Institut National
Polytechnique de Lorraine

Centre de Recherche en
Informatique de Nancy

THESE DE DOCTORAT D' ETAT ES SCIENCES

(Mention Informatique)

présentée à L'Institut National Polytechnique de Lorraine
par

Abdelwaheb BELAID

Méthodes et outils de programmation de systèmes de traitement d'images : le projet SAPIN

soutenue le 15 juin 1987

Composition du jury :

Président :	J. P. HATON
Rapporteurs :	S. LEVIALDI J. P. FINANCE
Examinateurs :	S. CASTAN R. MOHR P. L. WENDEL

Institut National
Polytechnique de Lorraine

Centre de Recherche en
Informatique de Nancy

THESE DE DOCTORAT D' ETAT ES SCIENCES

(Mention Informatique)

présentée à L'Institut National Polytechnique de Lorraine
par

Abdelwaheb BELAID

Méthodes et outils de programmation de systèmes de traitement d'images : le projet SAPIN

soutenue le 15 juin 1987

Composition du jury :

Président :	J. P. HATON
Rapporteurs :	S. LEVIALDI J. P. FINANCE
Examinateurs :	S. CASTAN R. MOHR P. L. WENDEL

REMERCIEMENTS

Ce travail m'a permis d'élargir mon champ de vision en me plaçant à l'intersection de plusieurs disciplines. Aussi, je suis heureux aujourd'hui de remercier tous ceux qui m'ont conseillé et aidé moralement ou intellectuement à réaliser ce projet dans de bonnes conditions :

Jean Paul HATON, professeur à l'université de Nancy1, de m'avoir accueilli dans son équipe et de m'avoir toujours accordé sa confiance et son soutien. Il me fait aujourd'hui l'honneur de présider ce jury.

Serge CASTAN, professeur à l'université Paul Sabatier à Toulouse et directeur du laboratoire CERFIA, pour l'intérêt qu'il a porté à mon travail en acceptant de faire parti du jury.

Jean Pierre FINANCE, professeur à l'université de Nancy1 et directeur du CRIN, pour avoir bien voulu examiner en détail ce travail dont il a suivi l'évolution.

Stefano LEVIALDI, professeur à l'université de Rome. Il me fait un grand plaisir en acceptant d'examiner mon travail et de le juger.

Roger MOHR, professeur à l'Institut National Polytechnique de Lorraine, d'avoir dirigé ce travail en me donnant conseils et orientations.

Pierre Louis WENDEL, professeur à l'université Louis Pasteur à Strasbourg, d'avoir été à l'origine de ce projet et d'avoir donné les bons conseils pour le démarrage.

Z. Boufriche, pour son concours dans l'élaboration du projet SAPIN que nous avons conçu et réalisé ensemble.

Ph. Clermont et V. Serfaty-Dutronc de l'ETCA, pour leurs précieux conseils et critiques avisées.

N. Levy, J. Souquières et A. Queré de l'équipe de programmation au CRIN, de m'avoir aidé à définir proprement les types image et à structurer de manière à peu près cohérente ce mémoire.

J. Jaray de l'équipe de génie-logiciel au CRIN, de m'avoir aidé à réaliser un prototype même simplifié du langage en Ada.

L'ADI et L'ETCA pour le soutien financier qu'ils ont accordé à ce travail.

Enfin, je tiens à témoigner ma profonde tendresse à Yolande et Nadia de m'avoir soutenu dans ma tâche. Sans leurs encouragements répétés, ce travail n'aurait pas vu le jour.

SOMMAIRE

INTRODUCTION

Les logiciels existants en traitement d'images sont trop spécialisés et difficilement accessibles à des non-informaticiens. L'objectif de ce travail est de concevoir un logiciel de traitement d'images qui soit efficace et simple d'accès à tous les niveaux de la programmation.

L'utilisation croissante du traitement d'images dans différents domaines pour observer les images de notre environnemnt et les analyser a conduit à un développement important du matériel et du logiciel.

Ce développement s'est traduit sur le plan du matériel, par la construction de machines spécialisées permettant la saisie d'images, l'archivage de grandes bases de données et possédant des capacités de traitement supérieures à celles des machines conventionnelles.

Ces différentes investigations ont porté sur la multiplication des unités de traitement et leur spécialisation dans l'exécution de certaines tâches. L'optimisation des traitements sur ces machines passe par une répartition judicieuse des différentes tâches exprimées entre leurs unités.

Pour le logiciel, ceci a d'abord permis la résolution des problèmes au coup par coup, obligeant souvent l'utilisateur à se servir d'outils logiciels non adaptés :

- l'insuffisance d'un langage de traitement d'images et la prépondérance de Fortran pour le calcul scientifique a limité le choix du langage. Plusieurs langages spécialisés ont été ensuite développés mais sont restés, pour la plupart, de trop bas niveau,

- le manque d'outils appropriés pour programmer les machines spécialisées a limité leur emploi. Les systèmes d'exploitation sont souvent conditionnés par l'architecture des machines, ce qui nécessite une certaine expérience pour les utiliser,

- l'absence de l'aide à l'utilisation des fonctions du système laisse peu de chance à un non-spécialiste pour apprendre à s'en servir.

Plus récemment, avec les progrès réalisés en génie logiciel dans la production de logiciel, plusieurs systèmes de traitement d'images adaptés aux besoins des utilisateurs dans ce domaine commencent à se développer.

En relation avec la volonté de développer des outils de programmation adaptés au traitement d'images mais essentiellement portables sur différentes machines spécialisées, nous avons défini un environnement-logiciel appelé **SAPIN**.

Le projet SAPIN est issu d'une proposition de collaboration entre le CRIN et l'ENSPS (Ecole Nationale Supérieure de Physique de Strasbourg) qui développe une nouvelle machine de traitement d'images : **ICOTECH**.

Pour cette machine, destinée à un public d'image assez diversifié, il a été demandé de fournir un ensemble d'outils logiciels permettant de l'utiliser facilement.

Une première étape de ce travail a consisté à faire l'étude de l'existant aussi bien dans le domaine du traitement d'images que dans celui du génie logiciel. Le chapitre 1 montre ce qui est typique au traitement d'images, les solutions matérielles adoptées ainsi que l'apport du logiciel et du génie logiciel pour le développement d'un environnement de programmation spécialisé.

L'étude de SAPIN a tout d'abord porté sur la définition des types image. A partir de leurs propriétés et des opérations usuelles s'est dégagée une idée unificatrice rassemblant les types image autour de leur constructeur. Ainsi s'est affirmée la nécessité d'une description méthodique et hiérarchisée des objets image dans les programmes, ainsi que nous le montrerons au chapitre 2. A partir de là, plusieurs orientations ont ensuite été prises :

- l'étude et l'implantation d'un langage de programmation structurée du traitement d'images : **LPSI**. A partir des définitions des objets, ce langage intègre différents niveaux de programmation :

  ◇ déclaration de types image par instanciation de types plus généraux, en vue de les adapter à chaque application.

  ◇ composition d'objets image afin de permettre des accès globaux à des portions d'images.

  ◇ définition d'opérations globales permettant de manipuler facilement ces objets et pouvant conduire, après compilation, à la production de codes optimisés pour la machine cible.

  ◇ création de tâches indépendantes susceptibles d'être parallélisées dans la phase d'implantation.

  Le chapitre 3 présente la syntaxe du langage LPSI et montre les différentes compositions possibles d'objets pour réaliser les accès globaux à des ensembles de points. Nos choix des structures de contrôle et de l'expression du parallélisme y sont exposés.

- la définition d'un logiciel interactif **SAPIN-NI** pour la manipulation d'images par des non-informaticiens. Ce logiciel comprend un noyau de fonctions de base sous forme de menu et plusieurs outils d'aide à l'apprentissage progressif des fonctions du système. Ce mode d'utilisation assisté fournit de l'aide à l'utilisateur à tous les niveaux du système.

  Dans une étape plus avancée, l'utilisateur peut créer son propre menu par la définition de nouvelles fonctions. Cette phase plus élaborée permet en particulier de créer des macro-fonctions à partir de traces d'exécution.

  Le chapitre 4 montre nos différents choix et les outils qui en découlent pour apporter de l'aide aux programmeurs, pour réaliser la communication entre les fonctions du menu et pour permettre l'intégration rapide de nouvelles fonctions.

- l'objectif visé étant la réalisation d'un système effectif, il est nécessaire de se préoccuper d'implantation. Pour les types image, cette étape est facilitée par la hiérarchisation des données qui ramène l'implantation des types à celle des constructeurs.

  Pour les instructions, un détecteur de parallélisme est étudié. Il propose des mécanismes qui permettent de produire des schémas de traduction optimaux sur l'architecture cible. La stratégie de parallélisation est différente suivant qu'il s'agit d'une machine multiprocesseurs ou pyramidale.

  Pour des machines multiprocesseurs, le parallélisme se fait par l'appel simultané à plusieurs fonctions cablées. Ce type de parallélisme impose, au niveau du programme, une expression concise des opérations.

  Dans ce cas, le précompilateur du langage détecte les opérations parallèles dans le programme et engendre les appels aux fonctions spécialisées de la machine cible.

  Pour des machines où le parallélisme est plus sophistiqué, comme les machines systoliques ou pyramidales, il faut en plus étudier les mouvements de données entre les différents processeurs, ce qui pourrait conduire malgré la complexité de l'architecture à réaliser des optimisations plus importantes que dans le cas précédent.

  Dans le cadre d'une collaboration avec l'ETCA (Etablissement Technique Central de l'Armement à Arcueuil), nous avons étudié la parallélisation de quelques algorithmes séquentiels sur la machine **SPHINX**. Cette étude nous a permis de trouver des solutions optimales pour des problèmes mal conditionnés pour le parallélisme.

  Nous nous intéressons dans le chapitre 5 à ces deux types d'architecture et proposons des schémas de traduction pour certaines opérations parallèles.

CHAPITRE 1 : PRESENTATION GENERALE

Suivant les objectifs dégagés dans l'introduction, ce chapitre comporte trois parties. La première montre les caractéristiques du traitement d'images et les problèmes qu'elles induisent lors du développement et de l'exécution de programmes. La seconde donne les idées et solutions architecturales. Le but étant de pouvoir communiquer avec la machine et avec les autres utilisateurs, la troisième partie montre l'apport du logiciel et du génie logiciel.

1. CARACTERISTIQUES DU TRAITEMENT D'IMAGES

De manière générale, le traitement d'images regroupe les techniques de codage, d'archivage, de filtrage et d'analyse pour la reconnaissance et l'interprétation de formes.

Ces techniques ne s'appliquent pas de manière universelle. Plusieurs contraintes liées au volume des données, à la nature et au domaine d'utilisation des images, nécessitent quelques adaptations. Elles conduisent le plus souvent à chercher des solutions optimales à tous les niveaux de la programmation.

1.1. Le volume et la vitesse

Malgré l'augmentation de la taille des mémoires centrales et des disques magnétiques, on s'aperçoit que l'imagerie est encore gourmande en place. Une seconde d'image standard occupe 6,4 M-octets, ce qui remplit vite la place dont on dispose. Le disque optique est considéré comme le futur support "convenable" avec une capacité de 1000 M-octets. Plusieurs images de la même application peuvent y être rassemblées, ce qui permet de résoudre de nombreux problèmes d'archivage.

Pour cette raison, plusieurs méthodes de compression ont été développées en vue de réduire le volume des données. Cette solution a été préférée à celle qui consiste à multiplier les supports de rangement et à traiter les images par bloc ce qui n'aurait fait que remplacer un problème de place par un problème de débit.

La réduction des données en traitement d'images devient la clé d'optimisation des programmes. Elle est utilisée à tous les niveaux d'analyse, car il n'existe pas de matériel assez puissant pour exploiter les données et en extraire les formes dans des limites de temps

raisonnables.

Pour donner un ordre de grandeur, considérons une image de télédétection mémorisée selon une résolution moyenne de 2000x3000 pixels. Chaque pixel est codé sur 8 bits. Si l'on suppose qu'un satellite envoie 50 images/s, cela signifie que les 50 images (représentant 400 Méga-pixels) seront traitées dans le même intervalle de temps, c'est-à-dire qu'un pixel sera traité toutes les 2.5 nanosecondes. Dans le but d'avoir des systèmes de traitement d'images fonctionnant à cette vitesse, plusieurs opérations sur pixel doivent être réalisées dans cet intervalle de temps, ce qui est impossible pour les machines conventionnelles.

Cela condamne le temps réel pour plusieurs applications sur ces machines mais oblige à faire deux types d'optimisation :

- pour le programmeur, il faut absolument éviter les méthodes de calcul faisant intervenir des nombres d'opérations augmentant trop rapidement avec le nombre des données,

- pour le concepteur de machine, il faut construire des architectures comprenant plusieurs unités de traitement pouvant se partager le travail.

1.2. Quelques objets propres

Le traitement d'images véhicule un certain nombre d'objets parmi lesquels nous trouvons bien sûr les **images**, mais aussi des sous-ensembles extraits de l'image comme les **contours**, les **régions** et un certain nombre de fonctions comme les **tables de couleurs** et les **masques de convolution**.

Cette liste n'est pas exhaustive. Elle regroupe les objets les plus propres au domaine.

La donnée centrale est l'image. C'est la représentation spatiale discrète d'un objet ou d'une scène. Pour des considérations topographiques, l'image est décomposée en cellules élémentaires appelées "pixels" ou "pels". Il existe une relation d'ordre sur les pixels introduite par cette décomposition liée au principe de tramage des appareils d'acquisition.

L'information contenue dans une image est variable. Elle peut être des niveaux de gris correspondant à des mesures d'intensité lumineuses perçues aux différents points. Quand les niveaux sont réduits à 2, l'image est dite binaire. Dans une image couleur, l'information en chaque point est une combinaison de la teinte, de la saturation et de la luminance. Dans une image tridimensionnelle, c'est la distance du point à l'observateur.

Une variété d'algorithmes existe pour traiter une image en vue d'en extraire des indices pertinents pour la reconnaissance. Parmi ces

indices, les plus courants sont les régions et leurs frontières appelées souvent contours.

Une région représente un objet entier où une partie d'un objet. C'est donc un élément essentiel à la reconstruction des formes représentées dans une image.

Une région est repérée dans une image par sa forme, sa texture ou sa couleur. Bien des méthodes existent pour extraire une région dans une image et lui faire subir des transformations.

Le contour est l'information duale à la région. Il est associé à la notion de contraste local dans l'image, synonyme de variation brutale de l'intensité lumineuse et donc de frontière. Il permet une représentation des objets par les frontières, ce qui constitue une sorte de codage de l'information.

Il existe deux grandes techniques d'extraction de contours : celles qui recherchent des éléments de contours et qui se basent sur l'examen local de la variation de l'intensité et celles qui construisent les vraies frontières par suivi de ces éléments et "bouchage" des interruptions.

Le masque représente bien souvent une fonction de convolution qui permet d'effectuer différents types de filtrage dans l'image et extraire, par exemple, les contours. Il est aussi utilisé comme élément structurant dans des opérations de morphologie mathématique pour réaliser des dilatations ou des érosions de formes [SER 82]. Sa grande fréquence d'utilisation et sa représentativité des opérations locales du traitement d'images permettent de le considérer comme un objet de grande utilité.

Les tables de couleurs permettent des conversions de couleurs dans les images. Elles équipent la plupart des machines spécialisées et se placent en aval de la chaîne de traitemnt avant l'affichage. Elles sont aussi utilisées comme anamorphoses et effectuent toutes sortes de transformations ponctuelles sur l'image. On les considère comme les exemples de fonctions tabulées utilisés en traitement d'images.

Nous reviendrons dans le chapitre 2 sur une définition plus approfondie de ces objets.

1.3. La diversité des utilisateurs et des machines

Le traitement d'images est un domaine qui englobe des activités très variées allant de l'inspection industrielle à l'animation. Ces applications ont des exigences différentes. Certaines traitent des images bruitées de grands formats et sont si limitées par la vitesse qu'elles sont prêtes à sacrifier la précision, tandis que pour d'autres la précision a une grande importance. Il ne serait pas réaliste d'avoir un seul système pour toutes ces applications, mais plutôt une

méthodologie de programmation commune.

Les utilisateurs du traitement d'images ont des expériences différentes de calcul. Certains sont météorologues travaillant sur des images prises par satellite, tandis que d'autres sont médecins désirant étudier des images microscopiques et pour lesquels tous les détails ont leur importance.

Parmi ces utilisateurs, certains sont non-informaticiens et désirent être assistés par le système, tandis que d'autres sont informaticiens et ont besoin d'outils spécialisés pour exprimer eux-mêmes leurs idées et les programmer.

Très peu de systèmes de traitement d'images ont intégré toutes les facilités nécessaires pour les rendre généraux. Bien que les techniques de structuration et de programmation existent déjà, la plupart de ces systèmes ont été écrits pour une optimisation d'une certaine application.

Les architectures des machines connaissent elles aussi des différences. Certaines sont de type VON NEUMAN ne réalisant qu'une instruction à la fois, tandis que d'autres offrent des degrés de parallélisme multiples (pipeline, multiprocesseurs, systolique, etc...). Dans le premier cas, l'optimisation passe par une organisation judicieuse du séquencement des instructions, tandis que dans le second cas, une méthode de programmation distribuée est plus indiquée. La programmation sur ces machines reste très orientée architecture, cherchant plutôt une meilleure exploitation des possibilités du matériel aux dépens d'une plus grande clarté des programmes.

2. LES IDEES ET SOLUTIONS DE L'ARCHITECTURE

Les applications de traitement d'images sont très variées. Elles englobent la restauration d'images, la compression et le codage, la gestion de bases de données, l'analyse et l'interprétation de scènes. Les algorithmes utilisés dans ces domaines peuvent être répartis en deux niveaux :

* le **bas niveau** contient l'acquisition, la compression et le codage, la restauration et le rehaussement du contraste, la segmentation pour l'extraction de contours et le regroupement de régions, la détection d'objets, l'analyse de textures, etc...

* le **haut niveau** est plus concerné par les problèmes d'analyse et d'interprétation faisant appel à des techniques d'intelligence artificielle. Il interagit avec le bas niveau pour disposer d'informations nécessaires extraites de l'image.

L'architecture pour le traitement d'images est concernée pour sa plus grande partie par le bas niveau. En effet, les opérations sont simples, facilement caractérisables, souvent indépendantes d'un pixel à

l'autre et se prêtent donc bien au parallélisme. Le format des données est unifié, représenté par une structure bidimensionnelle appelée **IMAGE**. Cette représentation est en général inadéquate pour les traitements de haut niveau où l'interdépendance des données exige souvent une représentation plus complexe.

2.1. Les opérations de base

Les opérations de base utilisées par le bas niveau et qui nécessitent une amélioration du temps de calcul sont :

- entrées/sorties,
- opérations ponctuelles,
- opérations de voisinage,
- transformations globales.

2.1.1. les opérations d'entrée/sortie

Pour être conformes avec les systèmes de télévision, qui sont les plus rapides en entrée/sortie (acquisition et affichage vidéos), il semble naturel d'adapter les organes de traitement des systèmes au format TV et à la vitesse TV. Avec la venue de composants très rapides (convertisseurs A/D, D/A, multiplieurs, RAM, etc...), une vitesse de transfert des données de l'ordre de 10 Méga-octets par seconde est tout à fait possible. Plusieurs systèmes commerciaux fonctionnent déjà à cette vitesse.

2.1.2. les opérations ponctuelles

Les opérations ponctuelles modifient la valeur de chaque pixel de l'image de manière indépendante de son voisinage. Elles englobent les opérations arithmétiques et logiques, les opérations de calcul de valeurs particulières comme le maximum, le minimum et les opérations de seuillage. Quand l'opération admet comme opérandes deux images, les opérations ponctuelles s'effectuent sur les points homologues dans ces deux images.

Cette indépendance de calcul entre les pixels et la répétition de la même opération élémentaire sur tous les pixels favorisent le parallélisme SIMD (Single Instruction Multiple Data). Les array-processeurs sont construits sur ce principe et permettent ainsi de réduire le temps de calcul de l'image à celui d'un seul pixel.

2.1.3. les opérations sur voisinage

Les opérations sur voisinage (appelées aussi opérations locales) s'appliquent uniformément sur tous les pixels de l'image. Elles calculent, pour chaque pixel, une valeur dont dépend tout son voisinage.

Ces opérations englobent non seulement les convolutions (filtres linéaires avec des tailles modérées), une variété de filtres non linéaires (médiane, zonale, direction sensitive, etc...) [GRA 82] [PRA 78], mais aussi des transformations de morphologie mathématique telle que la dilatation et l'érosion [SER 82].

2.1.4. les transformations globales

Les transformations globales nécessitent, pour le calcul de la valeur d'un pixel, tous les pixels de l'image. Elles concernent les transformées de Fourier, Cosinus, Hadamard, etc... Leur influence sur la réalisation de nouvelles architectures n'a pas été très importante ; réputées pour leur lenteur, elles servent surtout dans les comparaisons de performances des machines.

Dans tous les cas, les algorithmes de traitement d'images utilisent la valeur du pixel et celle de ses voisins. C'est cette notion de voisinage qui pose les plus grandes difficultés, raison pour laquelle plusieurs concepteurs de machines ont réalisé des architectures privilégiant la prise en compte de voisinages par la construction de diverses connexions entre les processeurs voisins. C'est ainsi que sont nés les array-processeurs, les réseaux systoliques et les machines pyramidales.

Sans rentrer dans la taxonomie des architectures qui a fait couler beaucoup d'encre ces dernières années [CAS 82] [PRE 82] [UHR 84] [BAS 85] [CAS 86], nous allons seulement rappeler les différents types de parallélisme. Cela aidera à comprendre certaines optimisations algorithmiques que nous évoquerons dans les prochains chapitres.

2.2. Les types de parallélisme

Les différents types de parallélisme réalisés dans les systèmes de traitement d'images sont :

- parallélisme des bits par pixel,
- parallélisme des voisinages,
- parallélisme des images,
- parallélisme des tâches.

2.2.1. le parallélisme des bits par pixel

C'est le parallélisme de très bas niveau utilisé dans les processeurs binaires. Les bits de la donnée sont traités simultanément pour réaliser l'opération. C'est un parallélisme de plus que l'on peut opposer au bit-série qui, malgré sa lenteur, reste plus commode et plus facile à réaliser que le bit-parallèle.

2.2.2. le parallélisme des voisinages

Un processeur spécialisé est utilisé pour réaliser les mêmes opérations sur tous les voisinages de l'image. Le processeur accède de manière parallèle à tous les pixels du voisinage, calcule une valeur pour le voisinage (de manière parallèle ou séquentielle) et passe ensuite au voisinage suivant. Une des premières machines à implanter le parallélisme de voisinage est GLOPR à Perkin-Elmer Corporation par PRESTON [PRE 71].

GLOPR opère uniquement sur des images binaires et utilise une grille hexagonale. L'image est décalée à chaque fois, de telle sorte que le processeur de voisinage puisse se déclencher et accèder de manière parallèle aux 7 bits dans le voisinage du pixel en cours. Là aussi, l'algorithme n'a pas d'action importante sur la parallélisation des tâches. Il peut seulement indiquer les opérations pour lesquelles une telle exécution est possible.

PICAP1 [KRU 82] contient deux processeurs de voisinage : un processeur logique et un processeur de convolution opérant sur des voisinages 3x3. Les chemins de données des registres image aux processeurs peuvent être reconfigurés de telle sorte que les neuf valeurs envoyées aux processeurs ne soient pas toujours celles du voisinage d'un

point. De cette manière plusieurs opérations sur voisinage peuvent être réalisées.

2.2.3. le parallélisme des images

Dans ce cas, un nombre important de processeurs (tableau nxn où n est important) est déployé pour exécuter, dans un total synchronisme, la même instruction sur des pixels différents. C'est le parallélisme SIMD dans la classification de FLYNN [FLY 72]. L'architecture des processeurs peut être une CPU générale ou de type bit-série. Tous les processeurs calculent conjointement les valeurs des pixels différents.

Plusieurs opérations ponctuelles ou locales signalées précédemment peuvent être réalisées de cette façon. Seulement, dans le cas des opérations locales, où tous les pixels n'ont pas le même type de voisinage (pixels sur et à l'intérieur des frontières), ce type de parallélisme est contraint à quelques limitations.

En effet, la programmation de ce type de parallélisme n'est pas évidente quand la taille de l'image dépasse celle du tableau de processeurs. Si la machine n'autorise pas le décalage automatique d'une zone d'image à la suivante, cela entraînera des problèmes de recouvrement qui ne sont pas faciles à résoudre [CAS 85].

Les exemples de machines réalisant ce type de parallélisme sont CLIP IV [DUF 76] et MPP [BAT 80].

Le MPP (Massively Parallel Processor) comprend un tableau de 128x128 processeurs. Chaque processeur est relié à ses quatre voisins les plus proches sous le contrôle du programme. De manière à éviter les problèmes aux frontières signalés ci-dessus, l'image est représentée sous la forme d'un tore où la première ligne et la dernière sont reliées ainsi que la première colonne et la dernière. Chaque processeur a une mémoire locale de 1k bits ce qui permet de faire des repliements d'images dans le cas où leurs dimensions sont grandes.

2.2.4. le parallélisme des tâches

Le traitement est réparti en tâches élémentaires. Chaque processeur exécute sa propre tâche. C'est le parallélisme MIMD dans la classification de FLYNN. Suivant que les processeurs sont synchrones ou asynchrones, le parallélisme sera réalisé de deux manières différentes.

• Dans le premier cas, tous les processeurs peuvent coopérer pour réaliser chacun une partie de l'opération demandée. Un problème important lié au bon fonctionnement d'un tel système est celui de la

synchronisation entre les processeurs. Les processeurs peuvent avoir des temps différents entre l'exécution et l'obtention des résultats. De plus, les données d'un processeur peuvent être les résultats d'un autre processeur. C'est le cas des machines pipeline.

Un exemple typique des sytèmes pipeline est celui des réseaux systoliques [KUN 82]. La circulation des données dans le réseau obéit à un rythme de cadencement qui impose, à toute donnée acquise par une cellule, d'être utilisée par toutes les cellules où elle transite.

Le pipeline est bien intégré dans les machines de traitement d'images comme le CYTOCOMPUTER [STE 79] pour réaliser des filtrages logiques (opérations de morphologie mathématique) sur des images médicales. Une tâche de traitement d'images est divisée en plusieurs opérations. Chaque opération est exécutée par une cellule. Ces cellules sont enchaînées ensemble dans une structure linéaire. Chaque cellule a une structure pipeline. Les images sont rentrées ligne par ligne à travers les cellules. Quand toutes les cellules sont remplies (9 dans le cas d'un voisinage 3x3), on leurs applique une opération de voisinage qui peut être différente d'un traitement à un autre. Le temps d'une opération sur un voisinage 3x3 est de l'ordre d'une milliseconde. Le débit du pipeline est 1.6 M octets/s. Cela signifie qu'une image 256x256 est traitée en 40 ms.

Un dialogue permanent existe entre le Cytocomputer et un ordinateur frontal qui détermine en fonction de la tâche, le nombre de cellules à utiliser.

• Dans le second cas, chaque processeur exécute ses propres instructions. Les instructions effectuées simultanément peuvent être de types très différents.

Ce type d'architecture est plus flexible que les précédents pour la réalisation d'applications différentes mais pose des problèmes de contrôle plus complexes que dans le cas du SIMD. De plus, la conception d'algorithmes orientés vers le traitement de tâches nécessite un grand effort d'optimisation pour utiliser au maximum la puissance des multiprocesseurs.

Le rendement de telles architectures est fonction du nombre de leurs processeurs et est aussi lié aux mécanismes de transfert et de partage des données qui constituent un goulot d'étranglement pour ce type d'architecture [BAS 82], [TAN 83].

L'exemple le plus classique des machines intégrant un tel parallélisme est la machine PICAP II [KRU 82].

La totalité de la machine PICAP est de type MIMD. Les processeurs sont indépendants mais partagent une mémoire commune. La mémoire est partagée en 16 tranches indépendantes de 256 k-octets chacune. La mémoire peut être accédée par n'importe quel processeur à l'aide d'un canal DMA relié au bus. L'accès simultané à plusieurs zones images rangées chacune dans une mémoire par les processeurs réalise une forme de parallélisme dans PICAP.

Pour augmenter davantage la vitesse, les processeurs utilisent eux-mêmes différents types de parallélisme. Les processeurs sont commandés par l'ordinateur hôte. Ils traitent l'information rangée dans une mémoire et renvoient le résultat à l'ordinateur. Le bus utilisé peut supporter jusqu'à 15 processeurs.

Un autre type d'architecture pouvant rassembler les deux formes de parallélisme SIMD et MIMD est l'architecture pyramidale. En effet, plusieurs algorithmes de traitement d'images conduisent à une représentation de l'image en multirésolution, comme la détection de contours, et sont plus efficaces sur des résolutions réduites de l'image, en affinant successivement les traitements.

La représentation pyramidale d'une image repose sur une décomposition récursive de celle-ci en quadrants. La racine de la pyramide représente l'image entière, ensuite de manière récursive, chacun de ses quatre fils représente un quadrant de la zone de l'image représentée par son père.

Les machines fondées sur ce type de représentation sont appelées pyramides [ROS 80] [CAN 85] [DYE 82] [TAN 83]. Les étages sont des tableaux de processeurs de dimensions décroissantes. Deux types de communication régissent les processeurs :

- une communication horizontale entre un processeur et ses voisins,

- une communication verticale entre le processeur et son père à l'étage supérieur, et entre le processeur et ses quatre fils à l'étage inférieur.

Les étages peuvent fonctionner de manière indépendante (MIMD) ou synchrone (pipeline). De plus, tous les processeurs d'un étage peuvent réaliser les mêmes traitements sur des données différentes (SIMD).

Les mécanismes de synchronisation, d'entrée/sortie et de contrôle des étages et des processeurs dans les étages constituent les facteurs prépondérants dans les choix des concepteurs de telles architectures.

Nous reviendrons plus loin sur l'architecture de la machine pyramidale SPHINX en cours de développement à l'ETCA, pour laquelle nous avons étudié un ensemble d'algorithmes et proposé un certain nombre de structures de contrôle.

3. L'APPORT DU LOGICIEL ET DU GENIE LOGICIEL

Le développement de programmes de traitement d'images nécessite l'assistance d'environnements de programmation adaptés fournissant des outils de mise au point et d'optimisation de la vitesse d'exécution.

Ce point de vue est sans doute commun à tout le domaine du génie logiciel, il a cependant une incidence particulière sur le traitement d'images à cause de la nature expérimentale des algorithmes (souvent

assujettis à des modifications) et de la diversité des utilisateurs et des machines.

L'assistance peut être réalisée sous forme d'aide à deux niveaux:

- interface utilisateur,
- développement de programmes.

3.1. L'interface utilisateur

L'interface utilisateur facilite les premiers contacts de l'utilisateur avec la machine et fournit une aide constante à toutes les étapes du travail. En effet, dans ce domaine, on fait souvent de l'expérimentation de fonctions sur de nouvelles images. Que l'expérimentateur soit utilisateur ou programmeur, l'existence de moyens interactifs réduit considérablement le temps d'apprentissage du système et de mise au point des programmes.

Un système interactif doit être avant tout un système éducatif ; un nouvel utilisateur ne sachant au départ que peu de chose sur son fonctionnement doit être capable d'apprendre à s'en servir rapidement. Cela veut dire qu'à chaque niveau d'utilisation, il est nécessaire de pouvoir être aidé dans le choix d'une commande, dans l'interrogation sur la nature d'une variable, ou même dans la compréhension du fonctionnement général du système lui-même. Cela impose de disposer de deux types de commandes :

- commandes conversationnelles,
- commandes directes.

3.1.1. Les commandes conversationnelles

Dans ce cas, le système dirige le travail de l'utilisateur non spécialiste en lui posant des questions et lui proposant même des solutions. Les formes d'aide peuvent parfois dépasser l'assistance pour aller jusqu'à l'initiation. Par exemple, prenons le dialogue interactif suivant avec un système qui utilise une **pile** d'images [BRU 84] :

```
saisir % une image %
saisir % une image %
soustraire % les deux dernières images saisies%
offset % taille de l'écran pour l'affichage % = ? 128
afficher % l'image résultat %
```

Deux images sont digitalisées à partir d'une caméra vidéo et rangées dans la pile d'images (du système). Les images sont soustraites point par point et le résultat est remis sur la pile. La fonction "soustraire" demande à l'utilisateur un paramètre appelé "offset". Finalement, le résultat est sorti de la pile puis affiché.

On peut noter que dans la pile, il n'est pas nécessaire de ranger toutes les images mais uniquement des pointeurs sur les images qui peuvent être rangées en mémoire ou sur disque.

D'autre part, l'utilisation d'une pile signifie que l'utilisateur n'a pas besoin de préciser les noms des fichiers images. Le fait de lui demander la valeur 128 signifie qu'il n'a même pas besoin de se souvenir de la valeur de ce paramètre.

3.1.2. Les commandes directes

Ces commandes servent aux utilisateurs expérimentés et capables de demander explicitement de l'aide quand cela devient nécessaire. La phase précédente n'est pas appropriée pour des tâches complexes. Quand il y a plus de trois ou quatre images, il devient difficile de revenir dessus et de les manipuler dans une pile. Une approche plus directe est préférée dans laquelle les images sont données comme des arguments des commandes.

Le même problème sera soulevé quand l'interrogation se fera sur une séquence de commandes encapsulées dans une macrocommande. A ce niveau, nous pensons que le système sera dans sa phase évolutive, dans laquelle il ne sera pas nécessaire de poser trop de questions.

Pour résumer cette action du système interactif, nous dirons qu'il est nécessaire en traitement d'images de disposer d'un système coopératif guidant le choix des valeurs pour les paramètres et acceptant les valeurs par défaut.

Par ailleurs, un système interactif doit être évolutif. L'extension du système peut servir à atteindre deux objectifs :

- l'adaptation du système initial à différentes applications. Chaque application enrichit le noyau de base par des structures de données adaptées au domaine et par des fonctions de traitement. Les mécanismes de base du système restent les mêmes pour interroger, aider et initier, seule la substance change.

- l'enrichissement du noyau de base par des commandes personnelles. Si toutes les commandes sont indépendantes les unes des autres, cette extension sera facile. Cependant, comme les fonctions ont souvent besoin d'interagir, une harmonisation de la communication des données entre elles devient obligatoire.

La version interactive du système SAPIN (SAPIN-NI) fonctionne selon ce principe. Tout comme les logiciels IPS [CHA 85] et LAMPION [SER 84], elle permet à l'utilisateur de construire interactivement des macro-commandes. L'interaction est nécessaire pour le choix des paramètres et leur initialisation.

3.2. L'aide au développement

La phase de développement peut être facilitée par l'usage d'un système coopératif et efficace qui procure un environnement adapté pour l'expérimentation.

L'étude de postes de travail puissants a été récemment à la base de plusieurs expérimentations sur la rapidité d'exécution de programmes tests dans le domaine. C'est par exemple le cas du poste de travail "Alto" chez Xerox PARC. Avec son grand espace d'adressage, son écran à haute résolution et surtout la possibilité de le connecter à un grand nombre de machines, il fournit des facilités de développement importantes. L'utilisation est graphique et rapide.

Des versions d'Interlisp [TEI 78] et de Smaltalk ont été implantées sur cette machine. Elles fournissent l'une et l'autre des environnements de programmation intéressants procurant des outils pour spécifier les objets manipulés, programmer, vérifier la correction des programmes d'une manière relativement coordonnée. La documentation en ligne et l'aide fournie à tous les niveaux de la programmation rendent les phases de développement et d'expérimentation courtes, attrayantes et surtout performantes.

L'utilisation d'un compilateur incrémental [RIS 70] permet de compiler une partie du programme ou l'ensemble. Si le programme est bien structuré et comporte plusieurs modules, il est possible de ne compiler qu'un module, celui qui a subi les dernières modifications. Des environnements de type LISP, PROLOG ou FORTH comportent de telles facilités.

L'aide au développemnt peut être fournie à trois niveaux du système:

- analyse du problème,
- développement de programmes,
- expérimentation.

3.2.1. L'analyse de problèmes

Afin de construire un programme lisible, correct et efficace, il est admis qu'il faut d'abord bien analyser le problème posé. Il s'agit, à partir d'un énoncé donné, de le spécifier formellement et de donner une définition assez "précise" de son contenu. L'énoncé ainsi spécifié, est exprimé sous la forme d'opérations dont on connaît, pour chacune d'entre elles, la forme et les propriétés de ses résultats.

Les avantages de la spécification sont importants dans l'analyse de problèmes [FIN 79]. Parmi eux, le plus marquant est celui relatif à la **structuration des objets**.

En traitement d'images, les objets ne sont pas clairement définis, leurs relations ne sont pas décrites de manière compréhensible, ce qui conduit le plus souvent dans les programmes à des définitions redondantes voire contradictoires. Ils sont souvent traduits brutalement dans les structures de données des langages de programmation, permettant peut-être une implantation rapide mais évitant en tout cas toute forme d'évolutivité possible.

La structuration des données permet de rétablir clairement les relations entre les objets et conduit à leur trouver des strutures de données les plus proches de leurs définitions.

Une de nos premières études dans SAPIN a été de préciser les objets qui interviennent en mettant en évidence les opérations utilisables. Ceci a conduit naturellement à la notion de **types abstraits** [GUT 77] [FIN 78] [LIS 75].

Pour décrire ces types, nous avons choisi d'adopter des définitions hiérarchiques faisant appel à des "constructeurs de types" prédéfinis tels que "table", "ensemble", "suite", etc... C'est ce que nous montrerons de manière détaillée dans le chapitre 2.

Des outils d'aide à la spécification commencent à se développer en génie logiciel. Dans notre collaboration avec l'équipe de programmation du CRIN, nous avons utilisé l'outil d'aide à la spécification SACSO [DUB 86].

Contrairement aux outils de spécification qui sont peu nombreux, plusieurs langages de types existent de nos jours introduisant la notion de type **concret**, de type **abstrait**, de **module** ou plus généralement d'**objet**. Nous donnons à la fin du deuxième chapitre un exemple d'implantation du noyau d'objets image spécifiés dans SAPIN en ADA [ICH 79].

3.2.2. Le développement de programmes

La phase de développement peut être assistée par les outils de mise au point des programmes comme l'éditeur, le compilateur, le debugger,

les contrôles et les aides (ou déductions) systématiques.

L'outil essentiel de développement de programmes reste le langage. La prépondérance de FORTRAN sur les machines conventionnelles a limité au début le choix du langage pour le traitement d'image. La programmation du traitement d'images peut être bien sûr faite en FORTRAN puisque FORTRAN est un langage qui manipule aisément les tableaux par indexation de zones mémoires [PRE 81]. En effet, pendant des années, l'influence de ce langage pour le calcul scientifique a été importante sur les machines séquentielles. Un grand nombre d'opérations image établies dans [PRE 80], [PRE 83] montre qu'il n'y avait pas d'autres langages capables d'inclure à un niveau primitif, toutes les opérations nécessaires à la manipulation des images comme les analyses spectrales, les filtrages, les tranformations spatiales, les entrées-sorties, etc...

Ainsi, il y a quelques années encore, pour la plupart des utilisateurs, un langage de traitement d'images était un ensemble de sous-programmes FORTRAN comme PAX [JON 70], SLIP [YOK 76], SPIDER [ROS 84], KANDIDATS [HAR 78] et INRIMAGE [CHI 78], rangés sous la forme de bibliothèques et appelés depuis un programme. Certains de ces sous-programmes simulaient le fonctionnement de machines parallèles sur les machines séquentielles. De cette façon, plusieurs constructions étaient réalisées sans aucune considération sur les problèmes matériels ou sur les temps d'exécution.

Ensuite, devant la poussée de nouvelles machines et l'évolution des langages de haut niveau (de FORTRAN à PASCAL d'abord, puis de PASCAL à ADA ensuite), d'autres choix se sont faits. Des extensions typiques d'ALGOL et PASCAL ont été proposées comme dans PASCAL-PL [UHR 81], L [RAD 81], PIXAL [LEV 81] et [REE 84], et plus récemment des extensions de C comme le PCL de Di Gesu' [DIG 86] et même l'utilisation de LISP [DEU 80].

L'idée est, tout en bénéficiant des possibilités du langage hôte pour les entrées-sorties, le calcul arithmétique et logique et la gestion des ressources, de définir son propre contexte de programmation en introduisant ses propres objets et opérations.

Plusieurs langages ont été enrichis pour fonctionner sur des machines spécialisées comme GLYPNIR sur ILLIAC IV [LAW 75] [MIL 73], STARAN HLL sur STARAN [LAN 76], PFOR sur PEPE [EVE 73]. Les deux premiers langages sont des extensions d'ALGOL, le dernier est une extension de FORTRAN.

Très peu de machines spécialisées ont été construites avec un langage de programmation propre, sans doute à cause de la forte préoccupation des architectes des problèmes matériels. Comme le signale Levialdi dans [LEV 86], même pour la machine MPP (Massively Parallel Processor) [BAT 80] qui est le symbole de la haute intégration, le langage de programmation a été défini après la construction du processeur.

L'idée de l'enrichissement de langages existants nous semble intéressante et nous l'avons adoptée dans notre projet, car elle permet de réfléchir sur ce qui est essentiel pour le traitement des images. La nouvelle couche ajoutée au langage hôte, si elle a été conçue de manière

générale, peut s'adapter facilement dans une seconde étape à l'architecture en place. Des schémas de traduction peuvent être étudiés dans ce sens pour chaque type de machine.

Dans le même ordre d'idées, beaucoup d'efforts ont été fournis dans le développement de langages entièrement spécialisés. Comme dans les enrichissements de langages, les innovations principales viennent des **structures de contrôle**, introduites essentiellement dans le but de caractériser les traitements séquentiels et surtout les traitements parallèles. Un colloque sur les langages d'images [DUF 81] définissait il y a quelques années, les qualités expressives des langages spécialisés. Maggiolo-Schettini [MAG 81] les a résumées dans les points suivants :

1) la possibilité de définir des tableaux sur lesquels on peut opérer en parallèle. Dans PASCAL PL [UHR 81], par exemple, la déclaration d'un tableau parallèle est effectuée comme suit :

```
a : parallel array [1..5,1..10,1..12] of integer ;
```

cette déclaration spéciale autorise dans la suite toute action parallèle sur le tableau a. Une affectation parallèle est de la forme :

```
||set a := b + c
```

où b et c sont des tableaux de même type que a. Ces indications sur le parallélisme sont introduites pour pouvoir détecter facilement lors de l'analyse le code externe et autoriser la parallélisation effective des opérations.

2) la posibilité de sélectionner un sous-tableau pour un traitement partiel, comme le hachurage de zones d'intérêt, l'extraction d'objets préalablement localisés dans l'image, etc...

Dans L [RAD 81], on définit une fenêtre d'accès :

```
w := <20,10>
```

Après positionnement en un point fixe <x,y> de l'image a, par exemple, par la fonction :

```
POSITION (w , a , <x,y>)
```

elle peut être utilisée pour définir un traitement local :

```
c := a:w
```


On range dans le tableau c le contenu de la zone du tableau a délimitée par w.

3) la possibilité de définir un voisinage de points et de le comparer avec un masque.

Cette fonction de voisinage est très utile pour réaliser les traitments locaux de l'image comme les opérations de filtrage de type convolution, érosion, dilatation, zoom, etc... Aussi, il est important de définir les outils nécessaires pour la réaliser.

Différents langages comme PIXAL [LEV 81] introduisent le type **masque** ainsi que les opérations concomitantes.

4) l'utilité d'instructions parallèles avec un contrôle global.

Ceci rend le contrôle des opérations parallèles, par test global sur les données, possible sur un tableau. Le test est fait pour un ensemble de pixels vérifiant un prédicat donné. Ceci est possible dans PIXAL et ACTUS [PAU 78] :

```
if a:w <> 0 then a := 1 ;
```

On affecte la valeur 1 à tous les points du tableau a si tous les points de la fentêtre w de a sont différents de 0.

ou encore :

```
for all <i,j> in a do if a<i,j> = 0
                      then a<i,j> := 1;
```

La construction **for all** indique que le traitement est indépendant d'un point à l'autre. Plusieurs opérations ponctuelles d'images comme le seuillage, le coloriage, etc... sont de ce type. Cette instruction peut être remplacée par un test global dans PIXAL dont la syntaxe est la suivante :

```
par if A < v then A = 1 else A = 0 parend
```

La structure **par** instruction **parend** indique que l'instruction entre les mots clés affecte la valeur 1 simultanément à tous les points de A vérifiant le test A < v et 0 simultanément à tous les autres.

Tous ces langages sont simples, faciles à utiliser, mais ont des possibilités de programmation parallèles limitées. Ils sont généralement insuffisants pour la programmation de machines plus complexes comme les machines systoliques ou pyramidales qui nécessitent des structures de contrôle plus adaptées aux différents mouvements de données qu'elles

engendrent.

Nous donnons dans le chapitre 3 quelques instructions destinées à la programmation des mouvements de données dans la machine pyramidale SPHINX.

3.2.3. L'optimisation de programmes

L'objectif visé étant de réaliser un système effectif, il est nécessaire de se préoccuper aussi d'implantation. Dans cette étape, on propose des mécanismes qui permettent, à la compilation, de produire du code optimisé pour les machines cibles.

Le programme de traitement d'images est souvent écrit sous forme séquentielle sans une réelle optimisation, sans doute pour des raisons de lisibilité et surtout de portabilité. Pour l'exécuter sur une machine parallèle, il est toujours intéressant de pouvoir en optimiser le code et activer ainsi sa vitesse d'exécution [SCH 75].

Afin de ne pas bouleverser la structure initiale du programme, les techniques élaborées proposent de faire ces optimisations dans une phase de pré-exécution de manière transparente à l'utilisateur.

Plusieurs compilateurs actuels optimisent le code du programme par suppression de la redondance dans les opérations de calcul [GOL 71]. La détection des opérations parallèles est souvent réalisée à l'exécution du programme dans les machines de type SIMD et MISD ainsi que dans celles qui sont séquencées par les données.

Dans les machines multiprocesseurs, on cherche surtout à regrouper les instructions dépendantes d'un même ensemble de données et à en créer des tâches parallèles. La détection du parallélisme est faite dans une étape de pré-compilation.

SAPIN propose des solutions pour deux types d'architectures :

- l'architecture multiprocesseurs où la programmation est souvent réalisée par l'appel à des fonctions cablées. La machine étudiée est la machine ICOTECH [TED 86], mais cela reste possible sur toute machine de ce type. La tâche du pré-compilateur sera de chercher les opérations globales indépendantes et de vérifier l'existence de processeurs matériels capables de les réaliser.

- l'architecture pyramidale qui comporte une connexion hiérachisée de processeurs [ROS 80]. La machine étudiée est la machine SPHINX [MER 83]. La programmation directe de cette machine est difficile et peut conduire à des exécutions peu efficaces. La solution adoptée dans SAPIN consiste à analyser des sous-programmes et à traduire ceux qui présentent des "indications de fonctionnement pyramidal" dans des schémas d'exécution standards pour la pyramide.

4. CONCLUSION

Nous avons montré dans ce chapitre la nécessité d'outils matériels et logiciels pour le traitement d'images dans le but d'avoir un meilleur rendement dans les domaines où il est utilisé. Les outils logiciels sont exigibles à deux niveaux :

- pour l'utilisateur, il faut un ensemble d'utilitaires hautement interactifs pour l'initiation et l'apprentissage progressif du système,

- pour le programmeur, un ensemble d'outils permettant le développement rapide des programmes d'une part et l'utilisation efficace des architectures spécialisées d'autre part.

Ce chapitre rend compte des progrès matériels réalisés et présente ce qui est commun et ce qui diffère du génie logiciel pris hors du contexte du traitement d'images. Les choix pris dans SAPIN sont essentiellement fondés sur l'aide à l'utilisateur et sur le développement d'outils de programmation indépendants de toute structure matérielle.

CHAPITRE 2 : DEFINITION DES TYPES IMAGE

1. INTRODUCTION

L'objectif de cette deuxième partie est de décrire de manière plus formelle les objets image les plus courants et les opérations qui les utilisent. Cela conduira naturellement à la définition structurée d'un noyau de types image et à la classification des algorithmes employés.

Dans plusieurs problèmes posés par la vision par ordinateur, les objets qui apparaissent sont "abstraits" au sens où ils expriment des propriétés logiques d'une scène. On manipule à ce niveau des surfaces, des régions, des frontières entre régions, et ces objets ne sont pas toujours exprimables en termes de structures de données d'un langage de programmation classique.

Pour toutes ces raisons, la programmation dans ce domaine doit avant tout permettre :

- (1) de se doter de critères objectifs permettant de caractériser les qualités des objets à étudier. Il serait souhaitable en ce sens d'aller au-delà de la forme syntaxique attendue et pouvoir exprimer les accès, les transformations mais aussi le comportement de ces objets vis à vis des erreurs de manipulation [FIN 79].

- (2) de représenter les objets image en termes d'autres objets plus proches de ceux qui sont utilisables dans les langages de programmation. Cela permet d'établir des relations logiques sur les données à partir des relations connues sur les types classiques des langages et d'unifier leurs représentations, ce qui conduit le plus souvent à des décompositions d'objets en des structures et opérations élémentaires.

De surcroît, cela permet d'effectuer rapidement la transformation du programme pour sa compilation dans un langage classique. Des idées de transformation de programmes ont été avancées dans [GAU 76].

- (3) les objets image ont un caractère dynamique. Ils évoluent au cours de la session par transformation de leur contenu et parfois même de leur forme. La spécification des types image doit permettre de décrire ces transformations et de définir de manière précise les relations entre objets de différents types. Elle doit aussi prévenir les erreurs et indiquer par conséquent les limites de chaque transformation.

2. DEFINITION DES OBJETS IMAGE

Parmi les objets les plus utilisés, nous trouvons :

image - masque - région - contour -
histogramme - fonction de transfert.

Nous allons donner dans la suite leur définition ainsi que la liste des principales opérations associées.

2.1. L'image

2.1.1. Définition générale

A l'échelle de l'observation visuelle, une image est une représentation bidimensionnelle d'objets tridimensionnels et de différentes natures (caractères, pièces industrielles, vues aériennes, images médicales, scènes naturelles, etc...). Elle contient en chaque point l'intensité lumineuse perçue par une caméra (ou tout autre système de mesure) en ce point. Elle rend compte de la variation du contraste dans la scène. L'intensité lumineuse varie de façon uniforme entre le blanc (beaucoup de réflexion de lumière, significative d'une forme non contrastée) et le noir (réflexion nulle de la lumière due à la présence d'un fort contraste). C'est donc une fonction de brillance continue définie sur un domaine borné.

Une **image multispectrale** est une image analysée dans différentes bandes spectrales de longueurs d'onde différentes. L'image comporte plusieurs images **monochromatiques** de la même scène prises dans différentes portions du spectre (couleurs). La mesure associée à chaque point est un vecteur spectral de n composantes correspondant chacune à une mesure différente.

Sous cette forme, l'image est inexploitable par machine. Il est nécessaire de la digitaliser. Cette opération, réalisée le plus souvent par l'appareil d'acquisition lui-même, se fait à deux niveaux. D'abord un échantillonnage du support de l'image la remplace par une grille de points dans un système d'axes, ensuite une quantification de l'intensité lumineuse est réalisée.

L'échantillonnage définit donc une grille de points discrets uniformément répartis. La disposition des points les uns par rapport aux autres est liée au type de tramage utilisé par le capteur. Le tramage

peut être carré, hexagonal ou rectangulaire. Selon le type de tramage, les points dans le même voisinage sont disposés suivant un carré, un hexagone ou un rectangle.

Dans l'étape de quantification, l'intensité lumineuse est convertie en un certain nombre de mesures numériques. Les valeurs obtenues, appelées **niveaux de gris**, sont codées par un intervalle d'entiers entre 0 et 2^q -1 où q est la taille de la représentation machine en nombre de bits. Quand les niveaux de gris sont réduits à deux, l'image est dite **binaire**.

Par ailleurs, le besoin de décrire le volume et le relief des objets, comme c'est souvent le cas en robotique, a conduit à définir une représentation tridimensionnelle pour les objets. Dans une image **tridimensionnelle**, chaque point représente la distance de l'observateur au point correspondant dans la scène.

2.1.2. Description topologique

L'image est considérée comme un ensemble fini d'un espace discret. Dans cet espace, on introduit une topologie permettant de décrire et d'analyser l'image de manière quantitative. On se limitera aux concepts topologiques les plus fréquents qui utilisent comme invariant fondamental l'ordre de connexion dans l'image.

Les concepts topologiques liés à la connexion sont : la notion de **boule**, la notion de **voisinage** et la notion de **connexité**. Les définitions de la métrique euclidienne ne s'appliquant pas de manière brutale sur l'image, nous allons d'abord donner une définition mathématique de ces notions topologiques dans le cas continu qui feront de l'image un espace métrique (E,d) où $E \neq \emptyset$ et d est une distance. Ensuite, nous verrons leurs définitions dans le cas discret [CHA 84].

A) Cas continu

a) Notion de boule

Une **boule** B de centre $a \in E$ et de rayon $\rho \in R^+$ est un sous-ensemble de E défini d'une des deux façons suivantes :

Une boule ouverte $B(a,\rho)$ est caractérisée par :

$$B(a,\rho) = \{p \in E \ / \ d(a,p) < \rho\}$$

Une boule **fermée** $\bar{B}$ est caractérisée par :

$$\bar{B}(a,\rho) = \{ p \in E \ / \ d(a,p) \leq \rho\}$$

b) Notion de voisinage

Soit $a \in E$, on appelle voisinage v de a, un ensemble $v \subset E$ tel qu'il existe $\rho \in R^{+*}$ avec $B(a,\rho) \subset v$. On note v(a) l'ensemble des voisinages de a :

$$v(a) = \{V \in P(E) \ / \ \exists \ \rho \in R^{+*}, \ B(c,\rho) \subset V\}$$

où P(E) est l'ensemble des parties de E.

Soit $A \subset E$. A est dit **ouvert** ssi $\forall \ a \in A, \ A \in v(a)$.

c) Notion de connexité

Un espace métrique (E,d) est dit connexe s'il n'existe pas de partition de E en deux ouverts.

B) Cas discret

Dans le domaine discret, on échantillonne le signal bidimensionnel sur une trame . Le choix de la trame influe sur la définition de la distance entre les plus proches voisins, ce qui va conditionner toutes les définitions topologiques qui en découlent. Nous allons montrer ces limitations dans le cas discret en nous restreignant à la **trame carrée** qui est la plus utilisée dans les systèmes de vision.

a) Notion de repère

Dans l'espace discret, une image est souvent liée à un repère de travail dans lequel on précise les coordonnées de l'origine de l'image et les intervalles de variation des indices x et y. Ces intervalles définissent ce qui est habituellement appelée **hauteur** et **largeur** de l'image et délimitent la zone d'adressage E des points. Le couple (position,mesure) est appelé **pixel** ou **pel**

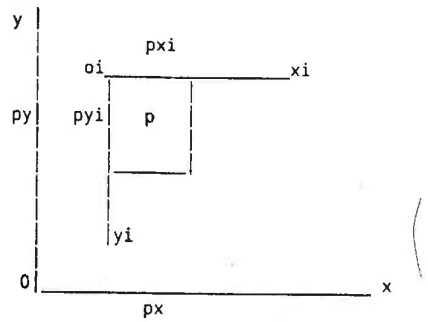


Figure 2.1 : Repère relatif (oi,yi,xi) associé à une image.

b) Distance entre deux points

Il est important de commencer par définir la distance constituant de base de toutes les autres définitions.

L'ensemble E des points de l'image est un sous-ensemble de $\mathbb{Z}^2$. Dans $\mathbb{Z}^2$, nous introduisons deux **métriques** différentes.

Soient p1 et p2 $\in \mathbb{Z}^2$ avec :

$p1 = (x1,y1)$ et $p2 = (x2,y2)$, alors

$d1(p1,p2) = \sup(|x1 - x2| , |y1 - y2|)$

$d2(p1,p2) = |x1 - x2| + |y1 - y2|$

On peut facilement vérifier que d1 et d2 sont effectivement des distances. Il existe bien sûr d'autres types de distance comme toutes les "chamfer-distances" [BOR 84] [DOR 86] que l'on peut définir sur E. Nous n'avons retenu que ces deux là car ce sont celles qui sont les plus utilisées dans le cas de la trame carrée.

La notion de distance permet de définir la notion de points voisins. On dira que **x est voisin de y** au sens de la distance d si et seulement si :

$$d(x,y) \leqq d0 \qquad \text{(où d0 est un seuil de distance)}$$

Pour un point $p \in \mathbb{Z}^2$ et d0 = 1, les points voisins au sens de d1 sont d'après la figure 2.2 ci-dessous les 8 points les plus proches autour de p dans les huit directions cardinales, tandis que l'ensemble des voisins de p au sens de d2 se limite aux 4 voisins dans les

directions horizontale et verticale.

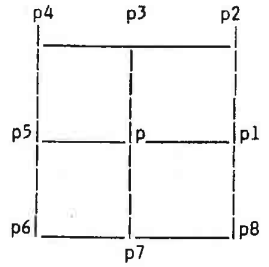


Figure 2.2 : Les 8 plus proches voisins du point p.

On appellera de ce fait, d1 la **8-distance** et d2 la **4-distance**. La différence entre les deux distances est importante, car elle précise la notion de proximité entre points et par conséquent la notion de connexité.

c) Notion de voisinage

De la définition précédente de la distance, on déduit immédiatement la notion de voisinage. Pour deux types de distance différents, on définit deux voisinages différents : le **4-voisinage** et le **8-voisinage**, comme le montre la figure 2.3. Le 4-voisinage comporte 4 points autour du point central considéré, et le 8-voisinage 8 points.

Figure 2.3 : Deux types de voisinages d'un point.

V4(p) (resp. V8(p)) est le plus petit voisinage de p pour la distance d2 (resp. pour d1) ; l'ensemble des voisinages de p pour d2 (resp. pour d1) est exactement l'ensemble des $V \subset E$ tels que $V4(p) \subset V$ (resp. V8(p)).

Les voisinages privés de leur point central seront notés $V4^*(p)$ et $V8^*(p)$.

d) Notion de masque

Le masque est une représentation discrète de la boule. On l'utilise pour représenter un voisinage de points dans une image et étudier la distribution locale des valeurs de l'image. Il sert aussi dans les opérations de filtrage pour détecter les contours. Dans ce cas, il contient des coefficients entiers permettant de signaler la présence de maximas locaux dans l'image. Il est aussi utilisé dans les méthodes d'appariement de modèle pour détecter un motif particulier dans l'image. Le masque contient une représentation discrète du motif. Le motif est recherché par déplacement du masque sur l'image et calcul de corrélation.

e) Notion de chemin

Soit E le domaine de variation des coordonnées de l'image. On définit dans E un **chemin** de longueur n composé de $(p1,p2,\ldots,pn) \in E$ tel que pi et pi+1 soient voisins au sens de la distance d. On dit que les pi sont connectés dans le chemin par la distance d.

Suivant que d = d1 ou d = d2, on définit un **4-chemin** ou un **8-chemin**. Dans le 4-chemin, les couples (pi,pi+1) sont des 4-voisins. Dans le 8-chemin, les couples sont des 8-voisins. La figure suivante illustre bien ces définitions.

4-chemin 8-chemin

Figure 2.4 : Deux exemples de chemins.

Ces définitions de la distance et des chemins conduisent naturellement à la définition de la connexité.

f) Notion de connexité

Soit $C \subset E$ un sous-ensemble de points de E. C sera dit 4-connexe (resp. 8-connexe) si $\forall$ pi,pj $\in$ C, $\exists$ un 4-chemin (resp. 8-chemin) de pi à pj formés de points de C. On notera pi $\asymp$ pj le 4-chemin (resp. 8-chemin) de pi à pj.

La relation $\asymp$ est une relation d'équivalence qui partitionne E en classes d'équivalence [LOU 80]. Une classe d'équivalence de E sera

appelée une **composante connexe** de E.

Il importe de faire attention à la distance utilisée dans l'étude de la connexité car la proximité des points n'est pas signe de connexion pour les deux distances. Ainsi dans la figure suivante, pour la 4-connexité, la courbe de gauche est complètement fermée et connexe. Elle isole parfaitement son extérieur de son intérieur. Ceci n'est pas le cas pour la courbe de droite qui, pour la 4-connexité, est considérée être formée par 8 composantes connexes [LOU 80].

1 seul objet 8 objets (pour la 4-connexité)

Figure 2.5 : Deux types de connexités.

g) Notion de points essentiels

Les notions précédentes liées à la connexité ont été définies sur tous les points de E sans distinction de formes dans l'image. Or souvent dans les images binaires surtout, nous sommes amenés à distinguer les points de la forme des points du fond.

Notons Q les points de la forme. Nous désignons par $\bar{Q}$ son complémentaire dans E. E, Q et $\bar{Q}$ sont liés par les relations suivantes :

$$E = Q \cup \bar{Q} \; ; \; Q = E - \bar{Q} \; ; \; \bar{Q} = E - Q \; ; \; Q \cap \bar{Q} = \emptyset$$

Il existe beaucoup de transformations qui font passer de Q à $\bar{Q}$ (opérations d'amincissement de formes épaisses ou d'approximation polygonale de contours de formes, par exemple). De cette manière, il convient d'établir un lien entre la topologie d'un objet et celle du fond de l'image [ROS 70].

Un point p de Q sera supprimé et donc ajouté à $\bar{Q}$ s'il est considéré inessentiel ou supprimable pour ces transformations.

Soit :

C4(Q) = nombre des composantes 4-connexes de Q,

C8($\bar{Q}$) = nombre des composantes 8-connexes de $\bar{Q}$

Un point $p \in Q$ est dit inessentiel pour Q s'il vérifie les deux conditions suivantes :

(1)- $C4[V8(p) \cap (Q \cup \{p\})] = C4(V8^*(p) \cap Q)$

(2)- $C8[V8(p) \cap (\bar{Q} \cup \{p\})] = C8(V8^*(p) \cap \bar{Q})$

La condition (1) signifie que la suppression de p dans Q ne doit pas changer l'ordre de dl-connexion de $V8(p) \cap Q$.

La condition (2) signifie que l'insertion de p dans $\bar{Q}$ ne doit pas changer l'ordre de d2-connexion de $V8^*(p) \cap Q$.

De manière similaire, un point p est dit 8-inessentiel s'il vérifie les deux conditions suivantes :

(1)- $C8[V8(p) \cap (Q \cup \{p\})] = C8(V8^*(p) \cap Q)$

(2)- $C4[V8(p) \cap (\bar{Q} \cup \{p\})] = C4(V8^*(p) \cap \bar{Q})$

Dans la figure suivante, le point p est 8-inessentiel mais non 4-inessentiel :

```
    *
  * p *
    *
```

Figure 2.6 : point 8-inessentiel et non 4-inessentiel.

De cette manière, on définit les points supprimables ne déconnectant pas leur 4-voisinage (resp. 8-voisinage).

Un point p est dit 4-supprimable s'il vérifie les conditions suivantes :

(1)- $V4^*(p) \cap Q \neq \emptyset$

(2)- $V8^*(p) \cap \bar{Q} \neq \emptyset$

(3)- $V4^*(p) \cap Q$ appartient à l'une des 4-composantes de $V8^* \cap E$.

Par symétrie, on définit les conditions pour les points 8-supprimable.

2.1.3. Description fonctionnelle

Dans ce cas, on représente l'image par une fonction à deux variables f(x,y) à valeur réelle dans l'intervalle de mesure [m1..m2].

Dans le cas de l'image monochromatique, la mesure f(x,y) est simple. Elle représente la valeur de la brillance au point (x,y). Dans le cas d'une image multispectrale, la mesure f est représentée par un vecteur de plusieurs composantes relatives à plusieurs bandes spectrales. Ainsi, pour une image couleur, f(x,y) est donnée par trois composantes $f_r(x,y)$, $f_v(x,y)$ et $f_b(x,y)$ correspondant respectivement à la valeur dans le rouge, le vert et le bleu.

Une phase importante du traitement d'images est celle qui consiste à faire passer f du cas continu au cas discret. Nous allons étudier dans la suite quelques notions fondamentales utilisées dans les transformations de f et étudier leur conversion du continu au discret.

A) Cas continu

a) Notion de gradient

Le gradient est un opérateur différentiel qui peut être défini en chaque point de l'image f par un vecteur à deux composantes :

$$\vec{\nabla} = (\delta f/\delta x \ , \ \delta f/\delta y)$$

Le gradient permet d'extraire les variations locales de niveaux de gris et par conséquent les points de contour.

L'image transformée f' sera donnée en chaque point, soit par la norme N du vecteur gradient, soit par la direction θ de ce même vecteur, soit par les deux. Un point de contour correspond à une valeur grande de N. θ est obtenu en calculant $\tan^{-1}(\nabla x/\nabla y)$. Ainsi, le gradient mesure la variation de l'intensité lumineuse le long des axes x et y.

b) Notion de laplacien

Le laplacien est une autre manière de calculer les variations locales de niveaux de gris. Il est donné par la somme des dérivées secondes en x et en y de f :

$$\frac{\delta^2 f}{\delta^2 x} + \frac{\delta^2 f}{\delta^2 y}$$

c) Notion de maximum local

On appelle **maximum local** l'ensemble des points (x,y) tels que : $\forall$ (x,y), il existe un voisinage v(x,y) tel que $\forall$ (x1,y1) $\in$ v(x,y) : f(x1,y1) $\leqq$ f(x,y). Dans le cas d'une image de niveaux de gris, les maxima locaux correspondant à des grandes valeurs du module du gradient de f(x,y), signalent la présence de contour.

De manière pratique, on exprime la recherche des maxima locaux comme suit : connaissant la propriété P des points à extraire, on cherche à transformer f en une fonction f' ayant une valeur grande aux points où la propriété P est bien vérifiée.

d) Notion de distribution

Une image peut être aussi considérée statistiquement comme une distribution particulière de la brillance. Dans ce modèle statistique, les mesures des pixels sont considérées comme la réalisation de variables aléatoires sousjacentes et les distributions correspondantes peuvent être estimées par les **histogrammes** de ces mesures. L'histogramme, quelque soit sa nature, locale, globale, mono ou multidimensionnelle est un outil puissant d'analyse de la distribution dans une image.

B) Cas discret

L'image n'est plus ici une fonction de $R^2 \rightarrow R$ mais plutôt de $Z^2 \rightarrow Z$. Cela pose un certain nombre de problèmes liés à l'échantillonnage et à la quantification mais aussi à l'approximation concomitante des fonctions de transformation.

Si l'on utilise le gradient, il faut fournir un vecteur dont chaque composante correspond à l'estimation du contour dans une direction donnée. Pour un voisinage de taille fixée dans le cas discret, le nombre de directions du contour se trouve limité. La discrétisation de l'espace des images limite par conséquent la richesse de ces opérateurs.

Pour le laplacien, le problème est plus difficile car il s'agit d'approcher la dérivée seconde. MARR [MAR 80] procède à un filtrage de l'image initiale avant d'appliquer un opérateur différentiel. Il évalue le laplacien sur l'image filtrée par un filtre gaussien.

Le laplacien présente un double inconvénient par rapport au gradient:

- l'information directionnelle si utile dans le gradient est absente,

- le laplacien, étant une approximation de la dérivée seconde, accentue le bruit dans l'image [BAL 82]. Son utilisation judicieuse nécessite de calculer localement ou globalement le gradient pour ne retenir que les passages par zéro significatifs du laplacien et filtrer le bruit inhérent à cette méthode.

2.1.4. Modes d'accès

Pour simplifier la représentaton de l'image, nous allons toujours la considérer dans son repère relatif avec une origine (0,0).

Parmi les modes d'accès les plus courants, nous trouvons :

- l'accès direct,
- l'accès indirect.

A) l'accès direct

On accède directement à un pixel de l'image en précisant ses coordonnées. I[x,y] permet d'accéder au pixel p de l'image I de coordonnées x et y dans le repère relatif de cette image. Peu importe la représentation choisie pour l'image (tableau, ensemble, ou liste de listes), cette notation a toujours un sens topographique.

B) l'accès indirect

L'accès indirect concerne l'accès **associatif** et l'accès **relatif** à un pixel.

- L'accès associatif permet d'accéder à un pixel à partir de sa valeur. Plusieurs applications en traitement d'images utilisent ce type d'accès. **premier** (I,25) permet d'accéder au premier pixel dans l'image I dont la valeur est égale à 25 ; cela suppose que l'on connaît le sens de parcours de l'image et par conséquent une relation d'ordre sur les pixels. Le sens de parcours correspond classiquement au sens de balayage physique.

- L'accès relatif permet d'accéder à un pixel à partir d'un des pixels voisins. Ce mode d'accès est souvent employé dans les algorithmes séquentiels en vue de lire la valeur du voisin ou de se déplacer de la position courante à une position voisine. Tous les pixels ont le même type de voisinage (comme nous l'avons déjà vu) et par conséquent le même nombre de voisins sauf pour ceux qui sont situés aux bords.

2.1.5. Opérations

L'image connaît un grand nombre d'opérations et transformations. Il existe plusieurs manières de classer ces opérations. La manière la plus conventionnelle d'après LIEN [LIE 77] est de les regrouper dans deux familles : **générique** et **assemblage**.

A) Les opérations génériques

Les opérations génériques sont des fonctions de transformation des valeurs des pixels en vue d'obtenir :

- une autre image,

- une donnée caractéristique de l'image traitée. Cette donnée peut être simple ou structurée (représentée par un objet d'un autre type) suivant la transformation appliquée.

- une description mathématique dans un but de reconnaissance des formes. La forme de la description dépend de l'approche utilisée et n'est pas facile à modéliser dans le cas général.

De manière historique, nous pouvons distinguer trois classes d'opérations génériques :

- les opérations issues de la théorie des ensembles ou morphologie mathématique,
- les opérations issues de la théorie du traitement du signal,
- les opérations liées aux transformations graphiques.

a) Les opérations morphologiques

La description ensembliste de l'image appelée morphologie mathématique [SER 82], considère l'image comme un ensemble d'objets ou de sous-ensembles définis dans un espace donné. Les objets forment une **partition** de cet espace. Cette approche permet de prendre en compte (plus que les suivantes), les propriétés ensemblistes, géométriques, topologiques et statistiques de l'image. Les opérations de transformation sont introduites par l'intermédiaire de l'**élément structurant** qui est une représentation plus concrète de la **boule** définie précédemment.

α) Notion d'élément structurant

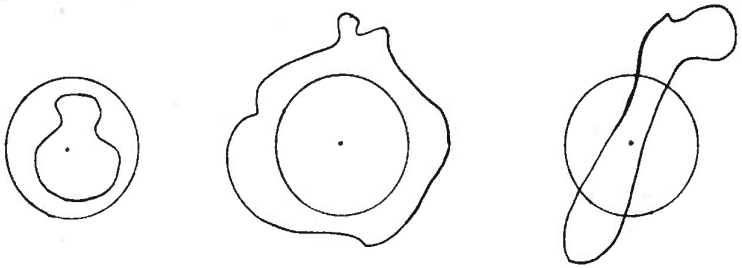
Un élément structurant est un ensemble qui sert à analyser localement l'image par l'intermédiaire de transformations simples. Il permet de mesurer en chaque point de l'image la correspondance de structure entre l'"objet géométrique" B qu'il représente et celui A du voisinage du point courant. En général, l'élément structurant est simple. Il a la forme d'un cercle ou d'un segment.

La nature de l'élément structurant est différente suivant qu'il s'agit d'une image binaire ou d'une image de niveaux de gris. Nous allons étudier le cas de la morphologie binaire pour laquelle l'élément structurant est plat.

β) Cas de la morphologie binaire

Une image binaire est composée d'un fond noté 0 et de la forme notée 1. Soit un élément structurant B centré successivement en chaque point de l'image, la morphologie binaire se pose trois types de questions :

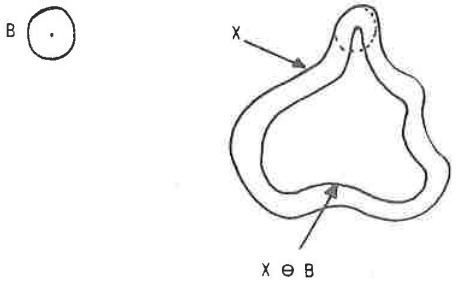
- est-il inclus dans la forme à analyser ?
- coupe-t-il sa frontière ?
- contient-il la forme à analyser ?



Notons Bx le translaté de B au point x de l'image et X un ensemble morphologique (la forme). Les opérations de base de la morphologie mathématique sont l'**érosion** et la **dilatation**.

On appelle érosion de X par B, l'ensemble :

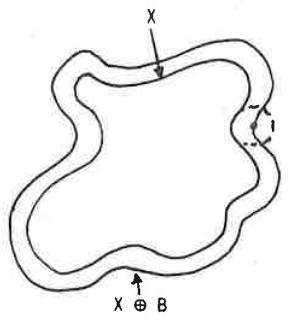
$$X \ominus B = \{x \in X \ ; \ Bx \subset X\}$$



On appelle dilatation de X par B :

$$X \oplus B = \{x \in X \; ; \; Bx \cap X \neq \emptyset\}$$

B ⊙



Ces opérateurs ont des propriétés intéressantes comme :

Dualité :

$$X^C \oplus B = (X \ominus B)^C$$

où X^C est le complémentaire de X dans l'espace de définition.

Composition d'éléments structurants :

$$X \oplus (B \cup B') = (X \oplus B) \cup (X \oplus B')$$
$$X \ominus (B \cup B') = (X \ominus B) \cup (X \ominus B')$$

Décomposition de gros éléments structurants :

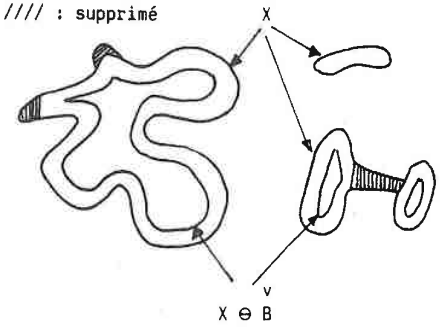
$$(X \oplus B) \oplus B' = X \oplus (B \oplus B')$$
$$(X \ominus B) \ominus B' = X \ominus (B \oplus B')$$

A partir des opérations d'érosion et de dilatation, on peut définir les opérations d'**ouverture** et de **fermeture**.

On appelle ouverture de X selon B, l'ensemble X_B défini par :

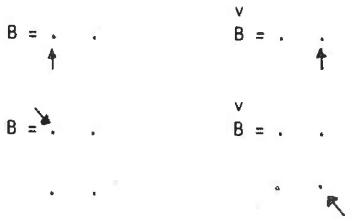
$$X_B = (X \ominus \overset{v}{B}) \oplus B$$

B . //// : supprimé



où $\overset{v}{B}$ est le symétrique de B par rapport au point central.

exemple



C'est une opération qui supprime les détails non significatifs de la forme et les déconnecte.

On appelle fermeture de X selon B, l'ensemble X^B défini par :

$$X^B = (X \oplus \overset{v}{B}) \ominus B$$

La fermeture élargit les isthmes et arrondit les formes en englobant plus. On peut facilement remarquer que l'ouverture et la fermeture sont des opérations duales. C'est la notion de dualité entre le fond et la forme.

Une autre propriété importante de l'ouverture et la fermeture est qu'elles sont idempotentes, ce qui en fait des filtres morphologiques

très puissants.

On définit aussi l'opération **tout ou rien**. Soit $B = (B^1, B^2)$ un élément structurant plan constitué de deux sous-ensembles B1 et B2. Soit X une image binaire ; l'opération "tout ou rien" de X selon B définit l'ensemble suivant :

$$X \otimes B = \{x \;/\; B^1_x \subset X \quad \text{et} \quad B^2_x \subset X^c\}$$

on peut remarquer que si B2 = ø, alors : $X \otimes B = \{x \;;\; Bx \subset X\} = X \ominus B$.

A partir des opérations précédentes, il devient facile de définir des opérations images liées à la détection de contour et à la squelettisation. En prenant commme élément structurant un carré approchant au mieux le disque unité (4-voisinage ou 8-voisinage dans la trame carrée), le **contour** de l'image sera donné par tous les points de X enlevés par l'érosion : $C(X) = X \;/\; X \ominus B$

L'**amincissement** d'une forme pleine consiste à la représenter par un squelette, ce qui correspond de manière grossière à sa courbe médiane. L'amincissement est exprimé par : $X \circ B = X \;/\; X \otimes B$

On définit l'**épaississement** d'une forme (opération duale de l'amincissement) par : $X \bullet B = X \cup (X \otimes B)$

D'après la dualité entre le fond et la forme, on peut écrire :

$$(X \circ (B^1, B^2))^c = X^c \circ (B^2, B^1)$$

On remarque que l'amincissement enlève des points à la frontière, alors que l'épaississement en rajoute. L'algorithme d'amincissement s'écrit :

Pour tout p **dans** X **faire**

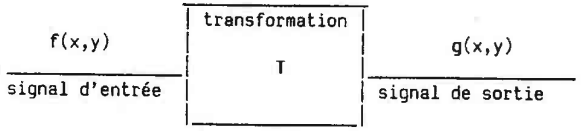
si voisinage = élément structurant **alors** p = 0 **fsi**

fpour

L'égalité ici est une égalité entre formes. Elle doit refléter la parfaite correspondance entre la forme dans le voisinage et celle de l'élément structurant.

b) Les opérations liées au traitement du signal

Si l'on considère qu'une image est un signal à deux dimensions f(x,y) avec des lois d'échantillonnage et de transformations physiques, toute transformation de f(x,y) sera définie comme l'application de l'ensemble des signaux d'entrée sur l'ensemble des signaux de sortie. Le système de transformation sera noté comme suit:



Un tel système de transformation peut s'écrire:

$$\mathbf{T(\, f(x,y)\,) = g(x,y)}$$

Parmi les opérateurs (ou systèmes) de transformation les plus usuels, on trouve les opérateurs linéaires, différentiels et fréquentiels.

a) Opérateurs linéaires

Un système de transformation est dit linéaire s'il obéit à une loi de superposition additive. Un tel système a la propriété suivante :

$$T(\lambda\, f_1(x,y) + \beta\, f_2(x,y)) = \lambda\, Tf_1(x,y) + \beta\, Tf_2(x,y)$$
$$= \lambda\, g_1(x,y) + \beta\, g_2(x,y)$$

où λ et β sont des coefficients.

Cette propriété dit simplement que si T est un opérateur linéaire, la réponse de la somme de deux signaux d'entrée est égale à la somme des deux réponses. Avec $f_2(x,y)=0$, l'équation précédente devient:

$$T[\lambda 1\, f_1(x,y)] = \lambda 1\, T(f_1(x,y))$$

Ceci reflète la propriété d'homogénéité qui dit que la réponse à multiple constant de n'importe quel signal d'entrée est égale à la réponse de ce signal multiplié par la même constante.

D'autres propriétés des opérateurs linéaires comme l'invariance en position sont données dans [CAS 77].

Nous donnons ci-après un exemple de transformation linéaire :

Exemple : **la convolution**

En représentant le signal continu sous la forme:

$$f(x,y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(t,s)\delta(x-t,y-s)\,dt\,ds$$

où $\delta(x-t,y-s)$ est la distribution de Dirac, on a par linéarité:

$$g(x,y) = T[f(x,y)] = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(t,s)\ T(\delta(x-t,y-s))dtds$$

En notant $h_t(x,y) = T(\delta(x,y))$ la réponse impulsionnelle de T, g(x,y) s'écrit :

$$g(x,y) = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(t,s)h_t(x-t,y-s)dtds$$

L'opérateur décrit par cette transformation est appelée **convolution** de $f \otimes h$. Dans le domaine discret, $f \otimes h$ s'écrit:

$$f \otimes h = \sum_{-\infty}^{+\infty}\sum_{-\infty}^{+\infty} f(t,s)h(x-t,y-s)$$

β) Les opérateurs différentiels

Les opérateurs différentiels les plus couramment utilisés sont le gradient et le laplacien. Le gradient est un vecteur dont chaque composante est une estimation du contour dans une direction donnée. On peut calculer la norme du gradient par les formules suivantes :

$$g(x,y) = \sqrt{a^2 + b^2} \quad \text{ou} \quad g(x,y) = |a| + |b|$$

avec :

$a = \delta f/\delta x = f(x+1,y)-f(x,y)$ et $b = \delta f/\delta y = f(x,y+1)-f(x,y)$

On peut utiliser des gradients plus complexes, dans le but de réduire la sensibilité du bruit. Duda [DUD 73] a proposé le gradient suivant sur un voisinage 3x3 :

$$a = (f(x+1,y+1) + 2f(x+1,y) + f(x+1,y-1)) - (f(x-1,y+1) + 2f(x-1,y) + f(x-1,y-1))$$

$$b = (f(x-1,y-1) + 2f(x,y-1) + f(x+1,y-1)) - (f(x-1,y+1) + 2f(x,y+1) + f(x+1,y+1))$$

Dans les deux cas a et b peuvent s'écrire comme :

$$\sum_{i=-1}^{+1}\sum_{j=-1}^{+1} f(x+i,y+j)h(u+i,v+j)$$

où :

$$ha = \begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \quad \text{et} \quad hb = \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix} \quad \text{et} \quad (u,v) = (2,2)$$

Aussi, devant la pauvreté liée à la forme initiale du gradient et la difficulté de trouver une bonne formule dans l'espace discret, beaucoup d'auteurs [FRA 75] [FRE 75] ont construit des fonctions h adaptées à la recherche d'un contour.

On peut aussi définir la dérivée seconde de f(x,y) dans les directions horizontale et verticale comme suit:

$$\Delta x = \frac{\delta^2 f(x,y)}{\delta x^2} \qquad \Delta y = \frac{\delta^2 f(x,y)}{\delta y^2}$$

Une première version du laplacien discret est donnée par:

$$\Delta x + \Delta y = f(x,y) - 1/4\ [f(x,y+1) + f(x,y-1) + f(x+1,y) + f(x-1,y)]$$

Pour le calcul effectif, le laplacien peut se mettre sous une forme matricielle et s'appliquer comme le gradient à travers un masque de coefficients.

D'autres problèmes en traitement d'images conduisent à étudier des voisinages de pixels pour signaler la présence ou non de motifs particuliers. Une des techniques employées est appelée **machine à masque** [CAS 77]. Elle est similaire à l'opération "tout ou rien" vue dans la classe des opérations morphologiques.

Soit m le masque recherché et f(x,y) l'image initiale. La détection du motif consiste à se déplacer sur l'image initiale et à rechercher pour quelle position, la distance entre le voisinage courant et le masque est minimale. Si la distance envisagée est la distance euclidienne, alors la distance d au point (x,y) sera égale à :

$$d(x,y) = \sqrt{\sum_u \sum_v (f(u,v) - m(u-x,v-y))^2}$$

Il importe de remarquer ici que les masques choisis sont des approximations en différence finie d'opérateurs différentiels extrêmement sensibles au bruit [ROS 62] [SHA 79]. En choisissant de meilleurs coefficients et en prenant en compte un plus grand voisinage (jusqu'à 5x5), on améliore notablement le lissage, par exemple. L'augmentation de la taille du voisinage conduit à une insensibilisation au bruit et permet souvent de mieux observer le contour [ROS 72] [HUE 73]. Toutefois, il est rare de prendre un voisinage supérieur à 7x7 pour les contours, d'abord parce que le contour se complique et son observation devient difficile au-delà d'un petit espace. D'autre part, il faut

calculer 49 opérations pour chaque pixel. Les convolutions les plus usuelles utilisent des masques 3x3, ce qui représente sur une image de 512x512 pixels environ 2,5 millions d'opérations ! D'où l'intérêt que présentent les machines parallèles.

γ) Les opérateurs fréquentiels

Dans plusieurs applications, il est utile de changer d'espace de représentation pour mieux y observer certaines caractéristiques des formes. Le domaine fréquentiel permet entre autres de mieux observer les variations de contraste par l'utilisation, par exemple, de la transformée de Fourier. Ainsi un filtrage passe-bas conduit à un lissage de l'image et a tendance à éliminer le bruit et à "tasser" les contours "abrupts". Un filtrage passe-haut, au contraire, renforce les bords et agit comme les méthodes de gradient.

c) Classes d'opérateurs génériques

Quelque soit la description mathématique de l'image que l'on choisit, les opérations génériques appartiennent à trois catégories:

- les opérations ponctuelles,
- les opérations locales,
- les opérations globales.

α) Les opérations ponctuelles

Soit l'opérateur T tel que $[T(f)](x,y)$ ne dépende que de $f(x,y)$, c'est-à-dire qu'il existe une fonction h d'une seule variable réelle telle que :

$$[T(f)](x,y) = h(f(x,y)) \quad \forall (x,y) \in E$$

Nous dirons que T est une fonction ponctuelle.

Une opération ponctuelle change les valeurs en un point sans changer la géométrie de l'image. Parmi les opérations ponctuelles sur l'image, notons l'intensification, l'atténuation, la multiplication ou l'addition de chaque point par une constante.

β) Les opérations locales

Dans les transformations locales, la nouvelle valeur du pixel est obtenue en fonction des valeurs de ses points voisins. Une transformation T est dite locale si elle s'écrit:

$$[T(f)](x,y) = h\{f(x+\delta x, y+\delta y) \text{ avec } \delta x \leq \Delta x \text{ et } \delta y \leq \Delta y\}$$

où δx et δy sont les variations de h dans le voisinage $[\Delta x, \Delta y]$ autour du pixel $p(x,y)$.

Le voisinage considéré est carré. De cette manière, il est défini par un masque contenant les coefficients de la fonction h.

Plusieurs formes de fonctions h existent en traitement d'images utilisant surtout les techniques de filtrage. L'application de h peut changer suivant que f est binaire ou non. Un cas particulier important est celui où g est binaire [PAV 82]. Il peut noter une caractéristique extraite au point (x,y) reflétant la présence ou l'absence d'une information particulière en ce point (présence d'un contour, d'un objet, etc...). Parmi les fonctions h utilisées, nous trouvons le lissage, la détection de contour et la détection d'objets ainsi que toutes les opérations de morphologie mathématique.

γ) Les opérations globales

Une transformation T est dite globale si on peut l'exprimer ainsi:

$$[T(f)](x,y) = h\{f(x+\delta x, y+\delta y) \text{ avec } \delta x, \delta y \in E\}$$

Au contraire de la transformation locale, h varie dans tout le domaine de définition des pixels, c'est-à-dire dans toute l'image. Les méthodes de filtrage fréquentiel sont des méthodes globales. La modification d'un point de l'image conduit à la modification de son spectre. En écrivant, par exemple, la transformée de Fourier sous une forme séparée comme suit dans le cas où M=N :

$$F(u,v) = \frac{1}{N} \sum_{x=0}^{N-1} e^{\frac{-2\pi jux}{N}} \sum_{y=0}^{N-1} e^{\frac{-2\pi jvy}{N}}$$

pour $u,v = 0,1,\ldots,N-1$, on peut observer que $F(u,v)$ peut être obtenue par deux étapes différentes de calcul en appliquant successivement la transformée de Fourier monodimensionnelle. On calcule d'abord $F(x,v)$ en faisant des transformations le long de chaque ligne ($v = 0,1,\ldots,N-1$) et en multipliant le résultat par N. Le résultat $F(u,v)$ est ensuite obtenu en faisant la somme de $F(x,v)$ le long de chaque colonne. Cela montre nettement que le résultat obtenu est de type complexe et qu'il dépend de tous les points de l'image. Il existe bien sûr une version plus rapide pour calculer la FFT mais cela ne change pas le caractère global de l'opérateur.

B) Les opérations d'assemblage

Les opérations d'assemblage agissent sur les surfaces des images sans affecter les valeurs de leurs pixels. Parmi les fonctions d'assemblage les plus courantes, nous trouvons le masquage, la césure et la jonction. Pour définir les deux premières, il est d'abord nécessaire d'introduire la notion de fenêtre.

Une fenêtre est un contour polygonal utilisé dans les images pour délimiter des zones compactes de pixels. La fenêtre la plus simple est le cadre. Lien a défini sur la fenêtre des opérations permettant d'élargir sa définition.

a) Les opérations d'union / intersection :

Ces deux fonctions ont été introduites pour exprimer l'union ou l'intersection de deux fenêtres. Ceci permet d'introduire des mécanismes de sélection de zones.

On définit l'union de deux fenêtres comme on définit l'union de deux ensembles. Soient F1 et F2 deux fenêtres et p un pixel dans l'image:

$$F1 \cup F2 = \{p \in F1 \text{ et/ou } p \in F2\}$$

La figure 2.7 montre un exemple de césure avec l'union de deux fenêtres.

De même, l'intersection de deux fenêtres est définie par :

$$F1 \cap F2 = \{p \in F1 \text{ et } p \in F2\}$$

L'intersection de fenêtres peut être vide si les fenêtres ne se recouvrent pas. Dans ce cas, la sélection de zone sera bien sûr sans effet.

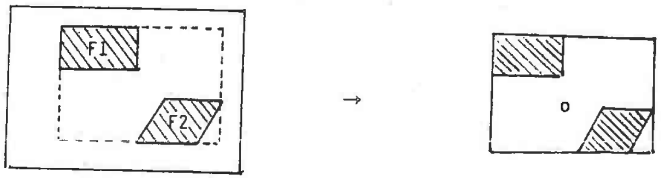


Figure 2.7 : Union de deux fenêtres.

b) L'opération de césure :

Césure est une fonction d'extraction de zones dans une image. Pour Lien, l'opération de césure se fait en deux étapes. La fenêtre qui est ici un polygone, est d'abord placée sur l'image donnée, ensuite une sous-image est extraite. Le cadre de l'image résultat est celui du cadre circonscrit à la fenêtre appliquée comme le montre la figure 2.8 ci-dessous. Tous les points en dehors de la fenêtre dans l'image résultat seront considérés comme appartenant au fond.

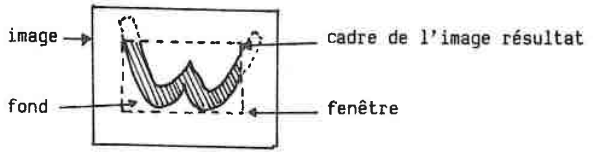


Figure 2.8 : Exemple de césure d'image.

c) L'opération de masquage :

Cette fonction permet d'obtenir une image par masquage d'une partie de l'image donnée désignée par un prédicat de sélection. Le prédicat introduit par Lien est une fenêtre rectangulaire. La fenêtre peut être inclusive (ouverte) ou exclusive (fermée). Si la fenêtre est ouverte, les pixels intérieurs à la fenêtre sont retenus et ceux extérieurs sont mis à zéro. Si la fenêtre est fermée, les pixels extérieurs sont retenus et les intérieurs sont mis à zéro. Dans tous les cas, l'image résultante a les mêmes dimensions que l'image donnée.

d) L'opération de jonction :

Jonction est une opération d'assemblage réunissant deux images. Les dimensions de l'image résultat doivent être assez grandes pour contenir les deux images données. La première image donnée est prise comme image de référence pour le calcul des dimensions. Si les deux images se recouvrent, c'est donc la première, normalement plus dominante, qui sera rangée dans l'image résultat. La figure 2.9 montre quelques exemples de jonction :

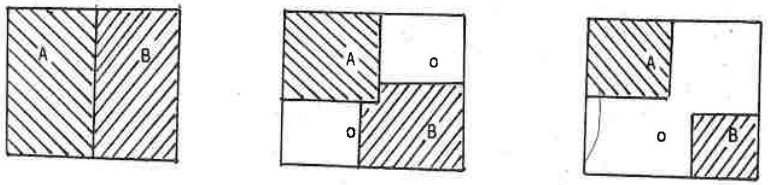


Figure 2.9 : Exemples de jonction de deux images A et B.
o désigne le fond qui n'est pas couvert par la jonction.

2.2. Le masque

2.2.1. Définition

Comme nous venons de le voir dans le paragraphe précédent, le masque est un concept très utilisé en analyse d'images tant pour la recherche d'un motif particulier appartenant ou non à la scène représentée (**modèle**), que pour étudier la distribution locale des mesures dans une image (**voisinage** ou **imagette**), ou encore pour réaliser un filtrage sur l'image (**masque de convolution** ou **filtre morphologique**).

C'est aussi une fonction de $R^2 \rightarrow R$ dans le cas continu ou ($Z^2 \rightarrow Z$ dans le cas discret). On le représente aussi par le couple (E',d) où E' est un espace métrique ($E' \neq \emptyset$) tel que card(E') << card(E) et d est la même distance que celle définie sur E pour l'image.

2.2.2. Opérations

Nous allons simplement résumer ici les opérations déjà vues sur l'image où intervient le masque. Nous pouvons regrouper ces opérations dans trois grandes familles :

- opérations de comparaison,
- opérations de filtrage,
- transformations géométriques.

A) Opérations de comparaison

On les appelle aussi opérations de mise en correspondance ou de corrélation. On cherche à calculer à partir d'un voisinage et d'un masque de même taille représentant un motif, une distance mesurant la similarité entre la forme et le modèle. Nous avons montré dans le paragraphe 2.1.2 un exemple de calcul de cette similarité. Il est évident que cette comparaison repose uniquement sur des notions de métrique locale et qu'il faut choisir, pour le motif recherché, la métrique qui peut conserver une bonne immunité aux distorsions.

Une simplification importante des techniques de corrélation consiste à remplacer les multiplications par des opérateurs logiques ET et OU. Ces opérateurs, appliqués surtout sur des images binaires, utilisent un masque binaire connu sous le nom de "peephole mask" [BAL 82]. Des opérateurs booléens plus complexes permettent de mieux caractériser la forme en détectant à la fois la présence des points de la forme à

certains endroits et ceux du fond à d'autres. L'opérateur morphologique "tout ou rien", fondé sur ce principe, utilise deux ET.

B) Opérations de filtrage

Les opérations de filtrage consistent à transformer la valeur d'un point pour mieux faire ressortir une propriété. Ainsi par des opérations de convolution, on peut détecter les points appartenant au contour dans une image. L'opération de convolution s'écrit dans le cas discret :

$$g(x,y) = \sum_{\Delta x,\Delta y} f(x+i,y+j).m(i,j)$$

où $f(x+i,y+j)$ avec $i \in \Delta x$ et $j \in \Delta y$ est le voisinage de (x,y) dans l'image et m est le masque. Il s'agit d'une somme de produits terme à terme du voisinage et du masque de convolution.

Tous les filtres morphologiques de type érosion, dilatation, tout ou rien fonctionnent selon ce même principe. Dans les images binaires, la valeur du point est forcée à 0 si la réponse du filtre morphologique est vraie. Nous pouvons voir dans [STE 82] des exemples de ces filtres. Les opérations élémentaires sont des opérations logiques de type ET, OU dans le cas d'images binaires, et des opérations arithmétiques simples de type addition et soustraction dans le cas d'images de niveaux de gris.

C) Transformations géométriques

Ces transformations permettent de modifier la disposition des coefficients du masque sans toucher à sa taille. Comme nous l'avons vu précédemment, dans la recherche du contour, la disposition des coefficients correspond à une direction de recherche. Modifier la disposition des coefficients conduit parfois à changer cette direction. Des exemples de transformation de masque consistent à faire subir au masque une rotation autour de son centre d'un angle multiple de 45°.

2.3. La région

2.3.1. Définition

La notion de **région** intervient dans la phase d'analyse d'une image qu'il est parfois nécessaire de décomposer en parties. L'extraction des régions permet de les situer les unes par rapport aux autres dans l'image, de parler de sous-régions, de régions englobantes, de régions voisines, de dissocier les régions du fond des régions de la forme. Elle sert aussi comme moyen de représentation condensée de l'image sous forme de primitives.

De manière formelle, on appelle région R, un ensemble de pixels d'une image vérifiant une propriété P. Cette propriété peut être d'ordre physique (même intensité), géométrique (même description), topologique (appartenance à une même cavité, au fond, à la même partie de la forme), statistique (mêmes indices de texture), etc... Une propriété physique simple est celle qui sélectionne les points d'un niveau de gris donné.

On définit alors une région R à l'aide des deux conditions suivantes:

(C1) : tous les points de R vérifient la propriété P,
(C2) : tout couple de pixels de R est relié par un chemin.

La deuxième condition est générale à toutes les opérations de regroupement. C'est elle qui conduit à se pencher sur les problèmes de frontières entre régions, de connexité et d'uniformité de régions.

Le résultat du regroupement est une **partition** de l'espace de définition E : $\Pi(E) = \{Ri\}$ telle que l'union des Ri est E et que les Ri sont deux à deux disjoints.

La propriété P définit une relation d'équivalence $\underline{R}$ sur l'image qui induit une autre relation d'équivalence en rapport avec la connexité des points. Ainsi, on dit que:

$\underline{R}$: $f(i,j)\ \underline{R}\ f(k,l)$ si $f(i,j)$ et $f(k,l)$ vérifient P.

$\underline{P}$: $f(i,j)\ \underline{P}\ f(k,l)$ si $f(i,j)$ et $f(k,l)$ peuvent être reliés par le même chemin.

On remarque :

$f(i,j)\ \underline{R}\ f(k,l) \Rightarrow f(i,j)\ \underline{P}\ f(k,l)$

2.3.2. Techniques de regroupement

L'algorithme de regroupement le plus classique est séquentiel. L'image est parcourue ligne par ligne. Chaque pixel est comparé à ses voisins ou prédécesseurs. S'il présente une certaine similitude avec ses voisins, il est ajouté à leur région, sinon il devient le premier pixel d'une nouvelle région. Dans ce cas, il est considéré comme point frontière entre ces deux régions.

De manière générale les techniques de regroupement peuvent être:

- **locales** : les pixels sont placés dans une région sur la base de leurs propriétés ou celles de leurs voisins les plus proches,

- **globales** : les pixels sont groupés en régions sur la base des propriétés d'homogénéité par fusion ou division de régions déjà construites.

Le principe de la fusion et de la division peut être formulé comme suit : se donnant un ensemble Rk (k=1,...,n) de régions vérifiant les conditions de regroupement (C1) et (C2), on définit une fonction booléenne H qui mesure l'homogénéité de chaque région Ri:

$$(C3) : \forall\ i\ H(Ri) = \text{vrai, et}$$
$$(C4) : \forall\ i \neq j\ H(Ri \cup Rj) = \text{faux}$$

Si (C3) n'est pas vérifiée pour une région Ri, cela signifie que la région Ri n'est pas homogène et doit par conséquent être partagée en sous-régions. Si la condition (C4) n'est pas satisfaite pour deux régions Ri et Rj, cela veut dire qu'elles forment ensemble une région homogène et doivent être dans ce cas fusionnées.

Un exemple classique de fusion et de division de région est donné par Horowitz et Pavlidis [HOR 74]. L'image est représentée sous la forme d'une pyramide où chaque étage contient une copie réduite de l'étage précédent.

Ayant défini la structure pyramidale Sp de l'image et la propriété d'homogénéité H, on examine les régions obtenues après regroupement. L'algorithme fonctionne en deux étapes :

première étape :

- si pour chaque région Rk dans Sp, H(Rk) = faux, on divise la région en 4,

- si pour tout groupement de 4 sous-régions Rk1, Rk2, Rk3, Rk4 on a H(Rk1 ∪ Rk2 ∪ Rk3 ∪ Rk4) = vrai, alors on les fusionne dans une seule région. L'algorithme s'arrête quand aucune opération de division puis fusion n'est encore possible.

deuxième étape :

- si pour tout couple de régions voisines Ri et Rj, on a H(Ri ∪ Rj) = vrai, alors on les fusionne.

2.3.3. Description des régions

Une fois identifiées, les régions sont décrites par un ensemble de données caractéristiques de leur forme. Les descripteurs choisis sont invariants par le changement de la taille, par la rotation, la translation et l'homothétie de telle manière qu'ils puissent servir au codage des régions pour la reconnaissance des formes. Il existe trois types de description des régions :

- description par les frontières,
- description par la topologie des formes représentées,
- descriptions par des mesures.

A) Desription par les frontières

Une des approches de description de la région consiste à extraire sa frontière et à la coder. Cela permet, sans prendre en compte les points intérieurs de la région, de disposer d'une description simplifiée et linéaire de la forme induisant facilement certaines propriétés géométriques liées à la surface telle, la compacité, la concavité, la finesse, etc...

Les points frontières peuvent être extraits directement par des techniques de filtrage déjà mentionnées au début de ce chapitre. Le "contour" de la région est donné par la liste de ses points ou de manière moins détaillée par un polygone dont les côtés sont des approximations locales de la frontière. Les opérations de fusion et de division consistent à allonger ou à raccourcir ces frontières comme le montre la figure suivante:

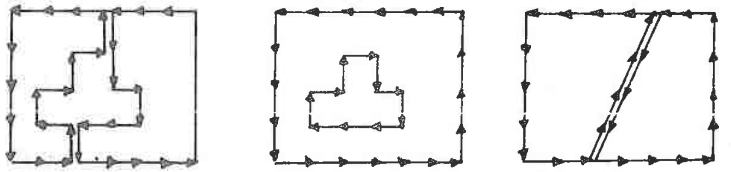


Figure 2.10 : Actions des algorithmes de regroupement sur les frontières des régions

La figure (a) représente deux régions voisines par leurs frontières. Les frontières sont représentées par des suites de vecteurs horizontaux et verticaux. La figure (b) montre le résultat de la fusion des deux régions de (a) en sommant les vecteurs frontières voisins (les vecteurs → et ← s'annulent). La figure (c) représente le résultat d'une division d'une région rectangulaire par une droite. Nous pouvons remarquer que les couples de vecteurs voisins sur cette frontière ont des sens opposés.

Cet exemple montre que malgré la simplicité du descripteur de région, on peut trouver des critères de fusion et de division bien fondés. On trouvera dans [BAL 82] des algorithmes de fusion et de division de régions par les frontières.

B) Description par la topologie

La description topologique de la région est une description globale de sa forme. Elle précise :

- le nombre de composantes connexes,
- le nombre de parties caractéristiques comme le nombre de trous, le nombre de formes concaves, convexes, etc...
- le nombre d'Euler.

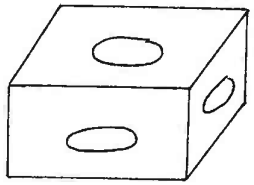
Soit C le nombre de composantes connexes de la forme F. Soit H le nombre de trous. On définit le nombre d'Euler comme : E = C - H. Les régions de la figure suivante ont un nombre d'Euler égal à 0 pour le A et -1 pour le B.

Figur 2.11 : Formes ayant un nombre d'Euler différent.

Dans le cas de frontières droites et où la forme peut contenir des faces (F), des trous (H), des arêtes (Q), des angles aux sommets (A), on peut établir la relation suivante entre toutes ses composantes appelée formule d'Euler:

$$E = C - H = A - Q + F$$

Le nombre d'Euler de l'image plane de l'objet à 3 dimensions de la figure suivante est égal à E = 6 - 9 + 1 = 1 - 3 = -2.



ai (i=1..6) = angles
ti (i=1..3) = trous
si (i=1..9) = arêtes

Figure 2.12 : Le nombre d'Euler pour une forme tridimensionnelle.

On peut trouver dans [SER 82] un algorithme efficace qui calcule cette relation pour des formes planes.

C) Description par des mesures

Une région peut être représentée par des mesures géométriques liées à sa surface, à son périmètre ou par des mesures liées à son irrégularité telle sa fermeture convexe. Ces mesures peuvent refléter immédiatement la similitude ou la dissimilarité entre régions dans la comparaison des formes.

Soit p le périmètre et s la surface d'une région, on peut par exemple définir une mesure de finesse F par :

$$F = 4\pi (s/p^2) \quad \text{(égale à 1 pour un cercle)}$$

La fermeture convexe H d'une forme peut être calculée comme la somme de sa surface S et de sa déficience convexe D :

$$H = S + D.$$

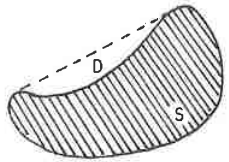


Figure 2.13 : Fermeture convexe d'une forme.

Afin de disposer d'une description fidèle des objets dans une image par le concept de région, nous la représentons par deux structures de données. La première est un **tableau** de labels (entiers) d'objets. Les labels ou numéros sont disposés dans le tableau de manière à décrire les surfaces des objets qu'ils désignent dans l'image. La deuxième est une **liste** d'informations sur les objets représentés, comme le cadre circonscrit, le premier point, la surface, etc... La mise à jour d'une région conduit immédiatement à la mise à jour de cette liste.

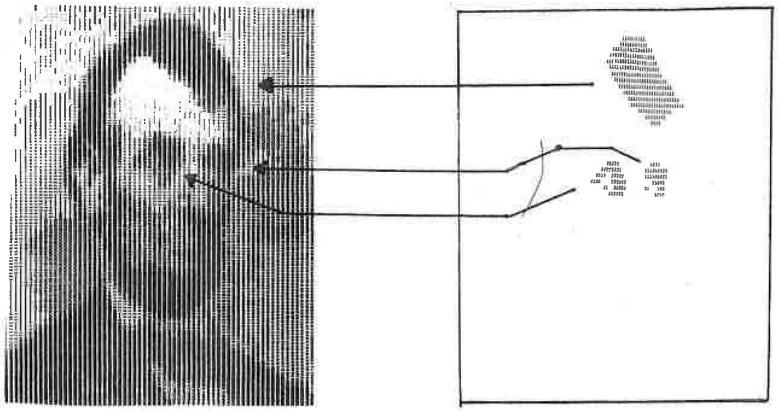


Figure 2.14 Représentation de régions dans une image.
(a) image originale,
(b) tableau des trois régions sélectionnées dans (a).

Cette représentation permet de dissocier sans les séparer, les pixels de l'image, leur emplacement dans la région et la propriété de la région qui les contient.

2.4. Le contour

2.4.1. Définition

On distingue les points de contour des lignes de contour.

A) Les points de contour

On définit intuitivement un point de contour comme un point de l'image appartenant à la frontière entre deux régions voisines. Il est de ce fait associé à la notion de contraste local résultant d'une variation rapide de l'intensité lumineuse en ce point.

De manière formelle, on définit les points de contour comme l'ensemble des points de l'image vérifiant une propriété P de contour. Suivant le type de l'image et suivant que l'on cherche une frontière entre deux régions différentes ou une ligne traversant la même région, P peut avoir différentes formes. Ces différentes formes de P ont engendré des classes d'opérateurs de détection de points de contours [ROS 76].

Dans les images de niveaux de gris, P corrrespond au maximum de la dérivée première de f(x,y). Les opérateurs de détection utilisés sont fondés sur l'idée de différence ou dérivée comme les gradients ou les laplaciens. Ils appartiennent à trois classes principales:

(1) la famille des opérateurs du gradient et du laplacien,

(2) les opérateurs de mise en correspondance utilisant des modèles d'éléments de contour dans différentes directions,

(3) les opérateurs qui détectent les variations locales d'intensité avec des modèles paramétriques.

Bien que différents, tous ces opérateurs présentent le point de contour de la même manière, par deux paramètres :

- la direction qui correspond à la direction du changement maximum de niveaux de gris,
- la valeur de ce changement maximum.

Une autre méthode appelée MAT (Median Axis Transform) appliquée souvent sur les images binaires, effectue un codage des points de l'image par des carrés de taille maximale, c'est-à-dire couvrant le maximum de forme autour de ce point et touchant la frontière. Chaque point est représenté par le "rayon" du carré dont il est le centre. Bien qu'adaptés à la détection du squelette (ensemble des centres des carrés maximaux), cette méthode permet de détecter les points de contour par recherche des points les plus éloignés des centres [BAL 82].

B) Les lignes de contour

Les lignes de contour correspondent aux frontières de régions. Elles séparent la scène d'un objet en deux ensembles complémentaires : l'intérieur de l'objet et son extérieur. Suivant le cas, les lignes de contour appartiennent à l'intérieur de l'objet ou à l'extérieur.

Cette idée de séparation conduit à définir une ligne de contour comme un ensemble de points de contour connexes. De cette manière, un contour sera obtenu à partir des points de contour par suivi de la frontière approximative de l'objet et bouchage des trous (où il y a absence de points de contours) pour préserver sa continuité.

La ligne de contour a donc une structure filiforme. Elle peut être fermée ou non. Elle peut comprendre des jonctions. Nous allons voir dans la suite quelques approches de suivi de contour dans deux types d'images.

a) Cas des images binaires

Le cas de suivi le plus simple est réalisé dans une image binaire. L'algorithme de suivi issu de [MAR 76] oeuvre comme suit :

- parcourir l'image jusqu'à rencontrer un point de l'objet,
- si c'est un point de l'objet, tourner à gauche et avancer de 1 pixel ; sinon tourner à droite et avancer de 1 pixel,
- s'arrêter à la rencontre du point de départ.

La figure suivante montre le chemin tracé par l'algorithme pour réaliser le suivi.

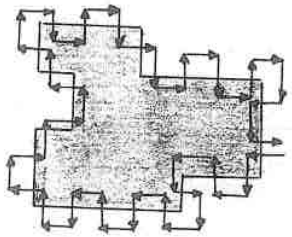


Figure 2.15 : Trace de l'algorithme de suivi de contour.

Cet algorithme a été d'abord introduit par De Letto et cablé puis formalisé par Rosenfeld sous forme d'algorithme BF. Il exige que l'objet soit 4-connexe et ne détecte donc que les angles droits. Il montre par ailleurs la nature séquentielle de ce type de problème.

b) Cas des images de niveaux de gris

Pour le suivi dans le cas des images de niveaux de gris, il faut d'abord trouver le premier point de contour appartenant à la frontière de l'objet et chercher ensuite les points suivants dans le sens des directions indiquées par le gradient.

Soit p un point de contour. On note par $|p|$ le module du gradient et par $\phi(p)$ sa direction. L'algorithme de suivi issu de [MAR 76] oeuvre comme suit :

1) balayer l'image jusqu'à la rencontre de l'objet O. Soit le point po. Passer au point p adjacent à po dans la direction perpendiculaire à $\phi(po)$. Appliquer l'opérateur de gradient sur p ; si $|p|>seuil$, alors p est ajouté au contour,

2) sinon, de manière locale dans le voisinage de p, chercher par une technique de seuillage simple, un autre point candidat.

KELLY [KEL 71] propose un algorithme de suivi de contour guidé par un plan de recherche. Il est similaire à celui de TANIMOTO et PAVLIDIS qui travaille sur des images réduites dans une pyramide.

- une image réduite est construite à partir de l'image initiale par des moyennes locales sur des voisinages disjoints de même taille $(2^K x 2^K)$,
- on détecte les points de contour dans l'image réduite par une des méthodes parallèles vues sur l'image. Le contour ainsi déterminé est un contour grossier où les trous ont été pour la plupart bouchés par la moyenne et la réduction,
- ce contour grossier sert dans la dernière étape à déterminer le contour exact. Kelly indique que pour un réduction de taille 2^K, le nouveau contour est cherché dans une bande de largeur 2^{K+1} centrée sur le contour grossier.

Il existe dans la littérature d'autres méthodes de suivi de contour [FIS 73] [CHI 74]. Elles obéissent toutes aux mêmes contraintes comme la rapidité du temps de calcul, la rapidité d'extraction dans les régions homogènes, la prise en compte du bruit et la prise en compte des différents types de contour.

Toutes les méthodes de suivi de contour sont séquentielles puisque chaque point est déterminé à partir du point précédent. De ce fait, la plupart des algorithmes sont réputés lents.

c) Cas des images multidimensionnelles

Dans les images multidimensionnelles, le suivi de contour est une opération plus complexe que dans le cas bidimensionnel. Bien que les points du contour soient définis de la même manière que dans le cas bidimensionnel, son interprétation est différente. Dans le cas tridimensionnel, par exemple, les éléments de contour sont des éléments de

surface séparant des volumes de différentes natures et couleurs. Le suivi de contour consiste à chaîner des éléments de surface dont le module du gradient est grand et dont l'orientation est la même. Ceci se généralise à des volumes dans le cas quadri-dimensionnel, etc...

2.4.2. Représentation

Suivant qu'il s'agisse de points de contour (appelés aussi indices visuels) ou de lignes de contour, on peut trouver deux représentations différentes :

● les points de contour sont représentés par un tableau qui peut être suivant les cas :

- l'image d'origine accompagnée d'un prédicat d'accès à ces points. Cette représentation permet de préserver les valeurs de niveaux de gris des points pour d'autres usages,

- l'image de contour où chaque point est donné par un couple de valeurs précisant le module et l'orientation du contour,

- l'image binaire des positions des points de contour. Un point est à 1 s'il contient un point de contour et 0 sinon. Cette image binaire peut constituer un prédicat de sélection des points de contour dans l'image d'origine.

● les lignes de contour sont représentées par des listes de valeurs dont le type peut varier en fonction de l'application :

- les lignes de contour sont représentées par des listes de points de contour. Chaque point est donné par ses coordonnées dans l'image d'origine et ses valeurs, si nécessaire,

- les lignes de contour sont codées par une liste de lignes polygonales (segments) approchant leur forme. Plusieurs algorithmes d'approximation polygonale existent dans la littérature. Cette codification du contour par des segments permet de représenter les lignes de contour par des primitives plus importantes que les points et de réaliser plus facilement les comparaisons de contours.

Quelque soit l'approximation utilisée, la structure de donnée contour est une structure dynamique qui se construit point par point ou primitive par primitive. Le contour peut être discontinu et représenté par plusieurs listes de sous-contours.

De manière à disposer d'informations globales sur le contour, il est nécessaire d'enrichir la représentation par des attributs relatifs à sa structure tels :

- nom de l'image d'origine,
- nombre de sous-contours,
- cadre circonscrit,
- longueur totale,
- linéarité (et direction générale, dans le cas où elle a un sens),
- homogénéité,
- compacité.

2.4.3. Opérations

Les opérations sur le contour se répartissent en quatre classes :

- opérations de consultation,
- opérations de construction,
- transformations géométriques,
- opérations de codage.

A) Opérations de consultation

Les opérations de consultation permettent de déterminer une mesure sur la forme du contour. Cette mesure peut correspondre à la valeur d'un attribut ou être calculée. Parmi ces mesures, nous pouvons indiquer la courbure d'une portion de contour, la surface engendrée s'il est fermé, le nombre de cycles, etc...

Les opérations de consultation sont aussi relatives à l'accès : accès à un point par ses coordonnées, à un point par son prédécesseur, à un point par son rang, à un point par son emplacement : en tête, en queue, au croisement de n contours, à l'intérieur d'une fenêtre d'observation, etc...

B) Opérations de construction

Parmi les opérations de construction usuelles, nous trouvons :

- la concaténation,
- le prolongement,
- la fermeture.

a) Concaténation de deux contours

Cette opération consiste à construire un contour en mettant deux contours bout à bout. Le premier contour sert de référence au prolongement par le deuxième qui subit des modifications de positions de points.

b) Prolongement de contours

La figure 2.16 montre un exemple de prolongement de deux contours C1 et C2. Le prolongement consiste à ajouter des points pour boucher l'espace entre deux ou plusieurs contours qui se suivent. Les méthodes de prolongement sont nombreuses. Elles n'ont de sens que si les contours sont prolongeables, c'est-à-dire que les espacements ont des tailles négligeables.

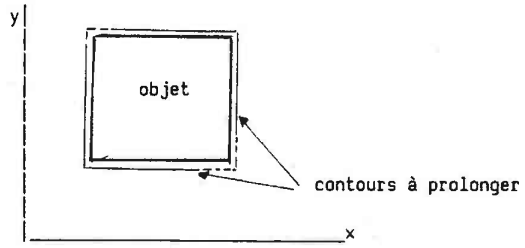


Figure 2.16 : Prolongement de deux contours.

Dans le cas où les contours sont représentés par des listes de points, la méthode de prolongement consiste à boucher les trous par des points très rapprochés ayant une valeur arbitraire. Souvent, on utilise les méthodes d'interpolation pour réaliser le bouchage.

c) Fermeture de contour

Cette opération est un cas particulier de la précédente. Elle consiste à essayer de fermer un contour en le prolongeant par les deux extrémités. Cette opération est utile dans les problèmes de délimitation de régions.

C) Transformations géométriques

Ces transformations sont relatives à la translation, à la rotation, autour d'un point, à l'homothétie et au changement de l'orientation générale du contour. Elles sont de nature ponctuelle et peuvent s'appliquer de manière parallèle bien que la nature de la forme soit mal conditionnée pour le parallélisme.

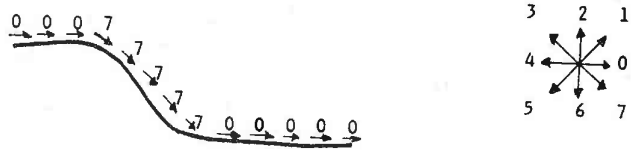
D) Opérations de codage

Nous avons indiqué dans la partie réservée à la représentation du contour, différents types de codage des contours comme le codage de Freeman, la MAT, la méthode des moindres carrés, ou encore le codage par des primitives structurelles. Toutes ces méthodes cherchent à alléger la

représentation des contours par les points tout en ajoutant des informations structurelles liées à la courbure et a` l'orientation utiles dans les phases d'analyse et de reconnaissance des formes.

a) Le codage par chaînage de contour

On suppose que la distance entre les couples de points consécutifs du contour est inférieur à $\sqrt{2}$. Le codage par chaînage de contour consiste à remplacer chacun de ces couples par la direction du vecteur composant codée par un chiffre entre 0 à 7 correspondant aux directions cardinales. Ce codage conduit souvent à réaliser des compressions par la mise en facteur des codes répétitifs comme le montre la figure 2.17 suivante :



Représentation : 0:3 7:5 0:5 codes

Figure 2.17 : Codage de contour par les directions de Freeman.

L'algorithme est séquentiel. Il consiste à parcourir les points un à un et à calculer à chaque point l'orientation du vecteur courant et de mettre à jour le nombre d'orientations. On peut l'écrire comme suit:

```
Function codage (c : contour) retourne tcontour;
Dec c' : tcontour;
(* c est le contour d'origine, c' contient le résultat du codage
de c *)
début
        lire p0; p = p0; nbo = 0; a0 = o;
        (* initialisation de la première direction et de son
        nombre *)
        jqa fin (c)
        faire
                p = succ(c,p0); o = orientation(p,p0);
                (* calcul de la nouvelle direction *)
                si o=a0 alors nbo := nbo+1
                (* poursuite de la même direction *)
                        sinon insert(c',<a0,nbo>);
                                a0 = o;
                                nbo = 1
                (* changement de direction, abondon de la direction
                précédente *)
                fsi
                p0 = p
        ftq
        retourne c'
fin
```

Cette méthode, bien que locale, conduit à des approximations polygonales du contour. Chaque segment du polygone est donné par son orientation et sa longueur. A cause de sa sensibilité aux petites variations locales, des méthodes plus globales sont utilisées. Nous pouvons trouver dans [BAL 82] d'autres techniques de codage fondées sur le même principe.

b) le codage par des courbes :

Ce problème est classique en reconnaissance des formes. Il permet de représenter une forme par une liste de codes ou primitives. Plusieurs applications en reconnaissance des caractères décrivent ces formes par des primitives structurelles de type arc de cercle (c) et segment de droite (s)

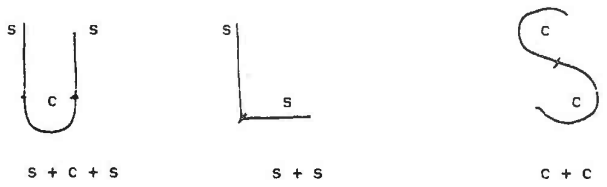


Figure 2.18 : Description structurelle des caractères.

2.5. L'histogramme

2.5.1. Définition

L'histogramme d'une image est une fonction de R → R qui à toute valeur ou mesure dans l'image, associe son nombre d'occurences. L'information contenue est relative à des comptes de valeurs. De ce fait, l'histogramme résume au mieux l'information globale que contient l'image et donc sa structure statistique [LOW 82].

La forme de l'histogramme est significative de la forme de la distribution des valeurs dans l'image. Elle donne des indications précises sur le caractère et la nature de l'image dont il est issue. Si l'histogramme est étroit, l'image a un faible contraste lumineux. Si l'histogramme est bi ou tri-modal, l'image peut être segmentée en deux ou trois régions de radiométries très différentes, etc...

2.5.2. Opérations

Plusieurs mesures statistiques de "premier ordre" telle que la moyenne, la variance, l'entropie sont obtenues sur l'image à partir de l'histogramme.

Une application intéressante consiste à égaliser un histogramme par regroupement des comptes. Pour cela, on transforme l'histogramme d'origine pour obtenir un histogramme cumulatif puis on regroupe les comptes tombant dans des intervalles équiprobables.

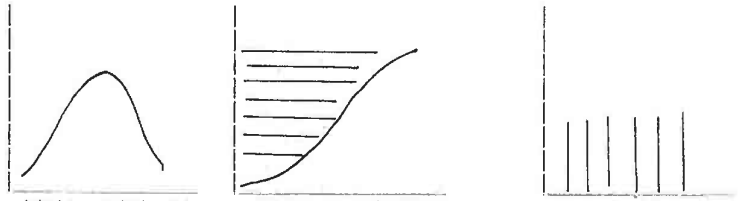


Figure 2.19 : Egalisation d'histogramme.

Si l'on transforme une image à l'aide de l'histogramme égalisé, ceci conduit à regrouper ses valeurs radiométriques, à augmenter

l'entropie locale et par conséquent à rehausser le contraste.

L'exploitation des valeurs de l'histogramme peut conduire, par ailleurs à extraire les différentes régions d'une image. Quand l'image contient un objet sur un fond (de faible niveau de gris), son histogramme est bimodal. La détection de la frontière entre ces deux modes fournit le seuil de séparation de l'objet de son fond. Soit s ce seuil. On dira qu'un pixel p est un point de l'objet si f(p)> s.

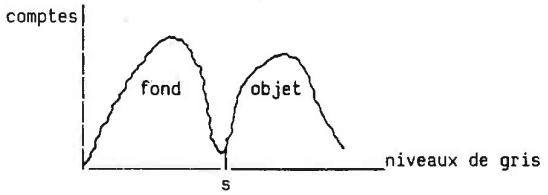


Figure 2.20 : Histogramme bimodal.

La recherche du seuil s n'est pas facile quand l'histogramme n'est pas lisse. Plusieurs techniques d'extraction de régions utilisent l'histogramme par observation de sa forme. La région est divisée en sous régions tant que son histogramme n'est pas étroit.

2.6. La fonction de transfert ou anamorphose

2.6.1. Définition

Une fonction de transfert est une table de conversion de couleurs dans une image. Soit D1 = [a,b] $\in$ R le domaine des couleurs à convertir et D2 = [c,d] $\in$ R le domaine des couleurs converties, on dit que la fonction de transfert est une fonction de D1 $\rightarrow$ D2.

Il existe deux types de fonction de transfert dans un système de traitement d'images. La première modifie l'affichage de l'image sans en modifier les valeurs en mémoire. La seconde agit sur les valeurs des pixels à la manière des fonctions ponctuelles et les modifie.

2.6.2. Différentes utilisations

Les fonctions de transfert agissent sur l'image en vue de réaliser différentes actions. Parmi elles, nous trouvons :

- la correction des valeurs (comme la correction de gamma, la correction de couleur et l'inversion de l'échelle de gris),

- le rehaussement du contraste (par l'utilisation par exemple de l'égalisation d'histogramme),

- l'augmentation du pas de quantification (par utilisation par exemple d'une fonction de transfert en marches d'escalier),

- le seuillage,

- l'affichage des images (comme l'affichage des images de niveaux de gris en fausses couleurs, par exemple).

A titre d'exemple, nous allons montrer comment le seuillage peut être réalisé par une fonction de transfert.

Le seuillage est une opération de sélection de points dont les valeurs appartiennent à la même plage de niveaux de gris. Suivant la valeur que l'on veut donner aux points sélectionnés et celle que l'on veut donner aux autres, le seuillage peut s'écrire de différentes manières :

une solution à ce problème est d'exprimer g(x,y) comme suit :

$$g(x,y) = \begin{cases} T(f(x,y)) & \text{si } f(x,y) \geq S \\ f(x,y) & \text{sinon} \end{cases}$$

où T est une fonction de transfert.

cela veut dire que tous les points de f(x,y) dont la valeur est supérieure au seuil S sont transformés par T(f(x,y)). Les autres points restent inchangés.

Une autre solution consiste à mettre à une valeur fixée v1 les valeurs transformées supérieures à S :

$$g(x,y) = \begin{cases} v1 & \text{si } f(x,y) \geq S \\ f(x,y) & \text{sinon} \end{cases}$$

Il est parfois utile de mettre les valeurs ne vérifiant pas le test à une valeur fixée v2. Le seuillage peut s'écrire :

$$g(x,y) = \begin{cases} T(f(x,y)) & \text{si } f(x,y) \geq S \\ v2 & \text{sinon} \end{cases}$$

Finalement, si on veut distinguer par deux valeurs arbitraires les valeurs du fond et de la forme, on écrit :

$$g(x,y) = \begin{cases} v1 & \text{si } f(x,y) \geq S \\ v2 & \text{sinon} \end{cases}$$

2.6.3. Opérations

Parmi les opérations sur la fonction de transfert, nous n'étudions ici que celles qui ont pour résultat une fonction de transfert. Il existe trois types d'opérations de modification : translation, composition et inverse :

A) Les opérations de translation

Cette opération a pour effet de déplacer verticalement ou horizontalement une partie ou la totalité de la fonction dans le cadre D1 x D2 dans lequel elle est définie. Ce déplacement crée une nouvelle fonction définie sur le même domaine D1, faisant la transformation dans D2 mais avec une correspondance différente.

B) Les opérations de composition

La composition est définie comme une composition de fonction f = g o h ; ceci permet alors très naturellement de définir l'inverse d'une fonction et l'identité (opération de transfert sans effet) : $Id = f \circ f^{-1}$ comme c'est toujours le cas.

3. UNIFICATION DES TYPES

Nous avons introduit précédemment les objets image en montrant leurs propriétés et leur utilisation. Cette illustration va servir dans la deuxième partie de ce chapitre à les définir en cernant mieux leurs propriétés ainsi que leurs opérations caractéristiques. Ceci permettra de définir dans une étape plus avancée une hiérarchie de types image où nous pouvons mieux observer les relations entre des types différents et des types d'une même famille.

3.1. Idées de base

Nous allons utiliser les **constructeurs de types** usuels avec les opérations caractéristiques liées à notre domaine d'application. Parmi ces constructeurs, nous trouvons : le produit cartésien, la table, la liste, etc ...

Ces constructeurs, ne correspondant pas toujours tout à fait à la description de nos types, seront adaptés à nos besoins en les enrichissant par des attributs relatifs à leur forme et contenu et en proposant de leurs appliquer les mêmes sortes de modification que celles introduites dans SACSO [LEV 87] pour la spécification des types abstraits algébriques. Il s'agit essentiellement :

- d'enrichissement grâce à de nouvelles opérations,

- de restriction de l'ensemble des objets du type à ceux satisfaisant une propriété ou invariant.

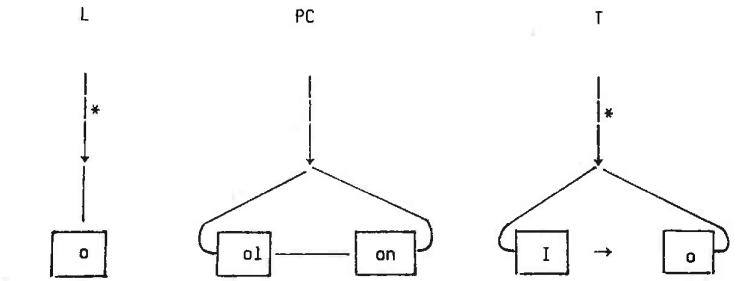
Un type image est défini par :

- une expression qui peut être un constructeur **paramétré instancié**,

- un ensemble d'opérations caractéristiques définies par un profil et un ensemble de définitions. Nous donnerons dans la suite la définition formelle des opérations les plus importantes,

- un invariant.

Nous utilisons les notations de SACSO pour schématiser les structures des types. Nous en donnons quelques exemples ci-dessous :



L est une liste d'objets o

PC est un produit cartésien d'objets o1 .. on

T est une table associant à tout indicatif I un objet o

Figure 2.20 : Structures de quelques constructeurs.

3.2. Les structures de données image

Les structures de données servant à la définition des types image appartiennent à cinq classes différentes :

- les structures élémentaires,
- les structures linéaires,
- les structures fonctionnelles,
- les structures planaires,
- les structures hiérarchiques.

3.2.1. Les structures élémentaires

Les structures élémentaires regroupent les éléments de base qui vont servir à la définition de types plus importants. Parmi ces éléments de base, nous trouvons :

- l'information,
- la position,
- le point,
- la fenêtre.

A) L'information

INFO est le type général de l'information véhiculée par les types image. C'est un type numérique. C'est donc une information sur laquelle est définie une relation d'ordre et un certain nombre d'opérations arithmétiques et logiques. L'information peut être simple ou multiple. Toutes les opérations arithmétiques et logiques sont valides sur chacun des champs de l'information multiple.

L'information contenue dans un objet image appartient à plusieurs types :

- scalaire : l'information scalaire est relative à des mesures entières ou réelles. Elle représente le module ou l'orientation du gradient, la distance entre pixels, etc...,

- intervalle : les intervalles sont utilisés dans le codage des valeurs de niveaux de gris d'une image ou dans la représentation des dimensions de la grille Ix et Iy, etc...,

- ensemble : l'ensemble est très utilisé pour représenter l'information image, comme pour représenter les valeurs binaires, les niveaux de gris ou l'information (valeur et numéro) contenue dans chaque noeud d'une pyramide, etc...

- produit cartésien :

Le produit cartésien peut contenir deux ou plusieurs champs d'informations numériques. Plusieurs informations image peuvent être de ce type comme :

position = <abscisse , ordonnée>,
point = <position , information>.

- liste :

L'information image peut être donnée par une liste de mesures correspondant à une succession de codages.

B) La position

La position sert à repérer l'emplacement d'un pixel dans l'image. Le type **POSITION** est défini comme le montre le schéma suivant par un produit cartésien d'une abscisse et d'une ordonnée de type INTERVALLE:

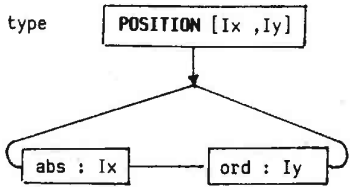


Figure 2.21 : Définition du type **POSITION**.

où Ix et Iy sont des intervalles d'entiers ou de réels respectivement suivant les axes des x et des y.

Les opérations définies sur **POSITION** sont les suivantes :

construction

creerp : **Ix x Iy → POSITON** :
crée une position à partir de son abscisse et de son ordonnée.

consultation

eqpos : **POSITION x POSITION → BOOLEEN** :
égalité de deux positions,
pvois4 : **POSITION x POSITION → BOOLEEN**
position voisine au sens de la 4-connexité ?
pvois8 : **POSITION x POSITION → BOOLEEN** :
position voisine au sens de la 8-connexité ?
distp : **POSITION x POSITION → ENTIER** :
distance entre deux positions.

modification

pvois : **POSITION x ENTIER → POSITION** :
position voisine dans une direction donnée.

C) Le point

Un point représente souvent un point de contour, un pixel dans une image, un point d'une région, un point d'un masque, etc ... On désigne par le type **POINT** le produit cartésien de deux champs où l'un est de type **POSITION** et l'autre de type **INFO**. On le représente schématiquement comme suit :

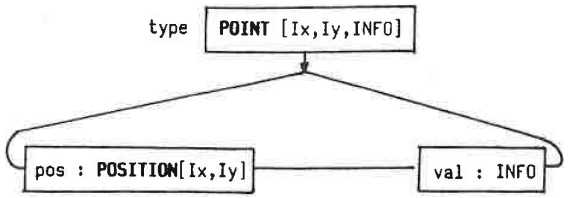


Figure 2.22 : Définition du type **POINT**

Les seules opérations définies sur **POINT** sont celles qui permettent l'accès à ses deux champs, c'est-à-dire **pos** et **val**.

D) La fenêtre

Nous désignons par fenêtre un cadre rectangulaire servant à délimiter des zones rectangulaires dans une image. Le type **FENETRE** est paramétré par les intervalles Ix et Iy. Il est défini comme un produit cartésien d'une hauteur et d'une largeur de type INTERVALLE.

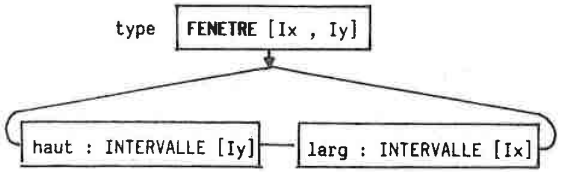


Figure 2.23 : Définition du type **FENETRE**.

Les opérations sur **FENETRE** sont définies comme suit :

construction :

initf : **Ix x Iy → FENETRE** :
initialise les dimensions d'une fenêtre.

consultation :

fvide : **FENETRE → BOOLEEN** :
indique si la fenêtre est vide,
surface : **FENETRE → ENTIER** :
calcule la surface de la fenêtre.

modification :

elargl : **FENETRE x Ix → FENETRE** :
élargissement ou rétrécissement de la fenêtre suivant x,
elargh : **FENETRE x Iy → FENETRE** :
élargissement ou rétrécissement de la fenêtre suivant y,
interf : **FENETRE x FENETRE → FENETRE** :
calcule l'intersection de deux fenêtres.

L'intersection est définie comme dans les opérations d'assemblage. Les opérations d'élargissement permettent d'agrandir ou de rétrécir une fenêtre en modifiant un de ses côtés.

3.2.2. Les structures linéaires

Le constructeur de type utilisé est la **LISTE**. On définit une liste comme un ensemble fini et ordonné d'éléments. D'un point de vue formel, la liste peut être définie comme une "suite" et des "opérations privilégiées sur la suite".

La construction d'un objet de type LISTE se fait à partir de la liste vide en ajoutant à chaque fois un élément. Nous donnons dans la suite les opérations définies sur la LISTE. LISTE désigne LISTE[INFO] (où INFO est le type de ses éléments) :

creerl : **→ LISTE** :
crée une liste vide,
ajout : **LISTE x INFO → LISTE** :
ajoute un nouvel élément en tête de la liste,
suprim : **LISTE → LISTE** :
enlève le premier élément de la liste,
lvide : **LISTE → BOOLEEN** :
vérifie si la liste est vide,
tête : **LISTE → INFO** :
donne le premier élément de la liste,
succ : **LISTE x ENTIER → INFO** :
donne le successeur d'un élément,
pred : **LISTE x ENTIER → INFO** :
donne le prédécesseur d'un élément,
subst : **LISTE x ENTIER x INFO → LISTE** :
remplace un élément par un autre,
acc : **LISTE x ENTIER → INFO** :
permet d'accéder à un élément par son rang,
concat : **LISTE x LISTE → LISTE** :
concatène deux listes.

Déclaration de variables :

```
l : LISTE,
i : INFO,
e : ENTIER.
```

On peut ainsi remarquer que :

```
lvide(creerl) = vrai,
lvide(ajout(l,i)) = faux,
suprim(creerl) = creerl,
suprim(ajout(l,i)) = l,
tête(creerl) = indéfini
tête(ajout(l,i)) = i
```

A partir de ces opérations, on peut définir d'autres opérations comme :

taille : **LISTE** → **ENTIER** : donne le nombre d'éléments,
reste : **LISTE** → **LISTE** : donne la liste privée de son premier élément,

ou des itérateurs comme :

```
e := l ; i := tête(l);
jqa succ(l,e) non défini

    faire

        traitement(acc(l,e));
        e := e+l

    fait
```

Cette forme d'itération, bien qu'elle soit cohérente par rapport à la définition de la liste, est complexe ; nous choisissons une forme plus simple comme :

```
pour i dans l faire

        traitement(i)

fait
```

La fonction "taille" s'écrit comme suit :


```
taille := 0
pour i dans l faire

        taille := taille + 1

fait
```

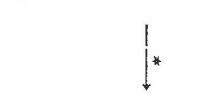
ou encore :

```
taille(l) = si lvide(l) alors 0 sinon 1 + taille(reste(l));
```

Des contraintes sur le rangement et la manipulation des éléments de la liste permettent de définir des structures particulières de **LISTE** comme la pile et la file.

Les listes linéaires peuvent être utilisées en traitement d'images pour représenter des **lignes de contour** (= liste de points), des **tracés graphiques** (= liste de positions), le résultat d'approximation polygonale de courbes (= liste de positions), le résultat de segmentation de courbes (= liste de codes). Dans la représentation tridimensionnelle des objets, un polyèdre par exemple peut être représenté par la liste de ses surfaces où chacune d'entre elles est donnée par la liste de ses côtés.

On représente dans SAPIN tous les objets image ayant une structure linéaire par une LISTE d'items. Item est un élément de LISTE et peut être de type simple ou structuré. On reporte le type d'item, INFO, au niveau du constructeur LISTE :



La figure suivante montre deux exemples d'objets image de type **LISTE** :

- **GRAPHIQUE** est défini comme une liste de positions paramétrée par les types des indices Ix et Iy,

- **NIMAGE** est une liste d'images paramétrée par INFO-IM (type de l'information contenue dans IMAGE) et par les types des indices Iix et Iiy correspondant respectivement à l'indice des colonnes et à l'indice des lignes :

GRAPHIQUE [Ix,Iy] = **LISTE** [POSITION[Ix,Iy]]

NIMAGE [Iix,Iiy,INFO-IM] = **LISTE** [IMAGE[Iix,Iiy,INFO-IM]]

Nous donnons dans la suite la spécification du type abstrait **CONTOUR** défini à partir du constructeur **LISTE** :

A) Définition du type abstrait : CONTOUR

CONTOUR est le type des lignes de contour. On le définit comme un produit cartésien d'une liste de points de contour consécutifs et d'un certain nombre d'attributs réels relatifs à des mesures d'homogénéité, d'épaisseur, de continuité, etc... La figure suivante montre la structure du type **CONTOUR** donnée avec deux attributs long et nbc où long donne la longueur absolue du contour et nbc, le nombre de ses sous-contours :

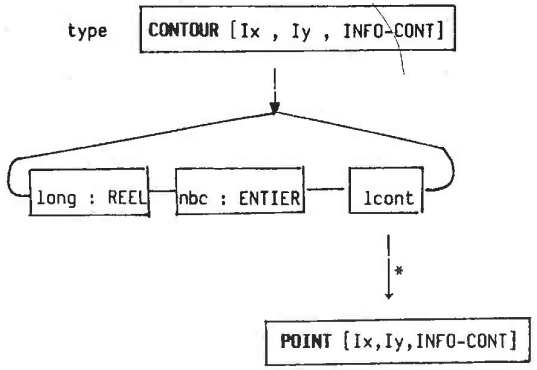


Figure 2.24 : Structure du type **CONTOUR**.

Les opérations caractéristiques de **CONTOUR** sont obtenues à partir des opérations sur **LISTE** :

consultation :

cadre : **CONTOUR → FENETRE :**
déterminе le cadre circonscrit au contour,
fermeture : **CONTOUR → BOOLEEN :**
rend vrai si le contour est fermé,
surface : **CONTOUR → ENTIER :**
calcule le nombre de positions à l'intérieur du contour.

modification :

inverse : **CONTOUR → CONTOUR :**
inverse le sens du contour,
echant : **(fech) x CONTOUR → CONTOUR :**
sélectionne les points essentiels dans le contour d'après fech,
où : fech : **POINT → BOOLEEN :**
est la fonction de sélection élémentaire définie à partir de l'invariant.

Remarque :

La notation usuelle de profil est :

f : **T1 x T2 → T3**

où T1, T2, T3 désignent des types, mais parfois on ne s'intéresse pas particulièrement à l'un des types (ou il est inutilement complexe) et alors, à la place de :

f : **T1 x T2 → T3**

on choisit une variable y de type T2 (d'après le contexte) et on note :

f : **T1 x (y) → T3**

Déclaration de variables :

c : CONTOUR,
p : POINT,
l : LISTE.

Quelques définitions

◇ **cadre**(c) = [minx..maxx,miny..maxy]

avec :

```
minx = +∞ ; maxx = 0;
miny = +∞ ; maxy = 0;

pour p dans lcont(c) faire

    minx = min(minx,abs(p));
    maxx = max(maxx,abs(p));
    miny = min(miny,ord(p));
    maxy = max(maxy,ord(p));

fait
```

où **abs** et **ord** sont des fonctions définies sur le type **POSITION** et qui donnent respectivement l'abscisse et l'ordonnée de la position du point courant.

◇ **fermeture**(c) = eqpos(tête(l(c)),acc(l(c),taille(l(c))))

où **eqpos** est une fonction définie sur **POSITION** qui vérifie si deux positions sont égales (ont mêmes coordonnées). acc(l(c),taille(l(c))) fournit le dernier point du contour.

3.2.3. Les structures fonctionnelles

La structure utilisée est la **TABLE**. Une table est une fonction qui associe à toute entrée ou **INDICE** une valeur ou élément de type **INFO**. A chaque indice correspond un seul élément de type **INFO**.

On construit la table élément par élément en associant à chaque fois un élément à un indice. Initialement, la table est vide.

Les opérations sur **TABLE**[INDICE,INFO] sont :

creert : → **TABLE** :
initialise la table à vide,

insert : **TABLE x INDICE x INFO → TABLE** :
ajoute un élément à la table,

suprim : **TABLE x INDICE → TABLE** :
supprime un élément donné par son indice,

tvide : **TABLE → BOOLEEN** :
teste si la table est vide,

appt : **TABLE x INDICE → BOOLEEN** :
teste si un indice appartient à la table,

accès : **TABLE x INDICE → INFO** :
accès à un élément donné par son indice,

mod : **TABLE x INDICE x INFO → TABLE** :
modifie un élément donné par son indice.

Les objets linéaires image de type **TABLE** sont : l'**histogramme**, la **projection** et la **table de transfert**. Ces objets sont limités par la taille. Pour l'histogramme, INDICE est en général un type niveaux de gris qui est un intervalle d'entiers $[0..2^q-1]$, et INFO est REEL. Pour la projection, INDICE est l'intervalle de variation des abscisses ou des ordonnées des pixels et INFO est aussi le REEL. Enfin, pour la table de transfert, INDICE et INFO sont tous les deux des intervalles de variation de niveaux de gris. Comme on peut le remarquer dans le cas de ces objets, INDICE est ordonné.

Il importe de savoir comment définir des opérations caractéristiques supplémentaires sur ces objets image à partir des opérations élémentaires définies sur la table, comme par exemple, comment remplacer dans une table de transfert toute une plage de niveaux de gris par une seule valeur, ou comment chercher dans un histogramme le premier élément ayant une valeur particulière. Pour ces deux types d'opérations, il faut des **itérateurs** adaptés à la structure de **TABLE**.

Pour le premier, LEVY [LEV 84] a introduit l'itérateur :

$$iter_{f,P} : \textbf{TABLE} \rightarrow \textbf{TABLE}$$

qui applique la fonction f sur tous les éléments de la table vérifiant la propriété P.

Pour le deuxième type d'opération, il s'agit de parcourir les éléments de la table et de s'arrêter sur le premier élément vérifiant la propriété P. Ceci est possible puisque nous avons une relation d'ordre sur les indices. On note ce nouvel itérateur comme suit :

$$iterp_{p} : \textbf{TABLE} \rightarrow \textbf{INDICE}$$

$iterp_p$ sera défini comme suit :

i : INDICE,
v : INFO,
t : TABLE.

$iterp_p$(creert) = indéfini

$iterp_p$(insert(t,i,v)) = **si** P(v) **alors** i
sinon $iterp_p$(t)
fsi

3.2.4. Les structures planaires

Les objets planaires image tels que **IMAGE**, **MASQUE** et **REGION** ont la même structure de base définie comme une table qui associe à toute position dans un cadre une information. Le cadre est paramétré par les intervalles de variation des abscisses Ix et des ordonnées Iy. Nous avons introduit un constructeur de type plus riche que **TABLE** appelé **GRILLE** pour les définir.

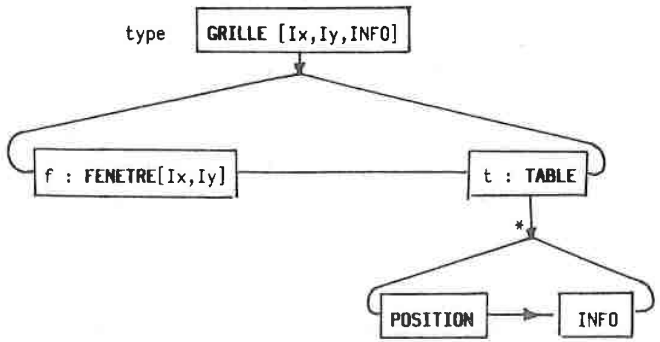


Figure 2.25 : Structure de **GRILLE**.

Ce schéma montre que le type **GRILLE** est défini comme un produit cartésien d'une fenêtre f et d'une table t. La fenêtre est définie par deux intervalles larg et haut définis respectivement sur Ix et Iy. La table t associe à toute position p une valeur de type INFO. Au type **GRILLE** sont associées des opérations caractéristiques réparties dans quatre classes comme suit :

construction :

creerg : **FENETRE → GRILLE** :
crée une grille vide,
defg : **GRILLE x POINT → GRILLE** :
construit une grille par adjonction d'un nouveau point,

consultation :

appart : **GRILLE x POSITION → BOOLEEN** :
vérifie l'existence d'un point,
val : **GRILLE x POSITION → INFO** :
donne la valeur d'un point,
consult : **GRILLE → INFO** :
calcule une valeur caractéristique de toute la grille,
succ : **GRILLE x POSITION → POSITION** :
donne la position du point suivant dans la grille,
voisin : **GRILLE x POSITION x ENTIER → POSITION** :
donne la position du point voisin relativement à la grille dans une direction donnée choisie parmi [0,..,7].

modification :

modfpts : **(fps) x GRILLE → GRILLE** :
modification ponctuelle simple à l'aide de la fonction élémentaire fps,
où : fps : **INFO → INFO** :
remplace une valeur par une autre,
modfptd : **(fpd) x GRILLE x GRILLE → GRILLE** :
opération sur deux grilles,
où : fpd : **INFO x INFO → INFO**:
détermine une valeur par modification des valeurs d'un couple de points homologues dans les grilles données.

itérateurs :

$pourtout_p$: **GRILLE → LISTE-POSITION** :
donne successivement les positions des points vérifiant la propriété P,
$premier_p$: **GRILLE x POSITION x ENTIER → POSITION** :
donne à partir d'une position et d'une direction de recherche, la position du premier point vérifiant la propriété P,
$dernier_p$: **GRILLE x POSITION x ENTIER → POSITION** :
idem que $premier_p$ mais donne le dernier point.

Déclaration de variables :

```
g,g',g1,g2,g3 : GRILLE,
f : FENETRE,
p : POINT,
e : ENTIER.
```

Définitions des opérations

◇ **creerg** : la définition d'une grille g impose de préciser son cadre f. La table t est vide au départ :

```
creerg(f) = <f,creert>
```

◇ **defg** : la construction de la grille g se fait par insertion de points en testant à chaque insertion le dépassement de capacité et le dépassement des bornes du cadre :

Dans la suite, le point p est confondu avec sa position ; p et pos(p) sont équivalents.

```
defg(g,p) = <f(g),t>

t = si dep-capacité(INFO,val(p))
    ou dep-bornes(f(g),p)
    alors t(g) sinon
        si appt(t(g),p)
        alors mod(t(g),p,val(p))
        sinon insert(t(g),p,val(p))
        fsi
    fsi
```

val est une fonction sur **POINT** donnant la valeur d'un point. Les fonctions **appt**, **mod** et **insert** sont des opérations associées au type **TABLE** utilisées ici sur (la table t) de la grille g. **dep-capacité** vérifie le dépassement de capacité de val(p) dans le domaine de variation de INFO. **dep-bornes** vérifie le dépassement des bornes de la fenêtre f(g) par la position du point courant p.

◇ **appart**(g,p) = appt(t(g),p).

◇ **val**(g,p) = accès(t(g),p) où accès est une fonction définie sur table pour accéder à la valeur d'un indicatif.

◇ **succ**(g,p) : est la fonction successeur donnant la position suivante sur la même ligne. Si la position courante est la dernière sur une ligne, alors la position suivante est indéterminée. Cela suppose qu'il existe un **ordre de parcours** implicite d'une image. Cet ordre existe en pratique ; il correspond à l'ordre de balayage physique de l'image ligne par ligne.


```
succ(g,p) = si dep-bornes(f(g),<abs(p)+1,ord(p)>)
            alors indéterminée
            sinon <abs(p)+1,ord(p)>
            fsi
```

où **abs** et **ord** sont des fonctions définies sur le type **POSITION** pour donner respectivement l'abscisse et l'ordonnée d'une position.

◇ **voisin**(g,p,e) : la structure de la grille se caractérise par l'opération plane **voisin** qui permet d'aller d'un point à son voisin. La position du voisin est déterminée à partir de la direction cardinale donnée par un entier e ∈ [0,..,7]. On définit cette opération comme suit:

```
voisin(g,p,e) = si tvide(t(g)) alors indéterminée
                sinon
                   si pvois(p,e) ∉ [Ix x Iy]
                   alors indéterminée
                   sinon pvois(p,e)
                   fsi
                fsi
```

où **pvois** est une fonction définie sur le type **POSITION** donnant la position voisine d'une position donnée dans une direction donnée.

◇ **consult**(g) : est une fonction de consultation de la grille permettant d'extraire une information élémentaire ou structurée caractéristique des valeurs présentes. Nous pouvons trouver plusieurs fonctions de consultation de la grille comme le maximum, le minimum, la dynamique, la moyenne, le moment, etc... Nous définissons la dynamique, par exemple, comme suit :

```
dynamique(g) = max(g) - min(g)
max(g) = mx ; mx = -∞ ;
pourtout p dans t(g) tq val(p)>mx : mx=val(p) ;
min(g) = mn ; mn = +∞ ;
pourtout p dans t(g) tq val(p)<mn : mn=val(p) ;
```

◇ **modfpts** (fps,g) : crée une grille g' par modification de tous les points de la grille g à l'aide de la fonction fps. On la définit à partir de la fonction de construction **defg** comme suit :

```
modfpts (fps,g) = <f(g'),t'>

t' = pourtout p dans t(g) : defg(g',<p,fps(p)>)
```

◇ **modfptd**(fpd,g1,g2) : crée une nouvelle grille g3 par modification des deux grilles g1 et g2. La fonction élémentaire **fpd** agit à chaque

fois sur un couple de points homologues respectivement dans g1 et g2. On la définit comme suit :

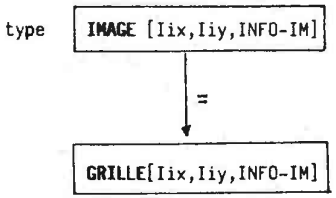
modfpd(fpd,g1,g2) = <f(g3),t3>

t3 = **pourtout** p1 **dans** t(g1), p2 **dans** t(g2) : defg(g3,<p1,fpd(p1,p2)>)

Nous allons maintenant définir les types image qui ont pour constructeur de type **GRILLE**

A) Définition du type abstrait : IMAGE

On définit **IMAGE** à partir de **GRILLE** comme suit :



où INFO-IM ⊂ INFO.

L'ensemble des opérations sur **IMAGE** est formé par l'union des opérations définies sur **GRILLE** et de celles caractéristiques d'**IMAGE**. Ces opérations caractéristiques sont définies sur les domaines de définition de **GRILLE** :

consultation :

histo : **IMAGE** → **HISTO** :
construit l'histogramme de l'image,

modification :

modvois : **(fvois x svois) x MASQUE** → **IMAGE** :
fonction de convolution utilisant un masque,
où : svois : **IMAGE x POSITION x ENTIER x ENTIER** → **MASQUE** :
détermine le voisinage d'un point donné et le range dans un masque,
et fvois : **MASQUE x MASQUE** → **POINT** :
détermine une valeur par convolution de deux masques.

zoomr : **(svois x spt) x IMAGE x ENTIER x ENTIER** → **IMAGE** :
fonction de réduction d'une image en x et y,
où : spt : **MASQUE** → **INFO-IM** :
détermine une valeur représentant tout le voisinage,
zoome : **(fagr) x IMAGE x ENTIER x ENTIER** → **IMAGE** :
fonction d'expansion d'une image en x et y,
où : fagr : **POINT x ENTIER x ENTIER** → **MASQUE** :
remplace tout point par un masque dont la taille est indiquée en paramètres.

Déclaration des variables :

```
i,i' : IMAGE,
h : HISTO,
p : POINT,
m : MASQUE,
f : FENETRE,
t,t' : TABLE,
ht,lg,x,y,x1,y1,x2,y2 : ENTIER.
```

Nous définissons ces fonctions de la manière suivante :

◇ **histo** (i) = <f(h),t'>

```
t' = ø , pourtout p dans t(i) :
     si appt(t',val(p))
     alors modif(t',val(p),accès(t',val(p))+1)
     sinon insert(t',val(p),1)
     fsi
```

◇ **modvois**(fvois,svois,i,m) : définit une image i' telle que :

```
modvois(fvois,svois,i,m) =
   si f(i') ⊂ f(i) alors erreur
   sinon <f(i'),t'>
où t' = pourtout p dans t(i) :
        defg(i',fvois(svois(i,p,haut(f(m)),larg(f(m))),m))
```

◇ **zoomr**(svois,spt,i,ht,lg) : définit une image i' telle que :

```
zoomr(svois,spt,ht,lg) =
        si haut(f(i))/ht < haut(f(i'))
        ou larg(f(i))/lg < larg(f(i'))
        alors erreur
        sinon <f(i'),t'>
        fsi
  où t' = pourtout p dans t(i) tq posvois(p,<ht,lg>)
          defg(i',spt(svois(i,p,ht,lg)))
```

% **posvois**(p,<ht,lg>) prend comme successeur de la position p celle

qui se trouve immédiatement en dehors du voisinage <ht,lg> de p (où p est le coin supérieur gauche) sur la même ligne %

◇ **zoome**(fagr,i,ht,lg) : définit une image i' telle que :

zoome(fagr,i,ht,lg) = **si** haut(f(i))*ht > haut(f(i'))
ou larg(f(i))*lg > larg(f(i')) **alors** erreur
sinon <f(i'),t'>
fsi

où t' = **pourtout** p **dans** t(i), p' **dans** m(svois(i,p,<ht,lg>) :
defg(i',<poscor(p,pos(p'),
f(m(svois(i,p,<ht,lg>)))),val(p))

où posvois(<x,y>,<ht,lg>) = (x mod ht=0) et (y mod lg=0)
poscor(<x1,y1>,<x2,y2>,<ht,lg>) =
<ht*(x1-1)+x2,lg*(y1-1)+y2>

B) Définition du type abstrait : MASQUE

MASQUE est une grille de petite taille contenant soit un voisinage de points dans une image, soit les coefficients d'une fonction locale. Le type **MASQUE** est défini par :

- son support de type **GRILLE** précisant ses dimensions et le type de ses valeurs,

- sa nature parmi voisinage, masque de convolution, élément structurant, défini par un code nt,

- son centre : le masque n'étant pas toujours carré, il est nécessaire de préciser le centre des points représentés. La figure suivante donne le schéma graphique du type **MASQUE** :

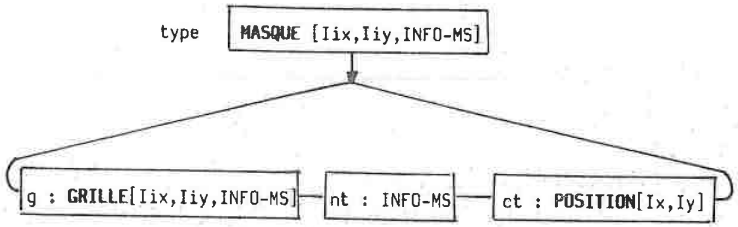


Figure 2.26 : Définition du type **MASQUE**.

INFO-MS est le type de l'information contenue dans le masque. Elle est moins générale que l'information contenue dans **GRILLE**.

Le type **MASQUE** est défini par toutes les opérations définies sur son constructeur **GRILLE** et par d'autres opérations plus caractéristiques dont nous donnons ci-après la définition de certaines d'entres elles.

construction :

svois : **IMAGE x POSITION x ENTIER x ENTIER → MASQUE** :
construit un masque à partir du voisinage d'un point d'image,
repv : **MASQUE x INFO-MS x ENTIER x POSITION → MASQUE** :
reproduction d'une valeur un nombre de fois à partir d'une position donnée,
repl : **MASQUE x MASQUE x ENTIER x ENTIER → MASQUE** :
reproduction d'une ligne de type MASQUE un nombre de fois à partir d'un indice donné,
repc : **MASQUE x MASQUE x ENTIER x ENTIER → MASQUE** :
reproduction d'une colonne de type MASQUE un nombre de fois à partir d'un indice donné.

consultation :

detptvois : **MASQUE → POINT** :
détermine un point d'intérêt à partir d'un masque,
detvalvois : **MASQUE → INFO-MS** :
détermine une valeur d'intérêt à partir d'un masque.

modification :

permutl : **MASQUE x ENTIER x ENTIER → MASQUE** :
permutation de deux lignes,
permutc : **MASQUE x ENTIER x ENTIER → MASQUE** :
permutation de deux colonnes,
rotation : **MASQUE x ENTIER → MASQUE** :
rotation du masque autour du point central d'un angle multiple de 45°,
distm : **MASQUE x MASQUE → REEL** :
comparaison de deux masques et calcul d'une valeur de ressemblance.

Définition des opérations

Nous donnons ci-après les définitions informelles des opérations sur **MASQUE** dans le but de mieux comprendre leur fonctionnement :

◇ svois : remplit un masque à partir d'un voisinage de points se trouvant dans une image donnée. La disposition des valeurs dans le masque est la même que dans le voisinage. Quand le voisinage est incomplet (cas des pixels sur les bords), le masque est rempli par des valeurs indéfinies et son utilisation est condamnée. Un message d'erreur sera transmis à chaque tentative d'utilisation.

◇ **repv, repl** et **repc** : ces opérations de répétition ne seront effectives que si les valeurs répétées sont de type INFO-MS, que si la taille de la ligne (resp. de la colonne) répétée est égale à la taille de la ligne (resp. de la colonne) du masque et que si la position à partir de laquelle s'effectue la répétition appartient au cadre du masque.

◇ **detptvois** et **detvalvois** : il s'agit de déterminer par ces opérations un point ou une valeur représentative du masque entier. Plusieurs fonctions de tri sont de ce type sélectionnant pour la valeur, le minimum, le maximum, la médiane, etc..., et pour le point, le centre ou un point dont la valeur a une particularité mathématique, géométrique ou topologique.

◇ **rotation** : la rotation se fait à partir d'une position appartenant au cadre et suivant un angle multiple de 45°. Ceci permet d'effectuer la rotation (autour du centre) par déplacement des points de la position qu'ils occupent vers une position voisine et ceci un certain nombre de fois.

◇ **distm** : distm est la fonction de calcul de la similarité entre deux masques. Les deux masques doivent avoir les mêmes dimensions et être de même type afin de donner un sens à la comparaison. Plusieurs fonctions de mise en correspondance permettent le calcul de cette distance.

C) Définition du type abstrait : REGION

Une région est définie par une grille représentant la carte géographique d'objets dans une scène. Cette carte est déterminée à partir d'une image et de propriétés de regroupement. Le type abstrait **REGION** est défini comme un sous-type de **GRILLE** où INFO est entier. Nous associons au support(**GRILLE**) de la région une table (tr) de description associant à un objet, la surface de l'objet (sr = nombre de points composant) et le cadre circonscrit (fn) qui permet de situer l'objet dans la région.

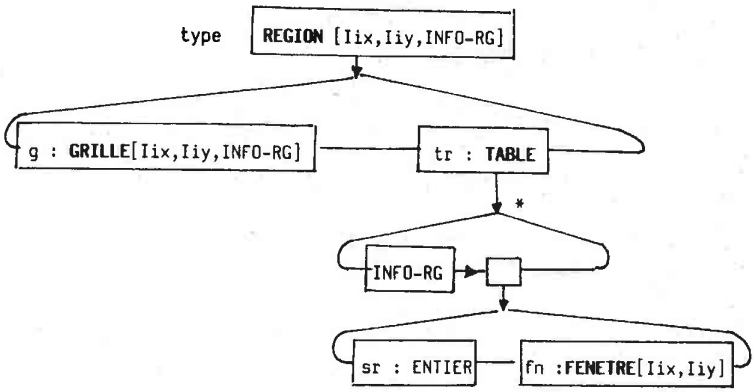


Figure 2.27 : Définition du type **REGION**.

Nous pouvons remarquer qu'à l'inverse de l'image et du masque, les points de la grille de la région ne sont pas tous définis car les objets dans la scène peuvent ne pas être tous représentés. Par ailleurs, les opérations définies sur **REGION** doivent agir en même temps sur la grille et sur la table (tr) de description des objets.

Nous donnons ci-après certaines des opérations les plus caractéristiques de **REGION**. Afin de mieux comprendre le fonctionnement des opérations suivantes, nous allons rappeler rapidement la notion de sous-région.

si r est une région, une sous-région r' de r est une région telle que :

g(r') = g(r),
tr(r') ⊂ tr(r),
taille(tr(r)) = 1 + taille (tr(r - r')) où r - r' signifie r privée de r'.

construction :

ajoutp : **REGION x POINT → REGION** :
adjonction d'un point à une sous-région donnée.

consultation :

densité : **REGION → REEL** :
rapport entre 0 et 1 de la surface des régions sur la surface du cadre général,

cdmin : **REGION → FENETRE** :
cadre circonscrit à tous les objets représentés,

numrg : **REGION x POSITION → ENTIER** :
numéro de la sous-région dont on connaît une position,

apptrg : **REGION x ENTIER → BOOLEEN** :
recherche d'une sous-région d'un numéro donné.

modification :

supr : **REGION x ENTIER → REGION** :
suppression d'une sous-région connaissant son numéro,

suppt : **REGION x POINT → REGION** :
suppression d'un point dans une sous-région,

modr : **REGION x ENTIER x ENTIER → REGION** :
changement du numéro d'une sous-région,

union : **REGION x ENTIER x ENTIER → REGION** :
union de deux sous-régions en leurs affectant le même numéro,

uniont : **REGION → REGION** :
union de toutes les sous-régions.

Nous donnons ci-après la définition algébrique de quelques-unes de ces opérations.

Déclaration des variables :

```
r : REGION,
p : POINT,
i : IMAGE,
f : FENETRE,
t,t' : TABLE,
n,n' : ENTIER.
```


Définition de quelques opérations :

```
ajoutp(r,p) = <defg(g(r),p),n',t'>
où :
n' = si val(p)<nr(r) alors
        n' = nr(r),
        t' = t(r) tq :
                fn(tr(r)) = [cminx..cmaxx,cminy..cmaxy]
                cminx = min(absmin(fn(tr(r))),abs(p))
                cmaxx = max(absmax(fn(tr(r))),abs(p))
                cminy = min(ordmin(fn(tr(r))),ord(p))
                cmaxy = max(ordmax(fn(tr(r))),ord(p))
             sinon
        n' = nr(r) + 1
        t' = ajout(t',(nr,<1,[abs(p)..abs(p),
                              ord(p)..ord(p)]>))
   fsi
```

◇ **suppt** (r,p) = <t,n,t'> tel que :

```
t,n,t' = si non appt(t,p) alors erreur
         sinon si sr(tr(r)[val(p)]) > 1
         alors
          t = suprim(t(g(r)),p)
          n = nr(r)
          t' = tr(r) tq :
               fn(tr(r)[val(p)]) = [cminx..cmaxx,cminy..cmaxy]
               cminx = -∞ ; cminy = -∞ ;
               cmaxx = 0 ; cmaxy = 0;
               pourtout p' dans g(r) tq
               val(p') = val(p) faire
               cminx = min(cminx, abs(p')
               cmaxx = max(cmaxx, abs(p')
               cminy = min(cminy, ord(p')
               cmaxy = max(cmaxy, ord(p')
               ...
```

3.2.5. Les structures hiérarchiques

A) Généralités

Plusieurs algorithmes considèrent une image comme une hiérarchie d'informations. La structure de données la plus utilisée pour représenter cette hiérarchie est l'**ARBRE**.

Un arbre peut être défini comme un graphe A = (N,R) où N est un ensemble fini d'éléments appelés **Noeuds** et R ⊂ NxN est une relation binaire définie par les deux propriétés suivantes :

(1) parmi les noeuds N de l'arbre, il en existe un particulier n0 appelé **racine** tel que : $\forall\, n \in N,\ n \neq n0 \Rightarrow (n,n0) \notin R$,

(2) $\forall\, n1 \in N,\ n1 \neq n0,\ \exists\, n2$ unique tel que $(n2,n1) \in R$.

Cette relation sert à définir les liens entre pères et fils d'un arbre : si $(n1,n2) \in R$, on dit que n1 est le père de n2 et que n2 est le fils de n1. Si $n \in N$, le sous arbre de racine n est défini par $An = (Nn,R^n)$ où:

$Nn = \{m \in N \,/\, (n,m) \in R^*\}$ où R^* est défini par :

R^0 = égalité,
$R^n = \{(x,y) \,/\, \exists\, z\ ;\ (x,z) \in R \text{ et } (z,y) \in R^{n-1}\}$,
$R^n = R^{n-1} \circ R$
$R^* = \bigcup_{n \geq 0} R^n$.

De cette manière, on voit qu'un arbre peut être défini de façon récursive à partir des deux propriétés ci-dessus. Pour compléter cette définition formelle de l'arbre, nous dirons que les familles (ensembles de noeuds fils d'un même père) sont totalement ordonnées. Nous allons donner dans la suite la définition algébrique de l'arbre binaire, c'est-à-dire un arbre tel qu'un père admet 0, 1 ou 2 fils. Ceci constitue un cas particulier de la structure d'arbre, mais dans les images, on utilise des structures plus riches avec accès au père.

Les opérations définies sur le type **ARBRE** sont :

initarb : → **ARBRE** :
crée un arbre vide,
consarb : **ARBRE x NOEUD x ARBRE → ARBRE** :
construit un arbre en définissant sa racine et ses sous-arbres gauche et droit,
agauche : **ARBRE → ARBRE** :
donne le sous-arbre gauche d'un arbre binaire,
adroite : **ARBRE → ARBRE** :
donne le sous-arbre droit d'un arbre binaire,
racine : **ARBRE → NOEUD** :
donne la racine d'un arbre,
avide : **ARBRE → BOOLEEN** :
vérifie si un arbre binaire est vide.

Déclaration des variables :

n : NOEUD,
a,a' : ARBRE

Définition des opérations :

avide(initarb) = vrai,
avide(consarb(a,n,a')) = faux,
agauche(initarb) = initarb,
agauche(consarb(a,n,a')) = a,
adroite(initarb) = initarb,
adroite(consarb(a,n,a')) = a',
racine(initarb) = erreur,
racine(consarb(a,n,a')) = n.

Parmi les représentations arborescentes de l'image, nous trouvons :

- les arbres de segmentation,
- les arbres quaternaires ou pyramidaux.

B) Les arbres de segmentation

Les arbres sont utilisés comme structure d'accueil pour représenter les résultats de segmentation d'une image et comme structure de travail pour agir sur cette hiérarchie de résultats. Horowitz et Pavlidis [HOR 76] ont utilisé la structure d'arbre pour ranger le résultat de segmentation d'images en régions homogènes obtenues par une technique de division-fusion en utilisant comme critère de fusion, l'égalité des niveaux de gris. Ce problème de segmentation peut être formulé ainsi :

Soit E le domaine de définition de l'image, S un sous-domaine de E et P(S) un prédicat booléen qui est vrai si S contient un seul niveau de gris et faux sinon. Une segmentation de E conduit à une partition de E en sous-domaines Si, i=1,2,...,n tel que :

$$(1)\ E = \bigcup_{i=1}^{n} Si$$

(2) $\forall\ i \neq j\ Si \cap Sj \neq 0$
(3) $\forall i\ P(Si) = Si$
(4) $\forall\ i \neq j\ P(Si \cup Sj) = faux$

La structure utilisée par Horowitz et Pavlidis est appelée "arbre de segmentation". La racine de l'arbre correspond au domaine entier E de l'image, le fils d'un noeud représente une partition de la région liée à son père. La figure 2.28 montre un exemple de segmentation de régions et l'arbre correspondant :

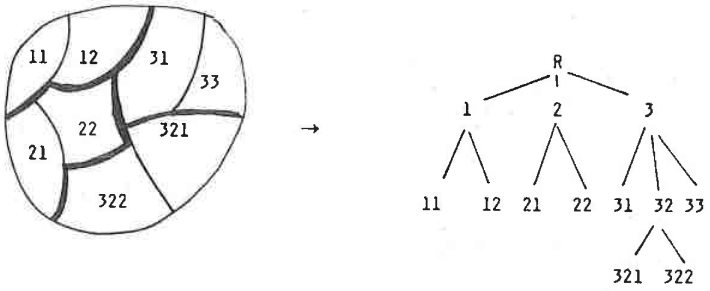


Figure 2.28 : Arbre de segmentation d'une image.

L'algorithme de division-fusion commence par une partition arbitraire de E et utilise les opérations de division et fusion pour réaliser la segmentation totale de E. L'algorithme divise chaque région en deux et essaie de fusionner deux régions voisines ayant le même niveau de gris. La division se réalise facilement ici par l'éclatement du noeud considéré en deux. La fusion regroupe les noeuds adjacents (représentants des régions adjacentes) et les remplacent par un seul.

Batchelor [BAT 79] utilise la structure d'arbre pour représenter des polygones en terme de leurs concavités et convexités. Etant donné un polygone, la racine de l'arbre contient la partie convexe de l'arbre et chaque autre noeud représente la partie convexe du trou associé au père (cf. figure 2.29).

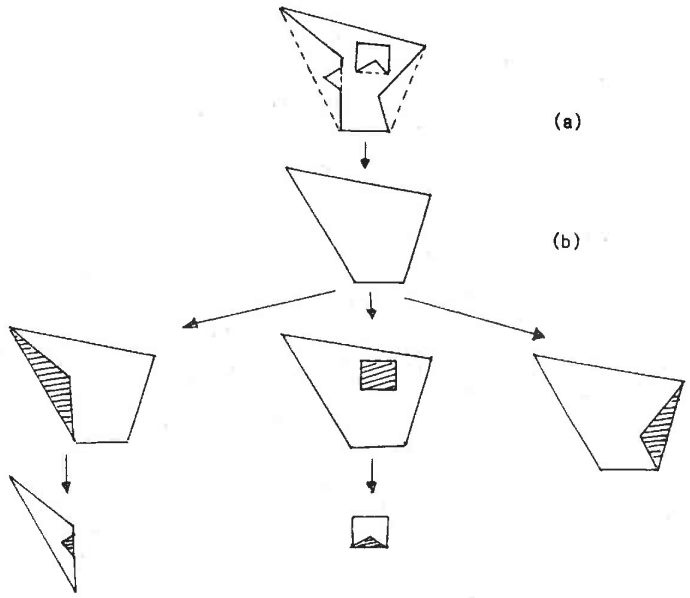


Figure 2.29 : Arbre de segmentation d'un polyèdre.

L'avantage de cette représentation est son invariance par rapport à la rotation et à l'homothétie. Son principal inconvénient est l'instabilité puisque le changement d'un seul trait dans le polyèdre modifie la structure totale de l'arbre.

C) La structure pyramidale

La structure pyramidale est une représentation hiérarchique d'une image en un ensemble d'étages où chaque étage dans la hiérarchie représente l'image avec une résolution plus fine [ROS 83] [TAN 83]. La structure pyramidale est définie par décomposition récursive de l'image en quadrants engendrant un arbre quaternaire. La racine représente l'image entière. Elle a quatre fils correspondant à la subdivision de l'image en quadrants. De manière récursive, chaque noeud a quatre fils représentant une subdivison du quadrant en quatre. Les feuilles contiennent les pixels de l'image.

Les avantages de cette représentation hiérarchique sont multiples. D'abord, elle permet de traiter une image à différents niveaux de résolution en utilisant une seule structure de données, ou d'accélérer certains algorithmes en n'utilisant la résolution maximale que lorsque cela s'avère vraiment nécessaire [FAU 83] [BAL 81] [SAM 80a] [SAM 80b] [TAN 75]. D'autre part, elle procure un codage optimal de l'image. Si une région de l'étage inférieur possède une homogénéité suffisante, le

noeud correspondant sera une feuille de l'arbre, ce qui permet de réduire considérablement la quantité d'information.

a) Notion de voisinage

La plupart des transformations opérant sur les pyramides nécessitent de pouvoir trouver les quadrants adjacents à un noeud donné. Plusieurs algorithmes ont été donnés dans la littérature [HUN 79], [SAM 82] fondés sur la recherche du père commun en partant des pixels. En collaboration avec l'ETCA (Etablissement Technique Central de l'Armement), nous avons développé sur le même principe un autre algorithme, meilleur les précédents, fondé sur l'ordre lexicographique.

La racine porte le numéro ø. Le numéro de chaque noeud est formé par le numéro de son père suivi d'un numéro d'ordre parmi [0,1,2,3]. De cette manière, il devient facile d'une part de situer de manière complète un noeud par rapport à ses voisins (latéraux et spatiaux) et d'autre part de trouver les quadrants adjacents d'un noeud. Le numéro de l'étage d'un noeud est donné par la longueur (nombre de chiffres) de son numéro. Le numéro du premier ancêtre commun de deux noeuds (plus petite pyramide englobante) est donné par la première partie commune de leur numéro.

Si la partie commune est inexistante, alors la partie commune est la racine de la pyramide. Le numéro d'ordre de leur quadrant est égal au premier chiffre après la partie commune. Enfin, à partir d'un numéro, il est facile de retrouver les coordonées x et y du noeud dans l'étage où il se trouve.

b) Exemple de numérotation

						000	003	030	033	300	303	330	333
						001	002	031	032	301	302	331	332
		00	03	30	33	010	013	020	023	310	313	320	323
0	3	01	02	31	32	011	012	021	022	311	312	321	322
1	2	10	13	20	23	100	103	130	133	200	203	230	233
		11	12	21	22	101	102	131	132	201	202	231	232
						110	113	120	123	210	213	220	223
						111	112	121	122	211	212	221	222

Figure 2.31 : Numérotation d'une pyramide de base 8x8 en mettant en évidence les familles.

c) Opérations

La structure pyramidale est utilisée pour réaliser plusieurs types d'algorithmes de transformation d'images. Tanimoto [TAN 86] distingue trois types de classification d'algorithmes :

α) Classification suivant les mouvements de données

Cette classification est dûe à Hanson et Riseman et est relative à une architecture pyramidale [HAN 74]. Elle prend en compte la nature des mouvements de données entre les différents processeurs dans les étages. Hanson et Riseman distinguent trois types de mouvement de données :

◆ réduction (de bas en haut) : les données se propagent de la base (niveau le plus large) vers la racine (niveau le plus fin) en subissant des réductions à leur passage par les étages. Dans ce type d'opération, le calcul de la nouvelle valeur du noeud dépend de celle de ses fils.

Parmi les opérations de réduction utilisées, notons la moyenne, le minimum, le maximum, mais aussi les opérations d'étiquetage et de segmentation de régions. Ces dernières opérations conduisent à la réalisation de marquages de chemins d'accès à des points d'intérêt dans une image représentant des contours ou des régions, par exemple.

◆ projection (de haut en bas) : au contraire des mouvements précédents, ici les données se propagent du niveau le plus fin vers le niveau le plus large. Les opérations qui réalisent ces projections sont utilisées pour affecter des valeurs aux pixels de la base en considérant des critères qui ne dépendent pas uniquement de leur valeur. Cela donne plusieurs moyens de localisation de points inférieurs. Parmi les plus caractéristiques, notons la transmission isotrope des données [MER 83] où les points marqués par les algorithmes ascendants reçoivent la même information caractéristique de la région qu'ils forment.

◆ flot latéral concernant le mouvement horizontal des données. Dans le flot latéral, les données restent au même niveau.

β) Classification par la nature des données

On peut classer les algorithmes pyramidaux en fonction des types de donnés manipulés. C'est ainsi que Tanimoto a défini trois types de pyramides :

◆ Pyramides binaires : les algorithmes manipulent des images binaires. A partir des opérations logiques de base AND, OR, Tanimoto a défini la pyramide AND et la pyramide OR.

Chaque noeud de la pyramide contient un vecteur de 14 composantes binaires relatives aux valeurs de ses voisins (spatiaux et latéraux). Ce vecteur est noté $Vx = \{x1,x2,...,x14\}$. Les vecteurs de tous les noeuds sont comparés à une forme $P = \{p1,p2,...,p14\}$ où les $pi \in \{0,1,D\}$ (où D signifie quelconque) à l'aide des opérateurs AND et OR en calculant pour

chaque noeud les quantités suivantes :

$$\mathbf{AND}(P,Vx) = \bigwedge_{i=1}^{14} (pi \wedge^* xi) \text{ où } x \wedge^* y = \begin{cases} 1 \text{ si } x=D \text{ ou } x=y \\ 0 \text{ sinon} \end{cases}$$

$$\mathbf{OR}(P,Vx) = \bigvee_{i=1}^{14} (pi \wedge^* xi) \text{ où } x \vee^* y = \begin{cases} 1 \text{ si } x=D \text{ et } x \neq y \\ 0 \text{ sinon} \end{cases}$$

Le choix judicieux des valeurs de P conduit à réaliser différentes formes de fonction d'appariement.

On utilise aussi la pyramide binaire pour le codage et la compression des images. Dans ce cas un noeud contient un 1 si la région vérifie une certaine propriété et un 0 sinon.

◆ Pyramides réelles : les algorithmes travaillent avec des approximations des valeurs d'images. Des opérations de type moyenne, somme, maximum et minimum en sont des exemples. Pour la moyenne, par exemple, chaque noeud calcule la moyenne arithmétique des valeurs provenant de ses fils et la code sur 8 bits. Après calcul par tous les noeuds, la moyenne de l'image entière se trouve dans la racine. La pyramide réelle est utilisée partout où on effectue des opérations réelles (par opposition à binaire) sur les valeurs des noeuds comme l'échantillonnage, la segmentation, etc...

◆ Pyramides d'objets : c'est une représentation tridimensionnelle des objets où chaque noeud représente, par exemple, les coordonnées d'un point dans l'espace d'observation.

γ) Classification par la stratégie d'analyse

Cette classification est fondée sur les structures de contrôle et les mouvements de données engendrés. Cela donne plusieurs types de stratégie de programmation pyramidale parmi lesquelles on peut citer :

- la construction pyramidale à l'aide de transformations binaires ou réelles qui se répètent sur tous les noeuds,

- l'utilisation de la pyramide comme structure d'arbre pour effectuer une recherche de valeurs pertinentes ou de pixels d'intérêt,

- la propagation qui décrit toutes les opérations de transmission ascendante, descendante et latérale des données et conduisant à des optimisations de certains problèmes séquentiels,

- la stratégie "diviser pour règner" où le traitement est réparti entre les noeuds de manière indépendante, ce qui conduit à une programmation souvent récursive donc concise et plus claire.

Nous donnons ci-après la liste des opérations que nous avons définies sur le type **PYRAMIDE** :

construction :

initpr : → **PYRAMIDE** :
crée une pyramide vide,
conspr : **PYRAMIDE x PYRAMIDE x PYRAMIDE x PYRAMIDE x NOEUD → PYRAMIDE** :
construit une pyramide à partir des quatre pyramides filles et de la racine,
numpyr : **PYRAMIDE → PYRAMIDE** :
affecte un numéro à chaque noeud en respectant l'ordre lexicographique.

consultation :

filspr : **PYRAMIDE x ENTIER → PYRAMIDE** :
donne la sous-pyramide fille désignée par son numéro,
valrac : **PYRAMIDE → INFO-PYR** :
donne la valeur de la racine,

image : **PYRAMIDE x ENTIER → IMAGE** :
donne l'image à un étage donné,
val : **PYRAMIDE x NOEUD → INFO-PYR** :
donne la valeur d'un noeud,

valp : **PYRAMIDE x NOEUD → INFO-PYR** :
donne la valeur du père,
nump : **PYRAMIDE x NOEUD → NOEUD** :
donne le numéro du père à partir d'un noeud,
valv : **PYRAMIDE x NOEUD x ENTIER → INFO-PYR** :
donne la valeur du voisin d'un noeud dans une direction donnée,

numv : **PYRAMIDE x NOEUD x ENTIER → NOEUD** :
donne le numéro du voisin d'un noeud dans une direction donnée,
valf : **PYRAMIDE x NOEUD x ENTIER → INFO-PYR** :
donne la valeur d'un fils désigné par son numéro,

numf : **PYRAMIDE x NOEUD x ENTIER → NOEUD** :
donne le numéro d'un des fils d'un noeud, désigné par son numéro d'ordre.

modification :

marquage : **(filtre) x PYRAMIDE x IMAGE → PYRAMIDE** :
construction de chemins d'accès à la base de la pyramide,
où : filtre : **NOEUD → BOOLEEN**

réduction : **(fcont) x PYRAMIDE x IMAGE → PYRAMIDE** :
réduction d'image à l'aide de la fonction fcont,
où : fcont : **IMAGE → IMAGE** :
expansion : **(fint) x PYRAMIDE x IMAGE → PYRAMIDE** :
expansion d'image à l'aide d'une fonction de projection ou d'interpolation,
où : fint : **IMAGE → IMAGE**

Appariement : **(fapp) x PYRAMIDE x INFO-PYR → PYRAMIDE** :
appariement de tous les noeuds avec un modèle,
où : fapp : **NOEUD x INFO-PYR → BOOLEEN**
modp : **(fmod) x PYRAMIDE x INFO-PYR → PYRAMIDE** :
modification des valeurs des noeuds de la pyramide,
où : fmod : **INFO-PYR x INFO-PYR → INFO-PYR**

La liste des opérations données ci-dessus n'est pas exhaustive. Elle donne seulement quelques exemples d'opérations de base utilisées sur la structure pyramidale.

Définition des opérations :

◇ **numpyr** : affecte à chaque noeud de la pyramide un numéro en respectant l'ordre lexicographique sur les noeuds. C'est une opération d'initialisation qui rend plus facile la manipulation de la pyramide. Comme nous l'avons dit précédemment, plusieurs opérations se définissent simplement à partir de cette numérotation.

◇ **valf** et **numf** : n'ont pas de sens sur les feuilles. Elles rendent une valeur indéfinie,

◇ **marquage** : l'opération de marquage construit des chemins d'accès à des points d'intérêt dans une image. Cette opération cherche d'abord les points d'intérêt dans l'image puis marque les noeuds pères qui y conduisent, ce qui permet d'accélérer les algorithmes de transformations d'ensembles de pixels.

◇ **réduction** : l'opération de réduction permet de réduire chaque étage à l'étage supérieur par une fonction fcont. La fonction fcont peut être la moyenne, le maximum, le minimum, etc...

Une autre représentation de l'image peut être obtenue en la considérant comme un ensemble d'objets liés entre eux. Une telle représentation peut s'exprimer à l'aide de **graphes**. Cette représentation est plutôt liée à des problèmes d'interprétation d'image et n'a donc que peu d'intérêt dans le cadre de cette spécification.

3.3. Les accès

Plusieurs transformations d'objets image s'effectuent en répétant une même opération élémentaire sur un ensemble de points représentant suivant le cas, une région, un élément de contour, une portion d'une fonction de transfert, etc... Dans le but de pouvoir exprimer globalement ces transformations à travers un langage de programmation et de bien formaliser les propriétés globales des objets, nous avons cherché à généraliser les fonctions d'accès aux objets étudiés et à les unifier autour de la notion de **filtre**.

3.3.1. Définition du filtre

Le filtre est un invariant associé à chaque type image. le filtre limite l'accès à un sous-ensemble des éléments de l'objet du type considéré.

Soit Obj un objet image et F un filtre, la sélection de points de Obj par le filtre F aura la forme syntaxique suivante :

Obj « F »

F est une fonction booléenne définie pour chaque point p de Obj.

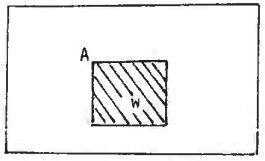
Dans APL [IVE 62], il existe plusieurs types de filtres pour accéder à des valeurs dans les tableaux, comme :

- **(v>0) / V** : permet de désigner tous les éléments positifs du vecteur V,

- **^/V ∈ {0,1}** : répond 0 ou 1 suivant que V est un vecteur logique ou non. Plus généralement cette notion existe dans les langages fonctionnels de type ML.

Dans L [RAD 81], Radhakrishnan utilise la notion de fenêtre pour localiser le traitement dans une image comme suit :

```
w := <3,4>
position(w,A,x,y)
A:w := v
```



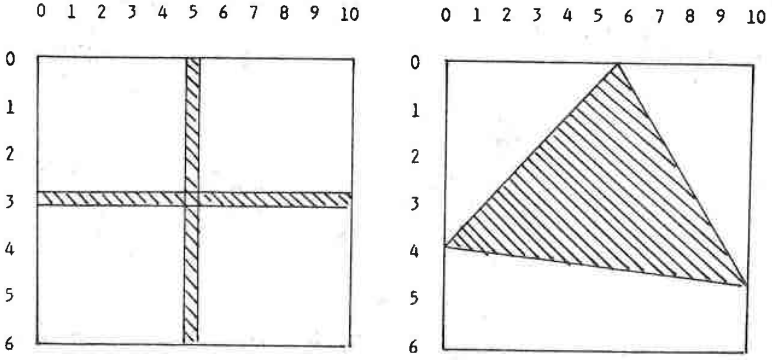
la fenêtre w de dimensions <3,4> est positionnée sur l'image A au point (x,y) et utilisée pour délimiter une portion de A dans laquelle on affecte la valeur v à tous les pixels.

Dans SAPIN, nous avons mis en évidence trois sortes de filtres : direct, relatif et associatif. Dans une image, le filtre peut sélectionner un pixel, une ligne, une colonne ou une région. Dans les structures linéaires, le filtre peut sélectionner un élément, une sous-liste, un ensemble d'éléments de la liste vérifiant une propriété (liée par exemple à la valeur de l'angle et à celle de la norme du gradient pour le contour). Dans les structures pyramidales, le filtre peut être le numéro d'un étage pour désigner une image, une sous-pyramide pour accéder de manière sélective à des pixels dans la base, etc...

De manière précise, le filtre est assimilée à une fonction d'accès appartenant à l'une des trois catégories suivantes :

- direct absolu : **«F1»**

F1 est une fonction d'accès direct à un ensemble de points. L'ensemble des coordonnés de points ou l'équation de la surface d'une région ou le rang d'un élément dans une liste ou enfin le numéro d'un noeud dans une pyramide en sont des exemples. Nous donnons dans la figure suivante deux exemples de ce type de filtre :



«y=3» ou «x=5»

«3y+2x-12>0 et 4y-5x+30>0 et 10y-x-40<0»

Figure 2.32 : Exemples de filtres d'accès direct à un tableau.

Dans l'exemple de gauche, on désigne globalement tous les points de la ligne 3 et ceux de la colonne 5. Dans l'exemple de droite, le filtre est composé de trois inéquations délimitant l'aire d'un triangle.

- direct translatable : **«F2»**

Pour réaliser l'accès direct translatable, on se sert d'objets image de type **IMAGE-BIN, REGION, CONTOUR** ou encore **FENETRE** pour accéder directement à des zones d'intérêt. Les objets filtre sont translatés ou positionnés sur l'objet à des endroits fixes. Les pixels d'une image filtrés par un contour seront ceux de la région de l'image délimitée par ce contour. Les pixels d'une image filtrés par une image binaire seront ceux de l'image dont les pixels homologues dans l'image binaire ont une valeur non nulle. Enfin les pixels filtrés par une fenêtre seront ceux appartenant à la portion de l'image encadrée par cette fenêtre.

exemple :

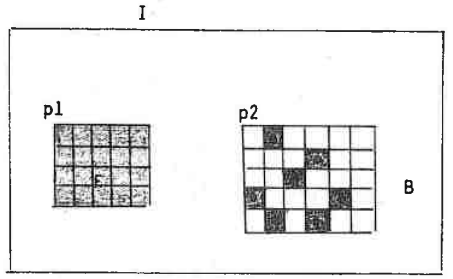


Figure 2.33 : Accès direct translatable à une image à l'aide d'une fenêtre F placée en p1 : I«F(p1)» et d'une image binaire B placée en p2 : I«B(p2)».

- associatif : **«F3»**

On accède à des éléments dont la valeur vérifie une certaine propriété. I«v > 5» permet d'accéder à tous les points de l'image I dont la valeur est supérieure à 5.

Dans le but de pouvoir désigner globalement des régions complexes, le filtre peut être construit par une combinaison des trois modes F1, F2 et F3 à l'aide d'opérations logiques **et** et **ou**. le **ou** exprime l'union de deux filtres permettant de pouvoir accéder à des régions disjointes caractérisées par des filtres disjoints. Le **et** permet de faire des intersections de filtres. Ainsi I««x-y<0» **et** «v>9 »» permet de sélectionner les pixels de l'image I dans la région désignée par «x-y<0» et dont la valeur est supérieure à 9.

Nous donnons dans le chapitre suivant les différentes formes syntaxiques du filtre. Nous allons voir maintenant comment il intervient dans la spécification des types image.

3.3.2. Utilisation du filtre

Toutes les opérations définies sur les types image peuvent être paramétrées par un filtre qui limite l'accès aux domaines de définition de ces opérations. Si l'on reprend, par exemple, l'opération de modification ponctuelle simple du type **GRILLE**, on peut la réécrire comme suit en tenant compte du filtrage :

modfpts : **(fps x filtre) x GRILLE → GRILLE**

où : filtre : **POINT → BOOLEEN**

qui sera définie comme suit :

```
g1,g2 : GRILLE,
p     : POINT,

modfpts(fps,filtre,g1) = <f(g2),t>

t = pourtout p dans t(g1) tq filtre(p): defg(g2,fps(p))
```

Ainsi, la fonction "filtre" est testée sur chaque point et autorise ou non sa modification par la fonction fps.

Les avantages du filtre sont nombreux. Il permet une simplification des expressions image avec une minimisation des erreurs syntaxiques. Le filtre peut être défini pour différents objets et servir comme paramètre général d'accès unifiant ainsi l'accès à des zones d'images. Enfin, l'écriture globale auquel ce filtrage peut conduire, facilite la détection d'opérations sur images et l'analyse de leur indépendance en vue d'une éventuelle parallélisation.

4. REPRESENTATION HIERARCHIQUE DES TYPES

4.1. Idées de base

La spécification formelle des types image effectuée précédemment a montré que ces types ont des structures de base communes et que certains types ont des opérations caractéristiques communes, d'où l'idée de les regrouper au sein de **classes**. Plus que le constructeur, chaque classe devient le moule général dans lequel seront créés les représentants ou instances. La déclaration d'un type image revient à décrire d'abord la classe à laquelle il appartient, à caractériser sa structure (constructeur,information) et son comportement (par la définition des opérations), puis à l'instancier. La hiérarchie suivante montre comment sont construites les classes de types image et comment elles sont instanciées.

4.2. Construction de la hiérarchie

Toutes les classes sont construites à partir des mêmes éléments (ou types) de base contenant le type de l'information INFO et ceux relatifs à la position (**POSITION**) et au point (**POINT**). Ensuite, pour chaque classe, un constructeur est désigné parmi **TABLE**, **GRILLE**, **LISTE**, et **ARBRE**. Les constructeurs sont définis chacun par une liste d'opérations caractéristiques paramétrées par le type ou les types d'information manipulés. Chaque constructeur ainsi défini permet de définir à son tour des classes de types plus proches des types image avec des opérations plus spécifiques et un type d'information instancié et donc plus précis.

Ainsi, **GRILLE** permet de définir les classes de types : **IMAGE**, **MASQUE** et IND_VIS (regroupant **REGION** et **POINTS_CONTOUR**). **HISTO**(GRAMME) et **LUT**(table_de_transfert) sont issus du type **TABLE** et **PYRAMIDE** du type **ARBRE**.

Enfin, au dernier niveau de la hiérarchie, apparaissent les types image. Ces types ont été munis d'un certain nombre de fonctions spécifiques. Le type de données est instancié par une structure de données concrète et les opérations de la classe sont complètement définies. Ainsi, du constructeur de types **IMAGE** sont dérivés **IMAGE_NVG** où le type de l'information est le niveau de gris (= intervalle d'entiers), **IMAGE_BIN** où l'information est binaire (= ensemble d'entiers {0,1}), **IMAGE_COL** où l'information est de type couleur (= <rouge : nvg , vert : nvg , bleu : nvg>), etc...

TYPES DE BASE

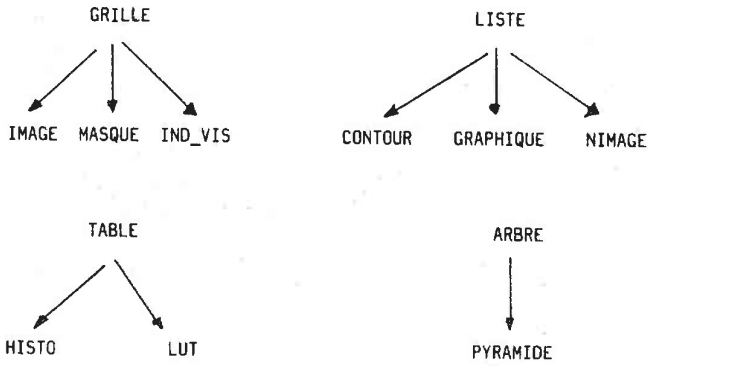
INFO, POSITION[Ix , Iy] , POINT[Ix,Iy,INFO], FENETRE[Ix , Iy]

CONSTRUCTEURS

GRILLE [Ix , Iy , INFO] LISTE [POINT] TABLE [Ix , INFO]

ARBRE [TYPE , INFO]

CLASSES DE TYPES



TYPES IMAGE =

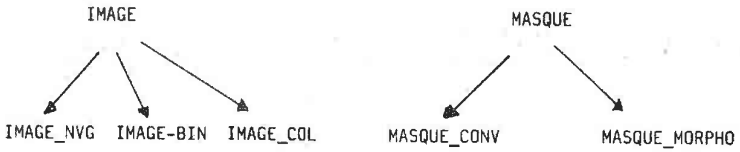


Figure 2.34 : Hiérarchie des types image.

4.3. Notions de généricité et d'héritage

L'une des particularités notables de la hiérarchisation des types est l'**héritage** par le type des propriétés du constructeur. Un objet de type **GRILLE**, par exemple, possède les propriétés du constructeur **GRILLE**.

Cette notion induit un nouveau style de programmation comme celui introduit par les langages orientés objets qui procèdent par affinage successifs de classes prédéfinies. L'écriture d'un programme dans ces conditions consiste à définir des sous-classes par instanciation de classes plus générales, c'est-à-dire augmenter les caractéristiques du langage en décrivant des classes plus spécifiques, plus adaptées au problème à résoudre.

L'autre particularité de la hiérarchisation est la **généricité.** Elle est liée à la paramétrisation des types qui constitue un mécanisme méthodologique puissant permettant de transporter des opérations et les propriétés caractéristiques associées. Elle se traduit à la programmation par la définition d'une bibliothèque de "constructeurs de types" utilisable par le programmeur et facilitant ainsi la définition de nouveaux types.

4.4. Exemple d'instanciation de la classe GRILLE

Au risque de nous répéter, nous allons reprendre la définition du constructeur **GRILLE** et montrer comment en dériver le type **IMAGE**. La dérivation se fait par instanciation des types et des opérations et par la définition d'opérations caractéristiques.

4.4.1. Instanciation des types

Le diagramme suivant montre l'instanciation des types de données du constructeur **GRILLE** :

GRILLE[Ix , Iy , INFO]

↓

IMAGE[Iix , Iiy , INFO_IM]

↓

IMAGE_NVG[1..512 , 1..512 , 0..255]

Les intervalles de variation des indices Ix et Iy ont été instanciés une première fois sur **IMAGE** par des intervalles d'entiers Iix

et Iiy puis une seconde fois sur **IMAGE_NVG** par des intervalles plus concrets : [1..512].

INFO, dans le type **GRILLE**, a été instancié une première fois par INFO_IM qui est le type des données d'**IMAGE**. INFO_IM est relatif aux codages du niveau de gris, à ceux de la couleur et de la distance, etc... INFO_IM est instancié ensuite par l'intervalle [0..255] précisant ainsi les variations du niveau de gris dans l'image **IMAGE_NVG**.

4.4.2. Instanciation des opérations

Nous montrons ci-après comment se fait l'instanciation des opérations dans le cas de la famille des images. Soit T un type, rappelons que OP(T) désigne les opérations de T. Si T' est une instance de T, nous noterons OP(T → T') l'instanciation dans T' des opérations de T.

GRILLE [Ix , Iy , INFO]

↓ OP(IMAGE) = OP(**GRILLE** → **IMAGE**) ∪
OP(**IMAGE**) ∪
OP(Ix → Iix) ∪ OP(Iy → Iiy) ∪
OP(INFO → INFO-IM)

IMAGE [Iix , Iiy , INFO_IM]

↓ OP(IMAGE_NVG) = OP(**GRILLE** → **IMAGE** → **IMAGE-NVG**) ∪
OP(**IMAGE-NVG**) ∪
OP(Ix → Iix → 1..512) ∪
OP(Iy → Iiy → 1..512) ∪
OP(INFO → INFO-IM → 0..255)

IMAGE_NVG [1..512,1..512.0..256]

le signe ↓ indique une dérivation ou induction. **GRILLE** → **IMAGE** indique qu'il y a instanciation des opérations du type **GRILLE** sur le type **IMAGE**. Ix → Iiy → 1..512 indique une double instanciation d'abord de Ix par Iix puis de Iix par 1..512. En d'autres termes, cela veut dire que les opérations sur l'intervalle 1..512 sont obtenues par instanciation de celles définies sur Iix, lesquelles sont obtenues par instanciation de celles définies sur Ix.

Nous allons montrer des exemples d'instanciation d'opérations sur la famille des types dont la classe est **IMAGE**.

Nous montrons d'abord quelques opérations sur le constructeur GRILLE en plaçant les types paramètres dans les profils :

construction :

creerg : **Ix x Iy → GRILLE[Ix,Iy,INFO]**

defg : **GRILLE[Ix,Iy,INFO] x POINT[Ix,Iy,INFO] → GRILLE[Ix,Iy,INFO]**

consultation : Les opérations suivantes complètent la liste des opérations de consultation associées au type **GRILLE**

accès : **GRILLE[Ix,Iy,INFO] x Ix x Iy → INFO** : accès direct à un point,

accèscond : **(filtre) x GRILLE[Ix,Iy,INFO] → INFO** : accès conditionnel moyennant un filtre,

où : filtre : **POINT[Ix,Iy,INFO] → BOOLEEN**

modification :

modfpts : **(fps) x GRILLE[Ix,Iy,INFO] → GRILLE[Ix,Iy,INFO]** : modification ponctuelle simple à l'aide de la fonction élémentaire fps,

où : fps : **POINT[Ix,Iy,INFO] → POINT[Ix,Iy,INFO]** : remplace un point par un autre,

modfptd : **(fpd) x GRILLE[Ix,Iy,INFO] x GRILLE[Ix,Iy,INFO] → GRILLE[Ix,Iy,INFO]** : opération sur deux grilles,

où : fpd : **POINT[Ix,Iy,INFO] x POINT[Ix,Iy,INFO] → POINT[Ix,Iy,INFO]**: calcule un nouveau point par modification de deux points homologues dans les grilles données.

Nous n'avons repris ici que quelques opérations de **GRILLE**. **GRILLE** hérite du type **TABLE** toutes les propriétés et opérations définies sur elle. D'ailleurs, la fonction defg est un renommage de la fonction **insert** définie sur **TABLE**.

Les opérations sur **IMAGE** sont :

construction

initim = creerg (Ix → Iix, Iy → Iiy, INFO → INFO-IM)
où Iix et Iiy = intervalles compacts d'entiers, i.e de pas = 1),

defim = defg (Ix → Iix, Iy → Iiy, INFO → INFO-IM)

consultation : héritage des opérations de consultation de **GRILLE** instanciées sur **IMAGE** plus :

accès = accès définie sur **GRILLE** avec :
accès(t(i),<x,y>) ≠ indéfini,
comparim : **(fcomp) x IMAGE[Iix,Iiy,INFO_IM] x IMAGE[Iix,Iiy,INFO_IM] → IMAGE[Iix,Iiy,INFO_IM]**

où : fcomp : **POINT[Iix,Iiy,INFO-IM] → POINT[Iix,Iiy,INFO-IM]**

dynamique: **IMAGE[Iix,Iiy,INFO-IM] → INFO-IM**

modification : héritage des opérations de modification de **GRILLE** instanciées sur **IMAGE** plus opérations nouvelles :

exemple d'instanciation :

seuillage : modfpts où fps est la fonction de seuillage qui annule tout point dont la valeur est inférieure à un seuil relatif déterminé à partir de la dynamique de chaque image,

Opérations nouvelles :

modvois : **(fvois x svois) x IMAGE[Iix,Iiy,INFO-IM] x MASQUE[Iix,Iiy,INFO-MS,ENTIER] → IMAGE[Iix,Iiy,INFO-IM]**

où : svois : **IMAGE[Iix,Iiy,INFO-IM] x POSITION[Iix,Iiy] x ENTIER x ENTIER → MASQUE[Iix,Iiy,INFO-MS,ENTIER]**

et : fvois : **MASQUE[Iix,Iiy,INFO-MS,ENTIER] x MASQUE[Iix,Iiy,INFO-MS,ENTIER] → INFO-IM**

zoomr : **(svois x spt) x IMAGE[Iix,Iiy,INFO-IM] x ENTIER x ENTIER → IMAGE[Iix,Iiy,INFO-IM]**

où : spt : **MASQUE[Iix,Iiy,INFO-MS,ENTIER] → INFO-IM**

zoome : **(fagr) x IMAGE[Iix,Iiy,INFO-IM] x ENTIER x ENTIER → IMAGE[Iix,Iiy,INFO-IM]**

où : fagr : **POINT[Iix,Iiy,INFO-IM] x ENTIER x ENTIER → MASQUE[Iix,Iiy,INFO-MS,ENTIER]**

Les opérations définies sur IMAGE_NVG sont :

construction :

opérations sur **GRILLE** et **IMAGE** instanciées,

consultation :

opérations sur **GRILLE** et **IMAGE** instanciées,

modification :

opérations sur **GRILLE** et **IMAGE** instanciées : ici, la convolution sera calculée par la somme des produits terme à terme du masque et du voisinage,

Les opérations définies sur **IMAGE_BIN** sont :

construction :

opérations sur **GRILLE** et **IMAGE** instanciées,

consultation :

opérations sur **GRILLE** et **IMAGE** instanciées,

modification :

opérations sur **GRILLE** et **IMAGE** instanciées : ici, la convolution sera remplacée par les opérations d'érosion et de dilatation.

5. CONCLUSION

A partir des définitions intuitives puis formelles des objets image les plus courants, nous avons spécifié quelques types image. La spécification a consisté à préciser pour chaque type, ses propriétés et l'ensemble de ses opérations caractéristiques. Les structures de données image ont servi à choisir les constructeurs de type de base. La plupart de ces constructeurs ont été eux-mêmes spécifiés dans [LEV 84], ce qui nous a simplifié la tâche de vérification de la consistance et de la redondance.

Dans une seconde étape, nous avons cherché à unifier les types image autour de la même fonction d'accès aux données. Ceci nous a conduit à définir le concept de filtre et permis de distinguer parmi les types image ceux permettant de définir des objets pouvant servir de filtre comme l'image binaire, la région, le contour et la fenêtre.

Enfin, dans le dessein de mieux résumer la démarche suivie, les types ainsi spécifiés, sont regroupés dans des classes. La hiérarchie bâtie sur cette classification a montré toutes les étapes de construction du type image à partir des éléments de base par choix du constructeur et par instanciation de ce constructeur.

L'annexe donne un exemple d'implantation de la classe **GRILLE** en ADA. Nous avons choisi ce langage pour ses grandes possibilités d'implantation de types abstraits avec toutes les facilités liées à la généricité et à l'héritage.

CHAPITRE 3 : DEFINITION DU LOGICIEL LPSI

1. INTRODUCTION

Nous présentons dans ce chapitre les fondements principaux du logiciel LPSI destiné essentiellement au développement de programmes de traitement d'images. Le lecteur peut trouver dans [BOU 87] une description plus détaillée et plus précise de ce logiciel.

LPSI est une extension du langage PASCAL. Il comprend à la fois des types image, des instructions adaptées à la programmation du traitement d'images et une bibliothèque de sous-programmes spécialisés. De cette manière, tout programmeur peut bénéficier des avantages qu'offre PASCAL tout en ayant la possibilité de développer facilement des applications particulières grâce aux primitives image introduites.

Nous avons choisi PASCAL parce que ce langage offre des facilités de structuration des données et des programmes et a été lui-même formellement spécifié. Il est facilement portable sur différents types de machine et s'est avéré, par ailleurs, une structure d'accueil intéressante pour la programmation image par Uhr [UHR 81] et pour la programmation graphique-image par Mallgren [MAL 82].

Les solutions apportées dans LPSI s'appuient sur un ensemble de concepts déjà vus au chapitre 1 pour lesquels on trouve différentes formes d'expressions dans les langages parallèles spécialisés. Il s'agit essentiellement :

(a) de se préoccuper d'*aspects ergonomiques* en fournissant à l'utilisateur différents niveaux de langage :

- primitives de suffisamment haut niveau qui peuvent être utilisées globalement en vue de réaliser des traitements en parallèle,

- modes de composition des primitives précédentes afin de permettre la définition de traitements conditionnels. Ceci a permis de mettre en évidence des concepts unificateurs de filtre.

- structures de contrôle globales facilitant la mise en oeuvre de fonctions itératives de recherche ou de traitement répétitif,

- variables contextuelles permettant de préciser certains choix liés à la configuration du système.

(b) de se préoccuper aussi de la réalisation effective du programme et son implantation sur des machines spécialisées. Cette étape propose des mécanismes qui permettent de produire du code optimisé pour la machine cible. Les ordres supplémentaires de LPSI sont écrits suivant la syntaxe PASCAL, ceux-ci sont traités avant la compilation à l'aide d'un préprocesseur écrit en PASCAL [BOU 87]. Cette forme d'implantation permet une grande souplesse de conception et un travail de traduction réduit.

Le but n'étant pas de construire un compilateur de plus, notre travail s'est surtout limité à rassembler dans LPSI un ensemble d'outils de programmation du traitement d'images en relation avec la spécification précédente des types. LPSI reste donc ouvert à toute forme d'extension comme par exemple une base de données ou des outils ergonomiques de construction de programmes.

2. ASPECTS SYNTAXIQUES DE LPSI

Un programme LPSI utilise une syntaxe similaire à celle de PASCAL. De ce fait il comprend la même structure qu'un programme PASCAL avec une partie déclaration et une partie instruction. A chacune de ces parties le pré-processeur associe la traduction correspondante en PASCAL.

2.1. Les types images

LPSI regroupe presque la totalité des types précédemment spécifiés et les classe dans deux familles :

- types communs,
- sélecteurs,

On distingue les sélecteurs des types communs du fait de leur utilisation aussi comme types d'objets filtre dans les accès conditionnels. Ces types sont : IMAGE_BIN, CONTOUR, FENETRE et PYRAMIDE. Ils peuvent par ailleurs servir à la déclaration d'autres objets filtre comme nous le verrons par la suite.

2.1.1. Déclaration

PASCAL ne permettant pas l'implantation directe de types abstraits, chaque type image sera donné par une définition dans la syntaxe LPSI et des opérations. Ces opérations sont implantées soit sous forme

d'opérateurs arithmétiques et logiques permettant d'écrire directement des expressions image, soit sous forme de sous-programmes intégrés dans la bibliothèque spécialisée du système d'environnement.

La déclaration précise d'une part les attributs relatifs à la définition du type tels ses dimensions, les caractéristiques relatives à sa forme ou le type de l'information et d'autre part l'environnement auquel les objets du type seront rattachés. Nous donnons dans la suite quelques exemples de déclaration :

a) Déclaration d'une image : **IMAGE**

```
type INFO-IM = ...
     ti = IMAGE [dimensions] OF INFO-IM;

var  i : ti [resident];
     j : ti [IN s(nbm)];
```

L'allocation de la mémoire centrale est précisée par un attribut d'environnement (*resident*) qui indique que l'image i sera résidente en mémoire et laissée sous le contrôle de l'utilisateur. Dans le cas où cet attribut est omis, l'image sera prise en charge par le système, c'est-à-dire rangée suivant les possibilités en mémoire image ou sur disque et gérée complètement par lui. L'attribut **IN** *s(nbm)* indique que l'image j se trouve sur un support externe dont les caractéristiques sont spécifiées dans s. nbm indique le nombre de blocs mémoire que l'utilisateur désire allouer à la gestion mémoire de sa variable.

b) Déclaration d'une liste d'images : **NIMAGE**

```
type tni = NIMAGE [dimensions] OF INFO-IM;

var  ni : tni [interlaced (type-entrelacement)];
```

Quand les images composant la liste d'images "ni" sont rangées de manière entrelacée, on indique le type d'entrelacement utilisé (entrelacement point par point ou ligne par ligne).

c) déclaration d'objets pyramide : **PYRAMIDE**

Ce type n'existe pas dans la version actuelle de LPSI. Sa déclaration ainsi que les opérations associées sont à considérer comme des exemples d'extensions futures de ce langage.

```
type tp = PYRAMIDE [dimensions-base][nb-etages] OF INFO-PYR;

var pyr : tp [connexion;] [resident];
```

où "dimensions-base" sont les dimensions de l'étage de base et "nb-

etages" est le nombre d'étages dans la pyramide. Une pyramide peut être privée de quelques-uns de ses étages supérieurs.

L'attribut "connexion" indique le type de réduction des étages dans la pyramide : réduction par deux conduisant à des connexions de type 1-père-2-fils ou réduction par quatre conduisant à des connexions de type 1-père-4-fils.

L'attribut "resident" permet d'indiquer, tout comme pour l'image, que l'objet pyr sera résident en mémoire centrale.

Tous ces attributs sont liés à la forme des objets et à l'environnement dans lequel ils seront implantés ; ils sont accessibles par le programmeur. L'accès aux valeurs de ces attributs se fait dans une syntaxe simplifiée en précisant le nom de l'objet suivi du nom de l'attribut recherché comme le montrent ces exemples :

---

i.cd : donne les dimensions (cadre) de l'image i,
ni.nb : précise le nombre d'images dans la liste ni,
pr.nb : indique le nombre d'étages dans la pyramide pr,
pr.cd : précise les dimensions de l'étage de base de pr...

Tous les attributs associés à la description des objets image ou à celle de leur type ont des valeurs par défaut et peuvent ne pas être précisés à la déclaration. Des constantes du système servent à définir ces valeurs par défaut.

Les objets image peuvent être eux-mêmes attributs d'autres objets image permettant ainsi d'évoluer avec eux au cours d'une session. Les objets "attributs" sont associés aux objets communs par une directive du langage et les accompagnent pendant toute leur durée de vie.

Exemple :

```
type ti : IMAGE[n,m] OF GL(8);
     tf : FTAB[256] OF integer;

var  i : ti;
     h : tf;

associate h : histogramme(i);
        ...
procedure histogramme(im:ti ; var histo : tf);
        ...
```

La procédure "histogramme" mettra à jour les valeurs de l'histogramme h associé à l'image i et ceci chaque fois que h sera

utilisé et i modifié au cours du programme.

2.1.2. Les filtres

Les filtres peuvent être appliqués sur tous les objets image quelque soit leur structure linéaire, surfacique ou hiérarchique. Les filtres permettent d'accèder à :

- un pixel dans une image à partir de sa position: **image-id[x,y]**,
- une ligne ou colonne d'image donnée par son rang : **image-id[x,]** ou **image-id[,y]**,
- une fenêtre f de pixels : **image-id«f(p)»**,
- un voisinage d'un pixel : **pvois(image-id,x,y,h,l)**, où (x,y) sont les coordonnées du pixel et (h,l) sont les dimensions du voisinage,
- un élément de contour donné par son rang : **contour-id[r]**,
- la racine ou l'étage d'une pyramide : **pyram-id[num]**,
- etc...

Un filtre peut être construit de différentes manières comme suit :

A) Par initialisation

a) à des positions connues :

◆ - dans une image binaire :

F := **B«f»**

seuls les points du binaire B vus à travers le filtre f sont pris en compte.

◆ - dans une région :

F := **R[r]**

seuls les points de la sous-région r de R sont considérés.

◆ - à l'intérieur d'un cadre :

F := **[haut,larg]**

initialise le filtre à une fenêtre de hauteur = haut et de largeur = larg et dont l'origine sera connue à l'application.

◆ - à un ensemble donné de positions :

F := (p0, ..., pi, ..., pn)

initialise le filtre à l'ensemble des positions pi (i=1,..,n).

b) à une expression filtre :

Pour les objets surfaciques de type IMAGE, MASQUE et REGION, le calcul des positions utiles peut se faire à l'aide d'une expression de deux variables génériques l$ (ligne) et c$ (colonne) qui détermine la surface à laquelle appartiennent les points du filtre. Nous donnons dans la suite un exemple d'un tel filtre :

F := l$ - c$

F désigne tous les points d'un objet planaire quel qu'il soit mais dont les positions sont au-dessus de la diagonale.

Pour les objets linéaires, on utilise une seule variable générique r$ qui désigne le rang de l'élément comme suit :

F := r$ > 5

F désignera tous les éléments de rang supérieur à 5.

Pour les objets de type pyramide, on peut aussi utiliser une expression en l$ et c$ qui sera valable pour tous les étages. De plus, on peut utiliser la variable générique n$ qui permettra de faire des sélections de noeuds par le numéro.

c) par sélection :

Le filtre peut être obtenu à partir d'une sélection dans un objet à l'aide d'un autre filtre. La fonction permettant cette nouvelle construction du filtre s'appelle **MAP**. *Elle s'applique comme suit :*

F := MAP(I«F'»)

Le filtre F' est d'abord appliqué sur I, ensuite, la fonction MAP range dans F les positions des points de I vérifiant F'. Si F est de type pyramide, alors la fonction MAP marque tous les chemins de la pyramide qui conduisent aux points de l'image (située à l'étage de base) vérifiant F'.

B) Par composition de plusieurs filtres

On utilise dans ce cas des connecteurs logiques **and** (pour l'intersection) et **or** (pour l'union). Ainsi :

F := W1 **and** W2

construit F par intersection des deux fenêtres W1 et W2 supposées avoir le même le coin supérieur gauche,

F := W1 **or** W2

construit F par union des mêmes W1 et W2,

F := B **and** W(p)

F est composé des points du binaire B vus à travers la fenêtre W placée sur B au point p, cette écriture est équivalente à :

F := B«W(p)»

F := (l$ + c$ ≥0) **and** W(p) **and** (v$ ≥ 10) ;

sélectionne les points au-dessus de la diagonale, compris dans la fenêtre W appliquée au point p et dont les valeurs sont supérieures à 10. V$ est la variable générique utilisée pour l'accès associatif.

Nous pouvons remarquer à travers le dernier exemple le caractère général du filtre. En effet, les trois composantes du filtre déterminent des positions de manière relative au cadre de chaque objet sur lequel le filtre sera appliqué.

2.2. Les instructions

On peut distinguer dans LPSI cinq familles d'instructions :

- les instructions d'affectation,
- les instructions de contrôle,
- les instructions de reconfiguration du système,
- les instructions de gestion du parallélisme,
- les instructions d'entrée/sortie.

2.2.1. Les instructions d'affectation

La syntaxe des instructions d'affectation a la forme suivante :

<affectation> ::= <identificateur> [<filtre>] **<opa>** <expression>

où **<opa>** est l'opérateur d'affectation classique et <expression> est soit un identificateur, soit une expression arithmétique ou logique composée d'objets image et d'opérateurs. L'expression est globale, c'est-à-dire que l'opération n'est pas explicitée par une itération sur tous les points des opérandes du fait de son homogénéité. Parmi les expressions les plus courantes, nous notons :

- les combinaisons linéaires d'images,
- les comparaisons logiques d'images,
- les convolutions par masque,
- les filtrages morphologiques.

Le tableau suivant donne quelques exemples d'opérateurs arithmétiques, morphologiques et logiques appliqués sur les objets image :

opérateur	expression	sémantique
Opérations arithmétiques		
opa= + - / *	A **opa** B*	a_r **opa** b_r $\forall a_r \in A$ et $b_r \in B$
Opérations de filtrage		
opf= ⊗ ⊖ ⊕ ⊞	A **opf** m	pvois(A,a,cadre(m)) **opf** m $\forall a \in A$
Opérations de comparaison		
opc= < > = ≧ ≦ ≠	A **opc** B*	vrai si a_r **opc** b_r $\forall a_r \in A$ et $b_r \in B$

B* indique que B peut être une constante ; dans ce cas, B est considéré comme un objet image dont toutes les valeurs sont égales à cette constante. a_r désigne l'élément de rang r dans l'objet A. b_r est l'élément homologue dans B. Dans les opérations de filtrage, ⊗ désigne la convolution, ⊖ l'érosion, ⊕ la dilatation et ⊞ le tout ou rien ; m est le masque de convolution pour le premier opérateur et l'élément

structurant pour les autres. Les opérations de comparaison logique n'ont de sens que sur des objets comparables comme les masques ou les images de niveaux de gris.

L'affectation agit comme les opérateurs binaires sur des éléments de même rang ou position. Il est donc primordial que l'identificateur et l'expression aient les mêmes dimensions. Le caractère isotrope de l'affectation permet d'envisager son exécution en parallèle sur tous les points.

Exemples :

A := B

Cette opération affecte l'image B à l'image A. A et B sont de même type IMAGE et ont donc les mêmes dimensions.

A := B«W(p)»

Ici, on n'affecte à A qu'une fenêtre de B de mêmes dimensions que A. L'affectation se fait entre les points situés à égales distances des coins supérieurs gauches des cadres de A et celui de W(p). Ceci reste valable pour tous les objets surfaciques.

A«w1(p1)>> := B«w2(p2)>>

Cet exemple montre que l'on peut aussi restreindre l'accès à l'identificateur (à gauche du signe d'affectation) à l'aide d'un filtre. Il s'agit, dans cet exemple, d'affecter une partie de l'image B désignée par la fenêtre w2 placée au point p2 à seulement une partie de A désignée par la fenêtre w1 placée au point p1.

objet«f» := cste ou ø

Le terme de droite est ici une constante. On l'assimile à un objet de même type que l'objet de gauche ayant pour tous les points la même valeur : cste ou ø. La valeur ø n'a de sens que pour les objets de type chaîne où elle a pour effet de supprimer les points inutiles.

L'affectation conduit parfois à des conversions de types comme nous l'avons vu pour les filtres. Un autre exemple est celui de l'affectation d'image à pyramide :

pyr := i ou **pyr[k] := i**

Dans l'expression de gauche, l'image i est affectée à la base de la pyramide et à son kème étage dans l'expression de droite. Ceci montre qu'il y a équivalence dans LPSI entre les étages des pyramides et les

images.

2.2.2. Les instructions de contrôle

LPSI fournit toutes les instructions de contrôle de PASCAL. Nous décrivons dans la suite d'autres instructions qui sont plus spécifiques à la programmation parallèle des images :

A) La conditionnelle globale

Cette conditionnelle est dite globale car elle porte sur des ensembles d'éléments (essentiellement des comparaisons de tableaux). La condition ne sera à vrai que lorsque toutes les conditions élémentaires sont toutes à vrai. Par exemple, dans l'instruction suivante :

IFG I1 > I2 **THENG** traitement1 **ELSEG** traitement2;

traitement1 ne sera exécuté que si tout point p(x,y) de l'image I1 a une valeur supérieure à celle de son homologue p'(x,y) dans I2, sinon c'est traitement2 qui sera exécuté.

B) L'itération globale

L'itération classique consiste à parcourir les images ligne par ligne et à répéter le traitement sur chaque point de la ligne en cours. Dans LPSI, on simplifie la syntaxe classique en ne précisant que ce qui est essentiel au déroulement de l'itération, c'est-à-dire l'itérateur, le point initial, la condition d'arrêt et le module de traitement. Là encore, il s'agit de mettre en valeur la structure de contrôle dans le but de permettre sa détection rapidement par le préprocesseur.

FORALL p **IN** objet [**FROM** p0] [**WHILE** condition] **DO** ...

où p est un élément de objet qui peut être une position, un point, une ligne, une colonne, un voisinage, ou toute forme que l'on peut décrire à l'aide des types image précédemment spécifiés. po est l'élément initial à partir duquel commence l'itération. L'introduction de condition permet de réaliser d'une autre manière le while de PASCAL.

Dans plusieurs problèmes séquentiels, la poursuite du traitement oblige à faire un choix parmi les éléments suivants à prendre (points suivants dans les problèmes de suivi de contour, par exemple) ; dans ce cas, c'est *condition* qui permettra de faire ce choix.

FORSEQ p **IN** objet [**FROM** p0] [**maximizing** fonction]
[**WHILE** condition] **DO** ...

Un autre problème est lié au choix du point de départ. Nous introduisons dans la structure de contrôle une fonction maximisant un critère de choix dans un voisinge où tous les points sont candidats.

2.2.3. Les instructions de contrôle pyramidales

Ces instructions réalisent la transmission des données à travers le réseau de noeuds en place. Suivant le type d'action demandé, la transmission peut être latérale, verticale ascendante ou verticale descendante. Nous distinguons trois formes de contrôle global comme suit :

A) Réduction ascendante des données

Le même traitement est appliqué sur les fils et transmis à leur père commun. La syntaxe de cette instruction est :

FORALL noeud **IN** pyramide«filtre» **REDUCE_BY** traitement

où "noeud" peut aussi désigner un voisinage de noeuds dont il faut préciser la taille. Seuls les points désignés par le filtre sont accessibles dans la pyramide.

B) Traitement des données

Cette opération inclut tous les traitements répétitifs effectués identiquement par les processeurs de la pyramide sans transmission explicite des données.

FORALL noeud **IN** pyramide«filtre» **DO** traitement;

Les noeuds considérés seront seulement ceux qui sont accessibles par le filtre.

C) Projection des données

La même donnée est affectée à l'image de base mais seulement aux points dont les chemins supérieurs dans la pyramide ont été marqués dans une étape précédente de filtrage. On utilise la fonction project pour effectuer la transmission de la donnée depuis la racine jusqu'aux points d'intérêt placés dans la base.

pyramide := **project**(donnée)

2.2.4. Les instructions de gestion du parallélisme

LPSI offre une possibilité de traitements parallèles, c'est-à-dire qu'il permet à l'utilisateur de subdiviser son traitement en tâches capables de se dérouler simultanément. Lorsque ces tâches se déroulent en parallèle, il y a des conditions de synchronisation entre elles et aussi des conditions d'exclusion mutuelle lorsqu'elles se partagent des données. LPSI n'offre pas les outils nécessaires pour exprimer ces conditions. C'est à l'utilisateur de faire de telle sorte que ces conditions soient gérées judicieusement par le système. Des règles simples sont mises en place pour résoudre les conflits. La syntaxe de ces instructions est :

COBEGIN T1 // T2 // ... // Tn **COEND**;

où Ti (i=1,..,n) sont les tâches exécutées en parallèle.

Là encore, il ne s'agit pas d'exprimer parfaitement le parallèlisme pour son exécution sur une machine particulière mais de le mettre seulement en évidence. Reste au préprocesseur la charge de le gérer et de l'implanter judicieusement sur chaque machine cible.

Cette instruction n'a pas été implémentée dans la version actuelle de LPSI par manque de temps.

2.2.5. Les instructions d'entrée/sortie

Les ordres d'entrée/sortie permettent le transfert des objets image depuis les périphériques vers la mémoire centrale et inversement. Le périphérique peut être le disque ou la bande, mais il peut aussi être la mémoire image ou un étage de la pyramide cible. La syntaxe de ces instructions est :

READG ou **WRITEG** (support) lvariable;

où :

- support : nom d'une variable définie à l'aide de la directive

support$ qui précise les caractéristiques du support externe, telles son numéro, la taille de l'enregistrement, le type d'entrelacement s'il s'agit d'images, le nombre de colonnes et de lignes du support, etc.... Pour la désignation du support, nous nous sommes inspiré d'un logiciel d'interface appelé IMAGE 8 [VOG 84] construit par l'ESNPS pour programmer la machine ICOTECH.

- lvariable : liste des variables à lire ou à écrire. Ces variables peuvent être filtrées ou non.

Cette généralisation des entrées/sorties permet de résoudre en partie des problèmes relatifs à la portabilité du logiciel.

2.2.6. Les instructions de reconfiguration du système

Tout programme LPSI se déroule dans un environnement de données prédéfinies. Les définitions de ces données sont regroupées dans une table du système appelée CONTEX et tout se passe automatiquement comme si cette table était intégrée au programme. L'utilisateur peut modifier ces données et définir ainsi le contexte dans lequel il souhaite aborder une session de travail. Les valeurs par défaut interviennent dans la déclaration des types (dimensions des images, type des valeurs dans une image binaire, etc...), dans la désignation des périphériques (numéro du périphérique, format des enregistrements, mode d'accès, etc...), dans le choix des couleurs de sortie pour une image, etc.... Toutes les variables du système contenant ces valeurs seront suivies d'un $. La forme syntaxique de ces instructions est :

set_contex identificateur$ = valeur [, identificateur$ = valeur]*

Les identificateurs sont connus du système et n'ont pas besoin d'être déclarés.

2.3. La bibliothèque

LPSI fournit un ensemble d'utilitaires pour exécuter des tâches spécialisées. Ils sont regroupés en partie dans une bibliothèque de service accessible par l'utilisateur dans le programme source. Ils sont relatifs aux tâches de traitement d'images définies dans le chapitre précédent.

3. EXEMPLES DE PROGRAMMES LPSI

Nous allons donner dans la suite quelques exemples de programmes écrits dans la syntaxe de LPSI :

3.1. Détection de contour

Dans cet exemple, nous allons exprimer des tâches indépendantes qui seront exécutées simultanément. Il s'agit d'extraire les contours dans une portion f de l'image i suivant les actions suivantes :

- convolution de i à l'aide d'un masque m,
- binarisation du résultat par seuillage à l'aide du seuil S ; le résultat est rangé dans C1,
- égalisation d'histogramme dans i donnant i2 comme résultat,
- différence entre i et i2 puis application du même seuillage que précédemment pour obtenir C2,
- calcul de l'image de contour résultat par union de C1 et C2.

La fonction d'extraction de contour s'écrit comme suit :

```
fonction extcont(i : ti ; f : fenetre) : ti;
        ...
        begin

            readg (support1) i; (* lecture de i sur support1*)
            readg (support2) m; (* lecture de m sur support2*)
            j := i; (* duplication de i *)

            (* convolution suivie du seuillage *)
                C1 := (i«f» ⊗ m)«v$ > S»
            (* calcul d'histogramme *)
                h := forall v$ in j«f» do h[v$] := h[v$]+1;
            (* calcul des fréquences de niveaux de gris cumulées *)
                for k:=2 to h.L do h[k] := h[k] + h[k-1];
            (* égalisation d'histogramme *)
                forall p in j«f» do i2[p] := h[j[p]];
            (* différence suivie de seuillage *)
                C2 := (i2 - j«f»)«v$ > S»
            (* union des deux images de contour *)
                extcont := C1 + C2
        end;
```

Ce sous-programme contient deux tâches indépendantes pouvant se dérouler en parallèle. La première exprime une convolution suivie d'un seuillage. Ces deux opérations sont locales et indépendantes d'un pixel à l'autre, c'est pourquoi nous les avons exprimées de manière parallèle.

La deuxième tâche est composée de trois actions : la première calcule l'histogramme h puis les fréquences cumulées de l'histogramme. La seconde effectue la différence entre l'image originale et l'image précédemment égalisée et accompagne ce traitement par le même seuillage que précédemment. Enfin, la troisième action effectue l'union des images de contour C1 et C2 obtenues.

3.2. Calcul de moyenne

Dans cet exemple, il s'agit de calculer la moyenne d'une image de niveaux de gris. Nous donnons, dans la suite, deux versions pour ce calcul :

En considérant l'image dans une pyramide

```
fonction moyenne(pyr : pyramide) : GL; (* GL = niveau de gris *)

    fonction m(p1,p2,p3,p4 : pixel) : GL;
      begin
            m := (val(p1) + val(p2) + val(p3) + val(p4))/4
      end;
    begin

        forall qd(4) in pyr reduce_by m(qd.1,qd.2,qd.3,qd.4)

        (* forall qd(4) : pour tous les voisinages de 4 points *)
        (* qd.i : ième pixel dans ce voisinage *)

        moyenne := val(pyr.r)

    end;
```

En la calculant directement dans l'image

```
function moyenne( i : image ) : GL;

var x,y : integer;
        p1,p2,p3,p4 : position;
        f : fenêtre;

    begin

        if (f.haut > 1) and (f.larg > 1) then

                y := (i.cd.haut+1)/2 ; x := (i.cd.larg+1)/2;
                p1 := <0,0> ; p2 := <0,x> ; p3 := <y,0> ;
                p4 := <y,x>;
                f := [y-1,x-1];
                m1 := moyenne(i«f(p1)»); m2 := moyenne(i«f(p2)»);
                m3 := moyenne(i«f(p3)»); m4 := moyenne(i«f(p4)»);
                moyenne := (m1 + m2 + m3 + m4) / 4

        else  moyenne := val(i)

    end;
```

4. PHASE DE REALISATION

La phase de réalisation a permis d'étudier le passage de LPSI en PASCAL en résolvant certains problèmes classiques liés à l'implantation des types, à la traduction des opérations globales et au passage des paramètres. D'autres problèmes système ont été également pris en considération lors de cette étape. Il s'agit essentiellement de la gestion des variables et des ressources en place. Nous allons donner dans la suite des solutions à certains des problèmes mentionnés ci-dessus.

4.1. Implantation des objets

Le problème à résoudre ici était d'abord de trouver la structure de donnée PASCAL la plus proche de la définition de chaque type permettant des traductions immédiates des opérations, ensuite de pouvoir lui associer des attributs et des filtres pour l'accès conditionnel.

Pour l'association des attributs, on cherche une structure de données souple permettant d'une part de bien décrire les relations

entre objets et objets associés et d'autre part d'effectuer facilement des mises à jour en cours d'exécution. Pour le filtrage, le problème se situe surtout au passage des paramètres. Nous tenons, en effet, à ce que les procédures acceptent indifféremment comme paramètres des images entières ou des parties de celles-ci. Or PASCAL n'autorise pas de manipuler dans les procédures des objets à dimensions variables.

4.1.1. Implantation du type IMAGE

D'après les remarques précédentes qui ont montré l'insuffisance du tableau pour représenter une image, le type IMAGE sera décrit par une structure à quatre champs comme suit :

Le champ 1 précise ses attributs implicites tels que ses dimensions et la taille de l'information.

```
champ1        = record
                  hauteur, largeur : integer;
                  taille-info      : integer
                end;
```

Le champ 2 précise la nature du lien suivant que l'image est résidente (lien avec un tableau) ou virtuelle (lien avec le descripteur du support).

```
champ2        = ^ environnement;

environnement = record
                  case etat : env of
                  resident  : ( imager :  ^ tableau );
                  virtuel   : ( imagev :  ^ desc-supp)
                end;
```

Le champ 3 précise les liens de l'objet avec ses attributs associés.

```
champ3        =  ^ list-ind;

list-ind      = record
                  indicateur : boolean;
                  indic-suiv : ^ list-ind
                end;
```

Le champ 4 indique si l'image est elle-même attribut associé et, précise dans ce cas le lien d'association.

```
champ4        =  ^ indicateur;
```

exemple :

```
.
.
var
  I, J : IMAGE  l, m  OF GL(n) RESIDENT;
.
.
associate
  A : histogramme (I);
  B : dynamique   (I);
  I : negatif     (J); % exemple où l'image I est elle-même
                         attribut associé %
```

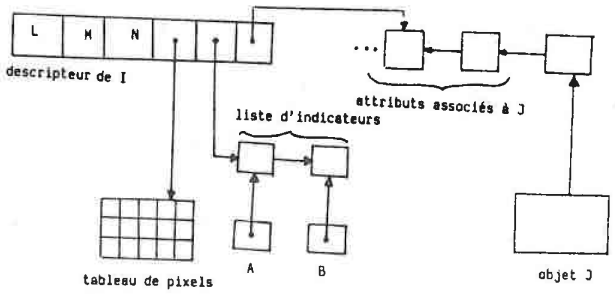


Figure 3.1 : Schéma de la représentation interne d'un objet image

La figure 3.1 donne les détails de la représentation de l'image I de l'exemple précédent :

Cette structure permet de résoudre les problèmes signalés précédemment. On a rattaché le descripteur à l'objet image dans le but de pouvoir connaître à l'intérieur des procédures les dimensions de l'image. La représentation des attributs associés par une liste d'indicateurs permet de les utiliser comme attributs d'autres objets. Toute modification de l'objet image s'accompagne d'une mise à jour dans la liste des indicateurs. On ne recalcule un attribut que lorsque l'objet auquel il est associé a été modifié. Cette information est fournie au système par la valeur de l'indicateur qu'il repère.

4.1.2. Passage des paramètres filtre

Un objet filtré est passé en paramètre sous la forme d'une structure à deux champs de type pointeur. Le premier repère la structure de l'objet et le second celle du filtre éventuel.

Afin de résoudre les problèmes liés aux contrôles des types entre les paramètres effectifs et les paramètres formels, on utilise la fonction ADDR de PASCAL qui retourne l'adresse de son paramètre et qui est compatible avec les types pointeurs. De cette manière, tout appel à une procédure LPSI se traduit par :

- champ1 := ADDR(structure-objet);
- champ2 := ADDR(structure-filtre);

Ainsi, on résoud de manière définitive le problème des paramètres de dimensions variables.

4.1.3. Traduction des opérations

Le préprocesseur de LPSI détecte les opérations image et les traduit en PASCAL. A chaque type d'opération LPSI correspond un schéma de traduction général qui peut être optimisé en fonction de l'opération. Ceci est obtenu à partir de quelques règles d'optimisation ad hoc comme suit : pour une expression donnée A := λ1 I1 + λ2 I2«F», l'optimisation permet de déterminer le cadre minimal par rapport auquel on opère et de décrire la combinaison linéaire dans ce cadre uniquement.

La boucle FORALL introduit des traitements répétitifs indépendants avec la possibilité de privilégier un parcours (ligne par ligne ou colonne par colonne). Une implémentation parallèle de cette boucle est tout indiqué. Si la machine contient un processeur matriciel et si le parcours n'est pas spécifié, tous les points peuvent être traités simultanément par le processeur matriciel.

5. CONCLUSION

Nous avons présenté dans ce chapitre un résumé de la syntaxe du langage LPSI ainsi qu'un certain nombre de solutions liées à l'implantation. LPSI contient des types image et des instructions spécialisées. La déclaration des variables image précise, outre la structure de l'objet, deux familles d'attributs, les uns relatifs à l'environnement et les autres aux relations avec les autres objets. Pour les opérations, LPSI propose des opérateurs globaux permettant de manipuler ces objets de manière aisée assurant, dans une phase

d'optimisation, une meilleure prise en compte du parallélisme.

La phase d'implantation propose des schémas de traduction qui permettent de produire du code optimisé pour la machine cible. LPSI comporte, par ailleurs, un environnement de traitement d'images avec tous les outils nécessaires à la gestion des données et des ressources en place.

Ce logiciel a été implanté sur une SM90 sur laquelle est connecté le système de traitement d'image VIDEOGRAPH. Des tests réels ont été faits sur ce matériel afin de valider les différents choix d'implantation.

CHAPITRE 4 : DEFINITION DU LOGICIEL SAPIN-NI

1. INTRODUCTION

Les systèmes spécialisés en traitement d'images connaissent deux types d'utilisateurs : l'informaticien qui utilise tous les moyens de la machine pour développer, créer et optimiser des programmes en cherchant à les rendre les plus performants possible et le non-informaticien qui, à l'opposé du premier, désire traiter ses images avec beaucoup de facilités et obtenir des résultats pour son application sans avoir à se préoccuper de toute forme de traitement informatique.

Ce dernier type d'utilisateur a été longtemps victime du progrès dans ce domaine et contraint à s'adapter à la machine. Les seules formes d'aide possibles étaient traduites sous forme de bibliothèques spécialisées, mais souvent l'utilisateur était obligé d'écrire les programmes FORTRAN qui les appellent.

Dans le cadre du logiciel SAPIN-NI, nous avons travaillé au développement d'outils permettant plus facilement le développement de programmes d'applications utiles pour les deux types d'utilisateurs mais aussi à l'écriture d'un système d'utilisation interactive.

Le logiciel écrit étant destiné à la programmation sur une machine spécialisée, sa tâche était donc de se charger aussi de la gestion des ressources de cette machine et de faire profiter l'utilisateur des moyens disponibles : mémoire image-graphique, processeurs spécialisés dans l'affichage et le traitement, archives d'images, etc...

SAPIN-NI est un langage de dialogue à base de menu où sont rangées et explicitées en termes clairs les fonctions du système et leur mode d'emploi. Les objectifs du système sont de permettre :

- un affichage clair des fonctions du menu par classe d'application (sous-menu),

- une mise à jour simple par adjonction de nouvelles fonctions et suppression des anciennes,

- une représentation claire des paramètres des fonctions et un contrôle automatique de leurs valeurs,

- une aide assurée à tous les niveaux par utilisation d'une fonction de secours **HELP**,

- une construction de macros par enchaînement des fonctions du menu,

- de disposer d'une trace mémorisant les traitements effectifs et de s'en servir pour rejouer certaines séquences et construire automatiqement des macro-commandes,

- de connaître à tout moment l'état du système (occupation mémoire, périphériques connectés, informations accessibles, ...),

- de manipuler les données image et graphique en tant qu'objets globaux et de ne pas se préoccuper de la représentation machine.

2. LE MENU PERSONNALISE

Le système-logiciel SAPIN-NI comprend essentiellement un **menu** regroupant de manière hiérarchisée les fonctions de traitement disponibles. Parmi ces fonctions, nous trouvons des sous-programmes de traitement spécifiques à l'application développée, des commandes destinées au traitement commun (acquisition, visualisation, archivage, ...) et des commandes d'aide à l'utilisation du système. Le système comprend, au départ, un noyau de fonctions de base nécessaires à une première prise en contact avec la machine et suffisantes pour faire les premiers traitements. Chaque utilisateur peut compléter ce menu de base par des fonctions personnelles liées à l'application qu'il veut développer. Les paragraphes suivants montrent l'organisation générale du menu et expliquent les modes d'enrichissement et d'utilisation.

2.1. Organisation du menu

Les commandes du système sont organisées en rubriques et fonctions dont l'accès peut être protégé. Chaque rubrique désigne un ensemble de rubriques secondaires ou fonctions. Contrairement à la rubrique, la fonction se place à une feuille de l'arbre du menu et est exécutable. On distingue quatre types de fonction :

2.1.1. Les fonctions de service

Ces fonctions modifient l'état du système. On distingue :

- Les fonctions de mise à jour de l'arbre du menu,

- les fonctions de mise à jour de l'environnement du système (comme

les objets partageables entre les fonctions),

- les fonctions de modification de zones d'interface des commandes du menu.

2.1.2. Les fonctions banalisées

Ce sont celles qui sont communément utilisées comme les fonctions d'acquisition et de visualisation. Ces fonctions, du fait de leur utilisation universelle et fréquente, seront visibles à n'importe quel endroit de l'arbre.

2.1.3. Les fonctions d'aide

Ces fonctions permettent la consultation du système sans modifier les valeurs des objets manipulés comme celles qui permettent le déplacement dans l'arbre du menu ou celles qui donnent accès à des connaissances sur l'état du système ou au commentaire de chaque fonction et expliquant son utilisation.

2.1.4. Les fonctions utilisateurs

Elles sont de trois types :

A) Les programmes : les programmes sont écrits dans le langage hôte en dehors du système ; ils y sont adjoints par une commande qui assure la compilation et l'édition de liens.

B) Les macro-commandes : ce sont des modules comprenant des enchaînements de fonctions du menu lui-même. Cela suppose que nous disposons d'un langage d'écriture de macros. La version actuelle de SAPIN-NI ne comprend que des macro-commandes linéaires.

C) Les rubriques exécutables : il existe un troisième type de fonction exécutable rattachée à des rubriques du menu. L'appel d'une rubrique provoque, pour certaines, l'exécution d'une fonction de préparation du système comme, par exemple, la mise en route des périphériques.

2.2. Utilisation du menu

A l'appel du système, celui-ci affiche un menu général sur le terminal de service et un dialogue s'instaure entre la machine et l'utilisateur. Pour travailler, l'utilisateur sélectionne une commande qui met le système dans un certain état de préparation de la fonction : le système charge le code du programme correspondant et demande à l'utilisateur de fournir les paramètres nécessaires à l'exécution de la fonction. Après chaque appel de fonction, une trace visionne sur l'écran la succession des traitements effectués dans la session en cours. La figure 4.1 donne le schéma fonctionnel du système :

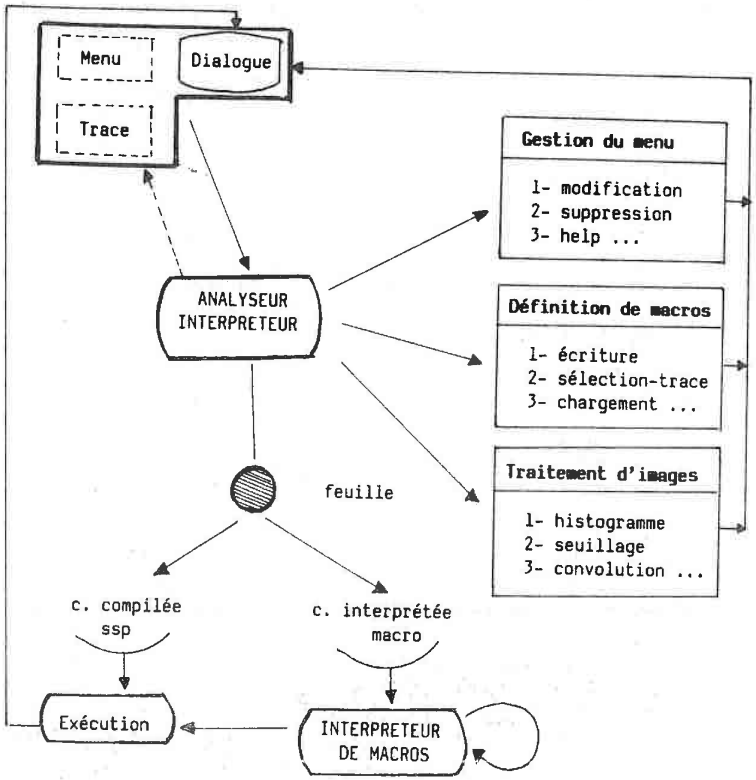


Figure 4.1 : Schéma fonctionnel du système SAPIN-NI.

2.2.1. Appel d'une commande

L'appel d'une commande pour l'exécution d'une fonction du menu, par exemple, se fait en précisant son nom et la liste éventuelle de ses paramètres. Le nom sera celui précisé par la rubrique englobante ou le nom absolu comportant le nom de la fonction précédé par les noms de toutes les rubriques englobantes. L'introduction des paramètres de la fonction peut se faire dans deux modes : le mode bavard (ou conversationnel) et le mode muet (ou fonctionnel).

Exemple :

Il s'agit d'exécuter la fonction de seuillage appelée SEUILLAGE dans la rubrique en cours. Cette fonction opère sur une image de niveaux de gris et range ses résultats dans une autre image. Le seuillage se fait par sélection de pixels dont la valeur est supérieure à un seuil qui sera donné en paramètre.

Les paragraphes suivants expliquent l'appel de cette fonction dans les deux modes de travail indiqués :

A) Le mode bavard

Cette possibilité d'interaction est très utile pour une première appréhension de la fonction à exécuter et surtout pour l'introduction de paramètres nécessitant une préparation éventuelle (entrée de donnés image ou données graphiques). Le système pose, dans ce mode, une série de questions pour aider l'utilisateur à introduire les paramètres relatifs à la fonction appelée. Nous donnons ci-dessous un exemple d'exécution conversationnelle de la fonction seuillage :

```
                ! SEUILLAGE <rc>
                ? nom de l'image d'entrée : <rc>
                ? parametre implicite : dernier <rc>
                ? faut-il conserver le parametre implicite (o/n) : o <rc>
                ? valeur du seuil en entier : 30 <rc>
                ? nom de l'image resultat : IMAGE2 <rc>
VALIDATION :    ? SEUILLAGE (dernier , 30 → IMAGE2) (o/n) : o <rc>
```

L'exemple est suffisamment explicite. Il aboutit à l'exécution d'un seuillage sur le dernier objet de type image manipulé, avec comme valeur de seuil 30 et le résultat est rangé dans l'image : IMAGE2.

B) Le mode muet

Dans ce mode, les paramètres de la fonction sont donnés en bloc après leur nom sans aucune forme d'interaction avec le système. C'est un mode "raccourci" qui suppose que l'utilisateur connaît parfaitement la syntaxe d'appel et que les paramètres ne nécessitent aucune

préparartion.

Reprenons le même exemple que précédemment. La fonction SEUILLAGE s'écrit dans ce mode comme suit :

```
! SUILLAGE (     ,     30   →    IMAGE2)
     ↑          ↑          ↑          ↑
  nom de     image à    seuil    image résultat
la fonction  seuiller
             prise par défaut
```

L'appel à la commande SEUILLAGE avec tous les paramètres choisis implicitement s'écrira :

```
! SEUILLAGE ()
```

2.2.2. Aide à l'utilisation du menu

La fonction HELP fournit à tout instant les informations sur la prochaine information qu'ellel attend (par exemple le type du paramètre). Appelée avec le nom d'une fonction, elle fournit les indications qu'elle possède sur cette foncton ; par exemple :

```
! HELP (SEUILLAGE) <rc>
SEUILLAGE :
        profil : IMAGE x REAL → IMAGE
        type fonction : fonction mettant à 0
            les valeurs supérieures au seuil.
        paramètres :
           données : 1) type : IMAGE de niveaux de gris,
                        valeur implicite : DERNIER.
                     2) type : INTEGER,
                        valeur implicite : 30
           résultats : 1) type : IMAGE de niveaux de gris
                           valeur implicite : VIDEO1
        méthode : ...
```


2.3. Insertion d'un programme

L'insertion d'un programme dans le menu consiste d'abord à écrire le source dans le langage de la machine, le compiler en dehors du système et le rattacher ensuite à une rubrique du menu en tant que module exécutable.

Afin de permettre un lien concret entre le module obtenu et le système, un fichier d'interface est ajouté au source permettant à l'utilisateur d'exprimer plus directement en terme d'objets image les échanges avec le système (paramètres donnés et résultats).

2.3.1. Présentation du fichier d'interface

Le fichier d'interface comprend trois zones :

- Une zone de commentaire : cette zone contient un commentaire qui indique en termes clairs l'utilité de la fonction et son mode d'utilisation. Le contenu de cette zone sera affiché lorsque l'utilisateur fait appel à la fonction d'aide (HELP).

- Une zone de mots clés : les mots clés sont des chaînes de caractères choisies dans un référentiel de mots clés. Cette zone permettra une recherche rapide d'une fonction en donnant le ou les mots clés qui correspondent à son utilisation. L'interrogation par mots clés n'est pas implantée dans la version actuelle de SAPIN-NI.

- Une zone de paramètres : la zone des paramètres est la plus importante des trois zones d'aide. Elle permet le passage des paramètres avec la vérification de leur nombre et de la concordance de leur type. Elle précise pour chacun des paramètres données ou résultats :

  - le nom,
  - le type (type image),
  - le nom du paramètre implicite ou sa valeur implicite,
  - le commentaire, affiché lors de l'utilisation en mode conversationnel.

L'exemple suivant présente la zone d'aide de la fonction SEUILLAGE :

ZONE D'AIDE DE SEUILLAGE

HELP : % commentaire d'aide %

SEUILLAGE est une fonction compilée opérant point à point sur une image de niveaux de gris en sélectionnant les points dont la valeur est supérieure au seuil S donné en paramètre.

KEYWORDS : % mots significatifs de l'utilisation de SEUILLAGE %

SEUILLAGE - IMAGE DE NVTG - FONCTION PONCTUELLE - ...

PARAMETERS % expression des paramètres donnés %

% premier paramètre : %

IDENTIFICATION : IM1 : IMAGE OF INTEGER;
IMPLICIT : DERNIER % dernière image d'entier rencontrée dans le système %
COMMENT : image d'entrée.

% deuxième paramètre : %

IDENTIFICATION : S : INTEGER;
IMPLICIT : 30;
COMMENT : valeur du seuil en réel.

RESULTS % expression des paramètres donnés %

% premier paramètre : %

IDENTIFICATION : IM2 : IMAGE OF INTEGER;
IMPLICIT : VIDEO1 % VIDEO1 désigne une zone particulière de la mémoire d'image %
COMMENT : nom de l'image résultat.

END % fin de la zone d'aide de SEUILLAGE %

2.3.2. Adjonction d'un programme au menu

Le programme à adjoindre au menu se compose de deux parties :

- un ficher texte comportant les différentes zones d'aide,
- un fichier résultant de la compilation du code source.

La procédure d'adjonction utilisée dans la version lelisp du système sur sm90 est la suivante : avant d'insérer un programme dans le menu, l'utilisateur doit, hors du système SAPIN-NI, faire appel à un utilitaire qui crée d'une part un programme appelant pour le module et qui fait d'autre part la liaison entre les paramètres formels et leurs valeurs. Le programme est ensuite compilé puis une édition de lien de l'ensemble est effectuée en rendant ce programme "relogeable" à l'adresse prévue à cet effet dans le système SAPIN-NI.

Une fonction de service est utilisée pour faire l'adjonction du programme au menu de SAPIN-NI. On doit donner le nom du programme à insérer, le nom de la rubrique qui va le contenir et son type. Ce type sera à choisir parmi EXEC (commande exécutable ou fonction), RUB (rubrique simple) ou RUBEX (rubrique exécutable).

2.3.3. Communication entre les commandes

Toutes les commandes du système sont physiquement indépendantes. De ce fait, elles peuvent être otées ou remplacées sans que cela perturbe l'architecture générale du système. Elles peuvent toutefois se partager, au cours de l'exécution, des variables de travail. Chacune des commandes parmi rubriques ou fonctions est associée à une description dans l'environnement du système. On dira que le système dispose d'un ensemble de variables partageables et que les commandes communiquent par l'intermédiaire de boîtes aux lettres où seront disposées les données communes.

L'écriture de certains programmes en langage évolué avec la possibilité d'interfacer chacun d'eux en précisant en terme d'objet image le type de chaque paramètre ainsi que le choix d'implantation adopté pour faire cohabiter plusieurs commandes ont pour objectif, tout en assurant une puissance dans le traitement, de permettre de gérer efficacement les objets manipulés ainsi que le menu.

2.4. Les types et leur représentation

Il existe trois classes de types manipulés par SAPIN-NI : les types simples, les types propres au traitement d'images et les types fichier. A chacun de ces types est associé un type dans un langage évolué qui va instancier le type SAPIN au sein des modules écrits par l'utilisateur.

2.4.1. Les types simples

Les objets de ces types sont gérés par le système dans la mémoire centrale de la machine. Ils correspondent aux objets simples de PASCAL ou C tels que les entiers, réels, caractères ... Un dernier type simple a été ajouté ici et qui est le nom d'une fonction du système. Ce type servira à préciser le paramètre de la fonction d'aide : HELP.

2.4.2. Les types propres à l'application

Afin de rendre aussi clairs que possible les programmes de traitement d'images, nous proposons à l'utilisateur un canevas de types image représentant globalement les données relatives à cette application. Ces types sont les mêmes que ceux qui ont été définis dans le chapitre 2.

Ainsi, si l'utilisateur a besoin de travailler, par exemple, sur une image de niveaux de gris, il lui suffira d'utiliser le nom du type réservé à cette donnée : ici IMAGE OF NVG et de préciser éventuellement sa taille en écrivant par exemple : IMAGE[256,256] OF NVG. Le type de donnée NVG signifie que le type des pixels sera un entier codé sur 8 bits (0..255). En cas d'absence de taille et de type des données, le système les prend par défaut.

2.4.3. Les types fichier

L'utilisateur a la possibilité de créer des objets de type fichier. Un objet de type fichier ne peut être paramètre que d'une commande spéciale appelée FILE (destinée à la lecture des fichiers).

2.5. Extension du langage de commandes

Le langage de commandes connaît quelques extensions des types :

2.5.1. Composition des commandes

Pour éviter d'introduire des variables intermédiaires et pour raccourcir la longueur des commandes, celles-ci peuvent être composées entre elles :

Exemple :

```
SEUILLAGE (CONVOLUTION (IMAGE1 , MASQUE1) , 30 → IMAGE2)
```

L'exécution de SEUILLAGE s'effectue sur l'image résultat de la convolution effectuée sur IMAGE1 à l'aide de MASQUE1 en utilisant le seuil 30.

2.5.2. Ordres de lecture et écriture

L'exemple suivant montre le calcul des caractéristiques statistiques sur l'image IMAGE2. L'indication de # MOYENNE # permettra d'afficher ce message accompagné de la valeur de l'identificateur qui le suit, c'est à dire MOY :

```
DYNAMIQUE ( IMAGE → MIN , MAX , # MOYENNE # MOY)
```

L'exemple suivant montre comment se fait l'écriture d'un résultat sans rangement dans un identificateur.

```
DYNAMIQUE (IMAGE2 → MIN , MAX , # MOYENNE #)
```

2.5.3. Sélection des résultats

Il arrive, à l'exécution d'une fonction, de ne s'intéresser qu'à un sous-ensemble de ses résultats. Nous remplaçons dans ce cas le paramètre effectif résultat que l'on souhaite ignorer par le signe $.

3. GESTION MEMOIRE

Dans cette partie, nous allons montrer les aspects du système qui sont tournés vers la gestion des objets manipulés et leur implantation mémoire.

3.1. Objets manipulés

Il existe deux classes d'objets manipulés par le système : les commandes et les variables. Parmi les commandes, nous distinguons les commandes de base des macro-commandes.

Les variables désignent les objets manipulées par les commandes. Chaque variable est représentée par un identificateur associé à un type et à une valeur. Une gestion dynamique de l'espace mémoire permet d'affecter à un identificateur une autre valeur même d'un autre type.

Certains identificateurs ont une adresse prédéfinie. Ainsi, la mémoire image est repérée par VIDEO et les quatre sous-images qui la composent sont repérées par VIDEOi (i =1,2,3,4).

3.2. Représentation de l'appel d'une commande

A chaque appel de commande sera associé, au cours de son analyse, un arbre syntaxique décrivant les liens entre les différents objets qui y sont représentés. La racine comprend le nom de l'appel (nom de la commande) et précise les liens de donnée et de résultat avec ses paramètres. Ce schéma est refait (voir exemple ci-dessous) pour chaque commande donnée en paramètre. Les paramètres de type variable seront précisés aux feuilles et leur type donné ou résultat sera précisé par le nom du noeud qui les précède. Cet arbre qui est en fait un arbre d'exécution pourra servir à mémoriser la trace de plusieurs appels au cours d'une session et utilisé par la suite dans une phase de "rejeu" pour rejouer ou sélectionner une sous-séquence.

Exemple de représentation :

a) Exemple d'appel avec imbrication et paramètre implicite

COM1 (COM2 (P1 , P2 → R1) , P3 → R2 , $, R3)

La commande COM1 a deux paramètres en entrée dont une commande (COM2) et trois paramètres en sortie ; au cours de cet appel, on ne cherche pas à obtenir la valeur du deuxième.

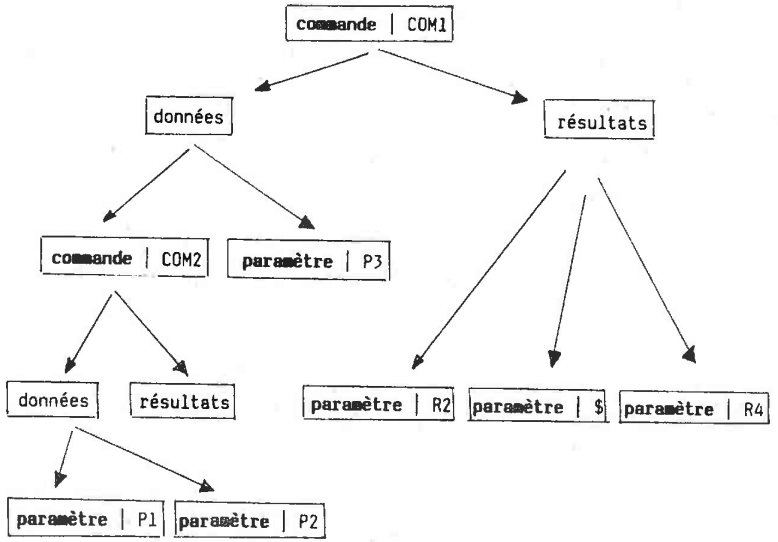


Figure 4.2 : Arbre d'exécution d'une commande.

b) Appel avec lecture et écriture

COM1 (# param1 = # P1 , P2 → #résultat = # R1)

Cet appel est équivalent à :

COM1 (READ (→ 'param1=' P1) , P2 → WRITE (→ 'résultat=' R1))

où READ est une fonction de lecture et WRITE, une fonction d'écriture. Ainsi, cette commande se traduit comme une commande normale.

4. CONSTRUCTION D'UNE MACRO-COMMANDE

4.1. Définition d'une macro-commande

Une macro-commande est un module construit sous le système SAPIN-NI comprenant un enchaînement d'appels à des commandes du menu. Ceci constitue un moyen d'utiliser efficacement les commandes existantes et de permettre à des utilisateurs de construire leur menu personnalisé en enrichissant le menu initial à l'aide de ces macro-commandes.

Tout comme une fonction, une macro-commande comporte une zone de liaison avec le système ou d'interface ajoutée à l'ensemble de ses définitions ou corps.

La zone d'interface comprend un commentaire d'aide et des paramètres qui serviront à l'échange avec le système et à la communication avec les autres commandes. Ces paramètres seront pris en compte par le système, comme pour la fonction, au moment de son insertion dans le menu et rangés dans une table des variables construite à cet effet.

Les objets manipulés dans la zone instructions sont essentiellement des noms de commandes du menu et ceux des types précédents. Ces derniers ne sont pas déclarés explicitement mais introduits comme paramètres de commandes spéciales de lecture ou de nommage de variables globales connues par le système. Une commande de service appelée IMPLICIT permet de mettre à jour les valeurs de ces variables.

La construction de la zone d'appel se fait de deux manières :

- manuelle à l'aide d'un langage d'enchaînement et un éditeur syntaxique,
- à partir d'une trace ou d'un historique des traitements.

Une fois construite, la macro-commande est insérée dans le menu comme toute autre commande.

4.2. Construction manuelle d'une macro-commande

Grâce à un éditeur de texte qui fait l'ananlyse syntaxique, l'utilisateur peut indifféremment utiliser le mode muet ou le mode bavard pour écrire les instructions du langage proposé pour l'enchaînement de ses commandes. Ce langage est emprunté à BASIC et de ce fait nous y retrouvons une structure de ligne et une syntaxe semblable à celle de ce langage. Parmi les instructions, nous trouvons les appels aux commandes mais aussi des instructions de contrôle et de branchement.

Exemple :

Il s'agit d'acquérir une image depuis la caméra et de lui faire subir un certain nombre de traitements. L'image sera acquise dans le premier espace disponible de la mémoire (V1). Ensuite, nous demandons de ranger dans V2 l'image seuillée de V1, dans V3 la convolution de cette même image par un premier laplacien et dans V4, le résultat de la convolution avec un deuxième type de Laplacien.

V1	V2
Image originale	Seuillage
V3	V4
Laplacien1	Laplacien2

Mémoire image

Voici un exemple de macro séquentielle pour réaliser cet affichage :

```
10 MACRO AFFICH
20 READ (→ V1) % Acquisition depuis la caméra et rangement dans V1
30 SEUILLAGE ( ,30 → V2) % la donnée est la dernière image
                           rentrée, c'est-à-dire V1 %
40 CONVOL (V1,L1 → V3) % L1 est un masque laplacien prédéfini %
50 CONVOL (V1,L2 → V4) % L2 est un autre laplacien prédéfini) %
60 END AFFICH
```

4.3. Construction automatique d'une macro-commande

Au cours du travail, l'utilisateur dispose d'une trace qui mémorise sous forme d'historique les traitements effectués.

La trace permet de vérifier l'enchaînement des commandes. Mais elle permet surtout de rejouer un enchaînement de commandes. En particulier, un tel enchaînement s'il est satisfaisant, va pouvoir devenir une macro-commande.

4.3.1. Construction d'une macro à partir de la trace

La trace est un outil simple pour construire automatiquement des macro-commandes avec plusieurs possibilités. L'utilisateur peut dans une phase de préparation rejouer les fonctions de la trace et s'assurer du bon enchaînement des fonctions sur les mêmes données ou sur des données différentes. Cependant, la sélection de certaines des fonctions de la trace pour en construire une macro-commande ne se fait pas sans difficultés. En effet, le chaînage des fonctions sélectionnées est simple quand les fonctions sont indépendantes, c'est-à-dire qu'elles ne se communiquent pas de données. Il devient un vrai jeu de style dès que les fonctions sont liées et que l'on introduit des boucles et des tests dans la macro.

Dans le système IPS [CHA 85], l'utilisateur peut construire interactivement une macro-commande à l'aide d'une trace. Le système offre les moyens de visionner les résultats intermédiaires et de pointer ceux qu'il faut garder.

Nous allons montrer dans ce paragraphe comment nous faisons, dans SAPIN-NI, la sélection d'une séquence de traitements à partir de la trace, afin de construire la macro correspondante.

L'exemple ci-dessous donne la liste des traitements conservés dans une trace lors d'une session de travail. Les flèches désignent les liens entre les divers appels A, B, ..., G. Un lien existe entre A et B si B utilise un paramètre résultat obtenu par A.

Exemple de trace :

A (1)		A (dA → rA)
B (res(A),2)		B (rA → rB)
C (res(A))	ou avec la syntaxe	C (rA → rC)
D (3,res(C))	précédente	D (rc → rD)
E (res(D),res(C))		E (rE,rC → rE)
F (res(E),res(C))		F (rD,rC → rF)
G (res(D),res(F))		G (rD,rF → rG)

où rA = res(A) = résultat de A,
rB = res(B) = résultat de B,
etc...

On peut distinguer deux formes de sélection :

- celle qui consiste à désigner un résultat et demande la séquence qui a abouti à ce résultat. Si le résultat est G, dans l'exemple, nous aurons toutes les commandes sauf B,

- celle qui consiste à sélectionner les commandes une à une. Dans ce cas, la trace est rentrée dans l'éditeur de texte de la macro et l'éditeur désignera celles qu'il désire conserver.

Dans les deux cas de sélection, le passage par l'éditeur est indispensable, d'abord pour constituer entièrement la macro à partir de la trace (compléter l'enchaînement par la zone d'aide...) et mettre à jour ou préciser certaines données qui ont dû être libérées au cours de la session, faute de place. Dans le cas où tous les paramètres (leurs valeurs) existent, une validation par l'utilisateur est néanmoins nécessaire.

4.3.2. Représentation de la trace

La trace des traitements effectués au cours d'une session sera représentée sous forme d'une suite d'arbres du type vu précédemment. Chacun de ces arbres correspondra à un appel de commande.

A) Représentation du lien d'appel

Connaissant un certain résultat, nous voudrions retrouver la séquence de traitements qui ont abouti à ce résultat. Ceci sera fait dans l'arborescence en ajoutant d'une part un lien entre chacun des paramètres résultats et de la commande qui l'a obtenu (lien(1) dans la figure 4.3) et en remplaçant d'autre part tout noeud paramètre de type donnée par un lien vers le même noeud paramétré mais de type résultat de la dernière commande.

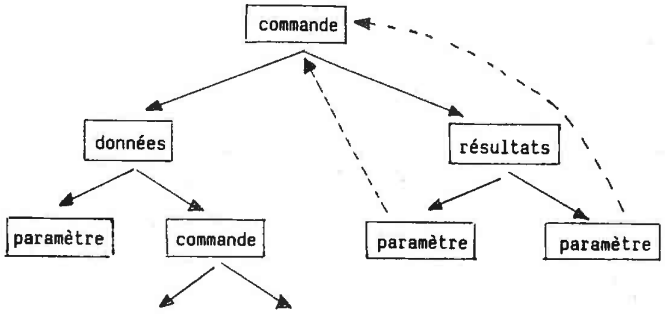


Figure 4.3 : Liens entre commandes, données et résultats

B) Exemple :

Soit la succession d'appels suivante :

COM1 (P1 , COM3 (COM4 (P5 → P3) → P4 , P5) → P2 , P3)

COM2 (P3 , P5 , P6 → P4 , P5)

La représentation arborescente correspondante est :

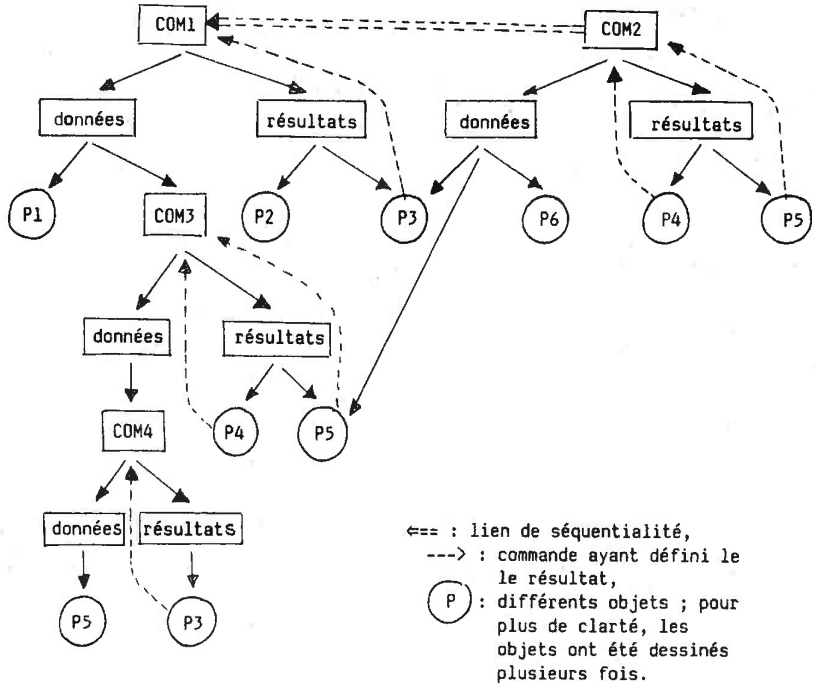


Figure 4.4 : Représentation d'une séquence de traitements par un arbre d'exécution.

5. CONCLUSION

Nous avons présenté dans ce chapitre les fondements essentiels d'un système de génération de menus destiné à être utilisé par des non-informaticiens dans des applications très variées. Ici, on le montre appliqué aux images à travers la version SAPIN-NI, mais, à part les types image, tout le reste du système peut être appliqué à des données de tout genre.

Dans une préface à cette étude, nous avons montré l'organisation du menu en un arbre de commandes classées par rubriques. Ce menu comprend au départ un noyau de commandes de base spécifiques à l'application choisie. Il est libre, ensuite, à l'utilisateur de l'enrichir avec des

programmes exécutables écrits dans le langage hôte ou avec des enchaînements des commandes du menu appelées macro-commandes. Plusieurs outils d'aide sont mis en place pour faciliter l'utilisation d'un tel système.

Les aspects système sont réservés à la représentation des objets et leur gestion interne. Toutes les commandes du système sont indépendantes mais peuvent inter-agir. Le système prend en charge totalement cette interaction en gérant mieux ses ressources.

Deux versions test de ce système ont été écrites, la première en PASCAL MS/DOS sur un micro-ordinateur de type SIRUS 1 [CAS 83], la seconde en Lisp sur SM90 [MOU 85]. Le travail a été réalisé en collaboration avec l'ETCA qui a développé son propre système appelé LAMPION [SER 85].

CHAPITRE 5 : PARALLELISME ET TRAITEMENT D'IMAGES

1. INTRODUCTION

Pour réaliser des programmes de manière optimale, il faut exploiter les possibilités du logiciel et de l'architecture.

Pour l'architecture, le maître-mot est le **parallélisme**. En effet, le défit permanent lancé aux concepteurs de machines pour couvrir des domaines de plus en plus spécialisés a provoqué une course qui s'est traduite par l'apparition d'une grande variété de structures parallèles en moins de trente ans. Le parallélisme architectural existe donc. Nous avons tenté de faire une classification rapide des principales structures dans le chapitre 1.

La distinction la plus étendue sépare les multiprocesseurs des réseaux de processeurs. Les premiers exécutent simultanément plusieurs tâches. Les processeurs communiquent entre eux par l'intermédiaire d'un réseau d'interconnexion. Pour l'utilisateur, cela signifie que chaque tâche sera exécutée par un processeur et qu'une tâche peut utiliser les résultats calculés par une autre. Mais cette communication entre tâches conduit l'ordinateur hôte à gérer les ressources de l'ensemble et à résoudre les conflits d'accès aux ressources communes. Il en résulte que la parallélisation d'algorithmes doit conduire à présenter le programme sous forme de tâches indépendantes exécutables par le multiprocesseurs.

Les réseaux de processeurs présentent la caractéristique d'exécuter en commun un seul programme et d'atteindre des vitesses supérieures à celles des architectures précédentes grâce au cadencement mesuré de leur exécution. La structure d'interconnexion entre les processeurs élémentaires correspond bien à la disposition des points dans une image ce qui permet de réaliser plus facilement les opérations ponctuelles et locales [ZAV 84] [DAN 81].

Il reste le cas des structures pipeline qui sont parmi les plus anciennes. Le parallélisme est réalisé au sein même de l'unité de traitement. Cette unité est divisée en plusieurs sections correspondant chacune à une étape de l'opération à exécuter. Le temps de traversée d'une section correspond à l'intervalle entre deux tops d'horloge. Le temps de latence du pipeline est égal au temps de son chargement et son déchargement, ce qui constitue son principal inconvénient. L'avantage du pipeline est qu'une fois chargé, il produit un résultat tous les tops d'horloge [CAS 84].

L'utilisation optimisée de ces architectures nécessite une bonne distribution du travail sur les différents processeurs. Ceci reste la tâche essentielle du logiciel et ce à travers deux moyens :

- la détection du parallélisme,
- l'expression du parallélisme.

détection du parallélisme

Cette approche consiste à transformer un programme séquentiel en un programme parallèle et en assurer l'exécution. Des méthodes de transformation de programmes abondent dans la littérature. Les premières d'entre elles ont été introduites pour optimiser au départ des programmes Fortran sur la machine multiprocesseurs ILLIAC IV. La nature scientifique des traitements réalisés dans ce langage manipulant d'énormes matrices de données a conduit à chercher des solutions optimales pour des vectoriseurs de machines puissantes comme le CRAY.

La détection des opérations parallèles est souvent réalisée à l'exécution du programme dans les machines de type SIMD et MISD ainsi que dans celles qui sont séquencées par les données. Dans les machines MIMD, on cherche surtout à regrouper les instructions dépendant d'un même ensemble de données et à en former des tâches. Pour cela, la meilleure phase pour détecter le parallélisme se situe avant la compilation puisque c'est à ce moment là que l'on connaît le mieux la structure du programme [MAZ 78].

Expression du parallélisme

Ce mode convient lorsque l'on travaille dans un environnement de programmation parallèle. L'expression du parallélisme conduit l'utilisateur à définir ses traitements sous forme de tâches indépendantes, leurs assimiler des processus qui seront ensuite répartis sur les différents processeurs de la machine. La difficulté du problème réside ici dans la répartition des tâches sur les processeurs disponibles et surtout dans la gestion de la communication et de la synchronisation entre eux.

Il existe des logiciels permettant de formuler des activités indépendantes et de lancer leur exécution en parallèle sur les processeurs de la machine. Notons le système d'exploitation CHORUS [GUI 84] développé sur les systèmes SM90 qui permet de structurer les processus dans un milieu réparti et en détermine la synchronisation.

Pour les machines vectorielles, il existe des langages évolués qui intègrent de tels mécanismes comme le langage TRANQUIL [ABE 69], de type ALGOL, développé pour la machine ILLIAC IV et comprenant des mécanismes d'expression d'opérations vectorielles pour cette machine. Des mécanismes similaires se trouvent dans des langages plus récents de type STARAN [LAN 76] sur la machine vectorielle CDC STAR.

les solutions dans SAPIN

Le parallélisme étudié dans SAPIN est de **nature simple** puisque nous ne cherchons pas à le gérer mais seulement à le mettre en évidence.

Pour l'expression, il existe deux grandes classes d'instructions qui expriment le parallélisme dans LPSI : celles qui définissent des opérations matricielles et vectorielles globales sur les images, contenant des opérateurs arithmétiques, logiques et de filtrage et pouvant être réalisées par des opérateurs SIMD cablés, et celles qui permettent d'exprimer des itérations où toutes les instances des instructions sont indépendantes.

Pour la détection, il s'agit de localiser les opérations image spécialisées pour lesquelles nous possédons des schémas d'exécution. Certaines sont faciles comme celles évoquées ci-dessus et qui sont bien visibles dans LPSI, d'autres sont moins faciles puisqu'elles ne correspondent visiblement à aucun modèle. Nous pensons bien sûr à tous les algorithmes pyramidaux dont il s'agira d'allier les instructions à un modèle de programmation associative. Une analyse en profondeur du programme est ici nécessaire car on exprime souvent dans ce cas le partage de tâches entre plusieurs quadrants de l'image où les problèmes aux frontières ne sont pas souvent simples.

Nous présentons dans une première partie de ce chapitre les principales techniques classiques de détection et d'expression du parallélisme. Dans ce contexte, il nous sera facile ensuite de parler des machines de traitement d'images auxquelles nous nous sommes intéressées : les principaux choix algorithmiques pour la parallélisation d'algorithmes apparaîtront comme une conséquence directe des objectifs assignés à ces deux systèmes.

2. L'ALGORITHMIQUE DU PARALLELISME

Nous présentons dans cette partie les solutions algorithmiques que l'on peut apporter à un programme pour l'exprimer de manière parallèle. Il importe en effet, pour un problème donné, de lui trouver des solutions algorithmiques de parallélisation qui prennent en compte au maximum les contraintes du matériel sur lequel on doit l'exécuter.

Ces solutions diffèrent suivant que le langage hôte comprend les primitives nécessaires à l'expression du parallélisme ou non. Dans le premier cas, il s'agit de bien faire correspondre la structure du programme avec celle de la machine en utilisant astucieusement les outils offerts par le langage. Dans le second, le problème est nettement plus difficile. Il s'agit de transformer le code du programme séquentiel en un code parallèle. Cette transformation doit en même temps tenir compte de l'indépendance entre les instructions et des contraintes de l'architecture pour réaliser une "parfaite" décomposition. Il faut en plus être capable de pouvoir exprimer correctement les éventuelles communications entre les parties parallèles du programme.

2.1. Détection automatique du parallélisme

D'une manière générale, La détection du parallélisme est fondée sur des règles de partage des variables entre les différentes instructions du programme et regroupement de celles qui, vis à vis de ces règles, sont fortement dépendantes. Cependant, l'analyse des instructions est souvent confrontée à l'analyse des boucles et des tests où la notion de dépendance prend une dimension plus importante que dans le cas d'instructions consécutives simples. Ces considérations vont donc conduire à examiner toutes les structures du programme et les transformer pour en simplifier l'exécution.

Parmi les entités du programme que l'on cherchera donc à paralléliser nous trouvons :

- les instructions différentes,
- les itérations,
- les expressions arithmétiques et logiques.

2.1.1. Les instructions différentes

les instructions différentes sont des instructions simples, des groupes d'instructions simples ou des sous programmes (appelons les

tâches). Pour étudier leur indépendance, on examine le flot de données qui circule entre elles.

Pour cela, BERNSTEIN [BER 66] a développé les conditions qui doivent être satisfaites pour que deux tâches séquentielles ti et tj puissent s'exécuter en parallèle. Elles sont fondées sur les différentes sortes d'utilisation d'une zone mémoire ou d'une variable par une tâche.

En se plaçant dans le contexte d'une machine contenant plusieurs processeurs se partageant la même mémoire, on peut établir des règles d'accès à cette mémoire par les processeurs à fin de réaliser de manière stricte une exécution parallèle de deux tâches.

◆ - l'espace mémoire commun à deux tâches ti et tj où ti assure une opération de lecture et l'autre une opération d'écriture doit être en exclusion mutuelle.

◆ - d'autre part à fin de continuer à respecter la séquentialité entre deux tâches pour l'accès en écriture dans la même zone, on interdit que deux tâches modifient en même temps la même zone.

Règles de BERNSTEIN

Deux tâches ti et tj peuvent s'exécuter en parallèle à condition qu'elles ne soient liées par aucune des trois relations :

$$PC: M(ti) \cap L(tj) = \emptyset$$
$$CP: L(ti) \cap M(tj) = \emptyset$$
$$PP: M(ti) \cap M(tj) = \emptyset$$

où M(ti) (resp L(ti)) désigne l'ensemble des variables modifiées (resp lues ou utlisées) par ti. Ces règles traduisent les échanges possibles (Production-Consommation) entre deux tâches successives ti et tj.

Quand l'une de ces trois règles n'est pas remplie, on dit que ti et tj sont en **collision** ou en **conflit**. La relation PC exprime que ti produit un résultat que tj exploite.

Les relations CP et PP indiquent qu'une variable a été réutilisée et que sa valeur courante ne doit pas être écrasée avant que toutes les instructions qui l'utilisent soient terminées. Elles peuvent être rompues en donnant à chaque fois un nouveau nom à la variable produite. Ceci se fait dans la programmation à **assignation unique**. Cependant, ce type de programmation consomme énormément d'espace mémoire.

Pour exploiter ces relations, un **graphe de dépendance** est construit à partir du programme source. Il matérialise d'une part l'ordre d'exécution des tâches (ce qui est important quand ces tâches sont itérées dans une boucle) et d'autre part les relations conflictuelles entre une tâche et la tâche suivante. On groupe ainsi les instructions du programme échangeant des résultats. L'analyse des échanges d'information est facile quand les instructions manipulent des variables

simples. Ce problème se complique rapidement dès que les données échangées sont complexes (tableaux, structures, ...) où il devient difficile de cerner les objets effectivement accédés par les instructions, en particulier quand les fonctions d'accès utilisent des pointeurs.

2.1.2. Les itérations

Le schéma séquentiel de déroulement d'une itération consiste en la répétition de plusieurs fois les mêmes instructions pour des instances de variables différentes. L'analyse d'une itération peut conduire à deux formes de parallélisation :

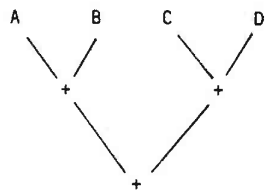
- si toutes les instances de l'itération peuvent s'exécuter simultanément, soit parce qu'elles ne sont pas en collision, soit parce que ces collisions peuvent être réduites au moyen d'une exécution synchronisée, alors l'itération est dite **vectorisable** [LAM 75],

- si les instructions de l'itération se prêtent à un partitionnement en groupes indépendants, alors l'itération est dite **parallélisable.**

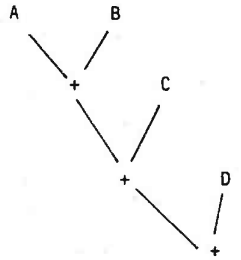
On reprend ici les idées de la parallélisation des instructions différentes. On construit un graphe de dépendance qui matérialise les collisions entre les différentes instructions après distribution de la fonction de boucle sur les expressions et les tests qu'elle contient. Ensuite, on utilise des techniques de transformation de programme comme la dérécursivation des variables, les substitutions ou encore les expansions des scalaires modifiés dans la boucle [REN 80] [ROU 73] [RAM 69] [TJA 70].

2.1.3. Les expressions arithmétiques et logiques

On exploite les propriétés d'associativité, de commutativité et de distributivité des opérations qui permettent de réduire le nombre de pas de calcul de l'expression. Ainsi l'expression A + B + C + D peut être analysée comme si elle a été écrite sous la forme (A + B) + (C + D) plutôt que par l'association habituelle gauche droite ((A + B) + C) + D). Mise sous forme d'arbre, cela donne :



plutôt que



Une telle analyse appelée par ALLEN et COCKE [ALL 71] analyse à profondeur minimum, minimise la profondeur de l'arbre d'exécution.

L'effet principal de cette transformation est de montrer l'indépendance entre les sous-expressions par exploitation des propriétés énoncées ci-dessus. Ces transformations sont sévèrement contraintes par des considérations de dépendance entre les données. Dans l'exemple ci-dessus, A + B est montré indépendant de C et D par cette analyse et peut être déplacé, transformé ou même exécuté de manière indépendante.

2.2. Expression du parallélisme

Plusieurs mécanismes classiques existent déjà pour expliciter le parallélisme d'un programme. Ils sont fondés sur la définition de processus correspondant au lancement d'une ou de plusieurs activités d'un programme.

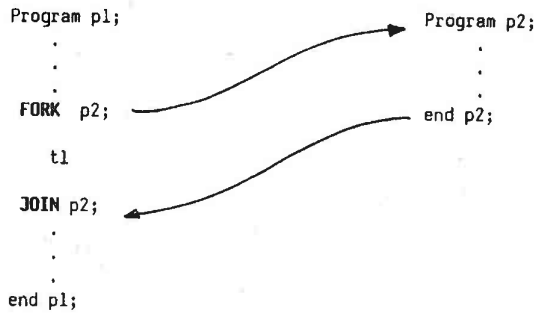
2.2.1. Les mécanismes de distribution

Parmi les mécanismes de distribution les plus importants, nous trouvons:

◆ - **FORK et JOIN**

Ces mécanismes développées par CONWAY [CON 63] permettent de lancer plusieurs calculs en parallèle et d'attendre la fin de ces calculs pour continuer. Le lancement est effectué par la primitive FORK, tandis que la synchronisation par la primitive JOIN.

exemple



Dans cet exemple, t1 et p2 s'exécutent en parallèle. Cette primitive générale connaît différentes variantes pour exprimer différentes formes de parallélisme comme le parallélisme disjoint qui permet d'exécuter plusieurs tâches en parallèle en un point donné du programme. Les instructions pour cela, comprennent une primitive FORK pour lancer les n tâches et n primitives JOIN pour synchroniser chaque tâche avec les autres. Une autre forme de parallélisme est le parallélisme itératif qui, à l'opposé du précédent, permet d'exécuter plusieurs fois en parallèle, selon le nombre de processeurs disponibles, la même section de programme.

Dans ces types de parallélisme, il est parfois souhaitable de disposer de mécanismes de synchronisation forcée permettant l'arrêt des processus lancés en parallèle. L'exécution d'une primitive appelée FSAIS permet de faire cela en tuant les processus frères ainsi que leurs descendants éventuels créés par une des primitives précédentes. Le processus comportant cette primitive continue en séquence.

Ces instructions, bien qu'elles soient simples à utiliser, peuvent réduire la lisibilité du programme quand elles figurent dans des itérations ou des conditionnelles.

◆ - **COBEGIN ... COEND**

COBEGIN A1 // A2 // ... // An **COEND**

Cette instruction indique que les activités A1, A2, ..., An peuvent s'exécuter en parallèle. L'exécution de cette instruction se termine après l'exécution de toutes les activités spécifiées.

Cette forme est moins puissante que l'instruction FORK et JOIN mais les tâches y sont mieux spécifiées et leurs entrées et sorties sont mieux définies.

Ces tâches parallèles auront besoin, pour leur exécution, d'espace mémoire contenant les données sur lesquelles elles travaillent ainsi que d'un espace mémoire pour ranger les variables intermédiaires nécessaires à leur calcul. Une gestion de ces espaces reste à définir dans tous les cas de parallélisme évoqués précédemment.

◆ - **les MODULES**

Les modules sont définis dans ADA comme des types abstraits. Ils spécifient des collections de processus comme des entités ne partageant pas de variables et ne pouvant communiquer que par des échanges de messages à travers des voies d'entrée et de sortie.

Cette notion de **module** offre une meilleure structuration des traitements que dans le cas des processus et clarifient la voie de communication entre eux.

2.2.2. Les mécanismes de contrôle

Lorsqu'une coopération devient nécessaire entre les différents processus établis, il est important de définir les protocoles liés à leurs activités parallèles. Cette définition conduit généralement à résoudre deux types de problème :

- problème de communication entre processus,
- problème de synchronisation.

Le problème de la communication concerne l'échange d'information entre les processus. La synchronisation est indispensable pour résoudre les problèmes de compétition pour l'accès à des ressources communes partagées et les problèmes de coopération pour mener à bien une application.

Les ressources d'un système (processeurs, mémoires) appartiennent en général à deux catégories : les ressources partageables susceptibles d'être utilisées par plusieurs processus simultanément et les ressources non partageables qui ne peuvent être utilisées que par un seul processus à la fois à un instant donné.

L'exemple classique en traitement d'images est la mémoire d'images qui peut être accédée par plusieurs processeurs. Si un processeur lit le contenu de la mémoire pendant que l'autre la modifie, le résultat sera imprévisible et désastreux.

A) Cas des systèmes centralisés

On dit qu'un système est centralisé lorsque les algorithmes de contrôle reposent sur l'existence d'une mémoire centrale que peuvent accéder concurremment tous les processus en lecture et en écriture. La communication et la synchronisation entre les activités parallèles se font par des variables partagées. On distingue en général deux types de synchronisation :

- la synchronisation pour assurer l'exclusion mutuelle, ce qui permet de garantir l'indivisibilité d'une opération pouvant être exécutée par plusieurs activités concurrentes,

- la synchronisation conditionnelle qui ne permet l'exécution d'une opération que si l'état des variables le permet.

Parmi les mécanismes proposés pour la synchronisaton basée sur des variables partagées, nous trouvons :

◆ - **les sémaphores** DIJKSTRA [DIJ 68]

Un sémaphore est défini comme une variable entière S à laquelle est associée une file d'attente F (initialement vide). Le mécanisme de sémaphores traduit la coopération des processus mais ne fournit aucun dispositif de communication. Ceci est utile dans un système où les processus s'ignorent mutuellement [CHA 80].

◆ - **L'attente active**

Le principe de l'attente active est simple. Pour signaler une condition, un processus positionne une variable à une valeur donnée. Pour demander une condition, un processus consulte répétitivement cette variable jusqu'à ce qu'elle prenne la valeur attendue. C'est ainsi que le problème du conflit est résolu sur certains systèmes par l'utilisation de verrous où les processus bouclent sur des instructions spéciales appelées T.A.S (Test and Set).

L'inconvénient majeur de cette méthode est la consommation du temps machine et la monopolisation d'un processeur pour l'exécution des instructions T.A.S.

◆ - **Les sections critiques conditionnelles**

Les sémaphores donnent des solutions qui sont en général de bas niveau. Par ailleurs dans un programme, on distingue mal exclusion mutuelle et synchronisation conditionnelle. Ce qui a conduit à écrire syntaxiquement une section critique. HOARE [HOA 72] introduit des déclarations de sections critiques où se trouvent regroupées de manière explicite les différentes variables partagées.

◆ - **Les moniteurs**

Un moniteur est un mécanisme qui, à la manière des types abstraits, définit des données et les procédures réalisables sur elles. Seules les procédures du moniteur accèdent à ses données. Ces procédures sont accessibles de l'extérieur et exécutées en exclusion mutuelle les unes par rapport aux autres. On autorise à l'intérieur d'un tel module, un seul processus actif. Donc il y a exclusion mutuelle pour l'accès de la ressource.

DIJKSTRA suggère de régler les problèmes de synchronisation entre n processus par le (n+1)ème qu'il appelle secrétaire. La définition du moniteur conduit dans ce cas à séparer la définition des opérations de synchronisation de leurs utilisations.

Dans les moniteurs de Hoare, les mécanismes sont des mécanismes d'actions indirectes par des files d'attente. Deux opérations sont définies sur elles:

WAIT : le processus actif sur ce wait est en attente.

SIGNAL : s'il existe un processus dans la condition, il est mis à l'état logiquement actif (il est prêt).

Ces deux opérations avec les files permettent de réaliser la synchronisation conditionnelle.

◆ - **Les expressions de chemins**

C'est l'outil de synchronisation qui sera, à notre avis, le plus facilement applicable en traitement d'images. Les expressions de chemins spécifient l'ordre dans lequel les opérations définies sur une donnée partagée, peuvent être exécutées par des activités parallèles.

CAMBELL et HABERMAN [CAM 78] qui sont à l'origine de cette idée définissent un type abstrait dans lequel ils expriment à côté des données et procédures d'un objet, les chemins possibles que les processus doivent respecter dans leurs exécutions.

Cet outil de synchronisation permet, comme les moniteurs, de regrouper dans le même module toutes les informations concernant la synchronisation. Seulement, au contraire des moniteurs, il ne possède pas les moyens d'expression des attentes.

B) Cas des systèmes répartis

La répartition, appelée aussi distribution, de systèmes est assez difficile à définir, pourtant plusieurs exemples de systèmes informatiques, que ce soit pour matérialiser des architectures spécialisées (de type multiprocesseurs) ou pour établir la communication entre plusieurs sites (réseaux), sont de ce type.

Les primitives de communication sont caractérisées par :

- le type de liaison : *il dépend du nombre de partenaires réalisant l'échange d'information,*

- le type de communication : *le type de communication le plus connu est le* ***rendez-vous*** *où l'émetteur reste bloqué jusqu'à ce que le message soit reçu comme dans CSP,*

- la forme d'expression : *plusieurs types d'expressions existent. Parmi les plus classiques, on peut distinguer* ***SEND / RECEIVE*** *utilisées dans CSP, FLEXI, etc... et les primitives de type appel de procédures exprimées dans ADA sous la forme suivante :* ***CALL*** *service (paramètres) et* ***ACCEPT*** *service (paramètres),*

- la tolérance aux défaillances : *plusieurs types de défaillance peuvent avoir lieu dans ces mécanismes de communication par messages. Notons principalement la défaillance du support de transmission ou celle d'un processeur. Des propositions sont faites pour rendre les primitives tolérantes à ces défaillances.*

Les lecteurs peuvent trouver dans [CHA 80] [COR 81] [KRA 85] plus d'explications sur ces problèmes de communication et synchronisation

Nous allons donner dans la suite un exemple de système réparti, le système Cm* (largement commenté dans CORNAFION [COR 81]) mais choisi ici pour illustrer ces principes dans le cadre d'une machine parallèle destinée au traitement d'images et à la reconnaissance de la parole [JON 79].

Il a une architecture multiprocesseurs comprenant une centaine de processeurs accédant à des objets communs (partageables). L'architecture est subdivisée en molécules comprenant chacune un processeur, une mémoire locale et des organes d'entrée-sortie. Chaque processeur peut accéder aux mémoires des autres processeurs par l'intermédiaire d'adresseurs. Ce qui va obliger à envisager des règles d'accès pour éviter les conflits entre les différents processeurs.

Les molécules sont regroupées en amas (voir fig. ci-dessous). Un amas peut contenir jusqu'à 14 molécules. Une molécule appartient à un amas et un seul. Les molécules sont rattachées entre elles, via des adresseurs locaux appelés Slocal, par un bus local.

Les amas sont reliés entre eux, via des adresseurs particuliers appelés Kmap, par des bus inter-amas.

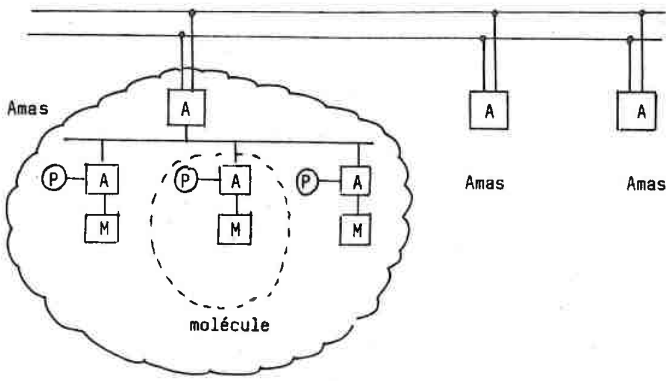


Figure 5.1 : Schéma de l'architecture du système Cm*

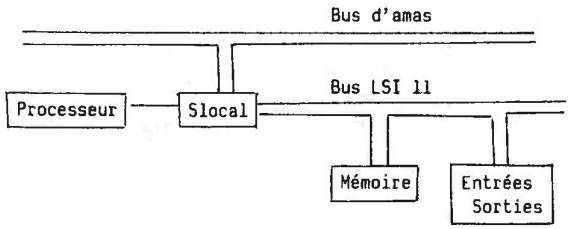


Figure 5.2 : Schéma d'une molécule dans Cm*.

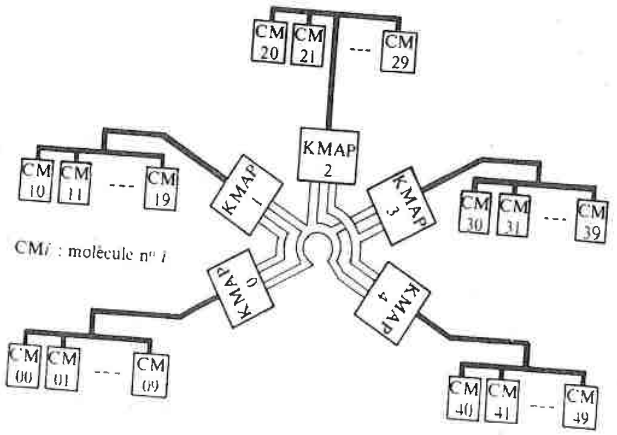


Figure 5.3 : Schéma général de Cm*.

Lorsqu'un processeur veut accéder à la mémoire, il envoie sa requête à son adresseur local qui réalise un premier aiguillage selon que la cellule désignée est présente ou non dans la molécule. Dans le cas où elle est présente, le bus LSI 11 réalise cet accès. Dans le cas contraire, le Kmap prend en charge la requête et détermine si la cellule désignée est présente ou non dans l'amas, et la demande est envoyée à un autre amas par le bus inter-amas.

Cette architecture n'est viable que si les réseaux de communication entre les amas sont très astucieusement étudiés ou si les références d'un processeur à la mémoire sortent rarement de sa molécule et exceptionnellement de son amas, c'est ce qui a été retenu pour la configuration finale de Cm*.

Les Kmap peuvent traiter jusqu'à 8 requêtes en parallèle. L'arbitrage entre les processeurs est réalisé ainsi : le bus du LSI 11 est arbitré par le LSI 11. Le bus d'un amas est arbitré par l'horloge de base du bus. Chaque bus inter-amas est un bus asynchrone avec arbitrage réparti.

Les techniques de transfert adaptées à cette architecture permettent d'augmenter la rentabilité des bus en rendant l'interblocage le moins probable possible.

Les applications développées sur Cm* passent par la création d'un ensemble de processus qui coopèrent à une tâche commune. La

communication entre ces processus peut être faite par partage de segments ou envoi de messages au moyen de boîtes aux lettres.

La programmation de l'application est réalisée à l'aide d'un langage qui permet d'exprimer le parallélisme d'exécution des différents processus sur les processeurs existants et de spécifier la contrainte de proximité entre informations dans la même molécule ou le même amas.

L'application développée sur Cm* concerne un programme d'analyse d'image. Une image est découpée en tranches puis chacune est traitée par un processus distinct. Chaque processus peut communiquer seulement avec ceux qui analysent les tranches voisines à la sienne. Un processus maître crée les processus de traitement, répartit les tâches entre eux, lance leur exécution et récupère leurs résultats. Il communique avec eux par des boîtes aux lettres.

Les aspects de la répartition dans ce système se trouvent liés d'une part aux considérations de la localité pour l'exécution des tâches, et d'autre part à la décomposition d'une tâche en processus coopérants.

2.3. Conclusion

Les paragraphes précédents montrent que des solutions algorithmiques pour le parallélisme existent dans les systèmes informatiques classiques. Certes, cette liste est courte et bien d'autres solutions sont imaginables mais, celles qui ont été données montrent les types de problèmes posés.

Cependant, en traitement d'images, les solutions en soi ne suffisent pas, encore faut-il savoir les utiliser sur les machines spécialisées avec un meilleur rendement. Les solutions que nous apportons n'utilisent pas ces techniques classiques. Elles s'en inspirent pour mettre en évidence les opérations parallèles pour lesquelles des solutions architecturales existent. Les problèmes de communication et de synchronisation resteront l'affaire des logiciels d'interface.

3. LE PARALLELISME DANS LES PROGRAMMES IMAGE

3.1. Présentation générale

On peut distinguer le parallélisme dans les programmes image à deux niveaux :

- le **bas niveau** où sont regroupées toutes les opérations de base comme le filtrage, les manipulations de couleurs, et l'extraction de données statistiques,

- le **haut niveau** qui exprime surtout des tâches d'analyse et d'interprétation d'images.

Dans les opérations de bas niveau, le parallélisme se situe au niveau du pixel car il s'agit souvent d'opérations élémentaires qui se répètent quasi-identiquement sur les tous les pixels de l'image. Suivant le nombre de pixels mis en jeu pour réaliser l'opération, on peut distinguer trois classes d'opérateurs :

- ponctuel : *l'opérateur s'applique sur chaque pixel de l'image en ne mettant en jeu que la valeur du pixel, ce qui évite tout problème de partage de données,*

- local : *l'opérateur s'applique sur le voisinage d'un pixel dont une partie est commune avec les voisinages des pixels voisins,*

- global : *l'opérateur a besoin de tous les pixels de l'image pour se réaliser. L'opération globale exécutée de manière parallèle sur tous les pixels de l'image nécessiterait plusieurs synchronisations.*

D'autres opérations de bas niveau comme les opérations d'étiquetage de régions existent mais ne relèvent pas de cette classification car elles sont mal conditionnées pour le parallélisme évoqué ci-dessus.

Ce type de parallélisme (parallélisme d'expressions distinctes) est conceptuellement facile à détecter automatiquement et à mettre en oeuvre sur différentes architectures et plus spécialement sur les architectures SIMD.

Dans le haut niveau, les algorithmes d'images sont constitués d'un aggrégat d'opérations réalisant des actions qui relèvent souvent de l'analyse d'images. A ce niveau, on manipule surtout des descriptions abstraites de l'image en utilisant des structures de données complexes. Le réarrangement des groupes d'opérations en opérations indépendantes relève des techniques d'analyse de programmes que nous avons évoquées précédemment.

3.2. Mise en évidence du parallélisme

Il s'agit en fait de deux sortes de parallélisme : celui de définir des opérations évidemment globales et indépendantes et celui de tenter d'exprimer le parallélisme entre les instructions image et de les rallier à un modèle parallèle dont on a fixé la structure.

3.2.1. Les opérations évidemment globales

Ce sont toutes les opérations matricielles et vectorielles pouvant s'exprimer globalement sans avoir besoin d'expliciter l'opération sur les éléments. Ces opérations englobent les expressions arithmétiques et logiques, les opérations de filtrage et les instructions d'affectation. Ainsi, l'instruction suivante :

$$I := \lambda 1\ I1 + \lambda 2\ I2$$

exprime une combinaison linéaire de deux images I1 et I2 et le rangement du résultat dans l'image I. Les opérations de multiplication par une constante $\lambda 1$ I1 et $\lambda 2$ I2 sont d'abord traitées simultanément, ensuite sont additionnées et affectées à I. Les opérations d'addition, de multiplication et d'affectation sont des opérations isomorphes. Elles agissent à chaque fois sur deux pixels homologues dans les opérandes image. De ce fait, elles se prêtent bien au parallélisme SIMD. Plusieurs types de langage existent depuis longtemps permettant l'exécution de telles opérations. Notons, en particulier, le Fortran développé par Wedel [WED 75] pour la machine TI ASC.

3.2.2. Les instructions potentiellement parallèles

Il s'agit là de sous-programmes dont l'action générale peut être ramenée à l'exécution d'une opération parallèle du type précédent. Le problème qui se pose est de se demander quels procédés utiliser pour faire cette détection sans avoir recours aux techniques classiques mais seulement à l'arbre syntaxique du sous-programme préalablement analysé.

Ce problème, si a priori ne pose que très peu de difficultés dans le cas de machines multiprocesseurs car les modèles d'instructions utilisés pour répéter des traitements sont connus, devient vite inextricable dans le cas de machines pyramidales où viennent s'ajouter aux instructions les contrôles des mouvements de données entre les différents processeurs.

Un exemple de traitement pyramidal est celui du calcul du maximum d'une image. En utilisant tous les étages de la pyramide, cette opération peut s'effectuer par réduction de l'image de départ en une succession d'images réduites de maxima locaux.

L'analyse du programme séquentiel doit conduire à l'extraction du noyau de traitement commun à tous les processeurs et à allier le programme au schéma de traitement suivant :

```
procedure noeud;

        max = - ∞

        pour tout fils faire

                reception-fils val;
                si val > max alors max = val
                fsi

        fpour

fin noeud;
```

Cet algorithme exprime de manière précise le travail de chaque noeud receptionnant les données demandées à ses fils et les réduisant au maximum qu'il transmet à son père. L'analyse du programme doit découvrir le caractère ascendant de la réduction par le maximum. D'autres problèmes sont plus difficiles à résoudre notamment quand le programme de réduction s'autorise des retours arrières se traduisant par des mouvements descendants puis ascendants.

4. LES SOLUTIONS INTRINSEQUES

Dans le chapitre 1, nous avons évoqué toutes les architectures construites pour le traitement d'images. Ici, nous allons nous intéresser de près à deux d'entre elles pour lesquelles nous avons examiner quelques solutions de parallélisation d'algorithmes : les architectures multiprocesseurs de type MIMD et les architectures pyramidales. Deux exemples de systèmes seront détaillés dans la suite.

4.1. Les machines multiprocesseurs

4.1.1. Définition générale

Les multiprocesseurs sont des systèmes informatiques capables d'exécuter simultanément plusieurs tâches. Les tâches peuvent être différentes ce qui permet d'assurer une grande diversité de calcul. De telles structures permettent, en augmentant le nombre d'unités de traitement, dans des limites raisonnables, d'augmenter la puissance de la machine et de disposer d'unités adaptées aux tâches à réaliser. Dans plusieurs systèmes, les unités de traitement peuvent avoir une architecture spécialisée, ce qui permet de choisir l'architecture de l'unité en fonction du traitement à effectuer. De plus, Elles sont souvent reconfigurables, ce qui permet d'offrir la même souplesse qu'avec les unités de traitement classiques. En cas de panne de certaines d'entre elles, l'architecture permet facilement au système de pouvoir continuer à fonctionner avec les unités restantes sans risque de perturbation. Cet aspect modulaire a été particulièrement étudié dans le projet ICOTECH.

Cependant, le principal inconvénient de ces architectures est le conflit d'accès aux ressources communes. Les unités de traitement accèdent souvent aux mêmes bancs mémoire par l'intermédiaire d'un réseau de communication qui constitue le goulot d'étranglement dans ce type d'architecture. La performance de ces systèmes est donc tributaire de l'occurence des conflits d'accès ce qui nécessite souvent la mise en place d'unités d'arbitrage qui sont très coûteuses en temps. De ce fait, le nombre d'unités se trouve limité : 16 semble être un maximum pour Lenfant [LEN 85].

4.1.2. Adéquation au traitement d'images

Cette structure multiprocesseurs s'adapte très bien au traitement d'images du fait de la décomposition facile des algorithmes image en tâches. Le parallélisme fonctionnel de ces architectures est un outil idéal pour conserver la souplesse de description des tâches que nous avons au niveau des algorithmes.

Ces architectures possèdent des processeurs répartis habituellement en deux catégories : des processeurs spécialisés ayant une architecture SIMD et utilisés pour réaliser des traitements au niveau du pixel. Plusieurs opérations matricielles comme les filtrages, les transformations géométriques et les opérations arithmétiques et logiques peuvent être réservés à ces processeurs. La vraie structure MIMD de la machine contient en général des processeurs banalisés réservés à des tâches de plus haut niveau et pouvant avoir des architectures différentes. Des opérations de squelettisation, de regroupement de régions ou de localisation d'objets sont des exemples de tâches qui peuvent être réservées à de tels processeurs.

4.1.3. Le système ICOTECH

A) Présentation générale

Le projet ICOTECH est né de la volonté de L'Ecole Nationale Supérieure de Physique de Strasbourg de développer une machine de traitement d'images qui soit aussi polyvalente que possible et offrant des possiblités de traitement interactif à tous les niveaux du système sans faire appel à des moyens de programmation poussés.

L'idée de vouloir couvrir à la fois des domaines aussi variés que la télédétection, l'imagerie médicale, et la C.A.O a conduit à réaliser un système modulaire comprenant des unités de traitement interchangeables et configurables. Le système de base est un processeur vidéo contenant les modules spécialisés nécessaires au traitement de bas niveau. Ce noyau qui est un poste de travail, évolue dans un environnement architectural plus important contenant des processeurs plus importants. Ceci permet de disposer d'une architecture graduellement complexe se complétant en fonction des besoins. L'ensemble des modules, organisés selon une architecture multiprocesseurs, est géré par un ordinateur hôte intégré au système.

Dans la configuration la plus étendue du système, la programmation du traitement d'images est effectuée sous forme de tâches qui sont réparties sur les différents processeurs présents. La gestion concomitante est assurée par un logiciel spécialisé qui tient compte des aspects multi-utilisateurs, multi-tâches, et temps réel de la machine.

De surcroît, le public auquel est destiné ICOTECH n'est pas toujours un public d'informaticiens. Les moyens de gestion mis en place tiennent compte de cette contrainte et assurent la complète transparence au niveau de la programmation. De ce fait, la programmation du traitement d'images sur ICOTECH est réalisée par appel à un ensemble de sous-programmes de haut niveau exécutant des tâches sur les processeurs en place et cachant ainsi à l'utilisateur les aspects liés à la gestion interne des processeurs.

B) Structure du multiprocesseurs PRIM

La partie multiprocesseurs d'ICOTECH est appelée PRIM. la structure de PRIM a été définie à partir des considérations de modularité, de souplesse et de transparence évoquées ci-dessus. Elle est organisée autour du multibus 1 de INTEL. Elle comprend dans sa version actuelle [TED 86] cinq processeurs spécialisés, un processeur vidéo (PV) et une mémoire d'image.

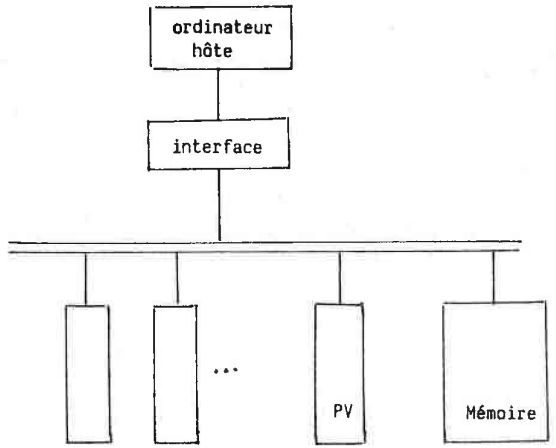


Figure 5.4 : Structure générale du système ICOTECH.

Les processeurs spécialisés disponibles sont :

◆ un processeur de transfert : *qui assure le transfert du contenu de la mémoire du PRIM dans celle du PV. La rapidité à laquelle le transfert peut être effectué permet de faire des animations d'images.*

◆ un processeur interimage : *permet d'effectuer toute sorte de combinaison linéaire ou non sur 2, 3 ou 4 images, parmi elles notons la comparaison de deux images, la déconvolution, l'augmentation de contraste et les manipulations de couleurs dans les images couleurs.*

◆ un processeur de convolution : *le masque utilisé a une taille de 16x16 (16 bits). La convolution est effectuée sur des images de 8 ou 16 bits par l'intermédiaire d'un multiplexeur et d'une table de transfert.*

◆ un processeur de "transformée de Fourier" : *le calcul de la FFT se fait sur des images de taille rectangulaire. La sortie du résultat se fait sous forme complexe avec partie réelle et partie imaginaire sur 22 bits. Le processeur FFT est réalisé pour effectuer des transformations plus générales décomposables en calculs sur les lignes et les colonnes. Nous trouvons les transformées en cosinus, en sinus et les transformées binaires rapides de Walsh, Paley et Hadamard.*

◆ un processeur de transformation géométrique : *permet d'effectuer les opérations suivantes : translation en x et y, loupe en x et y, rotation et perspective sans altération des valeurs radiométriques.*

◆ un processeur vidéo : *le processeur vidéo est considéré par l'ordinateur hôte comme un processeur spécialisé. Il peut fonctionner en parallèle avec les autres processeurs spécialisés pour réaliser des tâches concurrentes et partager avec eux les différents modules mémoire. Il peut aussi fonctionner seul (off line). De cette manière, il constitue un poste de travail avec une CPU, une mémoire locale d'image (512 x 512 x 16 bits) et des processeurs spécialisés fonctionnant en temps réel tels que l'histogramme, la LUT, le zoom et les modules d'entrée-sortie.*

C) Gestion du PRIM

L'ensemble du PRIM est utilisé à partir de l'ordinateur hôte à travers un logiciel d'interface appelé GESPRIM86. GESPRIM86 reçoit les programmes des utilisateurs, les distribue sous forme de tâches sur les processeurs dont il assure la gestion. Il arbitre en plus, dans le cadre de la programmation multi-utilisateurs, l'accès aux ressources et assure leur partage de manière équitable entre les différents utilisateurs.

Le PRIM est vu par GESPRIM86 comme un ensemble de ressources communes (bus, processeurs spécialisés et mémoire de travail) partagées entre les différentes tâches. A chaque ressource est associé un **sémaphore** dont la structure précise :

- l'état de la ressource,
- l'identification de la tâche utilisant la ressource,
- la liste des tâches voulant utiliser la ressource gérée en mode "FIFO",
- le nombre courant de demandes d'accès à la ressource et le nombre maximum autorisé.

GESPRIM86 évolue dans l'environnement de programmation du système IRMX86. Il se sert des outils de ce système pour réaliser la gestion de la synchronisation et de la communication entre les tâches.

Au moment d'une requête lancée à GESPRIM86, ce dernier cherche le processeur capable d'effectuer la tâche. Un processeur n'est activé par le logiciel de gestion que lorsqu'il est libre et que ses arguments sont disponibles. Dans le cas contraire, la requête est rangée dans une file d'attente et une autre requête est prise en compte. La file d'attente est consultée à chaque nouvel évènement correspondant soit à une libération des ressources soit à la disponibilité d'un argument.

D) Programmation du PRIM

Le PRIM est utilisé à travers un langage de commandes dont la fonction est de permettre à l'utilisateur de communiquer avec le système via un système interactif. Le langage de commandes offre une gamme importante d'utilitaires et sert de base au test de sous-programmes de traitement qui peuvent être intégrés dans un programme d'application.

Le langage évolue dans un environnement logiciel appelé IMAGE8 [VOG 84] qui offre des utilitaires de gestion des images indépendemment des supports sur lesquels elles se trouvent. IMAGE8 permet à l'utilisateur de travailler sur des images multiformats en l'affranchissant de tout problème de réservation mémoire, de manipulation de fichiers et de gestion de périphériques. Par des règles de formattage simples, il permet une standardisation de l'accès aux images indépendemment du support physique.

Le logiciel est écrit pour supporter tous les formats courants de stockage d'images sur périphériques : images entrelacées (canal par canal, pixel par pixel) ou non, mono- ou tri-chromes avec une dynamique radiométrique et une résolution variable. Il permet :

- d'accéder à des parties ou à l'ensemble de l'image,
- de gérer la mémoire centrale par transfert d'images depuis les périphériques (buffering),
- de définir des images virtuelles,
- de gérer l'historique des images.

IMAGE8 contient une bibliothèque de sous-programmes utilisables suivant plusieurs modes (rapide, lent, rapide avec test, ...)

E) Mise en évidence du parallélisme

Afin d'obtenir des performances élevées de la machine ICOTECH, on détecte dans le programme LPSI les opérations spécialisées qui peuvent être exécutées par ses processeurs. Ainsi, l'objectif que nous avons poursuivi est de trouver la solution la plus simple qui allie au mieux le parallélisme du programme LPSI au parallélisme d'ICOTECH.

Parmi les opérations spécialisées que l'on cherche à détecter, nous trouvons :

a) les entrées-sorties

Ces instructions regroupent :

- *acquisition depuis un support physique (caméra, disque, bande...) , avec possibilité de prétraitement,*

- *visualisation avec action sur la couleur, la taille, le grossissement, l'aspect, ...*

- *rangement sur un support externe.*

b) les opérations inter-images

Ce sont essentiellement les combinaisons linéaires et non linéaires contenant des opérations arithmétiques sur des images : multiplication d'image par image, d'image par constante, addition et soustraction d'images, etc...

c) les opérations sur voisinages

Ces opérations sont limitées à la convolution car les opérations de morphologie mathématique n'ont pas été prévues.

D'autres tâches peuvent être exécutées par ICOTECH mais ne seront pas détectées dans les programmes LPSI du fait de la difficulté de leur associer un modèle d'exécution. Ces tâches relatives à la transformée de Fourier et à des programmes de transformations géométriques ou de granulométrie (effectuant des mesures de surface, de périmètre dans les images binaires) nécessitent plusieurs instructions.

F) Expression de tâches parallèles

Les différentes tâches signalées précédemment sont exprimées explicitement ou implicitement dans le programme. LPSI dispose d'un certain nombre de primitives pour exprimer le parallélisme. Parmi ces primitives, nous trouvons :

a) Les expressions globales :

LPSI dispose d'une gamme importante d'opérateurs arithmétiques et logiques binaires pouvant s'appliquer sur des variables image. Ces opérateurs ont une sémantique bien propre et n'ont de sens que sur des variables d'un type donné. Ainsi, l'affectation suivante :

$$I := \lambda 1\ I1 + \lambda 2\ I2$$

exprime une combinaison linéaire entre une image I1 et une image I2. Les deux opérations de multiplication par les constantes peuvent être prises en compte de manière indépendante et exécutées par le processeur de

multiplication.

Comme nous l'avons vu dans la descripition du langage (cf. chapitre 3), les opérateurs arithmétiques et logiques sont empruntées à PASCAL pour définir les mêmes types d'opérations mais de façon globale sur des variables image.

b) Les itérations :

Les itérations LPSI de type **forall** sont des boucles parallèles dans lesquelles toutes les instances des instructions sont indépendantes. Elles constituent d'autres manières d'exprimer les opérations arithmétiques et logiques globales indiquées précédemment. La répétition se fait sur des pixels ou des ensembles de pixels comme des lignes, des colonnes, des voisinages de pixels, etc... Le traitement répété peut être plus compliqué que celui des opérations arithmétiques et logiques de dessus. De cette manière, elles couvrent toutes les opérations répétitives sur les objets image.

Les opérations séquentielles de type **forseq** imposent généralement un ordre de parcours sur les objets. Elles seront difficilement exploitables pour le parallélisme car elles expriment souvent des tâches dépendantes.

c) Les tâches

On exprime dans LPSI les tâches independantes à l'aide de l'instruction **Cobegin .. Coend**. Ces tâches peuvent être des opérations arithmétiques et logiques ou des sous-programmes. Toutes ces tâches seront susceptibles d'être exécutées par les processeurs spécialisées d'ICOTECH si leur action correspond à l'action de l'un de ces processeurs.

G) Détection du parallélisme

Dans certaines architectures MISD ainsi que dans les machines séquencées par les données, la détection est faite à l'exécution car il s'agit d'une parallélisation au coup par coup. Mais dans le cas que nous étudions où parfois toute une partie du programme est transformée, le moment idéal pour le faire est avant la compilation, plus précisément à la précompilation, c'est-à-dire au moment où la structure du programme est analysée, les identificateurs et les opérateurs connus.

Pour faciliter cette démarche, le programme est analysée syntaxiquement puis rangé sous la forme d'un **arbre syntaxique abstrait**. Ce dernier met bien en évidence la structure du programme et surtout celles des opérations auxquelles on s'intéresse [BOU 87]. La figure suivante montre le schéma général d'analyse d'un programme LPSI.

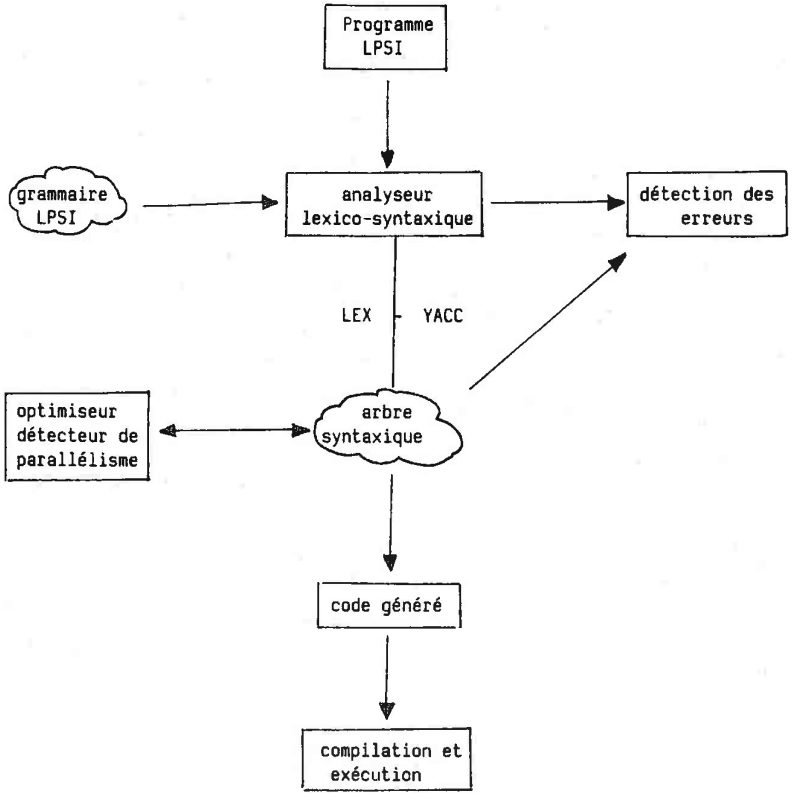


Figure 5.5 : Schéma général d'analyse d'un programme LPSI.

Compte tenu de la petite quantité de parallélisme recherchée, de la taille des programmes, nous pensons nous limiter aux algorithmes de détection les plus simples évitant ainsi la lourdeur de ces mécanismes. En réalité, nous limitons la détection aux expressions arithmétiques sur les images qui peuvent être exécutées par les processeurs de la machine. Ces expressions seront détectées directement dans les opérations image globales (parallélisme explicite) ou après analyse des instructions itératives (parallélisme implicite).

a) Le parallélisme explicite

Le parallélisme explicite concerne les opérations ponctuelles ou de voisinage écrites globalement comme suit :

A + B * C - D A ⊗ m A(F) - B

où A, B, C et D sont des images, m est un masque et F est un filtre.

Ces opérations seront facilement détectées à partir de l'arbre syntaxique. La nature des opérateurs et le type des opérandes conduisent à les trouver immédiatement. Cependant, sous cette forme, ces opérations ne sont pas prêtes à l'exécution. Un travail d'optimisation précède celui de la génération de code. Il consiste à ordonner les opérations élémentaires et les séparer dans l'expression afin de pouvoir engendrer pour chacune d'entre elles, l'appel adéquat au processeur spécialisé qui doit l'exécuter. Ce travail de préparation est de nature simple. Il s'agit essentiellement d'ajuster les dimensions des images de telles manières à ne réaliser l'opération que sur les éléments utiles :

- une image trop grande ne peut être traitée en une seule fois ! Il faut procéder à des partitionnements et des collages. Une image trop petite doit être agrandie pour s'adapter à des standards de processus. (Cette solution est normalement prévue par Image8).

- il faut éviter de faire l'opération sur tous les éléments des opérandes si ces derniers n'ont pas la même taille. Ceci concerne essentiellement les opérations arithmétiques et logiques. Dans la première expression ci-dessus, A et B peuvent avoir les mêmes dimensions, C et D aussi mais différentes de celles de A et B. Dans ce cas, on réduit celles des plus grandes de manière à avoir partout des dimensions égales. Ce travail n'est intéressant que si les dimensions des images sont plus grandes que celles des paramètres image des processeurs.

- les opérandes peuvent être filtrés. Les filtres réduisent en général le champ d'accès aux éléments des opérandes. Si les filtres sont des fenêtres rectangulaires, alors on utilise les techniques d'ajustage précédentes, sinon il ne faut normalement actionner le processeur que sur les points sélectionnés, ce qui n'est pas possible.

b) Le parallélisme implicite

Le parallélisme implicite concerne aussi les opérations spécialisées écrites sous une forme moins directe que précédemment et pour lesquelles nous possédons des schémas d'exécution. Il s'agit dans ce cas de les détecter et de les mettre sous une forme qui peut être prise en compte par un processeur.

Les schémas d'exécution indiquent les opérations élémentaires clés et un agencement possible de ces opérations, ce qui peut aider à les localiser dans l'arbre syntaxique. La description approximative des opérations par ces schémas d'exécution est suffisante pour la détection des opérations spécialisées et différentes. Il est sûr que dans une

démarche plus précise où l'on cherche à détecter des opérations différentes mais syntaxiquement semblables (au sens de l'approximation), une plus grande rigueur serait nécessaire.

Nous pouvons trouver dans [BOU 87] plus de détail sur la détection du parallélisme implicite. Je me contenterai ici d'en faire une illustration rapide sur quelques exemples.

exemple 1 : **calcul d'histogramme**

L'histogramme se calcule à partir d'une image de niveaux de gris en répétant pour chaque valeur dans l'image, un ajout de 1 à son nombre d'occurence rangé dans une fonction tabulée qui est l'histogramme. Ce calcul peut s'exprimer en LPSI comme suit :

```
forseq v$ in I do
histo[v$] := histo[v$] + 1
```

Ce n'est pas la seule façon de l'exprimer mais elle se trouve être la plus courante. Les **indices** qui vont nous permettre de savoir qu'il s'agit bien d'un calcul d'histogramme sont :

- la répétition sur l'image I par emploi de l'indice v$,
- le traitement sur une variable de type "fonction tabulée",
- l'ajout de 1 à histo dans le traitement et l'affectation du résultat au même histogramme.

Ces indices seront faciles à trouver dans l'arbre syntaxique si l'opération existe.

exemple 2 : convolution par un masque

La convolution est calculée à partir d'une image et d'un masque. L'opération de convolution est répétée identiquement sur tous les voisinages des pixels. Elle s'effectue en multipliant d'abord le masque par le voisinage point à point puis ensuite en totalisant les produits ainsi obtenus. On peut l'exprimer en LPSI sous la forme suivante :


```
s := 0;
forall p in I do
begin
   forseq v$ in pvois(I,p) x m do
        s := s + v$;
   I[p] := s;
end;
```

L'indice le plus informateur sera ici l'opérateur de multiplication par un masque. Cette information sera ensuite complétée par d'autres relatives à l'environnement qui doit permettre de trouver les boucles : **forseq** qui confirme la recherche séquentielle de tous les produits, puis de la boucle parallèle **forall** qui confirme le nom du pixel courant p utilisé dans les traitements ainsi que celui de l'image.

A partir de ces précisions, il sera ensuite facile de générer l'appel au convolueur d'ICOTECH. Les types des paramètres et leurs dimensions sont donnés dans l'arbre syntaxique.

c) Exemple de programme :

Nous reprenons l'exemple de la fonction d'extraction de contours données dans le chapitre 3 :

```
fonction extcont(i : ti ; w : fenetre) : ti;
    ...
    begin

        readg (sup1) i; (* lecture de i sur support1*)
        readg (sup2) m; (* lecture de m sur support2*)
        j := i; (* duplication de i *)

        (*tâche 1 : *)

            C1 := (i(w) * m)(v$ > S)

        (*tâche 2 : *)

            forall v$ in j(w) do  h[v$] := h[v$] + 1;
            for i := 2 do h.L do h[i] := h[i] + h[i-1];

            forall p in j(w) do i2[p] := h[j[p]];

            C2 := (i2 - j(w))(v$ > S);

        extcont := C1 + C2

    end;
```

Cette fonction contient deux tâches indépendantes pouvant se dérouler en parallèle. La première exprime une convolution suivie d'un seuillage. Ces deux opérations sont locales et peuvent s'exécuter chacune par un processeur spécialisé de la machine ICOTECH. Le processeur de seuillage devra attendre la fin de la convolution.

La deuxième tâche est composée de trois actions : la première calcule l'histogramme h égalisé en ajoutant à chaque fois 1 à tous les éléments d'indice supérieur ou égal à la valeur du point courant dans l'image. Cette opération peut être réalisée directement par le processeur "histogramme" du PV. La deuxième calcule i2 par transformation ponctuelle et parallèle de j à partir de h. h jouant ici le rôle d'une anamorphose, l'égalisation d'histogramme peut être effectuée par un processeur de transformation par les LUT du PV. Il suffit pour cela de charger une LUT avec l'histogramme égalisé et d'appeler ensuite le processeur spécialisé pour exécuter la transformation.

Enfin, la troisième calcule C2 par soustraction de j à i2 et seuillage. Là aussi, il s'agit de deux opérations dépendantes puisqu'il faudra d'abord faire la différence avant d'exécuter le seuillage.

Il reste l'opération d'addition des images de contours C1 et C2. Elle peut être réalisée facilement par le processeur inter-image d'ICOTECH.

Les étapes de génération de code sont les suivantes :

- génération de l'analyseur syntaxique à partir de la grammaire LPSI par les générateurs LEX-YACC,

- représentation du programme LPSI, donné de l'analyseur, sous la forme d'un arbre de syntaxe abstraite,

- parcours de l'arbre syntaxique et traduction systématique des unités syntaxiques du programme. A chaque unité syntaxique correspond un schéma de traduction connu par le système. La traduction se fait localement sans aucune incidence sur le reste du code.

Le code PASCAL généré pour le programme précédent est :

```
(* tout ce qui se termine par _ est généré par le
   pré-compilateur *)
```

---

lecture et initialisation des données

```
adr1_ := addr(i);  (* adr1 est un pointeur sur
readg(sup1,adr1_); la structure de l'image i *)

(* Toutes les données image sont manipulées par
adresse de manière à pouvoir disposer d'une structure
souple sans contrainte sur la taille et sur le filtre
associé. *)

adr1_ := addr(m);  (* adr1 est un pointeur sur la
readg(sup2,adr1_);    structure du masque m *)

adr1_ := addr(j);
adr2_ := addr(i);
affim_(adr1_,adr2_); (* affectation de i à j *)
```

calcul de la convolution

```
(* rangement du filtre dans la structure de l'image *)

        (* préparation de la structure filtre w *)
        new(i.ch5);
        i.ch5^.filt := fwin_;
        (* rangement du filtre  dans la structure de l'image *)
        i.ch5^.fw := w;

(* appel de la convolution *)

        (* préparation des adresses de l'image et du masque *)
        adr1_ := addr(i.ch2^);
        adr2_ := addr(m.ch2^);
        (* convolution *)
        convolution (adr1_, adr2_, imageinto_^.ch2^);
```

calcul de l'image de contour c1

```
(* préparation de la structure filtre : v$ > S *)

        new(filter1_);
        new(filter1_^.f);
        new(imageinto_);
        new(imageinto_^.ch5);
        imageinto_^.ch5^.filt := fbin_;
        imageintô.ch5.fb := filter1_;

        (* évaluation du filtre *)

        for indh_ := 1 to imageinto_^.H do
           for indl_ := 1 to imageinto_^.L do
                if imageinto_^.ch2^[indl_,indh_] > S
                then filter1_^.f^[indh_,indl_] := 1
                else filter1_^.f^[indh_,indl_] := 0;

        (* calcul de c1 *)

        adr1_ := addr(c1)
        affim_(adr1_,imageinto_);
        release(filter1_)
```


égalisation d'histogramme

```
(* calcul de l'histogramme *)

        (* préparation de la zone de calcul de l'histogramme

           dans l'image j *)
        new(j.ch5); (* image j *)
        j.ch5^.filt := fwin_; (* image j limitée à la fenêtre w *)
        j.ch5^.fw := w;

        (* calcul de l'histogramme *)

        for indh_ := 1  to inf_(j.H , w.H) do
           for indl_ := 1 to inf_(j.L,w.L) do
                begin
                        z_ := j.ch2^[indh_,indl_];
                        h.ch2[z_] := h.ch2[z_] + 1
                end;

(* calcul de l'histogramme cumulé *)

        for r := 2 to h.L do h.ch2[r] := h.ch2[r-1];

(* égalisation d'histogramme *)

        new(j.ch5);
        j.ch5^.filt := fwin_;
        j.ch5^.fw := w;
        for indh_ := 1 to inf_(j.H,w.H) do
          for indl_ := 1 to inf_(j.L,w.L) do
          begin
                z_ := j.ch2^[indh_,indl_];
                i2.ch2^[indh_,indl_] := h.ch2[z_]
          end;

(* calcul de la différence de i et de j *)

        j.ch5^.filt := fwin_;
        j.ch5^.fw := w;
        adr1_ := addr(i2.ch2^);
        adr2_ := addr(j.ch2^);
        moinsim_(adr1_,adr2_,imageinto_);
```

calcul de l'image de contour c2

```
(* préparation de la structure filtre *)

        new(filter1_);
        new(filter1_^.f);
        new(imageinto_^.ch5);
        imageinto_^.ch5^.filt := fbin_;
        imageintô.ch5.fb := filter1_;

(* évaluation de la structure filtre *)

        for indh_ := 1 to imageinto_^.H do
           for indl_ := 1 to imageinto_^.L do
                if imageinto_^.ch2^[indl_,indh_] > S
                then filter1_^.f^[indh_,indl_] := 1
                else filter1_^.f^[indh_,indl_] := 0;
        adr1_ := addr(c2); (* c2 est une autre image de contour *)
        affim_(adr1_,imageinto_);
        release(filter1_)

(* addition de c1 et de c2 *)

        adr1_ := addr(c1.ch2^);
        adr2_ := addr(c2.ch2^);
        addim_(adr1_,adr2_,imageinto_);

(* affectation du résultat final à extcont *)

        adr1_ := addr(extcont);
        affim_(adr1_,imageinto_);

end.
```


4.2. Les machines pyramidales

Nous avons décrit dans les chapitres précédents les caractéristiques importantes des architectures pyramidales ainsi que les classes d'algorithmes qui s'adaptent sur elles. Cette partie présente les traits de conception essentiels de la machine pyramidale SPHINX fondés sur la **programmation associative** ainsi que l'étude détaillée d'un algorithme de chaînage de contours pouvant être exécuté sur une machine de ce type. Par cet exemple, nous montrerons les avantages que procurent de tels systèmes même dans des cas mal conditionnés pour le parallélisme. Nous remarquerons alors les inconvénients relatifs à la rigueur du style de programmation qu'imposent de telles architectures.

4.2.1. Notion d'associativité

Le principe de l'associativité repose sur le fait que c'est le contenu du mot mémoire qui détermine la transformation qu'il va subir. Les types de traitement que ce code induit (recherche d'attributs, comptage de mots ayant une valeur donnée, etc...), imposent dans la réalisation matérielle de machines spécialisées, d'associer à chaque mémoire des capacités de recherche et de transformation adéquates. L'indépendance qui caractérise ces traitements conduit souvent à les paralléliser [MER 83].

En traitement d'images, plusieurs opérations sont associatives. Ces opérations s'appliquent sur des pixels vérifiant un certain critère lié à leur valeur et/ou position dans l'image. Ces opérations sont souvent identiques pour tous les pixels désignés.

Les machines associatives existantes proposent différents degrés de parallélisme. Ces machines comportent des unités de traitement plus ou moins sophistiquées liées aux mots mémoire. Le type d'interconnexion de ces unités et leur mode de fonctionnement (par bit, par mot ou par groupe de mots) réalisent cette logique associative.

Une autre caractéristique importante de l'associativité est le **mouvement de données** que cela entraîne par l'interrogation des mots mémoire en vue de répondre aux requêtes. Pour permettre d'accélérer ces mouvements de données, plusieurs structures de machines ont été proposées. Parmi elles, les structures arborescentes ont requis le maximum de succès à cause des performances réalisées. C'est ainsi que fût étudiée l'une des premières machines pyramidales pour le traitement d'images [DYE 81].

4.2.2. La machine SPHINX

La pyramide SPHINX est bâtie selon un double mode d'interconnexion de cellules élémentaires appelées PE : une interconnexion en une arborescence binaire entre les étages et une interconnexion orthogonale de type 4-connexité au sein d'un même étage.

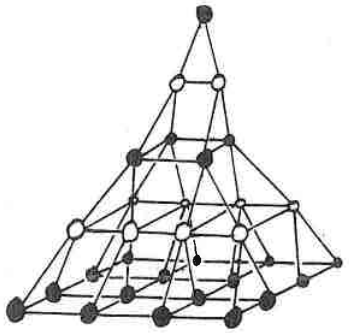


Figure 5.6 Structure générale de la machine SPHINX.

A) L'architecture d'un PE

Un PE comprend une UAL, une mémoire locale de 64 mots de 1 bit, des registres et des possibilités de communication avec les voisins, le père et les fils. L'architecture d'un PE est tourbionnaire, c'est-à-dire que tous les transferts de données passent par l'UAL, y compris les déplacements de données au sein de la mémoire. Seuls les transferts avec les fils, qui concernent 2 bits à la fois, utilisent un chemin différent. L'UAL effectue des opérations binaires de type addition, soustraction, incrémentation et comparaison.

L'adressage de la mémoire est réalisée à l'aide de trois pointeurs et par un bus qui permet l'adressage immédiat d'un bit de la mémoire. Les PE sont de type bit-série. Ils traitent à la volée les bits qui arrivent de la mémoire. Les PE réalisent, en plus des instructions de sélection des registres et le type de ressource sur le bus, plusieurs autres instructions comme :

- les instructions de transfert de données,
- les opérations arithmétiques et logiques,
- les instructions concernant les pointeurs (incrémentation et décrémentation),
- les chargements immédiats.

Exemple :

ADD/ PT1 VN.RA : RC signifie : ajouter le complément du contenu de la mémoire pointée par PT1 au contenu du registre A du voisin nord, et ranger le résultat dans le registre RC.

B) La communication entre les PE

Deux types de communication existent : la communication horizontale et la communication verticale.

a) La communication horizontale

Une connexion orthogonale physique existe entre les voisins d'un même étage. Chaque processeur fait une opération utilisant une donnée interne (sur le bus correspondant) et une donnée reçue du voisin, ce qui permet un calcul à la volée, et permet que chaque PE ait directement accès à l'ensemble de la mémoire de ses voisins. La communication entre les PE d'un même étage ne se fait donc que par lecture de la mémoire locale du voisin. Tous les PE sont **synchrones** par étage. Il existe un contrôleur par étage qui commande tous les processeurs de cet étage.

b) La communication verticale

Elle concerne des transferts entre des étages exécutant des instructions différentes. L'émission ou la réception des données passent par l'intermédiaire d'un registre "boîte aux lettre" associé à chaque PE. Des vérifications sont à faire par logiciel pour gérer les conflits d'accès à ces registres. La communication entre étages est donc **asynchrone**.

C) L'environnement de programmation

La gestion de la pyramide est assurée à partir d'un ordinateur hôte. L'environnement de programmation comprend :

- un interpréteur de commandes dont la structure est adaptée à celle de la pyramide pour assurer rapidement le contrôle de flots d'instructions vers les étages et vers les PE d'un même étage,

- une unité rapide d'entrée-sortie,

- une unité de gestion de la mémoire d'image et segmentation des traitements. La mémoire d'image contient les différentes images sources ou résultats ainsi qu'un certain nombre de données intermédiaires, ce qui permet de suppléer à la faible capacité de stockage de la pyramide. Un module de gestion est affecté à cette mémoire. L'importance des transferts des données entre les étages et la rapidité que ceux-là imposent, conduisent à utiliser une pyramide virtuelle programmable.

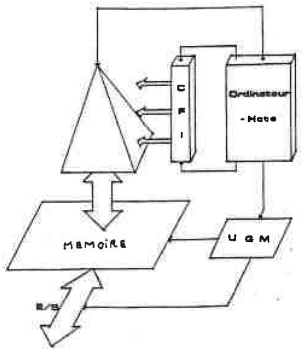


Figure 5.7 : Environnement de l'architecture de SPHINX.

4.2.3. Les aspects algorithmiques

La méthode de programmation à prendre en compte doit tenir compte des caractéristiques de la machine, à savoir plusieurs niveaux de parallélisme simultanés. Sur un même étage, les PE exécutent la même instruction (parallélisme SIMD) et les différents étages exécutent des instructions différentes (parallélisme multi-SIMD). **A) Les différentes opérations**

La stratégie de programmation est différente sur cette machine. Elle dépend du type d'opération effectuée comme suit :

a) Cas des opérations internes

Les opérations internes sont les opérations arithmétiques et logiques. Ces opérations n'utilisent qu'un seul étage de la pyramide et aucun transfert horizontal. Plusieurs d'entre elles peuvent s'exécuter en parallèle.

b) Cas des opérations sur voisinage

Les opérations sur voisinage combinent les données présentes dans un PE et celles dans les PE voisins. A l'aide des liaisons orthogonales entre les PE il est facile de lire les valeurs des voisins.

c) Cas des opérations de calcul de propriétés

Un exemple de calcul de propriétés est celui d'indiquer la présence d'un attribut donné dans la mémoire. On peut supposer que cet attribut est booléen ou qu'une requête intermédiaire peut fournir un attribut booléen. Il s'agit dans ce cas de faire le "OU" de tous ces attributs. Pour cela, chaque étage fera le "OU" de ses fils et transmettra le résultat à son père.

B) Les mouvements de données

La pyramide possède trois voies de communication permettant d'échanger de l'information avec l'extérieur :

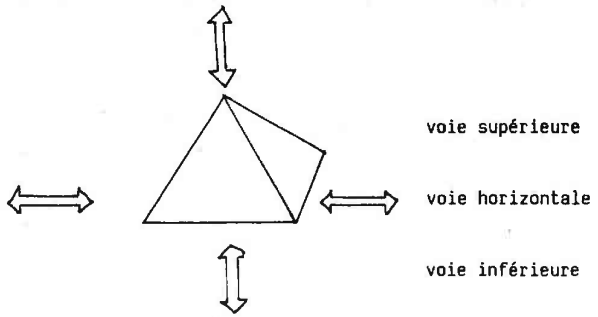


Figure 5.8 : Mouvements de données dans SPHINX.

La voie supérieure utilise la communication vers le père du dernier étage, la voie horizontale, les communications de voisinage des PE périphériques, et la voie inférieure, les chemins de donnée vers les fils du premier étage.

a) Communication supérieure

La communication par la voie supérieure ne pose pas de problème dans le cas où la réponse est unique (calcul de prédicat). Elle est plus complexe dans le cas de transfert de plusieurs données. Il y a, en effet, risque de "collisions", et il est nécessaire d'effectuer une résolution de réponses multiples. C'est un des mécanismes de base des mémoires associatives. Quelle que soit la requête ou l'attribution effectuée, il est probable que plusieurs cellules répondent.

Dans SPHINX, on cherche à définir un chemin unique dans l'arbre des PE permettant d'atteindre une des cellules ayant un certain attribut. Ce chemin étant ouvert, il est facile ensuite d'y faire passer les informations souhaitées. Plusieurs idées sont données dans [MER 83] pour

économiser les transferts de données et faire certaines opérations de transfert et de gestion de la pyramide en parallèle.

b) Communication inférieure

La voie de communication inférieure assure le transfert entre la pyramide et la mémoire d'image. Elle permet de "remplir" la pyramide par les données présentes en mémoire ou au contraire de la "vider" pour sauvegarder son contenu. C'est ce qui est appelé "mécanisme de pompage".

c) Communication horizontale

Cette communication est nécessaire lorsque l'on effectue des opérations mettant en jeu les communications entre voisins. Un mécanisme cablé mettra en communication la mémoire avec la voie horizontale. Ce mécanisme est étroitement lié à l'organisation mémoire.

C) Les Langages

L'ETCA développe actuellement deux types de langage pour la machine SPHINX. Un langage pyramidal de très bas niveau qui n'est autre qu'un interface de programmation du contrôleur CFI associé à la pyramide. Ce langage comportera un ensemble de procédures écrites en assembleur qui auront pour rôle de distribuer d'une part les tâches entre les étages de la pyramide et d'ordonner d'autre part le traitement de certains PE d'un même étage.

Un autre langage de plus haut niveau permettra la programmation directe d'actions associatives. Ce langage sera vu comme l'assembleur de la machine. Il sera assez général pour programmer la pyramide et utilisé pour différentes applications. Il comportera :

- des déclarations de variables associées aux mémoires locales des PE. Chaque variable sera donnée par sa structure (suite de bits dans la mémoire locale), sa longueur (nombre de bits) et sa nature (étages d'allocation),

- des modes de balayage des zones mémoire associées aux variables,

- des instructions de contrôle classiques,

- des appels de procédures. Les procédures appelées sont surtout celles de l'interface. Elles permettront de programmer les mouvements de données entre les étages (transfert des informations des fils vers les pères et réciproquement) et d'actionner les PE d'un étage pour réaliser des opérations locales. Les paramètres de ces procédures sont des paramètres sources ou constantes.

LPSI s'adapte parfaitement à ce type d'architecture. Il permet la déclaration de pyramides virtuelles et fournit un grand nombre

d'opérations assurant les mouvements de données entre PE vus précédemment. L'idée de l'associativité a été intégrée par la notion de filtre associatif et par les structures de contrôle adaptées à la projection et surtout à la réduction des données.

Il est tout à fait normal que nous ne puissions pas réaliser toutes les opérations sur la pyramide à travers SPHINX car LPSI n'est pas un langage pyramidal. La phase d'optimisation et de génération de code doit permettre de détecter certaines opérations pour lesquelles nous disposons d'un schéma de traduction.

La figure suivante montre le schéma d'utilisation de la machine SPHINX à partir de LPSI via les langages d'interface :

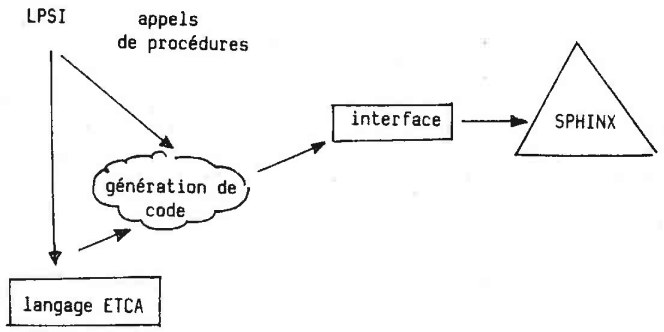


Figure 5.9 : Schéma de programmation de SPHINX.

L'interface est un langage de très bas niveau comprenant des commandes écrites en assembleur et destinées à la programmation du contrôleur du flot d'instructions et par conséquent de la pyramide. LPSI pourrait utiliser directement ces commandes ou être traduit dans le langage intermédiaire, comme le montre le schéma ci-dessus.

D) La détection du parallélisme

Le programme écrit en LPSI est analysé puis rangé sous la forme d'un arbre syntaxique. L'optimiseur-détecteur de parallélisme localise les opérations pyramidales et génère des appels à des fonctions spécialisées des langages d'interface qui réalisent effectivement ces opérations.

Suivant qu'il s'agit d'opérations globales ou de sous-programmes réalisant des tâches plus complexes, le parallélisme sera détecté de manière directe ou indirecte.

a) Le parallélisme explicite

Là-aussi, le parallélisme explicite concerne les opérations ponctuelles et locales pouvant être traitées indépendemment par les étages de la pyramide. Il s'agit essentiellement des opérations arithmétiques, logiques et de filtrage vues précédemment sur les images.

Elles nécessitent les mêmes préparations que celles vues dans ICOTECH en adaptant la taille de l'image à celle de l'étage d'exécution. Quand l'image a une taille plus grande que celle de l'étage qui la traite, une solution consiste à replier l'image dans les mémoires des processeurs de l'étage. Or, vues la capacité des mémoires locales et la complexité de la gestion des transferts mémoire-registre de traitement, cette solution ne paraît pas des plus judicieuses.

b) Le parallélisme implicite

La mise en forme des instructions pyramidales n'est pas facile car le traitement est généralement décomposé en plusieurs parties qui sont distribuées sur plusieurs processeurs. Etant donnée la structure hiérarchique de la pyramide, le traitement exprime à la fois la répartition, la décomposition du travail et la fusion des traitements élémentaires par les différents processeurs.

L'analyseur de code écrit par Boufriche [BOU 87] permet de vérifier si des sous-programmes récursifs ayant pour paramètre une image, ne peuvent pas se mettre sous une forme plus simple exprimant à la fois la division de l'image, le traitement sur les parties extraites de l'image et la fusion des traitements. Ce modèle peut s'écrire comme suit :


```
procedure frecursive(image,taille);

debut

        si condition alors (* renvoi du résultat *)

        sinon
                (* divsion de l'image *)

                divise(image,taillel,taille2);

                (* divise est un programme qui divise l'image en
                   deux, une fois suivant le sens de la largeur et
                   une autre fois suivant le sens de la longueur.
                   La longueur d'un côté permet de déterminer le
                   le sens de la division *)

                (* traitement sur les deux subdivisions *)

                        frecursive(image,taillel);
                        frecursive(image,taille2);

                        (* fusion des traitements *)

                        ....
                fsi

        fin frecursive;
```

C'est le seul schéma d'exécution retenu dans le cas de la pyramide. Il permet de servir de modèle à plusieurs sous-programmes récursifs de type réduction.

Nous allons donner dans la suite deux exemples de programmes écrits pour des machines pyramidales. Le premier montre comment peut se faire la transmission de données entre processeurs et insiste plus sur l'aspect communication et synchronisation. Le deuxième est un dossier d'analyse et de programmation d'un problème entier sur une pyramide de type 1-père-4-fils. Il montre les différentes étapes de résolution d'un problème a priori très mal adapté au parallélisme et encore moins au pipeline.

4.2.4. Algorithme d'extraction de données

Plusieurs problèmes se terminent par une extraction des données rangées dans les noeuds de la pyramide dans un ordre donné (parcours classique d'arbre), d'où l'intérêt de cet algorithme.

Ce problème peut être décrit de deux façons :

- soit en considérant les étages comme des processus en parallèle échangeant des messages avec les étages adjacents,
- soit récursivement en termes de père et de fils d'une pyramide (représentée par son sommet). C'est certainement la meilleure description dans le cadre de LPSI.

A) Description par processus

La description par processus peut être réalisée par le langage d'interface écrit par l'ETCA. Un exemple de description par processus est donné par A. Merigot et P. Gardat de l'IEF/Orsay :

processus ?x représente l'attente (bloquante) d'un message depuis **processus** et son affectation à **x** (= variable de rangement du message).

processus !valeur représente l'envoi (également synchronisé) de **valeur** à **processus**. C'est une syntaxe et une sémantique CSP. Tous les étages exécutent en parallèle le même programme (sauf le plan de base).

programme des étages

```
(* dans la suite, les variables sont confondues avec les messages *)

        père ?ordre d'émettre  (* réception à partir de père *)

        filsG !ordre d'émettre  (* émission vers le fils gauche *)
        répéter tantqu'il y a des données
                filsG ?donnée
                père !donnée
        fin répéter

        filsD !ordre d'émettre
        répéter tantqu'il y a des données
                filsD ?donnée
                père !donnée
        fin répéter

        père !fin des données
```


programme du plan de base

```
        père ?ordre d'émettre

        répéter tantqu'il y a des données locales
                père !donnée
        fin répéter

        père !fin des données
```

En particulier, un PE, dans le plan de base, qui n'a rien à transmettre, ne fait pas perdre de temps à l'algorithme. L'idée intuitive est simple : PE attend, avant d'émettre, l'autorisation du destinataire (le père), puis envoie successivement toutes les données filsG, puis toutes les données filsD

B) Description récursive

Un programme, sur une pyramide, est vu comme engendrant un flot de bits issus du sommet. Le programme se décrit alors récursivement par:

```
père (pyramide) : ...

        si  (pyramide = pixel) alors
            père := flot de données local (* éventuellement vide *)
                sinon
                    père := père (filsG(pyramide)) "+"
                    père (filsD(pyramide));
                    (* fusion des traitements dans père *)
                fsi

fin père;
```

remarque : Bien entendu, de tels algorithmes ont tendance à désynchroniser les PE sur un même étage, ce qui est incompatible avec l'archiecture de SPHINX. Une solution est de jouer sur les registres d'inhibition des PE qui ont terminé ; le temps d'occupation d'un étage est alors celui du PE de cet étage qui travaille le plus longtemps.

Dans cette optique de synchronisation, l'accès au filsG se fait de la façon suivante :

logique		implémentation
	:	
	:	
filsG ?donnée	:	donnée := filsG or filsD
	:	avec les registres conditions: CFG
	:	et CFD positionnés respectivement
	:	à 1 et 0 (le flot filsD est = 0)
	:	

Cette synchronisation est nécessaire car sur l'étage inférieur, (exécutant une instruction père !valeur), les PE ignorent s'ils sont filsG ou filsD de leur père. (**ces problèmes de synchronisation ne sont pas à prendre en compte au niveau de LPSI**).

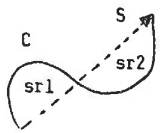
4.2.5. Algorithme d'approximation polygonale de contours

Nous allons présenter maintenant un algorithme de codage de contours filiformes dans une image binaire et sa version pyramidale. Le codage est réalisé par approximation polygonale en utilisant la méthode de Wall et Danielsson. L'approche pyramidale consiste à diviser l'image en quadrants, à appliquer le codage simultanément sur les éléments de contour dans ces quadrants, puis à les concaténer en résolvant les problèmes d'approximation aux frontières. Ce procédé étant récursif, il s'applique de manière identique sur des quadrants plus petits et ainsi de suite jusqu'à atteindre des points de l'image. Cette exemple illustre le gain de temps auquel peut conduire une machine massivement multiprocesseur même sur un problème fondamentalement séquentiel. Il montre aussi que dans ce cas, l'optimisation passe par une structuration plus complexe des données et nécessite un contrôle judicieux du mouvement des données entre les processeurs de la pyramide.

A) L'approximation polygonale

La méthode d'approximation polygonale utilisée est celle de Wall et Danielsson [WAL 84]. Cette méthode, fondée sur la mesure de la surface algébrique, permet de réaliser l'approximation en suivant le contour une seule fois.

Une courbe C est approchée par le segment S qui joint ses deux extrémités si la surface algébrique Φ de C est inférieure à $\lambda|S|$ où λ est un seuil donné et $|S|$ la longueur de S :

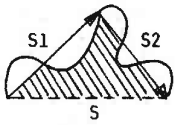


Φ = sr1 - sr2
où sr1 et sr2 sont les surfaces délimitées par C et S

Figure 5.10 : surface algébrique d'une courbe.

Le sens de la courbe influe sur le calcul de la surface, d'où s'est posé pour nous un problème supplémentaire d'**orientation** de contour.

Soient C1 et C2 deux courbes adjacentes approchées respectivement par les segments S1 et S2 et dont les mesures algébriques sont respectivement Φ1 et Φ2, alors la mesure Φ de la surface algébrique totale est égale à :



$$\Phi = \frac{(S1 \hat{} S2)z}{2} + \Phi 1 + \Phi 2$$

Figure 5.9 : Surface algébrique totale de deux courbes.

où (S1 ^ S2)z est la composante en z du produit vectoriel S1^S2 donnant la mesure algébrique du parallélogramme défini par S1 et S2. Si la courbe C2 est réduite à un point, alors la surface algébrique est égale à :

$\Phi = \Phi 1 + \Delta\Phi$ avec $\Delta\Phi = x\Delta y - y\Delta x$

où (x,y) sont les coordonnées du point et $(\Delta x,\Delta y)$, les incréments par rapport au dernier point de la courbe. La figure suivante montre la décomposition de la surface algébrique en surfaces élémentaires et la variation de la précision de l'approximation. La précision dépend, en effet, de l'écart λ que l'on s'autorise sur la largeur du rectangle englobant la courbe depuis l'origine jusqu'au point en cours de traitement.

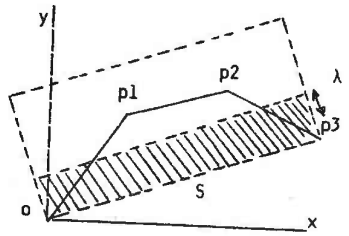


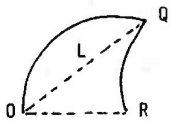
Figure 5.11 : Test du point de rupture.

a) Cas séquentiel

L'algorithme séquentiel suit le contour point par point et assimile ou non le dernier point au segment précédent suivant la précision de l'approximation. En ajoutant de cette manière des points aux segments, on construit un algorithme de l'approximation polygonale de l'image en une seule passe. Il suffit de connaître pour chaque point ses points voisins, ce qui ne nécessite que la présence de trois lignes consécutives de l'image en mémoire.

L'algorithme séquentiel présente néanmoins des limites. En effet, en utilisant uniquement la précision de l'approximation, on risque de

perdre des points anguleux importants. Aux alentours du point anguleux, la détection ne se fait pas car la précision de l'approximation est bonne. Elle se fait donc plus loin, ce qui empêche de prendre en compte les points anguleux et a pour effet de lisser le contour. L'algorithme de Wall et Danielsson prévoyait déjà un test sur les pics en mémorisant les points de la courbe pour lesquels la distance à l'origine est supérieure à celle de l'origine au dernier point. Mais ceci ne résoud pas tous les cas.



cas où $\overline{OQ} > \overline{OR}$:

on mémorise Q pour lequel on a un maximum de L.

Figure 5.12 : Test du point singulier.

b) Cas pyramidal

Aux limitations de Wall et Danielsson s'ajoute le problème de la segmentation parallèle d'une même courbe. La figure 5.13 illustre bien ce problème :

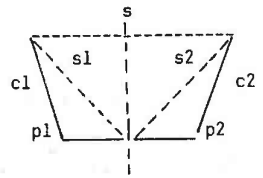


Figure 5.13 : Cas d'approximation aux frontières.

Les courbes c1 et c2 ont été approchées de manière indépendante respectivement par les segments s1 et s2. L'approximation de la courbe totale (c1,c2) peut être faite par le segment s si la mesure algébrique correspondante vérifie le test d'approximation. Le résultat de cette approximation aurait pu être différent si l'on avait commencé par approcher les segments horizontaux adjacents, ce qui aurait pu sauvegarder les points anguleux p1 et p2.

La figure 5.14 montrent d'autres cas où s'ajoutent d'autres problèmes relatifs à la suppression du bruit (a) et à la détermination du point de jonction (b). Dans le cas (a), les segments principaux sont colinéaires et ont des tailles importantes par rapport aux segments élémentaires centraux (conservés provisoirement). On décide dans ce cas de prolonger les deux segments principaux et d'éliminer les autres à

cause de leur petite taille. Les points anguleux provisoirement retenus à l'étape précédente sont supprimés aussi.

Dans le cas (b), les directions principales sont différentes, on décide alors de supprimer le petit segment (considéré comme du bruit) et de prendre comme point de jonction le point d'intersection des deux grands segments. Ce choix conduit tout en permettant de conserver les directions principales à prolonger ou à raccourcir artificiellement les segments.

Un dernier cas est présenté en (c) où les directions principales sont parallèles. Dans ce cas, on décide de les approcher par la ligne oblique joignant leurs extrémités les plus éloignées à condition que cela ne remette pas en cause l'approximation des segments dans les quadrants environnants.

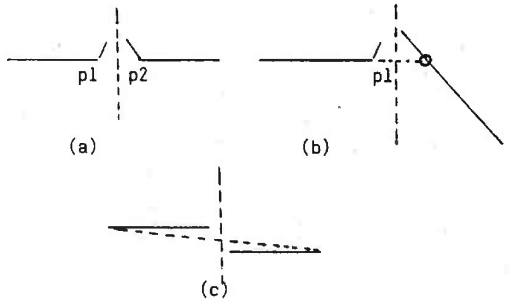


Figure 5.14 : Autres cas limites aux frontières.

Tous ces exemples montrent de manière évidente l'importance que peut avoir le contexte voisin sur la décision d'approximation. De manière pratique, ceci oblige à mémoriser, dans les cas douteux ou incomplets, des solutions intermédiaires au niveau des fils et de laisser le niveau supérieur décider de l'approximation finale.

B) L'algorithme pyramidal

L'algorithme pyramidal fonctionne en trois étapes :

- marquage des frontières,
- orientation locale des contours,
- approximation polygonale des contours.

a) L'étape de marquage

Nous allons d'abord donner quelques définitions utiles :

α) structure pyramidale

L'image nxn, représentée par un père, est divisée en quadrants représentés par ses fils d'ordre n/2. Chaque quadrant devenant père à son tour, représente quatre fils d'ordre n/4, etc... La structure pyramidale considérée est donc une structure 1-père-4fils composée de $\log_2 n$ étages. Elle permet à chaque noeud d'avoir au maximum neuf liaisons directes avec ses voisins parmi le père (à l'étage supérieur), les quatre fils (à l'étage inférieur) et les quatre voisins latéraux (sur le même étage).

β) pyramide englobante

La pyramide englobante de deux noeuds est la plus petite pyramide contenant ces deux noeuds comme le montre le schéma suivant :

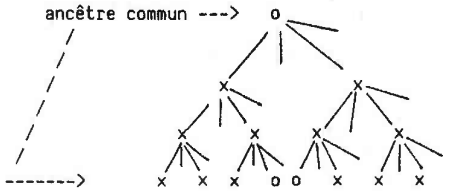
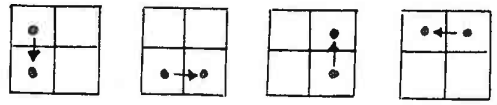


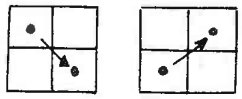
Figure 5.15 : Premier ancêtre commun de deux noeuds représentés dans la base par o.

γ) liaison entre fils

Deux fils sont liés si les quadrants image qu'ils représentent sont traversés par un même contour. On définit six types de liaison possibles comme suit :



1 : 0 → 1 2 : 1 → 2 3 : 2 → 3 4 : 3 → 0

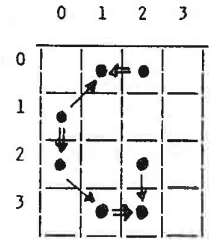


5 : 0 → 2 6 : 1 →3

Figure 5.16 : Sens des liaisons entre quadrants.

Il est important de remarquer que les liaisons sont orientées ; ceci évite d'avoir des cycles lors de la reconstitution du contour.

Une liaison est définie explicitement en se donnant le numéro de la liaison et la position des points de connexion que l'on transmet à leur père commun. De cette manière, on établit une hiérarchie des liaisons en fonction de l'ordre des fils liés. Nous donnons ci-après un exemple :



liaisons entre fils de dimensions 1x1 :

6 (1,0) (0,1)
5 (2,0) (3,1)
1 (2,2) (3,2)

liaisons entre fils de dimensions 2x2 :

4 (0,2) (0,1)
1 (1,0) (2,0)
2 (3,1) (3,2)

Figure 5.17 : Liaisons entre les fils.

point de jonction : un point de jonction est un lieu de croisement entre plusieurs éléments de contour. Nous avons répertorié quatorze configurations de jonction possibles formées de groupements de 4 ou 5 points de contours. Nous en donnons deux exemples ci-après :

Figure 5.18 : Exemples de jonction.

La jonction est signalée dans la liaison. Elle est considérée comme extrémité d'un élément de contour et n'intervient pas dans le codage ; nous ne codons que les éléments de contours entre deux extrémités.

Le marquage consiste à déterminer les liaisons entre les fils de même ordre et de les hiérarchiser en les rangeant dans leur père commun. Ce sont les processeurs de la base qui recherchent les liaisons entre fils.

δ) description du processus général de marquage

Pour une base de longueur n, nous commençons par rechercher les liaisons des fils d'ordre n/2 (fils dont le père est le sommet de la pyramide) et nous transmettons les informations à l'étage supérieur (étage $m = \log_2 n$). Ensuite, nous calculons les liaisons des fils d'ordre n/4, puis nous transmettons les informations à l'étage supérieur (m) qui lui-même a transmis les informations précédentes à l'étage au-dessus (m-1) et ainsi de suite jusqu'à arriver aux fils d'ordre 1.

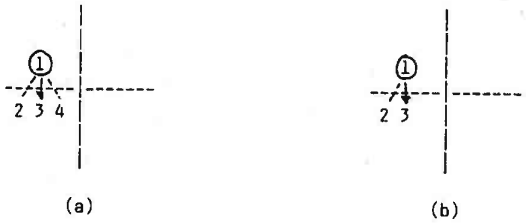
Nous avons ainsi établi un **pipeline** entre étages, ce qui permet à chaque processeur de n'avoir que deux liaisons (l'une avec l'un de ses quatre fils et l'autre avec son père) et de n'effectuer qu'un travail de transmission de données. A la dernière étape, le flot d'information se fige et chaque processeur possède en mémoire les informations concernant les liaisons de ses fils.

Cette hiérarchisation permet la répartition du contour dans les étages sans encombrement des mémoires des processeurs. Chaque noeud connaît les points frontières dans ses fils ce qui permet de retrouver facilement dans l'étape de chaînage les points de raccordement de contours.

ε) Calcul effectif des liaisons

Pour le calcul des liaisons, le problème est de savoir quels sont les processeurs qui vont travailler à chaque étape et comment se fait la prise en compte des liaisons entre les processeurs voisins. Pour le premier problème, chaque processeur contient pour chaque voisin, le niveau du père qu'il a en commun avec lui. Ces indications sont utilisées par le contrôleur de l'étage de base qui actionne à chaque étape les processeurs de l'étape.

Pour le second, chaque processeur désigne le ou les voisins avec qui il peut être lié. Un voisin n'est candidat que s'il contient un point de contour, s'il se trouve sur un sens de liaison permis et si la liaison à laquelle il conduit ne crée pas de cycle dans le chaînage. Nous étudions ci-après les liaisons possibles pour le processeur 1 dans des configurations différentes. Les numéros donnés 2,3,4 ... sont les numéros des processeurs voisins de 1 contenant des points de contour :

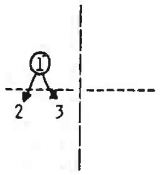


(a)

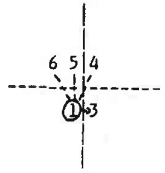
(b)

(a) cas d'une jonction : toutes les liaisons 1→2, 1→3 et 1→4 sont possibles (de type 1), seule la liaison 1→3 sera retenue.

(b) cas d'un coin : seule la liaison verticale 1→3 sera retenue car on privilégie les liaisons orthogonales.



(c)



(d)

(c) cas d'un pic : les deux liaisons diagonales 1→2 et 1→3 seront retenues.

(d) cas général : 2, 3, 4, 5 et 6 sont les processeurs voisins de 1, ayant le même père commun pour la partition dessinée. Parmi les liaisons possibles : 1→2, 1→3 et 1→4, seule la liaison 1→3 (de type 2) sera retenue.

Nous pouvons remarquer à travers ces exemples, qu'un processeur actif ne peut donc consulter qu'au maximum trois voisins à cause du sens des liaisons et ne peut générer que deux liaisons au maximum.

ζ) Algorithme de marquage

L'algorithme de marquage peut s'écrire dans une syntaxe proche de celle du langage LPSI comme suit :

```
Type  (* déclaration de la structure pyramide *)

tpr : pyramide[dimensions, nombre_etages, type_info];

...

Function marquage (pr:tpr;w:window;c: position) : tpr;

(* w précise à chaque appel les dimensions du quadrant
à traiter, c donne son coin supérieur gauche *)

VAR s : binary; (* s est ici un filtre *)
    p : position;

(* localisation des bandes frontières susceptibles
de contenir des liaisons. s est le filtre qui
détermine l'accès à ces bandes *)

begin

s := ((c$ = w.x/2 or c$ = w.x/2+1) or
     (l$ = w.y/2 or l$ = w.y/2+1));

if s ≠ 0 then (* test possible sur les binaires en LPSI *)
&[(* si le quadrant en cours n'est pas réduit à un point *)

begin

   for_all p in pr.base<s(c)> do

   (* traitement de chaque point des bandes frontières *)
   (a* 1 : test de jonction *)

     if jonction(p) then  pr[p].jonction = 1;

   (* 2 : recherche d'une liaison avec un voisin *)

     for_all pl in voisinage(p) do
       if jonction(pl) then  pr[pl].jonction = 1;
       if liaison(p,pl) then
       ranger(pr,pere_commun(p,p'),p,pl,type_lien(p,pl));
     end-for;
end-for;
(* rappel de marquage sur des quadrants plus petits *)
```

```
    w := [w.haut/2 , w.larg/2]
    marquage(pr,w,<c.y,c.x>);
    marquage(pr,w,<c.y,c.x+w.larg/2>);
    marquage(pr,w,<c.y+haut/2,c.x>);
    marquage(pr,w,<c.y+w.haut/2,c.x+w.larg/2>);
  end;
  return pr;
  end marquage;
```

b) L'étape d'orientation

L'orientation des contours se fait de manière locale. Chaque père d'un étage donné va regarder ses quatre fils et essayer de reconstituer le contour fragmenté. La reconstruction consiste à déterminer l'origine et l'extrémité du contour, ce qui va imposer une orientation identique pour tous les fragments de contours. A l'issue de cette étape, chaque père contient, pour chaque contour passant par ses fils, ses extrémités et la liste des numéros des fils de passage accompagnés chacun d'un signe de réorientation éventuelle. La figure 5.18 donne un exemple d'orientation de contours :

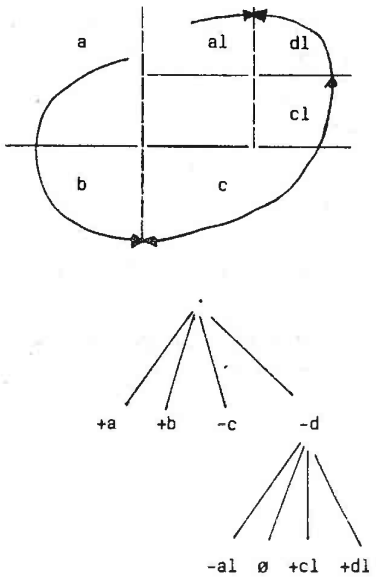


Figure 5.18 : Orientation de contours.

Au niveau des pères, la condition de chaînage est la même : les contours cf1 et cf2 provenant des fils f1 et f2 sont chaînés s'ils vérifient les conditions suivantes :

- le noeud père contient une liaison entre f1 et f2,
- cf1 et cf2 contiennent des extrémités voisines signalées dans ce noeud.

La procédure de chaînage et d'orientation se sert du sens de la liaison pour orienter le contour résultat. La liaison indique l'ordre dans lequel il faut prendre les contours pour les concaténer. La règle d'orientation est la suivante : deux contours cf1 et cf2 donnés dans cet ordre par la liaison, seront orientés dans le même sens si l'extrémité de cf1 est voisine de l'origine de cf2. Cette règle simple impose d'avoir dans chaque liaison deux points frontières où l'un est une extrémité et l'autre est une origine. L'ordre origine-extrémité ou extrémité-origine est fixé soit par le numéro de la liaison au démarrage du chaînage soit par le sens du contour en cours de chaînage. Dans le cas où l'ordre des points frontières est contradictoire avec le sens du contour, le ou les contours fils sont inversés.

c) L'étape d'approximation polygonale

Au cours de cette étape, chaque noeud père dispose des contours fils uniformément orientés. Dans le cas le plus général, pour chaque frontière, l'extrémité de chaque contour sera donnée par :

- l'avant dernier segment caractérisé par :
  + son origine p1,
  + sa longueur l1,
  + sa direction d1 (codée de 1 à 8),
  + sa surface algébrique $\Phi 1$,
- le dernier point anguleux p0,
- le dernier segment accompagné de :
  + son extrémité p2,
  + sa longueur l2,
  + sa direction d2 (codée de 1 à 8),
  + sa surface algébrique $\Phi 2$,
- la surface algébrique Φ de l'ensemble.

Un automate d'approximation polygonale est écrit, permettant en fonction de ces mesures :

- de décider où faire les jonctions,
- de supprimer le bruit,
- de lisser le courbe dans les limites imposées par les segments les plus externes afin d'éviter les effets de chaîne,
- ou d'attendre de voir les contextes voisins pour décider.

L'automate utilise des seuils de comparaison qui s'adaptent au contour en fonction de sa longueur et de l'étage dans lequel il se trouve. Généralement, plus on monte dans les étages, plus le contour devient important et les détails disparaissent, et plus le lissage devient

précis.

d) Complexité des algorithmes

α) l'algorithme de marquage

Le nombre de liaisons à ranger dans chaque noeud ne dépasse jamais le nombre de points frontières dans ses fils. Comme le montre la figure 5.19a, la répartition maximale des points autour des quatre frontières dans une partition d'ordre n/2 donne 4 x n/2 = 2n points frontières. Or, comme les contours sont d'épaisseur 1, on peut considérer qu'on a, au maximum, une frontière sur deux, soit n liaisons (cf. fig. 5.19b). Ce nombre varie bien sûr en fonction de l'étage.

Le temps d'éxécution se mesure en fonction du nombre d'activités des processeurs et de leur temps de traitement. A l'étage de base, l'activité du processeur se résume en la recherche d'une liaison d'un type donné et sa transmission au père. Cette activité se repète m fois. Dans les étages supérieurs, chaque processeur transmet ses données à son père. Sans considérer les problèmes de synchronisation, toutes ces activités se déroulent en parallèle. On peut estimer que le temps maximum nécessaire pour le marquage de la pyramide est égal au temps de recherche des m types de liaison et de leur remontée au père, soit pour une liaison, pour chacun des voisins visités par le PE :

- test d'existence d'un point de contour,
- recherche du sens de liaison,
- test de jonction. Ceci est à multiplier par 2 pour les deux points en liaison.

Malgré le nombre important de ces tests locaux, l'algorithme de marquage reste en O(m), c'est-à-dire en $\mathbf{O(\log_2 n)}$.

(a) répartition maximale des points frontières.

(b) répartition maximale possible des points frontières.

Figure 5.19 : Répartition maximale des points frontières.

β) L'algorithme d'orientation

On range dans chaque noeud et pour chaque chaîne, la liste des numéros des fils signés. Le nombre maximum de fils que l'on peut considérer est égal au nombre de liaisons entre les quatre fils, soit n liaisons (où n est l'ordre des fils).

La tâche de chaque processeur est de traiter chaque liaison rangée dans sa mémoire en concaténant et orientant les contours fils. Ce traitement se répétant n fois (autant de fois qu'il y a de liaison) sur des contours remontant des chemins de longueur maximale égale à $\log_2 n$, se réalise donc en $\mathbf{O(n\log_2 n)}$.

Nous n'avons pas étudié la complexité de l'algorithme d'approximation car il est en cours de test et d'optimisation.

C) Discussion

Nous avons en réalité décrit ici deux algorithmes complémentaires. L'un est relatif à la représentation hiérarchique du contour et à son orientation locale et uniforme. L'autre est relatif à la prise en compte de contextes fils pour réaliser l'approximation polygonale parallèle. La représentation hiérarchique des contours peut servir à d'autres applications comme la segmentation angulaire en primitives structurelles. Cet exemple nous a permis d'étudier les différents types de parallélisme de la machine et d'allier les algorithmes aux schémas d'exécution de la pyramide. Les algorithmes décrivent bien les transmissions de données vers les pères et distinguent correctement les traitements relatifs à chaque noeud. Ils expriment aussi de manière rigoureuse la synchronisation au niveau de chaque étage et les problèmes de résolution aux frontières qui ne sont pas toujours évidentes dans le cas d'une décomposition récursive et hiérarchique.

5. CONCLUSION

Nous avons montré dans ce chapitre qu'il était possible d'exploiter le parallélisme dans les programmes en tenant compte de la réalité des architectures et sans gros moyens théoriques et techniques comme c'est souvent le cas en programmation classique.

Les solutions apportéees pour extraire le parallélisme implicite dans les programmes montrent qu'en connaissant le fonctionnement des processeurs en place et leurs paramètres, il est toujours possible de trouver les opérations qui peuvent être exécutées par les processeurs même si ces dernières ne sont pas écrites globalement.

Nous avons montré que ces solutions peuvent conserver de l'intérêt dans des langages très concis comme LPSI où l'on ne cherche à exprimer que du parallélisme de type SIMD. Nous avons été conduits à préciser les mécanismes de mise en évidence du parallélisme qui paraissent les mieux adaptés à cette approche.

Deux cas d'architecture ont été étudiés. Pour les architectures multiprocesseurs, cette approche ne semble pas poser trop de problème car les processeurs sont souvent indépendants et ont un comportement facilement modélisable. Ce n'est pas le cas pour les machines pyramidales où le traitement est une coopération entre plusieurs processeurs pouvant décrire différents mouvements de données.

L'exemple du paragraphe 3.1.5 a été choisi pour mettre en évidence les principes discutés. En effet, la mobilisation à un moment donné d'un processeur pour exécuter un traitement est déclenchée par l'arrivage des données, ce qui explique l'organisation des traitements adoptée par étage et par l'ensemble des étages pour respecter le synchronisme qui gère les mouvements de données entre les processeurs. Nous avons essayé de montrer à travers cet exemple que la programmation pyramidale est beaucoup plus qu'un exercice de style ! Sa maitrise reste essentielle pour l'accélération des traitements d'images et exige une poursuite de l'étude commencée ici.

CONCLUSION

Après une analyse générale des problèmes posés par le traitement d'images et des besoins en matériel et logiciel pour disposer d'un ensemble d'outils répondant aux exigences des spécialistes dans ce domaine, nous avons proposé un modèle de système de traitement d'images appelé SAPIN.

Le système SAPIN fournit des outils logiciels à deux niveaux :

- pour l'utilisateur, un langage interactif permettant l'expérimentation rapide de nouvelles idées,

- pour le programmeur, des outils adaptés à la construction de programmes et leur implantation sur des architectures spécialisées.

L'environnement logiciel ainsi développé fournit trois niveaux d'aide :

◆ (1) une interface utilisateur facile qui est capable de fournir une aide constante à toutes les étapes du travail. Destinée à un non-informaticien, elle fournit plusieurs modes de fonctionnement, une assistance à l'utilisateur dans ses premières interrogations et surtout le moyen d'apprendre à se servir du système mais aussi de construire son environnement personnalisé. Cette interface a été écrite pour répondre aux exigences des programmeurs en traitement d'images, mais nous nous sommes vite rendu compte que mis à part les données image, cette interface peut être utilisée pour d'autres types d'applications. Le générateur de menus dans son ensemble reste général. Les variables, quels que soient leur type, ne sont pour lui que des ponts de communications entre les commandes en place.

◆ (2) le programmeur a besoin de tester rapidement ses algorithmes; aussi il a besoin d'outils capables de manipuler les objets image indépendamment des contraintes physiques liées au système, telle la capacité de la mémoire d'image ou le nombre de processeurs de la machine. Une telle abstraction permet d'élever le niveau du programme et permet d'envisager plus facilement des solutions de portabilité sur d'autres machines.

Le système SAPIN offre un ensemble d'outils de programmation de haut niveau introduits sous la forme d'un langage de programmation spécialisé appelé LPSI. LPSI a été étudié dans le but d'offrir les possibilités de programmation suivantes :

- des types de donnée adaptés aux images. Chaque type est donné par un ensemble de définitions et d'opérations caractéristiques. Pour décrire ces types, nous avons adopté des définitions hiérarchiques faisant appel à des constructeurs de types prédéfinis. Cette unification par les constructeurs a permis de définir proprement les types et d'écarter certaines définitions redondantes. Leur rôle est en tout cas important car elles permettent de mettre en évidence des concepts unificateurs tels que celui d'accès à une image qui recouvre à la fois celui d'image vue à travers une fenêtre, celui de région et celui d'ensemble

de points caractérisés par une propriété.

- L'étude des opérations a permis de définir proprement les fonctions d'accès et de distinguer facilement les traitements séquentiels des traitements parallèles. LPSI contient des opérateurs globaux et des structures de contrôle exprimant le parallélisme SIMD dans les opérations matricielles ponctuelles et locales.

◆ (3) Pour qu'un programme composé d'opérations indépendantes soit exécuté efficacement, il faut être capable de mettre en évidence ce parallélisme et le gérer, ce qui n'est pas toujours une tâche facile pour l'utilisateur.

Nous avons montré dans le chapitre 5 qu'il existe des mécanismes classiques d'expression du parallélisme, mais que ces mécanismes deviennent vite lourds pour programmer des machines massivement parallèlement. Les solutions que nous avons adoptées sont plutôt orientées vers la mise en évidence du parallélisme intrinsèque. Ceci est fait soit directement par l'écriture des opérations parallèles dans une syntaxe concise facilitant leur détection soit indirectement par analyse partielle du fonctionnement des sous-programmes et leurs traduction dans des standards de processus.

Enfin, nous ne pouvons pas nous contenter d'affirmer que les outils réalisés sont efficaces ; encore faut-il le prouver, ce qui n'a pas été fait. Nous souhaitons aussi convaincre les lecteurs de la démarche à suivre et de l'intérêt de poursuivre le travail dans cette voie.

Nous avons pu au cours de ce travail proposer une démarche à suivre depuis la spécification des types image jusqu'à la parallélisation d'algorithmes basiquement séquentiels. Les résultats sont intéressants, car ils nous ont permis de découvrir les propriétés des objets utilisés, ce qui nous a conduits à trouver des solutions logicielles plus adaptées que celles que nous utilisions par l'intermédiaire de Fortran : un langage adapté au traitement d'images avec tous les mécanismes nécessaires pour manipuler une image et pour agir sur son mode d'exécution et rangement.

Nous considérons ce travail comme un pas vers une meilleure prise en compte de la matière que nous utilisons tant logicielle que matérielle et surtout de la coopération des deux pour réaliser des programmes performants. Les solutions que nous avons apportées tiennent aussi compte de l'utilisateur quel que soit son niveau informatique.

BIBLIOGRAPHIE

[ABE-69] **N. E. ABEL et al**, TRANQUIL : a language for an array processing computer, Proc. S.J.C.C., pp 57-68, 1969.

[ALL-71] **F.E. ALLEN and J. COCKE**, A catalogue of optimizing transformations, Design and optimization of compilers, R. Rustin, ed., Courant computer science symposium 5, pp 29-33, March 1971.

[BAL-81] **D. H. BALLARD**, Strip trees : a hierarchical representation for curves, Comm. of the ACM, vol. 24, n. 5, 1981.

[BAL-82] **D.H . BALLARD and C. M. BROWN**, Computer vision, Prentice-Hall, Inc., Englewood cliffs, New jersey, 1982.

[BAS-82] **J. L. BASILLE**, Architectures parallèles en traitement d'images : le projet SY.MP.A.T.I, Journées AFCET-ETCA, Arcueil, 2-3 Déc. 1982.

[BAS-85] **J. L. BASILLE**, Structures parallèles et traitement d'images, thèse d'état, Univ. Paul Sabatier, Toulouse, Dec. 1985.

[BAT-79] **B. G. BATCHELOR**, Using concavity trees for shape description, Computers and digital techniques, vol. 2, n. 4, pp 157-168, 1979.

[BAT-80] **K. E. BATCHER**, Design of Massively Parallel Processor, IEEE Trans. on Comp., vol. c-29, no. 9, pp 836-840, 1980.

[BEL-85] **A. BELAID**, Un lanagage de traitement d'images fondé sur des concepts de types abstraits, T.S.I, vol. 4, nO. 3, 1985.

[BEL-87] **A. BELAID and Z. BOUFRICHE**, Hierarchical specification of image data types and its use in a high level language, IV International Conference on Image Analysis and Processing, Cefalu' SICILY, ITALY, Sept 23-25, 1987.

[BER-66] **A. J. BERNSTEIN**, Analysis of programs for parallel processing, IEEE Trans. on Computers, vol. 15, no. 5, pp 757-763, Oct. 1966.

[BOU-87] **Z. BOUFRICHE**, Thèse à paraître, INPL, Nancy, 1987.

[BOR-84] **G. BORGEFORS**, Distance transformations in arbitray dimensions, CVGIP, 27, pp. 321-345, 1986.

[BRU-84] **P. J. BRUMFITT**, Environments for image processing algorithm development, Image and vision computing, vol. 2, no. 4, pp. 198-203, Nov. 1984.

[CAM-78] **R. H. CAMBELL and A. N. HABERMAN**, The specification of process synchronisation by path expression, Traitement parallèle, école de l'INRIA, vol. 1, Mai-Juin 1978.

[CAN-85] **V. CANTONI, M. FERRETTI, S. LEVIALDI, M. MALBORTI**, A pyramid project using integrated technology for parallel image processing, ed. by S. Levialdi, Academic Press, N. Y. pp. 121-132, 1985.

[CAS-83] **D. CASSIGNAC**, SAPIN-NI : un système hautement interactif pour le traitement d'images numérisées, Rapport de DEA, Université de Nancyl, Sept. 1983.

[CAS-77] **S. CASTAN et G. STAMON**, Eléments de traitemnt d'images, Ecole d'été de l'AFCET, Montréal, Juil. 1977.

[CAS-82] **S. CASTAN**, Image processing architecture design, in conf. on Image Analysis and Processing, Selva di Fasano, Italy, Nov. 1982.

[CAS-85] **S. CASTAN**, Architectures adaptées au traitement d'images, T. S. I., vol. 4, no. 5, pp. 431-445, 1985.

[CAS-86] **S. CASTAN**, Architectural comparisons, Note ARW on: pyramidal systems for image processing and computer vision, Maratea, Italy, May 5-9, 1986.

[CHA-84] **J. M. CHASSERY**, Présentation discrète, interprétation numérique et description des images : des concepts à l'application, Thèse d'état, Grenoble, 1984.

[CHA-85] **J. M. CHASSERY**, Description du logiciel IPS, annexe technique TIM3, INPG, Grenoble, Nov. 1985.

[CHI-74] **Y. P. CHIEN and K. S. FU**, A decision function method for boundary detection, CGIP 3, 2, pp 125-140, June 1974.

[CHI-78] **J. P. CHIEZE et O. FAUGERAS**, Un système interactif de traitement d'images, Congrès AFCET-INRIA RFIA, Tome 2, Fev. 1978.

[CON-63] **M. E. CONWAY**, A multiprocessor system design, AFIPS, FJCC, G3, pp. 139-146, 1963.

[COR-81] **CORNAFIAN**, Systèmes informatiques répartis - concepts et techniques, ed. par Dunod, 1981.

[DAH-78] **O. J. DAHL and K. NYGAARD**, SIMULA 67 common base language, publication no. 5-2 Norvegian, Computer Science, May 1978.

[DAN-81] **P. E. DANIELSSON and S. LEVIALDI**, Computer architectures for pictorial information system, IEEE computer, pp. 53-67, Nov. 1981.

[DEM-78] **E. DE MASSAS**, Etude, réalisation et utilisation d'outils logiciels adaptés à l'écriture parallèle d'algorithmes, thèse de 3ème cycle, Grenoble, Sept. 1978.

[DER-79] **J. C. DERNIAME et J. P. FINANCE**, Types abstraits de données: spécification, utilisation et réalisation; cours de l'école d'été de L'AFCET, Monastir, Juil. 1979.

[DEU-80] **L. P. DEUTSCH**, Bytelisp and its Alto implementation. Conference Record of the 1980 LISP Conference. Stanford University, pp. 231-242, Aug. 1980.

[DIG-86] **V. DI GESU'**, An high level language for pyramidal architecture, Maratea, Italy, May 5-9 1986.

[DIJ-68] **E. W. DIJKSTRA**, Co-operating sequential processes, Programming languages, F. Genuys, ed., Academic Press, New york, 1968.

[DIJ-76] **E. W. DIJKSTRA**, A discipline of programming, Prentice-Hall, Englewood Cliffs, NJ, 1976.

[DOR-86] **L. DORST**, Pseudo-euclidean skeletons, Proc. of 8th IAPR, pp 286-288, Paris, Oct. 27-31, 1986.

[DUB-86] **E. DUBOIS, N. LEVY et J. SOUQUIERES**, SACSO : Méthodes et outils de construction de spécifications de systèmes, actes CGL3, Versailles 1986.

[DUD-73] **R. O. DUDA and P. E. HART**, Pattern recognition and scene analysis, New york wiley, 1973.

[DUF-81] **M. J. B. DUFF and S. LEVIALDI**, Languages and Architectures for image processing, ed. by the authors, Academic Press, 1981.

[DUF-76] **M. J. B. DUFF**, CLIP IV: a large scale integrated cicuit array parallel processor, 3rd IJCPR, Coronado, CA 1976.

[DYE-81] **C. R. DYER**, A VLSI Pyramid machine for hierarchical image processing, Proc. PRIP, pp 381-386, 1981.

[DYE-82] **C. R. DYER**, Pyramid algorithms and machines, in .BI [PRE-82].

[EVE-73] **A. J. EVENSEN and al**, Introduction to the architecture of a 288 element PEPE, Proc. of Int. Conf. on parallel processing, Sagamore, pp. 162-169, 1973.

[FAU-83] **O. D. FAUGERAS and J. PONCE**, Prism trees : a hierarchical representation for 3-D objects, Proc. IJCAI-83, Karlsrhue, 1983.

[FIN-78] **J. P. FINANCE**, De la spécification abstraite d'une donnée à sa représentation en mémoire : les états successifs d'une information, congrès AFCET, 1978.

[FIN-79] **J. P. FINANCE**, Etude de la construction de programmes : méthodes et langages de spécification et résolution de problèmes, thèse d'état, Nancyl, 1979.

[FIS-73] **M. A. FISCHLER and R. A. ELSCHLAGER**, The representation and matching of pictorial structures, IEEE Trans. computers C 21, pp 67-92, 1973.

[FLY-72] **M. J. FLYNN**, Some computer organizations and their effectiveness. IEEE Transactions in computers, vol. c-21, pp 948-960, Sept. 1972.

[GAU-76] **M. C. GAUDEL**, Structures de données et traduction des langages de programmation, Rapport IRIA-Laboria nO 193, 1976.

[GOL-71] **P. C. GOLDBERG**, A comparison of certain optimization techniques, in Design and optimization of compilers, R. Rustin ed., Courant computer science symposium 5, Mar. 29-30, 1971.

[GON-77] **R. C. GONZALEZ and P. WINTZ**, Digital image processing, Addison wesley, 1977.

[GRA-82] **G. H. GRANLUND**, Image Enhancement, Vision par ordinateur, support de cours, tome 2, INRIA Rocquencourt, Juin 1982.

[GRA-83] **G. H. GRANLUND and J. ARDIDSSON**, The GOP image computer, Fundamentals in computer vision, O. Faugeras ed., Cambridge University Press, 1983.

[GUI-84] **M. GUILLEMONT, H. ZIMMERMAN, G. MORISSET et J. S. BANINO**, CHORUS : Une architecture pour les systèmes répartis, INRIA, rapport de recherche nO 274, Mars 1984.

[GUT-77] **J. V. GUTTAG**, Spécification algébrique de types abstraits, Bulletin de l'AFCET, Groplan no. 2, 1977.

[HAN-74] **A. R. HANSON and E. M. RISEMAN**, Preprocessing cones : a computational structure for scene analysis, COINS Tech. Report nO 746-7, University of Massachusetts, Amherst, MA, USA, Sept. 1974.

[HAR-78] **R. M. HARALICK and G. MINDEN**, KANDIDATS: an interactive image processing system, CGIP 8, pp. 1-15, 1978.

[HOA-69] **C. A. R. HOARE**, An axiomatic basis for computer programming, Com. ACM 12 (10), pp. 576-581, Oct. 1969.

[HOA-72] **C. A. R. HOARE**, Towards a theory of parallel programming, Operating system techniques, Academic Press, London, 1972.

[HOR-84] **H. HORGEN et B. ZAVIDOVIQUE**, Correspondance entre algorithmes et architectures de machines, Traitement d'images, Journées de synthèse INRIA, 23-24 Jan. 1984.

[HOR-76] **S. L. HOROWITZ and T. PAVLIDIS**, Picture Segmentation by a directed split-and-merge procedure, Proc. 2nd IJCPR, pp 424-433, Aug. 1974.

[HOR-76] **S. L. HOROWITZ and T. PAVLIDIS**, Picture segmentation by a tree traversal algorithm, J. Assoc. Comput. Mach. 23, pp 368-388, 1976.

[HUE-73] **M. J. HUECKEL**, A local visual operator which recognizes edges and lines J. ACM, 20, 1973.

[HUN-79] **G. M. HUNTER and K. STEIGLITZ**, Operation on images using quatrees, IEEE Trans. on PAMI, Vol. 1, pp 145-153, 1979.

[ICH-76] **J. D. ICHBIAH**, The systems implementation language LIS, Manuel de référence CII 4549, EIIEN, Jan. 1976.

[ICH-79] **J. D. ICHBIAH et al.**, Preliminary Ada Reference Manual, ACM Sigplan Notices 14, 6 -A, 1979.

[JON-70] **E. G. JOHNSTON**, The PAX Processing system, in Picture processing and psychopictories, ed. by B. S. Lipkin and A. Rosenfeld, Academic Press, 1970.

[JON-79] **A. K. JONES and K. SCHWANS**, Task forces : distributed software for solving problems of substantial size, Proceedings of the 4th International Conference on Software Engineering, IEEE, 1979.

[KEL-80] **B. KELLER**, Contribution à la réalisation d'un système de traitement d'images piloté par microprocesseur, thèse de 3ème cycle, Nancyl, Juil. 1980.

[KEL-71] **M. D. KELLY**, Edge detection in pictures by computer using planing, Machine intelligence 6, pp 379-409, 1971.

[KRA-85] **S. KRAKOWIAK**, Principes des systèmes d'exploitation des ordinateurs, Dunod, 1985.

[KRU-82] **B. KRUSE, P. E. DANIELSSON and B. G. GUDMUNDSSON**, From PICAP1 to PICAP2, special computer architectures for pattern recognition, ed. by K. S. Fu and T. Ichikawa, CRC Press, London 1982.

[KUN-82] **H. T. KUNG**, Why systolic architectures?, computer, vol. 15-1, pp. 37-46, Jan. 1982.

[LAM-75] **L. LAMPORT**, On programming parallel computers, SIGPLAN notices, vol. 10, no. 3, March 1975.

[LAN-76] **R. G. LANGE**, High-level language for associative and parallel computation with STARAN, Proc. of the Inter. Conf. on parallel processing, 1976.

[LAW-75] **D. H. LAWIE, T. LAYMAN, D. BAER and J. M. RANDAL**, GLYPNIR: a programming language for ILLIAC IV, com. ACM, vol. 18, 3, 1975.

[LEN-86] **J. LENFANT**, Quelle architecture pour les gigaflops, Notes de cours, Ecole INRIA, Calcul parallèle à usage scientifique, St. Cyprien, 14-18 Octobre 1985.

[LEV-81] **S. LEVIALDI A. MAGGIOLO-SCHETTINI, N. NAPOLI, G. TORTORA and,** U. UCCELLA" On the the design and implementation of PIXAL, a language for image processing, in [DUF-81].

[LEV-86] **S. LEVIALDI**, Languages for image pocessing, In computer vision, Lecture notes n. 1, Rocquencourt June 1st-11th, pp 13-32, 1982.

[LEV-86] **S. LEVIALDI**, Programming image processing machines, Note ARW on: pyramidal systems for image processing and computer vision, Maratea, Italy, May 5-9 1986.

[LEV-84] **N. LEVY**, Outils d'aide à la construction et transformation de types abstraits algébriques, thèse de 3ème cycle, Nancyl, Juil. 1984.

[LEV-87] **N. LEVY, A. PIGANIOL and J. SOUQUIERES**, Specifying with SACSO, 4th International workshop on Software Specification and Design, April 3-4, 1987, Monterey, California.

[LIC-85] **A. LICHNEWSKY et F. THOMASSET**, Techniques de base sur l'exploitation automatique du parallélisme dans les programmes, dans Calcul parallèle à usage scientifique, notes de cours, St. Cyprien, 14-18 Oct. 1985.

[LIE-77] **Y. E. LIEN**, A data base system to support evaluation university of Kansas, Final technical report, 1977.

[LIS-75] **B. LISKOV and S. ZILLES**, Specification techniques for data abstractions, IEEE Trans. Softw. Eng. SE-1, 1, pp 7-19, March 1975.

[LOU-80] **J. R. LOUVION**, Détection du contour de lèvres et extraction de paramètres constitutifs : application à l'analyse automatique de labio-films, thèse de 3ème cycle, Université de Rennes 1, Déc. 1980.

[LOW-82] **G. E. LOWITZ et G. STAMON**, Douzième école internationale AFCET d'informatique, Images, Principes, Traitements et Applications, Facultés universitaires de Namur, Belgique, Juillet 1982.

[MAG-81] **A. MAGGIOLO-SCHETTINI**, Comparing some high-level languages for image processing, in [DUF-81].

[MAL-82] **W. R. MALLGREN**, Formal specification of graphic data types, ACM Trans. on Programming languages and systems, vol. 4, n. 4, pp 687-710, Oct. 1982.

[MAR-80] **D. MARR and E. HILDRETH**, Theory of edge detection, MIT AI-Memo 518, 1980.

[MAR-72] **A. MARTELLI**, Edge detection using heuristic search methods, CGIP1, 2, pp 169-182, Aug. 1972.

[MAR-76] **A. MARTELLI**, An application of heuristic search methods to edge and contour detection, Comm. ACM 19, 2, pp 73-83, Feb. 1976.

[MAZ-78] **G. MAZARE**, Structures multi-microprocesseurs, problèmes de parallélisme, définition et évaluation d'un système particulier, thèse d'état, Grenoble, Juin 1978.

[MER-83] **A. MERIGOT**, Une architecture pyramidale d'un multi-processeur cellulaire pour le traitement d'images. Thèse de 3ème cycle, Paris-sud, Oct. 1983.

[MIL-73] **R. E. MILLSTEIN**, Control structures in ILLIAC IV Fortran, CACM, vol. 16, no. 10, pp. 621-627, Oct. 1973.

[MIN-79] **R. MINOT**, ATM, un système de fabrication de programmes basé sur les concepts de modularité et de type abstrait, thèse de 3ème cycle, Nancyl, Mar. 1979.

[MOH-86] **R. MOHR et A. BELAID**, Software tools for image processing, Second mage symposium, Cesta, Nice, april 1980.

[MOU-85] **N. MOUADDIB**, SAPIN : système interactif de génération de menus, Rapport de DEA, Nancyl, 1985.

[ONO-81] **M. ONOE, K. PRESTON Jr. and A. ROSENFELD**, Real-time/parallel computing, ed. by the authors, Plenum Press, 1981.

[PAP-73] **S. PAPERT**, Uses of technolgy to enhance education, Technical report 298, AI Lab, MIT, 1973.

[PAU-78] **PAU**, ACTUS: A Language for array and vectorprocessors. NASA/AMES, Research Center, Aug. 1978.

[PAV-82] **T. PAVLIDIS**, Algorithms for graphics and image processing, Springer verlag, 1982.

[PRA-78] **W. K. PRATT**, Digital image processing, a wiley-interscience publication, 1978.

[PRE-71] **K. PRESTON Jr.**, Feature extraction by Golay Hexagonal Pattern Transforms, IEEE Trans. on Computers, C-20 : 1007, 1971.

[PRE-72] **K. PRESTON Jr.**, Use of the Golay Logic Processor in pattern recogintion studies using hexagonal neighborhood logic, in computers and automata, pp. 609-624, Polytechnic Press, New york, 1972.

[PRE-80] **K. PRESTON Jr.**, Image manipulative languages - A preliminary survey, in Pattern Recognition in Practice, ed. by E. S. Gelsema and L. N. Kanal, North Holland publishing Compagny, 1980.

[PRE-81] **K. PRESTON Jr.**, Languages for parallel processing for images, in [ONO-81].

[PRE-82] **K. PRESTON Jr. and L. UHR**, Multicomputers and Image processing, Academic Press, 1982.

[PRE-83] **K. PRESTON Jr.**, Progress in image processing languages, in Computing structures for image processing, ed. by M.J.B. Duff, Academic Press, pp. 195-211, 1983.

[RAD-81] **T. RADHAKRISHNAN, R. BARRERA, A. GUZMAN and A. JINICH**, Design of a high-level language L for image processing, in [DUF-81].

[RAM-69] **C. V. RAMAMOORTHY and M. J. GONZALEZ**, A survey of techniques for recognizing parallel processable streams in computer programs, Programs AFIPS, FJCC, pp 1-14, 1969.

[REE-84] **A. P. REEVES**, Parallel Pascal: an extended Pascal for parallel computers, Journal of parallel and distributed computing, vol. 1, no. 1, pp. 64-80, 1984.

[REN-80] **C. RENVOISE, C. ROUQUIE, J. C. COTTET et P. FEAUTRIER,** Détection automatique d'opérations dans les programmes, note de programmation, DCP/DS/LA/TP80-01, CII Honeywell BULL, Nov. 1980.

[RIS-70] **W. J. RISHEL**, Incremental Compilers, Datamation, pp. 129-136, Jan. 1970.

[ROS-62] **A. ROSENFELD**, Picture processing by computer, Academic Press, New york, 1962.

[ROS-70] **A. ROSENFELD**, Connectivity in digital pictures, Journal of the association for computing machinery, vol. 17, no 1, pp 146-160, January 1970.

[ROS-72] **A. ROSENFELD, M. THURSTON and Y. LEE**, Edge and curve detection; further experiments, IEEE Computers, C-21, 1972.

[ROS-76] **A. ROSENFELD and A. C. KAK**, Digital picture processing, Academic press, 1976.

[ROS-79] **A. ROSENFELD, M. THURSTON and Y. LEE**, Edge and curve detection: further experiments, IEEE computers, C-21, 1972

[ROS-80] **A. ROSENFELD**, Quadtrees and pyramids for pattern recognition and image processing. Proc. of ICPR, pp 802-811, 1980.

[ROS-83] **A. ROSENFELD**, Pyramids : Multiresolution image analysis, Proc. third Scaninavian Conf. on Image Analysis, Copenhagen, Denmark, pp 23-28, July 1983.

[ROS-84] **B. ROSSI and al**, An evaluation of SPIDER, a portable subroutine package for image analysis, and an outline of its use in the interactive ILLIAD environment, laboratory of image analysis, Uppsala university, Sweden, 1984.

[ROU-73] **G. ROUCAIROL et A. WIDORY**, Programmes séquentiels et parallélisme, R.A.I.R.O, B-2, pp 5-22, Juin 1973.

[SAM-80a] **H. SAMET**, Region representation : quadtrees from binary arrays, Comp. Gr. Im. Proc. 13, 1980.

[SAM-80b] **H. SAMET**, Region representation : quadtrees form boundary codes, CACM 23, 1980.

[SAM-82] **H. SAMET**, Neighbor finding for images represented by quadtrees, CGIP, Vol. 18, pp 37-57, 1982.

[SER-84] **V. SERFATY-DUTRON et L. BOL-BITTOUM**, Conception et normalisation d'un logiciel de traitement d'images, Contrat DRET no 83-R-1093, ETCA, Octobre 1984.

[SER-85] **V. SERFATY-DUTRON and B. ZAVIDOVIQUE**, Programming facilities in image processing, COMPINT 85, Montréal, Sep. 1985.

[SER-82] **J. SERRA**, Image analysis and mathematical morphology, Academic Press, 1982.

[SER-86] **J. SERRA**, Introduction to mathematical morphology, CVGIP 35, pp 283-305, 1986.

[SCH-75] **P. B. SCHNECK**, Automatic Recognition for vector and parallel operations in a higher level language, Sigplan notice, vol 10, n0 3, March 1975.

[SHA-77] **L. G. SHAPIRO and R. J. BARON**, ESP^3 : A language for pattern description and a system for pattern recognition, IEEE Trans. Software Eng. SE-3, pp 169-183, 1977.

[SHA-79] **L. G. SHAPIRO**, Data structures for picture processing : A survey, CGIP 11, pp 162-184, 1979.

[SHA-79] **G. B. SHAW**, Local and regional edge detectors : some comparisons, CGIP 9, 1979.

[STE-79] **S. R. STERNBERG**, Parallel architectures for image processing, in [ONO-81].

[STE-82] **S. R. STERNBERG**, Cellular computers and biomedical image processing, Proc. US-France Seminar Biomedical Image Procesing, Springer-Verlag, 1982.

[STE-86] **S. R. STERNBERG**, Grayscale morphology, CVGIP 35, pp 333-355, 1986.

[TAN-75] **S. L. TANIMOTO and T. PAVLIDIS**, A hierarchical data structure for picture processing, CGIP, vol. 4, n. 2, 1975.

[TAN-77] **S. L. TANIMOTO**, An iconic/symbolic data structuring scheme, in Pattern recognition and Artificial Intelligence (C. H. Chen ed.), pp 452-471, Academic Press, New york, 1977.

[TAN-80] **S. L. TANIMOTO**, Advances in software engineering and their relations to pattern recognition and image processing, 5th International Conf. on Pattern Recognition, vol. 1 of 2, pp 734-741, Dec. 1980.

[TAN-83] **S. L. TANIMOTO**, A pyramidal approach to parallel processing, 10th Annual Int. Conf. on Computer Artchitectures, Stockholm, sweden, pp. 372-378, 13-16 June 1983.

[TED-86] **H. TEDJINI-BAILICHE, P. COLIN, P. L. WENDEL**, Gestion intégrée d'un système multiprocesseur pour le traitement numérique d'image, Deuxième colloque IMAGE, Cognitiva, Tome 2, pp 843-848, Nice, 21/25 Avril 1986.

[TEI-78] **W. TEITELMAN and al.**, Interlisp Reference Manual, Xerox Parc, Palo ALTO, CA, 1978.

[TJA-70] **G. S. TJADEN and M. J. FLYNN**, Detection and parallel execution of independent instructions, IEEE Trans. on computers, vol. C-19, no. 10, pp 889-895, Oct. 1970.

[UHR-81] **L. UHR**, A language for parallel processing of array embedded in Pascal, in [DUF-81].

[UHR-84] **L. UHR**, Multicomputer parallel arrays, piplines and pyramids for pattern perception. VLSI and modern signal processing, pp. 406-421, 1984.

[VOG-84] **V. VOGLEY**, Logiciel d'aide au traitement numérique d'image orienté vers les applications en télédétection. Thèse de doctorat, Strasbourg, Déc. 1984.

[WAL-84] **K. WALL and P. E. DANIELSSON**, A fast sequential method for polygonal approximation of digitized curves, CVGIP 28, 220-227, 1984.

[WED-75] **D. WEDEL**, Fortran for the Texas Instruments ASC System, Sigplan notices, vol. 10, nO 3. March 1975.

[WEG-80] **P. WEGNER**, Programming with ADA: an introduction by means of graduated examples, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[YOK-76] **S. YOKOI, J. P. TORIWAKI and T. FUKOMURA**, Theoreitical analysis of parallel processing of pictures using algebraic properties of pictures operations, Proc. 3rd Internat. Joint Conf. Pattern Recognition, IEEE, pp 723, 1976.

[ZAV-84] **B. ZAVIDOVIQUE**, Parallélisme massif en traitement d'images, Traitement d'images, Journées de synthèse INRIA, 23-24 Jan. 1984.

ANNEXE : EXEMPLE D'IMPLANTATION EN ADA

Ce travail a été effectué en collaboration avec J. Jaray et avec l'aide de deux élèves ingénieurs J. D. Malpelet et E. Valette de l'école des mines de Nancy (EMN). Les programmes qui suivent ont été écrits sur un PC AT et validés sur cette machine.

1. Quelques notions d'Ada

Nous allons donner dans la suite quelques notions préliminaires d'Ada utiles pour représenter les types précédemment introduits.

1.1. Types dérivés et sous-types

```
type T is new R;
```

définit un nouveau type T isomorphe à R et dont les accès ont même nom que ceux de T. Ceci n'est pas à confondre avec le **sous-type** qui impose une contrainte sur le type, comme :

```
subtype ST IS T contrainte;
```

les objets de type ST ont pour type T.

1.2. Types composés

Ada permet de déclarer des types tableau paramétrés:

```
type matrice is array (integer range <> ; integer range <>)
                of float
```

Le type matrice peut alors être le type d'un paramètre formel d'un sous programme, auquel cas les véritables dimensions seront définies par l'argument effectif de l'appel.

1.3. Surcharge des opérateurs

La surcharge d'opérateurs arithmétiques et logiques consiste à les redéfinir sur des opérandes de types différents. En Ada, on peut aussi surcharger les fonctions et les sous-programmes dans le même niveau du programme, à condition que leurs arguments soient différents.

exemple:

```
function "+"(x,y : matrice) return matrice is
       i,j : integer;
       res : matrice;
       begin
          for i in x'first(1)..x'last(1) loop
                 for j in x'first(2)..x'last(2) loop
                     res(i,j) := x(i,j)+y(i,j)
                 end loop;
          end loop;
          return res
       end "+";
```

on peut ensuite écrire:

```
       k,i,j : integer;
       z : matrice; ...
       k := i+j;
       z := x+y;

       (* l'opérateur "+" dans x+y est différent de celui dans i+j *)
```

1.4. les types génériques

Un type générique est un type qui peut être instancié pour une utilisation précise.

exemple:

```
generic type T is PRIVATE
procedure echange(x,y : in out T) is
       z : T;
       begin
          z := x; x:=y ; y := z;
       end echange;
```

Pour l'utiliser pour échanger deux matrices, il faudra l'instancier par:

```
       procedure echmat is new echange (T => MATRICE);
```

puis, on écrira: echmat(A,B)

1.5. Le paquetage

Un paquetage est une construction du langage qui permet de réaliser modularité et abstraction et entre autres d'implanter les types abstraits. Le paquetage comprend deux parties:

(1) une **définition** où l'on donne l'interface du paquetage, c'est-à-dire la liste de toutes les procédures, paquetages et types qui peuvent être réutilisés dans le programme où est déclaré le paquetage.

Exemple :

```
package position is

       type position is record
                              x : integer;
                              y : integer
                           end record;

       type direction is range 1..8;

       function est_ce_posvoisine(p1,p2 : in position) return boolean;
       function est_ce_posegale(p1,p2 : in position) return boolean;
       function distposition(p1,p2 : in position) return integer;
       function posvoisine(p1: in position;dir: integer) return position;

end position;
```

(2) un **corps** où les fonctions sont réalisées: c'est la partie cachée qui donne le corps des fonctions utilisées. L'utilisateur n'a pas accès à cette partie et ne peut donc connaître l'implémentation des fonctions ni avoir accès aux objets qui servent à réaliser le paquetage.

```
package body position is

        function est_ce_posvoisine(p1,p2 : in position) return boolean is

        res: boolean;
        begin
          res := not (p1=p2) and ((abs(p1.x - p2.x) <= 1) or
                 (abs(p1.y - p2.y) <= 1));
          return res
        end est_ce_posvoisine;

        function est_ce_posegale(p1,p2 : in position) return boolean;
           begin
                ...
           end est_ce_posegale;

        function distposition(p1,p2 : in position) return integer;
           begin
                ...
           end disposition;

procedure posvoisine(p1: in position;dir: direction;p2: out position);
   begin
                ...
   end posvoisine;
end position;
```

Utilisation du paquetage : clauses **WITH** et **USE**

- La clause **with** CONTEXTE rend visible toute la partie déclarative de CONTEXTE. Les noms des procédures, fonctions et types pourront être utilisés en les préfixant par le nom de l'unité CONTEXTE.

- La nécessité de préfixer peut être levée en utilisant la clause **USE** mais au risque de créer des confusions entre les noms de CONTEXTE et ceux du programme qui l'utilise.

1.6. Package générique

La généricité est un des outils de Ada qui permet de réaliser des logiciels puissants. Elle permet d'écrire des sous-programmes ou des paquetages dont le type des paramètres est formel, ou dits **génériques**.

les paramètres génériques peuvent être:

- des types (c'est le cas le plus fréquent),
- des constantes,
- des fonctions.

---

exemple **de paquetage générique**

```
(* partie paramètres genériques *)

generic

   type info is private;
   type vector is array (positive range <>) of info;
   with function somme (x,y ; info) return info;

package vecteur is

        function addition (a,b : in vector) return vector;

end vecteur;

package body vecteur s

        function addition(a,b : in vector) return vector is

              iinf : vector'range := vector'rangéfirst;
              imax : vector'range := vector'rangélast;
              i    : vector'range;
              res  : vector;
              begin
                for i in iinf..imax loop
                      res(i) := somme (a(i),b(i));
                end loop;

        end addition;
end vecteur;
```

instanciations possibles :

```
with vecteur;

procedure additon5 is


        type info is integer;
        type champ is range 1..5;
        type vector is array (champ) of integer;

        % déclaration de la fonction somme %

        function somme_entier (a,b : integer) return integer is
          begin
            return a+b;
        end somme_entier;

        % instanciation du paquetage vecteur %

        package vecteur_entier is new vecteur (info => info,
                                               vector => vector,
                                               somme => somme_entier);
        use vecteur_entier;
          ...
        end vecteur_entier;

end addition5;
```


2. Le paquetage grille

2.1. paramètres de grille :

```
with p_position; use p_position;

generic
   type info  is private
   type champ_x is range <>;
   type champ_y is range <>;
    info_ind : info;
```

2.2. déclaration du paquetage grid:

```
package p_grille is

     type grid is private;
     type mask is private;
     procedure def_grid (gr : in out grid; p in position;
                         infor: in info);
     function acces(gr : in grid ; p : in position) return info;
     generic

     with function f(p : in position) return boolean;
     function acces_cond(gr: in grid; p: in position) return info;
     generic

     with function g(p : in position) return position;
     function modif_simple(gr : in grid) return grid;
     generic

     with function h(a,b : in info) return info;
     function modif_double(gr1,gr2 : in grid) return grid;

     private
        type grid is array (champ_x,champ_y) of info;
        type mask is record
                partl: grid;
                centre : position;
             end record;

end p_grille;
```

2.3. déclaration explicite

```
package body p_grille is

    procedure def_grid (gr: in out grid; p in position;
                        infor : in info) is
       begin
          gr(p.x,p.y):=infor;
    end def_grid;

    function acces(gr : in grid ; p : in position) return info is
    begin
           return gr(p.x,p.y);
    end acces;

    function acces_cond(gr: in grid; p: in position) return info
       is
       begin
         if f(p) then return gr(p.x,p.y) else return info_ind
         end if;
    end acces_cond;

    function modif_simple(gr : in grid) return grid;
       res : grid;
       i : champ_x;
       j : champ_y;
       p : position;
       begin
         for i in champ_x'first.. champ_x'last loop
            for j in champ_y'first.. champ_y'last loop
               begin
                  p.x := i;
                  p.y := j;
                  def_grid(res,g(p),acces(gr,p));
               end loop
         end loop;
         return res
       end modif-simple;

    function modif_double(gril1,gril2 : in grid) return grid;
       res : grid;
       i : champ_x;
       j : champ_y;
       p : position;
```



```
       begin
         for i in champ_x'first.. champ_x'last loop
            for j in champ_y'first.. champ_y'last loop
               begin
                  p.x := i;
                  p.y := j;
                  def_grid(res,g(p),h(acces(gril1,p),acces(gril2,p));
               end loop
         end loop;
         return res
       end modif-double;
end p_grille;
```

3. Le paquetage IMAGE

3.1. Le contexte

```
        with p_grille;
```

3.2. La déclaration

```
generic
        type info_im is range <>;
package p_image  is
        déclaration des paramètres

        type indice is new integer range 0..255;
        var_ind : integer;
        package image is new p_grille (info => info_im,
                                       champ_x => indice,
                                       champ_y => indice,
                                       info_ind => var_ind);
        type timage is new image.grid;

        déclaration des fonctions

        function dynamique (i: in timage) return info_im;
        function seuillage (i: in timage; seuil: in info_im) return timage;
        function voisinage (i : in timage ; p : in position ;
                        haut,larg : in integer) return tmask;

        generic
        with function fmask (m,m' : in tmask) return info_im;
        function convolution (i : in timage ; m : in tmask) return timage;
end p_image;
```


3.3. déclaration explicite

```
package body image is

% instanciation des fonctions paramètres %
        function filtre(p: in position) return boolean is
        begin
              if (p.x mod 2)=0 or (p.y mod 2)=0 then return true
              end if;
        end filtre;

        function symetrie(p: in position) return position is
            xi : gr := gr'first;
            yi : gr := gr'last;
            pl : position;
            begin
                pl.x := xi+yi-p.x;
                pl.y := p.y;
            return pl;
        end g;

        function somme_info (info1,info2 : integer) return integer is
        begin
           return info1+info2;
        end somme_info;

        function mult_info(info1,info2 : integer) return integer is
        begin
           return info1*info2;
        end mult_info;

% instanciation des fonctions sur grille %
function acces_cond1 is new acces_cond (f => filtre);
function symetriex is new transf_geom (g => symetrie);
function "+" is new modif_double (h => somme_info);
function "*" is new modif_double (h => mult_info);
```

définition des fonctions caractéristiques

```
function dynamique (im: in timage) return info_im is
   i,j : integer;
   p : position
   min : info_im := 255;
   max : info_im := 0;
   begin
        for i in champ_x'first .. champ_x'last loop
                for j in champ_y'first .. champ_y'last loop
                   p.x := i; p.y := j;
                   if min > acces(im,p) then min := acces(im,p)
                   else max := acces(im,p)
                   end if;
                end loop;
        end loop;
        return max-min

end dynamique;

function seuillage (im: in timage; seuil: in info_im) return timage;
i,j : integer;
res : timage;

begin

   for i in champ_x'first .. champ_x'last loop
        for j in champ_y'first .. champ_y'last loop
        p.x := i;
        p.y := j;
        if acces-condl(im,p) then if acces(im,p) < seuil then
        def_grid(res,p,0) else def_grid(res,p,acces(im,p))
        end if; end if;
        end loop
   end loop;
   return res

end seuillage;
```



```
function voisinage (im : in timage ; p : in position ;
                    haut,larg : in integer) return tmask is
   i,j,i0,j0,il,jl: integer;
   p : position;
   res : mask;
   begin
        i0 := p.y - haut div 2;il := -1;
        j0 := p.x - larg div 2;jl := -1;
        for i in i0..i0+haut-1 loop
                il := il+1;
                for j in j0..j0+larg-1 loop
                jl := jl+1;
                p.x := i; p.y := j;
                pl.x := il; pl.y := jl;
                def_grid(res,pl,acces(im,p));
                end loop;
        end loop;
        return res;

end voisinage;

function convolution (im : in timage ; m : in tmask) return timage is
   i,j,haut,larg : integer;
   p : position;
   res : timage;
begin
   haut := partl.champ_y'last - partl.champ_y'first + 1;
   larg := partl.champ_x'last - partl.champ-x'first + 1;
   for i in champ_y'first..champ_y'last loop
           for j in champ_x'first..champ_x'last loop
           p.x := i;
           p.y := j;
           def_grid(res,p,fmask(voisinage(im,p,haut,larg),m))
           end loop;
   end loop;
   return res;

end convolution;

end image;
```

3.4. l'instanciation d'image par une image binaire

```
with p_image;

procedure binaire is

        subtype bin is integer range 0..1;
        package p_binaire is new p_image (info_im => bin,
                                          info_ind => 0);
        use p_binaire;
        type tbin is new p_binaire.timage;

        function squelette( m,m' : in tmask) return bin is
        i,j : integer;
        res : integer range 0..1;
        p : position;
        begin
           for i in partl.champ_x'first..partl.champ_x'last loop
                   for i in partl.champ_x'first..partl.champ_y'last loop
                        p.x := i;
                        p.y := j;
                        if res=1 then if acces(m,p)<>acces(m',p) then res := 0
                        end if;
                    end loop
            end loop;
            return

        end squelette;

        function squelettisation is new convolution (fmask => squelette);

        % manipulation du binaire %

        .....

end binaire;
```


institut
national
polytechnique
de lorraine

Le Président,

AUTORISATION SOUTENANCE DE THESE

EN VUE DE L'OBTENTION DU GRADE DE DOCTORAT D'ETAT-SCIENCES

—

VU LES RAPPORTS ETABLIS PAR :

Monsieur le Professeur FINANCE,
Monsieur le Professeur LEVIALDI,
Monsieur le Professeur MOHR.

Le Président de l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, autorise

Monsieur BELAID Abdelwaheb

à soutenir devant l'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE, une thèse intitulée. :

"Méthodes et outils de programmation de systèmes de traitement d'images : le projet SAPIN".

en vue de l'obtention du grade de DOCTORAT D'ETAT-SCIENCES,

Spécialité : "Sciences"

Fait à VANDOEUVRE, le 9 Juin 1987

Le Président de l'I.N.P.L.

M. GANTOIS

INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE NANCY

2, avenue de la Forêt-de-Haye - Vandœuvre

RESUME

Le travail rapporté dans cette thèse a été effectué dans le cadre de deux propositions de collaboration, l'une de l'ENSPS qui développe une machine multiprocesseurs : Icotech et l'autre de l'ETCA qui étudie la réalisation d'une machine pyramidale : Sphinx.

Pour ces deux projets, l'objectif était de concevoir un logiciel cohérent avec ces architectures en le simulant afin de donner une idée précise des performances.

Bien que ces deux machines soient fondamentalement différentes, la démarche pour la conception du logiciel fut la même. Une première étude a porté sur la définition des types image. A partir de leurs propriétés et différentes utilisations, une spécification formelle a été effectuée. Elle a conduit à une vision plus unificatrice ramenant la définition des types à celle de leur constructeur. Ceci fut ensuite le point de départ de deux études :

- la définition et l'implantation d'un langage interactif pour le traitement d'images. Destiné à un public non-informaticien, ce langage a pris la forme d'un générateur de menus offrant plusieurs formes d'assistance à la programmation,

- la définition et la réalisation d'un surlangage comprenant des types image et des instructions spécialisées.

Mais cette thèse voulait aussi apporter un point de vue plus global sur la parallélisation des algorithmes et leur exécution efficace sur les architectures spécialisées. Deux reflexions ont été faites pour les deux machines étudiées.

MOTS-CLES

Traitement d'images - Environnement logiciel - Système SAPIN - Types abstraits image - Architectures parallèles - Détection du parallélisme.