorithm to Maximize Parallelism, IEEE Trans. Parallel Distributed Systems, Vol.2, No.4, p.452-471.

# ANNEXE : I

# Simulation based methodology for the development of complex VLSI systems

Nicolas Contandriopoulos ‡ Yves Blaquière † Yvon Savaria ‡

† Department of Computer Science
Microelectronics Group
Université du Québec à Montréal
Montréal, Québec, Canada

‡ Electrical and Computer
Engineering Department
École Polytechnique de Montréal
Montréal, Québec, Canada

October 8, 1998

*abstract*

***The design of large VLSI systems is systematically carried out by teams of designers. An environment for hardware simulation is proposed to manage this complexity. Combined with the proper design methodology, the environment helps to support coherence into team work and to increase engineering productivity. It is based on a tool that supports simulation of a large multiprocessor VLSI system. The same tool can be used by software and hardware designers. It supports simulation of several VLSI system configurations, from algorithmic application specifications to performance analysis and architectural exploration. Two version of this tool have been implemented. They based on HTML and Java, and they were heavily used as part of the PULSE project aiming at the development of the PULSE chip [1].***

**Key words:** processor design methodology, work team coherence, simulation, functionality verification, performance analysis.

## I. Introduction

Designing Very Large Scale Integrated (VLSI) systems is a complex endeavor, and advances in semiconductor technology result in an ever-increasing complexity. This complexity growth is combined with a continuous demand for shorter time to market and reduced design cost. That trend toward more complex systems forces software and hardware design teams to adopt a rigorous design methodology, in order to synchronize their efforts and to develop competitive products.

Design methodologies for large VLSI systems are a combination of a number of tasks. Each task is intrinsically complex and has its own sub-methodology, typically supported by various computer aided design (CAD) tools with their own modeling languages and data

formats, which require skilled users. For example, the increasing VLSI IC complexity has speed up the acceptance of Hardware Description Languages (HDL), which promote top-down design methodologies where abstraction levels are raised to facilitate design space exploration, in order to be as independent as possible on implementations details and technologies [2]. VHDL has been used as a language to support several tasks in Application Specific Integrated Circuit (ASIC) design, such as specification, design space exploration, description, simulation, synthesis, and functional verification. Each of these tasks has its own tool set to support the designer or group of designers, with their own input and output data formats.

To manage complexity, most VLSI systems development projects are performed by design teams which can be physically distributed in different organizations or companies. The organization of these teams can be site and/or project dependent. A usual project structure is a partition between hardware and software teams, each formed of sub-teams. For example, the software team can be composed of a software compiler team and an application-algorithm team. These multiple design teams execute their tasks concurrently and must be adequately managed [3].

There is, however, no unified standard methodology, language or tool for VLSI system design, where the system is composed of software and hardware components, designed as multi-team projects. Several Electronic Design Automation (EDA) environments can assist software and hardware designers in their design tasks [4]. These environments are designed to minimize human interactions and errors. They allow a more thourough design space exploration and therefore, increase the designer productivity and VLSI systems reliability [5]. Recently, the abstraction level of some EDA environments has been raised to target system synthesis methodologies starting from system specifications [6]. Although promising, these environments are targeted toward specific applications.

This paper presents a methodology used in the development of a complex multiprocessor system designed by a wide range of designer teams. The methodology is based on an unique tool that takes the form of an interface framework for simulation, capable of driving programming tools, hardware simulator and analysis tools. That interface creates a reference for all members of a development team and allows centralized version control. It supports multiple users and it is an adaptative interface with a client/server structure. This simulator was used at several levels of abstraction ranging from VHDL chip model validation to system level (mult-chip) application optimization.

The next section defines the interface framework objectives. The third section describes the user view of that framework, followed by a description of its internal structure. Two interfaces were implemented and statistical results obtained for the PULSE project are presented in section V.

## II. The Interface Framework

The main objective of our proposed framework is to offer a unified user view of VLSI system simulations embedding its software development tools. This framework abstracts

the complexity of commercial CAD tools and data formats, of in-house tools developed by various teams as part of the project and of the operating systems. This objective is achieved with an evolutive interface which can be easily adapted or scaled during the project development for different project team needs. These framework characteristics are detailed in the following paragraphs.

A large VLSI system development project can involve several teams, each composed of several members, with their own tasks, sub-project, commercial CAD tools, objectives, specifications and products to design and manufacture. A classical team partition could be:

- **ASIC team**, which designs one or many application specific integrated circuits, called *ASICs* here, with their tasks supported by specialized CAD tools, such as synthesis, simulation and layout tools.
- **System integration team**, which develops the *hardware system* based on PCBs, off-the shelf components ASICs and additional systems on boards.
- **Software team**, which creates the *programming tools*, like compilers and assemblers. These in-house tools directly depend on ICs and hardware systems, and are usually developed with commercial high level programming tools, such as a C compiler.
- **Application team**, which produces *application software, algorithms and data to be put in libraries*, such as communications, signal processing and mathematical functions. This team uses the programming tools developed by the software team.

Project partitioning among the teams leads to inevitable overlap for a number of tasks (Fig. 1). For example, some designers from the ASIC team and System integration team are put together to define the ASICs instruction sets or communication protocols between ASICs and system host. Some developers from the Software team and the Application team could define language extensions and compilation methods. Moreover, the Application team members could frequently propose useful architectural improvements to include in the ASICs for important classes of applications.

The proposed unified framework is a method to strengter links between teams, tools and products, in order to integrate abstracted models of each sub-project deliverables. These links can manage scripts and input/output files for each in-house and commercial tool, and secondly, can control each tool execution to carry out simulation. Designers are then freed from error-prone steps related to these tasks. The framework can also interpret, analyze and filter each output file to generate performance reports through a graphical or tabular user interface.

The interface framework is user-oriented and was developed to cut down the learning curve inherent to a complex simulation environment for VLSI systems development. A user can be a member of any team, and should be isolated as much as possible from the detailed information about in-house products, commercial tools and their operating systems. Moreover, each user seeks different information from the tools. The interface must therefore be sufficiently flexible to provide or filter simulation output data in order to support each category of users. Typical usage of these tools are:

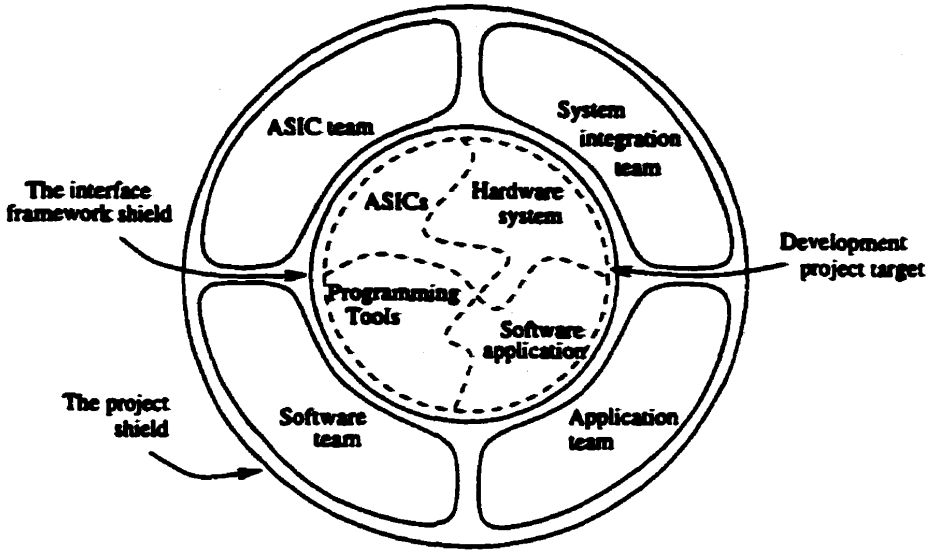


Figure 1: Possible structure of a large VLSI system project.

- **architectural exploration**, where the interface should provide information in order to explore several architectural tradeoffs at ASIC or system integration levels. This usage requires continuous adjustments of models and the interface should make it easy to add or modify ASIC or system models.
- **verification**, where detailed or very low level information, such as accurate signal or temporal values, must be supplied to designers during the development phase, specifically at ASIC and system integration levels. These information are used for functionality verification, validation and debugging of component or modules. For example, some internal signals or flags in a control unit must be available to verify several architectural functionalities.
- **application development and performance evaluation**, where the simulation environment must provide information about performances for different solutions at the architectural, software and application levels, and assistance in choosing the most interesting proposal. This information usually abstracts the detailed signals in the system under simulation, such as internal controller flags status and timing. For example, application team members prefer a higher level view where performance estimates or memory and register content in data-paths are made available for their application.

As soon as a first version of each deliverable is functional, the overall framework can be made available to users given then an abstracted and user friendly view of all deliverables, including those in which they were not involved. The framework must be organized in order

to be evolutive, where a new version can be easily and transparently made available to other teams. This hidden versioning control allows a unified and flattened vision of the updated system for each category of users. This unified vision allows quick interactions between the project's members.

## III. User View of the Simulation Framework Interface

The simulation framework should be usable by most teams members, whatever their special backgrounds (sec. II.). The simulation environment must be convivial, but powerful and flexible enough to support different specific requirements. For a user interacting with the framework, the work is divided into five interactive steps (Fig. 2) which are each coupled to respective window or window section (described in section IV.):

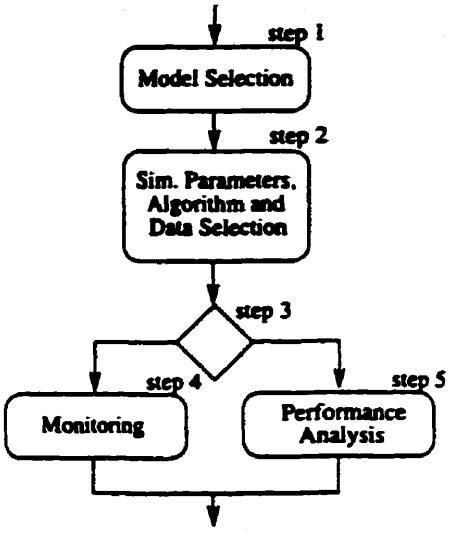


Figure 2: Steps to use the simulation environment.

The rest of this section briefly presents each of the five steps:

1. **Model selection**: In that step, the user selects ASICs and system hardware models. Several combinations of hardware/ASICs environment are proposed to the user according to the available project and their version.

2. **Simulation parameters, data and algorithm setup**: In that step, the user specifies simulation parameters such as system clock frequency, maximal simulation time, and defines data and application algorithms used for simulation. The data can be, for example, images stored in RAM, and application programs could be algorithms developed by the application team. Users can then apply different application algorithms on the hardware and ASIC models selected at the previous step.

3. **Analysis method selection:** In that step, the method used to analyze the simulation results is selected between monitoring (step 4) or performance analysis (step 5). These methods can be repeatedly applied, depending on the user's need.

4. **Monitoring:** In that step, one simulation is performed and its output results are monitored through a navigation tool. This method is used for verification and can present system status, memories and registers contents for each cycle. As in software debugging tools, the navigation tool allows scrolling through simulation cycles, step by step, or jumping using break points, based on instructions or simulation time. For example, application team users can observe functional unit contents through application algorithms simulation.

5. **Performance analysis:** In that step, simulation results are processed to extract performance data on hardware, software or applications. A library of built-in cost and analysis functions can be applied on restricted parts or the whole application, but specific processes can be programmed through a simple programming language [7].

   For example, performance analysis can be used to estimate the quality of a hardware/software partition. Some quality measures can be obtained on functional units such as the degree of parallelism between units, utilization rates or information transfer rate between units [8]. These measures use information in the controller, the input/output ports, and the communication between functional units, rather than the register or memory contents. This information can also be used to compare applications, or to extract effective performance metrics such as MIPS, MOPS, CPI.

The main advantage of this simulation environment approach is that analysis of simulation results and performance evaluation are closely related to real data, extracted from an efficient accurate and up-to-date version of the hardware model and its software support modules. Moreover, this framework automatically establishes several links between complex and powerful tools that could hardly be managed otherwise by each individual involved is project.

## IV. Internal Description of the Simulation Framework Interface

The simulation environment has a client/server interface and supports multi-user platforms. It could be used from remote physical sites across Internet and over the World Wide Web, WWW [9, 10]. These properties are very important for large evolutive VLSI system design projects, where several distant simulators could receive requests from users or from other tools. Its general structure is described in the following and the adopted implementation methods are also presented.

### A. *Components of the simulation framework*

The simulation framework is divided as a client part and a server part respectively running on client and server hosts (Fig. 3). The client part includes data, library and tools made available to the user. The server site manages all the simulation tools, libraries and data exchange between them, and centralizes the management of large data set and resources in the project. For example, version control and management of system models, simulation tools, programming tools and application libraries, each communicating together, are greatly facilitated if centralized on one site. Moreover, this centralized structure is very efficient since it ensures high bandwidth and low latency between tools. Another legitimate justification of a centralized site is security of confidential proprietary data and tools, commercial tools, and IP licensing. Only simulation and analysis results are made available to authorized clients. Notice that distributed tools and data could be made available on multiple sites as a natural extension of the proposed environment.

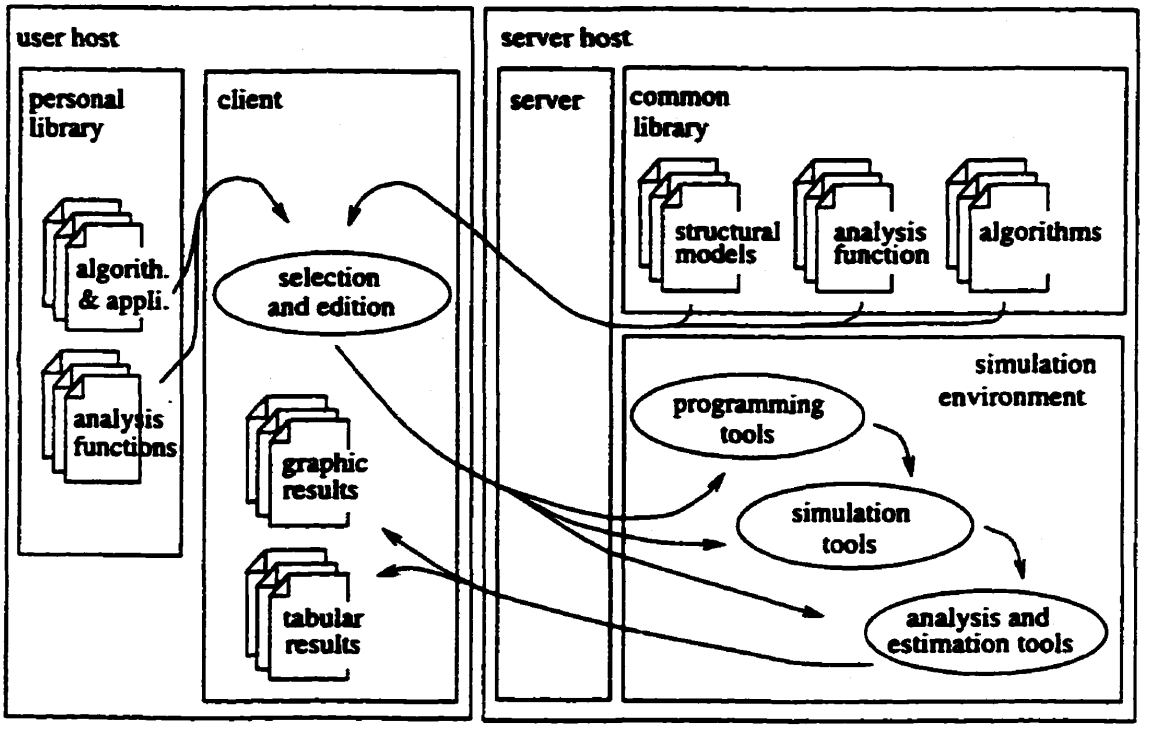


Figure 3: Information flow through the Framework Interface.

The part of the simulation framework residing on the server host site is built around a kernel that manages requests sent by clients (Fig. 3). As described in section III., a request can include information to set up a simulation and/or to specify an analysis method.

Each client's request is interpreted by the kernel, which invokes actions, or scripts, for each individual tool in the framework such as programming tools and simulation tools. The kernel can then return to its parent client, the simulations results or the result obtained from analysis and estimation tools, in table or graph form.

The tools in the framework can be divided into three families (Tab. 1). First, the client interface tools capture and parse user's inputs, display the state of available common libraries, send requests to the server's kernel and display results. These tools must be convivial, user friendly and must ensure a short learning curve. Two implementation methods have been experimented, as described in section IV.B.

Table 1: *Host site where tools are executed*

| Tool families | Client interface | Programming and Hardware tool script | Analysis and estimation |
|---|---|---|---|
| WWW based interface | server side | server side | server side |
| Java based interface | user side | server side | user side |

The second family in the kernel manages and generates scripts to drive programming and simulation tools, according to requests from the client. These scripts can individually drive the algorithm's compiler, assembler, code parallelizer and hardware simulator. Version control and verification are also performed on each common library, where for example, models selected by the client can be compiled and used for simulation. The data generated by these tools are generally large simulation results.

The simulation results are handled by the last family of tools. They parse, analyze and translate them in a readable form for the client user. Two methods are currently available: data monitoring or performance analysis (Fig. 2). In the performance analysis method, simulation results are processed from a library of mathematical and statistical functions. The data monitoring method offers a multitude of functions to manage tables and to display graphs. For example, users can easily set several performance metrics built from different equations.

This framework was completed with efficient communication mechanisms between client and server host sites through Internet. These mechanisms are based on services provided by a WEB site.

## B. *Implementation methods*

Two methods have been implemented to establish communications between client and server host sites. The first method is based on a traditional HTTP server (hypertext transfer protocol) [11] with an HTML browser client. The second is based on a Java client/server paradigm. The main difference is the site on which the application is executed (Tab. 1).

The first method is composed of an HTTP server on the server host site and uses the intrinsic interface of a WEB browser (Fig. 4). The HTTP server can interact with several

clients and carry out one simulation session per client. The server must then manage all the requests as a whole. The interface between external tools and the server consists of a set of CGI (Common Gateway Interface) scripts [12] written with the Perl language (Practical Extraction and Transfer Language) [7]. This language supports graphic interface, is convivial and compatible with a wide variety of operating systems. The interface between users and simulation environment kernel was developed with the HTML language (hypertext mark-up language) [13]. The HTTP server can be configured in intranet mode, giving limited access to selected users, for confidentiality purpose.

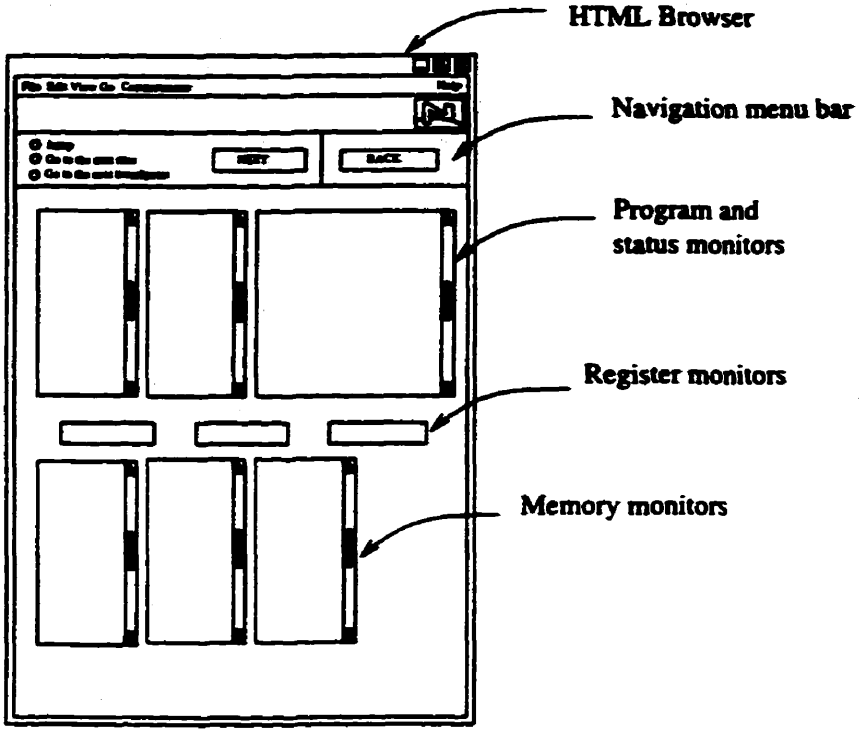


Figure 4: HTML based interface method.

The second method is based on a Java client/server approach instead of a HTML client/server. This solution is more flexible and powerful, but it is more complex to implement. For example, Java applications must create their own windows. Its main advantage is that the user interface management can be easily done on the client's host site, without connection to the server. Therefore, the selection step can be performed locally, and be transmitted to the server when ready. Moreover, simulation results and the family of tools for data monitoring and performance estimation, written in Java, can be completely transferred and used on the client's host site. This method increases the bandwidth available for examination of the results and decreases the load on the server.

# V. Statistical Results of Interface Implementation

The proposed simulation framework was developed for the PULSE (Parallel Ultra Large Scale Engine) project [1]. This project aimed at developing a linear SIMD multiprocessor with its support softwares and libraries. Different architectures to interconnect this scalable SIMD integrated circuit are supported and they can be interfaced with several system board hosts. Several hardware configuration models described in VHDL have been created, one for each system architecture. More that 25 persons were involved in this project from different universities or research groups. This groups was divided in several teams, such as described in figure 1. The final SIMD integrated circuit contains more than 900k transistor and two detailed system configuration have been design. The simulation framework was also used by team members involved with board development and/or algorithm evaluation.

The framework has been advantageously used by several team members. Statistics show that in a complete simulation cycle, 90-95% of the time is devoted to VHDL simulations and 5% is devoted to file management, algorithm compilation and to format data for the VHDL simulator. Only 0.1-5% of the time is used by the interface itself for management, interpretation and to display results. On a SUN UltraSparc1 workstation with 128 Mb of RAM, a throughput of 40 assembler instructions per second is being reached by the VHDL simulator running the actual synthesizable source model for which the chip was produced. For example, a simulation of 23000 clock cycles for one processor generates a 10 Mbytes ASCII results file. This file is transferred in a SQL database for efficient management and contains sufficient data to monitor a specific hardware configuration and to extract performance measures.

The first simulation framework development took four months person. At its peak utilization by PULSE team members, an average of 50 requests per week have been registered, 30% for analysis and performance estimations, and 70% for monitoring.

# VI. Conclusion

A new framework interface for simulation of a large VLSI system has been presented. This interface reduces considerably the learning time by abstracting the simulation flow and inherent complexity of a multi-team project requiring various programming, simulation and analysis tools. The cohesion between different specialized teams for design, verification, realization, and systems maintenance was ensured through a centralized framework, which eases version controls of proprietary tools, data and models. This unique framework interface supports the entire hardware simulation flow, from algorithmic application specification to analysis of simulation results. It is based on a client/server configuration where communications are controlled by HTTP and/or Java. Two implementations have been experimented in the PULSE project, one with HTML only and the other with a mix of Java and HTML. The framework has been used by approximately 25 team members, distributed over different physical sites.

# VII. Acknowledgment

This work was supported by a grant of the SYNERGIE program of the ministry of science and higher education of the province of Quebec as part of the PULSE project. This project was also sponsored by Miranda Technologies Inc., MiroTech Microsystems Inc., and Genesis.

# References


[1] P. Marriott, I. Kraljic, and Y. Savaria, "Parallel ultra large scale engine simd architecture for real-time digital signal processing applications," *Accepted for publication at ICCD 98*, Oct. 1998.

[2] M. L. White, "VHDL top-down design methodology and capability within boeing," *Northcon/94. Conference Record*, Oct. 1994.

[3] H. Jones, N. Giammarco, and D. Green, "A team oriented approach to department organisation in the microelectronics industry," *Proceedings of the Tenth Biennial University/Government/industry Microelectronics Symposium*, pp. 245–248, May 1993.

[4] L. Maliniak, "Process management will tap EDA productivity," *Electronic Design*, vol. 42, pp. 80–84, Jan. 1994.

[5] M. A. Hallwood, "VHDL and top-down methods prove successful in automotive ellectronic design," *IEE Colloquium on Computer Aided Engineering of Automotive Electronic*, pp. 1–10, Apr. 1994.

[6] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Prentice Hall, 1987.

[7] L. Wall, T. Chirstiansen, and R. L. Schwartz, *Programming Perl*. O'Reilly, 2nd edition ed., 1996.

[8] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw Hill, Inc., 1993.

[9] T. Berners-Lee, "The world-wide web," *CACM*, vol. 37, 1994.

[10] T. Berners-Lee and R. Caillian, "World wide web proposal for a hypertext project." URL: `http://www.w3.org/hypertext/WWW/Proposal.html`, 1990. (Page consulted by June 3rd).

[11] T. Berners-Lee, R. T. Fielding, and H. F. Nielsen, "Hypertext transfer protocol HTTP 1.0." URL: `http://www.w3.org/hypertext/WWW/Protocols/Overview.html`, 1997. (Page consulted by June 3rd).

[12] D. Robinson, "The WWW common gateway interface version 1.1." URL: `http://www.ast.cam.ac.uk/~drtr/cgi-spec.html`, 1997. (Page consulted by June 3rd).

[13] T. Berners-Lee and D. Connolly, "Hypertext markup language - 2.0. IETF, RCF: 1866,." URL: `http:/www.cis.ohio-state.edu/htbin/rfc/rfc1866.html`, 1997. (Page consulted by June 3rd).