Prototypage rapide d'applications parallèles de vision artificielle par squelettes fonctionnels

Dominique Ginhac

Numero d'Ordre : 1106
EDSPIC : 192

Université Blaise Pascal - Clermont ii

Ecole Doctorale

Sciences Pour l'Ingénieur de
Clermont-Ferrand

Formation Doctorale :

Electronique et systèmes

Thèse

présentée par

Dominique Ginhac

pour obtenir le grade de

Docteur d'Université

(SPÉCIALITÉ : Vision pour la Robotique)

Prototypage rapide d'applications parallèles de vision artificielle par squelettes fonctionnels

Soutenue publiquement le 25 janvier 1999 devant le jury :

Monsieur	B. Zavidovique	Président
Monsieur	M. Paindavoine	Rapporteur
Monsieur	Y. Sorel	Rapporteur
Monsieur	G. Cousineau	Examinateur
Monsieur	J.P. Dérutin	Examinateur
Monsieur	M. Dhome	Examinateur
Monsieur	J. Sérot	Examinateur

A mon grand père.

Remerciements

Les Travaux présentés dans cette thèse ont été effectués au sein du GRoupe Automatique : Vision et Robotique (GRAVIR) du LAboratoire des Sciences et Matériaux pour l'Electronique, et d'Automatique (LASMEA) de l'Université Blaise Pascal de Clermont-Ferrand, Unité Mixte de Recherche 6602 du Centre National de la Recherche Scientifique (CNRS).

Tout d'abord, je tiens à remercier Monsieur M. RICHETIN, Professeur à l'Université Blaise Pascal, directeur du LASMEA pour son accueil au sein du laboratoire.

J'exprime ma sincère reconnaissance à Monsieur B. ZAVIDOVIQUE, Professeur à l'Université Paris-Sud pour l'honneur qu'il m'a fait en acceptant de juger ce travail et de présider le jury de soutenance.

Je remercie Monsieur M. PAINDAVOINE, Professeur à l'Université de Bourgogne, ainsi que Monsieur Y. SOREL, Directeur de recherche à l'Inria pour l'intérêt qu'ils ont porté à ce travail en acceptant d'en être rapporteurs.

Je tiens également à remercier Monsieur G. COUSINEAU, Professeur à l'Ecole Normale Supérieure de Paris et Monsieur M. DHOME Directeur de recherche au CNRS pour avoir accepté de juger ce travail.

Enfin, merci du fond du coeur à Monsieur J.P. Dérutin et Monsieur J. Sérot respectivement Professeur et Maître de conférences au Centre Universitaire des Sciences et Techniques (C.U.S.T.) de Clermont-Ferrand pour l'attention constante avec laquelle ils ont suivi ces travaux et pour les qualités humaines dont ils ont su faire preuve durant cette belle aventure.

Enfin, je ne terminerai pas sans remercier toutes les personnes, qui ont contribué, à des titres divers, au bon déroulement de cette thèse et en particulier l'ensemble des membres permanents ou doctorants du laboratoire.

Remerciements Personnels

Après Les remerciements dits “officiels”, vient le tour des dédicaces personnelles envers ceux et celles de mon entourage qui ont contribué à cette inoubliable aventure qu'est une thèse.

Tout d'abord et de manière indiscutable, je tiens à exprimer ma reconnaissance envers mes parents qui ont toujours su m'apporter aide et soutien et surtout dans les moments difficiles voire stressants des phases de rédaction du mémoire de thèse et de préparation pour la soutenance. Merci Papa, Merci Maman d'avoir été là et de m'avoir poussé jusqu'à ce jour.

Puisque l'on parle de famille, comment oublier tous ceux et toutes celles qui ont fait le déplacement pour assister à ma soutenance, que ce soit mes deux grand mères fières de leur petit fils, mes oncles, mes tantes, mes cousins et cousines que je ne citerai pas personnellement tellement la liste est longue. Merci Marie-Claude d'avoir fait le voyage avec ma filleule Lise en espérant que, du haut de ses quelques mois de vie, elle prenne enfin exemple sur son parrain et réussisse de brillantes études. Merci à tous d'être venus ce jour là pour me soutenir et merci aussi pour tous les autres qui, travail oblige, n'ont pu se déplacer. Je sais qu'ils ont tous eu une pensée émue pour moi.

Enfin, n'oublions pas mon grand père qui nous a quittés en cours de voyage et qui était si fier de son petit fils ingénieur. Tu aurais été tellement heureux d'assister à cet événement.

Merci également à Steph qui partage ma vie et qui a su aussi bien vivre les moments heureux de cette thèse que m'encourager et me soutenir dans les moments difficiles. Merci Poupette d'avoir été là pour m'écouter, me parler, me rassurer, me faciliter la vie et tout simplement m'aimer.

Merci aux parents de Steph, mes futurs beaux parents, d'être venus assister à ma présentation, d'avoir stressé avec moi et d'être fiers de moi. Ne vous inquiétez pas, je prendrai bien soin de votre fille.

Passons maintenant aux membres du laboratoire dont certains méritent d'être cités ici. Commençons par Jean Pierre Dérutin alias “DéDé”, directeur de thèse, qui a su me faire découvrir le monde des architectures parallèles un beau jour de 1995 lors de ses cours de DEA. Merci de m'avoir fait confiance jusqu'au bout de cette belle aventure. Merci de m'avoir soumis tant d'idées géniales et révolutionnaires lors des nombreuses réunions tardives au bar du premier étage. Continuons par Jocelyn Sérot, défenseur des langages fonctionnels, qui a contribué plus que largement au développement de l'outil SKiPPER. Merci Jocelyn, sans ton aide précieuse, tes conseils avisés et

tes compétences techniques impressionnantes, ces travaux n'auraient pu être aussi bien menés. En espérant que nos routes ne vont pas se séparer à l'issue de ces quelques années qui resteront à jamais gravées dans ma mémoire.

Que dire sur les deux François ? Francois "Chucky" Berry, ami des bons et mauvais moments, toujours présent pour faire une quelconque bêtise que ce soit pour devenir root sur les HP, faire une page Web à notre collègue YoYo Bond ou acheter quelques bouteilles de vins pour un congrès de jeunes chercheurs. Merci François et désolé, je ne serais pas là pour ta consécration, mon devoir militaire m'appelant au 92^{eme} R.I. de Clermont-Ferrand[1]. Bisous à ta miss Lolo que j'espère revoir bientôt.

Merci à François "Marmoite" Marmoiton qui, malgré ce qu'il dit, fait une thèse géniale après tout de même des débuts difficiles et je ne sais combien de composants détruits.

Et puis pour finir, une énorme pensée pour Philippe "FiFi" Martinet, roi du repas au Casino et de la bonne bouteille, Roland Chapuis, premier béta testeur de l'outil SKiPPER, Alain Blanc, gourou PC et Microsoft[2], Michel Pizzocaro, ingénieur système du parc Unix qui distribue de l'espace disque au compte goutte, les secrétaires JJ, Pascale, Eliane et Christine qui nous supportent tous les jours, ainsi que tous les autres doctorants et permanents que je ne peux citer individuellement, clients assidus du bar du premier étage.

Merci à tous.

[1]Je ne remercie pas le Ministère de la Défense en raison de leur insistance à vouloir m'empécher de terminer ma thèse.

[2]Je ne remercie pas Bill G., PDG de Gigasoft qui m'a obligé à payer une license Windaube 98 lors de l'achat de mon PC.

Résumé

Les Travaux présentés dans ce mémoire s'inscrivent dans la problématique dite d'adéquation algorithme architecture. Ils concernent la conception et le développement d'outils logiciels permettant de faire du prototypage rapide d'applications parallèles de vision artificielle sur des architectures de type MIMD à mémoire distribuée. De tels outils ont pour objectif de faciliter l'évaluation rapide d'un ensemble de solutions vis à vis d'un problème donné en diminuant de manière drastique les temps de cycle conception-implantation-validation des applications.

L'outil SKiPPER développé dans le cadre de ces travaux est basé sur le concept des squelettes de parallélisation. Ceux-ci représentent des constructeurs génériques de haut niveau encapsulant des formes communes de parallélisme tout en dissimulant les détails relatifs à l'exploitation de ce parallélisme sur la plate-forme cible. Au niveau langage, la spécification des squelettes est réalisée au sein du langage fonctionnel Caml sous la forme de fonctions d'ordre supérieur. Ainsi, la spécification d'une application est un programme purement fonctionnel dans lequel l'expression du parallélisme est limitée au choix et à l'instanciation des squelettes choisis dans une base pré-définie.

L'environnement de développement SKiPPER est organisé autour de trois modules réalisant respectivement l'expansion du code fonctionnel en un graphe flot de données (outil Dromadaire), le placement-ordonnancement de ce graphe sur l'architecture matérielle (outil SynDEx développé à l'INRIA) et la génération de code cible final pour l'architecture cible (la machine Transvision du LASMEA dans notre cas).

L'applicabilité des concepts mis en œuvre dans SKiPPER et des outils développés conjointement est démontrée également dans les travaux présentés dans ce mémoire. Diverses applications de complexité réaliste (étiquetage en composantes connexes, détection et suivi de signalisation horizontale autoroutière) ont été parallélisées automatiquement par l'environnement SKiPPER validant ainsi l'objectif initial de prototypage rapide d'applications parallèles de vision artificielle à fortes contraintes temporelles sur architecture dédiée.

Mots clés : traitement d'images, parallélisme, squelettes de parallélisation, langages fonctionnels, SKiPPER.

Abstract

We present SKiPPER, a software dedicated to the fast prototyping of vision algorithms on MIMD/DM platforms.

This software is based upon the concept of algorithmic skeletons, i.e. higher order program constructs encapsulating recurring forms of parallel computations and hiding their low-level implementation details. Examples of such skeletons in low- to mid-level image processing include such as geometric decompositions, data or task farming. Each skeleton is given an architecture-independant functional (but executable) specification, a portable implementation as a process template and an analytic performance model.

The source program is a purely functional specification of the algorithm in which all parallelism is made explicit be means of composing instances of selected skeletons, each instance taking as parameters user-specific sequential functions written in C. This specification is turned into a process graph in which nodes correspond to sequential functions and/or skeleton control processes and edges to communications. This graph is mapped onto the actual physical topology using a third-party CAD software (SynDEx). The result is a dead-lock free, optimized (but still portable) distributed executive which can be run straightly on the target platform. The initial specification, written in ML language, can also be executed on any sequential platform to check the correctness of the parallel algorithm and to predict performances. In that case, the applicative semantics of skeletons guarantees the equivalence between sequential and parallel results.

The applicability of SkiPPER concepts and tools has been assessed by parallelizing several realistic real-time vision applications both on a multi-DSP platform and a network of workstations (connected component labeling, road tracking algorithm based upon marking detection). This experiment showed a dramatic reduction in development times (hence the term fast prototyping) with measured performances staying on the on the par with those obtained with hand-crafted parallel versions.

Keywords : image processing, parallelism, algorithmic skeletons, functional languages, SKiPPER.

Contents

List of Figures

List of Tables

Introduction

Lorsque l'on demande à une personne de décrire ce qu'elle voit dans son environnement proche, elle n'a aucune difficulté à énumérer de manière quasi instantanée les objets présents dans son champ visuel. De même, lorsque l'on lui demande de saisir tel ou tel objet et de le déplacer, cette même personne n'éprouve pas plus de difficultés à réaliser cette nouvelle action. Et pourtant, ces actes si banals et triviaux à l'échelle humaine dissimulent en réalité la mise en œuvre d'un ensemble de processus complexes de traitement de l'information visuelle visant à détecter, à identifier et enfin à déclencher une action musculaire pour se mouvoir et saisir un objet. Le plus frappant, voire déconcertant, est sans aucun doute la rapidité de ces traitements qui ne durent au maximum que 150 ms. En effet, durant ce court laps de temps, le système de perception humain effectue une analyse précise de l'information présente au niveau de la rétine sous la forme d'une collection de points (environ un million !), chacun étant caractérisé par ses propres indications de lumière, de couleur, de texture, etc. Face à cette multitude de données, le cerveau est ainsi capable d'extraire très rapidement l'information pertinente donnant en final le résultat approprié.

La Vision Artificielle, dont les premières bases théoriques remontent aux années 1960, a parmi ses objectifs la volonté de doter une machine d'un sens de perception visuelle similaire à celui de l'être humain. Le couple "œil-cerveau" est alors remplacé par le couple "caméra vidéo-ordinateur" capable théoriquement d'analyser des images naturelles. Un tel sens de perception permet ainsi à cette machine d'évoluer de la manière la plus autonome possible dans son environnement. Les robots d'intervention en milieu hostile ou la conduite automatique de véhicule automobile sont des exemples concrets d'actualité.

Le besoin toujours croissant en puissance de calcul des algorithmes de vision artificielle invoqués par une chaîne de perception visuelle (processus de pré-traitement, de segmentation d'images, d'identification, de décision, etc.) n'a jamais cessé depuis les premiers travaux relatifs au domaine. Chaque

nouvelle application semble nécessiter une puissance supérieure à celle déjà mise en œuvre lors des précédentes réalisations. Si les limites physiques et technologiques des composants ne sont pas encore atteintes — permettant ainsi d'accroître régulièrement les puissances des machines —, il n'en demeure pas moins que la seule croissance en puissance des processeurs séquentiels semble insuffisante pour satisfaire les besoins de telles applications.

Face à ce besoin de puissance, l'idée de multiplier les unités de traitement, autorisant ainsi l'exécution simultanée de plusieurs opérations de calcul, est une solution communément avancée pour satisfaire les contraintes temporelles des applications. Cette discipline, nommée *parallélisme* est un domaine de recherche à part entière. Elle ne se limite pas à la conception et à la réalisation d'architectures parallèles mais concerne également le développement et l'implantation d'applications ainsi que la mise en œuvre de méthodologies de programmation de telles machines.

C'est ce dernier point qui fait l'objet des travaux présentés dans ce mémoire. En effet, il apparaît de manière évidente que la programmation des architectures demeure dans la plupart des cas un exercice délicat réservé à un public de spécialistes. Ceci est d'autant plus vrai que dans le domaine de la vision artificielle, les architectures parallèles sont souvent dédiées et ne bénéficient pas du support logiciel traditionnellement offert pour des machines généralistes (stations de travail par exemple). La résultante est des temps de conception-implantation importants d'où une incapacité à évaluer *rapidement* un large spectre de solutions vis à vis d'un problème donné. Or, dans le contexte applicatif visé, seule une approche expérimentale fondée sur la validation *in situ* des solutions permet de juger de leur validité fonctionnelle et opérationnelle. La seule solution passe donc par une diminution drastique des temps de développement ce qui suppose l'existence de formalismes de haut niveau et des outils d'aide à la parallélisation. Dans ce contexte, notre approche vise donc le *prototypage rapide d'applications de vision artificielle* et a pour objectif final la mise en œuvre d'un outil d'aide à la parallélisation de telles applications.

Le premier chapitre de ce mémoire est entièrement consacré à la programmation d'applications de traitement d'images temps réel sur architectures dédiées à la vision. Il est organisé autour de deux thèmes principaux permettant de fixer précisément le cadre de notre étude. Premièrement, le recours à des exemples significatifs d'applications illustre l'aspect algorithmique des traitements invoqués dans la chaîne de perception visuelle et permet de dresser un large éventail des applications potentielles. Deuxièmement, les difficultés sous-jacentes au développement parallèle de ces applications sont décrites montrant d'une part la nécessité de méthodologies spécifiquement

dédiées au traitement d'images et d'autre part l'inadéquation des outils existants à notre problématique de prototypage rapide. Ce premier chapitre se termine par la définition des propriétés essentielles que doit respecter un outil d'aide à la parallélisation d'applications de vision artificielle.

Le deuxième chapitre présente les notions et principes de constructeurs parallèles génériques nommés *squelettes de parallélisation* lesquels apparaissent comme une solution intéressante à notre problématique de développement parallèle d'applications de vision artificielle. Après un état de l'art des diverses méthodologies fondées sur les squelettes, ce chapitre définit en particulier une collection restreinte de squelettes dédiés au traitement d'images à partir d'une analyse fine d'applications parallèles implantées manuellement.

Le troisième chapitre est consacré à la programmation fonctionnelle. Il montre les facilités offertes par les langages fonctionnels pour exprimer la notion de squelettes. Ce chapitre introduit également le principe de séparation des définitions fonctionnelle et opérationnelle des squelettes, satisfaisant ainsi les possibilités de prototypage rapide et de portabilité des applications.

Le quatrième chapitre présente l'outil de développement SKiPPER (SKeletal Parallel Programming EnviRonment), qui a fait l'objet des travaux décrits dans ce mémoire. Dans un premier temps, sont décrits les différents outils permettant de passer d'une spécification fonctionnelle des algorithmes à un code cible parallèle exécutable sur la machine. Dans un deuxième temps, une étude précise des implantations de chaque squelette permet de définir, pour chacun d'entre eux, un modèle analytique de performance validant ainsi la notion de prédictibilité des performances.

Le cinquième et dernier chapitre a pour objectif de valider l'ensemble des principes de SKiPPER. Pour cela, cinq applications de traitement d'images sont successivement étudiées et implantées automatiquement. Les trois premières sont relativement simples et n'utilisent qu'un seul des squelettes définis. L'implantation de telles applications a pour objectif de mesurer les écarts entre les performances prédites par les modèles analytiques et celles effectivement mesurées lors de l'exécution. Les deux dernières applications sont, quant à elles, de complexité beaucoup plus réaliste permettant ainsi de valider complètement notre problématique de prototypage rapide de vision artificielle sur architecture multi-processeurs.

Enfin, une conclusion permet de faire un bilan des travaux décrits dans ce mémoire et de présenter brièvement les perspectives de recherche à explorer pour améliorer l'outil SKiPPER.

Chapter 1

Le Contexte

1.1 Introduction

Dans un futur relativement proche, on peut s'attendre à ce que la majorité des véhicules soit équipée de systèmes d'aide à la conduite sophistiqués. A partir d'images délivrées par un capteur CCD, un calculateur embarqué sera capable de fournir en temps réel la position du véhicule et les possibilités d'évolution en fonction de son environnement immédiat (par exemple les autres véhicules). Ces résultats conditionneront éventuellement le déclenchement d'actionneurs agissant sur l'accélérateur, le frein ou la direction du véhicule réduisant ainsi les situations dangereuses, sources de nombreux accidents.

Un tel scénario, inimaginable encore il y a quelques années, est en passe de devenir réalité et suscite de vastes programmes de recherches de par le monde.

Une application temps réel de traitements d'images (TI) peut dans ce contexte être vue comme un système informatique — regroupant à la fois l'aspect applicatif et l'aspect matériel — recevant en entrée des données issues de capteurs, effectuant une série de traitements sur ces données et produisant en sortie une commande modifiant l'environnement dans lequel il évolue.

Pour le programmeur d'applications, l'objectif est de développer des programmes conformes aux spécifications et respectant les contraintes temporelles désirées.

Voici donc en quelques phrases les points essentiels que nous allons présenter tout au long de ce chapitre. Successivement, nous aborderons les thèmes suivants :

- dans un premier temps, nous évoquerons la large variété des applications de **TI** temps réel manipulant des types et des quantités de données radicalement différentes,
- dans un deuxième temps, nous nous consacrerons sans être exhaustif aux **architectures parallèles** dédiées au TI qui constituent une solution intéressante pour faire face aux besoins de puissance des applications de TI temps réel,
- dans un troisième temps, sera décrite la **complexité de programmation** de ces architectures. En particulier, on mettra en évidence les difficultés à concilier prototypage rapide et efficacité des applications de TI,
- dans un quatrième temps, nous évoquerons un ensemble de **méthodologies de développement** qui tentent de répondre à nos objectifs de prototypage rapide en fournissant des outils dits d'"adéquation algorithme architecture", facilitant ainsi la parallélisation des applications de TI,
- enfin, nous définirons les principales caractéristiques auxquelles devra répondre **l'outil d'aide à la parallélisation** décrit dans ce mémoire afin de pouvoir satisfaire les exigences de prototypage rapide de TI sur architecture dédiée.

1.2 Le Traitement d'Images Temps Réel

1.2.1 Quelques applications de Traitement d'Images

1.2.1.1 Premier exemple

Le premier des exemples présentés ici décrit une application complète de traitement et d'analyse d'images mettant en œuvre un certain nombre d'étapes séquentielles, chacune travaillant à partir des résultats délivrés par la précédente. Cette chaîne de traitement nommée **extraction de groupements perceptifs** est décrite complètement dans [Leg95]. Elle effectue un regroupement des segments caractéristiques de l'image en un ensemble de primitives (segments colinéaires, orthogonaux, symétriques, etc.) qui constituent un premier niveau d'interprétation de la scène observée.

La figure 1.1 décrit sommairement les différentes étapes de cette chaîne de traitement. On y trouve notamment :

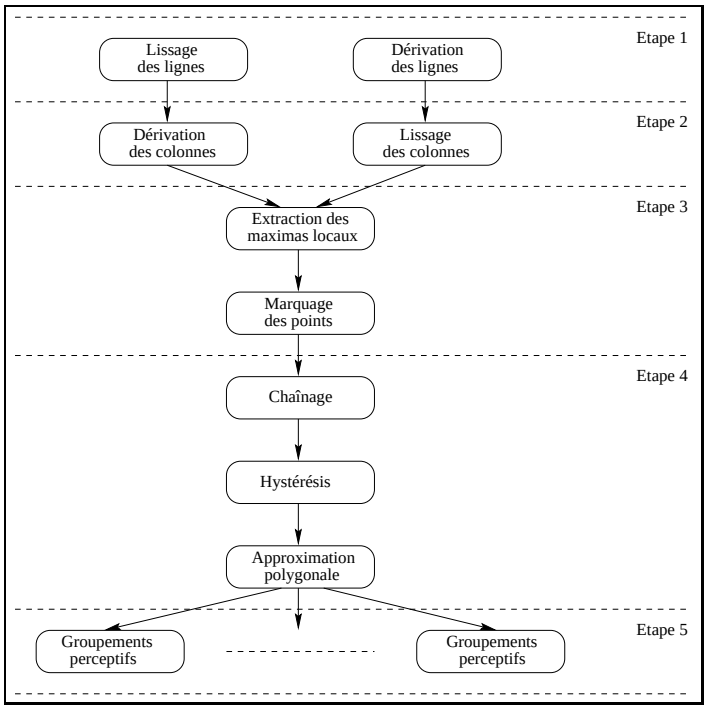


Figure 1.1: Chaîne de groupements perceptifs

- le calcul des gradients directionnels G_x et G_y de l'image source par le filtrage de Canny-Deriche[Can86][Der90] décomposé en deux étapes principales effectuant le lissage (respectivement la dérivation) des lignes de l'image et la dérivation (respectivement le lissage) des colonnes de l'image résultat intermédiaire.

- l'extraction des maxima locaux et la première étape du seuillage par hystérésis permettent l'extraction des points de contour effectifs (dont la norme du gradient est supérieure à un seuil haut) et potentiels (dont la norme du gradient est comprise entre un seuil bas et un seuil haut) de l'image.

- l'étape de chaînage consiste à lier les points contour pour former des chaînes de points et non plus des points isolés. Celle-ci permet de passer d'une représentation sous la forme de matrice image à une représentation sous la forme de liste de points connexes, diminuant ainsi le volume des données à traiter. Elle est également associée avec la deuxième phase du seuillage par hystérésis qui regroupe les points potentiels connexes et les points effectifs et élimine les chaînes de points correspondant à du bruit (la norme du gradient de chaque point de la

chaîne est comprise entre les deux seuils).

- l'approximation polygonale des chaînes de points contour réduit encore le volume des données en représentant chaque chaîne sous la forme d'une séquence de segments de droite. La méthode employée consiste à itérer un processus récursif de division de courbe jusqu'à ce que l'écart entre la courbe et les segments l'approximant soit inférieur à un seuil fixé. Le codage des segments est réalisé en ne retenant que les coordonnées des deux points extrémités.

- enfin, la dernière étape d'extraction de groupements perceptifs regroupe les segments satisfaisant certaines contraintes géométriques en évaluant toute combinaison de segments pris deux à deux. La segmentation n'étant pas parfaite, les relations géométriques entre segments sont très rarement satisfaites. Ceci implique que les algorithmes mis en œuvre doivent tenir compte de ces imprécisions et détecter des segments "presque" colinéaires, "presque" symétriques, etc.

L'illustration de cette chaîne complète de traitement est réalisée sur la figure 1.2 montrant les résultats de chaque étape sur une scène réelle.

La présentation succincte de cette application permet de mettre en évidence certaines caractéristiques du TI.

Il apparaît clairement que les volumes de données manipulées par chaque étape de la chaîne de traitement varient de manière significative. Les premières étapes dites de *bas niveau* opèrent à partir d'images numériques en niveaux de gris (matrices de taille $512*512*8$ bits) et fournissent en sortie une image résultat (image lissée, image de gradient, etc.). Les opérateurs mis en œuvre sont très souvent des opérateurs locaux travaillant au niveau du pixel et/ou d'un voisinage de pixels. Les traitements à effectuer sont indépendants de la valeur des données. Le nombre d'opérations élémentaires est fonction de la taille de l'image et donc le temps d'exécution de ces traitements est proportionnel au volume des données.

Les étapes intermédiaires et finales (respectivement nommées *moyen* et *haut* niveau) sont plus difficilement classifiables puisque la nature des traitements ainsi que le type des données manipulées sont très fortement liés à l'application. Les opérateurs mis en jeu extraient des indices visuels (sous la forme de listes de points, de listes de segments par exemple) à partir des images résultats issues des premières étapes. Ainsi, la nature locale des premiers traitements ne se retrouve pas ici. Il est impossible de prévoir le nombre et la taille des données à traiter. Par exemple, la phase d'approximation polygonale d'une chaîne de points en un ensemble de segments dépend à la fois du

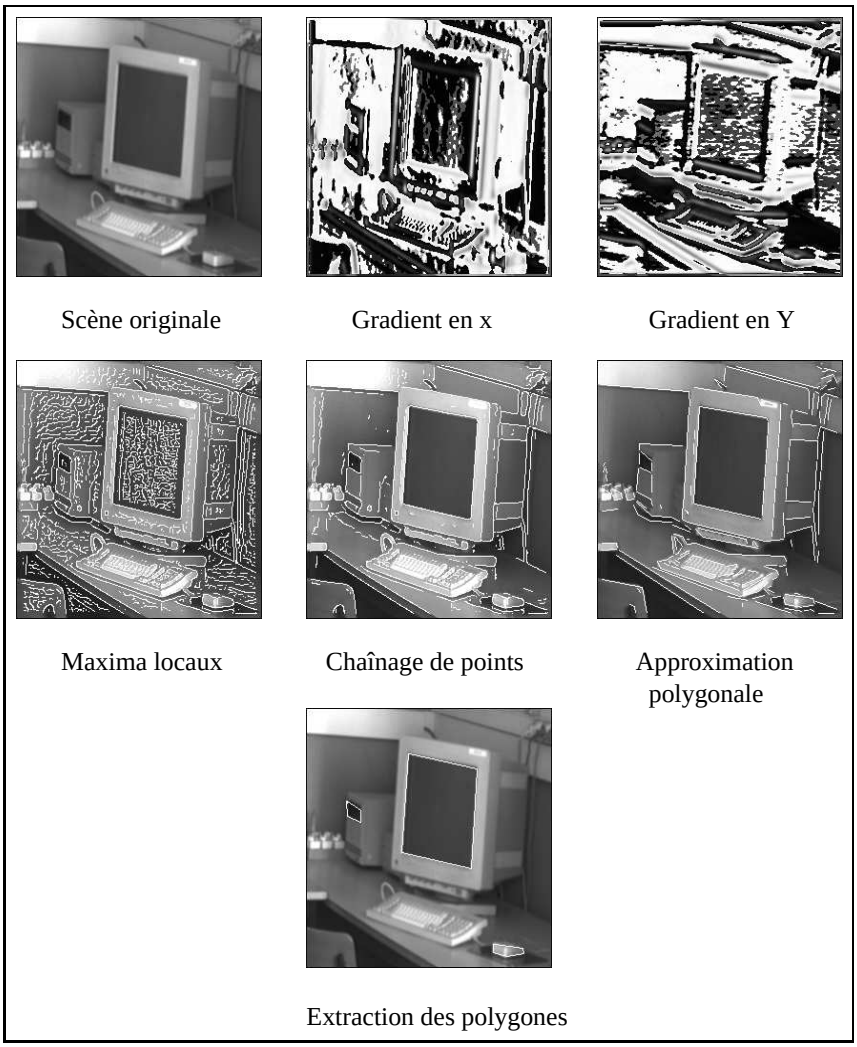


Figure 1.2: Résultats de la chaîne de groupements perceptifs

nombre de points et de la forme géométrique codée par la chaîne c'est-à-dire des valeurs successives des points. De fait, ces traitements sont difficilement prévisibles du point de vue temporel et au mieux, une borne supérieure du temps d'exécution peut être estimée.

1.2.1.2 Deuxième exemple

Cette seconde application aborde le problème des approches multi-fenêtres d'intérêt dont le nombre, la position dans l'image issue du capteur et la taille sont variables dans le temps et remis à jour à chaque itération. Ce type d'application a pour objectif premier d'accélérer les temps de calcul par une

réduction initiale du volume de données à traiter[1].

De plus, cette classe de chaîne de traitements faisant appel à une technique de *prédiction-vérification* manipule non plus des images prises séparément mais un flot continu d'images : les données traitées à l'itération i sont conditionnées par les résultats provenant de l'itération $i-1$.

Un exemple de **détection et suivi de véhicules routiers** repérés par un système de trois amers[2][MCMD98] illustrant parfaitement cette approche est décrit sur la figure 1.3.

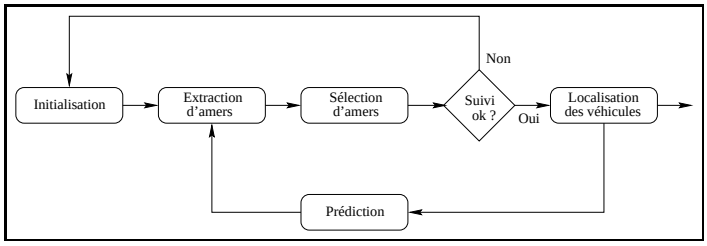


Figure 1.3: Détection et suivi de véhicules routiers

Afin d'expliquer le fonctionnement de cette chaîne de traitement, considérons le cas idéal d'exécution (cas d'une détection du véhicule sans perte de suivi). L'application comporte alors quatre étapes successives que nous allons détailler :

- une phase d'extraction des points lumineux dans les fenêtres d'intérêt permet de détecter les amers. Le recours à un seuillage adaptatif basé sur l'histogramme des niveaux de gris permet de s'affranchir des variations lumineuses de l'image.

- une phase de sélection des amers détectés extrait parmi les amers candidats ceux correspondant à une détection effective de véhicules. En effet, il est possible soit d'obtenir plus de trois amers dans les zones d'intérêt (amers parasites par exemple), soit moins de trois (amers occultés par exemple). Cette étape effectue une mise en correspondance des amers détectés avec ceux prédits ce qui permet d'éliminer les phénomènes d'occultation et de parasitage. La figure 1.4 présente les résultats de détection à la fois dans un cas idéal à trois amers et dans un cas parasité avec cinq amers.

[1]On ne manipule plus des images dans leur intégralité mais des zones restreintes contenant des informations jugées pertinentes.

[2]En navigation maritime, un amer est défini comme un objet caractéristique immobile et aisément repérable sur la côte[Gle95]. Dans le cadre de l'application, les amers sont trois feux placés sur les véhicules.

- une phase de localisation du véhicule permet de remonter à l'information de distance entre la caméra et le véhicule suivi en ayant recours à un modèle du véhicule,

- une phase de prédiction permet de définir la position et la taille des fenêtres d'intérêt qui seront traitées à l'itération suivante.

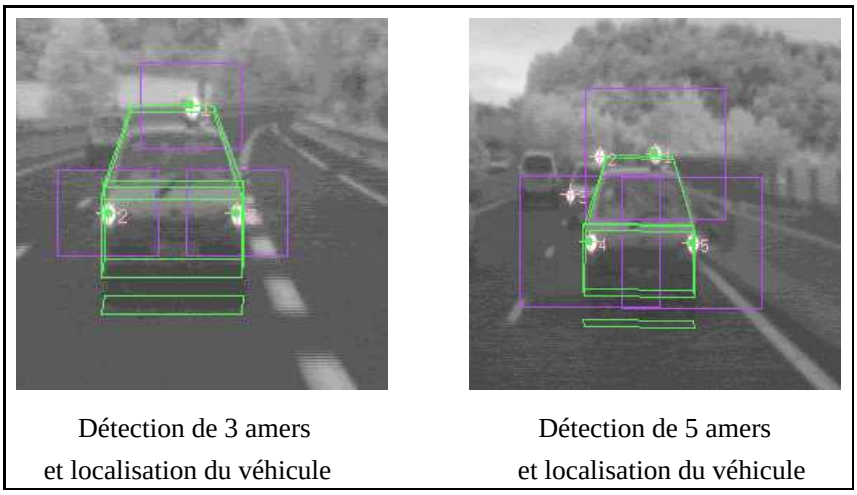


Figure 1.4: Résultats de détection de véhicules

Le fonctionnement décrit dans les quatre étapes correspond au régime permanent d'exécution et n'est pas toujours reproductible. En effet, les phases d'initialisation au démarrage de l'application ainsi que les ré-initialisations dans le cas de perte de suivi sont des étapes indispensables qui sont dues à la notion de rebouclage du schéma. Dans les deux cas, la stratégie de recherche des amers décrite auparavant peut être appliquée mais celle-ci doit opérer non plus sur des fenêtres d'intérêt mais sur l'image globale puisque aucune connaissance *a priori* ne peut indiquer la position des amers. La conséquence directe d'une telle approche est un rallongement systématique du temps de traitement d'une itération. La phase de suivi ne sera réellement lancée que lorsqu'un véhicule repéré par trois amers distincts aura été effectivement détecté. En cas de perte de suivi, le recours à des phases de ré-initialisation du système peut être très préjudiciable car le comportement temps réel de l'application n'est alors plus garanti.

De plus, considérons maintenant la même chaîne de traitement non plus dans un contexte mono-cible (détection et suivi d'un seul véhicule) mais dans une approche multi-cible (détection et suivi d'un nombre variable de véhicules). Dans ce cas, il est impossible de prédire à un instant donné

le nombre de véhicules présents dans l'image. Cela impose non seulement d'effectuer un suivi des véhicules déjà détectés durant les itérations précédentes mais également de mettre en place une stratégie opérant sur l'image globale dont le but est de détecter d'éventuels nouveaux véhicules apparaissant dans le champ de la caméra. Ceci accroît la complexité de la chaîne de traitement puisque plusieurs opérateurs vont traiter des données à des **cadences différentes** (suivi dans les zones d'intérêt à la cadence vidéo et détection de nouveaux véhicules par scrutation de l'image complète toutes les 10 images par exemple). Du point de vue temporel, cela implique inévitablement l'impossibilité de prédire la durée des itérations. D'un point de vue algorithmique, cela suppose un entrelacement des tâches de durées et de fréquence d'activation différentes.

1.2.1.3 Troisième exemple

Le troisième et dernier exemple présente une chaîne algorithmique dans laquelle une approche dynamique des traitements invoqués est mise en œuvre.

Cette approche utilise les caractéristiques présentes dans les données pour adapter au mieux la chaîne de TI utilisée. Suivant les données et les événements présents dans celles-ci, on assiste à des changements de *contexte algorithmique* privilégiant tel ou tel algorithme[3]. Le champ d'investigation de ces approches dynamiques est vaste : on peut citer en exemple une application de **détection et de localisation de plaques minéralogiques**[Bel96] en vue d'effectuer un péage autoroutier entièrement automatisé.

Le scénario d'une telle application se décompose en un ensemble d'algorithmes activés selon la présence ou non d'un véhicule à diverses distances de la caméra (cf. figure 1.5) :

- à l'initialisation, le système est en attente d'un événement lui signalant la présence d'un obstacle sur la route (algorithme de détection de mouvement dans l'image),
- dès la présence de mouvement, le suivi temporel de l'obstacle susceptible d'être un véhicule est activé (algorithme de type prédiction-vérification),
- à $d_1 = 20m$, une première reconnaissance du véhicule est effectuée (algorithme de reconnaissance de formes),

[3]Cette approche dynamique peut s'apparenter à de la vision active[AWB90] dans laquelle les contraintes pour la résolution du problème sont simplifiées parce que l'observateur peut changer d'état d'une manière active.

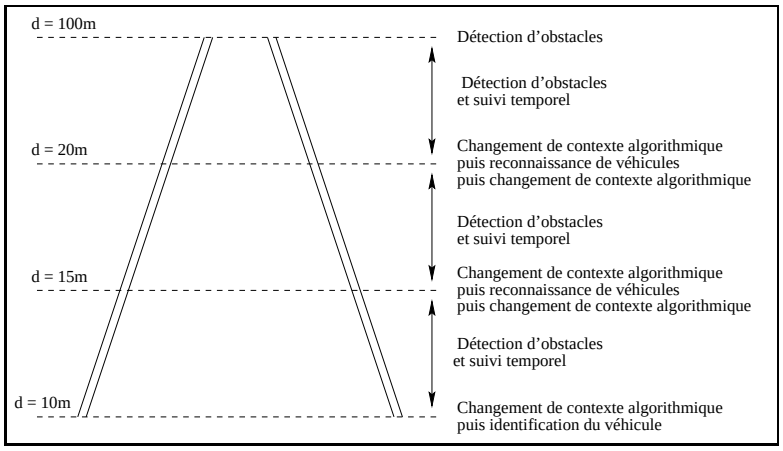


Figure 1.5: Scénario de détection des plaques minéralogiques

- par la suite, le suivi du véhicule est réactivé,
- à $d_2 = 15m$, la reconnaissance du véhicule est confirmée,
- ensuite, le suivi du véhicule est réactivé,
- à $d_3 = 10m$, la plaque minéralogique est localisée. Cette localisation est réalisé après un changement de contexte capteur (commutation sur une deuxième caméra à focale plus élevée).

La figure 1.6 montre des résultats sur scènes routières réelles de l'ensemble de ces étapes.

Par rapport à l'application précédente dans laquelle on applique toujours le même algorithme sur des données de taille différente (zones d'intérêt ou image), l'application présentée ici met en œuvre un ensemble d'algorithmes différents dont l'activation dépend des données présentes dans l'image. La mise en œuvre du scénario précédemment défini implique le développement d'un module supervisant l'ensemble des algorithmes de la chaîne de traitements. Celui-ci doit contrôler l'enchaînement des tâches selon le scénario préétabli et en fonction des événements présents dans les séquences d'images. Le module superviseur doit en effet réaliser — de manière cohérente et efficace — des changements de contexte algorithmique et/ou capteur impliquant des distributions dynamiques de traitement et/ou de données.

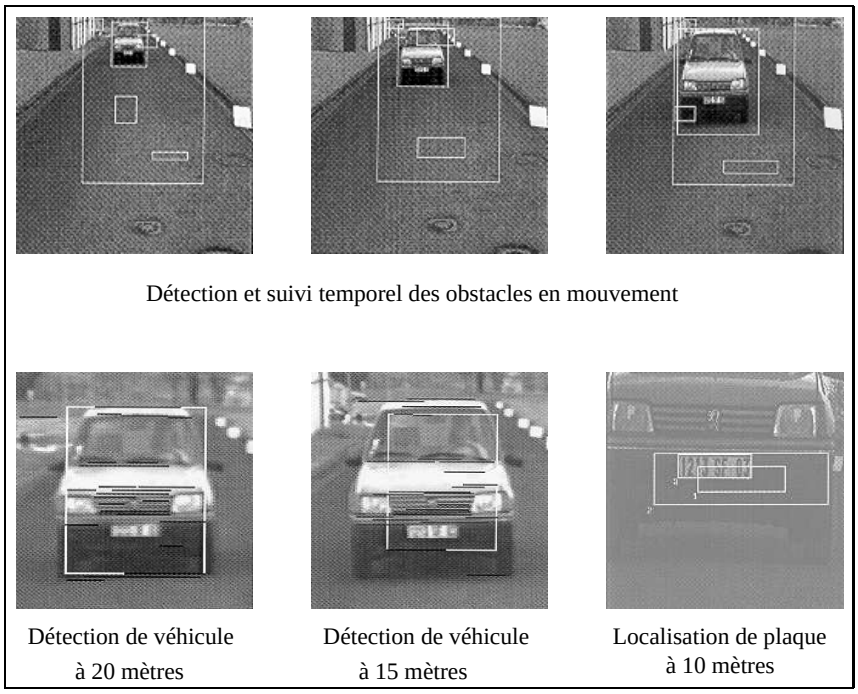


Figure 1.6: Résultats de la chaîne de localisation de plaques minéralogiques

1.2.2 Classification du Traitement d'images

Le TI, utilisé dans le domaine de la vision par ordinateur, constitue une activité majeure dans la recherche depuis de nombreuses années[4]. Les domaines d'application de cet axe sont multiples et variés : inspection de défauts dans les usines, aide à la conduite, manipulation dans des lieux à risques, etc.

Les trois exemples décrits précédemment, sans être exhaustifs, montrent la variété des structures algorithmiques intervenant dans le domaine du TI. Celles-ci sont caractérisées par une hétérogénéité :

① des structures de données traitées (allant des matrices bi-dimensionnelles de type image aux graphes d'attributs),

② des algorithmes mis en œuvre (allant d'un simple enchaînement séquentiel de fonctions jusqu'à une ordonnancement dynamique de tâches).

Concernant la nature des opérateurs mis en jeu, diverses classifications ont vu le jour. Traditionnellement, on décompose le TI en trois classes d'abstraction nommées respectivement bas, moyen et haut niveau[WA91].

[4]Cette activité est largement représentée dans les thèmes de recherche du LASMEA.

1.2.2.1 Le bas niveau

Les **traitements bas niveau** concernent essentiellement l'ensemble des opérations de pré-traitement et transformation dont la caractéristique première est de conserver la topologie bi-dimensionnelle de l'image (transformation image → image). Une même séquence d'instructions est appliquée à chaque point de l'image dans le but de faire ressortir certaines caractéristiques. Comme exemple, nous pouvons citer les opérateurs de calcul de gradient qui quantifient les variations de niveaux de gris entre pixels voisins.

A l'intérieur des traitements bas niveau, il est possible d'effectuer également une classification en sept catégories[Jon93] :

- Opérations ponctuelles : le pixel résultat issu de la transformation dépend uniquement de la connaissance du pixel source dans l'image de départ (Exemple : opérateurs de seuillage).
- Opérations sur voisinage local : les valeurs des pixels de l'image de destination sont fonction à la fois des pixels de mêmes coordonnées dans l'image source et également d'un voisinage local situé autour de ce pixel (Exemple : convolution).
- Opérations avec références non locales : ce type d'opérateurs se différencie du précédent dans le fait où le voisinage n'est plus local mais peut être situé n'importe où dans l'image source (Exemple : estimation de mouvement, calcul de flot optique).
- Opérations récursives sur voisinage : le résultat s'exprime à partir du pixel de mêmes coordonnées, d'un voisinage de pixels dans l'image source ainsi que d'un voisinage dans l'image résultat (Exemple : implantation récursive de filtres à réponse impulsionnelle infinie).
- Opérations globales : la valeur d'un pixel résultat est fonction de l'ensemble des pixels de l'image source (Exemple : FFT).
- Opérations géométriques : les pixels résultats sont fonction à la fois du pixel similaire dans l'image source, d'un voisinage local autour de celui-ci et d'une transformation de position.
- Opérations statistiques : le scalaire ou le vecteur de paramètres résultat est fonction de l'ensemble des pixels de l'image source (Exemple : histogramme des niveaux de gris).

1.2.2.2 Le moyen niveau

Les **traitements moyen niveau** ont pour but l'extraction de l'information utile dans les images traitées par les processus bas niveau. Ils constituent une étape intermédiaire entre le bas et le haut niveau, manipulant conjointement des représentations bi-dimensionnelles ou symboliques.

Ils ont essentiellement pour objectif de diminuer le volume des données (seule l'information utile est conservée). Les algorithmes mis en jeu sont plus complexes que ceux utilisés lors des étapes de bas niveau. On peut les classer en trois catégories distinctes[Leg95] :

- Opérations image → liste : ces opérations extraient l'information utile dans les images traitées par le bas niveau. Ces traitements permettent de lier les pixels ayant une caractéristique commune. Les pixels ne sont plus vus comme position dans l'image mais comme point caractéristique (Exemple : chaînage de points contour).

- Opérations liste → liste : ces traitements permettent de créer des entités pour lesquelles la représentation de l'information est beaucoup plus dense (Exemple : approximation polygonale d'une chaîne de points connexes).

- Opérations liste → graphe : dans cette phase sont recherchés les indices visuels liés par des caractéristiques géométriques (symétrie, colinéarité, orthogonalité, etc.), relationnelles (à droite de, à gauche de, etc.) à partir des données de type chaînes, segments, courbes, etc.

1.2.2.3 Le haut niveau

Les **traitements haut niveau** exploitent les primitives issues des processus de moyen niveau. Ils ont pour but l'interprétation de l'image ou la reconnaissance des formes. Leur sémantique est beaucoup moins bien définie que celle des algorithmes de bas et moyen niveau. Ils sont le plus souvent basés sur la mise en correspondance des primitives avec un modèle extrait d'une base de données. Ces opérations font souvent appel à des techniques issues de l'Intelligence Artificielle.

L'ensemble des caractéristiques des TI bas, moyen et haut niveau est résumé dans le tableau 1.1.

	Bas niveau	Moyen niveau	Haut niveau
Structure des données	matrice image	liste de points, graphe de relations	liste, graphe
Traitements	locaux opérations simples et répétitives	non locaux opérations récursives et itératives	algorithmes de reconnaissance de formes
Nombre d'opérations	constant	variable	variable

Table 1.1: Caractéristiques des opérateurs de TI

1.2.3 Contraintes relatives aux systèmes temps réel embarqués

On s'intéresse dans ce document à la programmation des systèmes informatiques pour des applications de traitement et d'analyse d'images soumises à des contraintes temps réel et d'embarquabilité opérant sur des flots continus d'images issues d'un capteur CCD classique.

Premièrement, la notion de temps réel est cruciale pour le type d'applications de TI que nous développons. Il est nécessaire de disposer d'une structure de calcul puissante autorisant une description et une interprétation de la scène observée en une *latence*[5] imposée par les contraintes liées à l'application. Par exemple, un véhicule automobile se déplaçant à une vitesse de 100 km/h parcourt une distance de 27 mètres en une seconde. Un pilotage automatique de ce véhicule ou seulement une aide à la conduite doit disposer des résultats d'interprétation de la scène routière à une latence inférieure à 100 millisecondes pour permettre un pilotage ou une aide efficace.

Deuxièmement, la notion d'embarquabilité de la structure de calcul est liée à des contraintes de type volumique et énergétique. Dans le domaine des applications routières, il est indispensable d'avoir des ordinateurs puissants occupant un volume minimum, nécessitant le moins d'énergie possible et capables d'être opérationnels quelles que soient les conditions (secousses, variations de température par exemple).

Troisièmement, la structure de calcul embarquée doit être réactive[BB91], c'est-à-dire produire une commande en réaction à chaque variation d'état de l'environnement. Elle reçoit les informations à traiter directement sous forme

[5]cf. paragraphe 1.4.2.3.2

numérique par l'intermédiaire d'un capteur (caméra vidéo, télémètre laser, radar, GPS, etc.), effectue des traitements sur ces données et produit en sortie une commande destinée aux actionneurs. Les conditions d'exécution et de *validation* des chaînes de traitements sont alors radicalement différentes de celles dans le cas d'exécution sur station de travail opérant sur des données stockées sur disque dur.

L'ensemble des trois contraintes discutées brièvement ici revient à résoudre un problème d'optimisation (minimisation de ressources matérielles) sous contraintes (temps réel)[LS97]. Cela impose donc au développeur d'applications d'utiliser au maximum les ressources architecturales mises à sa disposition afin de satisfaire au mieux les exigences temporelles liées à sa chaîne de traitements.

1.3 Architectures dédiées à la vision

Bien que les processeurs séquentiels soient de plus en plus performants et résolvent rapidement des problèmes très coûteux en temps d'exécution, certains systèmes de traitement et d'analyse d'images nécessitent des puissances de calcul qu'aucun processeur ne peut fournir à l'heure actuelle. Satisfaire les exigences temporelles impose de plus en plus la réalisation de calculateurs spécialisés construits sur mesure et répondant aux spécificités du domaine.

Ces architectures dédiées sont élaborées à partir de processeurs pouvant s'interfacer avec les capteurs fournissant les données et de médias de communication dont les débits doivent être en rapport avec les contraintes temps réel et avec le volume des informations à échanger.

L'objectif des travaux présentés dans ce mémoire n'est en aucun cas de faire un état de l'art des machines dédiées à la vision artificielle. Le lecteur désirant une liste plus complète et plus détaillée peut se rapporter à [Cha93]. Aussi, nous ne mentionnerons que les trois exemples suivants à titre d'illustration :

- Le **Calculateur Fonctionnel**[QZ92] est une architecture massivement parallèle dédiée à l'émulation temps réel d'applications de TI. L'originalité essentielle de ce projet réside dans la construction d'une architecture capable de réaliser physiquement des graphes d'opérateurs selon un mode d'exécution strictement flot de données. Cette architecture se présente sous la forme de deux réseaux hétérogènes de processeurs. Le cœur de la machine se compose d'une maille cubique tridimensionnelle de 1024 processeurs spécifiques flot de données

(μPCF[QZ91]) chargés d'exécuter des opérations bas niveau. Les traitements de plus haut niveau sont gérés par un co-réseau bi-dimensionnel de 12 Transputers connectés aux μPCF.

- **Symphonie**[CGJ+96] est une machine de type SIMD (Single Instruction Multiple Data) composée de 32 à 1024 processeurs super-scalaires organisés en anneau. Le point fort de cette architecture réside dans son réseau de communication agencé selon un anneau de cellules et pouvant fonctionner de façon autonome. Cette architecture permet de répondre de manière efficace à l'implantation de traitements de bas niveau.

- **Transvision**[LCD93] développée au LASMEA est une architecture hétérogène de type MIMD-DM[6] à mémoire distribuée reposant sur deux modèles d'exécution choisis pour leur adéquation et pour leur disponibilité sous forme de modules électroniques commerciaux. Premièrement, le modèle à recirculation d'images basé sur des modules Data Cube (digitalisation d'images, modules de traitement bas niveau). Deuxièmement, un modèle MIMD basé sur des modules nommés *noeuds vidéo* à base de Transputer T9000 s'interfaçant et échangeant des données avec le premier modèle. Cette architecture sur laquelle seront implantées les applications de TI développées dans ce mémoire sera plus longuement décrite au chapitre 4.

De telles architectures dédiées permettent en règle générale de satisfaire les exigences temporelles des applications. Cependant, celles-ci peuvent être tellement élevées que l'architecture ne suffit pas toujours. Cela peut conduire à utiliser, en complément, des circuits spécialisés (de type ASIC[7] ou FPGA [8]) qui implémentent de manière cablée des opérateurs bas niveau (filtre, convolution, etc.), déchargeant ainsi les processeurs de calcul.

1.4 Programmation des architectures dédiées

1.4.1 Présentation succincte des modèles de parallélisation

Du fait de la très grande variété des architectures cibles et également des applications candidates à l'implantation, de nombreuses recherches ont été

[6]Multiple Instruction Multiple Data - Distributed Memory.
[7]Application Specific Integrated Circuit.
[8]Field Programmable Gate Array.

menées dans le but de définir des modèles de programmation parallèles à la fois simples à utiliser et efficaces.

Skillicorn et Talia dans [ST98] font un large état de l'art de ces méthodologies et proposent une classification basée sur six niveaux hiérarchiques d'abstraction allant des langages totalement explicites jusqu'aux formalismes de très haut niveau entièrement implicites. Les niveaux intermédiaires sont définis en fonction de l'aptitude des langages à rendre transparents (ou non) l'expression du parallélisme, la décomposition en tâches concurrentes, la phase de placement-ordonnancement et les gestions des communications et des synchronisations.

Nous allons parcourir rapidement cette classification afin de montrer la large variété des modèles pouvant être utilisés. Toutefois, les exemples cités ci-après le sont à titre d'illustration et ne constituent donc pas une revue complète et détaillée de l'ensemble des modèles de programmation parallèle.

Au plus bas de la classification, on trouve effectivement les formalismes les plus explicites qui ne cachent au développeur aucun détail de la mise en œuvre du parallélisme rendant celle-ci délicate mais permettant en contrepartie une exploitation optimale des ressources architecturales. Une description relativement complète de ces langages est faite dans [Goo94]. Citons toutefois Occam[JG88][Inm89] en raison de ses liens privilégiés avec le Transputer. Occam est basé sur le modèle CSP[Hoa85] dans lequel plusieurs processus séquentiels coopèrent et communiquent. Cette coopération est assurée par un ensemble de constructeurs du langage (PAR, SEQ et ALT), les communications étant gérées de manière synchrone selon le principe du *Rendez-vous*.

Il est aussi intéressant d'évoquer les langages synchrones déclaratifs tels que Lustre[CPHP87], Signal[BGJ91] ou impératifs tels que Esterel[BS91]. Ces langages permettent l'expression du parallélisme et la modélisation du temps indépendamment de l'implantation de l'application sur l'architecture cible sous la forme de relations entre des suites d'événements valués.

Enfin, n'oublions pas de citer MPI[9][For93b] et PVM[10][GA93] — largement utilisés dans le monde industriel — fournissant des bibliothèques de communication qui, associées à un langage de programmation tel que le C, permettent de transformer un réseau de stations de travail en une machine parallèle.

Au niveau supérieur, les langages de programmation parallèle nécessitent une spécification explicite des mécanismes de communication mais réduisent leur complexité du fait d'une gestion automatique des synchronisations as-

[9]Message Passing Interface.
[10]Portable Virtual Machine.

sociées. En règle générale, de tels langages reposent sur des communications asynchrones. Par exemple Actors[Agh86][AMST97] est constitué d'objets — nommés *acteurs* — communiquant par le biais de queues de messages. Les messages sont donc envoyés de manière asynchrone et peuvent être délivrés dans un ordre quelconque. Chacun des acteurs exécute la même séquence de traitement : lecture des messages entrants dans la queue, renvoi de messages vers les autres acteurs, traitement des messages et attente d'un nouveau message.

La gestion automatique des communications caractérise les modèles de niveau supérieur. L'utilisateur est toujours contraint de décomposer en tâches son application ainsi que d'effectuer un placement manuel de l'ensemble de celles-ci sur le réseau de processeurs. Par contre, les échanges de données entre les modules nécessaires à la résolution du problème sont gérés par le système. Dans LINDA[ACGK88] (et tous ses dialectes utilisés dans des environnements temps réel tels que HLINDA[11][Yao96] et MLINDA[12][Par97]), les communications point à point sont remplacées par un mécanisme de lecture/écriture dans une "pseudo mémoire globale partagée"[13] nommée *tuple space*. Si un processus veut communiquer avec un autre, il place un message associé avec une clé dans l'espace de tuples (émission non bloquante). Lorsqu'un processus a besoin d'un message, il va le chercher dans l'espace de tuple en utilisant la clé. Si le message est présent, le processus le prend sinon il attend que le dépôt se fasse (réception bloquante). Ce mécanisme simple permet aux processus de communiquer entre eux sans se connaître (découplage spatial) et sans rendez-vous (découplage temporel).

Toutefois, si ces langages apportent au programmeur plus de facilités de développement, la phase de placement-ordonnancement (c'est-à-dire l'affectation des processus ou tâches aux processeurs et la définition de l'ordre d'exécution de ces processus sur les processeurs) gérée manuellement peut s'avérer complexe dans le cas d'applications réalistes. Aussi, de nombreux efforts ont été menés pour mettre en place des outils effectuant automatiquement cette phase de placement-ordonnancement. Parmi ceux-ci, il existe à la fois des langages textuels de programmation et de nombreux environnements graphiques. Au niveau langage, BSP[McC94] (Bulk Synchronous Parallelism) permet de décrire les applications sous la forme de *supersteps* (*i.e.* une séquence de calcul opérant sur des données locales associée à un mécanisme de communications globales et une barrière de synchronisation).

[11]Linda Hiérarchique.

[12]Micro Linda.

[13]En fait, cette mémoire peut très bien être distribuée dans le cas d'une architecture de type MIMD-DM. Elle est alors dite "virtuellement" partagée.

L'implémentation courante de BSP utilise une bibliothèque SPMD[14] (les mêmes calculs opérant sur des données différentes sont effectués sur chaque processeur). Au niveau interface graphique, il existe divers environnements tels que Millipede[ABa92], DiGraph[Aa95], HeNCE[15][BDG+94], MP[MD92]. Ils se différencient les uns des autres par la manière dont ils représentent le parallélisme. Ainsi dans les graphes de HeNCE, les noeuds sont des fonctions de calcul et les arcs les dépendances entre ces fonctions alors que dans les autres interfaces, les noeuds représentent des processus et les arcs des canaux de communication. De plus, Millipede, DiGraph et MP sont dédiés à des architecture à base de Transputers alors que HeNCE est plus destiné aux réseaux de stations de travail et utilise la bibliothèque de communication PVM.

Si les outils que nous venons de citer imposent au programmeur de décomposer son application en modules parallèles, les modèles appartenant au niveau d'abstraction supérieur nécessitent uniquement d'exprimer le parallélisme potentiel (souvent sous sa forme maximale). Les langages flot de données tels que Sisal[MSA+85] expriment parfaitement cette notion car l'exécution des opérations est uniquement conditionnée par les dépendances des données. Une opération est exécutée une fois que toutes ses entrées sont disponibles. De ce fait, deux opérations n'ayant aucune dépendance mutuelle sont potentiellement parallélisables. En dehors des langages flot de données et appartenant à cette catégorie de modèles, il existe également divers dialectes de langages de programmation classiques. En particulier, on peut citer HPF[16][For93a] — basé sur le langage Fortran — qui permet de spécifier des applications à parallélisme de données par l'ajout dans le code de directives de compilation permettant de spécifier les opérations parallélisables.

Enfin, au plus haut niveau de la classification, on trouve les langages de programmation au sein desquels l'extraction et l'exploitation du parallélisme se fait de manière totalement implicite (*i.e.* transparente pour le programmeur). L'extraction et l'utilisation du parallélisme des programmes sont gérées par des compilateurs spécifiques. Leur but est de parvenir à décomposer l'application en un ensemble de formes pouvant être exécutées concurremment. Les langages fonctionnels modernes comme Haskell[Tho96], basés sur le λ-calcul de Church[Chu41], appartiennent à ces modèles.

Cette présentation générale des modèles de programmation parallèle met en évidence la large variété des problématiques abordées et des solutions proposées. L'émergence de modèles très abstraits prouve le besoin de décharger

[14]Single Program Multiple Data.
[15]Heterogeneous Network Computing Environment.
[16]High Performance Fortran.

le programmeur des tâches bas niveau fastidieuses. Toutefois, leur implantation sur plate-forme cible avec de fortes contraintes temporelles est difficile. A l'opposé, les modèles très explicites autorisent des performances satisfaisantes mais au prix d'un coût important de développement et de maintenance. Si ces modèles sont encore largement représentés — en particulier MPI et PVM dans le monde industriel —, on assiste cependant à un fort engouement pour les modèles de niveau intermédiaire offrant un compromis entre la facilité de développement des applications au niveau utilisateur (notion d'**abstraction**) et les performances finales des applications (notion d'**efficacité**).

1.4.2 Problèmes liés à la parallélisation

Dans le cas des architectures monoprocesseurs, l'algorithme est traduit en un seul programme composé d'une séquence d'instructions s'exécutant séquentiellement sur le processeur. A l'opposé, un algorithme parallèle (devant être implanté sur la machine Transvision par exemple) est un ensemble de séquences d'instructions — chacune étant allouée à un processeur différent — au sein desquelles on introduit des instructions de communications[17]. Celles-ci ont pour but de gérer les dépendances de données entre les instructions s'exécutant sur des processeurs différents.

L'implantation d'un algorithme de TI sur une plate-forme parallèle MIMD-DM peut se décomposer en trois phases successives nommées respectivement **conception**, **implantation** et **validation**.

1.4.2.1 Conception d'une application parallèle

La parallélisation d'algorithmes cherche à décomposer un traitement global en un ensemble de d'actions s'exécutant sur des processeurs différents. Cette décomposition peut s'effectuer soit au niveau des opérations, soit au niveau des données.

Dans le premier cas, il s'agit de partitionner l'algorithme en un ensemble d'actions (ou tâches) s'exécutant de manière concurrente (*parallélisme de tâches*). Dans le second cas, la parallélisation revient à décomposer les données à traiter en différents blocs sur lesquels une même action va être exécutée en parallèle (*parallélisme de données*).

Dans les deux cas, ce partitionnement est le résultat d'une analyse approfondie des mécanismes de calcul mis en jeu dans l'application. Cela équivaut

[17] Ces communications se font soit par passage de messages dans le cas des architectures à mémoire distribuée, soit par partage de variable dans le cas des architecture à mémoire partagée.

à étudier les dépendances de données et à déterminer parmi l'ensemble des actions celles qui sont potentiellement exécutables en parallèle (*i.e.* celles qui n'ont pas de dépendance mutuelle). Cela revient à transformer l'ordre total d'exécution des instructions d'un programme séquentiel à un ordre partiel, exhibant ainsi le **parallélisme potentiel** de l'algorithme[GLS98].

L'identification du parallélisme potentiel est intimement lié à la notion de **granularité** des calculs et des données. La granularité de traitement d'un processeur est définie comme le volume de travail effectué par le processeur indépendamment des autres. On parle de parallélisme à *grain fin* lorsque le volume correspond à quelques instructions élémentaires (addition, comparaison, etc.) et/ou à quelques données élémentaires (pixel en TI). Le parallélisme potentiel est alors maximal. A l'inverse, on parle de parallélisme à *grain moyen* et *fort* lorsque l'on préfère regrouper et exécuter séquentiellement des instructions qui pourraient être exécutées simultanément et/ou des données qui pourraient être évaluées concurremment.

Cette notion est étudiée par exemple dans [GSD97] dans le cas de l'implantation parallèle d'une application d'étiquetage en composantes connexes. Le schéma de parallélisation utilisé est de type SPMD avec fusion. Pratiquement, l'image acquise est divisée en blocs de données de taille fixe, lesquelles sont réparties sur l'ensemble des processeurs. Une même fonction de calcul est alors appliquée sur chaque bloc de données. Les résultats respectifs issus de la fonction de calcul sont fusionnés pour produire le résultat final. Compte-tenu du format des images traitées, trois tailles de grains ont été envisagées : pixel, ligne et bande.

Avec une granularité fine (pixel), le parallélisme potentiel a été exprimé de façon maximale ce qui a conduit à spécifier 2^{16} opérations d'étiquetage (pour des images de taille $2^8 * 2^8$). L'implantation d'une telle application sur la machine Transvision s'est avérée impossible à réaliser en raison de l'explosion combinatoire des possibilités de placement des opérations.

Avec une granularité moyenne (ligne), les résultats temporels de l'application montrent un ralentissement de l'application par rapport à une exécution séquentielle en raison de la forte densité de communications.

Avec une granularité forte (bande), la limitation du nombre de communications permet d'obtenir des performances temporelles intéressantes.

Cet exemple permet de faire la conclusions suivante. Idéalement, le parallélisme potentiel maximal d'un algorithme de TI est exhibé en exprimant celui-ci avec une granularité pixel puisque c'est celle qui fait le moins d'*a priori* sur les modalités d'exploitation de ce parallélisme. En pratique, une telle approche peut soulever deux problèmes : primo, elle peut nécessiter

une étape de reformulation algorithmique plus ou moins profonde (avec une granularité ligne ou bande dans le cas de l'exemple) limitant sa portée auprès d'un public habitué aux spécifications séquentielles. Secondo, elle accroît la complexité de la phase d'implantation du fait de l'explosion combinatoire des possibilités de placement des actions sur les processeurs.

Cette conclusion montre que la phase de conception d'une application parallèle de TI est liée directement à la notion de granularité. Celle-ci conditionne l'expression du parallélisme potentiel de l'application. Théoriquement, l'extraction de ce parallélisme est décorrellée du parallélisme disponible sur l'architecture. En réalité, la pratique montre que les deux sont dépendants et que la prise en compte des caractéristiques architecturales influence fortement les performances de l'application. Ce constat met en évidence la difficulté de conception des applications pour des développeurs peu spécialistes et impose la mise en place d'outils de prototypage rapide permettant de tester différentes granularités par essais successifs de plusieurs solutions..

1.4.2.2 Implantation d'une application parallèle

La phase d'identification du parallélisme potentiel de l'application conduit à décomposer l'algorithme en un ensemble d'actions concurrentes et échangeant des données. Dès lors, l'implantation apparaît comme un problème d'allocation de ressources, que l'on désire réaliser de manière efficace, où les ressources sont les processeurs et les médias de communication et où les actions à allouer à ces ressources sont les instructions de l'algorithme ainsi que celles de communications.

La phase d'implantation de l'algorithme sur la machine cible consiste en un **placement-ordonnancement** des calculs et des communications sur le réseau de processeurs. D'une manière générale, il s'agit d'effectuer une réduction du parallélisme potentiel de l'algorithme à celui disponible sur l'architecture cible. Cela implique de distribuer et d'ordonnancer les programmes sur les processeurs en équilibrant au mieux la charge de calcul des processeurs et en minimisant certaines contraintes (latence, ressources, etc.). La distribution est une répartition spatiale ("Quelle action sur quel processeur ?") alors que l'ordonnancement représente une répartition temporelle ("Quelle action à quel moment ?"). Trouver LA solution minimisant par exemple le temps total d'exécution dans le cas d'applications possédant un fort parallélisme potentiel exécutées sur une architecture comportant une dizaine de processeurs conduit inévitablement à une explosion combinatoire des possibilités de répartition (problème NP complet).

La phase de placement-ordonnancement des actions sur le réseau de

processeurs impose en règle générale d'effectuer un contrôle explicite des transferts de données sur les liens physiques des processeurs. En effet, la dépendance de données entre deux actions de calcul placées sur deux processeurs distants entraîne la mise en place d'une primitive de communication entre les deux processeurs. Cette gestion des transferts de données devient d'autant plus complexe dans le cas où les communications ont lieu entre des processeurs non directement connectés. S'il n'est pas possible d'effectuer du routage automatique de données (par un routeur intelligent de type C104 pour les Transputers par exemple), le programmeur est obligé de contrôler explicitement ce routage sur l'ensemble des processeurs intermédiaires.

Enfin, écrire un programme parallèle peut, dans certains cas, ne pas se limiter à un ensemble de séquences de calcul exécutées concurremment sur différents processeurs. Certains processeurs autorisent le recouvrement calcul-communication (cas du Transputer) grâce à des DMA[18] ne sollicitant pas le séquenceur d'instructions pendant les transferts. Dans ce cas, on a tout intérêt à exécuter en parallèle les calculs et les communications sur chaque processeur. Ceci impose en contrepartie une gestion d'un mécanisme de synchronisation entre les calculs et les communications conduisant à un ordonnancement des opérations de calcul et des communications dû aux dépendances de données. Par exemple, si le transfert d'une variable contenant le résultat d'une action de calcul ne peut pas avoir lieu avant la fin de l'action, cette action ne peut non plus être re-exécutée avant que le transfert soit achevé.

L'ensemble de ces difficultés augmente fortement la complexité de programmation des architectures parallèles et ne peut être généralement résolu efficacement par des programmeurs habitués du domaine séquentiel. Seuls des programmeurs spécialistes des architectures parallèles, sont en mesure d'obtenir des implantations correctes et *a fortiori* performantes. Toutefois, dans le cas d'applications complexes, l'espace des solutions possibles et envisageables ne pourra être exploré exhaustivement du fait du temps important dépensé pour obtenir et tester une seule implantation.

Pour pallier ces difficultés, la seule solution passe par le recours à un **outil de prototypage rapide** fournissant d'une part un formalisme de spécification des programmes parallèles et automatisant d'autre part la phase d'implantation sur l'architecture cible. L'existence d'un tel outil devrait permettre à la fois aux programmeurs peu expérimentés d'implanter facilement leurs applications et aux spécialistes d'explorer plus efficacement de vastes sous-ensembles de l'espace des solutions afin d'optimiser leur programmes et

[18]Direct Memory Access.

ainsi d'accroître leur expertise dans le domaine.

1.4.2.3 Validation des implantations

En programmation séquentielle, l'exécution d'un programme a pour but de vérifier le comportement fonctionnel de l'algorithme c'est-à-dire la cohérence des résultats produits en fonction des données d'entrée.

En programmation parallèle avec de fortes contraintes temporelles, l'exécution d'une application doit non seulement valider son comportement fonctionnel (comme dans la programmation séquentielle) mais aussi que le résultat produit est obtenu en un temps suffisamment faible pour satisfaire les contraintes temporelles désirées (cadence vidéo par exemple).

1.4.2.3.1 Validation fonctionnelle

La validation fonctionnelle de l'application consiste à vérifier le comportement exact de l'algorithme. Etant donné les difficultés liées à la programmation parallèle (décrites dans les paragraphes précédents), la pratique montre que les phases de mise au point des implantations représentent une partie importante du temps de développement. En effet, la validation d'une application passe par la mise en place d'un cycle itératif d'exécution et de correction permettant l'extraction des erreurs provenant soit de la programmation des calculs (problèmes identiques à ceux du monde séquentiel), soit de l'implantation sur l'architecture (problèmes de nature parallèle résultant d'un mauvais ordonnancement des communications). Si les premières sont en général assez facilement identifiables et résolubles[19], les secondes posent des problèmes originaux et difficiles, liés notamment à la nature de l'architecture et à l'exécutif associé. Sans outil sophistiqué de mise au point, la détection et la correction de ces erreurs est une tâche difficile puisqu'il n'est pas toujours possible de déterminer à tout instant l'état du programme. Seuls des programmeurs spécialistes sont alors aptes à résoudre de telles difficultés rendant *de facto* alors la parallélisation inabordable aux programmeurs inexpérimentés dans le domaine.

1.4.2.3.2 Validation temporelle

Le recours aux architectures parallèles a pour objectif d'augmenter les performances des applications. De fait, pour que l'implantation d'un al-

[19]En tout cas, elles ne sont pas plus difficiles que dans le domaine séquentiel.

gorithme soit complètement validée, il est nécessaire de vérifier, en plus du comportement correct de l'algorithme, le respect des contraintes temporelles. Ces contraintes sont de deux types : *latence* et *cadence*.

La latence est la durée d'exécution d'une itération de l'application conduisant à produire le prochain résultat en sortie, en réponse à une donnée d'entrée. La cadence représente la durée qui s'écoule entre deux données d'entrée (dans le cas du TI, la cadence est de 25 images par seconde).

Dans un contexte temps réel, la première grandeur observée est la latence de l'application. En effet, dans la majeure partie des cas, le cahier des charges de l'application impose une borne maximale de temps à respecter.

On peut également mesurer la performance d'une application par deux grandeurs nommées respectivement **accélération** et **efficacité**. L'accélération (ou *speedup*) mesure le gain temporel entre la meilleure exécution séquentielle sur un processeur et l'exécution sur n processeurs :

$$A_{cc} = \frac{t_s}{t_p} \tag{1.1}$$

L'efficacité mesure le rendement de l'application et se définit comme suit :

$$E_{ff} = \frac{t_s}{n * t_p} = \frac{A_{cc}}{n} \tag{1.2}$$

Les temps t_s et t_p représentent respectivement les temps d'exécution séquentiel et parallèle de l'application.

Ces trois grandeurs (latence, accélération et efficacité) permettent de valider ou non le respect des contraintes imposées. Dans le cas d'un non respect, deux solutions sont envisageables : soit considérer que l'implantation est inefficace, soit considérer que la formulation algorithmique est inadaptée.

Dans le premier cas, il est nécessaire de refaire une étude du placement-ordonnancement de l'application sur l'architecture cible après avoir détecté les goulets d'étranglement (aussi bien au niveau calcul que communication). Dans le deuxième cas, il faut reprendre complètement l'étude algorithmique ce qui est fort coûteux en temps de développement.

Il apparaît donc clairement que les choix algorithmiques ne peuvent être validés (ou invalidés) que très tardivement. Le programmeur évolue longtemps dans l'inconnu sans savoir *a priori* si ses choix sont adaptés. Cela montre la nécessité d'avoir à disposition un outil effectuant des mesures prédictives de performances. Cet aspect doit permettre d'évaluer les solutions potentielles de spécification et d'implantation avant toute tentative

d'exécution sur la plate-forme cible.

1.4.3 Problèmes liés aux architectures dédiées

Pour faire face au besoin de puissance des applications de TI temps réel, nous avons montré au paragraphe 1.3 que les architectures parallèles dédiées au domaine constituaient une solution envisageable. De plus, les conclusions du paragraphe 1.4.1 concernant les modèles de programmation ont mis en évidence le besoin d'un formalisme propre au TI et conciliant **abstraction** et **performances**. La conciliation de ces deux objectifs semble difficile à atteindre.

En effet, une architecture dédiée est développée pour répondre à une classe de problèmes particuliers. En règle générale, ce développement s'accompagne de la mise en place de langages de programmation[20] tirant parti des caractéristiques bas niveau de l'architecture. Par exemple, on peut citer SPL[Via96] qui est un langage de développement assembleur pour Symphonie. Extrêmement spécifique, ce langage permet d'obtenir de bonnes performances mais demeure inutilisable pour d'autres architectures.

A l'opposé, la majeure partie des modèles de programmation décrits jusqu'à présent sont trop généralistes dans le sens où ils couvrent une large gamme d'architectures. Prenons comme exemple MPI. Des implémentations de MPI existent pour certaines architectures parallèles généralistes (IBM SP-2, Cray T3E, etc.) et pour un grand nombre de couples stations de travail-système d'exploitation du commerce. Par contre, en ce qui concerne les architectures dédiées, l'utilisation de MPI passe inévitablement par le portage de cette bibliothèque pour le type de processeurs et de médias de communication composant la plate-forme cible. Compte-tenu de la richesse du standard MPI, cette phase de portage peut nécessiter un temps important de développement sans toutefois garantir par la suite des implantations performantes.

1.4.4 Synthèse des problèmes et premières conclusions

La mise en œuvre d'applications parallèles de TI soulève de nombreux problèmes théoriques et pratiques que seuls des spécialistes peuvent maîtriser. En effet, le programmeur reste le plus souvent confronté à un problème délicat de mise en correspondance du parallélisme potentiel, présent au niveau de

[20]Par langage, on entend ici tant la définition d'un nouveau formalisme que l'ajout de mots-clés et/ou de bibliothèques à un langage existant (C, Fortran, etc.).

son application, avec celui disponible, potentiellement utilisable, sur la machine cible. Ce problème se décompose lui-même en deux sous problèmes interdépendants :

① Comment exprimer avec un minimum d'effort le parallélisme potentiel ?

② Comment exploiter au mieux le parallélisme disponible ?

De plus, la validation des choix algorithmiques n'intervient que très tardivement dans le processus de développement (au moment de l'exécution). Le processus de développement est alors un cycle Conception-Implantation-Validation avec rebouclage.

Ces caractéristiques entraînent forcément des temps de développement importants ne permettant pas d'évaluer un vaste sous-ensemble de l'espace des solutions[21].

D'un autre côté, l'étude des langages de programmation parallèle a mis en évidence le besoin d'un formalisme conciliant abstraction et performances des programmes, facilitant le développement des applications tout en respectant les contraintes temporelles imposées.

Ces aspects prouvent la nécessité de mettre en place des **outils de spécification de haut niveau** permettant :

① de décharger le développeur de la programmation bas niveau souvent fastidieuse (implantation des mécanismes de synchronisation et de communication par exemple). Cette caractéristique valide alors la notion de **prototypage rapide** d'applications en réduisant les temps de développement,

② de faciliter l'étude des relations entre le parallélisme potentiel présent au niveau de l'algorithme et celui disponible au niveau de l'architecture. La mise en place d'outils automatisant la phase de placement-ordonnancement permet alors de réaliser une **adéquation** entre **algorithme** et **architecture** c'est-à-dire calculer la meilleure implantation de l'application sur la plate-forme cible. L'intégration de mesures prédictives de performances au sein de ces outils permet alors de simuler le comportement de l'application placée et ordonnancée sans avoir besoin de réaliser physiquement l'implantation.

③ au programmeur de se consacrer au développement de son application et aux **aspects temporels** qui sont cruciaux dans le domaine du TI.

[21] Cet ensemble est souvent réduit à un singleton.

Un tel outil respectant ces trois caractéristiques essentielles valide alors la notion de **prototypage rapide optimisé des applications de TI**, objectifs des travaux présentés dans ce mémoire.

Les paragraphes suivants vont présenter un ensemble d'outils spécifiques au TI sur architectures parallèles, répondant à certaines des caractéristiques discutées auparavant.

1.5 Solutions existantes au problème du prototypage rapide

1.5.1 Le Calculateur fonctionnel

Le *Calculateur Fonctionnel* est une architecture massivement parallèle dédiée à l'émulation temps réel d'applications de TI (cf. paragraphe 1.3). La méthodologie de programmation employée repose sur le concept de décomposition fonctionnelle qui conduit à représenter les traitements sous la forme de graphes orientés d'opérateurs fonctionnels reliés uniquement par des dépendances de données.

L'étude d'applications de TI a permis d'extraire tout un ensemble d'opérateurs primitifs pour le bas, moyen et haut niveau. Chacun de ces opérateurs a été implanté physiquement sur le Calculateur Fonctionnel sur un ou plusieurs processeurs élémentaires (μPCF) pour les bas et moyen niveaux et sur un Transputer pour le haut niveau. Par la suite, ces opérateurs ont été intégrés à des bibliothèques, les rendant donc disponibles pour les programmeurs d'applications. Ainsi, la description fonctionnelle de l'application reflète simultanément la fonctionnalité de l'algorithme (*i.e.* sa description comportementale) et son implantation matérielle (*i.e.* sa description structurelle).

La programmation du Calculateur Fonctionnel peut donc se voir comme l'élaboration d'un graphe orienté dont les noeuds représentent les opérateurs primitifs de TI et dont les arcs traduisent les dépendances entre ces opérateurs. La tâche de l'environnement de programmation se réduit alors à placer ce graphe flot de données sur l'architecture tout en respectant le parallélisme exprimé par les dépendances. Cet environnement peut être décomposé en quatre entités indépendantes[Ser93] assurant respectivement l'acquisition du graphe d'opérateurs, la traduction de ce graphe en graphe de processeurs, l'exécution temps réel de ce graphe sur l'architecture et l'analyse des résultats.

Premièrement, l'acquisition du graphe d'opérateurs consiste à représenter le graphe flot de données de l'application sous la forme d'une association d'opérateurs primitifs issus des bibliothèques. Cette acquisition est réalisée sous la forme textuelle dans un langage fonctionnel proche de FP de Backus[Bac78] en raison de ses aptitudes à décrire de manière concise et précise les graphes d'opérateurs. De ce fait, un compilateur associé a été développé permettant de passer d'une représentation composée d'un ensemble d'expressions fonctionnelles à une représentation sous la forme d'un graphe d'opérateurs flot de données.

Deuxièmement, la phase de translation assure la conversion de ce graphe d'opérateurs décrivant le comportement de l'algorithme en un graphe de processeurs implantable physiquement sur l'architecture cible utilisant les opérateurs de bibliothèques. Ce graphe est finalement "plaqué" sur la maille cubique du Calculateur Fonctionnel, c'est-à-dire placé et routé[22].

Enfin, les phases d'exécution et d'analyse des résultats ont pour but de valider les applications au niveau du comportement et des contraintes (qu'elles soient de nature temporelle ou matérielle).

Un des objectifs du projet Calculateur Fonctionnel est de fournir une plate forme architecturale autorisant l'implantation temps réel des applications de TI. Cet objectif a été largement validé par le développement et l'implantation d'application de bas et moyen niveau. Dans [Pra93][SQZ93][QCSZ93], on trouve la description et les résultats d'applications d'étiquetage en composantes connexes, de filtrage de Nagao, d'égalisation statistique, de détection de droites dans les directions principales, etc.Toutes ces applications opèrent sur des flots d'images issues d'une caméra à la cadence vidéo et jusqu'à des résolutions $572 * 800$ pixels.

Le Calculateur Fonctionnel propose une solution originale pour répondre au problème d'adéquation algorithme architecture en ayant recours à des opérateurs primitifs de TI dont il existe une implantation efficace sur la plateforme cible. Toutefois, cette démarche possède certaines limites puisqu'elle impose au programmeur de construire son application à partir de cet ensemble de briques de base[23]. De fait, la classe des applications implantables sur le Calculateur Fonctionnel est limitée à celles pouvant s'exprimer sous la forme d'un assemblage de ces primitives. Ajouté à cela, le lien étroit entre le modèle de programmation et l'architecture sous-jacente limite la portabilité des applications vers d'autres types d'architectures.

[22]Cette opération est réalisée par un placeur routeur pouvant être utilisé manuellement ou automatiquement.

[23]L'écriture de nouvelles primitives est possible mais suppose des connaissances assez fines sur le fonctionnement du μPCF.

1.5.2 TRAPPER

TRAPPER (TRAffonic Parallel Programming EnviRonment) est un outil de développement et d'analyse d'applications parallèles réalisé par Daimler-Benz[SSKF95a][BOSS95]. Cet outil constitue un environnement graphique qui assiste le programmeur tout au long du développement de l'application parallèle : conception, placement, exécution, mesures, etc..

La philosophie de cette méthodologie est de laisser le programmeur spécifier de manière explicite la structure parallèle de son application et de l'aider par la suite dans toutes les phases de développement menant finalement à l'implantation de l'algorithme sur la machine[SSKF95b].

L'environnement de TRAPPER est décomposé en plusieurs entités différentes : *DesignTool*, *ConfigTool*, *VisTool* et *PerfTool*, chacune étant dédiée à une étape de développement de l'application parallèle.

L'outil *DesignTool* permet au programmeur de spécifier le graphe de l'application dans lequel les noeuds représentent des processus séquentiels communicants et les arcs des canaux de communication par passage de messages. Le graphe logiciel (Process Graph dans la terminologie TRAPPER) décrit graphiquement la structure parallèle de l'application et ceci, de manière indépendante de l'architecture cible. A chaque noeud du graphe, on associe un identificateur unique (pid), un nom de processus qui correspond au code C séquentiel associé ainsi que des interfaces de communication. Le code C du processus est écrit textuellement par le développeur à partir d'un fichier modèle (*template*) dans lequel on introduit les appels aux fonctions de calculs et de communications.

L'outil *ConfigTool* permet de spécifier la configuration du système matériel et de déterminer le placement de l'application sur l'architecture. La configuration matérielle est réalisée avec un outil graphique en tout point similaire à celui permettant la spécification de l'application. Le graphe matériel est représenté par un ensemble de noeuds (*i.e.* les processeurs) et d'arcs de dépendance (*i.e.* les liens de communications). Dans un deuxième temps, l'application doit être placée sur l'architecture. Cette phase de placement-ordonnancement peut être réalisée soit manuellement, soit automatiquement. Dans le premier cas, chaque processeur se caractérise par une couleur propre et l'utilisateur colorie son graphe logiciel pour indiquer sur quel processeur chaque noeud doit être placé. Dans le deuxième cas, un algorithme de placement a été développé. De ce fait, TRAPPER impose à l'utilisateur de spécifier à la fois les temps d'exécution des processus et les charges de communications associées. Dans un troisième temps, le système de contrôle doit être configuré afin de permettre la collecte d'informations d'exécution au

niveau logiciel (communication, accès mémoire, etc.) et au niveau matériel (charge de calcul et de communication). Enfin, l'outil *ConfigTool* génère l'ensemble des fichiers compilables pour l'application. A l'heure actuelle, TRAPPER supporte les architectures à base de Transputer[SSBO94] et de Power PC.

Après exécution du code et collecte des informations désirées, deux outils nommés respectivement *VisTool* et *PerfTool* analysent le comportement de l'application en vue d'une éventuelle optimisation. L'outil *VisTool* permet l'animation du programme sous la forme d'affichages graphiques des phases du programme, du contenu des variables et certaines informations spécifiques. Cet outil est particulièrement adapté pour toutes les phases de mise au point.

A l'opposé, l'outil *Perftool* est plus adapté aux phases d'optimisation puisqu'il permet la visualisation du comportement matériel au niveau des charges de calcul et de communications.

Une application complexe de reconnaissance de panneaux routiers a été développée, implantée et validée en utilisant la chaîne de développement TRAPPER. Cette application est complètement détaillée dans [Est96].

TRAPPER est une chaîne complète de développement d'applications de TI permettant de satisfaire des exigences de prototypage rapide et d'optimisation. Le prototypage est possible puisqu'à partir des représentations algorithmiques et architecturales, l'implantation peut être réalisée automatiquement diminuant ainsi le temps de développement.

De plus, l'intérêt majeur d'un outil tel que TRAPPER repose sur la mise en œuvre d'outils graphiques paramétrables permettant l'analyse fine des performances des applications tant au niveau des charges de calcul des processeurs qu'au niveau des communications inter-processus. L'utilisation de tels outils facilite alors la phase d'optimisation des implantations.

En contrepartie, la représentation du graphe de l'application sous la forme de processus séquentiels communicants possède certaines limites. En effet, le programmeur est obligé d'une part de spécifier explicitement le parallélisme de son application et d'autre part de décrire le déroulement des opérations au sein des processus (appels des fonctions de calcul et de communication). Cette condition, parce qu'elles suppose une connaissance significative des concepts et mécanismes de programmation par passage de message, peut alors constituer un frein à l'utilisation de TRAPPER par des programmeurs œuvrant traditionnellement dans le monde séquentiel.

1.5.3 SynDEx

SynDEx [LSSt91a][LSSt91b] (acronyme pour Synchronous Distributed Executive) est un environnement graphique interactif d'aide à la parallélisation d'application de traitement du signal et des images et de contrôle-commande. SynDEx a été développé à l'INRIA dans le cadre de recherches sur l'adéquation algorithme architecture[Sor96] (AAA). La méthodologie AAA consiste à étudier simultanément les aspects algorithmiques et architecturaux ainsi que leurs interactions dans le but d'effectuer une implantation efficace et optimisée de l'algorithme sur l'architecture tout en satisfaisant certaines contraintes exécutives[Sor94] (cf. figure 1.7).

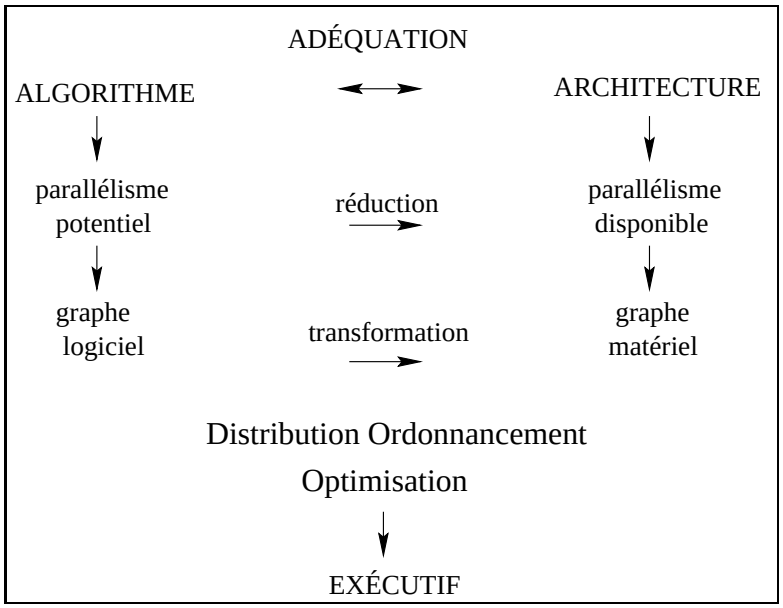


Figure 1.7: La méthodologie AAA

Cette méthodologie repose sur deux modèles de graphe distincts représentant d'une part le logiciel et d'autre part l'architecture. Cette séparation permet de dissocier clairement la spécification de l'application et son implantation sur l'architecture cible (facilitant ainsi le portage des applications vers d'autres types de machines puisque la spécification reste inchangée).

Du point de vue logiciel, l'algorithme est modélisé sous la forme d'un graphe flot de donnée conditionné synchrone (via éventuellement un langage synchrone à travers leur format commun DC[24][PBM+93][Hal95]). Ce

[24]Declarative Code.

type de graphe est une généralisation du modèle défini par Dennis dans [Den75] auquel on a rajouté certains sommets particuliers (mémoire, sous échantillonnage, mélange). Ce type de représentation induit un ordre partiel sur les opérations, permettant ainsi d'exprimer aisément le parallélisme potentiel de l'application. En effet, deux sommets du graphe (*i.e.* deux fonctions de calcul développées par le programmeur) n'ayant pas de dépendance mutuelle de données (*i.e.* n'étant pas reliés par un arc orienté) sont potentiellement exécutables en parallèle.

Du point de vue matériel, l'architecture cible est modélisée par un graphe non orienté dans lequel chaque sommet est un processeur et chaque arc une liaison physique de communication connectant deux processeurs. Ce graphe dit matériel exprime donc le parallélisme effectif de l'architecture cible.

Le terme *adéquation* sous entend une mise en correspondance efficace des graphes logiciel et matériel. Ceci consiste à réduire le parallélisme potentiel de l'algorithme (décrit dans le graphe logiciel) au parallélisme matériel disponible (décrit dans le graphe matériel). Cette réduction est obtenue en transformant le graphe flot de donnée de l'application afin de le faire correspondre au graphe matériel. Ces transformations représentent une distribution (allocation spatiale) et un ordonnancement (allocation temporelle) des calculs sur les processeurs et des communications sur les liens de transfert. Etant donné un algorithme d'application et une architecture, l'adéquation consiste à choisir parmi toutes les transformations possibles celle permettant de respecter les contraintes temps réel tout en minimisant les composants matériels utilisés. Ce problème NP complet est résolu par l'utilisation d'une heuristique qui minimise une fonction de coût dépendant à la fois des temps d'exécution des calculs (fournis par l'utilisateur) et des temps de transfert de données (calculés par SynDEx). L'objectif de cette heuristique est de fournir, en un temps relativement court, une solution généralement sous-optimale[LS93] minimisant le temps de réponse de l'application, c'est à dire la longueur du chemin critique du graphe logiciel.

L'exécution de l'heuristique de placement-ordonnancement permet d'obtenir les dates de début et de fin des opérations de calcul et de transfert de données. A partir de celles-ci, SynDEx trace un diagramme temporel décrivant une prévision du comportement temps réel de l'algorithme sur l'architecture. Ce point est important car il permet de faire l'évaluation des performances de l'application sans réaliser réellement son implantation. Dans le cas où les contraintes temporelles désirées ne sont pas respectées, on peut remettre en cause les résultats donnés par l'heuristique en imposant manuellement que certains sommets du graphe soient placés sur des processeurs particuliers, puis exécuter de nouveau l'adéquation. On peut par

la suite réitérer ce processus jusqu'à obtention de résultats prévisionnels satisfaisants. Enfin, s'il n'est vraiment pas possible de satisfaire les contraintes temporelles, il faut modifier la structure soit du graphe matériel pour augmenter éventuellement le parallélisme disponible, soit du graphe logiciel pour reformuler l'expression du parallélisme potentiel. Avec cette approche, le temps de cycle de développement étant relativement court, il est alors possible de réaliser des modifications au sein des deux graphes dans le but d'atteindre les performances voulues.

Dès qu'un placement et un ordonnancement satisfaisants ont été déterminés, SynDEx génère automatiquement un exécutif statique distribué sans inter-blocage composé d'une part des opérations de calculs et d'autre part d'un noyau générique d'opérateurs assurant les communications, synchronisations et autres opérations spécifiques aux programmes parallèles. L'exécutif généré est en fait composé d'un macro-code intermédiaire totalement indépendant des spécificités de l'architecture cible. Son expansion en code compilable se fait à l'aide du macro processeur standard GNU *m4* utilisant les macros définitions du noyau générique propres au processeur cible. Plusieurs implantations de ce noyau générique ont été réalisées pour les processeurs T800, T9000, TMS320C40, SHARC (ADSP21xxx), i80386, i80196, MC68332 ainsi que pour les réseaux de stations de travail sous Unix et Linux.

L'exécutif distribué peut être généré optionnellement avec des instructions de chronométrage[ELS92], permettant la mesure des durées effectives d'exécution de toutes les opérations. Ces durées sont indispensables pour l'évaluation et l'optimisation éventuelle des performances de l'application.

De nombreuses évaluations et validations de SynDEx ont été effectuées dans le domaine du TI. On peut citer par exemple :

- Etiquetage en Composantes Connexes[Gin95].
- Compression d'images[CV96].
- Filtre de Canny-Deriche[HYMP96],[HYMP97].
- Traitement automatique de visage[Yan98],
- Véhicule CyCab[KS98] (Véhicule électrique semi-autonome).

SynDEx, reposant sur la méthodologie AAA, est un outil de développement d'applications parallèles de TI répondant à certains de nos objectifs de prototypage rapide optimisé.

Tout d'abord, la spécification des algorithmes à l'aide de graphes flots de données conditionnés respectant la sémantique des langages synchrones

limite les erreurs de conception des applications et est réalisée de manière indépendante par rapport à l'architecture.

Ensuite, la phase de placement-ordonnancement de l'algorithme sur l'architecture est gérée par une heuristique paramétrable rapide donnant des résultats proches de la solution optimale.

Enfin, l'exécutif distribué généré libère le programmeur de toute la programmation bas niveau fastidieuse et souvent source de nombreuses erreurs, Cet exécutif, construit à partir d'un noyau générique propre au processeur cible, est garanti sans inter-blocage et donne des résultats très performants car il introduit un très faible surcoût.

Néanmoins, comme dans l'outil TRAPPER, le programmeur a la lourde tâche de spécifier explicitement le parallélisme potentiel de son application en décrivant complètement le graphe flot de données correspondant. Dès lors, les performances de l'application — et donc le respect des contraintes temporelles — dépendent inévitablement de la capacité du développeur à effectuer cette tâche de manière optimale en choisissant la granularité de traitement adéquate.

De plus, SynDEx est basé sur un modèle d'exécutif purement statique (*i.e.* le placement et l'ordonnancement des calculs et des communications sont connus à la compilation) ce qui rend impossible la description des applications basées sur un schéma dynamique de communications mais correspond à un surcoût d'exécutif minimal.

Malgré ces limites, SynDEx sera utilisé comme dorsale de notre outil (cf. chapitre 4).

1.5.4 ESPION

L'outil ESPION[Dub91] (Efficacité des Systèmes multiProcesseurs pour le traitement d'Images et ses applicatIOns) diffère des autres méthodologies présentées dans ce chapitre. En effet, ESPION ne constitue pas une chaîne complète de développement d'applications parallèles mettant en jeu divers concepts de description et techniques d'implantation. ESPION représente en fait une méthodologie de modélisation de l'implantation d'un algorithme sur une architecture multi-processeurs permettant ainsi une *évaluation prédictive* des performances de l'application. Cependant, il est intéressant de citer et de décrire un tel outil car les phases de modélisation de performances, indépendamment de toute exécution sur la plate forme cible, constituent un premier maillon indispensable de tout outil de prototypage rapide.

Le domaine d'applications dans lequel l'outil ESPION opère est limité

au TI bas niveau caractérisé par des calculs relativement simples à répéter régulièrement sur un grand volume de données. Le modèle d'implantation est de type parallélisme de données du fait des caractéristiques de régularité des données. Au niveau matériel, les architectures retenues sont de type MIMD-DM communiquant par passage de messages.

Le but d'un outil tel que ESPION est de modéliser d'une part le comportement de l'algorithme et d'autre part celui de l'architecture de manière à déterminer le meilleur compromis architecture/algorithme ou algorithme/architecture. Pour une application de TI bas niveau donnée, le problème se résume en la résolution des contraintes suivantes :

- Choix de la découpe des données au niveau algorithme.
- Choix du processeur et du réseau d'interconnexion au niveau matériel.

Chacune de ces contraintes met en jeu un certain nombre de paramètres permettant de modéliser à la fois l'algorithme et l'architecture. Ces paramètres sont détaillés dans [DSPM92]. La détermination de l'ensemble de ces paramètres se fait selon différentes techniques : soit par espionnage du code séquentiel, soit par connaissance du code séquentiel, soit par caractérisation des processeurs et du réseau.

A partir des valeurs de ces paramètres, des formules analytiques permettent de prédire l'accélération et l'efficacité dans les deux principaux cas suivants : avant et après saturation du réseau de communication. Le phénomène de saturation correspond au point d'équilibre pour lequel le calcul cache ou ne cache plus les communications. Ainsi, considérant que les calculs et les communications s'exécutent en parallèle, le calcul pourra cacher les communications avant la saturation tandis qu'une fois la saturation atteinte, le calcul sera pénalisé et devra attendre la fin des communications (dans ce cas là, le temps d'exécution sera en définitive le temps de transfert des données).

La validation de la simulation par ESPION peut être suivie par une phase de synthèse d'architectures. A cet effet, GAUT[MSDP93][MSP94] (Génération Automatique d'Unités de Traitement) est un outil de synthèse d'architectures pipelines dédiées aux applications de traitement du signal et des images sous contraintes de temps réel d'exécution.

L'approche présentée ici a été largement validée. Des applications telles que la transformée de Fourier rapide (FFT) et le filtrage de Kalman 2d sont décrites dans [SMD+93] et montrent des erreurs de prédiction de performances inférieures à 5%. Une utilisation conjointe de ESPION et de GAUT est présentée dans [Sen93] dans le cadre d'une application de restauration d'images en temps réel.

1.6 Définition d'un outil d'aide à la parallélisation et conclusion

Les paragraphes précédents consacrés aux modèles de programmation parallèle (et à leurs outils associés) nous ont permis de parcourir un large éventail des solutions envisagées au problème d'adéquation algorithme architecture dans le domaine du TI temps réel.

Si chacun des outils dédiés au TI que nous avons décrit permet d'étudier cette adéquation, il n'en reste pas moins que chacun d'entre eux a un objectif initial différent : simulation des solutions algorithmiques et architecturales pour ESPION, chaîne complète d'aide à la parallélisation pour SynDEx et TRAPPER, réalisation d'applications de TI pour le Calculateur Fonctionnel.

Toutefois, si chacun de ces outils possèdent des caractéristiques intéressantes, il n'en demeure pas moins qu'ils possèdent certaines limites : spécification explicite du parallélisme (SynDEx, TRAPPER), limitation au parallélisme de données (ESPION), portabilité limitée du fait de relations étroites entre le modèle de programmation et l'architecture sous-jacente (Calculateur Fonctionnel).

En résumé, chacun des outils présentés facilite le développement des applications parallèles en automatisant certaines tâches fastidieuses et en particulier la génération de code pour la plate-forme cible. Néanmoins, ils ne peuvent répondre complètement à nos objectifs de prototypage rapide optimisé d'applications de TI (cf. les conclusions du paragraphe 1.4.4) puisque, en règle générale, ils ne fournissent pas un formalisme de spécification des applications suffisamment proche des préoccupations des programmeurs. En imposant au programmeur de déterminer explicitement le parallélisme potentiel des applications, ces outils restent essentiellement dédiés à un public de spécialistes habitués aux spécifications parallèles. Pour ce type de public, de tels outils conduisent à des performances d'implantations satisfaisantes tout en diminuant les temps de développement.

Ces constats montrent une fois de plus l'intérêt d'avoir un modèle suffisamment abstrait pour la description des applications. Idéalement, la spécification d'une application ne doit contenir aucune annotation concernant une quelconque forme de parallélisme et de fait, elle ne doit pas être plus complexe à réaliser que dans le monde séquentiel. Toutefois, si ce style de programmation peut présenter des attraits indéniables pour les programmeurs, il faut constater que la tâche des compilateurs devant paralléliser les applications ainsi spécifiées est alors difficile à réaliser puisqu'ils doivent extraire et exploiter le parallélisme implicite des programmes. Une conséquence

directe est alors la difficulté à satisfaire les contraintes temps réel. De plus, si l'extraction du parallélisme implicite des applications est réalisée avec une granularité trop fine, la phase de placement-ordonnancement est alors prohibitive du fait de l'explosion combinatoire des possibilités de placement.

De ce fait, un modèle de programmation parallèle, pour être utilisable et utilisé, se doit de concilier à la fois ces deux notions d'**abstraction** et d'**efficacité** des applications. A ces deux notions, il est possible d'associer six critères qu'un modèle dit idéal se doit de respecter[ST98]. Pour satisfaire les exigences d'abstraction et d'efficacité, un modèle doit donc posséder les propriétés suivantes :

① **Facilité de programmation** : Quel que soit le domaine, la programmation d'applications est une tâche complexe. Aussi, un modèle de programmation doit cacher autant que possible les détails bas niveau liés à la mise en œuvre effective du parallélisme sur l'architecture cible.

② **Facilité de compréhension** : Un modèle de programmation doit être facile à comprendre sinon son utilisation ne peut être large et reste limitée à un public restreint de spécialistes. Si un modèle est capable de masquer la complexité de programmation (illustrée au point 1) et de fournir une interface de développement simple et conviviale, alors celui ci a de fortes chances d'attirer bon nombre de programmeurs désirant accroître les performances de leurs applications.

③ **Indépendance de l'architecture** : Un modèle se doit d'être décorrelé de toute architecture cible sur laquelle le programme doit s'exécuter. En effet, l'expérience montre que les applications possèdent une espérance de vie largement supérieure à celle des machines. A chaque changement — même minime — de l'architecture, il est très coûteux d'effectuer une phase de re-développement plus ou moins profonde de l'application. Aussi, un modèle indépendant du matériel offre des opportunités de portabilité des applications non négligeables. Ajouté à cela, l'abstraction vis à vis de la plate forme cible facilite la programmation. Il n'est plus nécessaire de maîtriser les caractéristiques de l'architecture pour effectuer du développement d'applications parallèles. Ce facteur contribue fortement à un élargissement du champ des utilisateurs potentiels.

④ **Mesures prédictives** : L'implantation d'une application sur une architecture parallèle a généralement pour premier objectif une réduction du temps d'exécution. Or, dans la majorité des cas, la validation ou non de ce but n'est possible que lors de l'exécution effective du programme.

Un modèle de programmation parallèle doit donc fournir un ensemble d'outils de mesures prédictives autorisant l'évaluation précise des performances avant toute implantation réelle. Cette mesure prédictive facilite de même le développement de nouveaux algorithmes ce qui ne pourrait être rapidement fait avec une méthodologie classique de type choix-développement-exécution-vérification.

⑤ **Outils de développement** : Les quatre premiers points discutés impliquent qu'il existera un large fossé entre la spécification de l'application fournie par l'utilisateur et les informations nécessaires à l'exécution du programme sur l'architecture. Ces informations manquantes concernent, par exemple, la décomposition en tâches, le placement-ordonnancement, les communications au niveau logiciel et le nombre de processeurs, le réseau d'interconnexion au niveau matériel. Il est donc nécessaire d'associer à tout modèle de programmation un ensemble d'outils intégrés effectuant la transformation de la spécification minimale utilisateur en une représentation détaillée et implantable des programmes.

⑥ **Efficacité des applications** : Enfin, un modèle se doit d'être efficacement implantable sur une ou plusieurs architectures parallèles. Le but n'est pas forcément d'obtenir la meilleure implantation possible pour la machine cible mais d'établir un compromis idéal entre performance et facilité de développement. En effet, il est indéniable qu'un programmeur expérimenté obtiendra toujours de bien meilleures performances que celles que pourrait donner le plus sophistiqué des outils de parallélisation, mais au détriment d'un coût de développement prohibitif.

Les six critères, que nous venons de présenter, sont relativement exigeants et difficilement compatibles dans le sens où il est pratiquement impossible de les satisfaire globalement. En effet, il apparaît que les notions d'abstraction (critères 1 à 3) facilitent le travail de l'utilisateur mais décalent, en contrepartie, la tâche de parallélisation vers les outils de compilation (critère 5). De même, la portabilité des applications, conséquence directe du critère 3, permet de laisser inchangées les phases de spécification des programmes mais impose au programmeur système une lourde tâche de modification des outils en cas de changement d'architecture. Enfin, le critère 4 permet de déterminer, avant toute implantation, les performances attendues des algorithmes. Ces mesures obtenues analytiquement sont difficilement compatibles avec un modèle totalement indépendant de la cible puisque la phase de modélisation du comportement de l'application nécessite l'utilisation de

caractéristiques liées à l'architecture : nombre et type de processeurs, réseau de communication, vitesse des liens, etc.

En conclusion, dans le cas général, définir un modèle de programmation satisfaisant l'ensemble des six critères mentionnés semble difficile voire utopique en raison de la multitude des domaines d'application candidats à la parallélisation — chacun ayant ses propres spécificités vis à vis de la parallélisation — et du nombre important d'architectures dédiées pouvant exécuter ces applications. *La restriction à un domaine particulier (*TI *dans notre cas) et à une classe d'architectures donnée (machine de type MIMD à mémoire distribuée)* constitue une moyen pragmatique permettant de cerner les interactions entre les aspects algorithmiques et architecturaux. Dès lors, il est envisageable de définir un compromis entre **facilité de programmation** et **performances des implantations** autorisant ainsi le **prototypage rapide optimisé** des applications.

Le chapitre suivant de ce mémoire décrira les notions et principes de constructeurs parallèles nommés **squelettes de parallélisation**. Ceux-ci, base de l'outil d'aide à la parallélisation qui fait l'objet de ces travaux, apparaîtront comme un compromis pragmatique face aux contraintes soulevées par les critères précédents.

Chapter 2

Les squelettes de parallélisation

2.1 Introduction

Le précédent chapitre a montré — dans le domaine du TI — les difficultés spécifiques de la programmation parallèle, partagée entre les notions antagonistes de spécificité et de généricité. D'un côté, la spécificité se traduit par un très faible niveau d'abstraction offrant des opportunités pour optimiser les performances. En contrepartie, ce niveau d'abstraction conduit souvent à des coûts de développement, de maintenance et de portabilité prohibitifs. A l'opposé, la généricité se traduit par un haut niveau d'abstraction, facilitant le développement et offrant des opportunités de portabilité intéressantes. En contrepartie, de telles méthodologies sont souvent synonymes de faible efficacité.

L'objectif de ce chapitre est de présenter un formalisme de développement d'applications parallèles basé sur le concept des squelettes de parallélisation[Col89]. Ceux-ci, du fait de nombreuses propriétés qui seront décrites ultèrieurement, offrent l'opportunité d'équilibrer la balance entre spécificité et généricité tout en s'appuyant sur des fondements à la fois théoriques et pratiques.

La suite de ce chapitre se décompose donc de la manière suivante. Premièrement, nous débuterons par la définition et l'énoncé des propriétés générales des squelettes de parallélisation, montrant ainsi l'adéquation entre leurs caractéristiques et les besoins d'abstraction et d'efficacité. Deuxièmement, le chapitre se poursuivra par un état de l'art détaillé des diverses méthodologies reposant sur les squelettes. Enfin, forts de cette étude, nous définirons et présenterons brièvement quelques squelettes, associés à notre domaine d'applications, qui constitueront les briques de base de notre

outil d'aide à la parallélisation.

2.2 Définition et propriétés

D'un point de vue très général, une application parallèle est décomposable en deux entités distinctes mais intimement liées. On trouve d'une part les fonctions séquentielles de calcul liées aux différentes étapes algorithmiques de l'application et d'autre part divers mécanismes de coordination regroupant et liant ces fonctions de calcul. Une application parallèle est donc un ensemble constitué de **fonctions de calcul séquentielles** et d'un **harnais de communication**. C'est généralement dans cette deuxième partie de code que l'on prend en compte les caractéristiques de la machine cible (allocation des ressources, synchronisation, communication, etc.).

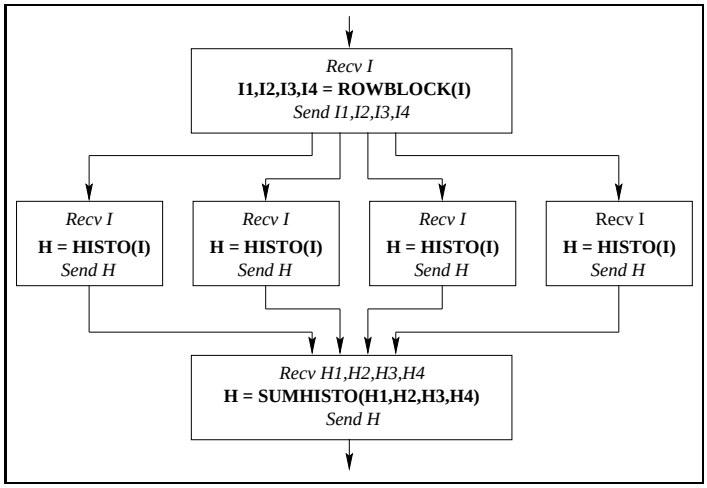


Figure 2.1: Exemple de harnais de communication

L'examen *a posteriori* d'applications parallélisées "à la main"[1] montre que l'exploitation du parallélisme (à travers la mise en œuvre des mécanismes de coordination des calculs) n'utilise qu'un ensemble limité de techniques et de schémas de parallélisation récurrents. Cette constatation est particulièrement évidente dans le cas d'algorithmes de TI bas niveau pour lesquels le résultat final est obtenu par fusion de résultats intermédiaires calculés sur un ensemble de domaines issus d'une partition initiale. Le harnais de communication utilisé possède alors toujours la même structure illustrée sur la

[1]Par exemple, dans le corpus d'applications développées pour la plate-forme Transvision au LASMEA.

figure 2.1. Cet exemple décrit succinctement la parallélisation d'un algorithme de calcul d'histogramme des niveaux de gris de l'image où *rowblock* est la fonction gérant la partition des images en bandes, *seqhisto* une fonction séquentielle de calcul d'histogramme et *sum_histo* somme les histogrammes calculés sur chaque bande.

A partir de là, on peut être tenté de franchir un niveau d'abstraction et d'encapsuler ce couple (fonctions séquentielles-harnais de communication) sous la forme d'un constructeur générique réutilisable et paramétrable par les fonctions de calcul spécifiques à une application donnée. Formalisée par Cole[Col89] et Skillicorn[Ski90], cette approche débouche sur la notion de **squelettes de parallélisation**. Un squelette est donc une *spécification incomplète* d'une forme de parallélisme commune à un grand nombre d'applications, que le programmeur va *spécialiser* avec ses fonctions de calcul séquentiel. Pour ce programmeur, l'implantation du squelette sur une plate-forme est complètement cachée : les squelettes encapsulent tous les aspects — placement, communications, synchronisations — relatifs à l'expression d'une forme de parallélisme. En un sens, ils sont à la programmation parallèle ce que la programmation structurée est à celle reposant sur l'utilisation de instructions *goto/label*[BDP93].

Cette encapsulation des détails relatifs au parallélisme offre des propriétés extrêmement intéressantes. Premièrement, le programmeur d'applications voit son travail de parallélisation fortement s'amoindrir puisqu'il n'a plus à traiter les aspects bas niveau d'implantation. Le travail de parallélisation est dès lors limité au **choix** et à l'**instanciation** des squelettes en dehors de toutes considérations sur les caractéristiques de la machine.

Deuxièmement, l'implantation d'un squelette sur une architecture donnée, étant réalisée une fois pour toute, peut être précisément étudiée et optimisée par le programmeur système garantissant ainsi une grande **efficacité**.

Troisièmement, étant donné que le programmeur d'applications voit son travail réduit au développement de fonctions de calcul séquentiel (par exemple dans un langage impératif classique comme le C), une plus grande **portabilité** des applications est garantie. En cas de changement architectural, voire de migration vers une autre plate-forme, le travail de réimplantation est limité au portage des squelettes par le programmeur système[2], la spécification des applications demeurant inchangée.

Quatrièmement, l'implantation d'un squelette sur une architecture étant

[2]Notons tout de même que ce portage n'est pas forcément trivial mais qu'au moins, celui-ci n'est effectué qu'une seule fois.

parfaitement connue, il est envisageable de modéliser son comportement et d'en déduire un **modèle** analytique de performances paramétré à la fois par les caractéristiques matérielles (nombre de processeurs, vitesse des liens, etc.) et par les caractéristiques algorithmiques (temps d'exécution des fonctions de calcul, type et taille des données, etc.).

En résumé, les squelettes de parallélisation offrent des propriétés de spécification haut niveau permettant de concilier des exigences d'**abstraction** et d'**efficacité** du code. Ils répondent à la plupart des exigences d'un modèle idéal de programmation parallèle énoncées au chapitre précédent à savoir facilité de programmation, indépendance vis à vis de l'architecture, efficacité des implantations et prédiction des performances. Un outil de développement basé sur ces constructeurs génériques semble donc naturellement bien placé pour satisfaire nos objectifs de **prototypage rapide** d'applications sur architecture dédiée.

Toutefois, malgré ces qualités, il subsiste une restriction majeure à leur utilisation en tant que modèle de référence en programmation parallèle. Ils imposent en effet au programmeur de construire ses applications à partir d'une collection *finie* de constructeurs et rien ne peut garantir que cette collection de constructeurs permettra de couvrir *toutes* les applications possibles. La profusion de propositions, mise en évidence dans [Cam96] par exemple, montre clairement les difficultés rencontrées pour définir une "collection idéale et polyvalente" de squelettes. Même en supposant qu'il soit possible de produire une telle base "idéale", l'exploitation de celle-ci passerait inévitablement par le développement d'une méthodologie de programmation au sein de laquelle toute application pourrait être spécifiée naturellement sous la forme d'une composition des squelettes composant la base. Du fait de ces inconvénients, l'applicabilité des squelettes en tant que modèle général de programmation demeure incertaine.

Cependant, nous pouvons remarquer que les difficultés liées à la définition d'une collection de squelettes peut être amoindrie si l'on se place délibérément dans un domaine d'applications restreint. En effet, nous avons déjà remarqué que la mise en œuvre du parallélisme au sein des applications — en particulier dans le cas du TI — n'est basée que sur un nombre restreint de schémas de parallélisation propres au domaine envisagé. Dès lors, la définition d'une collection de squelettes peut être réalisée de manière ascendante à partir d'une part de l'analyse *a posteriori* d'applications de TI implantées manuellement et d'autre part de l'expérience accumulée par les programmeurs œuvrant dans le domaine.

Cette restriction de la classe d'applications autorise alors le développement d'une méthodologie de programmation basée sur les

squelettes de parallélisation facilitant nos objectifs de prototypage rapide d'applications.

Avant de décrire cette approche, on présente au paragraphe 2.3 une revue des méthodologies de programmation parallèle fondées sur les squelettes.

2.3 Revue des méthodologies fondées sur les squelettes

2.3.1 Introduction

Les paragraphes suivants présentent une revue des méthodologies de programmation parallèle basées sur les squelettes. Notre objectif n'est pas de parcourir exhaustivement les projets actuels de la communauté œuvrant dans ce domaine. Aussi, ne seront relatés que les travaux qui, à nos yeux, sont les plus aboutis. De fait, cinq méthodologies seront décrites plus précisément :

① Les travaux de Cole sont, du point de vue historique, les premiers relatifs à l'utilisation des squelettes dans le développement d'applications parallèles.

② Les squelettes homomorphiques du modèle BMF (Bird-Meertens Formalism) ont constitué au début des années 90 un important axe de recherche et ont largement contribué à la reconnaissance des squelettes de parallélisation.

③ Les travaux de Darlington *et al.* font partie des travaux les plus avancés en matière de développement parallèle fondé sur les squelettes spécifiés au sein des langages fonctionnels.

④ Les travaux de Michaelson *et al.*, similaires à ceux de Darlington, sont plus dédiés au TI et, à ce titre, constituent pour nous une base d'étude privilégiée.

⑤ Enfin, P^{3}L est un système de programmation parallèle basé sur les squelettes tout en reposant, à l'opposé des méthodologies précédentes, sur des langages impératifs.

Le lecteur intéressé par ces projets ou d'autres non cités ici peut obtenir de plus amples informations en consultant une bibliothèque en ligne[3] des projets de recherche sur les squelettes de parallélisation.

[3]http://www.dcs.ed.ac.uk/home/mic/skeletons.html

2.3.2 Les travaux de M. Cole

Historiquement, M. Cole fut un des premiers à formaliser la notion de squelette de parallélisation. Celui-ci propose un environnement de développement dont les briques de base sont constituées par un ensemble restreint de squelettes pré-définis. Ces squelettes peuvent s'exprimer sous la forme de fonctions d'ordre supérieur[4] (*fos*) au sein d'un langage fonctionnel[5][Col88].

La tâche du programmeur est limitée d'une part au choix de ces différents squelettes pour exprimer le parallélisme de son application, et d'autre part au développement du code spécifique des fonctions de calcul dans un langage séquentiel classique.

Cole propose un ensemble de quatre squelettes pour lesquels ont été définis des modèles de performances et une implantation sur un réseau à deux dimensions de Transputers[Col87] :

- **Fixed Data Divide and Conquer** (FDDC) est une version restreinte des stratégies classiques de type Divide-and-Conquer dans lesquelles tout problème ne pouvant être résolu directement est divisé récursivement en sous tâches plus aisément évaluables (cf. les opérateurs *Divide* et *Solve* de la figure 2.2). L'idée retenue est alors d'exploiter le parallélisme potentiel de résolution des différents sous-problèmes générés. L'implantation d'une telle forme de parallélisme peut s'avérer délicate du fait de l'impossibilité de prédire le degré de division. Aussi, Cole propose une restriction du schéma DC en imposant de borner statiquement le nombre de divisions récursives.

- **Iterative Combination** (IC) effectue une réduction sur un ensemble d'objets par fusion des paires d'objets répondant à un prédicat de combinaison. Le squelette IC effectue itérativement la combinaison en parallèle des paires d'objets présentant la plus forte valeur de la règle de fusion. L'arrêt de l'exécution survient soit lorsqu'il ne subsiste plus aucune paire, soit lorsque les objets restants ne sont plus fusionnables au sens du critère.

- **Cluster** (C) est une généralisation des deux précédents squelettes. La solution d'un problème peut s'exprimer sous la forme d'une division

[4]Une fonction d'ordre supérieur est une fonction pouvant accepter en argument d'autres fonctions et retourner une fonction comme résultat (cf. chapitre 3).

[5]Dans [Bai94] par exemple, les squelettes développés par Cole sont décrits en paraML[BN93].

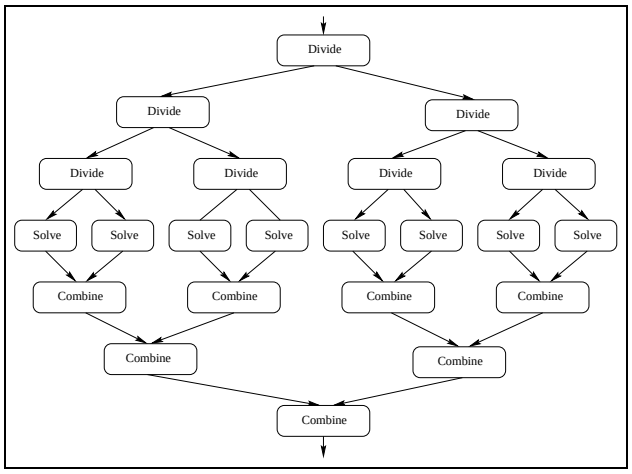


Figure 2.2: Schéma général de type Divide-and-Conquer

récursive des données en groupes distincts qui sont évaluables en parallèle par une stratégie de type FDDC. Le résultat global est ensuite obtenu par l'intermédiaire d'un squelette IC.

➟ **Task Queue** (TQ) est une généralisation des stratégies de type ferme de processeur utilisant une queue de messages globale (cf. figure 2.3). Initialement, les données sont divisées par un processus maître en un ensemble de sous-domaines placés dans la queue de messages. Par la suite, chaque esclave exécute séquentiellement les trois phases suivantes : extraction d'une donnée, traitement et dépôt du résultat associé dans la queue de messages. Cette exécution se poursuit jusqu'à vidage complète de la queue de messages.

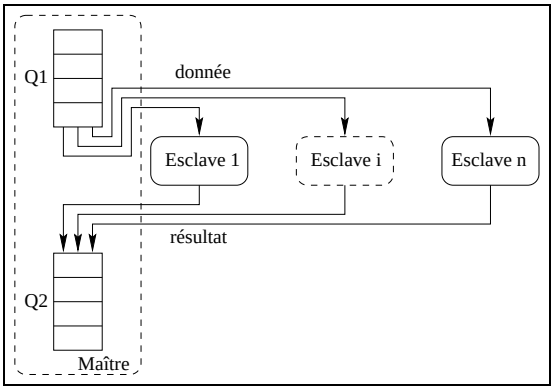


Figure 2.3: Principe du squelette *Task Queue*

Les travaux originaux de Cole ont permis de définir la notion de squelette de parallélisation et d'envisager le développement d'un modèle de programmation parallèle reposant sur ces constructeurs génériques. Les quatre exemples décrits par Cole montrent le besoin de formaliser certaines stratégies de parallélisation courantes telles que Divide-and-Conquer.

Cependant, ces travaux possèdent des limites importantes. Premièrement, le programmeur est obligé de spécifier son application en n'utilisant qu'un seul squelette. L'imbrication des squelettes n'étant pas autorisée, il est alors difficile voire impossible de pouvoir spécifier certaines applications complexes.

Deuxièmement. l'approche retenue par Cole n'est pas totalement indépendante de l'architecture. Le squelette TQ par exemple utilise une queue de messages globale implémentée sous la forme d'une mémoire partagée entre les processeurs. La prise en compte de tels aspects limite inévitablement la portabilité des squelettes sur d'autres types d'architectures.

Malgré ces inconvénients, les idées défendues par Cole ont fortement influencé le développement de nombreuses méthodologies basées sur les squelettes de parallélisation.

2.3.3 BMF

BMF (Bird-Meertens Formalism) a été développé originellement par R. Bird et L. Meertens dans le but de fournir un modèle de programmation autorisant la dérivation des spécifications des problèmes en des programmes efficaces[Bir87a].

BMF est constitué d'un ensemble de règles et d'opérateurs spécifiques à une classe particulière de données. La plupart des travaux ont été effectués dans le cadre de la théorie des listes[Bir87b] mais on trouve aussi des études relatives aux tableaux[Ban95], aux arbres[GCS94] ou aux graphes[Sin93]. Une théorie associée à un type de données décrit son comportement, c'est à dire qu'elle fournit un ensemble de fonctions opérant sur ce type. De plus, sont définies des règles de transformation spécifiques permettant d'obtenir une représentation plus efficace tout en garantissant la même sémantique.

Afin d'illustrer cette notion, considérons la théorie des listes. Par exemple, on trouve comme fonction de base associée à cette théorie l'opérateur *map* (noté $*$) permettant l'évaluation concurrente d'une fonction f sur chacun des éléments constituant la liste :

$$f * [a, b, c, \cdots, z] = [fa, fb, fc, \cdots, fz] \tag{2.1}$$

De même, est défini l'opérateur *reduce* (noté /) effectuant une réduction des éléments de la liste par l'intermédiaire de la fonction $\oplus$:

$$\oplus/[a, b, c, \cdots, z] = a \oplus b \oplus c \oplus \cdots \oplus z \tag{2.2}$$

BMF comporte également un ensemble de lois de transformation. A titre d'exemple, on peut citer les règles de distributivité des opérateurs *map* et *reduce*. Supposant que l'opérateur . représente la composition de fonctions et :: la concaténation de listes, nous pouvons écrire :

$$(f.g)* = (f*).(g*) \tag{2.3}$$

$$\oplus/(x :: y) = (\oplus/x) \oplus (\oplus/y) \tag{2.4}$$

L'utilisation récursive de l'ensemble des règles associées à un type de donnée permet éventuellement d'extraire, à partir d'une spécification originale, un plus important parallélisme potentiel conduisant à une meilleure efficacité des programmes.

Les opérateurs de BMF tels que $*$ et / contiennent un fort parallélisme implicite et sont vus comme des squelettes de parallélisation. Skillicorn, dans [Ski92] et [Ski93], préconise l'utilisation de BMF comme modèle de programmation parallèle. BMF, en effet, est suffisamment abstrait pour permettre des spécifications d'applications décorrellées des caractéristiques architecturales. Par ailleurs, il est possible d'associer à chacun des opérateurs génériques des modèles de performance[SC94]. Ces caractéristiques permettent à BMF de satisfaire simultanément des exigences méthodologiques de prédictabilité, efficacité et portabilité.

En contrepartie, spécifier une application complexe sous la forme d'une combinaison de squelettes *map* et *reduce* peut être relativement complexe et nécessiter une importante étape de reformulation algorithmique.

2.3.4 Les travaux de J. Darlington *et al.*

L'approche mise en œuvre par l'équipe de J. Darlington à Imperial College est une approche classique que l'on retrouve dans la majeure partie des travaux actuels sur les méthodologies fondées sur les squelettes.

L'introduction des squelettes au niveau langage est effectuée directement sous la forme de fonctions d'ordre supérieur au sein d'un langage fonctionnel. Le choix d'un langage fonctionnel permet d'obtenir une séparation distincte entre la sémantique *déclarative* ("quoi") et *opérationnelle* ("comment") des squelettes[DFH+93].

La sémantique déclarative, exprimée au sein du langage, est unique et indépendante de toute spécificité matérielle. Cette propriété est vitale dans le sens où tout programme peut alors être prototypé rapidement sur n'importe quelle machine séquentielle prouvant ainsi son comportement algorithmique correct avant toute implantation parallèle.

A l'opposé, chaque squelette possède une ou plusieurs implantations parallèles sur un ensemble de plates-formes tirant parti des spécificités de l'architecture et encapsulant tous les détails liés à l'exploitation du parallélisme sur cette architecture. En principe, tout squelette est implantable sur toute architecture mais, naturellement, les implantations des squelettes peuvent se révéler plus ou moins efficaces selon les architectures.

A chaque squelette on associe donc une définition fonctionnelle et un ensemble de n définitions opérationnelles pour chacune des m plates-formes cibles sur lesquelles le squelette possède une implantation.

Les fonctions de calcul des applications étant séquentielles, le seul parallélisme des programmes provient de l'utilisation des squelettes. Pour une architecture donnée, le comportement parallèle des squelettes est parfaitement défini et connu. Ceci permet d'associer à tout couple squelette-machine un modèle de performances[DGT93].

Enfin, l'utilisation de règles de transformation et d'équivalence inter-squelettes favorise l'efficacité et la portabilité des applications. Une application comprenant un squelette n'ayant pas d'implantation facile et/ou efficace sur une machine donnée peut alors être aisément réécrite à l'aide de squelettes équivalents possédant un comportement connu et/ou efficace[DFH+93].

Darlington propose un ensemble de cinq squelettes :

- **PIPE** englobe les stratégies de parallélisme de type pipeline (cf. figure 2.4),

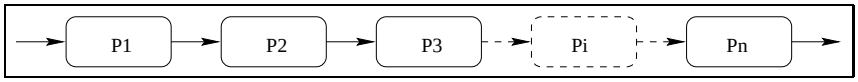


Figure 2.4: Schéma général de type Pipeline

- **FARM** encapsule les formes simples de parallélisme de données dans lesquelles une fonction est appliquée concurremment sur les éléments d'une liste de données,

- **DC** est une stratégie de type Divide-and-Conquer telle que nous l'avons définie auparavant (cf. figure 2.2),

- **RaMP** (Reduce-and-Map-over-Pairs) est dédié aux systèmes dans lesquels tout objet peut potentiellement interagir avec n'importe quel autre objet. Le but d'un tel squelette est de calculer itérativement ces interactions et de combiner les résultats pour réactualiser les objets. Il est similaire au squelette IC présenté précédemment.

- **DMPA** (Dynamic-Message-Passing-Architecture) est un squelette autorisant des communications dynamiques par passage de message entre les différents processus le constituant.

La mise en œuvre et l'utilisation de ces squelettes sont décrites dans [DGT93] et [DT95]. La sémantique déclarative des squelettes et la spécification des applications sont décrites en Hope$^+$[Per88]. Par exemple, une application de lancer de rayons est présentée. La spécification originale repose sur un squelette **RaMP**. Elle est ensuite transformée en deux représentations équivalentes — la première exploitant un squelette **FARM**, la deuxième un squelette **PIPE** — qui ont toutes deux une implantation efficace. L'exécution des applications permet de vérifier la bonne adéquation entre les résultats temporels mesurés et ceux prédits par les modèles.

L'inconvénient majeur de la méthodologie présentée ici est qu'elle repose sur le langage fonctionnel Hope$^+$ aussi bien pour la spécification des squelettes que pour la programmation des fonctions de calcul spécifiques à l'application. Or, la majorité des d'applications développées par les programmeurs sont écrites avec des langages plus "conventionnels" (Fortran, C par exemple). De fait, d'autres travaux ont été également effectués à Imperial College dans le but de fournir un langage de programmation parallèle regroupant les aspects fonctionnels au niveau des squelettes et les aspects impératifs au niveau des calculs. Cela a conduit au développement de SCL[6][DGTY95a][DGTY95b]. SCL effectue une distinction claire entre les aspects haut niveau relatifs à la mise en œuvre du parallélisme et ceux bas niveau liés à la programmation des fonctions séquentielles de calcul développés en langage impératif. A l'opposé de la précédente méthodologie dans laquelle les squelettes opèrent sur des types de données conventionnels, SCL introduit une nouvelle classe de données distribuée nommé **ParArray**. Les squelettes associés à SCL opèrent donc sur ce type de données parallèles. On trouve trois catégories distinctes de squelettes :

- les squelettes de *configuration* sont ceux de plus bas niveau et ont pour tâche majeure la gestion du type **ParArray**. Ils permettent la création

[6]Structured Coordination Language.

des tableaux parallèles à partir de données séquentielles (squelette **partition**), la localisation des données distribuées les unes par rapport aux autres (squelette **align**), la distribution des données (squelette **distribution**), la reconstruction des tableaux séquentiels (squelette **gather**), etc.

- à un niveau supérieur, les squelettes *élémentaires* effectuent les tâches de parallélisme de données. On retrouve les opérateurs classiques tels que **map** et **fold** par exemple mais opérant désormais sur des données distribuées de type **ParArray**,

- enfin, les squelettes de niveau supérieur regroupent les schémas de parallélisation classique. On peut citer les squelettes **farm**[7], **SPMD** et **MPMD** réalisant respectivement l'abstraction des modèles Single Program Multiple Data et Multiple Program Multiple Data, **IterUntil** effectuant des itérations d'un calcul jusqu'à vérification d'une condition de test.

De nombreux exemples d'applications de calcul numérique sont présentés dans [ADG+96] montrant l'intérêt d'une telle approche. L'implantation des programmes décrits est réalisée à l'aide de Fortran-S[DGT+94], utilisant Fortran comme langage impératif de développement des fonctions séquentielles de calcul.

En conclusion, l'avantage majeur de SCL est la possibilité d'utiliser un langage impératif pour développer les fonctions de calculs de l'application. Cela facilite également le portage parallèle d'applications séquentielles déjà existantes. En contrepartie, l'utilisation des squelettes bas niveau de gestion des types **ParArray** peut constituer un obstacle important quant à la diffusion large d'une telle méthodologie auprès d'un public inexpérimenté.

2.3.5 Les travaux de G. Michaelson *et al.*

Le Computer Vision Group de G. Michaelson à l'université Heriot-Watt d'Edimbourg possède une expérience significative dans le domaine des langages fonctionnels et de leurs applications à la parallélisation des problèmes de vision artificielle. Les travaux réalisés ont pour but la mise en place de méthodologies et d'environnement de programmation, reposant sur le concept des squelettes de parallélisation, permettant le prototypage rapide et l'implantation parallèle automatique des applications de TI.

[7] Identique au squelette de même nom défini auparavant

De manière similaire aux projets précédemment cités, les squelettes constituent le moyen unique d'expression du parallélisme des applications. Ceux-ci sont des fonctions d'ordre supérieur "classiques" développées en SML[8][Pau91]. On trouve des squelettes déjà décrits auparavant tels que **map** et **reduce** mais également **filter** qui permet la sélection d'éléments répondant à un critère parmi une liste et **compose** qui autorise la composition de fonctions. Il existe également un ensemble de squelettes (**filtermap**, **mapfilter**, **foldmap**, etc.) représentant des variations de compositions des premiers[Bra94b]. La spécification des applications en SML repose donc sur cet ensemble restreint de squelettes avec pour paramètre le code spécifique des fonctions de calcul propres au problème. Dans [MSW95], des exemples de spécification d'applications sont décrits couvrant les trois niveaux (bas, moyen et haut) du TI.

Cependant, les squelettes ne constituent que des *indications* de parallélisme potentiel au sein des applications décrites en SML. L'exploitation efficace de ce parallélisme est décidée par le compilateur sur la base d'une prédiction de performances. En effet, l'implantation résultante des spécifications peut éventuellement négliger l'expression d'une forme de parallélisme si son exécution n'est pas efficace. Pour cela, des modèles de performances spécifiques aux squelettes associés à une phase d'instrumentation réalisée sur des données types conduisent à une extraction du *parallélisme utile* des applications. Le *parallélisme utile* est défini comme le sous-ensemble du *parallélisme potentiel* dont l'implantation sur une architecture cible conduit à une accélération des traitements[Bra92]. Ajouté à cela, des relations d'équivalence et de transformation des squelettes favorisent similairement l'obtention du meilleur parallélisme utile. Dès lors que le parallélisme implicite est jugé exploitable, le harnais d'implantation associé est instancié avec le code des fonctions utilisateur. Dans le cas contraire, le compilateur produit un code séquentiel.

La mise en œuvre de ces caractéristiques a conduit au développement d'un outil de parallélisation nommé *SkelML*[Bra93][Bra94a][Bra94b]. A partir d'une spécification purement fonctionnelle, *SkelML* effectue l'extraction du parallélisme utile, transforme la spécification en graphe de processus, réalise le placement de ce graphe sur l'architecture et génère le code cible implantable en Occam pour une architecture à base de Transputers[9].

Néanmoins, la méthodologie présentée ici repose sur l'utilisation de squelettes très généraux correspondant aux fonctionnelles standards de SML

[8]Standard-ML.

[9]Des travaux plus récents[MIK97] reprennent cette méthodologie et génèrent par contre du code C utilisant la bibliothèque de communications MPI.

(**map**, **fold**, etc.) et pouvant être appliqués à différents domaines. La restriction au TI permet la création d'autres squelettes, reposant sur ces constructeurs universels, plus spécialisés et mieux appropriés au domaine. Dans [SMW97], un ensemble de quatre squelettes dédiés est présenté. Ils reposent tous sur un schéma général de type Divide-and-Conquer. Chacun des quatre squelettes est en fait une spécialisation plus ou moins fine de ce type de schéma pouvant être appliquée selon le contexte algorithmique. Suivant un ordre croissant de complexité d'implantation, nous trouvons :

- **GD** (Geometric Decomposition) encapsule les schémas dans lesquels une donnée initiale est divisée statiquement en un ensemble de sous domaines de taille égale, chacune des partitions étant par la suite évaluée en parallèle avant qu'ait lieu une phase finale de fusion des résultats intermédiaires.

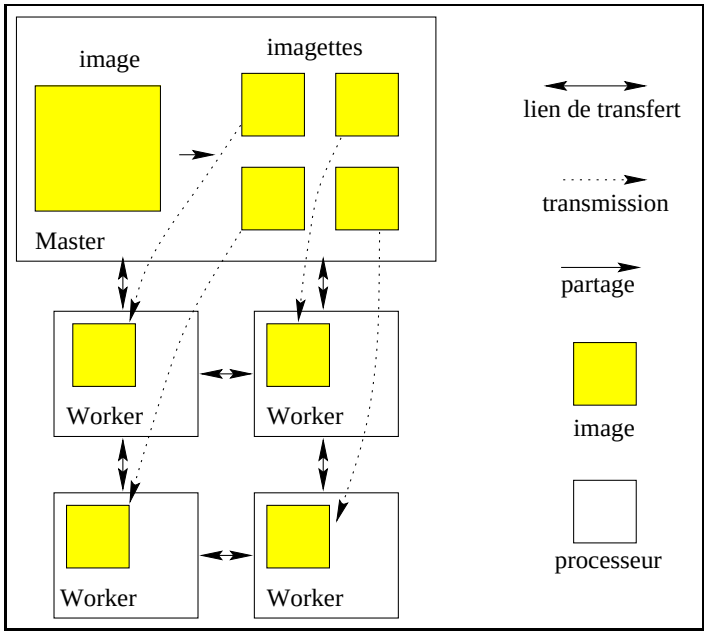


Figure 2.5: Le squelette GD

- **DF** (Data-Farming) correspond au schéma classique de type *ferme de processeurs*[Cha91a][Nic92] utilisé pour équilibrer la charge de calcul lorsque le temps d'exécution des opérations est fonction des données à traiter. Un tel schéma introduit une notion de hiérarchie entre les processeurs. Un processeur appelé *maître* dispose d'un ensemble de donnée à traiter. Un ensemble de processeurs consommateurs nommés *esclaves* est exploité par le processeur *maître*. Celui-ci distribue les données à

tout processeur consommateur inoccupé et accumule les résultats. Les processeurs *esclaves* exécutent successivement les trois étapes suivantes : réception d'une donnée, traitement de la donnée, renvoi du résultat associé puis se remettent en attente d'une nouvelle donnée. La figure 2.6 illustre un tel comportement dans le cas de données de type imagette.

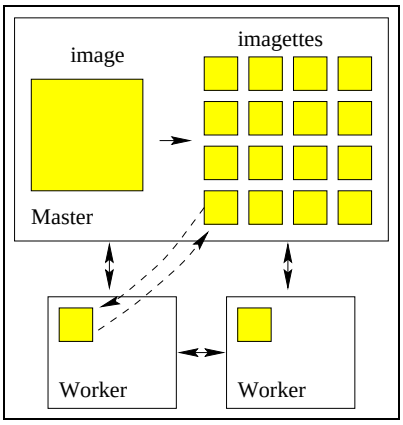


Figure 2.6: Le squelette Data-Farming

➟ **TF** (Task-Farming) est une généralisation du précédent squelette dans le sens où son implantation repose sur le même principe. La différence provient du fait que chaque tâche distribuée à un esclave peut soit être traitée, soit être renvoyée au maître sous la forme de n tâches plus élémentaires (dans le cas de la figure 2.7, $n = 2$).

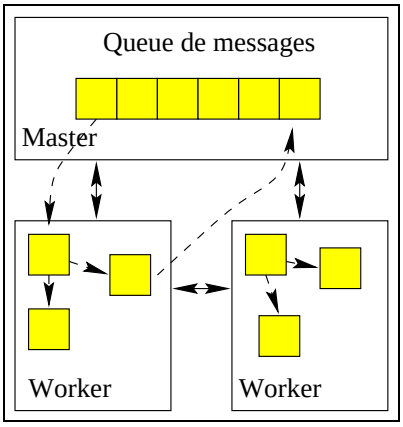


Figure 2.7: Le squelette Task-Farming

➟ **GDIW** (Geometric-Decomposition-with-Interworker-Communications) est une variante du premier squelette dans laquelle des canaux de communications sont installés entre tous les processeurs leur permettant d'échanger des données ou des résultats intermédiaires pendant le traitement de leurs sous-domaines respectifs.

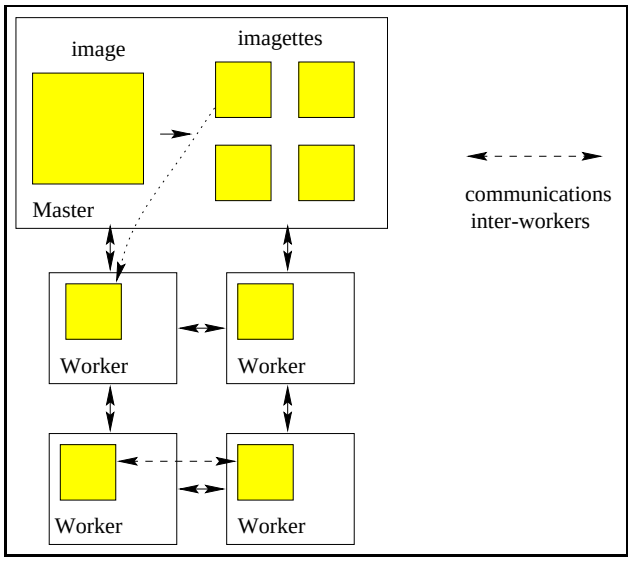


Figure 2.8: Le squelette GDIW

La mise en œuvre et l'utilisation de ces quatre squelettes sont également réalisées dans [SMW97] au travers d'une application complète de reconnaissance d'objets dans des scènes visuelles.

Les derniers travaux de Michaelson *et al.* montrent l'intérêt de définir des squelettes de parallélisation dédiées à un domaine d'applications particulier. Cette approche est une solution possible pour concilier facilité de programmation et efficacité des applications. L'inconvénient majeur d'une telle méthodologie est qu'elle repose uniquement sur les langages fonctionnels y compris au niveau des fonctions de calculs de l'application. Ceci peut constituer un frein :

① à son utilisation par des programmeurs non avertis,

② à la portabilité des applications.

2.3.6 P^{3}L

P^{3}L (Pisa Parallel Programming Language)[Pel93][BCD$^+$97] développé par l'université de Pise et Hewlett-Packard est un système parallèle de programmation basé sur les squelettes de parallélisation.

P^{3}L est un langage *structuré* de *haut niveau* permettant l'expression *explicite* du parallélisme[BDO$^+$95]. Premièrement, P^{3}L est un langage *haut niveau* car la programmation des applications est débarrassée des tâches bas niveau relatives au développement de programmes parallèles : placement et ordonnancement des processus, communications, synchronisations, etc. Deuxièmement, P^{3}L est un langage *structuré* car l'opportunité est offerte au programmeur d'exprimer son application sous la forme d'une composition hiérarchique et éventuellement imbriquée de squelettes. Troisièmement, P^{3}L est un langage *explicite* puisque l'expression du parallélisme est obligatoirement et entièrement effectuée par le programmeur. Celui-ci a donc pour tâche de décrire et de déclarer quels schémas de parallélisation doivent être utilisés. Toutefois, l'instanciation et la mise en œuvre de ces squelettes avec le code spécifique de l'application sont effectuées automatiquement par les outils de compilation.

A l'opposé des méthodologies précédentes reposant sur l'utilisation des langages fonctionnels aussi bien pour la description des programmes que pour la définition des squelettes[10], P^{3}L est basé uniquement sur un langage impératif séquentiel[11] (langage C ou C++). Les squelettes sont exprimés au sein de ce langage et y apparaissent comme des constructeurs sous la forme de mots-clés spécifiques. Ceux-ci sont répartis en trois catégories bien distinctes :

① Les squelettes à parallélisme de données[DPP97] :

- **map** est un squelette permettant l'affectation de sous-domaines d'entrée issus initialement d'une partition à un ensemble de processus concurrents chargés de les traiter.
- **reduce** modélise le schéma de réduction opérant sur un tableau de données et fusionnant deux à deux les paires à l'aide d'une fonction binaire, associative et commutative.
- **comp** est un squelette de haut niveau autorisant la composition de fonctions et de squelettes. Il est utilisé en particulier pour

[10] Sauf pour SCL dans lequel la programmation des fonctions de calcul spécifiques à l'application est réalisée en Fortran.

[11] Récemment, Danelutto *et al.*[DCLP98] ont proposé une reformulation des squelettes de P^{3}L dans le langage fonctionnel CAML.

former le schéma classique **Map&Reduce** à partir des squelettes **map** et **reduce**.

② Les squelettes à parallélisme de tâches[DMO+92] :

- **pipe** modélise un pipeline à plusieurs étages chargés respectivement d'une étape conduisant à l'obtention du résultat final (cf. figure 2.4).
- **farm** représente le schéma classique de ferme de processeurs et possède un processus chargé de distribuer les données (*emitter*), un groupe d'esclaves traitant les données et un processus terminal de réception et de fusion des résultats intermédiaires (*collector*).

③ Les squelettes de contrôle[DMO+92] :

- **sequential** contient les blocs de code séquentiel des fonctions spécifiques de l'application.
- **loop** permet la mise en œuvre d'itérations des modules de calcul effectuées jusqu'à ce qu'une condition de terminaison soit validée.

L'implantation automatique d'applications décrites avec P^{3}L s'effectue grâce à des outils de compilation dédiés. Ceux-ci opèrent en trois phases distinctes[CDF+97] :

- Premièrement, le code source de l'application décrite en P^{3}L est parcouru afin d'extraire l'arbre de représentation contenant de manière hiérarchique les squelettes utilisés dans la spécification. Chacune des feuilles de l'arbre est constituée d'un squelette **sequential** prenant en argument la fonction de calcul de l'application.

- Deuxièmement, chaque squelette est remplacé par un groupe de processus le modélisant (*templates*) issus d'une bibliothèque spécifique. Eventuellement, une phase d'instrumentation du code séquentiel est effectuée afin de prédire le comportement temporel final des squelettes sur l'architecture cible. Cette phase de modélisation peut conduire à une optimisation par transformation du graphe reposant sur des relations d'équivalence inter-squelettes.

- Enfin, partant du graphe de processus placé, le code cible, spécifique à une architecture donnée, est généré. Celui-ci prend en compte les caractéristiques du matériel en terme de processus, de communications ou de synchronisations, etc.

A l'heure actuelle, le code généré est destiné soit à une machine cible à base de Transputers, soit à un réseau de stations de travail communiquant par l'intermédiaire de la bibliothèque MPI.

En conclusion, l'approche retenue dans P^3L se démarque des autres méthodologies présentées précédemment puisqu'elle repose entièrement sur les langages impératifs. Les avantages majeurs de P^3L sont la possibilité de ré-utilisation de code séquentiel déja développé, la définition de modèles de performances pour chaque squelette et l'opportunité de spécifier les applications sous la forme d'une imbrication plus ou moins complexe des squelettes. En contrepartie, la validation de P^3L n'a été réalisée — jusqu'à présent — qu'à travers des exemples simples et non pas en mettant en œuvre des applications parallèles de complexité réaliste.

2.3.7 Bilan des approches existantes

Les principales caractéristiques des méthodologies que nous venons de présenter sont résumées sur le tableau 2.1.

	Cole	Darlington	Michaelson	P^3L
Langage de spécification des applications	Langage Fonctionnel	Langage Fonctionnel	Langage Fonctionnel (SML)	Langage Impératif (C/C++)
Langage de spécification des fonctions séquentielles	Langage Fonctionnel	Fortran	SML	C/C++
Types de données privilégiées		Tableaux	Listes	
Modèles de performances	oui	oui	oui	oui
Machines cibles		Meiko CS, Fujitsu AP1000	Meiko CS, Fujitsu AP1000	Meiko CS, Fujitsu AP1000
Imbrication	non	oui	oui	oui

Table 2.1: Bilan des principales caractéristiques des méthodologies fondées sur les squelettes

2.4 Notre approche

2.4.1 Choix d'une bibliothèque de squelettes

Comme nous l'avons vu au paragraphe 1.2.2, le TI peut être subdivisé en trois catégories distinctes : bas, moyen et haut niveau. Il est difficilement envisageable de constituer une bibliothèque de squelettes pouvant faire face aux traitements issus d'un domaine aussi large. Face à ce délicat problème, notre approche est de restreindre volontairement le champ d'application des squelettes aux seuls algorithmes classés en bas et moyen niveau. Cette limitation délibérée est aisément justifiable par les raisons suivantes :

- Les traitements de bas et moyen niveau constituent de manière évidente les premières étapes de toute chaîne de vision artificielle. De fait, ceux-ci possèdent un fort potentiel de réutilisation.
- Ces mêmes traitements (et principalement ceux de bas niveau) manipulent des quantités de données considérables et sont donc naturellement candidats à des phases de parallélisation.
- Enfin, au moment où ces travaux ont débutés, nous avions en notre possession une expérience conséquente aussi bien dans le développement d'applications de TI bas et moyen niveaux que dans leur parallélisation sur la machine Transvision (par exemple, l'aide à la conduite automobile a constitué un axe de recherche important à travers le projet européen Promotheus sur le véhicule expérimental Prolab[BDHR94]).

Toutefois, malgré cette restriction supplémentaire du domaine d'application, il n'est pas trivial d'effectuer le choix d'une base de squelettes. Le problème majeur à résoudre consiste à équilibrer la balance entre **spécificité** et **généralité**. La spécificité impose un nombre relativement important de squelettes hautement spécialisés tandis que la généralité est plutôt synonyme d'un faible nombre de squelettes d'ordre général.

Premièrement, considérons le cas de la spécificité. Cette approche offre des opportunités d'efficacité puisque les implantations parallèles associées à chacun des squelettes sont "taillées sur mesure". A chaque nouveau problème ne pouvant être exprimé facilement et/ou efficacement sous la forme d'une composition de squelettes déjà existants, il est tentant d'insérer un nouveau squelette dans la bibliothèque. Cette démarche a des répercussions directes aux niveaux système et utilisateur. Du point de vue du système, l'effort requis pour créer le nouveau squelette peut être suffisamment élevé pour rendre cette approche rapidement impraticable. En effet, l'implantation de chaque

nouveau squelette est à étudier pour l'ensemble des plates-formes pouvant le supporter. De plus, le temps consacré au re-développement des implantations des squelettes en cas de changement d'architecture cible est proportionnel au nombre de ces squelettes et peut de fait devenir prohibitif. Enfin, une partie non négligeable des nouveaux squelettes créés est très spécifique à des applications particulières entraînant ainsi un taux d'utilisation faible.

Du point de vue de l'utilisateur, l'augmentation du nombre de squelettes réduit inévitablement leur visibilité. Les développeurs inexpérimentés dans le domaine de la programmation parallèle risquent alors d'être "noyés" sous la pléthore d'opérateurs. L'expérience montre en effet qu'un programmeur maîtrise intuitivement mieux un faible nombre de squelettes à large spectre qu'un grand nombre à spectre plus étroit. Une collection restreinte de squelettes de plus haut niveau, possède donc intrinsèquement un plus grand potentiel de réutilisation. En contrepartie, l'optimisation des performances et le développement de modèles analytiques de performances sont de manière générale plus délicates à obtenir pour des squelettes très généraux.

Dans notre cas, le choix du nombre de squelettes et de leurs caractéristiques est résolu simplement de manière très pragmatique. La bibliothèque de squelettes associés au TI est définie à partir d'une analyse *a posteriori* des applications de TI. Par chance, nous possédons dans le domaine de nombreuses réalisations dont quelques exemples ont été donnés au paragraphe 1.2.1.

Les résultats de cette analyse mettent en évidence quatre classes principales de schémas pouvant être modélisés en squelettes :

- Les schémas dédiés au traitement géométrique des données. Ils représentent les formes les plus simples du parallélisme de données effectuant des transformations de type image → image dans lesquelles on applique un ensemble d'instructions sur chaque pixel de l'image. L'image originale est divisée initialement en sous-domaines réguliers (bandes horizontales ou verticales, imagettes, etc.) par une fonction de découpage spécifique à l'application. Chacune des partitions est alors traitée indépendamment des autres par le même opérateur. Finalement, l'image résultat est reconstruite par concaténation des sous-domaines résultats. Ce type de schémas de parallélisation est utilisé principalement pour des opérateurs de TI bas niveau tels que convolution, seuillage, etc.

- Les schémas dédiés aux phases d'extraction de caractéristiques à partir des images. Dans ce cas là, une stratégie d'implantation similaire à celle décrite précédemment est employée. La différence majeure vient

du fait que chaque résultat partiel ne représente plus une portion de l'image mais un ensemble d'attributs calculés d'où le recours à une fonction spécialisée (et propre à l'application envisagée) de fusion de ces résultats. Ce type de schémas est à la fois utilisé en TI bas niveau (histogramme des niveaux de gris par exemple) et en TI moyen niveau (chaînage de points contour par exemple).

- Les schémas encapsulant des structures de contrôle de type fermes de processeurs opérant soit sur des données (*data farm*), soit sur des tâches (*task farm*). Ceux-ci sont principalement utilisés dans les traitements de moyen niveau (approximation polygonale d'une chaîne de points connexes, division récursive d'images. etc.) lorsque la complexité des traitements est fortement dépendante des données d'entrée.

- Les schémas traduisant la nature itérative des algorithmes de vision. En effet, un grand nombre d'applications embarquées dans des systèmes complexes manipulent non plus des images fixes mais des flots continus de données. Parmi ceux-ci, citons les algorithmes de type prédiction-vérification pour lesquels le traitement de l'image i dépend des résultats issus des images $i-1, \cdots, i-k$. L'application de détection et de suivi de véhicules décrite au paragraphe 1.2.1.2 repose sur ce type de schéma.

La première classe d'algorithmes apparaît comme un cas particulier de la deuxième catégorie puisque la phase de fusion finale est réduite à sa plus simple expression c'est à dire une opération purement géométrique de concaténation d'images.

Deuxièmement, la distinction entre les structures de contrôle de type *data farm* et *task farm* doit être effectuée. Même si les différences entre ces squelettes semblent relativement fines en raison essentiellement de la nature similaire de leurs modèles d'implantation, l'analyse des applications reposant sur de tels schémas montre qu'ils s'adressent réellement à deux classes d'algorithmes séparables.

Ces constations étant effectuées, il apparaît désormais que quatre squelettes élémentaires vont constituer les briques de base de notre bibliothèque de programmation parallèle :

- **SCM** (Split-Compute-Merge) regroupe les schémas des deux premières catégories.

- **DF** (Data-Farming) et **TF** (Task-Farming) représentent les structures de contrôle de type ferme de processeurs opérant respectivement sur des données et des tâches.

- **ITERMEM** (ITERate-with-MEMory) prend en compte la nature continue des flots d'images.

Ces quatre squelettes seront décrits plus précisément dans les paragraphes 2.4.2 à 2.4.5. Mais, auparavant, il est indispensable d'effectuer les remarques suivantes.

Tout d'abord, notre proposition reprend la séparation classique des schémas à parallélisme de données et de contrôle. Cette distinction a déjà été mise en évidence dans des méthodologies décrites précédemment comme P^3L. Dans notre cas, le squelette SCM est classé dans la première catégorie alors que les squelettes DF, TF et ITERMEM sont rattachés à la deuxième famille.

Deuxièmement, notre public cible est celui des programmeurs œuvrant en TI qu'ils soient familiers ou non avec le parallélisme. Ce but ne pourra être atteint que si les squelettes proposés encapsulent des schémas de parallélisation usuels dont la sémantique opérationnelle est aisément compréhensible. C'est le cas par exemple du squelette SCM qui regroupe les formes classiques de parallélisme de données dans lesquelles une même fonction est appliquée sur des partitions différentes des données d'entrée. De même, le squelette ITERMEM possède un comportement relativement simple à comprendre pour un programmeur habitué aux formulations itératives des algorithmes de TI. Par contre en ce qui concerne les squelettes DF et TF, on ne peut que constater que ceux-ci atteignent probablement la limite supérieure de complexité pour les programmeurs peu avertis.

Enfin, la proposition faite des quatre squelettes — à partir de l'analyse d'applications existantes — constitue une première bibliothèque de base. Rien ne peut assurer *a priori* que l'intégralité du TI bas et moyen niveau puisse être couverte par cet ensemble restreint. Cependant, nous avons jugé que la mise en place d'outils de parallélisation reposant dans un premier temps sur ces quatre squelettes entraînerait naturellement un accroissement de notre expertise dans le domaine. Dès lors, si le besoin se fait sentir d'insérer de nouveaux squelettes pour répondre à d'autres problèmes plus spécifiques, rien ne pourra empêcher leur développement.

2.4.2 Le squelette SCM

Le squelette SCM encapsule les stratégies à parallélisme de données dans lesquelles la donnée d'entrée est divisée (*Split*) en un nombre fixe de partitions. Chacun des sous-domaines ainsi généré est alors traité (*Compute*) indépendamment par un processeur. Le résultat final est obtenu par combi-

naison (*Merge*) (selon une stratégie plus ou moins complexe) des solutions intermédiaires.

Ce squelette est relativement proche des squelettes **map** de certaines méthodologies présentées auparavant dans le sens où est effectué un ensemble de calculs identiques sur des données distribuées. Toutefois, le squelette SCM réalise en plus du calcul parallèle, la division des données initiales et la fusion des résultats intermédiaires. Ces caractéristiques supplémentaires le rendent étroitement similaire au squelette GD décrit au paragraphe 2.3.5. Le squelette SCM apparaît également à la fois comme un squelette de configuration et comme un squelette de traitement dans l'approche de Darlington *et al.* décrite au paragraphe 2.3.4.

La figure 2.9 décrit un exemple d'implantation d'un tel squelette sur une architecture à quatre processeurs.

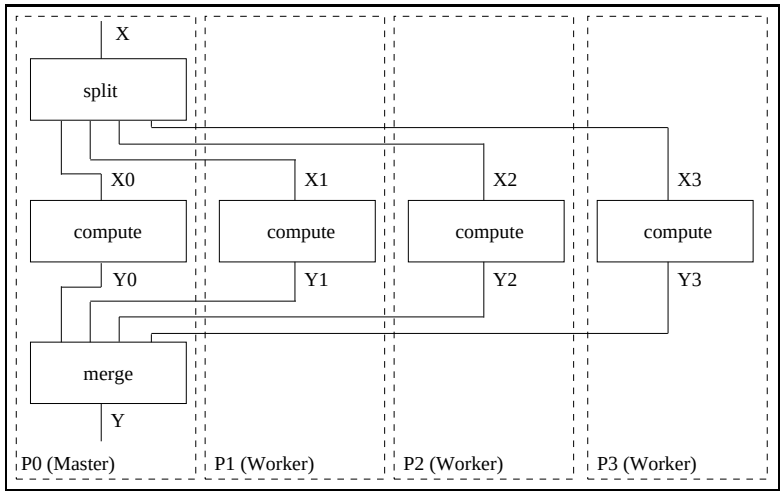


Figure 2.9: Un exemple d'implantation du squelette SCM sur quatre processeurs

La caractéristique principale de son comportement parallèle est sa nature totalement statique. Du côté du processeur maître, on assiste à un enchaînement séquentiel des opérations suivantes :

- génération de n partitions de données par la fonction utilisateur *Split*,
- distribution de $n-1$ paquets aux autres processeurs du réseau,
- application de la fonction *Compute* sur le paquet non distribué,
- collecte des $n-1$ résultats intermédiaires,

- fusion des résultats par la fonction spécifique *Merge*.

Les autres processeurs exécutent, quant à eux, la séquence suivante :

- réception d'un paquet de données,
- traitement local du paquet par la fonction *Compute*,
- envoi du résultat obtenu vers le processeur maître.

Du fait de ses propriétés statiques, ce squelette n'est utilisable que pour des algorithmes caractérisés par une uniformité temporelle des calculs sur chaque partition. L'efficacité d'un tel squelette dépend en effet implicitement de l'équilibre de charge des calculs entre les processeurs. Ce squelette est donc essentiellement dédié aux algorithmes de bas niveau de pré-traitement d'informations (convolutions, filtres, histogrammes, etc.) dans lesquels la durée des traitements dépend uniquement de la taille des images.

2.4.3 Le squelette DF

Le TI abonde d'algorithmes dont la complexité (et donc le temps d'exécution) s'exprime en fonction des données. Par exemple, l'opérateur d'approximation polygonale de chaînes de points connexes évoqué au paragraphe 1.2.1.1 utilise une stratégie récursive de division de courbe dont l'arrêt est conditionné par la distance séparant la courbe et les segments l'approximant. Dans ce cas, il est aisément compréhensible que le temps de traitement dépend de la taille et de la forme de la courbe. L'utilisation d'un squelette SCM — qui suppose une partition de la liste des chaînes de points en n sous listes de taille fixe — peut alors entraîner un important déséquilibrage de charge dont la première conséquence est une chute significative des performances. L'introduction du squelette DF a pour objectif de prendre en compte cet aspect et de gérer dynamiquement la répartition des données sur le réseau de processeurs.

Le modèle d'implantation parallèle associé est une classique ferme de processeurs. Le processeur maître (*farmer*) possède une liste de données qui sont distribuées dynamiquement à un ensemble d'esclaves (*worker*) chargés eux d'effectuer des traitements sur ces mêmes données. L'équilibre de charge est réalisé automatiquement selon le protocole suivant. A chaque fois qu'une donnée est traitée et donc que le résultat correspondant est produit, le maître renvoie immédiatement une nouvelle donnée et accumule le résultat produit. Ce mode de fonctionnement est réitéré jusqu'à ce qu'il ne subsiste plus de données dans la liste. Une telle exécution est illustrée sur la figure 2.10

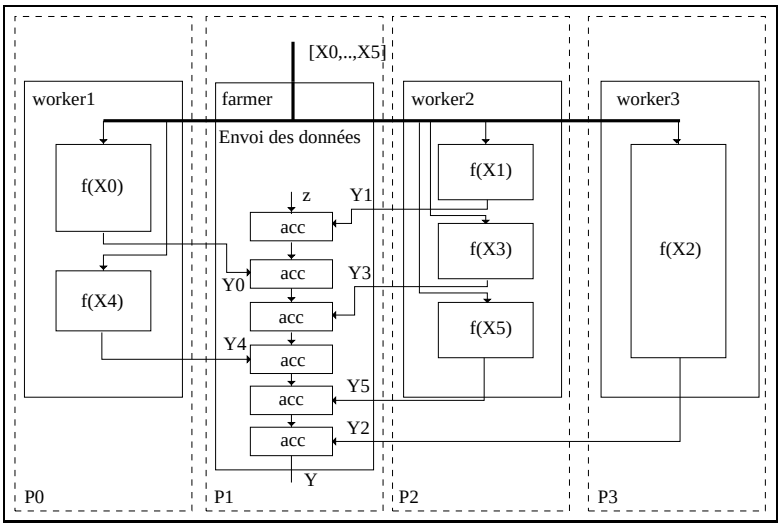


Figure 2.10: Un exemple d'implantation du squelette DF sur quatre processeurs

sous la forme d'une "pseudo" vue statique dans laquelle les processeurs se répartissent sur l'axe horizontal et le temps figure sur l'axe vertical.

Ce cas simple utilise une liste de données comportant 6 éléments (X_0 à X_5). Le *farmer* envoie successivement les données X_0, X_1 et X_2 aux trois *workers* et se met en attente d'un résultat. La première valeur retournée est $Y_1 = f(X_1)$ provenant du *worker* 2. Dès lors, le *farmer* renvoie une nouvelle donnée (X_3) au *worker*, accumule le résultat Y_1 et se remet en attente. Cette même stratégie est appliquée pour tous les autres éléments de la liste.

Cette simulation de comportement révèle deux principales caractéristiques liées au modèle d'exécution dynamique :

- premièrement, du fait des variations de la durée des traitements, l'ordre d'arrivée des résultats provenant des *workers* peut être radicalement différent de celui de distribution des données. Cela implique que la fonction d'accumulation des résultats partiels (fonction *acc* sur la figure 2.10) doit être commutative,

- deuxièmement, au niveau calcul, il n'est pas possible de prédire le nombre d'appels de la fonction de traitement sur chaque *worker*. En moyenne, ce nombre a pour valeur théorique N/n, N et n étant respectivement les nombres des données et de processeurs. L'efficacité du squelette DF n'est donc garantie que dans le cas où N est très largement supérieur à n afin de tirer pleinement parti des possibilités de répartition dynamique des données. De plus, il ne doit pas exister

de donnée dont la complexité temporelle de traitement domine toutes les autres. Pour illustrer cette notion, reprenons l'exemple de la figure 2.10 et supposons que le traitement $f(X_2)$ ait une durée d'exécution deux fois plus longue. Un tel comportement entraîne dès lors un déséquilibrage de charge et une forte séquentialité annihilant tous les gains résultant de la parallélisation des autres données.

Le squelette DF est relativement proche des squelettes **FARM** de Darlington ou **Data-Farming** de Michaelson qui appliquent une fonction de calcul sur les éléments d'une liste. Cependant, dans notre approche, le squelette DF de manière identique au squelette SCM réalise en plus la fusion des résultats intemédiaires.

2.4.4 Le squelette TF

Le troisième squelette introduit ici peur être vu comme une généralisation du précédent, dans lequel le traitement d'une donnée peut éventuellement générer récursivement de nouvelles données à distribuer. Ces deux squelettes utilisent une même stratégie de type ferme de processeurs opérant sur des données. La différence principale, peu évidente au premier abord, provient du fait de leur possibilité de générer ou non de nouvelles données en cours d'exécution. Le squelette DF opère sur un ensemble de données dont le nombre demeure constant tout au long d'une même itération. Ce nombre peut par contre varier d'une itération à l'autre. C'est le cas typique des traitements sur fenêtres d'intérêt qui opèrent sur n_i fenêtres à l'itération i et n_{i+1} à l'itération $i+1$. A l'opposé, le squelette TF opère sur une donnée qui peut éventuellement générer récursivement de nouvelles données en cours d'exécution.

Pratiquement, dans le squelette TF, le maître a pour tâche de distribuer les données à traiter et de collecter les résultats correspondants tout en équilibrant au mieux les charges de calcul. Toutefois, le comportement des *workers* est ici plus complexe dans le sens où la donnée reçue va conditionner le traitement opérant sur celle-ci. En effet, chaque donnée est préliminairement testée par le biais d'un prédicat. A titre d'exemple, nous pouvons évoquer les prédicats d'homogénéité des régions de l'image basés sur des calculs statistiques de moyenne et d'écart-type des valeurs des pixels : une région est déclarée homogène si l'écart-type des pixels est inférieur à un seuil fixé. En cas de succès du prédicat, la région est traitée et le résultat correspondant est ensuite renvoyé au *farmer* pour y être accumulé. Dans le cas contraire, la donnée est retournée au *farmer* qui applique une fonction de division afin de générer récursivement de nouvelles données.

Ce schéma d'exécution est similaire aux stratégies de type *Divide-and-Conquer* des squelettes **FDDC** de Cole et **DC** de Darlington que nous avons décrits auparavant. Il est résumé sur la figure 2.11.

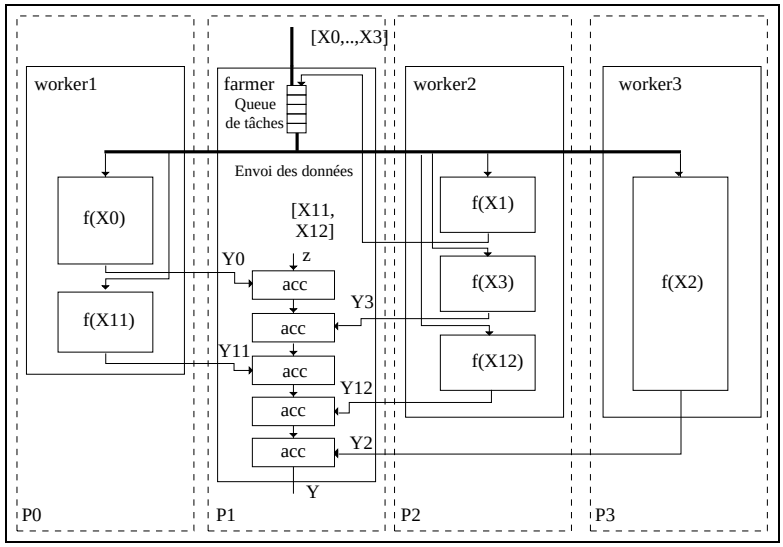


Figure 2.11: Un exemple d'implantation du squelette TF sur quatre processeurs

Ce cas relativement simple d'exécution utilise un ensemble de quatre données ($[X0, X1, X2, X3]$). Seule la donnée $X1$ traitée par le processeur $P2$ ne répond pas au prédicat. Cela conduit donc à la génération de deux nouvelles données nommées respectivement $X11$ et $X12$ qui seront traitées ultérieurement par le réseau de *workers*.

2.4.5 Le squelette ITERMEM

Le squelette ITERMEM peut être vu comme un squelette de haut niveau puisqu'il prend en paramètre aussi bien une simple fonction de calcul qu'un des squelettes précédemment cités. Ce deuxième cas représente la seule situation dans laquelle notre modèle de programmation supporte l'imbrication de squelettes.

Le squelette ITERMEM ne comporte pas en réalité de parallélisme implicite. Il n'est utilisé que lorsqu'il est indispensable de faire apparaître explicitement la notion de flots continus d'images, lorsque les traitements sur l'image i dépendent des précédents résultats obtenus sur les images $i-1$, $i-2$, $\cdots$, $i-k$. De telles opérations sont fréquemment utilisées dans les algorithmes de suivi d'objets dans une séquence d'images. Ceux-ci emploient

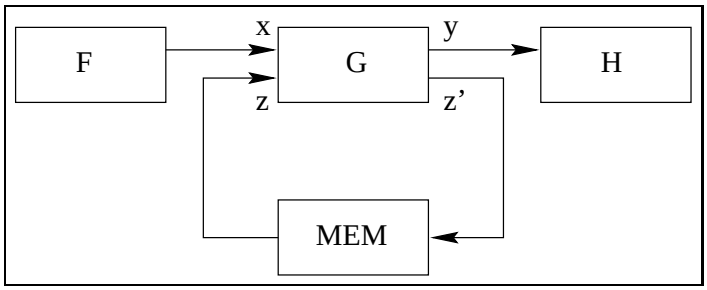


Figure 2.12: Un exemple d'implantation du squelette ITERMEM

alors un modèle de l'état du système qui permet de prédire la position des objets à suivre à chaque nouvelle itération.

Ce squelette nécessite trois fonctions nommées F, G et H et une mémoire notée *MEM*. A l'itération i, la fonction G utilise la donnée x provenant de la fonction F et la donnée z stockée dans la mémoire lors de l'itération $i-1$ et produit les résultats y passé à la fonction H et z' qui est stocké temporairement dans la mémoire afin d'être disponibles à l'itération $i+1$ (cf. figure 2.12).

2.5 Conclusion

Dans ce deuxième chapitre, nous avons présenté et mis en avant le concept de **squelettes de parallélisation**. Ceux-ci sont vus comme des constructeurs génériques de haut niveau exprimant certaines formes classiques et récurrentes de parallélisme. En encapsulant les aspects bas niveau relatifs à l'exploitation d'une forme de parallélisme, les squelettes répondent aux objectifs suivants : **facilité de programmation**, **efficacité des implantations**, **portabilité des spécifications** et **prédictabilité des performances**.

En résumé, ces caractéristiques offrent le moyen de concilier les notions opposées d'**abstraction** et d'**efficacité** du code qui font souvent défaut aux méthodologies classiques de développement parallèle.

Cependant, l'étude de divers travaux de recherche sur les squelettes a mis en évidence la difficulté de définir une base de constructeurs garantissant une couverture totale d'un domaine d'applications. Nous avons montré que nous pouvions extraire et définir un **ensemble restreint** de squelettes dédiés au TI à partir d'une analyse fine des applications de vision artificielle implantées manuellement sur la machine Transvision.

Ce sous ensemble formé de quatre squelettes constitue la base de l'environnement dédié au prototypage rapide d'applications de TI qui fait

l'objet de ce mémoire. Cependant, ce chapitre a uniquement présenté de manière succincte les caractéristiques de ces squelettes et de fait, de nombreux points n'ont pas encore été abordés : définition déclarative des squelettes, implantation parallèle des squelettes, mise en œuvre de modèles de performances, etc.Le premier point concerne l'utilisation et l'insertion des squelettes au sein d'un langage de programmation. Dans l'étude des méthodologies existantes, deux voies principales ont été explorées :

- Les squelettes sont vus comme des constructeurs spécifiques à un langage impératif classique (approche utilisée dans P^3L).
- Les squelettes sont exprimés directement sous la forme de fonctions d'ordre supérieur au sein d'un langage fonctionnel (approche retenue dans les travaux de Darlington et Michaelson par exemple).

Cette deuxième approche fera l'objet du chapitre suivant dans lequel nous montrerons l'intérêt d'utiliser les langages fonctionnels comme langage de spécification et de preuve des applications de TI.

Chapter 3

Les langages fonctionnels

3.1 Introduction

Les premières années de l'informatique ont vu l'émergence de langages reflétant fidèlement la structure interne des machines. Ceux-ci ont été développés dans le but premier de contrôler le comportement des ordinateurs. Cependant, face à la prolifération des ordinateurs, chacun ayant ses propres spécificités, des langages de haut niveau plus éloignés des architectures et de fait, plus proches du raisonnement humain ont été développés. Initiée avec Fortran, cette politique a vu l'émergence de nombreux langages dont certains sont devenus aujourd'hui des standards, alors que d'autres sont par contre tombés en désuétude.

De nos jours, le nombre et la diversité de ces langages sont tels que l'habitude a été prise de les classer par famille reflétant un modèle de calcul ou un style de programmation commun. Dans cette classification, on distingue par exemple les langages dits **impératifs** dont les représentants les plus connus sont Fortran, Pascal ou C. Leur caractéristique première est la notion implicite d'état modifié par une succession d'instructions. A titre d'exemple, l'affectation d'une valeur à une variable est une opération triviale de modification de l'état courant du système. La sémantique des programmes est déterminée par le séquencement des instructions, c'est-à-dire l'ordre dans lequel elles vont modifier l'état.

Par opposition, les langages dits **fonctionnels** (LFs) reposent principalement sur la notion de fonction au sens mathématique du terme. Un programme peut être vu comme une collection de définitions de fonctions dont l'appel à l'une d'entre elles déclenche le calcul voulu. De tels langages reposent donc sur un ensemble d'expressions proches des mathématiques

traduisant des dépendances fonctionnelles. Par rapport aux précédents, les LFs possèdent des propriétés intéressantes largement mises en avant par leurs défenseurs : rapidité d'écriture et concision des programmes, transparence référentielle, etc. lesquelles seront expliquées dans la suite du chapitre.

Dans ce chapitre, nous aurons pour but de dresser un bref aperçu des LFs (et de leurs propriétés) qui constituent, à nos yeux, un moyen particulièrement bien adapté à notre problématique de parallélisation d'applications de TI.

Tout d'abord, nous ferons un bref historique des langages fonctionnels. Cette étude nous permettra également de dresser un inventaire des propriétés des LFs modernes.

Dans un deuxième temps, nous montrerons l'**adéquation** des LFs pour la spécification des applications parallèles basées sur les squelettes. En particulier, nous décrirons comment ceux-ci s'expriment naturellement en LF et nous définirons successivement l'ensemble de nos quatre squelettes spécifiques à la vision en langage Caml.

Enfin, nous consacrerons une dernière partie à un aspect crucial à nos yeux que nous nommons l'**émulation séquentielle**. Cette phase permet la preuve et la vérification des programmes parallèles en dehors de tout contexte d'architecture cible.

3.2 Définition des LFs

3.2.1 Historique

Avant d'évoquer les LFs modernes, il est nécessaire de citer quelques dates clés de la programmation fonctionnelle :

- 1940 : les LFs trouvent leurs origines dans le λ-calcul[Chu41] inventé par Church dans le cadre de travaux sur les fondements des mathématiques. Le λ-calcul contient de manière plus ou moins explicite les concepts principaux des LFs modernes : possibilité d'appliquer une fonction à une autre, notion de fonction curryfiée, théorème de Church-Rosser, etc.

- 1958 : Mac Carthy invente LISP[McC60] dont les programmes possèdent de fortes similarités avec le λ-calcul. Ces travaux étaient motivés par le besoin d'un langage algébrique de traitement des listes. LISP a exercé une influence non négligeable sur le développement des

LFs modernes. Les dialectes de LISP tels que Scheme[RC86] ont une sémantique proche de celle purement fonctionnelle des langages modernes.

➟ 1965 : Landin propose le langage ISWIM[Lan66] (If You See What I Mean) que l'on peut considérer comme le précurseur des langages de la famille de ML.

➟ 1978 : Backus introduit FP[Bac78] qui fut un des premiers LFs à avoir reçu une large attention. Un programme FP est un ensemble d'expressions construites à partir de primitives et d'objets par unique emploi de formes fonctionnelles pré-définies. Une des principales originalités de FP est l'absence de noms de variables, ce qui permet une extrême concision des programmes.

➟ 1978 : Milner propose un langage nommé ML[GMW79] (pour Meta-Language) qui reprend une grande partie des concepts d'ISWIM. Son originalité réside dans l'utilisation du système de typage de Hindley-Milner (cf. paragraphe 3.2.2.2). Le langage ML est basé sur le λ-calcul avec un mode d'évaluation dit par valeur (*i.e.* les fonctions sont appelées une fois que tous leurs arguments sont évalués). Les travaux sur ML ont conduit en 1984 au développement de Standard-ML[Mil84] qui fut un des premiers LFs à être largement diffusé et utilisé.

➟ 1984 : En collaboration avec Milner, le projet *Formel* à l'INRIA conduit au développement et à la distribution de la première version du langage Caml[LVD96]. Il existe à l'heure actuelle deux dialectes de Caml (Objective Caml et Caml Light). Caml est un langage fonctionnel de la famille de ML, disponible publiquement[1].

➟ 1985 : Turner propose le langage Miranda[Tur85] qui reprend le typage de ML et fait un usage généralisé des fonctions d'ordre supérieur (cf. paragraphe 3.2.2.3) et du mécanisme d'évaluation souple[2] Miranda est le résultat de nombreux travaux sur les LFs qui ont vu naître successivement les langages SASL[Tur75], KRC[Tur81] et Hope[BMS80].

➟ 1992 : Le langage Haskell[HPJWe92] reprend la plupart des caractéristiques de SML ou Miranda auxquelles il ajoute des fonctionnalités nouvelles telles que les types de données abstraits, la définition de classes de types ou un système d'entrée-sortie purement fonctionnel.

[1] http://pauillac.inria.fr/caml/

[2] L'évaluation souple consiste schématiquement à attendre que la valeur d'un argument soit absolument nécessaire avant de la calculer.

3.2.2 Propriétés des LFs

3.2.2.1 Transparence référentielle

Un programme impératif est un processus évolutif qui, à partir d'un état initial, exécute un ensemble d'instructions jusqu'à parvenir à un état final qui "contient" le résultat voulu. Toutes ces opérations modifient physiquement le contenu des adresses mémoire représentant l'état courant du système. De fait, la valeur d'une expression à un instant donné peut dépendre non seulement de sa définition syntaxique mais aussi de l'état courant du programme. Une fonction impérative peut très bien, par exemple, modifier des variables globales et par conséquent retourner des valeurs distinctes d'une invocation à l'autre.

Ces **effets de bord** n'ont pas lieu d'être dans le domaine des LFs. Un programme fonctionnel est par définition une fonction mathématique sans état interne qui, appliquée à un ensemble d'arguments, produit un ensemble de résultats. Etant donné que la notion d'état n'existe pas, on est assuré que pour les mêmes arguments, une fonction retournera toujours les mêmes résultats, et ceci, indépendamment du contexte dans lequel la fonction est évaluée.

Cette propriété dite de **transparence référentielle** est un atout considérable puisqu'elle garantit une plus grande lisibilité et fiabilité des programmes. Dans le contexte des architectures parallèles à mémoire distribuée, cette propriété est extrêmement intéressante car elle s'oppose au concept de mémoire globale.

3.2.2.2 Typage et polymorphisme

3.2.2.2.1 Synthèse de types

Depuis ML, la plupart des LFs bénéficient du système de typage de Hindley-Milner[Hin69][Mil77]. Connaissant les types des valeurs de bases et des opérations primitives, le contrôleur de types produit une signature pour chaque expression fonctionnelle en suivant un ensemble de règles de typage. De plus, le type inféré contient le plus petit ensemble de contraintes nécessaires au bon déroulement du programme[3]. Le contrôleur de types détermine le type le plus général pour chaque expression.

Pour illustrer cette notion, prenons quelques exemples de programmes Caml extraits de [WL93].

[3] "Bon déroulement" signifie seulement qu'il n'y aura pas d'erreur de type à l'exécution.

Définir une fonction en Caml est simple et naturel car la syntaxe est très proche des notations mathématiques usuelles. A la définition *"Soit successeur la fonction définie par successeur*$(x) = x+1$*"*, correspond la définition Caml suivante[4] :

```
> let successeur x = x + 1
# val successeur : int -> int
```

Cette définition (introduite par le mot-clé *let*) reçoit comme type inféré, c'est-à-dire comme **signature**, $int \rightarrow int$: la fonction s'applique à des nombres entiers et retourne des entiers, puisqu'elle effectue une addition avec son argument. Notons que la signature de la fonction est déduite automatiquement par le système et que le programmeur n'a pas à la spécifier explicitement.

Cet exemple nous amène inévitablement à évoquer la notion de **polymorphisme**. Ethymologiquement, polymorphe signifie plusieurs (*poly*) formes (*morphes*). En informatique, ce terme désigne des objets ou des programmes qui peuvent servir sans modification dans des contextes très différents. Par exemple, une fonction de tri d'objets sera monomorphe si elle ne s'applique qu'à un seul type d'objets (par exemple les entiers) et polymomorphe si elle s'applique à tous les types d'objets. Dans ce cas là, le même programme de tri peut s'appliquer à des nombres entiers ou réels, à des chaînes de caractères, etc.Pour exprimer le polymorphisme de type, il existe une notation spéciale nommée **variable de type** qui correspond à une quantification universelle[5] sur un type. Cette notation se distingue des types ordinaires en les faisant précéder d'une apostrophe (par exemple $'a$, $'b$, etc.).

Pour illustrer ces notions, considérons la fonction identité qui renvoie comme résultat la valeur de son argument :

```
> let identite x = x
# val identite : 'a -> 'a
```

Notons que la signature de la fonction *identité* indique que le domaine du résultat est une variable de type ($'a$) de même nature que celui de l'argument. La fonction *identité* possède donc un degré de liberté dans le sens où le type

[4] On note ici "$> \cdots$" les phrases entrées par l'utilisateur et "$\# \cdots$" les réponses du compilateur.

[5] Le polymorphisme de Caml fonctionne en "tout ou rien" c'est-à-dire qu'il n'existe pas de moyen de définir une fonction qui s'applique à un ensemble limité de types (par exemple *int* et *string* mais pas *bool*).

d'entrée peut être n'importe quel type. Ceci revient à dire du point de vue mathématique que *"quel que soit le type $'a$, la fonction est de type $'a \rightarrow' a$"* (notion de **schéma de type**). Le schéma de type correspond à l'ensemble de tous les types obtenus en remplaçant $'a$ par un type quelconque.

3.2.2.2.2 Les types Caml

Les types utilisables en Caml appartiennent à l'une des trois catégories suivantes : types de bases (*int*, *float*, *string*, etc.), types construits ($int \rightarrow int$, $int\ list$, $int * int$, etc.) et variables de type ($'a$). Les premiers et derniers ont déjà été évoqués, aussi nous ne nous intéresserons ici qu'aux types construits.

Etant donné deux types t_1 et t_2, le constructeur $\rightarrow$ crée le type $t_1 \rightarrow t_2$ qui est le type des fonctions de domaine t_1 et de codomaine t_2. Ce constructeur est un opérateur binaire (prenant deux arguments) et infixe (placé entre les deux arguments).

En revanche, le constructeur de types *list* est unaire et postfixe (placé après l'argument) puisque, à partir d'un unique type t_1, il construit le type $t_1\ list$. Tous les constructeurs de type unaire sont postfixés en Caml.

Etant donné deux types t_1 et t_2, $t_1 * t_2$ représente le type des paires d'un élément de t_1 et d'un élément de t_2. Les paires sont largement utilisées en tant qu'argument ou résultats des fonctions. Par exemple, la fonction suivante calcule simultanément le reste et le quotient d'une division entière :

```
> let quotient_reste (x,y) = ((x/y) , (x mod y))
# val quotient_reste : (int * int) -> (int * int)
```

Les notations pour les paires peuvent se généraliser aux triplets, quadruplets, etc.

3.2.2.2.3 Notation curryfiée

Soit la fonction suivante *addition* effectuant l'addition de deux entiers :

```
> let addition (x,y) = x + y
# val addition : (int * int) -> int
```

Cette fonction possède en réalité un seul argument constitué par la paire d'entiers et non deux arguments distincts. Elle est de ce fait différente de la fonction *add* suivante :

```
> let add x y = x + y
# val add : int -> int -> int
```

Du point de vue pratique, la différence est minime. En effet, les expressions "`addition(1,2)`" et "`add 1 2`" ont toutes deux pour résultat "`# 3 : int`". Du point de vue technique, une fonction qui reçoit ses arguments un par un (cas de la fonction *add*) est dite **curryfiée**. La différence majeure entre *add* et *addition* tient dans la manière de les appliquer. En effet, l'opérateur *addition* doit obligatoirement recevoir les deux arguments de la paire simultanément. Par contre, dans le cas de *add*, le constructeur $\rightarrow$ associe à droite ce qui signifie que le type de *add* peut s'écrire $int \rightarrow (int \rightarrow int)$. Cette écriture explicitement parenthésée indique qu'il est possible d'appliquer un seul argument à *add* (notion d'**application partielle**). Etant donné un entier, *add* retourne une autre fonction dont le type est $int \rightarrow int$. Ce mécanisme d'application partielle permet de redéfinir la fonction *successeur* comme étant :

```
> let successeur = add 1
# val successeur : int  -> int
```

3.2.2.3 Fonctions d'ordre supérieur

Les **fonctions d'ordre supérieur** (*fos*) sont des fonctions dont les arguments ou les résultats sont eux-mêmes des fonctions. Une *fos* est encore appelée une **fonctionnelle**. En règle générale, ces fonctions sont polymorphes. Cette notion est un trait caractéristique des LFs. Elle résulte du mécanisme généralisé d'abstraction des données qui fait des fonctions des données à part entière.

Afin d'illustrer le concept de *fos*, nous allons définir la fonction *compose* qui effectue la relation mathématique de composition $f \circ g$. En Caml, cette fonction s'exprime naturellement et simplement de la manière suivante :

Le type de la fonction *compose* reflète fidèlement les restrictions que l'on doit imposer aux deux fonctions pour pouvoir les composer. Il faut que l'ensemble de de départ de f soit identique à celui d'arrivée de g (variable

```
> let compose f g x = f (g x)
# val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

de type $'a$). De plus, par définition $f \circ g$ a pour domaine celui de g (variable de type $'c$) et pour codomaine celui de f (variable de type $'b$).

Le contrôleur de types retrouve tout seul ces contraintes tout en les exprimant de manière minimale sous la forme de variables de type. A chaque composition de fonctions, il les vérifie et remplace les variables de type par les types des deux fonctions à composer. A titre d'illustration, considérons l'exemple suivant qui calcule le successeur de la longueur d'une chaîne de caractères[6].

```
> let longueur_plus_un = compose successeur string_length
# val longueur_plus_un : string -> int
```

3.2.2.4 Filtrage

La définition d'une fonction en LF se fait souvent au moyen d'un mécanisme sophistiqué de **filtrage** c'est-à-dire d'un ensemble d'équations décrivant le comportement de la fonction selon les configurations des paramètres d'entrée.

A titre d'exemple, considérons la fonctionnelle *map* qui applique une même fonction f à tous les éléments d'une liste de données l. Par exemple *map successeur* $[1; 2; 3]$ retourne la liste $[2; 3; 4]$.

L'expression *map* $f[x_1, x_2, \ldots, x_n]$ retourne donc $[fx_1, fx_2, \ldots, fx_n]$ ce qui se traduit en Caml par :

```
> let rec map f l =
         match l with
              []   -> []
            | x::r -> f x :: map f r
# map : ('a -> 'b) -> 'a list -> 'b list
```

Cette définition nécessite pour le lecteur peu averti un certain nombre d'explications. La fonction *map* utilise un appel explicite au filtrage,

[6]La longueur d'une chaîne de caractères est effectuée par la fonction pré-définie *string_length*.

mécanisme introduit par le mot-clé *match*. On lit ce filtrage de la manière suivante : si l est la liste vide ([]), alors on renvoie comme résultat une liste vide ([]); dans le cas contraire, on extrait le premier élément de la liste (x) pour lui appliquer la fonction f et on applique la fonctionnelle aux éléments restants de la liste (r).

Hors du contexte du mécanisme de filtrage présenté ici, il faut noter que la définition de la fonction *map* est en fait récursive d'où l'utilisation du mot réservé *let rec*.

3.3 Squelettes et LFs

3.3.1 Adéquation LFs-Squelettes

Le chapitre 2 a montré l'intérêt de l'utilisation des squelettes pour le développement d'applications parallèles. Notre expérience dans le domaine du TI a conduit à l'extraction d'un ensemble restreint de quatre squelettes. Le problème se pose désormais d'exprimer et d'exploiter ces constructeurs génériques. Plus pratiquement, la spécification d'applications "squelettiques" passe inévitablement par le recours à un formalisme d'expression des squelettes.

Dans notre approche, toutes les fonctions de calculs développées par le programmeur sont purement séquentielles. La seule façon d'exprimer une quelconque forme de parallélisme est l'utilisation des squelettes. Par définition, ceux-ci sont des constructeurs paramétrés par des données et des fonctions spécifiques à une application particulière. De fait, ils peuvent être naturellement exprimés sous la forme de *fos* polymorphes au sein de n'importe quel LF[7].

Par exemple, une définition possible du squelette SCM décrit au paragraphe 2.4.2 peut s'écrire sous la forme suivante :

Cette définition utilise la fonctionnelle *map* (définie au paragraphe 3.2.2.4) ce qui permet d'exprimer le squelette de manière purement applicative, c'est-à-dire sans aucune référence à un quelconque modèle d'exécution.

Remarquons également la présence de variables de types dans la signature du squelette SCM. Cette signature sera utilisée par le synthétiseur de types pour vérifier la cohérence des types des fonctions passées en argument du squelette SCM. Celui-ci possède donc trois arguments :

[7]Cet argument n'est pas nouveau dans le sens où, mis à part le projet P^{3}L, l'ensemble des travaux effectués sur les squelettes de parallélisation est intimement lié à la programmation fonctionnelle.

```
> let scm split compute merge x = merge ( map compute (split x))
# val scm : ('a -> 'b list)  (* Fonction de partition  *)
         -> ('b -> 'c)       (* Fonction de traitement *)
         -> ('c list -> 'd)  (* Fonction de fusion     *)
         -> 'a               (* Donnee                 *)
         -> 'd               (* Resultat               *)
```

① Une fonction *split* de type $'a \rightarrow' b\ list$

② Une fonction *compute* de type $'b \rightarrow' c$

③ Une fonction *merge* de type $'c\ list \rightarrow' d$

et retourne une fonction dont le type est $'a \rightarrow' d$.

Du point de vue de l'utilisateur, les programmes décrivent uniquement les dépendances de données entre les différentes étapes de l'application, chaque étape correspondant soit au calcul d'une fonction séquentielle, soit à l'instanciation d'un squelette comme illustré sur l'exemple suivant :

```
(* Un exemple simple d'utilisation du squelette SCM *)
(* dans le cadre d'un calcul de gradient par le masque de Sobel *)
(* sur quatre partitions en bandes de l'image *)
let im1 = read_img 256 256 in
let im2 = scm (rowblock 4) sobel (blockrow 4) im1 in
display_img im2
```

Les fonctions *read_img*, *rowblock*, *sobel*, *blockrow* et *display_img* représentent les fonctions séquentielles de calcul. Elles sont soit fournies par l'environnement, soit développées par le programmeur.

La succession des définitions imposées par les déclarations *let* $\cdots$ *in* définit l'enchaînement séquentiel des étapes et de fait, tout le parallélisme potentiel de l'application est restreint à celui du squelette SCM utilisé dans cet exemple. Cette approche — dans laquelle le programmeur est responsable de la description explicite des sites de parallélisme — est un trait caractéristique de beaucoup de méthodologies basées sur les squelettes.

Il faut souligner que si un formalisme fonctionnel est utilisé pour la définition des squelettes ainsi que pour la description des applications, rien n'interdit que le développement des fonctions passées en *argument* des squelettes soit réalisé en langage impératif (en l'occurence le langage C dans

notre approche). Cet aspect est crucial puisqu'il autorise la réutilisation d'opérateurs déjà développés dans un contexte impératif, séquentiel et peut donc faciliter la mise en œuvre d'applications parallèles par un public inexpérimenté.

Notre approche fait donc appel à un sous ensemble du langage Caml pour décrire les applications parallèles de TI basées sur les squelettes.

Cette démarche nous permet en fait d'associer à chaque squelette deux définitions distinctes :

① une définition fonctionnelle écrite en Caml séquentiel à l'aide des *fos* séquentielles du langage. C'est cette définition indépendante de toute caractéristique architecturale que nous allons détailler dans la suite de ce chapitre.

② une définition opérationnelle sous la forme d'un graphe de processus paramétrable et implantable sur architecture parallèle dédiée. Cet aspect, intimement lié à l'architecture cible, sera décrit au chapitre 4.

3.3.2 Définition fonctionnelle des squelettes

3.3.2.1 Le squelette SCM

Le squelette SCM est utilisé pour exprimer le parallélisme géométrique. Il associe trois opérateurs (*split*, *compute* et *merge*) effectuant respectivement la partition des données d'entrée en sous domaines, le calcul sur chacun des sous-domaines et la fusion des résultats intermédiaires (cf. paragraphe 2.4.2).

Une définition fonctionnelle possible de ce squelette est donnée ci-dessous :

```
> let scm n split compute merge x = merge n (pmap compute (split n x))

# val scm :
           int                       (* Nombre de partitions   *)
        -> (int -> 'a -> 'b tuple) (* Fonction de partition  *)
        -> ('b -> 'c)              (* Fonction de traitement *)
        -> (int -> 'c tuple -> 'd) (* Fonction de fusion     *)
        -> 'a                      (* Donnee                 *)
        -> 'd                      (* Resultat               *)
```

Cette définition utilise une variante de la fonctionnelle *map* opérant sur des données de type *'a tuple*. Fonctionnellement parlant, les *tuples* sont

équivalents aux listes. La différence majeure est au niveau de l'interprétation opérationnelle des squelettes qui sera décrite au paragraphe 4.3.2. Par rapport à la définition donnée à la page 83, on peut aussi noter que le squelette SCM prend un argument supplémentaire déterminant le nombre de partitions (n). Cette valeur transmise aux fonctions *split* et *merge* ne doit en aucun cas excéder le nombre de processeurs disponibles mais peut éventuellement être inférieure.

3.3.2.2 Le squelette DF

Le rôle du squelette DF est d'appliquer une même fonction de traitement (*compute* sur tous les éléments d'une liste de données. Ces résultats sont accumulés itérativement par la fonction *acc* dès qu'ils sont produits (cf. paragraphe 2.4.3).

La définition Caml correspondante est alors :

```
> let df n compute acc z xs = foldl acc z (map compute xs)

# val df :
          int              (* Nombre de processeurs             *)
       -> ('a -> 'b )      (* Fonction de traitement             *)
       -> ('c -> 'b -> 'c) (* Fonction d'accumulation            *)
       -> 'c               (* Valeur initiale de l'accumulateur *)
       -> 'a list          (* Liste de donnees                   *)
       -> 'c               (* Resultat                           *)
```

Ici, *compute* est la fonction utilisateur de calcul, *acc* la fonction d'accumulation des résultats partiels (avec z la valeur initiale de l'accumulateur). Notons que dans cette définition fonctionnelle, l'argument n représentant le nombre de processeurs n'est pas utilisé.

L'accumulation des résultats partiels est réalisée à mesure que ceux-ci sont produits d'où l'emploi de la fonctionnelle *foldl* qui permet d'appliquer itérativement la fonction *acc* à la liste des résultats. Du point de vue mathématique, on peut définir *foldl* avec la relation suivante : $foldl\ f\ z\ [x_1, x_2, \ldots, x_n] = (\ \ldots(f\ (f\ z\ x_1\ )\ x_2\ )\ \ldots x_n)$ ce qui donne en Caml :

```
> let rec foldl f z xs =
             match xs with
                 []   -> z
               | x::l -> foldl f (f z x) l
# val foldl : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

3.3.2.3 Le squelette TF

Le squelette TF peut être vu comme une généralisation du squelette DF. Le principe est basé sur une ferme de processeurs mais les processeurs testent chaque paquet de données (en utilisant un prédicat h). En cas de succès, la fonction *solve* est appliquée et le résultat est renvoyé au *maître* et accumulé (opérateur *combine*). Dans le cas contraire, la fonction *divide* est appliquée afin de générer récursivement de nouveaux paquets de données (cf. paragraphe 2.4.4).

Une définition en Caml séquentiel d'un tel squelette est donnée ci-après :

```
> let rec tf n h solve divide combine z xs =
    let f x =
      if (h x) then combine z (solve x)
      else tf n h solve divide combine z (divide x) in
    foldl combine z (map f xs)

# val tf :
          int               (* Nombre de processeurs             *)
       -> ('a -> bool)      (* Fonction de predicat              *)
       -> ('a -> 'c)        (* Fonction de traitement            *)
       -> ('a -> 'a list)   (* Fonction de partition             *)
       -> ('b -> 'c -> 'b)  (* Fonction d'accumulation           *)
       -> 'b                (* Valeur initiale de l'accumulateur *)
       -> 'a                (* Donnees                           *)
       -> 'b                (* Resultat                          *)
```

L'expression fonctionnelle du squelette TF est basée sur la notion de récursivité (définition du type *let rec*) en raison des éventuelles divisions successives de la donnée : si la donnée est triviale, alors le résultat est accumulé, sinon on applique le squelette TF sur la partition générée de la donnée.

3.3.2.4 Le squelette ITERMEM

Le squelette ITERMEM est utilisé dès lors qu'il est nécessaire de faire apparaître l'aspect itératif des traitements et typiquement, dans le cadre d'applications de TI lorsque les calculs sur la trame i dépendent des résultats calculés sur les trames précédentes $i-1$, $i-2$, $\cdots$, $i-k$ (cf. paragraphe 2.4.5).

En Caml, la définition d'un tel squelette est donc nécessairement récursive :

```
> let itermem f1 f2 f3 z x =
    let rec h z =
      let z', y = f2 (z, f1 x) in
      f3 y; h z'
    in h z

# val itermem :
               ('a -> 'b)            (* Fonction d'acquisition     *)
            -> ('c * 'b -> 'c * 'd)  (* Fonction de traitement     *)
            -> ('d -> unit)          (* Fonction de restitution    *)
            -> 'c                    (* Valeur initiale de memoire *)
            -> 'a                    (* Donnee d'entree            *)
            -> unit
```

3.3.3 Emulation séquentielle

Comme précisé au paragraphe 3.3.1, l'approche retenue pour la spécification des squelettes et des applications parallèles de TI nous amène à associer à chaque squelette deux définitions équivalentes : une définition déclarative en Caml séquentiel et une autre opérationnelle implantable sur une architecture cible.

La première est indépendante de toute implantation sur architecture cible. Elle s'exprime à l'aide des *fos* séquentielles pré-définies du langage (*map*, *foldl*, etc.). Etant écrite en Caml, cette première définition peut être donc vue comme une **spécification exécutable** de la deuxième. Ce point a des conséquences pratiques significatives puisqu'il offre l'opportunité au programmeur d'effectuer une validation fonctionnelle de ses applications sur une machine séquentielle traditionnelle avant implantation sur une architecture parallèle dédiée. Cette validation que nous appelons **émulation séquentielle**

peut utiliser tous les mécanismes et outils séquentiels de mise au point permettant de vérifier le comportement correct de l'application. Dès lors que l'implantation parallèle des squelettes utilisés dans l'application est supposée correcte[8], la phase d'émulation séquentielle garantit que l'implantation parallèle d'une application ainsi vérifiée sera correcte. Cette possibilité d'émuler les programmes parallèles sur des machines séquentielles permet donc de séparer les phases de validation fonctionnelle (le programme retourne-t'il le résultat attendu ?) et de validation opérationnelle (les contraintes temporelles sont-elles satisfaites ?). Cet aspect a déjà été mis en avant par Danelutto *et al.* dans [DCLP98] sous le nom de *débogage logique*.

La coexistence de deux définitions pour chaque squelettes confère par ailleurs une plus grande portabilité aux applications puisque la première, indépendante de toute architecture cible, demeure inchangée face aux évolutions architecturales. Seule la seconde devra être adaptée par le programmeur système pour une nouvelle architecture. L'effort requis pour mettre à jour l'implantation d'un petit nombre de squelettes est généralement faible par rapport à celui consistant à effectuer manuellement le portage intégral d'une application. Notons toutefois que cette phase peut compromettre l'application à grande échelle des méthodes fondées sur les squelettes et expliquer pourquoi, jusqu'à présent, les outils basés sur les squelettes sont restés au stade expérimental et toujours dédiés à des architectures particulières[Kes95].

Cette faiblesse sous estimée des méthodologies fondées sur les squelettes peut être partiellement résolue en introduisant un niveau intermédiaire d'abstraction effectuant un lien entre la définition fonctionnelle des squelettes et leur implantation sur architectures dédiées. Ce niveau intermédiaire représente en définitive une spécification différente des squelettes faisant appel non plus aux seules *fos* séquentielles du langage mais à une bibliothèque portable de communication. Par exemple, dans [Ser97], une telle approche utilisant la bibliothèque de communication MPI est présentée. Cet aspect ne faisant pas partie intégrante des travaux décrits dans ce mémoire, nous allons seulement en présenter succinctement les deux propriétés principales.

Le choix d'une bibliothèque de communication en lieu et place de mécanismes bas niveau spécifiques aux architectures offre l'opportunité d'accroître la portabilité des implantations des programmes sur différents types d'architectures. En effet, quelle que soit l'architecture, l'implantation du squelette est identique et seules les primitives de communications de la bibliothèque doivent être développées pour répondre aux caractéristiques de

[8]Il revient bien entendu au programmeur système *i.e.* celui qui développe les squelettes d'assurer l'équivalence de la définition séquentielle des squelettes et celle opérationnelle.

l'architecture.

L'utilisation d'une bibliothèque portable de communication telle que MPI permet d'étendre le champ des architectures — en dehors de la plate-forme cible — sur lesquelles l'application peut être implantée. Par exemple, il devient possible d'émuler les applications sur des réseaux de stations de travail permettant ainsi de vérifier la distribution des calculs sur les processeurs.

En résumé, la phase d'émulation séquentielle apparaît comme vitale dans une optique de prototypage rapide d'applications. L'exécution séquentielle des programmes en dehors de tout contexte architectural dédié et l'exécution parallèle sur réseau de stations de travail offrent l'opportunité de valider les algorithmes du point de vue comportemental[9]. Ces deux aspects sont illustrés sur la figure 3.1 et constituent donc une première étape importante dans notre méthodologie de développement parallèle d'applications de TI.

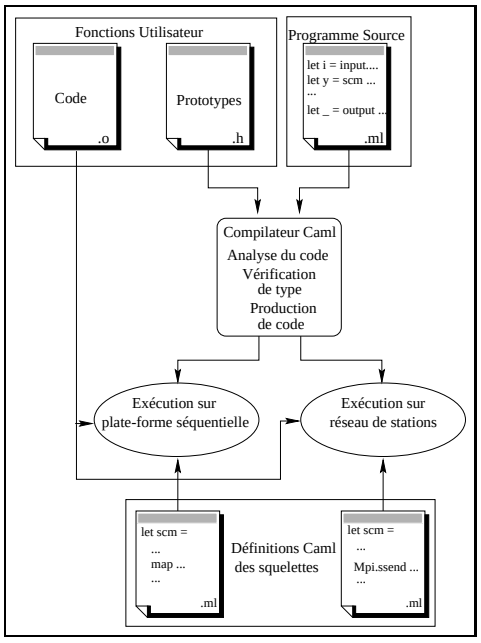


Figure 3.1: La phase d'émulation séquentielle

3.4 Conclusion

Ce chapitre, consacré à la programmation fonctionnelle, a mis en avant le caractère naturel des LFs pour exprimer simplement la notion de squelettes

[9]L'émulation séquentielle répond ainsi aux problèmes de validation des programmes parallèles évoqués au paragraphe 1.4.2.3.1.

de parallélisation. Le concept de fonctions d'ordre supérieur "colle" parfaitement avec celui des squelettes génériques (paramétrés par des fonctions spécifiques à une application donnée) présentés et développés au chapitre 2. L'exploitation de cette propriété nous a donc conduits à exprimer chaque squelette dédié sous la forme de fonctionnelle pure décrite en langage Caml.

La mise en œuvre de ce formalisme fonctionnel pour la phase de spécification des squelettes permet de répondre à certains des critères requis pour les modèles efficaces de programmation parallèle discutés au chapitre 1 :

- **Facilité de programmation** : Les squelettes fonctionnels sont vus par le programmeur d'applications comme des constructeurs de haut niveau décrivant explicitement certaines formes usuelles de parallélisme mais dont les détails de mise en œuvre sont totalement cachés.
- **Facilité de compréhension** : Le nombre réduit de squelettes confère au modèle de programmation parallèle que nous avons développé des facilités de compréhension même pour des programmeurs inexpérimentés.
- **Indépendance vis à vis de l'architecture** : La spécification des squelettes étant décrite en langage Caml est décorrellée de toute architecture cible sur laquelle elle pourrait être implantée. Cet aspect crucial accroît la portabilité des programmes.

Enfin, le prototypage rapide d'applications, objectif principal de ces travaux semble plus facile à atteindre de fait de la séparation distincte entre les deux définitions équivalentes des squelettes. En effet, la définition fonctionnelle, constituant une spécification exécutable des squelettes, offre des facilités de développement rapide. La validation comportementale (en dehors de toute considération temporelle) est plus aisée grâce aux possibilités offertes par les outils de mise au point traditionnels.

La présentation faite ici laisse évidemment entière la question de la *définition opérationnelle* (parallèle) des squelettes, *i.e.* leur implantation sur l'architecture cible. Celle-ci fait l'objet du chapitre suivant.

Chapter 4

L'outil d'aide à la parallélisation

4.1 Introduction

Les chapitres précédents ont posé les fondations d'un formalisme dédié au développement d'applications parallèles de TI. La méthodologie présentée repose sur l'utilisation de *squelettes de parallélisation* admettant à la fois une spécification fonctionnelle et plusieurs implantations parallèles plus ou moins efficaces.

Si les aspects théoriques relatifs à ce formalisme ont été largement discutés, seule la phase de spécification fonctionnelle des squelettes a été jusqu'à présent abordée.

Or, la validation et la crédibilité de notre approche passent inévitablement par ce que l'on peut nommer la "vérité de terrain" à savoir le développement d'applications conséquentes de TI exploitant efficacement les concepts précédemment cités. Satisfaire cet objectif conduit à mettre en œuvre un ensemble d'outils dédiés effectuant de manière la plus automatique possible la transformation de la spécification fonctionnelle en une implantation efficace s'exécutant sur une machine dédiée.

De fait, ce chapitre sera entièrement consacré à la présentation de l'outil de développement **SKiPPER** (SKeletal Parallel Programming EnviRonment), environnement de programmation dédié au prototypage rapide des applications de vision artificielle par squelettes fonctionnels.

Cependant, préliminairement au travail de présentation de cet outil informatique, il nous apparaît indispensable de consacrer quelques instants à l'architecture hétérogène Transvision sur laquelle les applications de TI vont

s'exécuter. Par la suite, nous nous attacherons à décrire les différents modules composant l'outil SKiPPER. Trois aspects seront plus particulièrement mis en avant à savoir l'extraction du parallélisme sous-jacent des squelettes, la mise en correspondance de ce parallélisme avec l'architecture cible et la génération de code cible dédié à la machine Transvision.

Une dernière partie sera consacrée à la phase d'implantation des squelettes de TI. Celle-ci sera détaillée minutieusement démontrant ainsi qu'elle tire pleinement parti des spécificités architecturales. Une telle étude nous permettra également de développer la notion de *prédictabilité des performances* des applications sous la forme de modèles analytiques de performance relatifs à chaque squelette.

4.2 La machine parallèle Transvision

4.2.1 Architecture générale

La machine Transvision, développée au LASMEA, est un calculateur parallèle dédié à la vision artificielle. Cette machine présente une architecture hétérogène construite autour d'un monde *pipe-line* évoluant à la cadence vidéo et d'un monde MIMD à mémoire distribuée. Elle a été conçue pour l'évaluation de schémas de parallélisation et ce avant que cette notion de schémas de parallélisation ne soit formalisée dans le concept de squelettes. Sa structure matérielle permet d'étudier différentes configurations de machines cibles, c'est-à-dire de machines de vision dédiées à une application précisée par un cahier des charges spécifique.

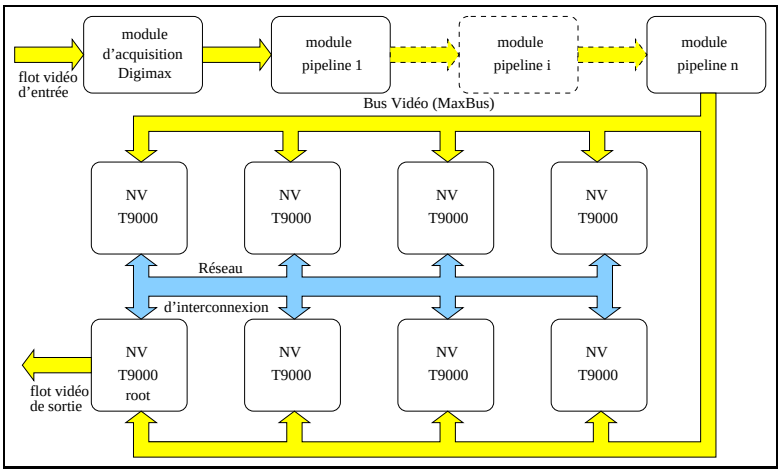


Figure 4.1: Synoptique de la machine Transvision T9000

Cette machine hétérogène s'articule autour de deux structures principales (cf. figure 4.1):

- la première structure repose sur un modèle d'exécution dit à "recirculation d'images" issu du modèle d'exécution pipeline. Il est composé des modules de traitements temps réel vidéo de la firme DATA-CUBE. Ces modules évoluent au rythme du flot de données numériques délivré par un module d'acquisition[1] selon le format MAXBUS[2]. Ce même module permet de restituer[3] des images dans le même format.

- la seconde structure fait appel à un modèle d'exécution de type MIMD à mémoire distribuée. Chaque module appelé *Noeud Vidéo* est constitué autour d'un processeur de type Transputer T9000, d'une mémoire locale (8 Mo), d'une mémoire vidéo d'entrée-sortie double accès et d'un contrôleur vidéo chargé des transferts entre le bus vidéo MAXBUS et les différents bancs de mémoire vidéo.

4.2.2 Principe du Noeud Vidéo

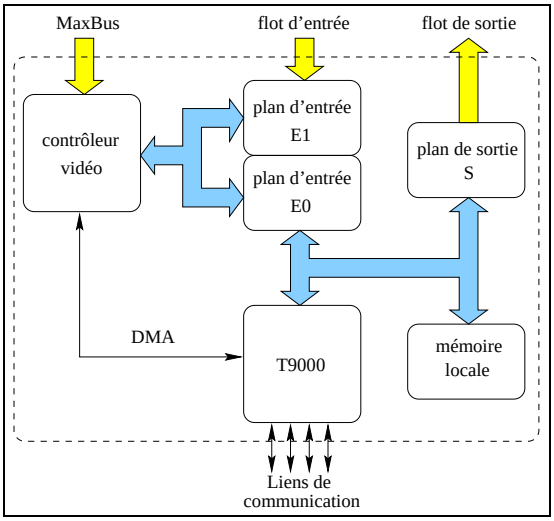


Figure 4.2: Principe du noeud Vidéo

Le principe du Noeud Vidéo, représenté sur la figure 4.2, repose sur la mise en œuvre de deux plans mémoire vidéo d'entrée ($E0$ et $E1$) et d'un

[1] Convertisseur A/N DIGIMAX de la firme DATA-CUBE.
[2] Bus vidéo spécifique $512 * 512$ codé sur 8 bits.
[3] Convertisseur N/A.

plan mémoire vidéo de sortie (S). Chaque plan d'entrée est utilisé en permutation (*swapping*) avec accès protégé entre le contrôleur vidéo et le processeur. Pendant que le processeur travaille sur un des deux plans d'entrée, le second est alimenté par le bus vidéo en provenance du monde pipeline. Durant le traitement d'une image, le processeur dispose donc du même plan d'entrée. Pendant ce temps, le contrôleur vidéo stocke les données dans le second plan. A la fin de l'image et en l'absence de requête du processeur, le processus de stockage est réitéré sur le même plan image, écrasant ainsi la précédente image. Cet accès protégé permet au processeur de conserver l'image durant tout le traitement.

A la fin du traitement, l'ensemble des processeurs libère leur plan d'entrée vidéo et attendent la fin de l'acquisition courante. A cet instant, il y a permutation des plans d'entrée. Ce mécanisme permet au processeur d'avoir toujours accès à la dernière image acquise.

4.2.3 Un exemple de machine cible

Une machine cible est définie par le nombre et les fonctions des modules pipeline, le nombre de Noeuds Vidéo ainsi que la topologie de connexion du réseau de processeurs. L'exemple représenté sur la figure 4.3 est composée d'un module d'acquisition/restitution vidéo et d'un anneau de huit Noeuds Vidéo.

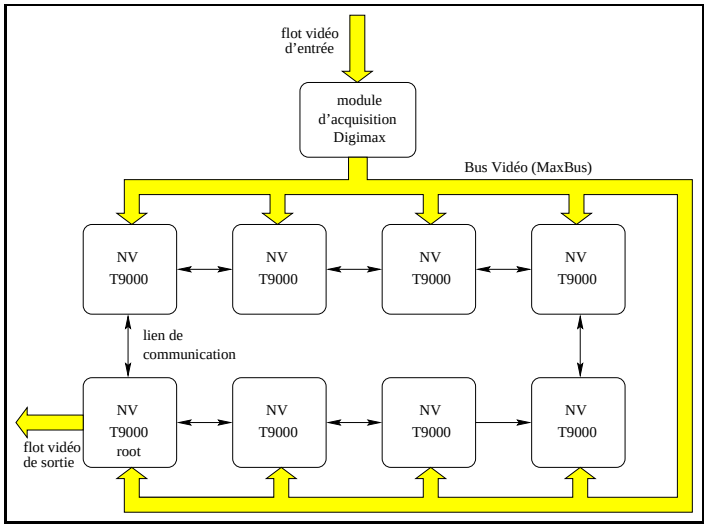


Figure 4.3: Un exemple d'architecture cible

Cette configuration d'architecture, très générale, a été utilisée pour implanter l'ensemble des applications de TI bas et moyen et niveau qui sont

décrites dans le chapitre 5.

4.3 Présentation de SKiPPER

4.3.1 Généralités

SKiPPER est un environnement logiciel de développement d'applications parallèles de TI bas et moyen niveau, basé sur le concept des squelettes fonctionnels de parallélisation. Cet outil permet d'obtenir, à partir d'une spécification fonctionnelle d'une application faisant appel à un certain nombre de squelette pré-définis, une implantation parallèle sur la machine cible. SKiPPER a été développé initialement pour la machine Transvision décrite au paragraphe précédent mais nous verrons que les concepts et outils sur lesquels SKiPPER se fonde sont suffisamment généraux pour assurer une portabilité aisée vers toute machine de type MIMD à mémoire distribuée.

Pour réaliser la transformation de la spécification fonctionnelle en une représentation implantable des programmes, SKiPPER s'articule autour de trois modules indépendants réalisant successivement une phase d'expansion des squelettes, une phase de placement-ordonnancement et une phase de génération de code cible.

① à partir de la description de la spécification selon le principe illustré au paragraphe 3.3.1, un compilateur spécifique effectue l'**expansion des squelettes**, c'est-à-dire le passage d'une représentation purement fonctionnelle à une représentation sous la forme d'un graphe de processus exhibant le parallélisme potentiel des programmes,

② une phase de **placement-ordonnancement** de ce graphe de processus sur l'architecture cible est effectuée. Cette tâche complexe est confiée à l'outil SynDEx dont les principales caractéristiques ont été exposées au paragraphe 1.5.3,

③ après exécution de l'heuristique de placement-ordonnancement, SynDEx génère un exécutif distribué sous la forme d'un macro-code générique. Celui-ci est finalement transformé en **code optimisé** implantable sur la machine Transvision. Ce code final est dédié à l'architecture et tire pleinement parti des caractéristiques du processeur T9000 et du réseau de communication associé.

La figure 4.4 présente une vue générale de l'environnement de développement SKiPPER et décrit l'enchaînement des trois points présentés

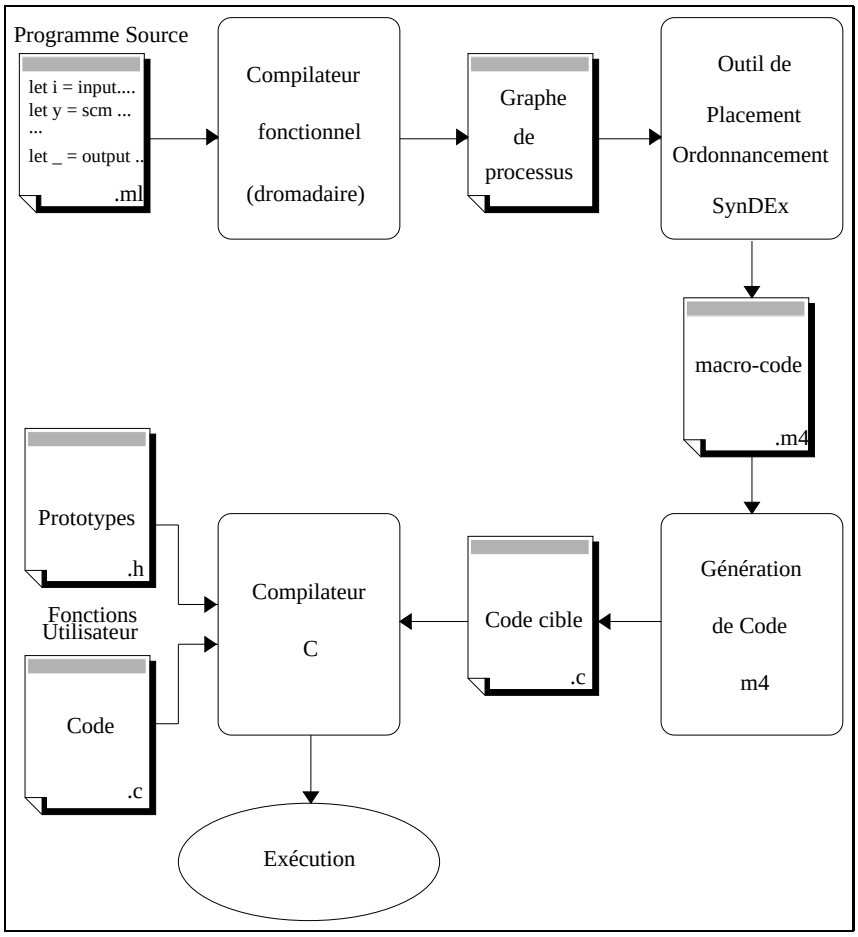


Figure 4.4: L'environnement de développement SKiPPER

précédemment de manière très succincte. Ceux-ci vont être largement détaillés dans les paragraphes suivants (paragraphes 4.3.2 à 4.3.4).

Il est à noter que ces trois phases sont effectuées de manière **automatique**. Il est donc extrêmement facile, après visualisation des résultats d'exécution, de modifier la spécification initiale afin de corriger un problème ou d'évaluer une solution alternative. Ce point justifie pleinement l'appellation d'outil de PROTOTYPAGE RAPIDE donné à SKiPPER.

4.3.2 Expansion des squelettes

Par **expansion des squelettes**, nous sous-entendons les techniques mises en œuvre permettant d'extraire la sémantique opérationnelle parallèle des squelettes encapsulée dans la spécification fonctionnelle des applications.

Concrètement, cela consiste à rendre explicite le comportement parallèle des squelettes utilisés dans l'application. Cette phase nécessite donc la mise en place d'une **représentation intermédiaire** des programmes qui rend explicite les schémas de calcul-communication caractérisant chaque squelette. Cette représentation intermédiaire doit être relativement proche des caractéristiques architecturales pour pouvoir les exploiter le plus efficacement possible lors de la phase finale de génération de code cible. Dans le même temps, elle ne doit pas pour autant être complètement dédiée à une machine cible particulière pour ne pas restreindre fortement la portabilité de l'approche. Une fois de plus, il est nécessaire de trouver un compromis entre généricité et efficacité.

La classe d'architectures que nous visons est celle des machines MIMD à mémoire distribuée. Pour celles-ci, les applications parallèles sont classiquement formalisées sous la forme de **graphes de processus** dialoguant par passage de messages. Les sommets composant le graphe sont alors des processus séquentiels de calcul et les arcs traduisent les échanges de données entre les processus. Un processus est vu comme une entité informatique indépendante opérant sur des données propres (auquel lui seul a accès) stockées dans sa mémoire locale. Cette notion de graphe de processus a été formalisée par Hoare qui a introduit le concept de *processus séquentiels communicants* (CSP)[Hoa85].

Une telle représentation semble bien adaptée pour répondre à nos exigences de généricité. En effet, les graphes de processus prennent en compte certains traits caractéristiques des architectures MIMD à mémoire distribuée — mémoire locale, réseau de communications, etc.— tout en restant éloignés des aspects bas niveau — type des processeurs, nature du réseau d'interconnexion, etc.— relatifs à une architecture donnée.

La transformation de la spécification fonctionnelle des applications en un graphe de processus est réalisée par un compilateur spécifique[4] nommé *Dromadaire*^{TM}. Le développement du compilateur *Dromadaire* n'ayant pas fait l'objet du travail décrit dans ce mémoire, nous nous contenterons de décrire ici ses fonctionnalités externes, sans nous attacher à ses fondements théoriques ou son mécanisme de fonctionnement.

Basiquement, *Dromadaire* ne fait que traduire les dépendances des fonctionnelles exprimées dans un programme Caml sous la forme d'un graphe de processus dans lequel :

- les noeuds représentent les fonctions séquentielles de calcul de l'application,

[4]Développé en Caml.

➟ les arcs décrivent les types et les volumes de données échangés par les noeuds de calculs.

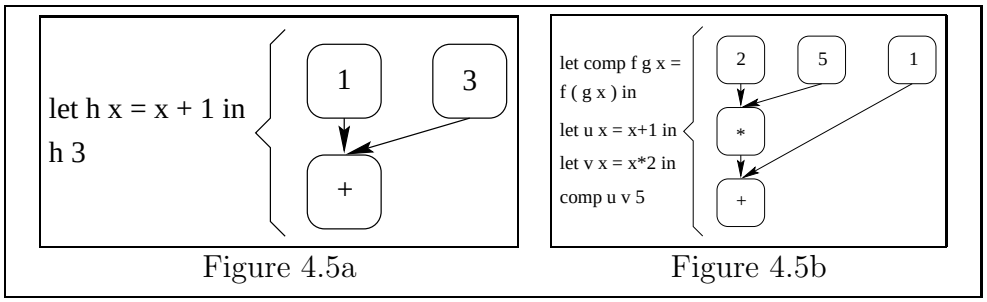


Figure 4.5a Figure 4.5b

Figure 4.5: Deux exemples simples de graphes de processus générés par *Dromadaire*

Les figures 4.5a et 4.5b illustrent ces aspects avec des exemples simples de programmes Caml.

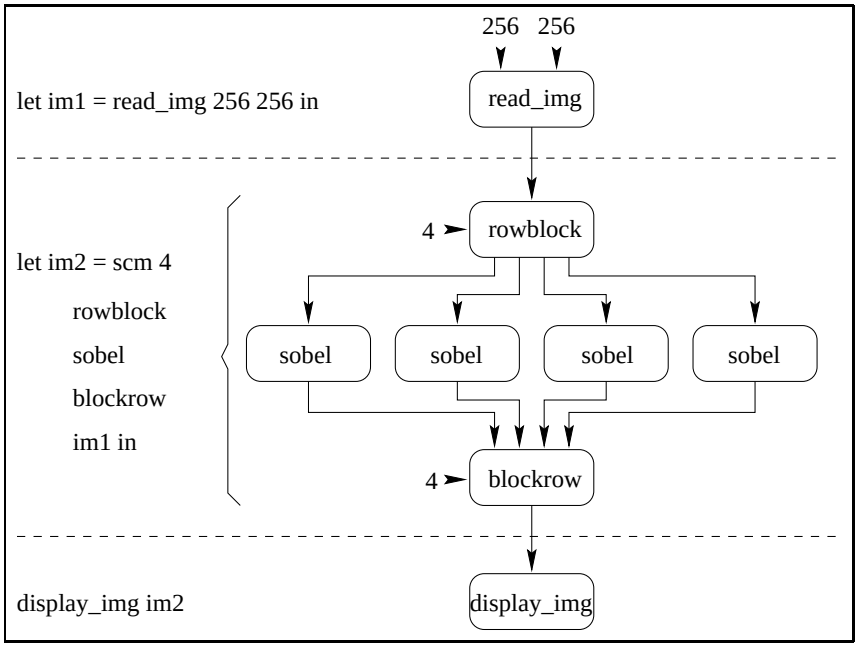


Figure 4.6: Un exemple d'expansion de squelettes

Un autre trait particulier de *dromadaire* est son interprétration des données de type *tuple*. Un *tuple* correspond en effet à un ensemble d'arcs du graphe associé à l'application parallèle d'une même fonction. La figure 4.6 par exemple décrit l'expansion des squelettes pour l'application de calcul de

gradient par le masque de Sobel présenté au paragraphe 3.3.1, application faisant appel aux fonctions *rowblock* et *blockrow* manipulant des données de type *tuple* et dont les signatures sont :

```
val rowblock : int -> image -> image tuple
val blockrow : int -> image -> image tuple
```

4.3.3 Placement-ordonnancement du graphe de processus

4.3.3.1 Choix de l'outil SynDEx

Cette phase réalise une allocation spatiale et temporelle du graphe de processus sur l'architecture cible. Cela revient à allouer d'une part les processus de calcul aux processeurs et d'autre part les arcs du graphe aux liens de communication.

En règle générale, l'architecture cible peut être vue comme un graphe comprenant un ensemble de processeurs éventuellement de type différent reliés entre eux par un réseau d'interconnexion composé de différents médias de communications (liens mono ou bi-directionnels, bus, etc.).

Dès lors, la phase de placement-ordonnancement de l'application sur l'architecture peut se formaliser en terme de transformation de graphes. Ces transformations ont pour objectif de faire coïncider le graphe de l'application (décrivant le parallélisme potentiel) et le graphe de l'architecture (décrivant le parallélisme disponible) tout en respectant certaines contraintes (latence minimum dans notre cas). La résolution d'un tel problème — connu comme étant NP-complet — a fait l'objet de nombreuses études. De fait, plutôt que de développer un nouvel algorithme de placement-ordonnancement, notre approche fut plus pragmatique et nous avons opté pour l'utilisation d'un outil déjà existant, à savoir SynDEx.

Le choix de SynDEx fut motivé premièrement par le fait que nous possédions une expérience non négligeable sur les possibilités d'un tel outil dans le domaine du TI temps réel[Gin95].

Deuxièmement, SynDEx offre des caractéristiques intéressantes dans une optique de prototypage rapide d'applications à fortes contraintes temporelles (cf. paragraphe 1.5.3). La gestion automatique du placement-ordonnancement de l'algorithme sur l'architecture suivie d'une génération d'un exécutif distribué sur réduisent fortement le temps de développement

des applications libérant ainsi l'utilisateur de tâches lourdes de programmation bas niveau.

De plus, l'exécutif distribué traduit le placement-ordonnancement produit par l'heuristique sous la forme d'un macro-code générique indépendant de tout type de processeur. La transformation de ce code intermédiaire en un code implantable sur une architecture particulière ne nécessite que le développement d'un jeu de macro-définitions spécifiques. Cet aspect a constitué un facteur important dans le choix de SynDEx puisqu'il offre des opportunités de portabilité des applications. L'effort de portage est limité au re-développement des macro-définitions constituant l'exécutif générique. Ainsi, de la phase de spécification fonctionnelle des applications jusqu'à la génération de l'exécutif distribué par SynDEx, l'approche que nous avons retenue est décorrelée des spécificités de l'architecture cible. Seule la phase finale de génération de code cible est dépendante des caractéristiques architecturales.

4.3.3.2 Contraintes liées à SynDEx

Comme expliqué au paragraphe 1.5.3, l'utilisation de SynDEx en tant qu'outil de placement-ordonnancement et de génération d'exécutif n'est possible que si les graphes de processus issus de la phase d'expansion des squelettes peuvent être vus comme des graphes purement flot de données.

En opérant sur de tels graphes, SynDEx a une vision purement statique du parallélisme. Par "vision statique", nous entendons que le placement et l'ordonnancement respectivement des calculs sur les processeurs et des communications sur les liens sont prédéterminés lors de la phase de génération d'exécutif.

De fait, pour que les graphes de processus associés aux squelettes soient utilisables sous SynDEx, il est indispensable de les voir comme des graphes reposant sur un *schéma de calcul-communication entièrement statique* dans lequel tous les aspects relatifs au placement-ordonnancement des opérations de calcul et de communications peuvent être fixés à la compilation.

4.3.3.3 Cas des squelettes "statiques"

Parmi les quatre squelettes que nous avons développés, les squelettes SCM et ITERMEM peuvent facilement s'exprimer sous la forme de graphes flots de données utilisables sous SynDEx.

Premièrement, le squelette SCM est composé de $n+2$ processus de calculs (un processus *split*, n processus *compute* et un processus *merge*), n étant fixé

statiquement dans la spécification de l'application (cf. paragraphe 2.4.2). Chacun des processus le composant peut être vu comme un opérateur flot de données qui exécute successivement les trois opérations suivantes à chaque itération :

① réception des données sur chacun des arcs de communications d'entrée,

② combinaison de ces entrées et production de résultats par une fonction de calcul séquentielle,

③ émission des résultats sur chacun des arcs de communications de sortie.

Le graphe flot de données associé au squelette SCM est en donc en tout point similaire au graphe de processus décrit par exemple sur la figure 4.6.

Deuxièmement, considérons le cas du squelette ITERMEM. Celui-ci possède la particularité de stocker les résultats de l'itération i pour pouvoir les utiliser à l'itération $i+1$ (cf. paragraphe 2.4.5). Du point de vue pratique, cela consiste à mettre en place un tampon mémoire dans lequel on va successivement écrire des données à l'itération i et les lire à l'itération $i+1$.

Les graphes flots de données utilisés par SynDEx possèdent un constructeur spécifique nommé *memory*[5] permettant d'implanter le mécanisme de stockage que nous venons de décrire. Contrairement aux autres opérateurs flots de données, le retard consomme une donnée sur son arc d'entrée après avoir produit sur son arc de sortie la donnée stockée à l'itération précédente[GLS98].

Ainsi, un simple opérateur retard permet d'exprimer le squelette ITERMEM sous la forme d'un graphe flot de données comme indiqué sur la figure 4.7.

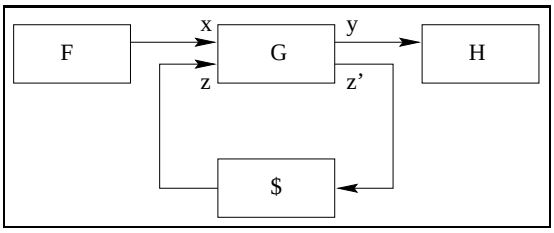


Figure 4.7: Représentation flot de données du squelette ITERMEM

[5]Correspondant au z^{-1} en traitement du signal.

4.3.3.4 Cas des squelettes "dynamiques"

En ce qui concerne les squelettes DF et TF, il est impossible de représenter directement leurs graphes de processus associés sous la forme de graphes flot de données statiques "digérables" par SynDEx. En effet, de tels squelettes sont basés sur un schéma d'implantation de type ferme de processeurs (cf. paragraphes 2.4.3 et 2.4.4). Une ferme de processeurs est par nature fondée sur un ordonnancement dynamique des communications. Le nombre et l'ordonnancement des communications entre le maître et les esclaves dépendent en effet intrinsèquement des données d'entrée et de fait ne peuvent pas être prédites lors de la phase de placement-ordonnancement et de génération de code cible.

Pour illustrer ces aspects, considérons le graphe de processeurs illustré sur la figure 4.8 représentant simplement et classiquement une ferme de processeurs.

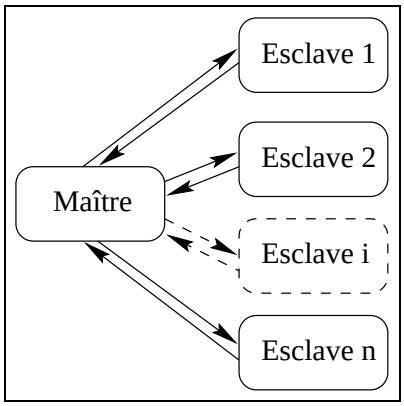


Figure 4.8: Graphe de processus d'une ferme de processeurs

Dans le cas général, le graphe est composé de $n + 1$ processus de calcul (un maître et n esclaves) et de $2n$ canaux de communications servant respectivement à acheminer les données du maître vers les esclaves et à rapatrier les résultats des esclaves vers le maître. Pour mettre en évidence les différents types de communication transitant sur ces canaux, nous pouvons décomposer le processus maître en trois sous-processus s'enchaînant séquentiellement :

① le premier d'entre eux a pour unique tâche d'initialiser la ferme de processeurs ce qui se traduit au niveau communication par l'envoi à chaque esclave d'un message de synchronisation de départ, leur indiquant de se mettre en attente de données à traiter (cf. figure 4.9a),

② le processus intermédiaire réalise effectivement la tâche de "Farming" en distribuant dynamiquement les données aux esclaves et en collectant les résultats correspondants (cf. figure 4.9b).

③ le processus final réceptionne les messages de synchronisation de fin de traitement provenant de l'ensemble des esclaves (cf. figure 4.9c).

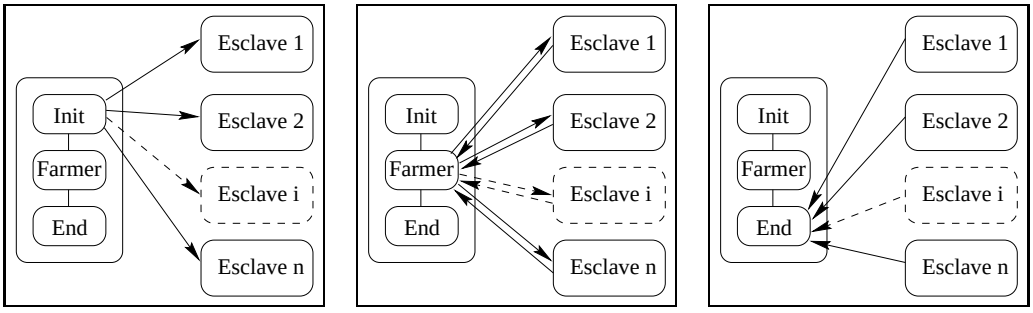


Figure 4.9a Figure 4.9b Figure 4.9c

Figure 4.9: Décomposition du processus maître

De cette décomposition, il ressort que seul le processus intermédiaire est caractérisé par des communications dynamiques entre le maître et les esclaves. Ne pouvant apparaître dans le graphes flot de données, elles doivent dès lors être "masquées", le graphe ne servant alors qu'à traduire les synchronisations (statiques cette fois) de début et de fin de fonctionnement de la ferme de processeurs. Ce principe est illustré sur la figure 4.10 avec un ensemble de quatre esclaves. Les communications dynamiques n'étant pas des dépendances de données apparaissant en pointillés dans le graphe flot de données.

Les communications dynamiques entre le maître et les esclaves sont cachées dans les fonctions génériques *farmer* et *worker*. Ces fonctions sont génériques dans le sens où, durant la phase de génération de code cible, elles sont construites à partir de harnais paramétrables dans lesquels le compilateur va insérer les appels de fonctions utilisateurs et les communications dynamiques. Ces harnais sont largement décrits dans les paragraphes 4.4.2 et 4.4.3.

Ce masquage des communications dynamiques impose une forte contrainte au niveau du placement-ordonnancement sous SynDEx. En effet, le temps d'exécution des fonctions effectuant ces communications étant inconnu *a priori* , il est obligatoire de contraindre l'heuristique de placement-ordonnancement. Pour cela, la définition opérationnelle des squelettes DF

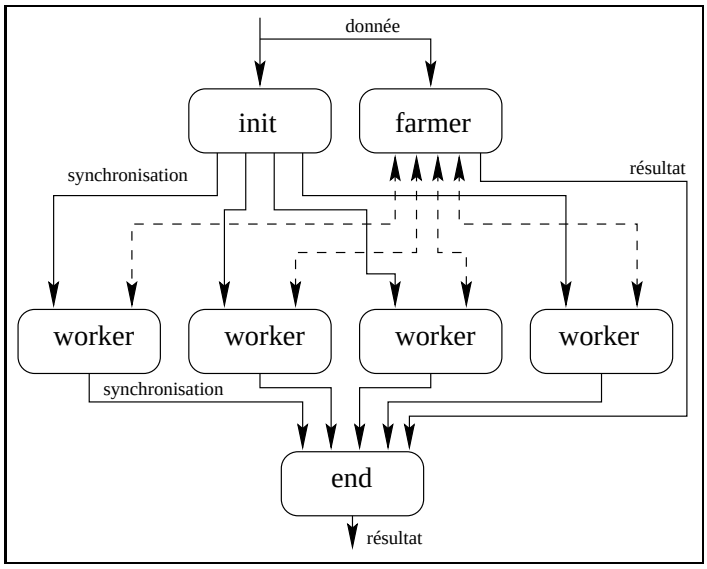


Figure 4.10: Représentation "flot de donnée" d'une ferme de processeurs

et TF comprend pour chaque noeud du graphe le numéro du processeur sur lequel la fonction sera placée.

4.3.3.5 Conclusion

En conclusion, pour pouvoir répondre aux exigences statiques de SynDEx, le graphe généré par le compilateur fonctionnel *Dromadaire* peut être vu comme un graphe "pseudo" flot de données.

Les noeuds le constituant représentent soit des fonctions séquentielles de calcul développées par l'utilisateur (cas des noeuds *split*, *compute* et *merge* du squelette SCM), soit des processus de contrôle encapsulant des schémas de communications dynamiques non explicités (cas des noeuds *farmer* et *worker* des squelettes DF et TF).

Les arcs associés, quant à eux, décrivent les communications entre ces sommets exprimant ainsi les dépendances statiques de données entre les opérations de calcul du graphe.

4.3.4 Génération de code cible

4.3.4.1 Principes de l'exécutif généré

Dans le cadre de la méthodologie A^3 supportée par SynDEx, les décisions d'allocation des opérations de calcul et de communications sont prises lors

de la phase de placement-ordonnancement. Dès lors, SynDEX peut générer un exécutif principalement statique[6]. Cette caractéristique apparaît comme essentielle dans le contexte des applications embarquées à fortes contraintes temporelles. En effet, il semble judicieux d'éviter les exécutifs dynamiques entraînant la mise en œuvre d'une couche logicielle supplémentaire gérant les aspects dynamiques de l'application et dont le surcoût temporel n'est pas à négliger.

De plus, l'exécutif généré reflète fidèlement les décisions prises par l'heuristique de placement-ordonnancement. Ceci se traduit sous la forme d'un code distribué sur chaque processeur effectuant d'une part les opérations de calculs et d'autre part les opérations de communications. Ce code distribué est donc généré sur mesure pour l'algorithme et l'architecture et n'est pas conçu comme un noyau séparé de l'application et chargé indépendamment de celle-ci[GLS98].

Enfin, pour que l'exécutif généré soit facilement portable en cas de changement d'architecture, celui-ci a été divisé en deux parties distinctes :

- un macro-code générique décrivant pour chaque processeur l'ordonnancement des opérations sous la forme d'appels de macro-définitions génériques de calcul et de communication,
- un noyau d'exécutif contenant le jeu de macro-définitions spécifiques à une architecture donnée.

La transformation du macro-code en code compilable et implantable sur l'architecture cible se fait à l'aide du macro-processeur *m4* de Unix qui remplace les appels de macro-définitions de calcul et de communication par leurs définitions spécifiques à l'architecture.

4.3.4.2 Structure de l'exécutif

Les algorithmes implantés sont par nature itératifs : un ensemble d'opérations de calcul est répété jusqu'à ce qu'il soit décidé de mettre fin à l'application. La distribution et l'ordonnancement de ces opérations sur plusieurs processeurs se traduisent par une séquence itérative de calcul sur chaque processeur. Au sein de cette séquence, l'ensemble des opérations devant être exécutées lors d'une itération de l'application est parfaitement connu. Chacune de ces opérations nécessite en entrée des valeurs produites par d'autres calculs et fournit en sortie des résultats destinés à d'autres

[6]La faible partie dynamique correspond à l'évaluation des booléens de conditionnement.

opérations. Dès lors qu'un calcul produit une valeur pour un autre calcul et que ses deux opérations ne sont pas exécutées sur le même processeur, il faut que la valeur produite soit transférée de la mémoire du processeur émetteur vers celle du processeur récepteur. Toutes les informations relatives à ce transfert — type et taille des données, adresse mémoire contenant la donnée, calcul ayant produit la donnée, adresse mémoire devant recevoir la donnée, calcul devant consommer la donnée, etc.— sont donc parfaitement identifiées. A l'instar des calculs, le placement-ordonnancement des communications se traduit par une séquence de communications sur chaque lien de transfert au sein de laquelle toutes les opérations sont fixées à la compilation.

Ainsi, chaque processeur se verra confier la charge de gérer :

① une séquence de calcul,

② n séquences de communication[7].

4.3.4.3 Synchronisation des calculs et des communications

Le parallélisme, dans un processeur de type Transputer, se situe au niveau des unités fonctionnelles distinctes : une unité de calcul arithmétique et logique (ALU) pour le calcul et une unité de transfert (DMA) pour chaque liaison physique de communication. La séquence de calcul et les séquences de communication exécutées sur un même processeur partagent l'unique séquenceur du processeur et sont exécutées de manière concurrente.

Toutefois, une communication ne requiert qu'épisodiquement le séquenceur d'instructions pour charger les registres du DMA avec l'adresse de base et la taille de la zone mémoire à transférer à travers la liaison physique de communication. Une fois activé, le DMA séquence seul les accès mémoire jusqu'à la fin du transfert laissant la possibilité au CPU d'effectuer des calculs pendant tout le temps de transfert de données sur le lien physique.

Il est donc indispensable de mettre en place un mécanisme permettant de bloquer ou libérer les séquences. Ce mécanisme d'arbitrage du séquenceur a pour but de synchroniser les processus associés à chaque séquence de calcul et de communication. Un ensemble de primitives de synchronisation développées en assembleur[8] gère le passage d'une séquence à l'autre. Les séquences de communications sont utilisées prioritairement aux séquences de calcul afin de pénaliser le moins possible les opérateurs de calcul demandeurs

[7] n représente le nombre de liens de communications par processeur. Dans le cas d'un Transputer, n a pour valeur 4.

[8] Pour des raisons d'efficacité.

de données. Cependant, cette allocation prioritaire doit se limiter au strict minimum. Pendant qu'une communication attend la fin d'une opération de calcul, il faut que la séquence de calcul soit active pour qu'effectivement la donnée attendue par la séquence de communication soit produite. De même, pendant que le DMA gère les transferts, il faut que la séquence de calcul soit également active pour pouvoir tirer pleinement parti des opportunités de recouvrement calcul-communication. Ainsi, l'activation de la séquence de communication ne doit avoir lieu que lorsque la donnée est prête à être transférée et que le média de communication est libre pour effectuer ce transfert.

En pratique, les opérateurs de communication sont conçus au niveau matériel pour requérir le séquenceur d'instructions en fin de transfert ce qui entraîne une sauvegarde du contexte du séquenceur et le démarrage du programme d'interruption associé à la communication. De fait, il n'est pas nécessaire d'allouer le séquenceur d'instructions à une séquence de communication pendant les périodes d'attente de fin de transfert ou de synchronisation. Ainsi, la séquence de calcul ne sera interrompue automatiquement que le temps nécessaire soit à la programmation du DMA effectuant la communication sur le lien physique, soit à l'exécution des synchronisations entre les différentes séquences de communication et de calcul.

Les synchronisations traduisent les dépendances de données entre les opérations de calcul et de communication. Pour chaque dépendance, est mis en place un couple de primitives de synchronisation nommé respectivement *Pre* du côté de l'opération productrice de la donnée et *Suc* du côté de l'opération consommatrice. La primitive *Pre* est de type *passant* puisqu'elle peut s'exécuter indépendamment de l'état de la primitive *Suc* associée. Par contre, la primitive *Suc* est de type *bloquant* puisque pour s'exécuter, elle est obligée d'attendre la fin de la primitive *Pre* correspondante.

Etant donné qu'il existe une différence de priorité d'exécution entre la séquence de calcul et les séquences de communication, les primitives *Suc* doivent obligatoirement posséder un comportement différent selon les séquences (calcul ou communication) au sein desquelles elles sont appelées. Du côté calcul, la primitive *Suc* peut attendre activement car la fin du transfert de la donnée va requérir automatiquement le séquenceur d'instructions pour exécuter la primitive *Pre* associée. A l'opposé, du côté communication, la primitive *Suc* doit obligatoirement faire de l'attente passive, c'est-à-dire ne pas monopoliser le séquenceur d'instructions, pour que la séquence de calcul puisse exécuter le *Pre* correspondant. De fait, cette dissymétrie impose l'implantation de deux couples distincts de primitives *Pre0-Suc0* et *Pre1-Suc1* [GLS98], les indices 0 et 1 correspondant respectivement aux priorités

basses et hautes des séquences de calcul et de communications.

A partir de là, il existe quatre cas de synchronisation traduisant :

① une précédence communication(*Pre0*)-calcul(*Suc0*),

② une précédence calcul(*Pre1*)-communication(*Suc1*),

③ une précédence communication(*Pre1*)-communication(*Suc1*),

④ une précédence calcul(*Pre0*)-calcul(*Suc0*).

Le premier cas est le plus simple. Il correspond à l'attente par la séquence de calcul d'une donnée transmise par un lien de communication. Comme la séquence de communication est exécutée sur interruption de la séquence de calcul, *Suc0*, côté calcul, n'a rien de mieux à faire qu'à attendre activement la fin de la communication et l'exécution du *Pre0* associé.

Le deuxième cas est plus complexe. La synchronisation *Suc1* de la séquence de communication ne peut attendre activement ce qui aurait pour conséquence directe un blocage mortel puisque l'exécution de la synchronisation associée *Pre1* ne pourrait avoir lieu. En effet, dans le cas où la séquence de communication exécute *Suc1* avant que la séquence de calcul exécute le *Pre1* correspondant, la séquence de communication doit suspendre son exécution pour permettre à la séquence de calcul d'exécuter la fin de son calcul. Par la suite, l'exécution de *Pre1* permet de relancer la séquence de communication et d'exécuter le transfert de données.

Le troisième cas est dédié aux routages de transferts de données inter-processeurs. Les deux séquences de communication se synchronisent directement sans passer par la séquence de calcul. L'utilisation du couple de synchronisation *Pre1*-*Suc1* est obligatoire puisque les séquences de communication ne peuvent effectuer de l'attente active.

Enfin, le dernier cas correspond aux transferts de données par mémoire partagée entre séquences de calcul exécutées sur différents processeurs. Chaque séquence pouvant faire de l'attente active, on utilise le couple de synchronisations *Pre0*-*Suc0*.

Pour illustrer les mécanismes de synchronisation mis en œuvre entre les séquences, nous allons reprendre l'exemple de l'application de calcul de gradient décrit au paragraphe 3.3.1 dans le cas simplifié d'une division en deux bandes.

Le placement-ordonnancement d'une telle application sur deux processeurs est représenté sur la figure 4.11. Le placement est représenté sur

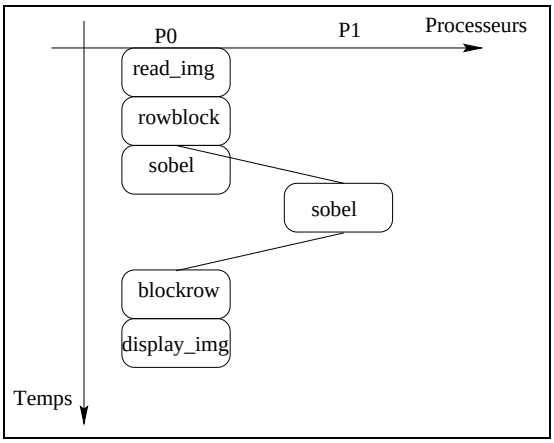


Figure 4.11: Placement-ordonnancement sur deux processeurs

une échelle horizontale sur laquelle on affecte une colonne par processeur. L'ordonnancement est décrit verticalement par une échelle temporelle.

L'exécutif généré par SynDEx conduit à la mise en place sur chaque processeur de deux séquences distinctes, une étant réservé aux calculs, l'autre aux communications.

L'extrait de macro-code suivant montre une situation de synchronisation de type *Pre1-Suc1* entre la séquence de communication et la séquence de calcul exécutées par le processeur P0. Cette synchronisation traduit une précédence calcul-communication. En effet, la donnée *rowblock4_s1Image*, issue de la division de l'image par la fonction de calcul *rowblock*, ne peut être transférée vers le processeur P1 que lorsque l'opération de division de l'image est achevée.

```
comment{Extraits de la sequence de communication}comment
.......
Suc1_(NV1_rowblock4_s1Image_full)
PPL_o_integer(3,1,rowblock4_s1Image)
.......
comment{Extraits de la sequence de calcul}comment
.......
rowblock( rowblock4_1, readimg2_s0Image,
          rowblock4_s0Image, rowblock4_s1Image)
Pre1_(NV1_rowblock4_s1Image_full)
.......
```

De même, au niveau de la phase de fusion des bandes d'images traitées,

la fonction de calcul *blockrow* ne peut avoir lieu avant que le processeur P1 n'ait envoyé la donnée *sobel6_s0Image*. Nous sommes ici dans le cas d'une précédence communication-calcul qui se traduit par l'utilisation d'un couple de primitives de synchronisation *Pre0-Suc0*.

```
comment{Extraits de la sequence de communication}comment
.......
PPL_i_integer(3,1,sobel6_s0Image)
Pre0_(root_sobel6_s0Image_full)
.......
comment{Extraits de la sequence de calcul}comment
.......
Suc0_(root_sobel6_s0Image_full)
blockrow( blockrow7_1, sobel5_s0Image,
          sobel6_s0Image, blockrow7_s0Image)
.......
```

4.3.5 Mesure de performances

La mesure de performances de l'application a pour objectif de collecter des informations temporelles relatives à l'exécution de l'algorithme sur l'architecture. Ces mesures permettent entre autres d'analyser le comportement temps réel de l'application, de vérifier que les performances prédites correspondent à la réalité, de caractériser précisément les fonctions de calcul utilisateur, de confirmer ou infirmer les choix de parallélisation, etc.

Ces mesures reposent sur la génération d'un exécutif instrumenté dans lequel sont insérées des macro-opérations de chronométrage permettant de stocker les dates de début et de fin de toutes les fonctions de calcul utilisateur présentes dans l'application.

La génération d'un tel exécutif est une option paramétrable de l'outil SynDEx. Le chronométrage d'une application repose sur quatre phases distinctes :

① une phase d'*initialisation* du tampon mémoire de stockage des informations effectuée préliminairement à l'exécution des itérations de l'application,

② une phase de *mesure* des dates de début et de fin des opérations de calcul durant les itérations de l'application. Ces mesures sont relatives à l'horloge locale de chaque processeur,

③ une phase de mesure des *décalages* des horloges des processeurs afin de rendre compatibles les mesures effectuées sur des processeurs différents,

④ une phase de *collecte* de l'ensemble des mesures dont le but est de rapatrier les informations vers le processeur hôte.

L'approche que nous avons choisie pour la mesure des performances repose sur les principes énoncés ci-dessus. Cependant, dans le cas de SynDEx, l'exécutif instrumenté est uniquement généré optionnellement ce qui impose à l'utilisateur de disposer de deux versions différentes de son application. La première version exécute l'application itérativement alors que la deuxième version instrumentée exécute uniquement un nombre fixe d'itérations et fournit en sortie les informations de chronométrage. Pour éviter à l'utilisateur de gérer simultanément ces deux exécutifs, nous avons adopté une démarche différente. L'utilisateur ne génère qu'un seul exécutif dans lequel le générateur de code cible insère automatiquement des macro-opérations de mesure — similaires à celles que peut générer SynDEx — autour de chaque opération de calcul. Le choix par l'utilisateur d'une exécution instrumentée ou non se fait alors à la compilation du code cible sous la forme d'une validation ou non d'une directive de compilation.

Dans notre approche, l'exploitation des mesures effectuées se fait sous la forme graphique en ayant recours à un utilitaire existant nommé *Upshot*. Cet outil permet une visualisation et une analyse aisée des mesures de performances. Un exemple d'une exécution instrumentée d'applications est donné sur la figure 4.12. L'échelle horizontale définit le temps alors que l'échelle verticale représente les différents processeurs sur lesquels s'exécutent les fonctions de calcul. Chacune d'entre elles est représentée par une boite coloriée de taille proportionnelle à la durée d'exécution mesurée.

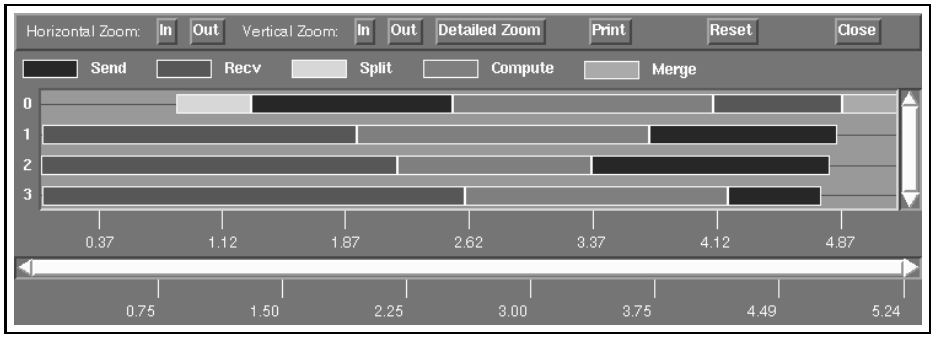


Figure 4.12: Un exemple de visualisation de performances sous *Upshot*

4.4 Implantation des squelettes et modèles de performances

Les squelettes de parallélisation encapsulent des schémas de calcul-communication parfaitement identifiés. A partir de là, l'implantation résultante sur une architecture donnée est également connue. De fait, celle-ci peut être étudiée et modélisée sous la forme d'une formule analytique permettant de prédire précisément le comportement temporel des applications. Concrètement, cela consiste à identifier l'ensemble des paramètres — algorithmiques et architecturaux — pour chaque couple squelette-machine caractérisant la durée d'exécution des applications.

Dans notre cas, cette prédiction de performances se traduit sous la forme d'une équation exprimant la latence de l'implantation de chaque squelette pour une topologie donnée en fonction des coûts de traitement et de transfert des données.

En règle générale, de tels coûts sont difficilement estimables à partir des seules informations provenant du code source des fonctions développées par l'utilisateur[9]. Pour résoudre un tel problème, chaque application candidate peut passer par une phase d'instrumentation séquentielle sur un unique processeur. A partir de là, les paramètres des modèles analytiques de performances sont aisément déductibles directement pour les coûts de traitements et indirectement pour les coûts de transferts à partir des volumes de données échangées entre les opérateurs de calcul[10].

Dans les paragraphes suivants, nous allons donc décrire précisément les implantations des squelettes SCM, DF, TF et ITERMEM et en déduire pour chacun d'entre eux un modèle analytique de performances. A l'heure actuelle, les applications implantées ont pour plate-forme cible la machine Transvision composée de Transputers. Etant donné que ces processeurs possèdent un nombre limité de liens physiques de communication, une topologie en anneau — telle que celle décrite sur la figure 4.3 — possède l'avantage d'être arbitrairement extensible et possède un degré[11] fixe et égal à deux.

Ainsi, les modèles de performances que nous avons définis ne sont valables que pour une topologie en anneau et doivent être reformulés pour tout autre

[9]En particulier, dans le cas où la complexité de traitement est liée à la nature des données comme dans l'algorithme d'approximation polygonale de chaînes de points décrit au paragraphe 1.2.1.1.

[10]En utilisant une formule du type $T_{Transfert} = T_{Init} + \frac{Volume}{Débit}$. où T_{Init} est le temps d'initialisation de la communication et *Débit* représente le débit du lien de communication exprimé en octets par seconde

[11]Le degré d'un processeur est le nombre de liens de communications utilisés.

configuration architecturale.

4.4.1 Cas du squelette SCM

4.4.1.1 Implantation du squelette SCM

Le squelette SCM est caractérisé par un comportement opérationnel statique qui a été largement décrit au paragraphe 2.4.2. De fait, le placement-ordonnancement du graphe flot de données associé est géré entièrement par l'heuristique de SynDEx et donne le résultat suivant :

- enchaînement séquentiel des opérateurs *split*, *compute* et *merge* sur le processeur dit *root*,
- allocation d'un seul opérateur *compute* sur les autres processeurs dits de *traitement*.

Pour pouvoir définir un modèle analytique de performances, il est aussi indispensable de déterminer précisément le placement-ordonnancement des communications traduisant les dépendances de données entre les opérations de calcul.

Dans le cas d'une architecture "idéale" dans laquelle le processeur *root* est connecté directement à l'ensemble des processeurs de traitement, cette détermination est triviale. En effet, la diffusion des partitions issues de l'opérateur *split* et la collecte des résultats destinés à l'opération *merge* sont réalisées en parallèle sur chacun des liens du processeur *root*[12]. Cependant, une telle configuration est rapidement limitée au niveau de l'extension du nombre de processeurs de traitement. Par exemple, une configuration de ce type à base de Transputers comprend au maximum cinq processeurs (dont le processeur *root* placé au centre).

Il apparaît donc que, dans la majorité des cas, le nombre de liens de communication du processeur *root* est inférieur au nombre de processeurs de traitement. C'est en particulier le cas des architectures organisées en anneau pour lesquelles nous définissons le modèle de performances. Pour de telles configurations, chaque processeur utilise uniquement deux liens de communication les connectant avec leurs deux plus proches voisins. Ainsi, les transferts de données entre processeurs non directement connectés doivent inévitablement transiter par d'autres processeurs servant de routeurs de message.

[12]Ceci n'est évidemment possible que si chaque lien possède son propre DMA comme dans le cas des Transputers.

Le placement et l'ordonnancement des communications sur les liens des processeurs sont alors directement déductibles des principes régissant l'heuristique de SynDEx. En effet, la fonction de coût utilisée a pour objectif de minimiser la latence en tenant compte simultanément des durées d'exécution des calculs et des transferts de données entre les processeurs (via éventuellement des processeurs de routage).

Dès lors, pour minimiser la durée des transferts entre le processeur *root* et les processeurs de traitement, l'heuristique place toujours les communications sur la route la plus courte dans le cas où plusieurs routes sont possibles[13]. Ainsi la diffusion des paquets de données (respectivement la collecte des résultats) sera répartie pour une moitié sur le lien "gauche" et pour l'autre moitié sur le lien "droit" du processeur *root*.

L'ordonnancement des communications peut également se déduire des principes de l'heuristique. En effet, lors de la phase de placement-ordonnancement des opérations de calculs, l'heuristique évalue pour chaque couple calcul-processeur de combien est rallongé le temps d'exécution de l'algorithme et place systématiquement celui qui minimise la fonction de coût. De fait, les opérateurs *compute* sont toujours placés en premier sur les processeurs les plus proches du processeur *root* puisque, dans ce cas là, on minimise le temps de transfert relatif à l'acheminement des données. Ceci implique donc que les communications à destination des processeurs les plus proches du *root* sont toujours ordonnancées avant celles devant transiter par un plus grand nombre de processeurs intermédiaires.

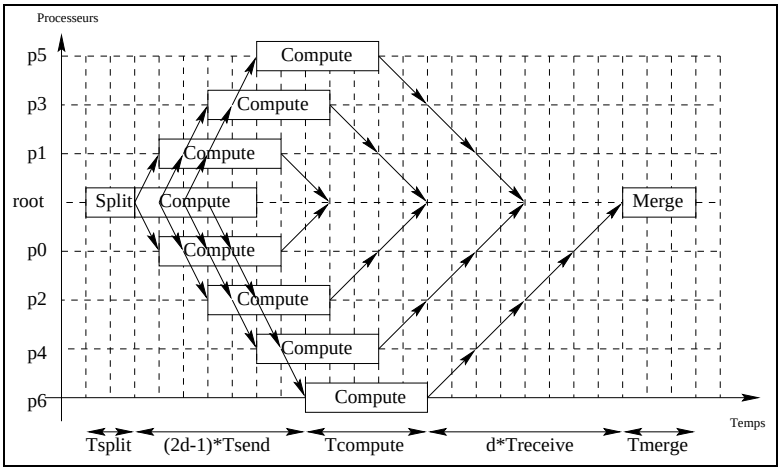


Figure 4.13: Profil d'exécution du squelette SCM

[13]Sur une architecture en anneau, il existe deux routes possibles pour transférer des messages entre deux processeurs.

Toutes ces caractéristiques sont synthétisées sur la figure 4.13 décrivant un profil typique d'exécution du squelette SCM sur un anneau comportant huit processeurs. L'échelle verticale représente les processeurs alors que l'échelle horizontale décrit le temps. Afin de faciliter la compréhension d'un tel profil, le temps est découpé en pas élémentaire et toutes les durées de calcul et de communications sont exprimées sous la forme de multiples de ce pas.

Chaque opération de calcul (*split*, *compute* et *merge*) est représentée par une boite dont la longueur est proportionnelle au temps d'exécution de la fonction séquentielle associée. De même, les communications sont symbolisées par des flèches de taille variable. Ces communications sont de type point à point : par exemple entre le processeur *root* et le processeur p_6, il faut quatre étapes de communication pour acheminer les données ou rapatrier les résultats.

De plus, ce profil d'exécution illustre le principe de recouvrement calcul-communication qui est traduit dans l'exécutif par la mise en œuvre de séquences concurrentes de calcul et de communication dont l'ordonnancement est géré par les primitives de synchronisations décrites au paragraphe 4.3.4.3. On constate par exemple que la fonction *compute* sur le processeur *root* s'exécute concurremment avec les émissions de données vers les processeurs w_i. En fait, la séquence de calcul sur laquelle s'exécute la fonction *compute* est interrompue par les séquences de communications effectuant les émissions des paquets de données. Néanmoins, le temps d'interruption peut être considéré comme négligeable puisqu'il correspond à la programmation du DMA.

Enfin, on peut remarquer l'ordonnancement séquentiel des communications sur un même lien physique. Par exemple, le transfert de la donnée à destination du processeur p_1 est exécuté avant celui à destination du processeur p_3 ce qui explique le décalage temporel entre les deux communications.

4.4.1.2 Modèle de performances du squelette SCM

A partir de ce profil d'exécution, il est direct de déduire une formule analytique[14] donnant la latence du squelette SCM en fonction des temps d'exécution des fonctions de calcul et des temps de transfert des données :

$$\begin{cases} T_{SCM} = T_{Split} + T_{Compute} + T_{Merge} + T_{Comm} \\ T_{Comm} = (2d-1) * T_{Send} + d * T_{Recv} \end{cases} \quad (4.1)$$

[14]Notons que le modèle de performances présenté ici est en tout point similaire avec les prévisions temporelles que peut donner l'heuristique de placement-ordonanncement de SynDEX.

avec

- n est le nombre de partitions de données (équivalent au nombre de processeurs),
- d est le diamètre de l'anneau c'est-à-dire la distance maximale séparant deux processeurs qui peut être définie ainsi :

$$\begin{cases} d & = \frac{n}{2} \quad si\ n\ est\ pair, \\ & = \frac{n-1}{2} \quad si\ n\ est\ impair. \end{cases} \tag{4.2}$$

- T_{Split} est le temps d'exécution de la fonction *split*,
- $T_{Compute}$ représente le temps de traitement de la fonction *compute* une partition. Pratiquement, $T_{Compute}$ est égal à $\frac{T_{Cseq}}{n}$ où T_{Cseq} est le temps de calcul de la fonction *compute* opérant sur l'ensemble des données initiales,
- T_{Merge} est le temps d'exécution de la fonction *merge*,
- T_{Send} (respectivement T_{Recv}) est le temps de transfert d'un paquet de données (respectivement de résultat) sur un lien point à point.

En pratique, les durées d'exécution des fonctions de calcul T_{Split}, T_{Cseq} et T_{Merge} sont obtenus après une phase d'instrumentation séquentielle. Les temps de communication T_{Send} et T_{Recv} sont déduits de la formule donnée au paragraphe 4.4.

4.4.2 Cas du squelette DF

4.4.2.1 Protocoles de communication du squelette DF

Comme annoncé au paragraphe 4.3.3.4, l'implantation du squelette DF est plus complexe que celle du squelette SCM puisqu'elle implique un ordonnancement dynamique des communications encapsulées dans les fonctions génériques *farmer* et *worker*.

Par ailleurs, le réseau de communication à mettre en œuvre pour la distribution des données puis la collecte des résultats nécessite que tous les processeurs esclaves puissent communiquer avec le processeur maître. Comme dans le cas précédent, certains esclaves ne sont pas reliés directement au maître.

De ce fait, chaque processeur (esclave ou maître) doit pouvoir gérer constamment un message venant sur un de ses liens de communication tout en

ayant en charge le traitement d'une donnée. Une solution possible est de séparer la phase de communication de celle des traitements au sein des fonctions génériques *farmer* et *worker* en créant deux séquences distinctes (une de calcul et une de communication) se synchronisant par le biais de primitives identiques à celles décrites au paragraphe 4.3.4.3. Par conséquent, chaque processeur se verra confier les deux séquences suivantes (cf. figure 4.14) :

- une séquence de communication gérant l'ensemble des transferts de données avec l'environnement proche (*i.e.* les séquences de communication des processeurs directement connectés). Recevant un message, cette séquence le propage via le lien adéquat suivant l'adresse du destinataire ou indique à la séquence de traitement la disponibilité de la donnée à traiter.
- une séquence de calcul recevant de la séquence de communication précédemment décrite les données, effectuant sur ces données un ensemble de traitements et indiquant à la séquence de communication la disponibilité des résultats correspondants à émettre à destination du maître.

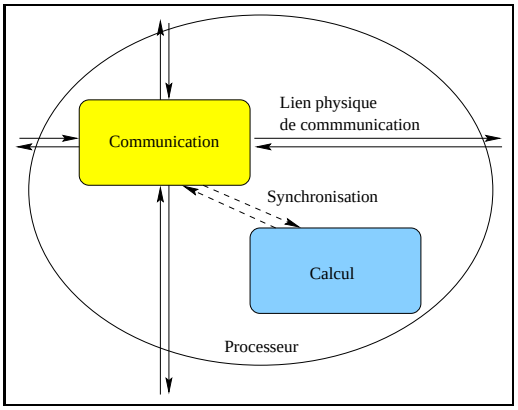


Figure 4.14: Séparation des calculs et des communications dynamiques dans un squelette DF

L'inconvénient d'une telle stratégie est qu'elle peut conduire à l'apparition d'une situation d'interblocage entre deux processeurs directement connectés. Ce cas se produit si deux processeurs veulent émettre un message l'un vers l'autre au même instant. Ces messages sont de nature différente : l'un a comme source le maître et véhicule des données vers un processeur esclave, l'autre a comme source un processeur esclave et véhicule des résultats à destination du maître. Pour empêcher l'apparition simultanée de deux messages

de nature différente, il est donc indispensable de définir un protocole de communication dynamique adéquat. Pour cela, deux situations sont envisageables :

① se restreindre à une topologie de l'architecture telle qu'aucun processeur ne soit un processeur intermédiaire entre le maître et un esclave. Une telle architecture est forcément rapidement limitée par le nombre de liens de communications du maître.

② gérer simultanément la diffusion des données et le rassemblement des résultats en séparant les chemins de communications gérant les messages provenant du maître et ceux provenant des esclaves. Dans [Nic92], deux approches sont proposées : une effectuant une séparation physique des chemins de communication, l'autre une séparation temporelle.

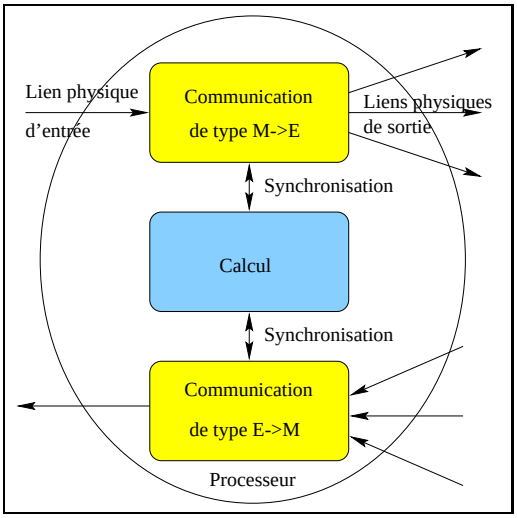


Figure 4.15: Séparation physique des chemins de communications dans un squelette DF

Nous avons opté pour la solution effectuant une séparation physique des chemins de communications puisqu'elle évite tout risque d'interblocage tout en restant relativement simple à mettre en œuvre. La gestion simultanée de la diffusion des données et de la collecte des résultats repose sur la mise en place de deux séquences de communication au lieu d'une seule sur chaque processeur (cf. figure 4.15) :

➟ la première séquence gère la diffusion des messages de type maître vers esclaves (M→E),

- la deuxième séquence gère l'acheminement des messages de type esclaves vers maître (E→M).

Dans le cas d'une architecture organisée en anneau, le nombre de liens utilisés par les séquences de communication est limité à un seul lien d'entrée et un seul lien de sortie comme indiqué sur la figure 4.16.

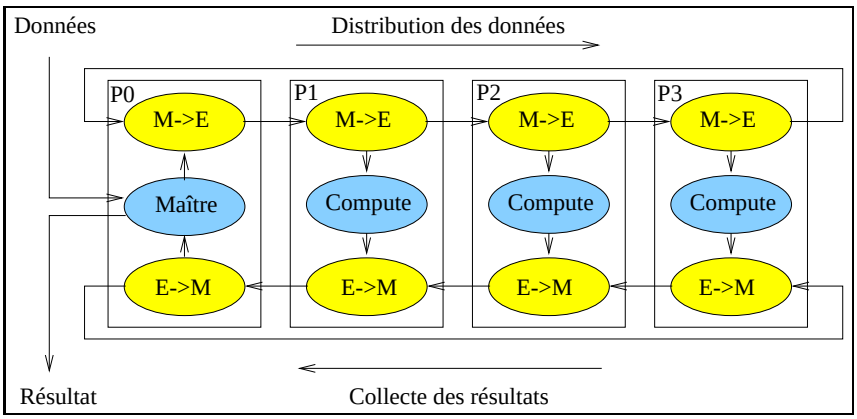


Figure 4.16: Implantation du squelette DF sur un anneau à quatre processeurs

4.4.2.2 Implantation du maître

4.4.2.2.1 Séquence de calcul

Dans cette approche de séparation physique des chemins de communication, la séquence de calcul du processeur maître a pour tâches principales la gestion de la distribution des données (en coopération avec la séquence de communication M→E) ainsi que l'accumulation des résultats provenant des processeurs esclaves (en coopération avec la séquence de communication E→M).

Le comportement du maître est fonction d'un certain nombre de conditions relatives au nombre de données restant à traiter, au nombre de résultats non encore retournés et au nombre de processeurs esclaves non occupés par un traitement.

Soient C_1 la condition signifiant *"il reste des données"*, C_2 la condition exprimant *"il reste des résultats"* et C_3 la condition représentant *"il reste des esclaves"*. La combinaison de ces trois conditions permet d'établir un scénario de fonctionnement de la séquence de calcul du maître reposant sur quatre phases distinctes s'enchaînant séquentiellement (cf. figure 4.17) :

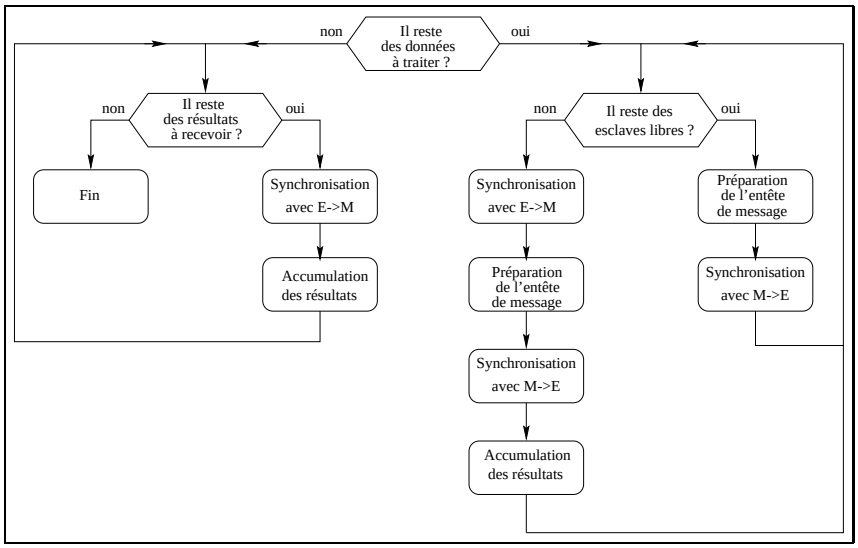


Figure 4.17: Organigramme de fonctionnement de la séquence de calcul du maître

① une phase d'initialisation (conditions C_1 et C_3 validées) dans laquelle le maître va successivement allouer une donnée à chacun des processeurs esclaves,

② une phase de régime permanent (conditions C_1 et $\overline{C_3}$ validées) dans laquelle le maître attend une réception de résultats pour successivement distribuer une nouvelle donnée au processeur qui est devenu demandeur en lui émettant ses résultats et accumuler le résultat,

③ une phase de terminaison de réception de résultats (conditions $\overline{C_1}$ et C_2 validées) dans laquelle le maître ne possède plus de données à distribuer mais attend en contrepartie des résultats provenant des processeurs esclaves pour pouvoir les accumuler,

④ une phase de fin de calcul (conditions $\overline{C_1}$ et $\overline{C_2}$ validées) qui termine la phase de distribution dynamique de données lorsque tous les résultats issus des traitements de l'ensemble des données ont été accumulés.

4.4.2.2.2 Séquences de communication

Le comportement des séquences de communication est en relation directe avec celui de la séquence de calcul avec laquelle elles se synchronisent pour échanger des données.

Une communication d'un paquet de données (respectivement de résultat)

est toujours précédée par l'envoi d'un message d'entête contenant les informations nécessaires et suffisantes pour identifier la donnée : numéro du processeur émetteur, numéro du processeur destinataire, nature du message (de type donnée, résultat ou fin).

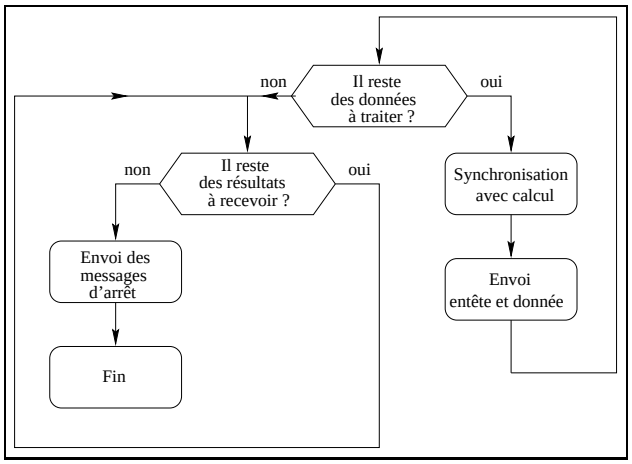


Figure 4.18: Organigramme de fonctionnement de la séquence M→E

Ainsi, dans les phases d'initialisation et de régime permanent, la séquence de communication M→E effectue successivement l'envoi d'une entête et d'une donnée après s'être synchronisée avec la séquence de calcul qui a initialisé l'entête. Enfin, lorsque tous les résultats des traitements ont été accumulés, cette même séquence de communication a pour rôle de propager à l'ensemble des processeurs esclaves un message de fin de traitement (cf. figure 4.18).

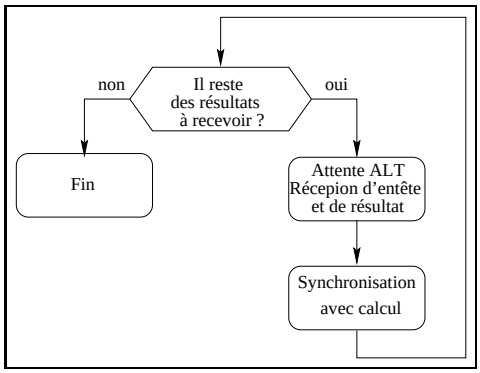


Figure 4.19: Organigramme de fonctionnement de la séquence E→M

En ce qui concerne la séquence de communication E→M, celle-ci est toujours en attente de réception de résultats en provenance de processeurs esclaves. Cette attente sur plusieurs canaux de communication impose le re-

cours à une fonction de réception non déterministe de type *ALT* dans le cas des Transputers. Celle-ci permet effectivement de recevoir des données sur un ensemble de liens de communications sans connaître *a priori* le nombre et l'ordonnancement des transferts sur les liens. Dès réception sur un des liens de communication, elle signale à la séquence de calcul par le biais d'une synchronisation la disponibilité du résultat devant être accumulé. Ce fonctionnement est réitéré tant qu'il subsiste des résultats qui n'ont pas été renvoyés par les esclaves (cf. figure 4.19).

4.4.2.3 Implantation des esclaves

L'implantation des trois séquences de calcul et de communication sur un processeur esclave est similaire à celle que nous venons de décrire pour le processeur maître. Toutefois, dans le cas des esclaves, il est en plus nécessaire de gérer les routages de données (dans la séquence M→E) et de résultats (dans la séquence E→M).

Le fonctionnement des séquences de calcul et de communication est réitéré jusqu'au moment où le processeur maître envoie à chacun des esclaves un message de fin indiquant la fin des traitements.

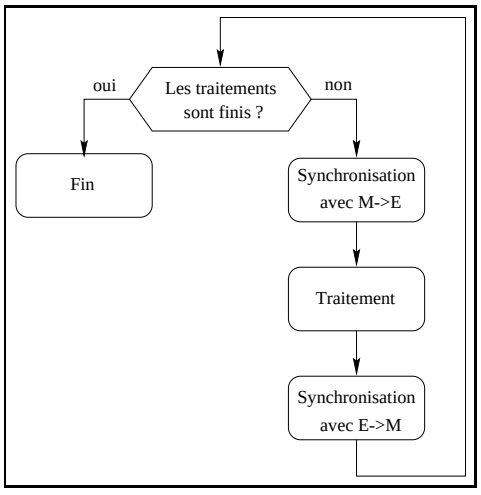


Figure 4.20: Organigramme de fonctionnement de la séquence de calcul

4.4.2.3.1 Séquence de calcul

Au niveau de la séquence de calcul, le comportement opérationnel est relativement simple. En effet, le traitement ne peut et ne doit avoir lieu que lorsqu'une donnée destinée à cette séquence est présente en mémoire d'où la

nécessité de bloquer la séquence de calcul par une synchronisation (libérée par la séquence de communication M→E après réception de la donnée). Après exécution du traitement, la séquence de calcul indique à la séquence de communication E→M la disponibilité du résultat à renvoyer en direction du maître. Ce comportement est illustré sur la figure 4.20.

4.4.2.3.2 Séquences de communication

Comme nous venons de le préciser, l'implantation des séquences de communication doit mettre en œuvre une stratégie de routage de messages à destination soit des autres processeurs esclaves pour l'acheminement des paquets de données, soit du processeur maître pour le renvoi des résultats correspondants.

Premièrement, la séquence de communication M→E est toujours en attente de réception d'un couple (entête, donnée) sur son lien d'entrée. Recevant l'entête, la séquence est en mesure de déterminer le destinataire. Dans le cas où celui-ci est la séquence de calcul, elle réceptionne la donnée et autorise le traitement à s'effectuer. A l'opposé, dans le cas où le destinataire est un autre esclave, la séquence de communication ne fait que router le message c'est-à-dire réceptionner la donnée et la renvoyer immédiatement via le lien adéquat à destination du bon processeur (cf. annexe A). A la fin des traitements de l'ensemble des données, cette même séquence de communication propage de manière similaire les messages de fin de traitement envoyés par le maître. Ce comportement peut être représenté sous la forme de l'organigramme de la figure 4.21.

Deuxièmement, la séquence de communication de type E→M gère l'envoi des résultats des traitements vers le processeur maître. Un résultat peut provenir soit de la séquence de calcul placée sur le même processeur, soit d'une autre séquence de communication de type E→M placée sur un processeur voisin[15].

Dans le premier cas, la séquence de communication prépare l'entête de message et envoie successivement vers le maître cette entête et le résultat de traitement.

Dans l'autre cas, la séquence de communication réceptionne l'entête et le résultat et les renvoie vers le maître (cf. figure 4.22).

[15]L'attente se fait simultanément — par le recours à une fonction *ALT* — sur les liens physiques pour les communications avec les autres esclaves et sur un canal interne pour la communication avec la séquence de calcul.

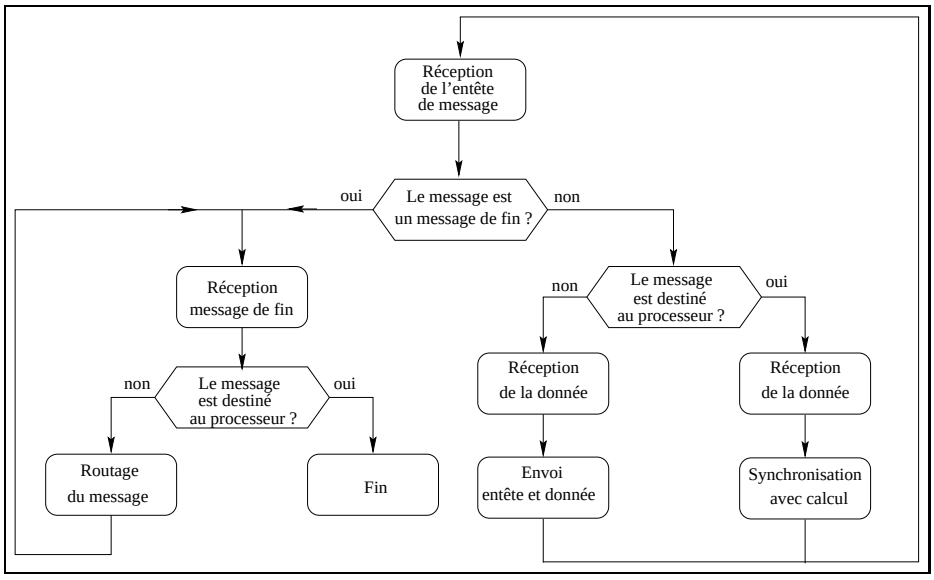


Figure 4.21: Organigramme de fonctionnement de la séquence M→E

4.4.2.4 Modèle de performances du squelette DF

Après avoir décrit fonctionnellement la stratégie d'implantation mise en œuvre au niveau du maître et des esclaves, il est désormais possible de définir un modèle analytique de performances — associé à une architecture en anneau — permettant de prédire le comportement temporel du squelette DF.

Les figures 4.23 et 4.24 décrivent des profils typiques d'exécution du squelette DF sur un anneau comprenant cinq processeurs (un maître et quatre esclaves) dans les deux principaux cas suivants :

① le temps d'exécution de la fonction d'accumulation des résultats partiels sur le maître est assez important pour provoquer une attente significative lors de la réception des résultats en provenance des processeurs esclaves, (figure 4.23),

② le temps d'exécution de la fonction d'accumulation est suffisamment faible pour que le maître soit obligé d'attendre les résultats à accumuler (figure 4.24).

Dans les deux cas décrits, nous pouvons effectuer la remarque suivante concernant le déroulement des opérations de communications. En effet, on peut constater sur les deux profils que les communications avec le processeur maître sont bloquées pendant que celui-ci exécute l'accumulation

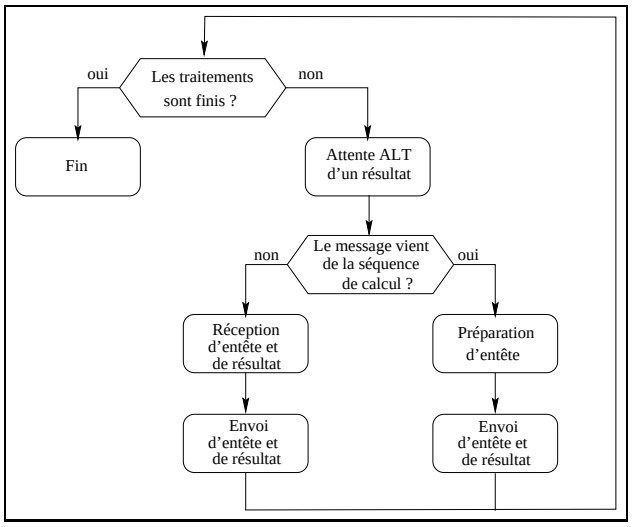


Figure 4.22: Organigramme de fonctionnement de la séquence E→M

d'un résultat (fonction A). Ce blocage est dû au fait qu'il existe une synchronisation de type *Pre1-Suc1* entre la séquence de calcul et la séquence de communication E→M du processeur maître. Cette synchronisation est indispensable car elle permet de garantir que le tampon mémoire associé au résultat en cours d'accumulation n'est pas écrasé par un nouveau résultat provenant d'un processeur esclave. Une telle situation se retrouve par exemple sur la figure 4.23 au niveau du résultat produit par le premier opérateur *compute* placé sur le processeur E_1.

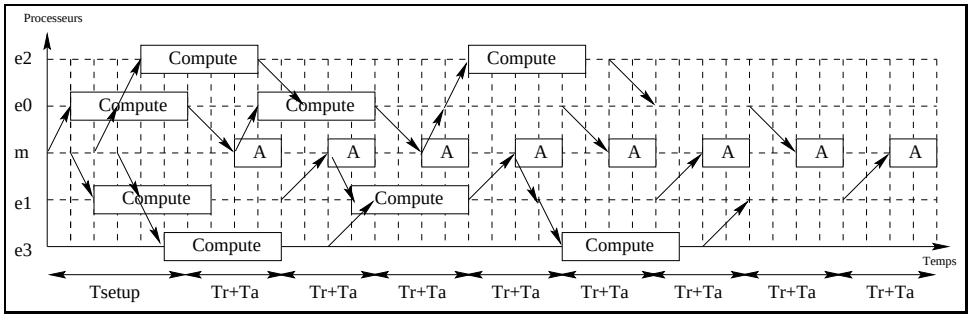


Figure 4.23: Profil typique d'exécution du squelette DF : Cas 1

4.4.2.4.1 Cas 1

Dans le premier cas, la latence du squelette DF peut s'exprimer avec la formule suivante :

$$\begin{cases} T_{DF} = T_{Setup} + N * (T_{Recv} + T_{Acc}) \\ T_{Setup} = T_{Send} + T_{Compute} \end{cases} \quad (4.3)$$

avec

- N est le nombre de données à traiter par l'ensemble des processeurs esclaves. Dans le cas des figures 4.23 et 4.24, N a pour valeur 8,
- $T_{Compute}$ représente le temps d'exécution de la fonction de traitement d'une donnée par un processeur esclave. Nous faisons en fait l'hypothèse que cette valeur est un temps moyen égal à $\frac{T_{Cseq}}{N}$ où T_{Cseq} est le temps de calcul de la fonction *compute* opérant sur l'ensemble des N données.
- T_{Acc} est le temps pris par la fonction d'accumulation d'un résultat,
- T_{Send} (respectivement T_{Recv}) est le temps de transfert d'une donnée (respectivement d'un résultat) sur un lien point à point.

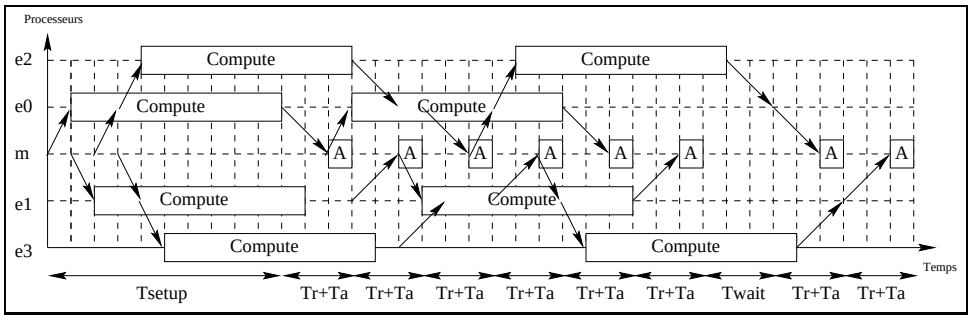


Figure 4.24: Profil typique d'exécution du squelette DF : Cas 2

4.4.2.4.2 Cas 2

Dans le deuxième cas, le modèle analytique repose sur une équation similaire prenant en compte les temps d'attente du maître :

$$\begin{cases} T_{DF} = T_{Setup} + N * (T_{Recv} + T_{Acc}) + T_{Wait} \\ T_{Setup} = T_{Send} + T_{Compute} \\ T_{Wait} = (\frac{N}{n} - 1) * (T_{Compute} + (T_{Send} + T_{Recv}) * d - n * (T_{Recv} + T_{Acc})) \end{cases} \quad (4.4)$$

avec

- n est le nombre de processeurs esclaves,
- d est le diamètre du réseau tel que nous l'avons défini au paragraphe 4.4.1.2.

L'estimation du temps d'attente T_{Wait} s'effectue de la manière suivante. En supposant que les temps de traitement des processeurs esclaves soient uniformes sur l'ensemble des données, chaque esclave se voit confier la charge de traiter $\frac{N}{n}$ données qu'il reçoit périodiquement toutes les n données. Considérons le processeur situé à la distance d du maître, celui-ci réceptionne donc une nouvelle donnée tous les $T_{SCR} = T_{Send} * d + T_{Compute} + T_{Recv} * d$. Pendant ce temps là, le processeur maître doit réceptionner et accumuler n résultats partiels ce qui se traduit temporellement par l'équation $T_{RA} = n * T_{Recv} + n * T_{Acc}$.

Pour un ensemble de n données, l'attente du maître correspond donc à la différence de ces deux temps ce qui donne :

$$T_{Wait} = T_{Compute} + (T_{Send} + T_{Recv}) * d - (T_{Recv} + T_{Acc}) * n \tag{4.5}$$

Ce temps d'attente doit être multiplié par un facteur de valeur $\frac{N-n}{n}$ correspondant au nombre de distributions de paquets de n données par le maître à partir du moment où le système est en régime permanent (c'est-à-dire après la phase initiale de distribution d'une donnée à l'ensemble des esclaves).

Le passage de la première relation donnant la latence du squelette DF à la seconde équation est régi par la validité de la formule suivante :

$$T_{Compute} + (T_{Send} + T_{Recv}) * d \geq n * (T_{Recv} + T_{Acc}) \tag{4.6}$$

4.4.3 Cas du squelette TF

4.4.3.1 Implantation

Le squelette TF est une généralisation du squelette DF et utilise de fait une stratégie d'implantation similaire à celle employée par le squelette DF. L'implantation des séquences de communication est en tout point similaire à celles que nous venons de décrire. Il faut seulement noter que les séquences de communication de type E→M doivent gérer indifféremment des transferts de paquets de données et de résultats. Ceci est possible du fait que la communication est toujours précédée par le message d'entête indiquant le type de communication à mettre en œuvre.

La différence majeure provient du fait que les séquences de calcul aussi bien du processeur maître que des processeurs esclaves ont un comportement opérationnel plus complexe (cf. paragraphe 2.4.4).

La séquence de calcul du processeur maître a pour tache principale la gestion d'un ensemble de données dont le nombre peut varier dynamiquement au cours d'une même itération. En effet, l'application de la fonction de traitement d'une donnée sur un processeur esclave est conditionnée par la validité du prédicat. En cas de succès, le processeur maître reçoit en retour le résultat du traitement et applique la fonction d'accumulation. Dans le cas contraire, la donnée est renvoyée et le maître effectue une partition de celle-ci, générant ainsi de nouvelles données devant être traitées ultérieurement. Ce comportement est représenté sur la figure 4.25.

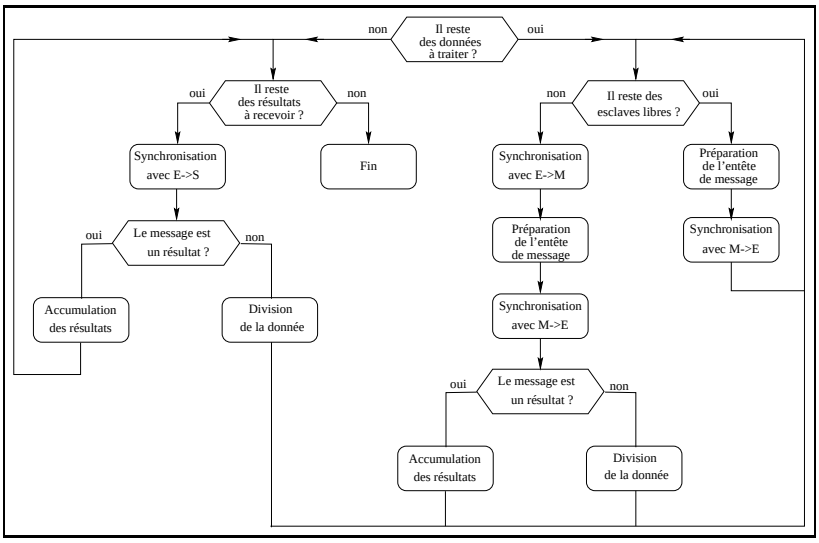


Figure 4.25: Organigramme de fonctionnement de la séquence de calcul du maître

Dans le cas des processeurs esclaves, la donnée reçue va préliminairement à tout traitement être testée par le biais du prédicat. En cas de succès, la donnée est traitée et le résultat correspondant est ensuite renvoyé au processeur maître. (cf. figure 4.26).

4.4.3.2 Modèle de performances

Etant donné le caractère dynamique du nombre de données à traiter, il est ici difficile de tracer et d'interpréter un profil typique d'exécution du squelette TF. La figure 4.27 illustre une situation particulière dans laquelle l'ensemble

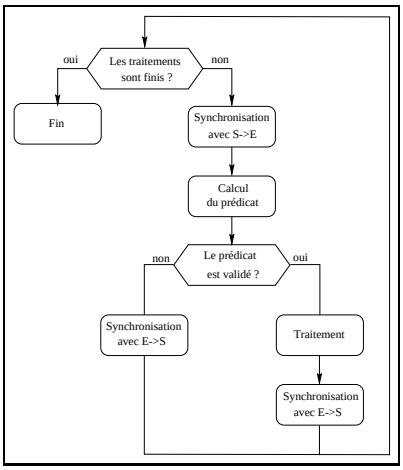


Figure 4.26: Organigramme de fonctionnement de la séquence de calcul

des données distribuées ne valide pas le prédicat H. L'application du prédicat H au niveau 0 est noté H_0 sur le profil. Pour chacune des données, est alors appliquée la fonction de division D qui crée k nouvelles données (dans le cas illustré sur la figure, on a $k = 2$). Les données ainsi générées passent ensuite avec succès le test du prédicat H au niveau 1 — noté H_1 — et sont ensuite traitées par la fonction *Solve* sur chacun des processeurs esclaves. A la fin de ces traitements, les résultats sont transférés en direction du processeur maître qui les accumule (fonction C).

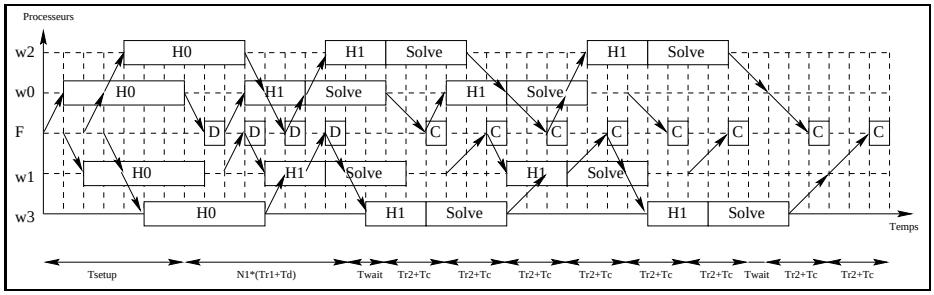


Figure 4.27: Profil d'exécution du squelette TF

Il semble donc difficile de donner un modèle analytique de performances associé au squelette TF puisqu'il apparaît impossible de déterminer le nombre de données générées récursivement. Cependant, il est possible d'estimer le temps maximum d'exécution pour chacun des niveaux de récursivité en supposant que sur ce niveau toutes les données vont être divisées. Cette estimation a pour unique objectif de donner une borne maximale de la latence du squelette TF.

En arrêtant la récursivité au niveau l, le temps d'exécution du squelette TF peut être évalué par la relation suivante :

$$\begin{cases} T^l_{TF} = T_{Setup} + \sum_{i=0}^{l}(N_i * (T^i_{Receive} + T^i_{Farmer}) + T^i_{Wait}) \\ T_{Setup} = T^0_{Send} + T^0_H \\ T^i_{Wait} = (\frac{N_i}{n} - 1) * (T^i_{Worker} + (T^i_{Send} + T^i_{Receive}) * d - n * (T^i_{Receive} + T^i_{Farmer})) \\ T^i_{Farmer} = \begin{cases} T_{Divide} & si\ i\ <\ l \\ T_{Combine} & si\ i = l \end{cases} \\ T^i_{Worker} = \begin{cases} T^i_H & si\ i\ <\ l \\ T^l_H + T_{Solve} & si\ i = l \end{cases} \end{cases} \tag{4.7}$$

avec

- n est le nombre de processeurs esclaves,
- d est le diamètre de l'anneau de processeurs,
- N_i est le nombre de données à traiter au niveau i (dans l'exemple de la figure 4.27, on a $N_0 = 4$ et $N_1 = 8$)
- T^i_{Send} (respectivement $T^i_{Receive}$ représente le temps de transfert d'une donnée (respectivement d'un résultat) au niveau i,
- T^i_{farmer} représente le temps d'exécution de la fonction de division (T_{Divide}) sur les l_1 premiers niveaux et le temps d'exécution de la fonction d'accumulation ($T_{Combine}$) au dernier niveau,
- T^i_{worker} est le temps pris pour calculer le prédicat au niveau i (T^i_H) auquel on ajoute le temps d'exécution de la fonction de traitement (T_{solve}) au dernier niveau,
- T^i_{Wait} est le temps d'attente du processeur maître tel qu'il a été défini au paragraphe 4.4.2.4.

Le modèle analytique de performances du squelette TF ainsi présenté est défini comme une généralisation de celui du squelette DF. Il peut être vu comme un enchaînement séquentiel de squelettes DF, chacun d'entre eux gérant un niveau de division. Chaque squelette DF traite un ensemble de N_i données auxquelles on associe les coûts de traitements T^i_{Worker} pour les processeurs esclaves et T^i_{Farmer} pour le processeur maître.

4.4.4 Cas du squelette ITERMEM

L'implantation du squelette ITERMEM nécessite seulement l'utilisation d'un tampon mémoire dans lequel sont stockés les résultats de l'itération $i-1$ pour être disponibles à l'itération i (cf. paragraphe 2.4.5).

De fait, le modèle analytique de performances associé au squelette ITERMEM peut s'écrire sous la forme suivante :

$$T_{ITERMEM} = T_{Compute} + T_{Copy} \tag{4.8}$$

avec

- $T_{Compute}$ est le temps d'exécution de la fonction de traitement passée en argument du squelette ITERMEM (cette fonction peut être soit une fonction séquentielle de traitement, soit une fonction complexe faisant appel à d'autres squelettes),

- T_{Copy} est le temps pris pour effectuer la copie du résultat dans le tampon mémoire.

4.5 Conclusion

Dans ce chapitre, nous avons présenté l'outil de développement SKiPPER, environnement de programmation parallèle dédié au prototypage rapide d'applications de TI à fortes contraintes temporelles.

Nous avons successivement décrit les différents modules composant l'outil à savoir le compilateur fonctionnel *Dromadaire* effectuant la phase d'expansion des squelettes, SynDEx réalisant l'adéquation Algorithme-Architecture et enfin la génération de code cible tirant parti des spécificités architecturales de la plate-forme Transvision.

Par la suite, pour chacun des squelettes de parallélisation, nous avons étudié précisément l'implantation résultante sur un anneau de processeurs ce qui nous a permis de formaliser leur comportement sous la forme d'un modèle de performance effectuant une estimation prédictive de la latence de chaque squelette.

Le chapitre 3 consacré à la programmation fonctionnelle a mis en avant certains avantages — facilité de programmation, facilité de compréhension et indépendance vis à vis de l'architecture — de notre approche de prototypage rapide d'applications de TI. L'environnement de développement SKiPPER,

présenté dans ce chapitre, permet de répondre à d'autres exigences requises par les modèles efficaces de programmation parallèle :

- **Développement d'outils dédiés** : De la phase de spécification fonctionnelle des applications jusqu'à la génération et la compilation du code cible, SKiPPER gère automatiquement les différentes phases transformant la spécification minimale utilisateur en un code optimisé pour l'architecture cible, libérant ainsi le programmeur de tâches complexes souvent sources d'erreurs. Dès lors, on assiste à une accélération significative du cycle de développement Conception-Implantation-Validation permettant alors au programmeur de tester éventuellement plusieurs solutions algorithmiques. De même, la réduction importante des temps de développement favorise l'évaluation de différentes granularités de traitement dont le rôle est crucial tant au niveau de la facilité d'expression des algorithmes que de l'efficacité des implantations résultantes. Ces aspects confirment l'intérêt d'un outil comme SKiPPER et valident pleinement notre objectif de *prototypage rapide* d'applications,

- **Efficacité des applications** : L'exécutif distribué généré par l'heuristique de placement-ordonnancement de SynDEx constitue un code optimisé prenant en compte les possibilités de recouvrement calcul-communication au sein des processeurs et gérant ce parallélisme interne par le biais d'un mécanisme efficace de synchronisation,

- **Mesures prédictives** : Les modèles de performances associés à chacun des squelettes développés facilitent l'évaluation précise du coût temporel des applications avant implantation réelle sur l'architecture cible,

- **Portabilité** : L'utilisation de SynDEx nous offre certaines opportunités de portabilité des applications puisque l'effort de portage des applications se limite au re-développement des macro-définitions de l'exécutif générique. Cependant, nous avons vu que l'implantation de l'ordonnancement dynamique des communications dans les squelettes DF et TF suppose l'existence de fonctions de transfert de données non déterministes (ALT). Cet aspect peut donc fortement limiter la portabilité de l'approche sur des machines cibles ne possédant pas un tel mécanisme de communication.

En conclusion, l'environnement SKiPPER reposant sur le concept de squelettes fonctionnels de parallélisation répond en règle générale aux six

critères (présentés au chapitre 1) requis par les modèles efficaces de programmation parallèle.

Le chapitre suivant de ce mémoire s'attachera à démontrer l'applicabilité de l'outil SKiPPER en décrivant l'implantation de plusieurs applications de complexité réaliste sur l'architecture Transvision.

Chapter 5

Applications

5.1 Introduction

Les chapitres précédents ont présenté SKiPPPER, un environnement de développement d'applications parallèles dont l'objectif est de permettre le prototypage rapide d'algorithmes de vision artificielle sur une architecture de type MIMD à mémoire distribuée.

L'objectif de ce chapitre est de valider l'approche retenue en décrivant des exemples concrets d'applications développées et implantées en utilisant cet outil.

Dans un premier temps, nous présentons un ensemble d'algorithmes simples dont la spécification ne repose que sur un seul des squelettes développés. Ceci nous permet d'une part de valider l'implantation du squelette sur l'architecture et d'autre part de mesurer les écarts entre les performances mesurées directement lors de l'exécution et celles prédites par les modèles analytiques.

Dans un deuxième temps, nous décrivons un ensemble d'applications plus complexes mettant en œuvre plusieurs squelettes. Le développement de ces applications a pour objectif de montrer l'applicabilité de notre approche à des problèmes de complexité réaliste de TI bas et moyen niveau.

5.2 Un exemple d'algorithme utilisant le squelette SCM

L'utilisation du squelette SCM est illustrée dans ce paragraphe sous la forme d'un algorithme simple calculant l'histogramme des niveaux de gris d'une

image.

Du fait de la régularité des traitements invoqués, cet algorithme de bas niveau est naturellement candidat pour une implantation reposant sur un schéma de parallélisation à parallélisme de données. Le squelette SCM, encapsulant une telle stratégie, apparaît alors comme une solution adéquate pour l'implantation parallèle de l'application de calcul d'histogramme des niveaux de gris.

5.2.1 Spécification fonctionnelle

La spécification fonctionnelle d'une telle application en langage Caml utilisant par exemple huit processeurs et opérant sur des images de taille 512∗512 pixels est donnée ci-dessous :

```
let img   = read_img 512 512 in   (* Lecture d'image         *)
let histo = scm
            8                     (* Nombre de processeurs   *)
            row_block             (* Division de l'image     *)
            seq_histo             (* Calcul d'histogramme    *)
            merge_histo           (* Fusion des histogrammes *)
            img in                (* Image                   *)
display_histo img histo           (* Affichage               *)
```

avec :

- la fonction *read_img* effectuant la lecture d'une image acquise par la caméra,
- la fonction *row_block* décomposant l'image originale en un ensemble de n bandes horizontales,
- la fonction *seq_histo* réalisant le calcul de l'histogramme sur une image,
- la fonction *merge_histo* combinant les histogrammes locaux calculés sur chacune des bandes pour obtenir l'histogramme global de l'image,
- la fonction *display_histo* permettant de visualiser l'histogramme calculé sur l'écran.

La programmation en langage C de ces cinq fonctions représente en définitive la seule tâche de développement que l'utilisateur doit effectuer pour obtenir une implantation parallèle de son algorithme. On donne ci-dessous, à

titre d'exemple, les prototypes C des fonctions séquentielles que l'utilisateur doit fournir.

```
void read_image(int nrow, int ncol, image *out);
void row_block(image in, int n, Tuple(image *, out));
void seq_histo(image in, histo *out);
void merge_histo(histo *out, int n, Tuple(histo, in));
void display_histo(image i, histo h);
```

Ce travail de programmation peut même dans certains cas être moindre puisque certaines fonctions largement utilisées — en particulier les fonctions *read_img* et *row_block* dans le cas de cette application — sont fournies par le système au sein d'une librairie et n'ont pas à être ré-écrites à chaque fois.

Techniquement parlant, les fonctions *row_block* et *merge_histo* doivent pouvoir accepter un nombre variable d'arguments en entrée (respectivement en sortie) afin de s'insérer dans des instances différentes du squelette SCM (Il n'est en effet pas envisageable de requérir du programmeur qu'il écrive autant de versions de ces fonctions que de valeurs possibles du paramètre n du squelette SCM!). C'est le rôle de la macro *Tuple(type,nom)* apparaissant dans le prototype de ces fonctions. Le système fournit par ailleurs un ensemble de macros permettant d'accéder physiquement aux différentes composantes d'un *tuple*[1].

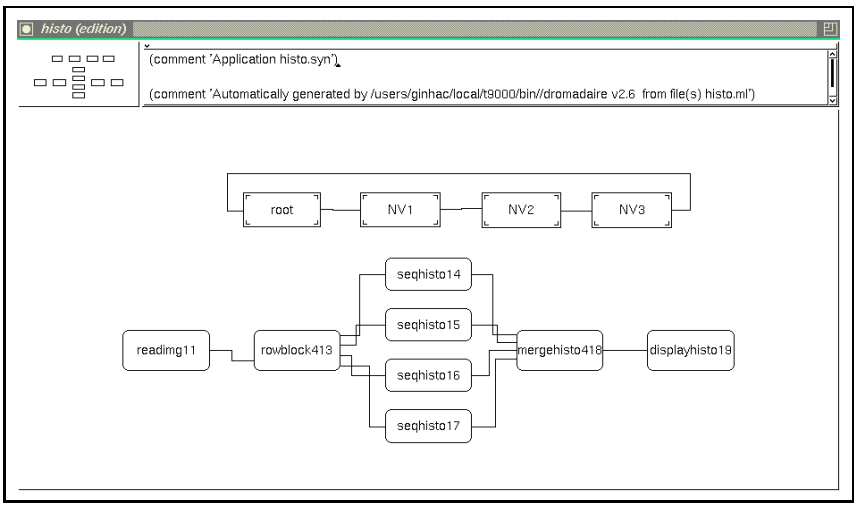


Figure 5.1: Graphes logiciel et matériel visualisés par SynDEX

[1]Pratiquement, les *tuples* sont implantés à l'aide du mécanisme de *va-list* du langage C.

5.2.2 Phase de placement-ordonnancement

Sur la figure 5.1, sont représentés d'une part le graphe logiciel de l'application généré par *Dromadaire* dans le cas où le paramètre n est égal à 4 et d'autre part le graphe matériel de l'architecture sous la forme d'un anneau de processeurs.

L'heuristique de placement-ordonnancement de SynDEx donne les résultats suivants (cf. figure 5.2) :

- exécution en parallèle des différents noeuds de calcul d'histogramme (fonction *seq_histo*) sur les processeurs de l'anneau,
- séquentialisation sur le processeur *root* des noeuds de calcul effectuant la lecture de l'image (fonction *read_img*), la division de l'image en bandes (fonction *row_block*), la sommation des histogrammes locaux (fonction *merge_histo*) et l'affichage de l'histogramme global (fonction *display_histo*).

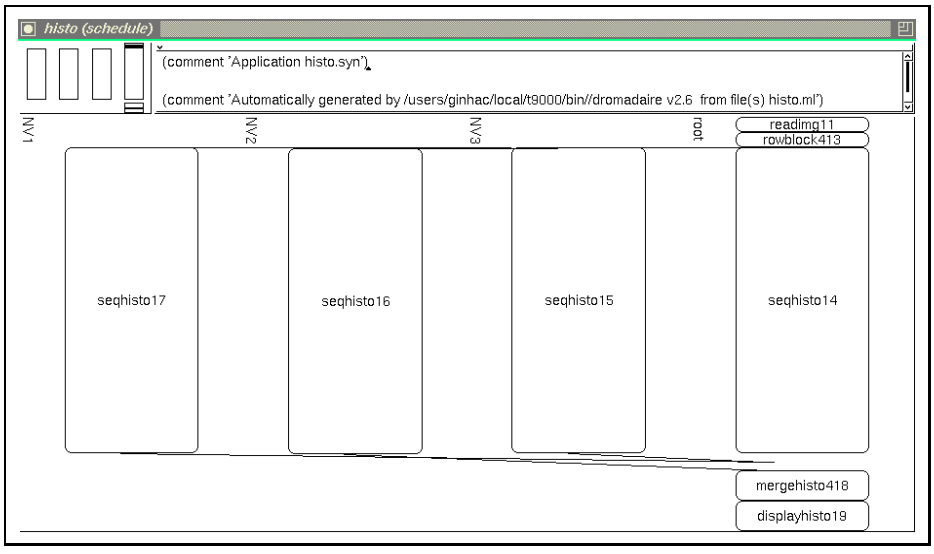


Figure 5.2: Résultat de la phase de placement-ordonnancement de SynDEX

5.2.3 Implantation et résultats

Cette application a été implantée sur la machine Transvision en utilisant différentes configurations architecturales en anneau comportant de 1 à 8 processeurs. Pour chacune d'entre elles, des mesures de performances ont été

réalisées en ayant recours à la version instrumentée du squelette SCM. Elles sont regroupées sur la figure 5.3a montrant l'évolution du temps d'exécution de l'application en fonction du nombre de processeurs.

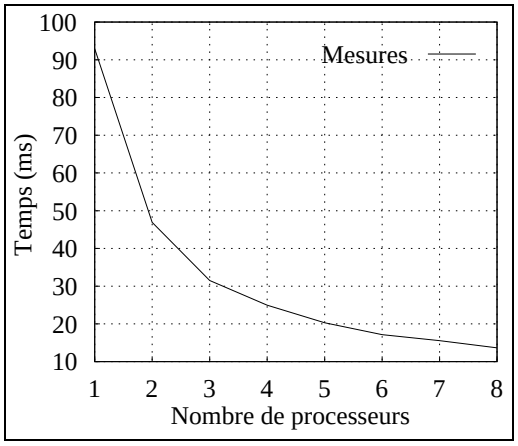


Figure 5.3a

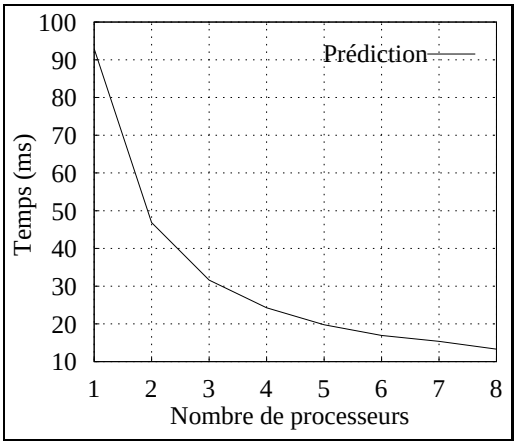


Figure 5.3b

Figure 5.3: Résultats temporels du squelette SCM

De manière similaire, on trouve sur la figure 5.3b la courbe d'évolution des temps d'exécution prédits par le modèle analytique de performance du squelette SCM. Nous pouvons constater une grande similarité entre ces deux tracés et dans le pire des cas, nous ne notons qu'un écart inférieur à 3% entre la mesure effective et celle prédite (cf. figure 5.4).

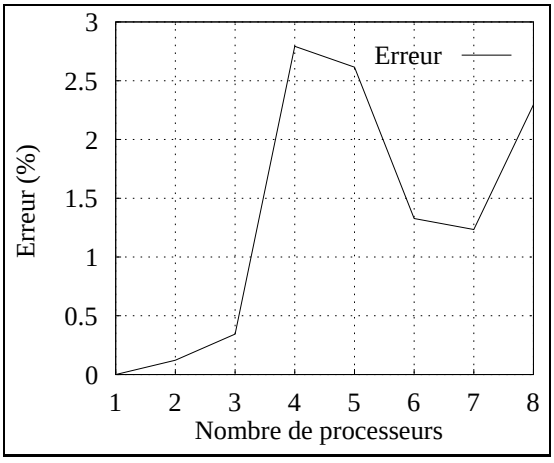


Figure 5.4: Erreur du modèle de performances du squelette SCM

Cette remarquable précision du modèle de performance provient du fait que l'implantation du squelette SCM est basée sur un placement et un or-

donnancement statique des calculs et des communications. De fait, le modèle analytique de performances présenté au paragraphe 4.4.1.2 décrit de manière extrèmement fidèle le comportement temporel du squelette SCM. De même, nous pouvons noter que le modèle de performances donne des résultats tout à fait similaires avec les prévisions temporelles calculées par l'heuristique de placement-ordonnancement de SynDEx.

Ajouté à cela, ce modèle de performances fait l'hypothèse que les temps de traitement associés à la fonction *compute* sont identiques, et ce quelle que soit la partition de l'image. Dans le cas de l'algorithme de calcul de l'histogramme des niveaux de gris, cette condition est vérifiée puisque le temps de traitement de la fonction *seq_histo* dépend uniquement des dimensions de l'image à traiter.

Les écarts minimes entre les deux courbes peuvent être en partie imputées aux facteurs suivants :

- les primitives de synchronisation effectuent des interruptions des séquences de calcul et de communication dont le coût est certes faible mais non négligeable. En particulier, les séquences de calcul placées sur les processeurs effectuant du routage de données sont désactivées momentanément pour programmer les DMA des liens de communication réalisant le transfert des partitions de l'image.

- les primitives de chronométrage, insérées dans la version instrumentée des squelettes, conduisent à un allongement systématique de la latence de l'application.

De tels aspects ne sont pas pris en compte dans la définition du modèle de performances, entraînant inévitablement des écarts entre les mesures réelles et celles prédites par le modèle analytique.

Deuxièmement, la figure 5.5 représente respectivement l'accélération et l'efficacité[2] de l'algorithme de calcul d'histogramme déduites des mesures temporelles réalisées. Le comportement local et régulier de la fonction *seq_histo* garantit un parfait équilibre de charge de chacun des processeurs du réseau. De fait, l'accélération est quasiment linéaire d'où une efficacité voisine de 1. La diminution progressive de l'efficacité est due d'une part aux temps de communication et d'autre part au temps d'exécution des fonctions *row_block* et *merge_histo*. Ceux-ci sont incompressibles et devienne de moins en moins négligeables par rapport au temps d'exécution de la fonction *seq_histo* lorsque l'on augmente le nombre de processeurs.

[2]Les définitions de ces deux grandeurs sont données au paragraphe 1.4.2.3.2.

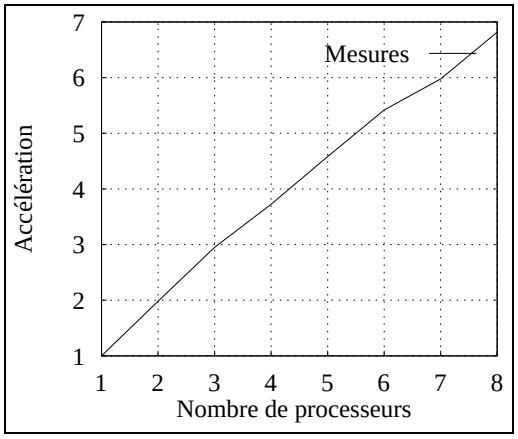


Figure 5.5a

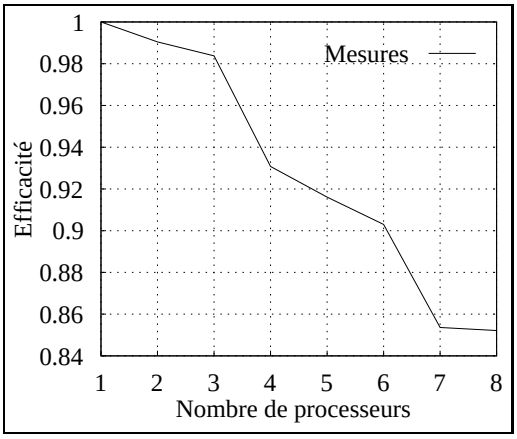


Figure 5.5b

Figure 5.5: Accélération et efficacité du squelette SCM

5.3 Un exemple d'algorithme utilisant le squelette DF

5.3.1 Choix d'une application

L'application choisie pour illustrer le squelette DF a pour objectif de détecter des taches lumineuses dans les images. Elle est notamment à la base de l'algorithme de détection d'amers placés sur des véhicules que nous avons présenté au paragraphe 1.2.1.2 du chapitre 1.

La donnée d'entrée sur laquelle le squelette DF opère est constituée d'une liste de fenêtres d'intérêt de taille variable. Chaque fenêtre contient un nombre indéterminé de taches lumineuses (0, 1 ou plus). Sur chacune de ces fenêtres, l'algorithme de détection de taches opère de la manière suivante. Les pixels dont la valeur est supérieure à un seuil[3] sont déclarés points candidats et sont agrégés aux taches lumineuses existantes. Celles-ci sont modélisées sous la forme d'un descripteur comprenant le nombre de points de la tache, les coordonnées de son centre de gravité ainsi que les coordonnées d'un rectangle englobant la tache. Dès lors qu'un point candidat est connexe au rectangle englobant la tache, il est considéré comme étant inclus dans la tache et le descripteur est alors mis à jour. Dans le cas où le point ne peut être agrégé à une tache existante, cela entraîne la génération d'une nouvelle tache. Le résultat final, après traitement de l'ensemble des fenêtres d'intérêt, est donc composé d'une liste de descripteurs de taches lumineuses (cf. figure 5.6).

[3]Pour des raisons de simplicité, le seuil est fixé de manière statique à la compilation.

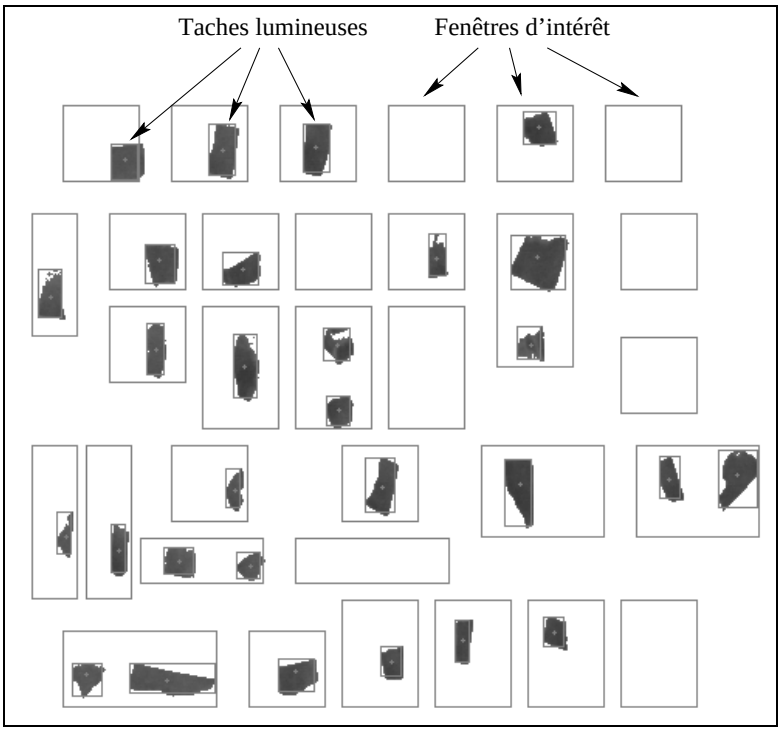


Figure 5.6: Un exemple de détection de taches lumineuses

Par rapport à l'application complète de détection et de suivi de véhicules par amers, l'algorithme présenté ici n'effectue pas itérativement une mise à jour des informations relatives aux fenêtres d'intérêt en ayant recours à une technique de type prédiction-vérification. En effet, le nombre et la taille des imagettes sont ici déterminés et fixés par le programmeur et ne varient pas d'une itération à l'autre.

Le choix de cette application pour valider le squelette DF est aisément justifiable du fait de la possibilité de faire varier d'une part la taille des fenêtres à analyser et d'autre part le nombre de taches présentes au sein de chaque fenêtre. De fait, le temps nécessaire au traitement d'une fenêtre dépend fortement en fonction de ces deux paramètres d'où la nécessité d'avoir un squelette gérant une distribution dynamique des données comme le squelette DF.

5.3.2 Spécification fonctionnelle

La spécification fonctionnelle de l'application (utilisant 8 processeurs) est la suivante :

```
let img     = read_img 512 512 in     (* Lecture de l'image       *)
let windows = extract_windows img in  (* Extraction des imagettes *)
let spots   = df
              8                       (* Nombre de processeurs    *)
              detect_spots            (* Detection de taches      *)
              accum_spots             (* Accumulation de taches   *)
              empty_list              (* Accumulateur initial     *)
              windows in              (* Donnee                   *)
display_spots img spots               (* Affichage                *)
```

avec :

- la fonction *extract_windows* effectuant l'extraction d'une liste de fenêtres d'intérêt à partir d'une image d'entrée acquise par la caméra,
- la fonction *detect_spots* détectant les taches lumineuses à l'intérieur d'une fenêtre selon les principes décrits précédemment et retournant une liste (éventuellement vide) de descripteurs de taches,
- la fonction *accum_spots* accumulant dans une liste les descripteurs des taches détectées dans l'image,
- la fonction *display_spots* affichant les taches détectées (la figure 5.6 représente le résultat de l'affichage).

Les prototypes des fonctions C associées sont donnés ci-dessous :

```
void extract_windows(image i, windowList *ws);
void detect_spots(window w, spotList *sps);
void accum_spots(spotList old, spotList item, spotList *new);
void display_spots(image i, spotList sps)
```

5.3.3 Implantation et résultats

Comme dans le cas de l'application précédente de calcul d'histogramme, l'algorithme que nous venons de décrire a été implanté sur un anneau de processeurs dont le nombre varie entre 1 et 8. De plus, pour chacune de ces configurations, nous avons également fait varier le nombre de fenêtres

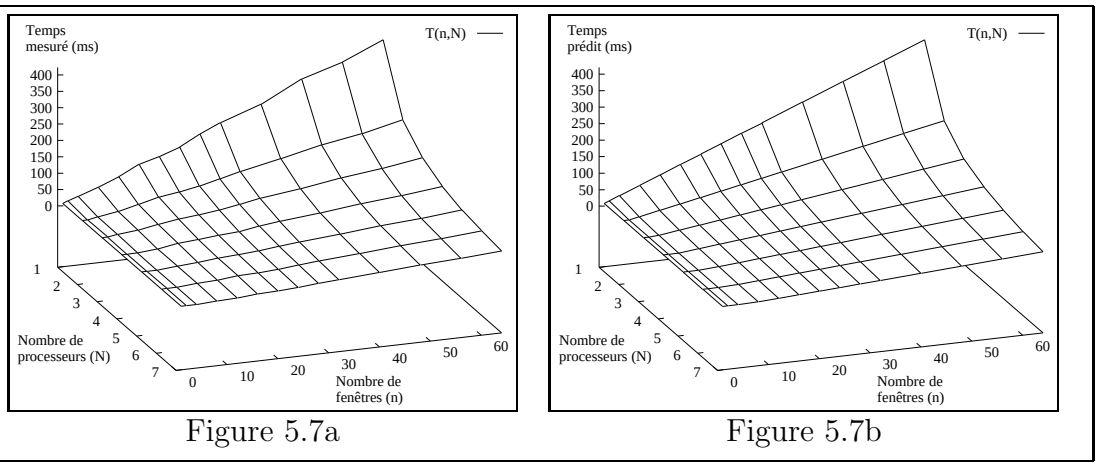


Figure 5.7a Figure 5.7b

Figure 5.7: Résultats temporels du squelette DF

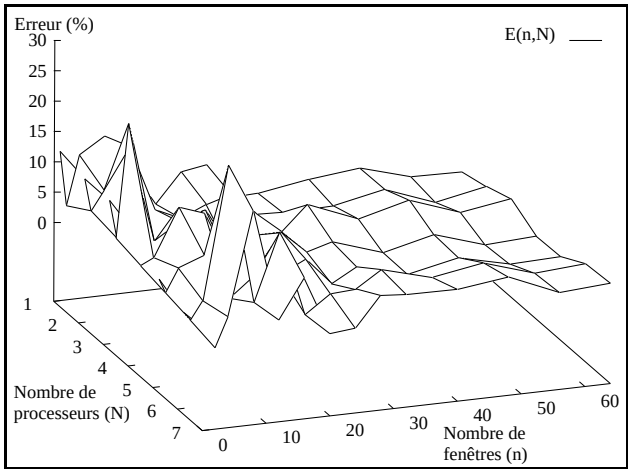


Figure 5.8: Erreur du modèle de performances du squelette DF

d'intérêt à traiter entre 1 et 64. Les résultats temporels mesurés et prédits sont respectivement représentés sur les figures 5.7a et 5.7b.

Malgré les disparités des temps de calcul effectuant la détection des taches lumineuses, nous constatons que les temps d'exécution prédits par le modèle de performances sont relativement proches des mesures réelles (cf. figure 5.8). Les écarts sont inférieurs à 10% en règle générale. Des erreurs plus importantes (maximum de 30%) sont à noter dans un seul cas typique d'exécution à savoir lorsque le nombre de fenêtres d'intérêt est relativement faible par rapport au nombre de processeurs utilisés. Ceci peut être expliqué par le fait que, dans cette situation particulière, chaque processeur ne traite qu'un faible nombre (voire nul) de données et de fait, il est difficile d'obtenir un

équilibre de charge de travail des processeurs esclaves.

Cette constatation se retrouve de manière similaire sur les tracés de l'accélération et de l'efficacité de l'application (respectivement sur les figures 5.9a et 5.9b). En effet, l'accélération ne devient linéaire et environ égale au nombre de processeurs esclaves que dans le cas où le nombre de données à traiter est largement supérieur au nombre de processeurs utilisés. L'efficacité résultante oscille alors autour de 0.9.

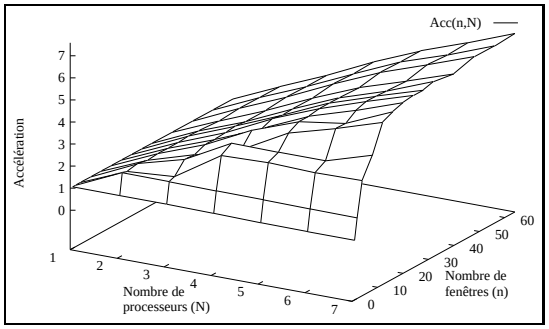


Figure 5.9a

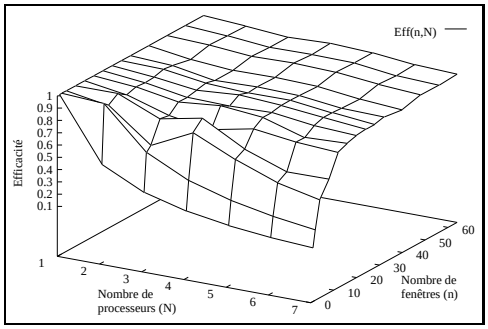


Figure 5.9b

Figure 5.9: Accélération et efficacité du squelette DF

5.4 Un exemple d'algorithme utilisant le squelette TF

5.4.1 Choix d'une application

La dernière application simple que nous présentons ici est un algorithme de type division récursive d'images. Un tel algorithme est classsiquement utilisé commme première étape d'une application de segmentation d'image par division-fusion ("*split and merge*") en utilisant un critère d'homogénéité basé sur les niveaux de gris, le mouvement ou la texture. Cette méthode de segmentation a été introduite par Horowitz et Pavlidis[HP74].

Soit I une image et C un critère d'homogénéité. Segmenter I revient à partitionner I en n sous-ensembles S_i, $i \in [1, n]$ tels que :

$$\begin{array}{lll} 1 & - & I = \bigcup_{i=1}^{n} S_i \\ 2 & - & S_i \bigcap S_j = \varnothing \ \forall i \neq j \\ 3 & - & C(S_i) = vraie \ \forall i \\ 4 & - & C(S_i \bigcup S_j) = faux \ \forall i \neq j \end{array} \tag{5.1}$$

Les trois premières critères correspondent à la phase de division de l'image qui revient à partitionner l'image originale I en un ensemble de n régions disjointes satisfaisant le critère C. Pour illustrer ces aspects, un exemple de découpage d'une image triviale ainsi que sa représentation en arbre quaternaire ("quadtree") sont décrits sur la figure 5.10. La racine de l'arbre représente l'image initiale, les feuilles étant les régions homogènes.

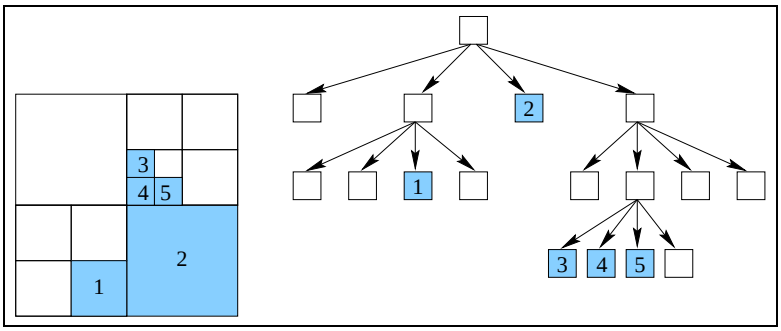


Figure 5.10: Un exemple trivial de division récursive d'image

Le quatrième critère concerne la phase de fusion des régions homogènes adjacentes dont la réunion satisfait également le critère d'homogénéité. Ainsi, les régions 1 à 5 de la figure 5.10 peuvent être agrégées et ne constituer en final qu'une seule région homogène.

Seule la phase de division effectuant une décomposition récursive de l'image initiale est abordée dans ce paragraphe. La figure 5.11 est un exemple d'une division d'une image réelle.

Lorsqu'une région de l'image est déclarée non homogène au sens du critère, il est nécessaire de la partitionner en un ensemble de k nouvelles régions[4] : k nouveaux traitements sont ainsi générés. De fait, le nombre de régions évolue en permanence en cours d'itération sans prévision possible.

Du fait de ses caractéristiques, l'algorithme de division récursive est alors naturellement candidat à une implantation parallèle reposant sur un squelette TF. Un tel squelette nécessite le développement de trois fonctions séquentielles (cf. paragraphe 3.3.2.3) : une fonction h de calcul de prédicat, une fonction *solve* de traitement des régions homogènes et une fonction *divide* de division des régions non homogènes.

Premièrement, le prédicat que nous avons mis en œuvre est basé sur une statistique des niveaux de gris : une région est déclarée homogène si l'écart-type des valeurs des pixels la composant est inférieur à un seuil donné.

[4]La valeur de k constitue un paramètre de l'opérateur de division.

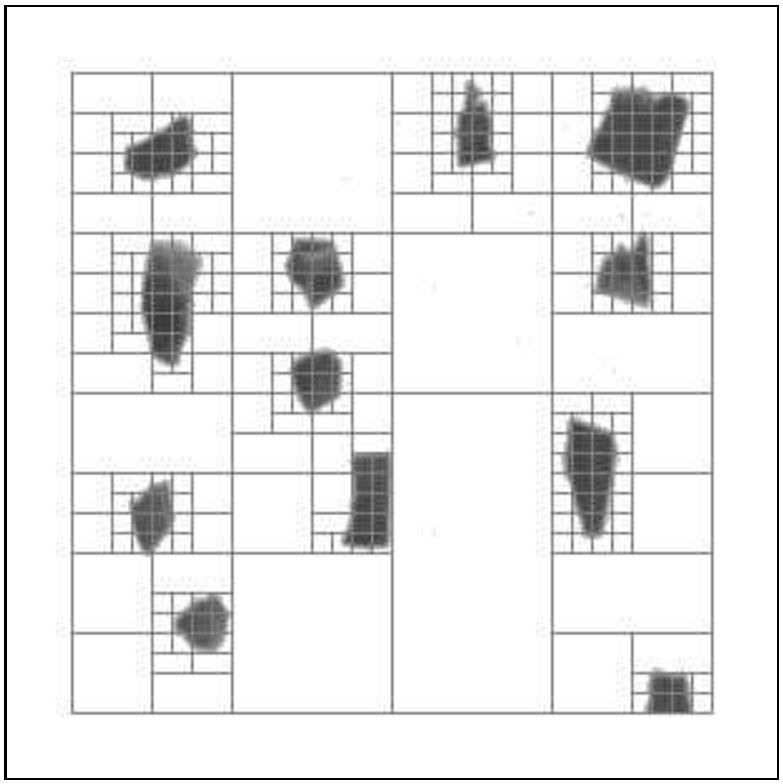


Figure 5.11: Un exemple réel de division récursive d'image

Deuxièmement, dans le cas où le prédicat d'homogénéité est validé, la fonction de traitement retourne comme résultat un descripteur de région homogène composé des informations suivantes : coordonnées de la région, moyenne et écart-type des pixels.

Troisièmement, dans le cas où la région est déclarée non homogène, la fonction de division est appliquée et $k = 2$ nouvelles régions à tester sont générées. Le sens de division (horizontal ou vertical) dépend des dimensions de la région. La division horizontale (respectivement verticale) est appliquée si la hauteur (respectivement la largeur) est la plus grande des deux dimensions.

5.4.2 Spécification fonctionnelle

Avant de décrire la spécification fonctionnelle de l'application de division récursive d'images, il est nécessaire de faire la remarque suivante.

En théorie, le squelette TF ne nécessite pas de division initiale de l'image à traiter. Ainsi, la première donnée envoyée à un des processeurs esclaves est

l'image complète sur laquelle est appliquée la fonction h de prédicat. Cependant, les images traitées sont issues de scènes réelles acquises par une caméra (cf. l'exemple de la figure 5.11) et de fait, elles ne sont que très rarement homogènes au sens du critère. Ceci implique que de nombreuses divisions récursives sont appliquées avant qu'aient lieu les premiers traitements.

Dans le cas de notre application, le temps d'exécution de la fonction h est directement proportionnel à la taille de la région testée. De fait, une telle approche est fortement pénalisante du point de vue temporel puisqu'elle introduit une séquentialité quasi-systématique au démarrage de l'application. L'équilibre de charge des processeurs n'est atteint que lorsque le nombre de données générées est supérieur au nombre de processeurs utilisés.

Il est alors judicieux de réaliser un "pré-découpage" de la donnée initiale en un ensemble de régions avant de la passer au squelette TF. Pour cela, il suffit de modifier légèrement la signature du squelette TF donnée au paragraphe 3.3.2.3 afin qu'il accepte une *liste* d'argument en entrée. Cette nouvelle signature est donnée ci-dessous :

```
val tf :
        int                   (* Nombre d'esclaves                *)
        -> ('a -> bool)       (* Fonction de predicat              *)
        -> ('a -> 'c)         (* Fonction de traitement            *)
        -> ('a -> 'a list)    (* Fonction de partition             *)
        -> ('b -> 'c -> 'b)   (* Fonction d'accumulation           *)
        -> 'b                 (* Valeur initiale de l'accumulateur *)
        -> 'a list            (* Liste de donnees                  *)
        -> 'b                 (* Resultat                          *)
```

La spécification en langage Caml de l'application peut s'écrire de la manière suivante :

```
let img      = read_img 512 512 in    (* Lecture de l'image       *)
let imgs     = init_split img in      (* Division initiale        *)
let hregions = tf
               8                      (* Nombre de processeurs    *)
               is_homogeneous         (* Test de l'homogeneite    *)
               region_desc            (* Calcul du descripteur    *)
               split_image            (* Division de la region    *)
               append_region          (* Accumulation des regions *)
               empty_list             (* Accumulateur initial     *)
               imgs in                (* Liste de regions         *)
display hregions img                  (* Affichage                *)
```

avec :

- la fonction *init_split* effectuant le pré-découpage intial de l'image acquise,
- la fonction *is_homogeneous* déterminant l'homogénéité des régions à partir du calcul de l'écart-type des valeurs des pixels,
- la fonction *region_desc* retournant le descripteur des régions homogènes,
- la fonction *split_image* effectuant la génération de deux nouvelles régions à partir d'une région déclarée non homogène,
- la fonction *append_region* accumulant dans une liste les descripteurs des régions homogènes[5].

On décrit ci-dessous les prototypes C des fonctions séquentielles que l'utilisateur doit fournir :

```
void init_split(image in, regionList *xs);
void is_homogene(region in, bool *out);
void region_desc(region in, region *out);
void split_image(in region, regionList *xs);
void append_region(regionList old, region item, regionList *new);
```

[5]Pour des raisons de simplicité de mise en œuvre, nous utilisons une représentation des descripteurs des régions homogènes sous la forme de liste et non d'arbres.

5.4.3 Implantation et résultats

L'implantation de l'application de division récursive d'images a été réalisée sur la machine Transvision sur une configuration architecturale en anneau (comportant de 1 à 8 processeurs).

Comme précédemment, l'objectif était ici de valider le modèle de performances associé au squelette TF. Or, comme nous l'avons énoncé au paragraphe 4.4.3.2, ce modèle ne donne qu'une borne maximale de la latence puisqu'il suppose que sur chaque niveau de récursivité, toutes les données sont divisées.

De fait, pour chacune des configurations architecturales, nous avons réalisé l'implantation de l'application pour les niveaux successifs de récursivité (jusqu'à des régions de taille $16 * 16$ pixels) en imposant que pour chaque niveau toutes les données soient partitionnées[6].

De plus, une partition initiale de l'image en 8 régions est effectuée préliminairement à l'exécution du squelette TF. Ceci permet alors d'augmenter de manière simple l'efficacité de l'application puisque dès le départ des traitements, on assiste à un équilibre de la charge de travail de l'ensemble des processeurs esclaves. Ce comportement est illustré sur la figure 5.12 qui représente un profil d'exécution de l'application — sur un anneau comportant huit processeurs — visualisé avec l'utilitaire *upshot*[7]. Le processeur maître est représenté par la première ligne (numéro 0), les processeurs esclaves par les lignes suivantes numérotés de 1 à 7.

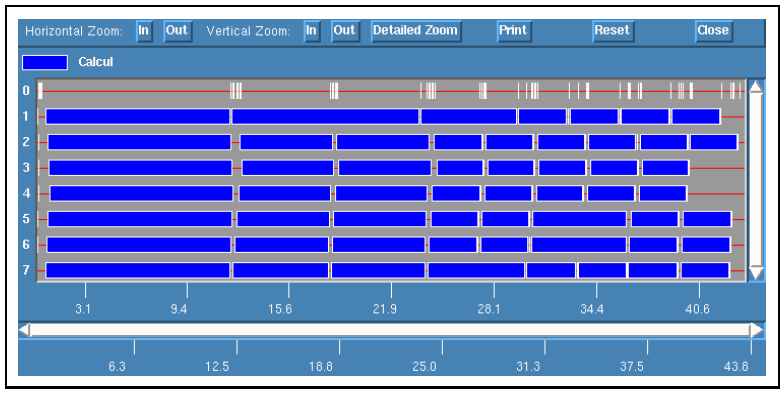


Figure 5.12: Profil d'exécution de l'application de division récursive

[6]Pour cela, il suffit de rendre le prédicat h dépendant uniquement du niveau de récursivité.

[7]Les données temporelles visualisées par cet outil sont générées automatiquement par les versions instrumentées des squelettes.

Sur les figures 5.13a et 5.13b, on trouve respectivement les mesures réelles effectuées lors de l'exécution de l'application et celles prédites à partir du modèle de performances.

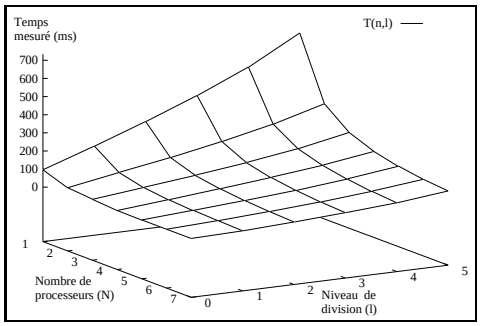


Figure 5.13a

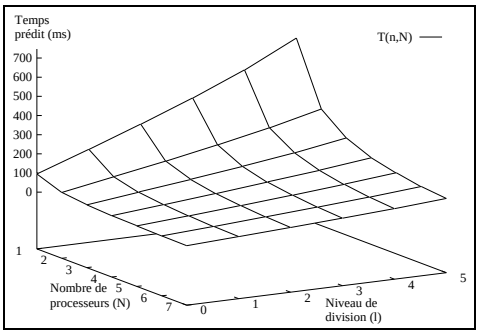


Figure 5.13b

Figure 5.13: Résultats temporels du squelette TF

Lors de la définition du modèle de performances du squelette TF, nous avons émis l'hypothèse suivante : le squelette TF peut être vu comme un enchaînement séquentiel de squelettes DF, chacun gérant un niveau de récursivité. De fait, le modèle associé est défini sous la forme d'une somme de modèles analytiques de performances de squelettes DF.

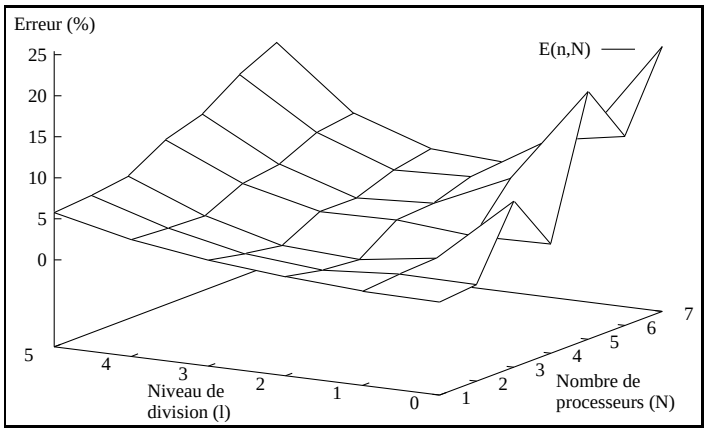


Figure 5.14: Erreur du modèle de performances du squelette TF

La figure 5.14 représente en fonction du nombre de processeurs et du niveau de récursivité atteint le taux d'erreur entre les valeurs prédites par le modèle de performances et les mesures effectives réalisées pendant l'exécution de l'application. A partir de cette courbe, nous pouvons faire les constatations suivantes :

① dans la majorité des cas, le pourcentage d'erreur demeure relativement faible (entre 10% et 15%),

② pour les faibles niveaux de récursivité, le nombre limité de régions[8] ne permet pas d'obtenir un bon équilibre de charge et de fait, entraîne des erreurs de prédiction plus importantes comme dans le cas du squelette DF,

③ à l'opposé, lorsque le nombre de niveaux de récursivité est élevé, des erreurs conséquentes apparaissent du fait que le modèle de performances suppose un enchaînement parfaitement séquentiel de squelettes DF. Or, dans la pratique, il existe un entrelacement des traitements des données au niveau l avec ceux du niveau $l-1$. Cet entrelacement, non pris en compte dans le modèle de performances peut expliquer les erreurs engendrées.

5.5 Etiquetage en composantes connexes

5.5.1 Présentation générale

L'étiquetage en composantes connexes (ECC) est un algorithme de TI moyen niveau largement utilisé en vision artificielle dont l'objectif est d'associer une étiquette unique à chaque composante connexe de l'images. Le terme étiquette fait référence à un nombre entier compris entre la valeur 1 et le nombre total de composantes connexes dans l'image. Ainsi, dans une image étiquetée, deux pixels possèdent la même étiquette si et seulement si ils appartiennent à la même composante connexe, c'est-à-dire au même objet. La figure 5.15 donne un exemple simple d'étiquetage d'une image binaire.

Etiqueter les différentes régions d'une image est une opération fondamentale en TI. En effet, en assignant une étiquette unique à chaque objet, l'ECC permet une différenciation ainsi qu'un dénombrement des objets constituant l'image. Cela peut constituer une première étape vers l'identification des objets de la scène. Des opérations de plus haut niveau peuvent alors prendre place et en particulier des algorithmes de reconnaissance de formes.

Ces caractéristiques font de l'ECC une application de TI largement utilisée et qui de fait, a suscité de nombreux travaux[CSS90][NJR90][AK91][Aln94]. De manière classique, l'implantation de l'ECC nécessite trois phases successives (cf. figure 5.16) :

[8]En particulier, au premier niveau, seulement 8 régions, issues de la division initiale, doivent être traitées.

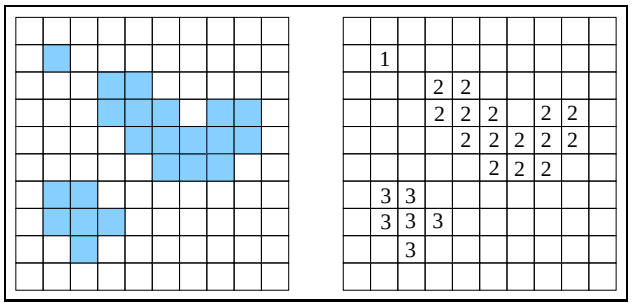


Figure 5.15: Un exemple simple d'ECC

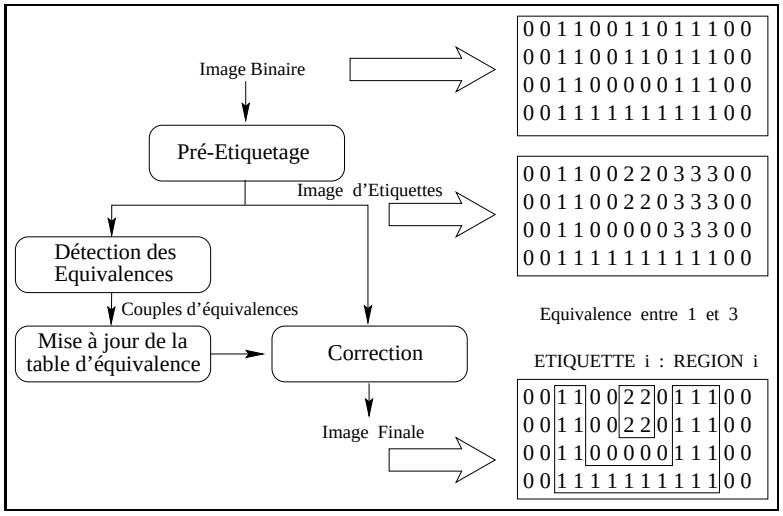


Figure 5.16: Schéma fonctionnel de l'ECC

① une phase de pré-étiquetage calculant une image provisoire des étiquettes en fonction du voisinage immédiat des pixels,

② une phase de détection des conflits d'étiquettes, conduisant à la construction d'une *table d'équivalence*,

③ une phase de correction réalisant la fusion des étiquettes équivalentes.

Une étude bibliographique[Gin95][GSD96] nous a conduits à envisager 3 algorithmes correspondant à ce schéma :

① L' algorithme classique, décrit par exemple dans [Ecc92] et basé sur un pré-étiquetage local par un masque en L inversé,

② Une première variante de cet algorithme[Pra93], pour laquelle une seule étiquette est affectée à chaque segment horizontal, la propagation de

cette étiquette étant assurée par détection des connexités verticales entre segments,

③ Une seconde variante, décrite par Selkow[Sel72], pour laquelle la phase de pré-étiquetage se déroule concurremment avec celle de détection des équivalences et peut donc effectuer certaines pré-corrections [9].

L'algorithme choisi dans le cadre de notre étude est l'algorithme par balayage de segment car le nombre d'étiquettes générées est relativement faible et plus proche du nombre réel de composantes connexes présentes dans l'image.

5.5.2 Spécification fonctionnelle de l'algorithme d'ECC

5.5.2.1 Phase de pré-étiquetage

La phase de pré-étiquetage est par nature "très séquentielle" puisque l'étiquette associée au pixel situé en bas à droite de l'image dépend éventuellement de l'étiquette du coin supérieur gauche de l'image. Cependant, une solution possible à l'implantation parallèle de cet algorithme consiste à diviser l'image acquise en un ensemble de partitions disjointes — par exemple des bandes horizontales — et à affecter chacune d'entre elles à un processeur chargé d'effectuer séquentiellement le pré-étiquetage local de ses données. Par la suite, il est indispensable de mettre en œuvre une stratégie de fusion chargée de gérer les équivalences aux frontières des bandes.

Pour effectuer l'implantation parallèle du pré-étiquetage de l'image, l'utilisation d'un squelette SCM semble alors parfaitement appropriée d'où la spécification CAML associée suivante :

```
let (preetiq,table) = scm
                      nbproc      (* Nombre de partitions    *)
                      row_block   (* Division d'image        *)
                      label       (* Pre-etiquetage          *)
                      merge_table (* Fusion des equivalences *)
                      img         (* Image                   *)
```

avec :

[9] Ces deux variantes ont pour but de diminuer le nombre de conflits introduits par la phase de pré-étiquetage et donc de réduire la taille et le temps de construction de la table d'équivalence.

- *row_block* la fonction de découpage de l'image en *nbproc* partitions (cf. paragraphe 5.2.1),

- la fonction *label* qui effectue séquentiellement d'une part le pré-étiquetage d'une bande de l'image et d'autre part la génération de la table d'équivalence locale à la bande traitée. Pour éviter tout conflit d'étiquette de numéro identique lors de la phase de détection des équivalences inter-bandes, chaque bande se voit attribuer une gamme d'étiquettes différentes : la première étiquette allouée à la bande i a pour valeur $ne_{max} * i$ où ne_{max} est le nombre maximum d'étiquettes attribuées par bandes[10]. En fin de traitement, la fonction *label* retourne comme résultat une donnée structurée comportant d'une part la bande pré-étiquetée et d'autre part la table locale d'équivalences associée à la bande.

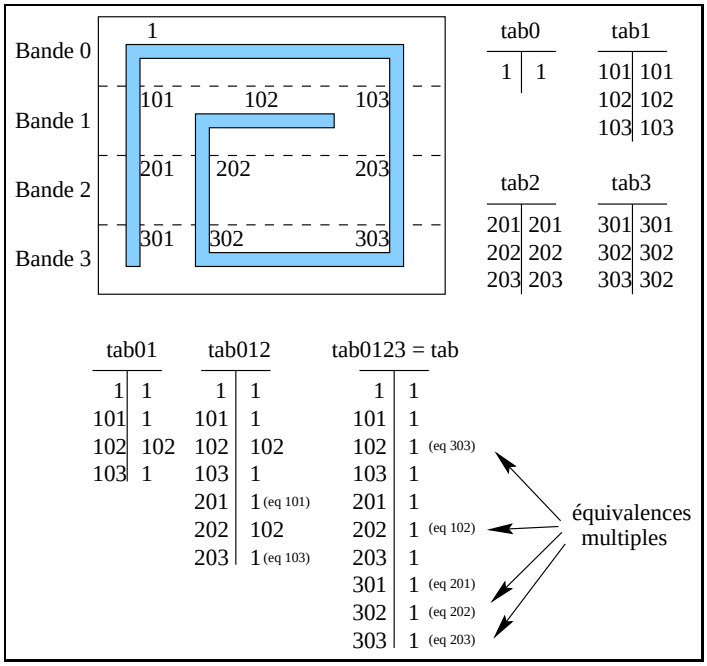


Figure 5.17: Gestion des équivalences d'étiquettes dans le cas d'une spirale

- la fonction *merge_table* qui reconstruit l'image pré-étiquetée par concaténation des bandes traitées et génère, à partir des tables locales et

[10] Ainsi, la bande 0 correspondant au haut de l'image attribue des étiquettes appartenant à l'intervalle $[1, ne_{max}[$ et la dernière bande, correspondant au bas de l'image, travaille sur l'intervalle $[(n-1) * ne_{max}, n * ne_{max}[$.

des lignes frontières des bandes, la table d'équivalences globale. La construction de cette table s'effectue de la manière suivante. Pour chaque frontière inter-bande, on fait un balayage de la dernière (respectivement première) ligne de la bande supérieure (respectivement inférieure) et on détecte les différentes étiquettes possédant une connexité verticale. A chaque détection, on modifie la table en faisant pointer l'étiquette de numéro maximal vers celle de numéro minimale. Dans le cas de la figure 5.17, les trois premières fusions sur les bandes 0,1 et 2 permettent de détecter que les étiquettes 102, 202 et 302 sont équivalentes et égales à l'étiquette 102 ($tab[302] = tab[202] = 102$). On détecte de manière similaire que l'étiquette 303 est équivalente a l'étiquette 203 et donc à l'étiquette 1 ($tab[303] = 1$). Etant donné que l'étiquette 303 est dans le même temps équivalente à l'étiquette 302[11] ($tab[303] = tab[302] = 102$), la stratégie de fusion par pointage vers l'étiquette minimale permet donc de détecter que les étiquettes 102 et 1 sont alors équivalentes ($tab[102] = 1$) ce qui permet donc de déduire que toutes les étiquettes de la spirale sont équivalentes à l'étiquette 1.

5.5.2.2 Phase de correction

Après avoir étudié la première phase de l'application d'ECC, il faut considérer maintenant l'étape de correction des pré-étiquettes qui consiste à remplacer chaque étiquette par sa valeur équivalente minimale issue de la table globale. Dans ce cas encore et compte tenu de la répartition homogène de travail attendue sur l'image, un squelette SCM permet de gérer de manière optimale la parallélisation de cette phase. L'image des pré-étiquettes est divisée en un ensemble de bandes horizontales, chacune d'entre elles étant ensuite traitée par la fonction de correction. A l'issue des différents traitements sur les bandes, l'image finale des étiquettes est générée par simple concaténation des bandes locales.

La spécification Caml associée s'écrit donc ainsi :

```
let etiq = scm
           nbproc              (* Nombre de partitions      *)
           split_etiq          (* Division d'image          *)
           (correct table)     (* Correction des bandes     *)
           concat_bande        (* Concatenation des bandes  *)
           preetiq             (* Image des pre-etiquettes  *)
```

[11] Cette équivalence est détectée lors le la phase de pré-étiquetage de la bande 3.

avec :

- la fonction *split_etiq* réalisant la partition de l'image des étiquettes. Cette fonction est différente de l'opérateur *row_block* qui ne fait que générer des coordonnées transmises aux processeurs effectuant leur traitements sur les données présentes dans leur plan vidéo d'entrée[12]. La fonction *split_etiq*, quant à elle, génère effectivement des partitions d'une matrice de pixels étiquetés qui sont transférées à destination des processeurs de traitement,

- la fonction *correct* effectuant la correction des bandes pré-étiquetées paramétrée[13] par la table globale d'équivalences *table*,

- la fonction *concat_bande* reconstituant l'image complète des étiquettes corrigées.

5.5.2.3 Phase préliminaire de seuillage

L'application d'ECC ainsi spécifiée fait l'hypothèse que les données d'entrée sont en réalité sous la forme d'images binaires. Or, l'image acquise par la caméra est de type niveaux de gris composé de pixels dont la valeur est comprise entre 0 et 255. Il est donc nécessaire de seuiller cette image préliminairement à tout étiquetage des composantes connexes afin d'extraire les objets (représentés par des pixels de valeur 1) du fond de l'image (représenté par des pixels de valeur 0).

Une solution triviale pour effectuer ce seuillage consiste à fixer de manière arbitraire un seuil et a considérer que tout pixel dont la valeur est supérieure à ce seuil appartient aux objets présents dans la scène.

Toutefois, les images acquises sont en général très sensibles aux variations même faibles des conditions d'éclairement. De fait, nous avons préféré utiliser une technique de seuillage adaptatif. Cette stratégie extrait, à chaque itération de l'application, une valeur de seuil à partir du calcul de l'histogramme des niveaux de gris. L'algorithme mis en œuvre est détaillé précisément dans [Tsa85] et permet de faire abstraction des variations de l'intensité lumineuse et ainsi d'assurer une cohérence des résultats

[12]La diffusion des bandes d'images est alors virtuelle puisque l'on ne fait que transmettre la position et la taille des bandes à traiter, ces bandes résidant implicitement dans la mémoire de chaque processeur grâce au mécanisme du noeud vidéo décrit au paragraphe 4.2.2.

[13]On note ici tout l'intérêt de la *notation curryfiée* des fonctions CAML qui permet de définir un argument de squelette par application *partielle* d'une fonction utilisateur.

d'une itération à l'autre. En contrepartie, la méthode employée rallonge systématiquement la latence de l'application d'ECC.

L'implantation du seuillage adaptatif repose sur un squelette SCM et est en tout point similaire avec celle décrite au paragraphe 5.2. La seule différence notable provient du fait que la fonction *threshold_compute* de fusion des histogrammes locaux donne comme résultat une valeur de seuil et non un histogramme global. La spécification Caml s'écrit alors sous la forme suivante :

```
let threshold = scm
                nbproc                (* Nombre de partitions    *)
                row_block             (* Division d'image        *)
                seq_histo             (* Calcul d'histogramme    *)
                threshold_compute     (* Calcul du seuil         *)
                img                   (* Image                   *)
```

5.5.2.4 Spécification fonctionnelle de l'ECC

Toutes les étapes étant désormais décrites, il est alors possible de donner une spécification complète de l'application d'ECC :

```
let img             = read_img 512 512 in    (* Lecture d'image  *)
let threshold       = scm nbproc
                          row_block          (* Division d'image *)
                          seq_histo          (* Histogramme      *)
                          threshold_compute  (* Calcul du seuil  *)
                          img in
let (preetiq,table) = scm nbproc
                          row_block          (* Division d'image *)
                          (label threshold)  (* Pre-etiquetage   *)
                          merge_table        (* Fusion           *)
                          img in
let etiq            = scm nbproc
                          split_etiq         (* Division d'image *)
                          (correct table)    (* Correction       *)
                          concat_bande       (* Concatenation    *)
                          preetiq in
display_etiq img etiq                        (* Affichage        *)
```

Nous pouvons noter que par rapport à la présentation originale de la phase de pré-étiquetage la fonction *label* prend désormais un argument supplémentaire à savoir la valeur du seuil calculé par le premier squelette SCM.

Les prototypes des fonctions C associées à cette spécification sont :

```
void read_image(int nrow, int ncol, image *out);
void row_block(image in, int n, Tuple(image *, out));
void seq_histo(image i, histo *out);
void threshold_compute(int *threshold, int n, Tuple(histo, in));
void label(int threshold, image in, etiq_table *out);
void merge_table(etiq *e_out, table *t_out, int n, Tuple(etiq_table, in));
void split_etiq(etiq in, int n, Tuple(etiq *, out));
void correct(table t_in, etiq e_in, etiq *e_out);
void concat_bande(etiq *out, int n, Tuple(etiq, in));
void display_etiq(image i, etiq e);
```

Sur la figure 5.18, nous donnons le graphe flot de donnée généré par *Dromadaire* et visualisé par SynDEx à partir de cette spécification fonctionnelle dans le cas où le nombre de processeurs est égal à 4.

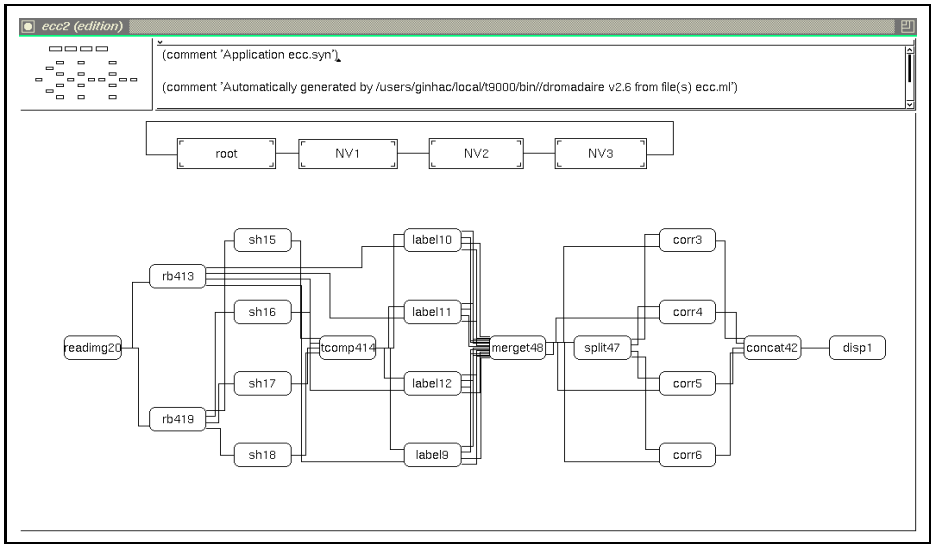


Figure 5.18: Graphe flot de données de l'application d'ECC

5.5.3 Résultats d'implantation

5.5.3.1 Premiers résultats

L'application d'ECC ainsi décrite a été implantée sur la machine Transvision sur des configurations en anneau comportant de 1 à 8 processeurs en ne retenant que les topologies comportant un nombre de processeurs égal à 2^k c'est-à-dire 1, 2, 4 et 8 processeurs.

De plus, afin d'obtenir un éventail de mesures de performances le plus complet possible, nous avons également fait varier la taille des images sur laquelle opère l'algorithme d'ECC entre une taille minimum de $16*16$ pixels et une taille maximum de $512*512$ pixels[14].

Ainsi, en ayant recours aux versions instrumentées des squelettes, nous avons réalisé un ensemble de $4*11$ exécutions chronométrées de l'application d'ECC. La figure 5.19 représente les valeurs de la latence mesurée pour chacune des configurations en fonction d'une part du nombre de processeurs (N) et d'autre part de la taille des images (t).

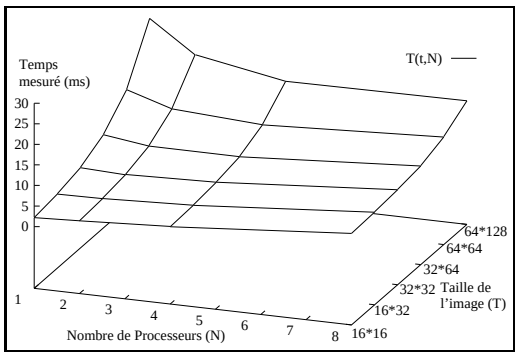


Figure 5.19a

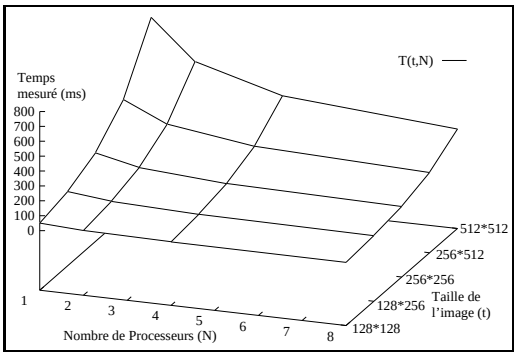


Figure 5.19b

Figure 5.19: Mesure de la latence de l'application d'ECC

Les résultats d'exécution sont scindés en deux tracés : le premier (figure 5.19a) décrit l'évolution de la latence en fonction d'une taille d'image variant entre $16*16$ pixels et $64*128$ pixels, le deuxième (figure 5.19b) représente cette même évolution en fonction d'une taille d'image comprise entre $128*128$ et $512*512$ pixels.

Ces mesures nous permettent d'effectuer la remarque suivante. Il apparaît clairement sur la figure 5.19a que la latence de l'application augmente

[14]L'incrémentation de la taille des images consiste à multiplier par un facteur deux la plus petite des dimensions : $32*32$ pixels, $32*64$ pixels, $64*64$ pixels, etc. ce qui donne un total de 11 tailles d'images différentes.

légèrement avec le nombre de processeurs pour des faibles tailles d'image (en particulier pour des tailles d'images inférieures à $32 * 32$ pixels). En effet, dans ce cas, la parallélisation de l'application sur un anneau comportant 4 ou 8 processeurs implique la mise en œuvre de communications dont la durée ne peut alors être négligée devant les durées d'exécution des fonctions de traitement (qui sont dans ce cas relativement faibles en raison des dimensions réduites des bandes de l'image). De fait, le gain de parallélisation apporté par l'exécution concurrente des fonctions de calcul des différents squelettes SCM est masqué par la mise en place des transferts de données inter-processeurs.

Ce phénomène s'atténue lorsque les dimensions des images à traiter augmentent et en particulier pour des tailles supérieures à $128 * 128$ pixels (cf. figure 5.19b). Dans ce cas, le temps de calcul des fonctions séquentielles distribuées sur les processeurs devient prépondérant par rapport aux durées de communication et on constate une diminution de la latence de l'application lorsque le nombre de processeurs croît.

Ce constat se retrouve sur les courbes représentant respectivement l'accélération (cf. figure 5.20a) et l'efficacité (cf. figure 5.20b).

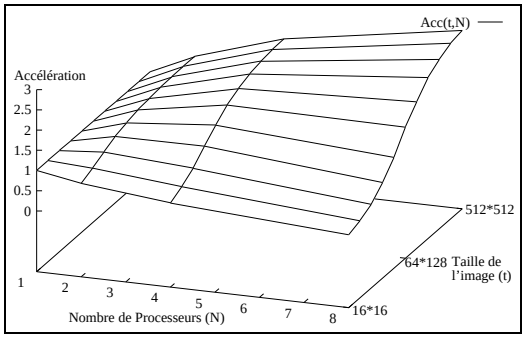


Figure 5.20a

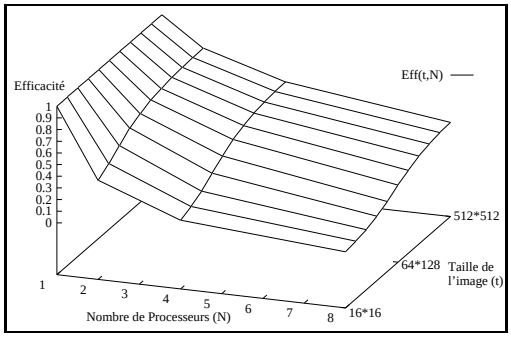


Figure 5.20b

Figure 5.20: Accélération et efficacité de l'application d'ECC

Dans le meilleur des cas[15], nous n'observons toutefois qu'une accélération maximale légèrement inférieure à 3. Ceci peut s'expliquer par le fait que les performances globales de l'application sont ici limitées par le volume des communications mis en œuvre au sein de l'implantation, et en particulier dans l'enchaînement séquentiel des trois squelettes SCM. Un exemple typique est la redistribution complète de l'image des pré-étiquettes entre les deuxième et troisième squelettes SCM (effectuant respectivement la phase de pré-étiquetage et la phase de correction).

[15]Implantation réalisée sur des images de taille $512 * 512$ sur un anneau de 8 processeurs.

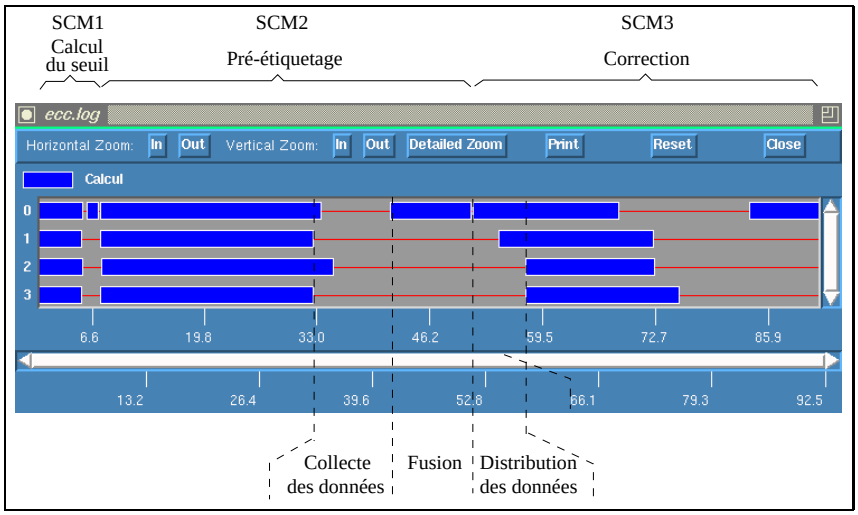


Figure 5.21: Profil d'exécution de l'application d'ECC

Ce phénomène apparaît sur la figure 5.21 qui donne un profil d'exécution de l'application d'ECC implantée sur un anneau à 4 processeurs et opérant sur des images de taille $256 * 256$ pixels. Ce profil met clairement en évidence la durée de collecte des bandes de pré-étiquettes et des tables d'équivalences locales (8 ms) ainsi que le coût de redistribution de ces mêmes bandes (6.3 ms) après l'opération de fusion (9.6 ms). L'exécution séquentielle des ces opérations sur le processeur *0* possède donc comme durée d'exécution $t = 8 + 6.3 + 9.6 = 23.9$ ms ce qui représente environ 25% du temps total de l'application. Cela entraîne donc un déséquilibre de charge important des processeurs dont la conséquence directe est une chute de l'efficacité de l'application.

5.5.3.2 Une optimisation de l'application d'ECC

Compte tenu des observations faites au paragraphe précédent, une optimisation des performances de l'application d'ECC passe inévitablement par la limitation du volume de communications entre les opérations de calcul s'exécutant sur des processeurs différents.

La fonction de calcul *merge_table* terminant le deuxième squelette SCM effectue en réalité deux opérations distinctes (cf. paragraphe 5.5.2.1) :

① la génération de la table d'équivalence globale à partir d'une part des tables locales et d'autre part des lignes frontières des bandes,

② la reconstruction de l'image des pré-étiquettes par concaténation des

bandes traitées.

Ces opérations étant indépendantes, la fonction peut alors être décomposée en deux fonctions distinctes : la première effectuant la fusion des tables d'équivalence (et nécessitant qu'un très faible volume de communication à savoir le transfert des tables et des frontières des bandes) et la deuxième réalisant la concaténation des bandes de pré-étiquettes (et nécessitant quant à elle un volume significatif de communication entre les différents processeurs de traitement et le processeur chargé de la concaténation) en vue de reconstruire l'image globale pré-étiquetée. Par la suite, le troisième squelette SCM effectue un partitionnement de cette même image, chaque bande générée étant redistribuée à un processeur réalisant la phase de correction.

Ces deux phases de collecte et de distribution des bandes d'images ne sont pas nécessaires si on fait l'hypothèse que chaque processeur de traitement effectue les phases de pré-étiquetage et de correction sur la même bande de données. De fait, une optimisation possible consiste à n'utiliser qu'un seul squelette particulier basé sur un schéma à parallélisme de données de type SCM mais effectuant entre deux opérations de calcul (pré-étiquetage et correction des bandes d'étiquettes) une fusion interne des tables d'équivalence. La mise en œuvre d'un tel schéma a été réalisée ici manuellement à partir du graphe flot de données de l'application et a conduit à la génération du graphe représenté sur la figure 5.22.

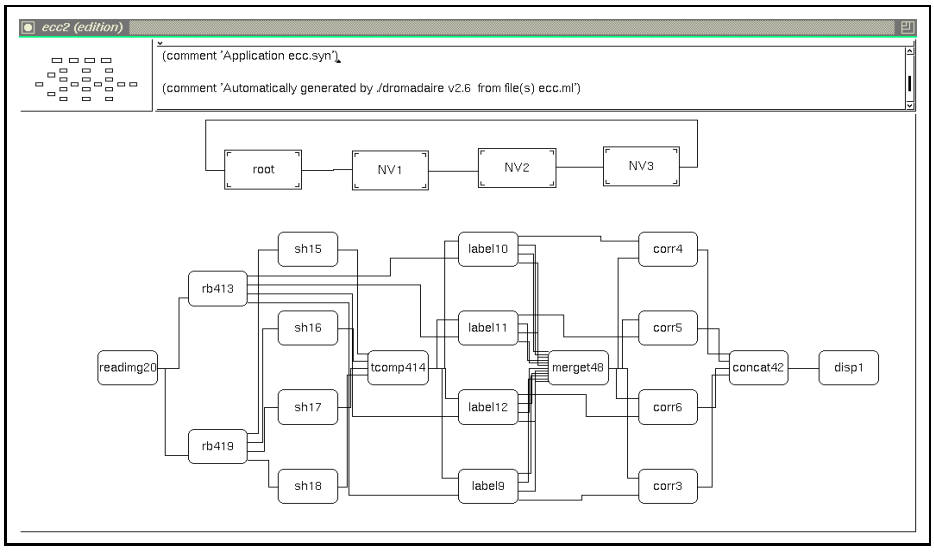


Figure 5.22: Graphe flot de données de l'application optimisée d'ECC

L'implantation d'une telle optimisation a été effectuée pour l'ensemble

des configurations précédentes (nombre de processeurs variant entre 1 et 8, taille d'image comprise entre $16 * 16$ et $512 * 512$ pixels).

La figure 5.23 représente les gains de durée d'exécution en fonction du nombre de processeurs et de la taille des images. De manière générale, on constate que les gains les plus importants (25% à 35%) sont obtenus pour des tailles d'images supérieure à $128 * 128$ pixels et une topologie comprenant au minimum 4 processeurs.

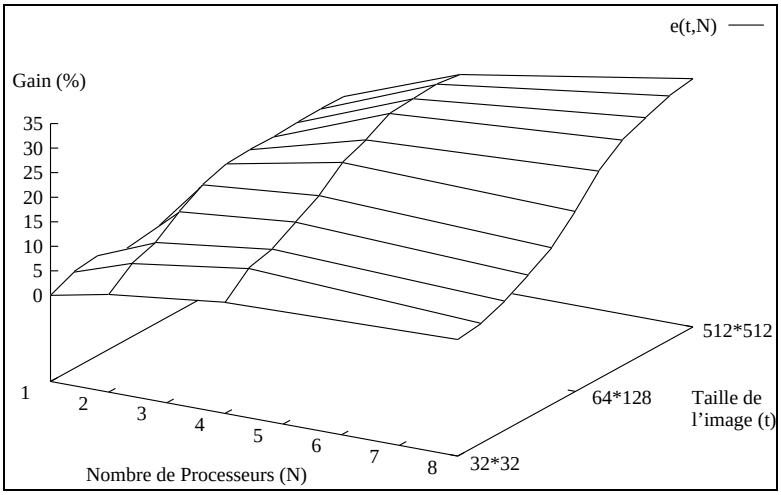


Figure 5.23: Gain d'exécution de l'implantation optimisée

Ceci est aisément justifiable dans le sens où dans le cas de l'implantation parallèle non optimisée, de telles configurations impliquent d'une part d'importants volumes de communications inter-processeurs du fait de la taille des images et d'autre part la mise en place de mécanismes de routage de ces mêmes données pour les processeurs non directement connectés. Ces transferts n'ayant plus lieu dans l'implantation optimisée, on assiste alors à une augmentation conséquente de l'efficacité de l'application. De manière similaire à la figure 5.21, la figure 5.24 donne le profil d'exécution de l'application d'ECC traitant des images de taille $256 * 256$ implantée sur un anneau de 4 processeurs. La phase de fusion des tables d'équivalences à la fin du deuxième squelette SCM a pour durée d'exécution environ 3 ms, ce qui ne représente plus que 4% de la latence de l'application (au lieu de 25% précédemment).

Cependant, nous pouvons noter que l'efficacité des implantations optimisées est tout de même relativement faible (accélération maximale de 4.2 pour une image de taille $512 * 512$ pixels traitée par 8 processeurs). Ces performances médiocres sont dues essentiellement à la phase finale de collecte des bandes d'étiquettes corrigées suivi de leur concaténation par la fonc-

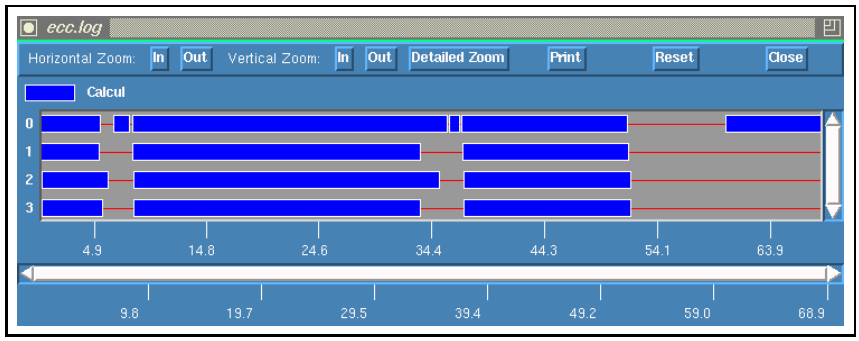


Figure 5.24: Profil d'exécution de l'application optimisée d'ECC

tion *concat_bande* afin de reconstruire l'image finale. De manière similaire à la phase intermédiaire de fusion des bandes pré-étiquetées qui a conduit à l'implantation optimisé présentée ici, la phase de reconstruction d'image finale de durée 8 ms est précédée d'une phase de transfert des bandes corrigées de l'ensemble des processeurs à destination du processeur *0* de durée d'exécution 8 ms (cf. figure 5.24). Ainsi, l'image globale des étiquettes n'est disponible qu'au bout de $t = 16$ ms après la fin de la phase de correction, ce qui représente environ 20% de la latence de l'application.

5.5.4 Conclusion

L'ECC est la première application de complexité réaliste implantée automatiquement par l'outil de parallélisation SKiPPER. Elle valide ainsi **l'applicabilité** d'un tel outil face à nos problématiques de prototypage rapide d'application de TI. La mise en œuvre de l'implantation parallèle étant entièrement automatisée, l'utilisateur peut alors se concentrer sur le développement et l'optimisation des fonctions séquentielles spécifiques à l'algorithme.

La description de l'application d'ECC repose sur **l'enchaînement** séquentiel de trois squelettes SCM effectuant respectivement l'évaluation d'une valeur de seuil, le calcul de l'image pré-étiquetée et la correction des conflits d'étiquettes. Ce type d'enchaînement est relativement classique en TI puisqu'une application de complexité réaliste peut en général se décomposer en un ensemble modules de complexité variable allant du bas vers le haut niveau. De fait, l'implantation de l'application d'ECC sur la machine Transvision a permis de valider l'enchaînement de squelettes.

Cependant, nous avons mis en évidence que la réorganisation des données entre chaque squelette SCM peut conduire à des performances médiocres

dans le cas où le volume de données manipulées est relativement important. Face à ce problème, nous avons proposé une solution visant à diminuer ce volume. L'implantation résultante a par contre été entièrement réalisée manuellement conduisant à des performances accrues. L'automatisation de telles optimisations est envisageable dans un futur relativement proche. Cela passe par la formalisation de **règles de transformation** inter-squelettes et leur intégration dans l'outil SKiPPER. De telles règles permettraient alors de transformer une spécification d'application en une autre équivalente dont l'implantation sur la machine cible est plus performante.

5.6 Suivi de signalisation horizontale autoroutière

5.6.1 Présentation générale

La deuxième application de complexité réaliste présentée dans ce mémoire est extraite du vaste domaine d'applications liées au concept de véhicule intelligent. L'algorithme développé dans le cadre de cette étude a pour objectif de détecter la position d'un véhicule[16] par rapport à la route sur laquelle il évolue[Cha91b].

Une telle application peut constituer une première étape de projets plus ambitieux de conduite automatique ou d'aide à la conduite destinés à pallier les éventuelles erreurs de pilotage du conducteur. Elle peut par exemple être utilisée conjointement avec l'application de détection et suivi de véhicules par amers lumineux (présentée au paragraphe 1.2.1.2). Dans un tel contexte, le positionnement du véhicule par rapport à la route permet alors de restreindre la zone de recherche des véhicules potentiels à la zone de l'image représentant la route.

La mise en œuvre de l'implantation parallèle de cette application en ayant recours à l'outil de prototypage SKiPPER s'inscrit dans notre objectif de développement d'un environnement de programmation destiné à un vaste public. Ainsi, le développement de cette application a été confié à un programmeur non spécialiste des architectures parallèles.

Dans le contexte autoroutier, la route est constituée de deux[17] voies parallèles de largeur constante dont les bords sont matérialisés par des bandes blanches de signalisation horizontale. L'algorithme mis en œuvre a pour ob-

[16] A bord duquel est embarquée une caméra effectuant l'acquisition des données images.
[17] Le cas des autoroutes à trois voies n'est pas envisagé ici.

jectif de détecter ces bandes blanches et d'en déduire la position du véhicule sur la chaussée. Pour cela, on utilise un modèle des deux voies de l'autoroute réactualisé après chaque itération de l'application à partir de résultats de détection effectuées dans l'image. De plus, afin d'obtenir ces résultats à une cadence la plus élevée possible, le choix a été fait d'opérer non pas sur l'image complète mais sur des zones de faible taille (fenêtre d'intérêt) dont les caractéristiques (position, taille et nombre) sont déduites du modèle de la route.

De fait, il apparaît clairement que la stratégie employée au sein de cette application est de type prédiction-vérification (cf. paragraphe 1.2.1.2) et peut être décomposée en trois étapes :

- la première phase a pour objectif de déterminer les caractéristiques des fenêtres d'analyse sur lesquelles va porter la détection. Cette opération est réalisée à partir des informations données par le modèle de la route,
- la seconde phase effectue la détection des bandes blanches dans chaque fenêtre d'analyse,
- enfin, la troisième phase consiste à mettre à jour les paramètres du modèle de la route en fonction des résultats de détection afin que ce modèle soit le plus conforme à la réalité.

L'enchaînement de ces trois étapes est décrit sur la figure 5.25.

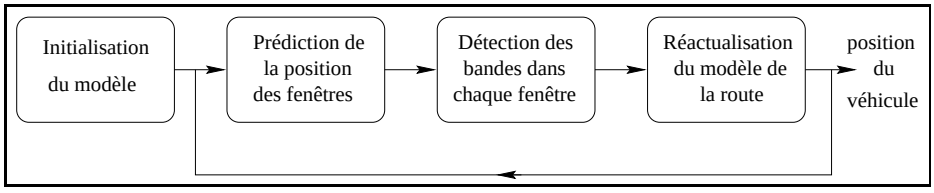


Figure 5.25: Organigramme de l'application de suivi de bandes blanches

Nous pouvons également noter la nécessité de mettre en œuvre un module supplémentaire destiné à initialiser le modèle de la route lors de la première itération de l'application. Cette étape est indispensable du fait que les paramètres du modèle dans ce cas là ne peuvent être extraits des résultats précédents.

5.6.2 Description des trois modules

5.6.2.1 Phase de prédiction

Comme indiqué précédemment, la phase de prédiction est réalisée à partir des informations tirées du modèle de la route. Le modèle retenu dans [Cha91b] est basé sur un ensemble de cinq paramètres réactualisés à chaque itération (cf. figure 5.26) :

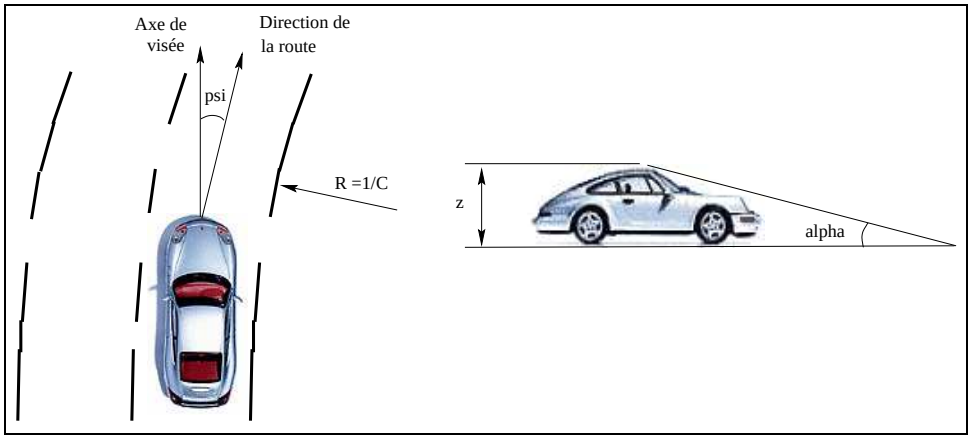


Figure 5.26: Paramètres du modèle de la route

- C : courbure de la route considérée comme localement constante,
- x_0 : position latérale du véhicule par rapport à la bande de la route la plus à droite,
- ψ : angle de déviation horizontale de la caméra,
- α : angle d'inclinaison de la caméra,
- z : hauteur de la caméra.

Pour des raisons de simplification de mise en œuvre, l'algorithme implanté par l'outil de développement SKiPPER n'utilise en réalité que les trois premiers paramètres. L'angle d'inclinaison et la hauteur de la caméra sont supposés constants et fixés respectivement à des valeurs initiales α_0 et z_0.

La position estimée de la route par ce modèle permet de définir un ensemble de fenêtres d'intérêt répondant aux caractéristiques suivantes :

- le centre des fenêtres d'intérêt est placé sur la position estimée des bandes blanches,

- le nombre des fenêtres est au maximum égal à 15 (réparties uniformément sur les trois bandes de signalisation). Dans la pratique et suivant les configurations de la route, certaines de ces fenêtres sont positionnées en dehors de l'image acquise et de fait ne sont pas distribuées et traitées,

- la taille des fenêtres est fonction de leur position dans l'image. En effet, les zones d'analyse placées dans le haut de l'image c'est-à-dire vers l'horizon sont de taille inférieure à celle positionnées dans le bas de l'image pour ne pas risquer d'inclure plusieurs bandes dans une même fenêtre et ainsi fausser les résultats.

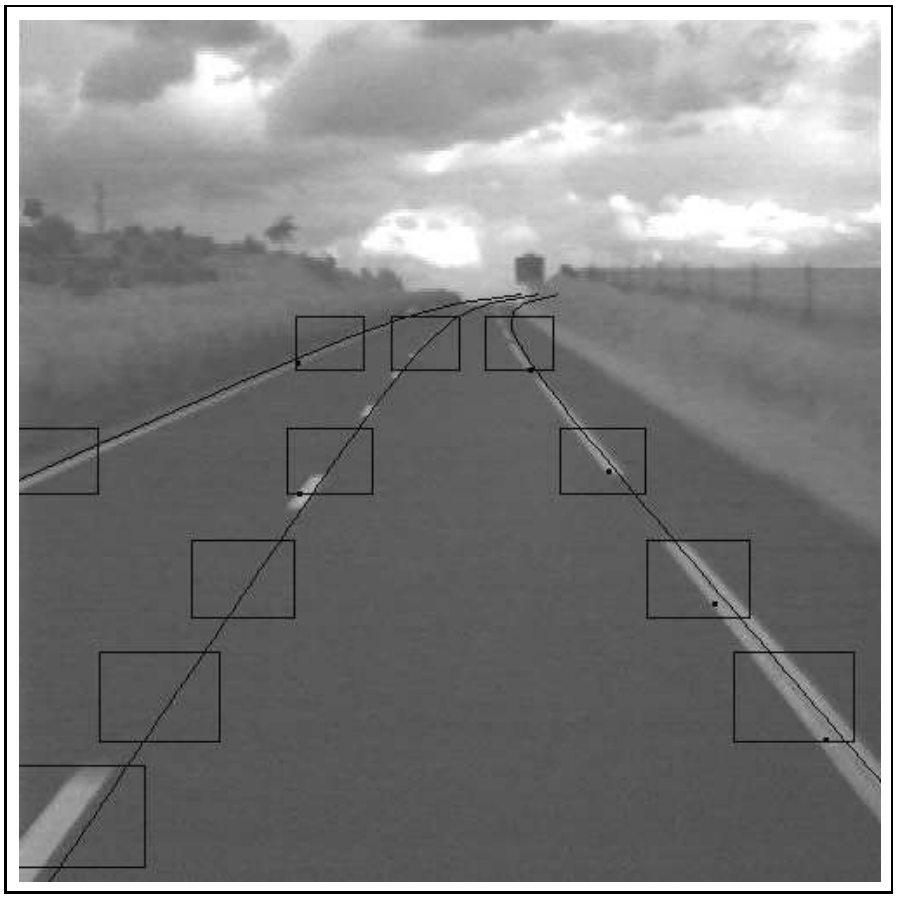


Figure 5.27: Positionnement des fenêtres d'intérêt sur le modèle de la route

La figure 5.27 représente un exemple de scène routière sur lequel sont superposés d'une part le modèle de la route et d'autre part les fenêtres d'intérêt traitées. On peut constater en particulier que le nombre de zones d'analyse de la bande située à gauche du véhicule est plus faible que pour les autres bandes.

5.6.2.2 Phase de détection

La phase de détection a pour objectif l'extraction des bandes de signalisation dans chaque fenêtre d'intérêt. Pour cela, ces bandes sont assimilées à deux segments parallèles. Leur détection fait appel à un calcul de gradient horizontal suivi d'une recherche des segments par transformée de Hough.

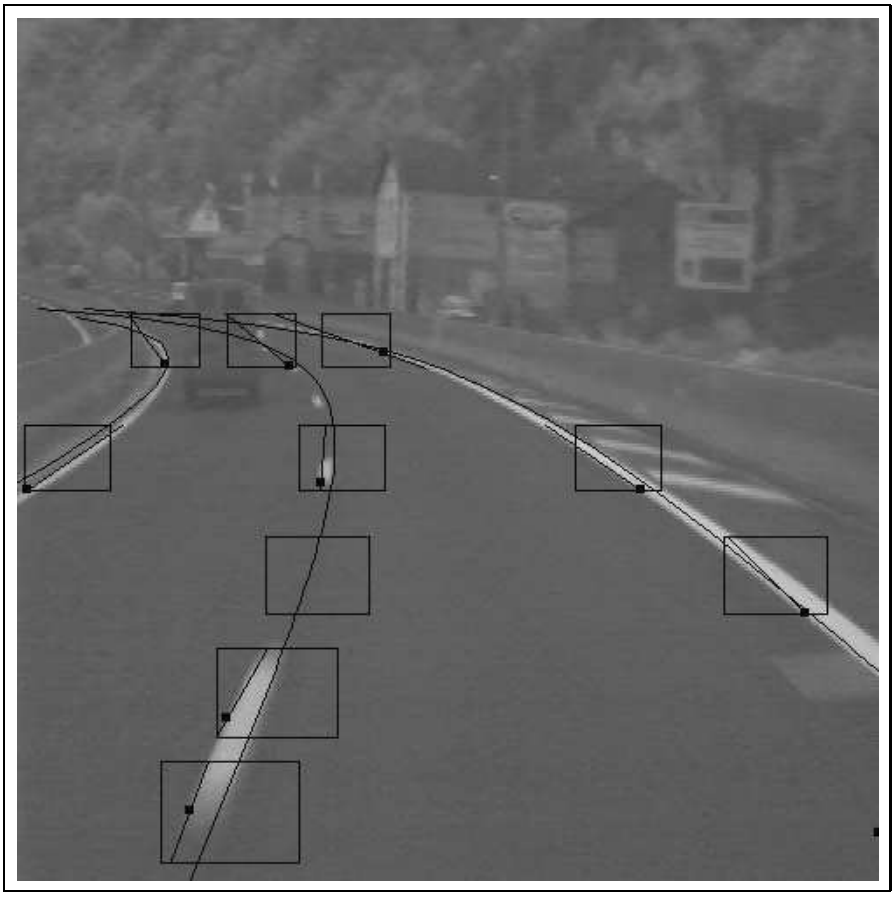


Figure 5.28: Résultats de la phase de détection

Premièrement, dans chaque fenêtre, il est nécessaire de détecter les points de gradient maximum traduisant une variation des niveaux de gris de l'image.

Afin de limiter le nombre d'informations à traiter par l'algorithme de transformée de Hough[18], on ne retient pour chaque ligne de la fenêtre que le point de gradient maximal positif correspondant au bord gauche de la bande de signalisation. Afin de s'affranchir des points parasites correspondant au bord de la route ou à des aspérités de la chaussée, on effectue préliminairement au calcul du gradient une évaluation de la moyenne des niveaux de gris des pixels de la fenêtre d'intérêt et on ne retient que les pixels dont la valeur est supérieur à cette moyenne. En fin de traitement de l'ensemble des lignes, on obtient donc une liste de coordonnées de pixels correspondants aux points de forte dérivée positive.

On recherche alors la droite passant au mieux par cet ensemble de points et correspondant normalement à la bande blanche de signalisation. Cette recherche est effectuée à l'aide d'une transformée de Hough. Celle-ci fournit les paramètres (ρ, θ) (pente et distance au coin supérieur gauche) de la droite passant au mieux par l'ensemble des points détectés. Elle renvoie aussi le point le plus proche de cette droite.

Ainsi, pour chaque fenêtre d'intérêt, la phase de détection donne les informations suivantes (cf. figure 5.28):

① pente de la droite représentant le bord gauche de la bande blanche,

② coordonnées du point optimal par lequel la droite passe.

5.6.2.3 Phase de réactualisation

La phase de réactualisation du modèle est confiée à un filtre de Kalman qui, à partir des résultats de détections dans chaque fenêtre d'intérêt, met à jour les trois paramètres du modèle de la route. La mise en œuvre d'un filtre de Kalman permet d'identifier les systèmes évoluant de manière dynamique au cours du temps. Pour cela, les deux équations suivantes sont utilisées :

$$\begin{cases} x(k+1) &= M(k+1/k).x(k) + G(k).u(k) & \textit{Equation d'état} \\ y(k) &= H(k).x(k) + v(k) & \textit{Equation de mesure} \end{cases} \tag{5.2}$$

avec :

- x le vecteur des trois paramètres du modèle,

[18]La complexité de l'algorithme de transformée de Hough croit comme le carré du nombre de points.

- M la matrice d'état traduisant l'évolution de x dans le temps,
- u le processus générateur régissant l'évolution de x,
- G la matrice de transfert de u,
- y le vecteur d'observation,
- H la matrice de mesure,
- v le vecteur de bruit de mesure.

La mise en œuvre d'un tel filtre permet d'estimer le vecteur d'état x à l'instant $k+1$ à partir des informations disponibles à l'instant k.

5.6.3 Spécification fonctionnelle de l'application

Les paragraphes précédents ont mis en évidence la structure particulière de l'algorithme fondée sur une approche *multi-fenêtres d'intérêt* dont le nombre, la position dans l'image acquise et la taille sont variables dans le temps et remis à jour à chaque itération par une technique de type *prédiction-vérification*.

Premièrement, l'application est basée sur une stratégie de calcul opérant sur un ensemble de fenêtres d'intérêt sur lesquelles on applique successivement différents traitements (seuillage, gradient horizontal et transformée de Hough). Etant donné que ces traitements sont indépendants d'une fenêtre à l'autre, il est possible d'utiliser un schéma de parallélisation encapsulant une stratégie à parallélisme de données. Un simple partage de données (utilisé dans le squelette SCM) n'est pas préconisé du fait que la taille et le nombre des fenêtres d'intérêt n'est pas figé et évolue en fonction des configurations de la route. L'utilisation d'un squelette SCM pourrait alors entraîner un important déséquilibre de charge conduisant à une faible efficacité des implantations résultantes. Pour obtenir une répartition équilibrée sur l'ensemble des processeurs, il est nécessaire de distribuer les traitements de manière dynamique en fonction des besoins de l'application d'où le recours à un squelette de type DF.

Deuxièmement, la stratégie de type prédiction-vérification fait apparaître explicitement la notion de flot continu d'information : les traitements à l'itération i se terminent par la réactualisation du modèle de la route qui sert de point d'entrée de la phase de distribution des fenêtres d'intérêt. La mise en œuvre de ce "rebouclage" nécessite donc l'utilisation d'un squelette ITERMEM.

Ainsi la spécification fonctionnelle de l'application de détection et de suivi de lignes blanches peut s'écrire de la manière suivante :

```
let x0          = init 0 in              (* Initialisation du systeme  *)
let f (x,i)     =                        (* Definition de fonction     *)
  let fenetres = prediction x i in       (* Phase de prediction        *)
  let x'       = df nbproc
                      detection          (* Phase de detection         *)
                      maj                (* Accumulation de detections *)
                      x                  (* Accumulateur initial       *)
                      fenetres in        (* Liste de fenetres          *)
  evolution x' in                        (* Reactualisation du modele  *)

itermem read_img                         (* Lecture d'image            *)
        f                                (* Appel de la fonction f     *)
        affiche                          (* Affichage                  *)
        x0                               (* Etat initial               *)
        (512,512)                        (* Taille de l'image          *)
```

avec :

- la fonction *init* générant l'état initial du système. Cet état comprend d'une part les valeurs initiales des paramètres du modèle de la route et d'autre part les coordonnées et la tailles des zones d'analyse. Cette fonction est appelée lors de la première itération de l'application et en cas de perte de suivi de la route[19],
- la fonction *prediction* extrayant les fenêtres d'intérêt sur l'image courante en fonction du modèle de la route estimée à l'itération précédente,
- la fonction *detection* effectuant l'analyse d'une fenêtre c'est-à-dire la détection des segments correspondant à une bande de signalisation,
- la fonction *maj* représentant la fonction d'accumulation du squelette DF. Elle effectue la mémorisation de l'ensemble des résultats de détection en vue de leur affichage ultérieur et met à jour de manière incrémentale le modèle de la route à chaque fois que la détection sur une fenêtre distribuée est terminée (première phase du filtre de Kalman),

[19]Une perte de suivi peut par exemple être détectée dès qu'un nombre insuffisant de détections est réalisé dans l'ensemble des fenêtres d'intérêt.

- la fonction *evolution* générant le modèle de la route pour l'itération suivante. Cette fonction est appelée une fois que l'ensemble des traitements portant sur les fenêtres d'intérêt est terminé et constitue la deuxième phase du filtre de Kalman,

- la fonction *affiche* permettant de visualiser les informations suivantes : zones d'analyse, bandes blanches détectées, projection du modèle de la route calculé.

Les prototypes des fonctions utilisateur associées à cette spécification sont donnés ci-dessous :

```
void read_image(int nrow, int ncol, image *out);
void init(int val, etat *x0);
void prediction(etat xc, image in, fenetreList* fs);
void detection(fenetre fen, detect *out);
void maj(etat xc, detect r, etat* xs);
void evolution(etat xc,  etat* xs,  etat *xc);
void affiche(etat xc);
```

A partir de cette spécification et des prototypes des fonctions séquentielles, le compilateur fonctionnel *dromadaire* a généré le graphe flot de données représenté sur la figure 5.29 dans le cas où le nombre de processeurs esclaves est égal à 4.

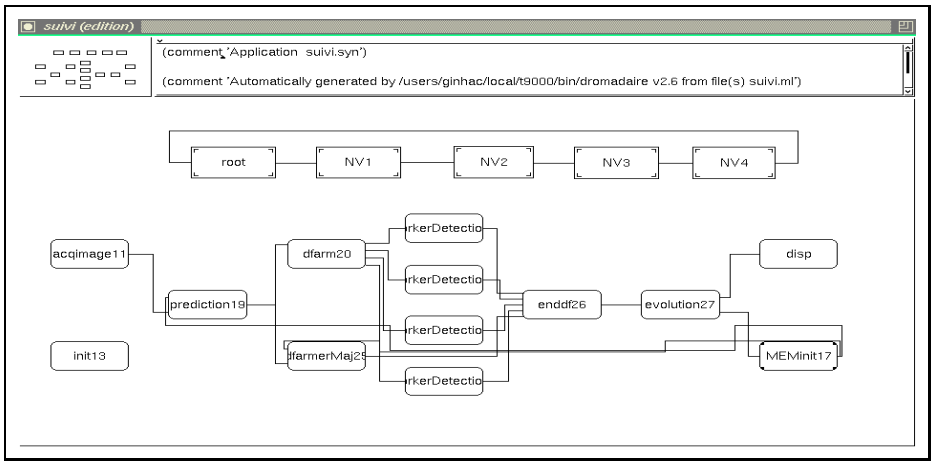


Figure 5.29: Graphe flot de données de l'application de détection de bandes blanches

5.6.4 Résultats d'implantation

L'implantation de cette application reposant sur l'imbrication d'un squelette DF et d'un squelette ITERMEM a été réalisée sur un anneau de processeurs comportant de 2 à 8 processeurs (un processeur maître et un nombre de processeurs esclaves variant entre 1 et 7). Les figures 5.30a et 5.30b décrivent respectivement la latence et l'accélération des implantations parallèles en fonction du nombre de processeurs.

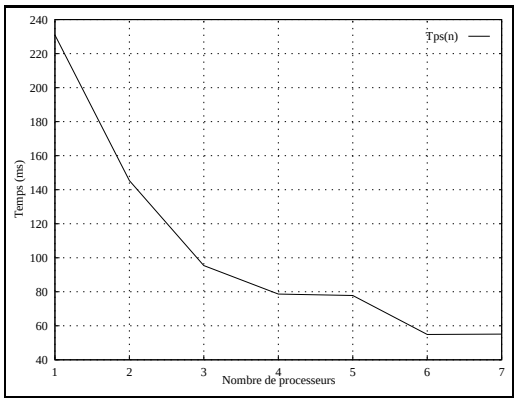


Figure 5.30a

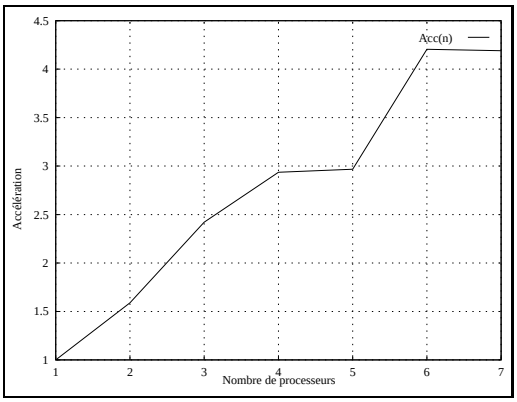


Figure 5.30b

Figure 5.30: Performances de l'application de détection de lignes blanches

Premièrement, nous pouvons noter que la latence minimale est voisine de 55 ms (cas d'un anneau à 8 processeurs). La cadence du flot vidéo étant de 25 Hz, une telle implantation parallèle peut alors traiter une image sur deux ce qui est tout à fait acceptable dans le cadre de l'application. Durant le laps de temps de 80 ms séparant deux images traitées, une voiture roulant à une vitesse moyenne de 100 km/h parcourt une distance de 2.2 m. De fait, tout changement intervenant dans la scène routière — virage, changement de voie du véhicule, etc.— est alors pris en compte rapidement, permettant ainsi au modèle de suivre de manière satisfaisante la route.

Deuxièmement, nous remarquons sur la figure 5.30b que l'évolution de l'accélération n'est pas complètement régulière et se fait par palier[20]. Cela s'explique simplement par le fait que le nombre maximum de fenêtres traitées par l'ensemble des processeurs esclaves n'est égal qu'à 15 (5 par bande). Dans la pratique (cf. figures 5.27 et 5.28), ce phénomène est d'autant plus accentué que le nombre de fenêtres d'intérêt réellement traitées est en général voisin de

[20]Il n'y a par exemple aucun gain de parallélisation entre l'implantation du squelette DF sur 4 processeurs esclaves et celle sur 5 processeurs.

dix seulement puisqu'une partie des zones d'analyses prédites par le modèle de la route est située en dehors de l'image.

Il est alors difficile d'obtenir un équilibre de charge sur 8 processeurs (cf. figure 5.31) d'où des performances inégales très dépendantes des données à traiter.

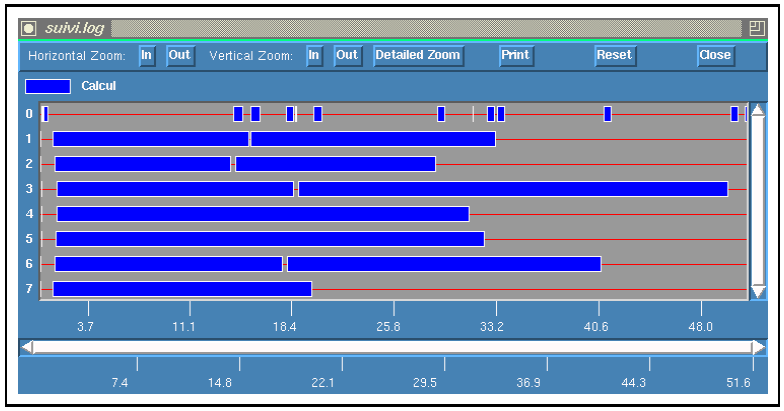


Figure 5.31: Profil d'exécution de l'application de détection de bandes blanches

Afin de quantifier cet effet de déséquilibre, nous avons augmenté délibérément le nombre maximal N_{max} de fenêtres d'intérêt respectivement à $N_{max} = 6$ et $N_{max} = 7$ zones par bande (au lieu de $N_{max} = 5$ dans la spécification originale). Ceci permet par ailleurs de faire croître le nombre de bandes effectivement traitées et de robustifier l'algorithme puisque la précision du modèle de la route estimé à chaque itération est fonction du nombre de détections effectuées dans chaque zone d'analyse.

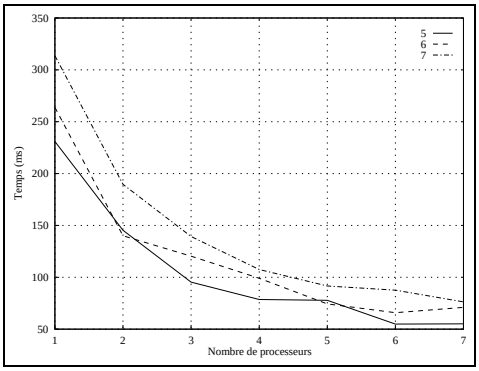


Figure 5.32a

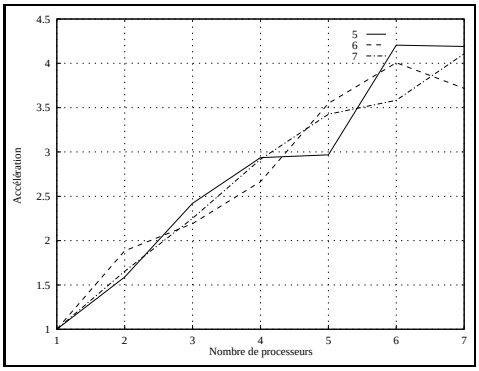


Figure 5.32b

Figure 5.32: Comparaison des performances temporelles

Les figures 5.32a et 5.32b décrivent l'évolution de la latence et de l'accélération de l'application pour chacune des trois valeurs de N_{max}. De manière générale, nous pouvons constater une certaine similarité entre les courbes. Etant donné que le nombre de fenêtres effectivement traitées augmente avec N_{max}, nous pouvons en conclure que pour un nombre de fenêtres plus important, l'implantation parallèle est plus efficace car l'équilibre de charge des processeurs est plus facile à réaliser. De fait, sur un anneau à 8 processeurs et pour des durées d'exécution toujours inférieures à 80 ms, il est préférable d'opter pour la configuration $N_{max} = 7$ qui donne des estimations plus précises du modèle de la route pour des temps de traitement similaires (70 ms).

5.6.5 Conclusion

L'algorithme de détection et de suivi de bandes de signalisation en milieu autoroutier constitue la deuxième application de complexité réaliste implantée sur la machine Transvision par l'intermédiaire de l'outil SKiPPER. Reposant sur deux squelettes imbriqués de type DF et ITERMEM, une telle application a permis de valider l'implantation des techniques dites de **prédiction-vérification**

Dans le même temps, cette application représente une **première expérience** dans la diffusion vers un large public de l'outil de prototypage rapide. L'implantation résultante est le fruit d'une collaboration entre d'une part des développeurs d'applications habitués aux spécifications séquentielles sur station de travail et d'autre part des programmeurs spécialistes de l'architecture Transvision, les premiers étant chargés de fournir le code des fonctions séquentielles de l'application, les seconds réalisant la mise en œuvre de l'implantation et les mesures de performances associées.

Cette expérience a par ailleurs permis de constater l'intérêt de la phase d'émulation séquentielle des programmes évoquée au paragraphe 3.3.3, la mise au point de l'application parallèle ayant eu lieu hors-cible sur simple station de travail.

En ce qui concerne l'implantation sur la machine cible, les performances temporelles de l'application sont tout à fait satisfaisantes (inférieures à 80 ms sur un anneau à 8 processeurs). Des perspectives à court terme sont l'amélioration de la robustesse du suivi. En effet, il est envisageable d'optimiser la phase de réactualisation du modèle en développant une stratégie permettant d'éliminer les détections jugées non pertinentes.

5.7 Conclusion

Dans ce chapitre, nous avons présenté un ensemble d'applications de complexité croissante dont la parallélisation et l'implantation sur la machine cible Transvision ont été réalisés grâce à l'outil de prototypage rapide SKiPPER.

Nous avons tout d'abord décrit un ensemble de trois applications relativement simples de TI bas niveau ne mettant en jeu qu'un seul squelette soit de type SCM, DF, TF. Ceci nous permis de montrer **l'applicabilité** de l'outil SKiPPER en tant qu'outil de **prototypage rapide** d'applications. En effet, chacune de ces applications a été successivement implanté automatiquement sur différentes configurations d'anneaux et pour des volumes de données variables. Les mesures de performances associées ont permis alors de valider les **modèles de performances** de chacun des squelettes présentés au chapitre 4.

Dans un deuxième temps, nous nous sommes intéressés au développement et à la mise en œuvre d'applications de TI bas et moyen niveau de complexité réaliste. La première d'entre elles, à savoir l'étiquetage en composantes connexes, repose sur un enchaînement séquentiel de trois squelettes SCM manipulant chacun des données de type et de taille différents. De tels schémas algorithmiques sont relativement classiques en TI du fait qu'une application est souvent décomposable en un ensemble d'étapes séquentielles allant du bas vers le haut niveau. L'implantation résultante et les mesures de performances associées ont permis d'évoquer le problème de l'enchaînement des squelettes — nécessitant une phase de réorganisation complète des données — dont la formalisation peut conduire à la définition de **règles de transformations** inter-squelettes visant à optimiser les performances des implantations.

Enfin, la dernière application présentée dans ce chapitre constitue une première étape de **rapprochement** entre d'une part les algorithmiciens habitués aux spécifications séquentielles et d'autre part les architectes spécialistes des implantations. Cette expérience a conduit à la mise en œuvre d'un algorithme de détection et de suivi de lignes blanches en contexte autoroutier dont l'implantation sur la machine Transvision donne des résultats à une cadence de traitement tout à fait acceptable.

Conclusion

Les travaux que nous venons de présenter ont pour objectif le prototypage rapide d'applications de traitement d'images sur architecture multiprocesseurs. Nous avons montré que cet axe de recherche conduit à la définition d'outils d'aide à la parallélisation capables de concilier les deux notions antagonistes d'abstraction et d'efficacité des programmes. L'étude de diverses méthodologies existantes et de leurs outils associés nous a permis de définir les principales propriétés auxquelles se doit de répondre un modèle "idéal" de programmation parallèle : facilité de programmation et de compréhension, indépendance vis à vis de l'architecture, mesures prédictives de performances, outils de développement et efficacité des applications.

L'ensemble de ces critères est difficile voire impossible à satisfaire simultanément dans le cas général en raison de la multitude des domaines d'applications candidats à la parallélisation et des architectures dédiées pouvant exécuter ces applications. Cependant, la restriction à un domaine précis — le traitement d'images bas et moyen niveau ici — et à une classe donnée de machines — architecture de type MIMD à mémoire distribuée ici — offre de bonnes opportunités pour définir un compromis entre facilité de programmation et efficacité des implantations. Dès lors, il est possible de définir un modèle de programmation parallèle satisfaisant la majeure partie des critères requis et autorisant en particulier le prototypage rapide et optimisé des applications.

La méthodologie de programmation présentée dans ce mémoire repose sur l'utilisation de constructeurs parallèles nommés squelettes de parallélisation. Ceux-ci sont vus comme des constructeurs génériques de haut niveau exprimant certaines formes classiques et récurrentes de parallélisme et encapsulant les détails relatifs à la mise en œuvre concrète de ces formes de parallélisme. Par rapport aux travaux existants dans le domaine, notre approche a été plus pragmatique et de fait innovante dans le sens où nous avons délibérément limité le domaine d'application au seul traitement d'images bas et moyen niveau. Une telle restriction nous alors permis de définir une collection re-

streinte de squelettes de parallélisation à partir d'une analyse fine *a posteriori* des applications de vision artificielle implantées "manuellement" sur la machine Transvision dans le cadre d'études antérieures. Cette approche "ascendante" offre de meilleures garanties de complétude pour la base de squelettes, complétude identifiée comme un *problème majeur* dans les travaux existants (Comment définir *a priori* une base de squelettes permettant de paralléliser n'importe quelle application ?)

Dans le même temps, le recours à un formalisme fonctionnel permet d'exprimer très naturellement la notion de squelettes dans les programmes. Dans notre approche, ceux-ci sont donc vus comme de simples fonctions d'ordre supérieur définies en Caml et paramétrées par des fonctions séquentielles de calcul développées en langage C traditionnel autorisant ainsi la réutilisation d'opérateurs déjà existants. Une telle approche nous a également permis de dissocier la sémantique déclarative et la sémantique opérationnelle des squelettes. La première, indépendante de toute architecture cible, constitue une spécification exécutable permettant d'émuler les programmes sur machine séquentielle facilitant ainsi les phases de mise au point algorithmique.

La sémantique opérationnelle des squelettes est, quant à elle, entièrement prise en charge par l'outil SKiPPER qui permet de passer automatiquement d'une spécification purement fonctionnelle des applications à un code cible parallèle dédié à une plate-forme MIMD donnée (la machine Transvision dans notre cas). Pour cela, trois modules ont été intégrés à l'outil SKiPPER : le compilateur fonctionnel *dromadaire* effectuant l'expansion des squelettes sous la forme de graphe de processus, l'outil SynDEx réalisant la phase d'Adéquation Algorithme Architecture et enfin le générateur de code cible — dont le noyau a été modifié pour prendre en compte les squelettes fondés sur un schéma de communications dynamiques — tirant parti des spécificités architecturales et donnant en final un code parallèle optimisé.

La démonstration de l'applicabilité d'un tel outil à des problèmes concrets et réalistes de traitement d'images constituait un objectif majeur de nos travaux. Elle a été démontrée dans ce mémoire en décrivant l'implantation de cinq applications de traitement d'images temps réel de complexité croissante. Trois caractéristiques principales ont été ainsi validées lors de ces différentes études d'applications. Premièrement, la notion de prototypage *rapide* a été démontrée, le gain en temps de développement observé pour chacune de ces applications étant au moins de un à deux ordres de grandeurs. Par ailleurs, la mise en œuvre de l'implantation parallèle étant entièrement automatisée et sécurisée, l'utilisateur peut alors se concentrer 1) sur les aspects algorithmiques des solutions (modification des fonctions séquentielles

par exemple), 2) sur les aspects temporels (grâce aux facilités offertes pour évaluer les performances). Deuxièmement, nous avons pu vérifier la fidélité des modèles analytiques de performances permettant de prédire les performances des applications avant toute implantation sur l'architecture cible. Enfin, l'implantation relatée au paragraphe 5.6 a constitué une première expérience de collaboration entre des algorithmiciens et des architectes et a conduit à une première expérience de diffusion de l'outil SKiPPER vers des programmeurs habitués à des spécifications purement séquentielles.

A la lumière de ces observations, SKiPPER apporte d'ores et déjà une réponse originale à la problématique de prototypage rapide optimisé d'algorithmes de traitement d'images. Premièrement, la mise en œuvre d'applications de complexité réaliste confère à l'outil SKiPPER un caractère innovant vis à vis d'autres approches fondées sur les squelettes telles que P^3L ou SCL qui représentent les travaux les plus avancés dans le domaine. Deuxièmement, par rapport à d'autres outils de prototypage rapide tels que SynDEx, SKiPPER constitue un outil de plus haut niveau d'abstraction autorisant l'encapsulation des schémas de calcul-communication exprimés dans les structures de graphes récurrentes sous la forme de squelettes, l'utilisateur n'ayant plus à spécifier le parallélisme potentiel de son application en décrivant complètement le graphe flot de données correspondant.

Il n'en demeure pas moins que les premières réalisations applicatives si elles ont permis de valider l'approche retenue ont également soulevé certaines interrogations et de fait ouvert un grand nombre de perspectives de recherche à plus ou moins court terme.

Premièrement, les applications de traitement d'images étant classiquement décomposées en un ensemble de fonctions allant du bas niveau au haut niveau, la spécification fonctionnelle qui en découle se traduit typiquement par un enchaînement séquentiel d'instanciation de squelettes. L'application d'étiquetage en composantes connexes présentée au paragraphe 5.5 illustre pleinement cet aspect. Cette application a clairement montré le problème posé par la réorganisation des données entre chacune des étapes algorithmiques. Cet aspect est crucial dans le sens où il peut influencer fortement les performances finales de l'application. Si ce point a été géré "manuellement" pour l'application d'étiquetage en composantes connexes (cf. paragraphe 5.5.3.2), l'optimisation systématique des performances temporelles des applications passe inévitablement par la définition et la formalisation des règles de transformation et d'optimisation inter-squelettes. De telles règles utilisées en coopération avec les modèles analytiques de performances des squelettes permettraient alors de transformer (automatiquement ou par suggestion au programmeur) une spécification fonctionnelle d'une application en une autre

équivalente plus performante.

Deuxièmement, la mise en œuvre du parallélisme au sein des applications reposant sur une collection limitée de squelettes, se pose la question de la complétude de cette base (au sens mathématique du terme) vis à vis de notre domaine d'applications. Les travaux présentés ici se sont contentés de définir une bibliothèque constituée de quatre squelettes. Cependant, rien ne peut assurer *a priori* que cette proposition permette de couvrir intégralement les besoins requis par les applications de traitement d'images bas et moyen niveau. Deux questions antagonistes peuvent alors se poser :

① "Est-il nécessaire d'introduire de nouveaux squelettes spécifiques à certaines classes d'applications ?"

② "Ne peut-on pas généraliser certains squelettes existants et les remplacer par un seul et même constructeur d'ordre plus général ?"

Dans le premier cas, nous pouvons constater que certaines applications ne peuvent être facilement et efficacement spécifiées sous la forme de squelettes existants. C'est en particulier le cas de l'application d'étiquetage en composantes connexes pour laquelle l'enchaînement des squelettes SCM s'avère inefficace. On peut alors être tenté d'introduire un nouveau squelette effectuant une étape algorithmique de fusion entre deux opérations de calcul. Ce nouveau squelette, moins général que le squelette SCM, permettrait alors d'accroître de manière significative les performances de l'application d'étiquetage en composantes connexes et, s'il en existe, des applications reposant sur un schéma de parallélisation similaire.

A l'inverse, il semble également possible de limiter encore plus le nombre de squelettes. Par exemple, le squelette DF peut être vu comme une instance particulière du squelette TF dans lequel le nombre de données à traiter par l'ensemble des processeurs esclaves est figé et ne peut évoluer au cours d'une même itération. De même, le squelette SCM peut être vu comme un cas particulier du squelette DF pour lequel la distribution et l'ordonnancement des transferts de données ne sont plus gérés de manière dynamique mais statique. En final, ces trois squelettes ne constitueraient donc que des instances spécifiques d'un même squelette général de type TF. Dans ce cas, la collection de squelettes serait limitée à cet unique squelette de calcul et à un constructeur de type ITERMEM prenant en compte l'aspect itératif des flots d'images.

A l'heure actuelle, ces deux solutions opposées semblent possibles à mettre en œuvre et laissent donc le problème de définir une collection "idéale" de squelettes entièrement ouvert.

Troisièmement, les travaux ayant conduit à la réalisation de l'outil de développement SKiPPER ont abordé les aspects liés à la notion de granularité des traitements. En effet, l'utilisation des squelettes de parallélisation laisse au programmeur le choix de définir la granularité adéquate vis à vis de son application. Ceci peut se faire par le biais, par exemple, de la fonction de partition initiale des données du squelette SCM ou de la liste des données à traiter des squelettes DF et TF. Le temps de cycle de développement particulièrement court au sein de l'outil SKiPPER permet alors d'adopter une approche véritablement expérimentale pour le choix de la bonne granularité.

Il est évident que ces aspects liés à la granularité sont fortement liés aux problématiques d'optimisation intra et inter-squelettes qui font l'objet des travaux succédant à ceux-ci.

Quatrièmement, la méthodologie fondée sur les squelettes de parallélisation mise en œuvre dans ces travaux laisse supposer une plus grande portabilité des applications développées. Cela est vrai dans le sens où nous avons montré que la spécification fonctionnelle des applications ainsi que le code des fonctions séquentielles développées par l'utilisateur restent inchangés en cas de changement d'architecture. Seule la définition opérationnelle des squelettes est à re-écrire par le développeur système. L'effort de portage se limite au portage d'une part d'un ensemble de macros-définitions de l'exécutif générique de SynDEx et d'autre part d'un ensemble de macros-définitions spécifiques aux squelettes effectuant en particulier la gestion dynamique des communications dans les squelettes DF et TF. A l'heure actuelle, ces transferts de données reposent sur la mise en œuvre d'instructions de communications non déterministes (de type *ALT* dans le cas du Transputer). La portabilité de l'implantation des squelettes n'est alors garantie que pour des architectures dont les processeurs supportent ce type d'instructions. C'est le cas pour la nouvelle machine récemment acquise par le LASMEA — batie autour de noeuds de calcul DEC ALPHA et de noeuds de communication T9000 — et qui constituera à court terme la plate-forme cible sur laquelle seront implantées les applications de traitement d'images grâce à l'outil SKiPPER. C'est aussi le cas pour des architectures de type réseau de stations de travail pour lesquelles une implantation des squelettes décrits ici a déjà été proposée (cf. paragraphe 3.3.3)

Cinquièmement, les travaux présentés dans ce mémoire sont dédiés aux applications de traitement d'images bas et moyen niveau. Cependant, certains aspects de ces chaînes de traitement n'ont été que partiellement abordés. Nous pouvons citer en particulier les phases de ré-initialisation de l'application de détection et de suivi de véhicules décrite au paragraphe 1.2.1.2. Quelle que soit l'approche suivie (mono-cible ou multi-cible), ces

phases de ré-initialisation impliquent la gestion de différents opérateurs de traitement à des cadences différentes impliquant l'impossibilité de prédire la durée des itérations. De plus, dans l'approche multi-cible, la complexité de la chaîne algorithmique est accrue du fait de l'entrelacement des tâches de durée et de fréquences d'activation différentes. Nous pouvons également évoquer les systèmes de vision avec coopération de chaînes de traitement comme l'application de localisation de plaques minéralogiques décrite au paragraphe 1.2.1.3. Ce type d'applications pose le problème général de la gestion des applications dont la spécification fonctionnelle est de type $let\ x = f1\ \cdots\ and\ y = f2\ \cdots$. De tels aspects n'ont à aucun moment été abordés et seul l'enchaînement séquentiel d'algorithmes est à l'heure actuelle géré par l'outil SKiPPER. Cette forte limitation ouvre de plus un large champ d'investigation : celui de la composition. Comme dans le cas de l'optimisation de l'enchaînement séquentiel des squelettes, cette notion de composition (au sens de l'imbrication) a de fait des répercussions tant au niveau de la spécification fonctionnelle initiale (mise en évidence de règles de transformation) qu'au niveau de leur implantation (possibilités éventuelles d'optimisation).

En conclusion, les perspectives de recherche que nous venons de décrire offrent d'intéressantes opportunités d'amélioration de l'outil SKiPPER de prototypage rapide d'applications parallèles de vision artificielle sur architecture multi-processeurs.

Bibliography

[Aa95] M. Aspnäs and T. Långbacka. Developping a Customisable Programming Environment for Message Passing based Systems. In *Proceedings of the 4^{th} Nordic Transputer Conference*, pages 370–380, Linkoping, Sweden, Mai 1995.

[ABa92] M. Aspnäs, R.J.R. Back, and T. Långbacka. Millipede - A Programming Environment providing Graphical Support for Parallel Programming. In *Proceedings of the European Workshop on Parallel Computing*, pages 236–247, Barcelone, Spain, Mars 1992.

[ACGK88] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the LINDA machine. *IEEE Transactions on Computer*, 37(8):921–929, Août 1988.

[ADG+96] P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating Heterogeneous Parallel Computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EURO-PAR'96 Parallel Processing*, volume 1, pages 601–614. Springer-Verlag, Août 1996.

[Agh86] G. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[AK91] H.M. Alnuweiri and V.K.P. Kumar. Fast image labeling using local operators on mesh-connected computers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-13(2):202–207, Fev 1991.

[Aln94] H.M. Alnuweiri. Constant-time parallel algorithms for image labeling on a reconfigurable network of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):320–326, Mars 1994.

[AMST97] G.A. Agha, I.A. Mason, S.F. Smith, and C.L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan 1997.

[AWB90] J.Y. Aloimonos, I. Weiss, and A. Bandyopadhyay. Active vision. In *International Conference on Computer Vision*, pages 35–54, 1990.

[Bac78] J. Backus. Can Programming be Liberated from the Von Neumann Style: A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[Bai94] P. Bailey. Algorithmic Skeletons in paraML. TRACS Research Report, Edinburgh Parallel Computing Centre, 1994.

[Ban95] C. Banger. *Construction of Multidimensional Arrays as Categorical Data Types*. PhD thesis, Queen's University, 1995.

[BB91] A. Benveniste and G. Berry. The synchronous approach to reactive and real–time systems. *IEEE Trans. Autom. Control*, 9(79):1270–1282, Sept 1991.

[BCD+97] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An anvironment for structured parallel programming. In L. Grandinetti, M. Kowalick, and M. Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234, 1997.

[BDG+94] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and K. Moore. HeNCE : A Heterogeneous Network Computing Environment. *Scientific Programming*, 3:49–60, 1994.

[BDHR94] A. Bellon, J.P. Dérutin, F. Heitz, and Y. Ricquebourg. Real-time collision avoidance at road-crossings on board the Prometheus Prolab-2 vehicle. In *Intelligent Vehicles Symposium*, Paris, Oct, 24–26 1994.

[BDO+95] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, Mai 1995.

[BDP93] N. Berrington, D. De Roure, and J. Padget. Guaranteeing unpredictability (programming model). *The Computer Journal*, 36(8):723–733, ???? 1993.

[Bel96] A. Bellon. *Détection et suivi de véhicules en mouvement dans une séquence d'images - Implantation parallèle sur un système expérimental à mémoire distribuée*. PhD thesis, Université Blaise Pascal, Oct 1996.

[BGJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, Sept 1991.

[Bir87a] R. Bird. A calculus of functions for progam derivation. Technical Monograph PRG-64, Oxford University Computing Laboratory, 1987.

[Bir87b] R. Bird. A Introduction to the Theory of Lists. In M. Bray, editor, *Logic of Programming and Calculus of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 5–42. Springer-Verlag, 1987.

[BMS80] R.M. Burstall, P.B. McQueen, and D.T. Sannella. HOPE - An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pages 136–143. ACM, ACM, Août 1980.

[BN93] P. Bailey and M. Newey. Implementing ML on Distributed Memory Multiprocessors. In *Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors, Boulder, CO, 1992.*, pages 56–59, 1993. Published as ACM SIGPLAN Notices, volume 28, number 1.

[BOSS95] T. Born, W. Obelöer, L. Schäfers, and C. Scheidler. The Monitoring Facilities of the Graphical Parallel Programming Environment TRAPPER. In *Proc. of EUROMICRO*, Jan 1995.

[Bra92] T. A. Bratvold. Determining Useful Parallelism in Higher Order Functions. In *Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages*, 1992.

[Bra93] T.A. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages*, pages 23–34, Sept 1993.

[Bra94a] T.A. Bratvold. Parallelising a Functional Program Using a List-Homomorphism Skeleton. In Hoon Hong, editor, *Proceedings of PaSCo'94*, pages 44–53. World Scientific Publishing Company, Sept 1994.

[Bra94b] T.A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs*. PhD thesis, Heriot-Watt University, Nov 1994.

[BS91] F. Boussinot and R. De Simone. The ESTEREL language. *Proc. IEEE*, 79(9):1293–1304, Sept 1991.

[Cam96] Duncan K. G. Campbell. Towards the classification of algorithmic skeletons. Technical Report YCS 276, Department of Computer Science, University of York, 1996.

[Can86] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, Nov 1986.

[CDF+97] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO : A Template-based P^3L Compiler. In *Proceedings of the Seventh Parallel Computing Workshop*, Août 1997.

[CGJ+96] T. Collette, C. Gamrat, D. Juvin, J.F. Larue, L. Letellier, R. Schnit, and M. Viala. SYMPHONIE Calculateur Massivement Parallèle : Modélisation et Réalisation. In *3emes Journées Adéquation Algorithme Architecture en Traitement du Signal et Image*, pages 279–286, Jan, 17–19 1996.

[Cha91a] F. Chantemargue. *Segmentation d'images par approche de type division-fusion*. PhD thesis, Université Blaise Pascal, Dec 1991.

[Cha91b] R. Chapuis. *Suivi de primitives image, application à la conduite automatique sur route*. PhD thesis, Université Blaise Pascal, Jan 1991.

[Cha93] F. Charot. Architectures parallèles spécialisées pour le traitement d'images. Rapport de Recherche 1978, Inria, 1993.

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941.

[Col87] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. PhD thesis, University of Edimburgh, 1987.

[Col88] M. Cole. Higher Order Functions for Parallel Evaluation. In C. Hall et al., editor, *Proceedings of the 1988 Glasgow Workshop on Functional Programming*, pages 8–20, 1988. Proceedings published as technical report CSC 89/R4, Glasgow University Computing Science Dept.

[Col89] M. Cole. *Algorithmic skeletons: structured management of parallel computations*. Pitman/MIT Press, 1989.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE : A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, New York, NY, 1987. ACM.

[CSS90] R. Cypher, J.L.C. Sanz, and L. Snyder. Algorithms for image componed labeling on SIMD mesh connected computers. *IEEE Transactions on Computers*, 39(2):276–281, Fev 1990.

[CV96] C. Cudel and B. Vigouroux. Implantation d'un algorithme de compression d'images par transformée en ondelettes sur un réseau de transputers à l'aide de SynDEx. In *3emes Journées Adéquation Algorithme Architecture en Traitement du Signal et Image*, pages 123–130, Jan, 17–19 1996.

[DCLP98] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Ocamlp3l: a functional parallel programming construct. Information available from: `http://qui.di.unipi.it/ocamlp3l.html`, 1998.

[Den75] J.B. Dennis. First Version Data Flow Procedure Language. Technical Memo MAC TM61, MIT laboratory for Computer Science, 1975.

[Der90] R. Deriche. Fast algorithms for low level vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(1):78–86, Jan 1990.

[DFH+93] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming Using Skeleton Functions. In *Parallel Architectures and Languages Europe*, pages 146–160. Springer-Verlag, Juin 1993.

[DGT93] J. Darlington, M. Ghanem, and H.W. To. Structured Parallel Programming. In *Programming Models for Massively Parallel Computers*, pages 160–169. IEEE Computer society Press, Sept 1993.

[DGT+94] J. Darlington, Y. Guo, H.W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Proceedings of the Third Parallel Computing Workshop*, Fujitsu Laboratories Ltd, Kawasaki Japan, Nov 1994.

[DGTY95a] J. Darlington, Y. Guo, H.W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In S. Haridi, K. Ali, and P. Magnussin, editors, *EURO-PAR'95 Parallel Processing*, pages 55–69. Springer-Verlag, Août 1995.

[DGTY95b] J. Darlington, Y. Guo, H.W. To, and J. Yang. Parallel Skeletons for Structured Composition. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 19–28. ACM Press, Juil 1995.

[DMO+92] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8(1-3), Juil 1992.

[DPP97] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *LNCS*, pages 619–628. Springer, Août 1997.

[DSPM92] H. Dubois, O. Sentieys, J.L. Philippe, and E. Martin. Evaluation prévisionnelle de performances d'architectures MIMD : Application au traitement d'images. In *Journées Adéquation Algorithme Architecture en Traitement du Signal et Image*, Sept, 14–15 1992.

[DT95] J. Darlington and H.W. To. Building Parallel Applications without Programming. In J.R. Davy end P.M. Dew, editor, *Abstract Machine Models for Highly Parallel Computers*, pages 140–154. Oxford University Press, 1995.

[Dub91] H. Dubois. *Analyse des Systèmes Multiprocesseurs : Application à la Mise en œuvre sous Contraintes d'Algorithmes de Traitement d'Images*. PhD thesis, Université de Rennes, Jan 1991.

[Ecc92] M. Eccher. *Architecture parallèle dédiée à l'étude d'automates de vision en temps réel*. PhD thesis, Université de Franche Comté, Nov 1992.

[ELS92] F. Ennesser, C. Lavarenne, and Y. Sorel. Méthode chronométrique pour l'optimisation du temps de réponse des exécutifs SynDEx. Research Report 1769, INRIA, 1992.

[Est96] S. Estable. *Reconnaissance d'objets en environnement dynamique - Application à la reconnaissance des panneaux routiers*. PhD thesis, Université Blaise Pascal, Juil 1996.

[For93a] High Performance Fortran Forum. *High Performance Fortran Language Specification*. Rice University, Jan 1993.

[For93b] MPI Forum. Document for a standard message passing interface. Technical Report CS-93-214, Univ. of Tennesse, Nov 1993.

[GA93] A. Geist and All. PVM3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Mai 1993.

[GCS94] J. Gibbons, W. Cai, and D. Skillicorn. Efficient Parallel Algorithms for Tree Accumulations. *Science of Computer Programming*, 23:1–18, 1994.

[Gin95] D. Ginhac. Spécification et implantation d'un algorithme flots de données d'Etiquetage en Composantes Connexes sur la machine multiprocesseurs à mémoire distribuée Transvision. Mémoire de DEA d'Electronique et Systèmes, Université Blaise Pascal de Clermont-Ferrand, Juin 1995.

[Gle95] Glenans. *Le nouveau cours des Glénans*. Editions Le Seuil, 1995.

[GLS98] T. Grandpierre, C. Lavarenne, and Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. Rapport de Recherche 3476, Inria, Août 1998.

[GMW79] M.J. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF*. Lecture notes in computer science ; 78. Springer Verlag, Berlin, 1 edition, 1979.

[Goo94] D.M. Goodeve. *Performance of multiprocessor communications networks*. PhD thesis, University of York, Mai 1994.

[GSD96] D. Ginhac, J. Sérot, and J.P. Dérutin. Evaluation de l'outil SynDEx pour l'implantation d'un algorithme d'étiquetage en composantes connexes sur la machine Transvision. In *3emes Journées Adéquation Algorithme Architecture en Traitement du Signal et Image*, pages 37–44, Jan, 17–19 1996.

[GSD97] D. Ginhac, J. Sérot, and J.P. Dérutin. Evaluation de l'outil SynDEx en vue de prototypage rapide d'applications de traitement d'images sur machine MIMD-DM. *Traitement du Signal*, 14(6):605–613, 1997.

[Hal95] N. Halbwachs. *The declarative code DC, version 1.2a*. Vérimag, Grenoble, France, Oct 1995. unpublished report.

[Hin69] J.R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc*, 146:29–60, 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HP74] S. Horowitz and T. Pavlidis. Picture segmentation by a direct split and merge procedure. In *Second International Conference on Pattern Recognition*, pages 424–433, 1974.

[HPJWe92] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (version 1.2). *SIGPLAN Notices*, Mar, 1992.

[HYMP96] L. Haas, F. Yang, C. Millan, and M. Paindavoine. Deux approches d'implantation parallèle sous SynDEx du filtre de Canny-Deriche optimisé. In *3emes Journées Adéquation Algorithme Architecture en Traitement du Signal et Image*, pages 247–254, Jan, 17–19 1996.

[HYMP97] L. Haas, F. Yang, C. Millan, and M. Paindavoine. Adéquation de l'algorithme de Canny-Deriche généralisé sur architecture DSP avec l'environnement SynDEx. *Traitement du Signal*, 14(6):625–637, 1997.

[Inm89] Inmos. *Occam2 : Manuel de référence*. Masson, 1989.

[JG88] G. Jones and M. Goldsmith. *Programming in Occam2*. Prentice-Hall, 1988.

[Jon93] P.P. Jonjer. A SIMD MIMD architecture for image processing and pattern recognition. In *Computer Architecture for Machine Perception*, pages 222–230, New-Orleans, USA, Dec, 15–17 1993.

[Kes95] Marco Kesseler. Constructing skeletons in clean: The bare bones. In A. P. Wim Bohm and John T. Feo, editors, *High Performance Functional Computing*, pages 182–192, Avril 1995.

[KS98] R. Kocik and Y. Sorel. A Methodology to Design and Prototype Optimized Embedded Robotic Systems. In *2nd IMACS International Multiconference CESA'98*, Hammamet, Tunisia, Avril 1998.

[Lan66] P.J. Landin. The Next 700 Programming Languages. *CACM*, 9:157–166, 1966.

[LCD93] P. Legrand, R. Canals, and J.P. Dérutin. Edge and region segmentation processes on the parallel vision machine Transvision. In *Computer Architecture for Machine Perception*, pages 410–420, New-Orleans, USA, Dec, 15–17 1993.

[Leg95] P. Legrand. *Schémas de parallélisation d'applications de traitement d'images sur la machine parallèle Transvision*. PhD thesis, Université Blaise Pascal, Dec 1995.

[LS93] C. Lavarenne and Y. Sorel. Performance Optimization of Multiprocessor Real-Time Applications by Graphs Transformations. In *Proc. of PARCO'93 conference*, France, 1993.

[LS97] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, 14(6):569–578, 1997.

[LSSt91a] C. Lavarenne, O. Seghrouchni, Y. Sorel, and *al.* The SynDEx software environment for real-time distributed systems design and implementation. In *Proc. of the European Control Conference*, Juil 1991.

[LSSt91b] C. Lavarenne, O. Seghrouchni, Y. Sorel, and *al.* SynDEx un environnement de programmation pour application de traitement du signal distribués. In *13eme colloque Gretsi*, Juan-les-Pins, Sept 1991.

[LVD96] X. Leroy, J. Vouillon, and D. Doligez. The objective caml system. Software and documentation available from `http://pauillac.inria.fr/ocaml`, 1996.

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine. *Communications of the ACM*, 3(4):184–195, 1960.

[McC94] W. F. McColl. The BSP approach to architecture independent parallel programming. Technical report, Oxford University Computing Laboratory, Dec 1994.

[MCMD98] F. Marmoiton, F. Collange, P. Martinet, and J.P. Dérutin. A real time car tracker. In *International Conference on Advances in Vehicle Control and Safety*, Amiens, France, Juil 1998.

[MD92] J. Magee and N. Dulay. MP : A Programming Environment for Multicomputers. In *Proceedings of the IFIP WG 10.3 on Programming Environment for Multicomputers*, Edinburgh, Scotland, Avril, 6–8 1992.

[MIK97] G.J. Michaelson, A. Ireland, and P.J.B. King. Towards a Skeleton-Based Parallelising Compiler for SML. In C. Clark, T. Davie, and K. Hammond, editors, *Proceedings of the Nineth International Workshop on Implementation of Functional Languages*, pages 539–546, Sept 1997.

[Mil77] R. Milner. A Theory of Type Polymorphism in Programming. *J. Comp. Syst. Scs.*, 17:348–375, 1977.

[Mil84] R. Milner. A Proposal for Standard ML. In *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pages 184–197, Août 1984.

[MSA+85] J. MCGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. Sisal : Streams and iteration in a single assignment language. Technical report m-146, Lawrence Livermore National Laboratory, March 1985.

[MSDP93] E. Martin, O. Sentieys, H. Dubois, and J.L. Philippe. GAUT : An Architectural Synthesis Tool for Dedicated Signal Processors. In *EURO-DAC '93*, Sept, 20–24 1993.

[MSP94] E. Martin, O. Sentieys, and J.L. Philippe. GAUT, un outil de synthèse de coeur de processeur dédié appliqué au traitement du signal. *Traitement du Signal*, 13(2):251–279, 1994.

[MSW95] G.J. Michaelson, N.R. Scaife, and A.M. Wallace. Prototyping parallel algorithms in Standard ML. In *Proceedings of British Machine Vision Conference*, Sep 1995.

[Nic92] S. Nicolle. *Gestion parallèle d'applications en vision artificielle dans un environnement distribué.* PhD thesis, Université Blaise Pascal, Dec 1992.

[NJR90] H.T. Nguyen, K.K. Jung, and R. Raghavan. Fast parallel algorithms : from images to level sets and labels. In *Parallel Architectures for Image Processing*, volume 1246, pages 162–176, 1990.

[Par97] Y.H. Park. *Etude en vue de la réalisation d'un noyau temps réel multiprocesseur et l'environnement de développement intégré.* Thèse de doctorat, Université de Technologie, Compiègne, France, Mars 1997.

[Pau91] L. C. Paulson. *ML for the Working Programmer.* CUP, 1991.

[PBM+93] J.P. Paris, G. Berry, F. Mignard, P. Couronne, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, F. Dupont, and C. Le Maire. Projet SYNCHRONE : les formats communs des langages synchrones. Technical Report RT-0157, Inria, Institut National de Recherche en Informatique et en Automatique, Juin 1993.

[Pel93] S. Pelagatti. *A methodology for the development and the support of massively parallel programs*. PhD thesis, Dipartimento di Informatica, Mars 1993.

[Per88] N. Perry. Hope$^+$. Technical Report IC/FPR/LANG/2.5.1/7, Department of Computing, Imperial College, Londres, 1988.

[Pra93] S. Praud. Implantation d'un Algorithme d'Étiquetage en Composantes Connexes sur le Calculateur Fonctionnel. Dea d'électronique, Université d'Orsay - Paris XI, Juin 1993.

[QCSZ93] G.M. Quénot, C. Coutelle, J. Sérot, and B. Zavidovique. Implementing image processing applications on a real-time architecture. In *Computer Architectures for Machine Perception*, 15-17 December 1993.

[QZ91] G.M. Quénot and B. Zavidovique. A data-flow processor for real-time low-level image processing. In *IEEE Custom Integrated Circuits Conference*, Mai, 13–16 1991.

[QZ92] G.M. Quénot and B. Zavidovique. The ETCA massively Parallel Data-Flow Computer for Real-Time Image Processing. In *IEEE International Conference on Computer Design*, pages 492–495, Oct, 14–16 1992.

[RC86] J. Rees and W. Clinger. The Revised Report on the Algorithmic Language Scheme. *SIGPLAN Notices*, 21(12):37–79, Dec 86.

[SC94] D.B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. Technical Report 322, Departement of Computing and Information Science, Avril 1994.

[Sel72] S.M. Selkow. One pass complexity analysis of digital picture properties. *Journal of ACM*, 19(2):283–295, Avril 1972.

[Sen93] O. Sentieys. *Analyse et Synthèse d'Architectures en Traitement du Signal et des Images : Vers la Conception d'Architectures Hétérogènes*. PhD thesis, Université de Rennes, Fev 1993.

[Ser93] J. Serot. *Mise en œuvre d'un formalisme fonctionnel pour la programmation d'une architecture flot de données dédiée au traitement d'images temps reel*. PhD thesis, Université de Paris-sud, Oct 1993.

[Ser97] J. Serot. Embodying parallel functional skeletons: an experimental implementation on top of MPI. In M. Griebl and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 629–633. Springer, Août 1997.

[Sin93] P. Singh. *Graph as a categorical data type*. PhD thesis, Queen's University, 1993.

[Ski90] D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, Dec 1990.

[Ski92] D.B. Skillicorn. Parallelism and the Bird-Meertens Formalism. Technical report, Departement of Computing and Information Science, 1992.

[Ski93] D.B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.

[SMD+93] O. Sentieys, E. Martin, H. Dubois, J.L. Philippe, and M. Corazza. Application de l'outil ESPION pour l'analyse des architectures multiprocesseurs au filtrage de Kalman 2-D rapide. *Traitement du Signal*, 10(1), 1993.

[SMW97] N.R. Scaife, G.J. Michaelson, and A.M. Wallace. Four skeletons and a function. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the Nineth International Workshop on Implementation of Functional Languages*, pages 529–538, Sept 1997.

[Sor94] Y. Sorel. Massively parallel systems with real time constraints. The "Algorithm Architecture Adequation" Methodology. In *Proc. Massively Parallel Computing Systems*, Ischia Italy, Mai 1994.

[Sor96] Y. Sorel. Real time Embedded Image processing Application using the A^3 Methodology. In *IEEE International Conference on Image Processing*, Nov 1996.

[SQZ93] J. Sérot, G.M. Quénot, and B. Zavidovique. Functional programming on a data-flow architecture: Applications in real time image processing. *Intl Journal of Machine Vision and Applications*, 7(1):44–56, December 1993.

[SSBO94] L. Schäfers, C. Scheidler, T. Born, and W. Obelöer. Monitoring the T9000 - The TRAPPER Approach. In *Proceedings of the World Transputer Congress (WTC 94)*, Cernobbio, Italy, Sep 1994.

[SSKF95a] C. Scheidler, L. Schäfers, and O. Krämer-Fuhrmann. Software Engineering for Parallel Systems : the TRAPPER Approach. In *Proc. of HICSS*, Jan 1995.

[SSKF95b] C. Scheidler, L. Schäfers, and O. Krämer-Fuhrmann. TRAPPER : A Graphical Programming Environment for Industrial High-Performance Applications. In *PARLE'93*, volume 694 of *Lecture Notes in Computer Science*, pages 403–413, Jan 1995.

[ST98] D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, Juin 1998.

[Tho96] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, Harlow, England, 1996.

[Tsa85] W.H. Tsai. Moment preserving thresholding : a new approach. *Computer vision, graphics and image processing*, 29:377–393, 1985.

[Tur75] D.A. Turner. An Implementation of SASL. Technical Report TR/75/4, Department of Computer Science, St. Andrews University, St. Andrews, 1975.

[Tur81] D.A. Turner. The Semanticc Elegance of Applicative Languages. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architectures*, pages 85–92. ACM, 1981.

[Tur85] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architectures, Nancy, France*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, New York, N.Y., Sep 1985.

[Via96] M. Viala. Le langage de programmation de SYMPHONIE. Rapport de Recherche 96-24, Cea-Leti, 1996.

[WA91] C.C. Weems and Al. The DARPA Image Understanding Benchmark for Parallel Computers. *Journal of Parallel and Distributed Computing*, 11:1–24, 1991.

[WL93] P. Weis and X. Leroy. *Le Langage Caml*. InterEditions, Paris, 1993.

[Yan98] F. Yang. *Traitement automatique d'images de visages : algorithmes et architecture*. PhD thesis, Université de Bourgogne, Juin 1998.

[Yao96] E. Yao. *H-Linda : un noyau temps réel multiprocessor*. Thèse de doctorat, Université de Technologie, Compiègne, France, Nov 1996.

Appendix A

Implantation des communications dynamiques

A.1 Principe de communication

Dans le cas de l'implantation de la ferme de processeurs avec séparation des chemins physiques de communications (cf. paragraphe 4.4.2.1), chaque processeur (esclave ou maître) doit pouvoir gérer constamment un message venant sur un de ses liens de communication. La gestion simultanée de la diffusion des données et de la collecte des résultats repose sur la mise en place de deux séquences de communication (cf. figure A.1) :

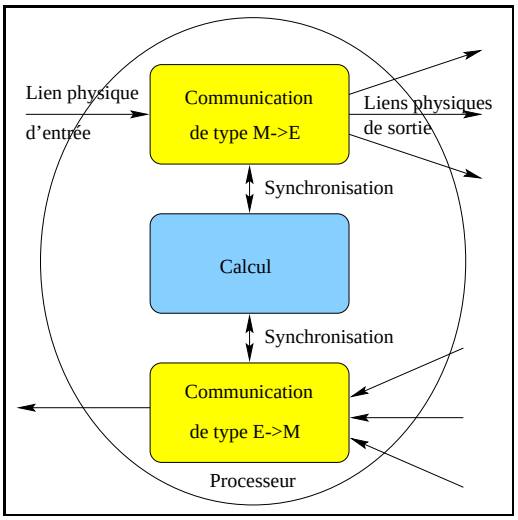


Figure A.1: Séparation physique des chemins de communications dans un squelette DF

- la première séquence gère la diffusion des messages de type maître vers esclaves (M→E),
- la deuxième séquence gère l'acheminement des messages de type esclaves vers maître (E→M).

Chaque séquence de communication d'un processeur esclave possède donc un lien unique à destination du processeur maître (correspondant à la route la plus courte entre ce processeur et le maître) et un ensemble de liens de communication à destination des autres processeurs esclaves. La mise en œuvre des communications dynamiques impliquent donc que chaque processeur connaisse d'une part le numéro du lien de communication avec le maître et d'autre part la correspondance entre les autres liens et les esclaves. Par exemple, dans le cas de la figure A.2, le processeur W_0 utilise trois liens de communications :

① le lien l_0 pour communiquer avec le maître,

② le lien l_1 pour communiquer avec les processeurs esclaves W_2 et W_4 (par l'intermédiaire du processeur W_2),

③ le lien l_2 pour communiquer avec le processeur esclave W_3.

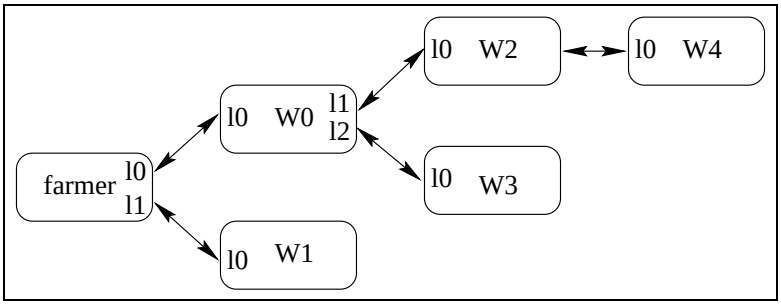


Figure A.2: Un exemple de topologie quelconque

Ainsi, pour pouvoir communiquer avec le processeur W_4, le processeur W_0 réalise un transfert de données sur le lien l_1 à destination du processeur W_4. Le message reçu par le processeur W_2 est ensuite routé à destination du processeur adéquat. Chaque processeur connaît de fait uniquement son environnement proche.

A.2 Mise en œuvre des communications

Pour connaître la correspondance entre les liens de communications et les processeurs esclaves, le principe mis en œuvre utilise les synchronisations statiques de démarrage de la ferme de processeurs (cf. paragraphe 4.3.3.4. Chacune de ces

synchronisations est initialisée par la fonction *init* avec l'indice d'un esclave (de 0 à $n-1$). L'envoi de ces synchronisations permet alors aux processeurs esclaves de s'identifier. Par exemple, l'esclave nommé W_2 sur la figure A.3 est le processeur ayant reçu la synchronisation $sync = 2$.

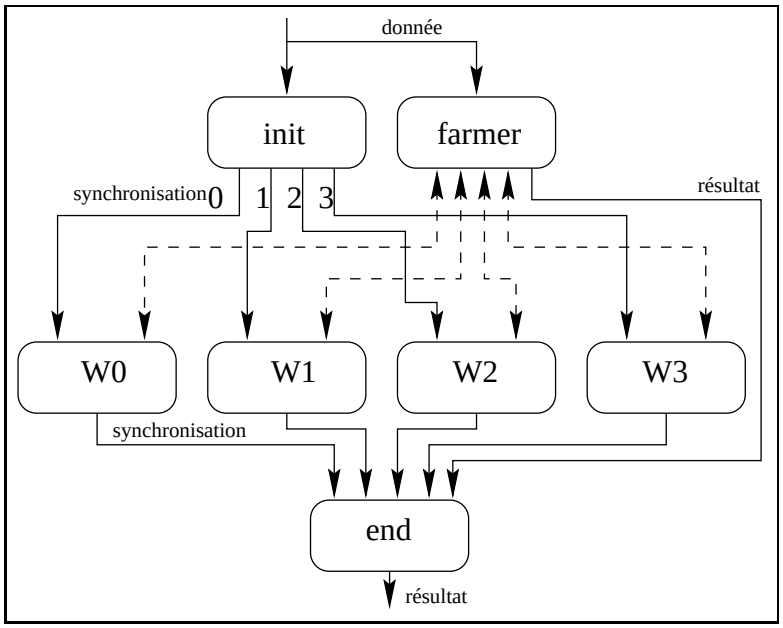


Figure A.3: Détermination des indices des esclaves

De plus lors de l'envoi des synchronisations, il est facile de déterminer pour chaque processeur esclave le lien de communication avec le processeur maître puisque c'est sur ce lien que sont réceptionnées les synchronisations de démarrage.

Enfin, dans le cas où certaines synchronisations doivent être routées à destination d'autres processeurs, il est également facile de définir d'une part le lien utilisé pour effectuer le transfert et d'autre part l'indice du processeur auquel est destinée la synchronisation.

Pour illustrer ces aspects, reprenons l'exemple de la figure A.2 et considérons le cas du processeur W_0. La première synchronisation reçue sur le lien l_0 va permettre de définir que le processeur est l'esclave d'indice 0 ($worker\_nb = 0$) et que le lien utilisé pour communiquer avec le processeur maître est le lien 0 ($link\_to\_root = 0$). Dans un deuxième temps, la réception des trois synchronisations respectivement de valeur 2, 3 et 4 permet de définir que le lien de communication à destination du processeur esclave 2 est le lien 1 ($link[2] = 1$), que le lien de l'esclave 3 a pour numéro 2 ($link[3] = 2$) et que le lien de l'esclave 4 est le lien 1 ($link[4] = 1$).

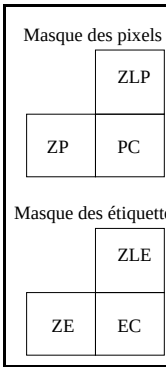
Ce mécanisme général s'appliquant à tout type de topologie permet alors à chaque processeur de connaître son environnement immédiat et ainsi de pouvoir gérer l'ensemble des communications dynamiques mises en œuvre au sein des squelettes DF et TF.

Appendix B

Différents algorithmes d'ECC

B.1 L'algorithme classique

Cet algorithme décrit par exemple dans [Ecc92] et [Pra93] utilise un voisinage de pixels (cf. figure B.1) en 4-connexité situé autour du point à étiqueter. Il repose sur un balayage séquentiel de l'image binaire de haut en bas et de gauche à droite de l'image binaire.



Règles d'attribution des étiquettes			
pc	zlp	zp	ec
0	X	X	0
1	1	0	zle
1	0	1	ze
1	0	0	cpt+1 création d'une nouvelle étiquette
1	1	1	ec = min(ze,zle) et détection d'une équivalence

Figure B.1: Principe de l'algorithme en L inversé

Lors du balayage de l'image, l'algorithme assigne une étiquette au pixel courant en fonction uniquement de son environnement immédiat selon le principe suivant :

- si un des deux voisins[1] possède une étiquette, on affecte cette étiquette au point courant,
- si aucun des voisins n'est étiqueté, on assigne une nouvelle étiquette au pixel courant,

[1]Etant donné le sens du balayage, les pixels situés à gauche et à la verticale du point considéré ainsi que les étiquettes associées sont connus.

- si les deux voisins possèdent leur propre étiquette, il y a détection d'une équivalence entre deux étiquettes et celle de valeur minimale est attribuée au point. Cette situation est caractéristique des composantes connexes en forme de U qui se voient attribuer deux étiquettes. La détection de l'équivalence est découverte au moment de l'attribution de l'étiquette du coin bas et droit de l'objet[2].

Cette première passe de l'algorithme que l'on peut appeler pré-étiquetage produit donc une image d'étiquettes provisoires que l'on doit corriger. Il est alors nécessaire d'effectuer un balayage sur cette image afin de détecter les conflits éventuels d'attribution d'étiquettes. A chaque fois que deux pixels adjacents en 4-connexité possèdent des étiquettes différentes, une table d'équivalence est mise à jour. Le but de cette table est d'associer une seule étiquette à tous les pixels d'une même composante connexe en faisant pointer les étiquettes équivalentes vers celle de numéro minimal.

Par la suite, l'image étiquetée finale s'obtient à partir d'une part de l'image provisoire et d'autre part de la table d'équivalence globale. Cette phase consiste uniquement à remplacer les étiquettes par leur valeur équivalente dans la table.

Cet algorithme présente l'inconvénient majeur de générer un très grand nombre d'étiquettes pour une même composante connexe de l'image. (cf. figure B.2). Cette situation, si elle se reproduit sur des images de taille réaliste, entraîne inévitablement une phase de gestion des équivalences complexe et de fait coûteuse en temps d'exécution.

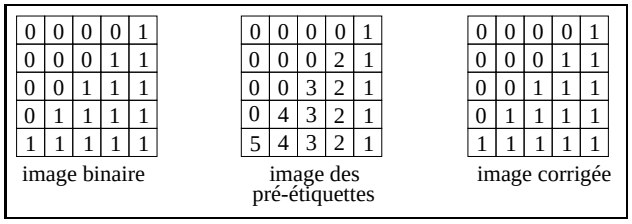


Figure B.2: Un exemple particulier de pré-étiquetage

B.2 L'algorithme par balayage de segment

L'algorithme présenté précédemment génère une nouvelle étiquette dès lors que les pixels situés à gauche et à la verticale du pixels ne font pas partie d'un objet. Pourtant, dans la majorité des cas, l'étiquette attribuée sera équivalente à une autre de numéro inférieur et située sur la ligne précédente.

Pour éviter de telles situations qui se produisent très régulièrement dans le cas d'images réelles, un deuxième algorithme a été envisagée. Celui-ci nommé

[2]Dans le cas de la figure 5.16, on détecte une équivalence entre les étiquettes de numéros respectifs 1 et 3 au coin bas et gauche de l'objet en U

étiquetage par balayage de segment[Pra93] possède une fonctionnement strictement identique à l'algorithme en L inversé sauf dans le cas où les pixels voisins du pont courant sont de valeur nulle. En effet, au lieu de générer une nouvelle étiquette, on effectue un balayage de gauche à droite du segment horizontal[3] auquel appartient le pixel courant. Dès qu'une connexité verticale est découverte, c'est-à-dire dès qu'il y a adjacence de deux segments horizontaux, l'étiquette du segment supérieur est propagée sur le premier pixel du segment de la ligne courante (cf. figure B.3). Une nouvelle étiquette est créée seulement dans le cas où tous les pixels situés au-dessus du segment ont pour valeur zéro.

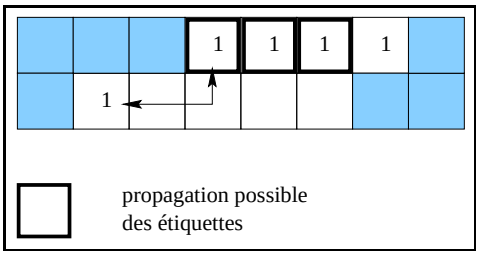


Figure B.3: Principe de l'algorithme par balayage

Cet algorithme d'étiquetage est beaucoup plus performant que l'algorithme classique car le nombre d'étiquettes générées est plus faible et plus proche du nombre réel de composantes connexes présentes dans l'image.

B.3 L'automate de Selkow

L'automate de Selkow[Sel72], troisième et dernier algorithme présenté dans ce mémoire, peut être vu comme un algorithme intermédiaire. Il reprend le principe de l'algorithme classique en utilisant un masque en L inversé, générant ainsi beaucoup d'étiquettes. Néanmoins, cet algorithme intègre simultanément le pré-étiquetage, la gestion des équivalences et une première phase de correction en une seule étape de balayage de l'image.

En effet, dès qu'un conflit entre deux étiquettes est détecté durant le pré-étiquetage, la table d'équivalence est mise à jour. De plus, l'affectation d'une étiquette à un pixel fait appel à une stratégie de pointage "vers le bas" en propageant toujours la valeur de l'étiquette équivalente de numéro minimal (cf. figure B.4).

[3]Par segment horizontal, on entend la suite de pixels de valeur 1, encadrés soit par des pixels de valeur nulle, soit par une fin de ligne à droite, soit par un début de ligne à gauche.

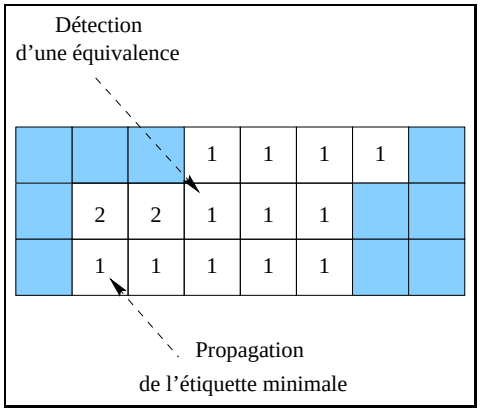


Figure B.4: Principe de l'automate de Selkow