République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Ferhat Abbas -Sétif-

Institut d'Electronique

Thèse de Magister

Option : Contrôle industriel

THEME

PARALLELISATION D'ALGORITHMES DE TRAITEMENT D'IMAGES SUR UN RESEAU DE TRANSPUTERS ET APPLICATION A L'ANALYSE DES SURFACES DE SILICIUM METALLISEES

Présentée par : **TAREK ABED**
soutenue le : 23 / 05 / 1996

Devant le jury composé de :

Messieurs :

A. KHELLAF	**M.C.**	**Université de Sétif**	**Président**
A. BAKHTI	**C.C.**	**Université de Sétif**	**Rapporteur**
T. MOHAMADI	**M.C.**	**Université de Sétif**	**Examinateur**
M. BOUAMAR	**C.C.**	**Université de M'sila**	**Examinateur**

A mes parents

et tous ceux qui me sont chers

REMERCIEMENTS

Je tiens à remercier très chaleureusement Monsieur A. KHELLAF, maître de conférences à l'institut d'électronique, de l'honneur qu'il me fait en présidant le jury de cette thèse.

Je tiens à exprimer toute ma gratitude à Monsieur A. BAKHTI chargé de cours à l'institut d'électronique, pour m'avoir orienté et pour la confiance et l'intérêt qu'il a témoigné tout au long de l'élaboration de ce travail.

Que Monsieur T. MOHAMADI, maître de conférences à l'institut d'électronique, trouve ici l'expression de mes remerciements pour avoir accepté de juger mon travail.

Je remercie également Monsieur A. MOSSER, chargé de recherche au CNRS à Strasbourg et responsable de l'accord programme Constantine-Sétif-Strasbourg, d'avoir jugé ce travail et pour sa contribution à son élaboration.

Je suis fort reconnaissant à Monsieur M. BOUAMAR, chargé de cours à l'université de M'sila, pour l'intérêt qu'il a accordé à mon travail et d'avoir bien voulu participer au jury.

Enfin, ma profonde gratitude va à tous ceux qui, de près ou de loin, m'ont aidé, que ce soit par leur amitié ou leur soutien et encouragements.

SOMMAIRE

INTRODUCTION

La capacité d'apprendre, de communiquer, de raisonner, de prendre des décisions et d'agir, regroupe les éléments essentiels de l'intelligence humaine. L'un des principaux objectifs de cette fin de siècle, est de traduire ces facultés en termes de systèmes artificiels. Ce défi technologique ne pourra être relevé que grâce aux progrès fantastiques réalisés principalement en électronique et en informatique.

Les facultés d'apprentissage et de raisonnement traduisent l'intelligence artificielle. La communication et l'aide à la décision se traduiront en termes de traitement du signal et de l'image. L'action, décomposée en contrôle et commande définira les outils fondamentaux de l'automatique. Nous créons de cette manière une chaîne de traitement qui permet de passer de l'observation à la prise de décision. Le traitement d'images forme la discipline clé qui assure la continuité de la chaîne de traitement de l'information, depuis l'acquisition des signaux jusqu'à leur interprétation. Le traitement d'images joue déjà un rôle important dans de très nombreuses disciplines comme la médecine, la robotique, les télécommunications et la physique pour ne citer que celles-ci.

L'utilisation croissante du traitement d'images dans ces différents domaines pour observer les images de notre environnement et les analyser a conduit à un développement considérable 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 celle des machines conventionnelles.

Ces différentes investigations ont porté sur la multiplication des unités de traitement (architectures parallèles) 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. Cette répartition est le problème essentiel que le logiciel doit résoudre dans le but de tirer le meilleur profit des possibilités offertes par l'architecture.

En traitement d'images, les algorithmes usuels sont conçus pour être traités sur une machine séquentielle. L'exploitation de ces algorithmes sur ce type de machines nécessite un temps de traitement très important.

Dans ce travail, on se propose donc de paralléliser ces algorithmes en vue de leur exploitation sur une machine à architecture parallèle dans le but de diminuer le temps de traitement. Cette machine est conçue autour d'un micro-ordinateur type PC/AT et d'une carte comportant cinq (5) Transputers. Ces derniers travaillent en parallèle sur cinq processus concurrents. Il s'agit donc de décomposer chaque algorithme en cinq processus concurrents et de les implanter sur la machine parallèle. Pour réaliser le test de ces algorithmes, nous avons choisi comme application, l'étude de la morphologie des surfaces de Silicium métallisées.

Le premier chapitre de ce manuscrit, décrit brièvement les architectures parallèles de base, puis détaille l'architecture parallèle que nous avons adoptée pour l'implantation des algorithmes parallèles : celle des réseaux de Transputers.

Le second chapitre est consacré à l'étude du langage de programmation parallèle Occam et du système de développement pour Transputers TDS d'INMOS. Ceci a pour but d'introduire les différentes structures de l'Occam exploitées pour la parallélisation des algorithmes.

Le troisième chapitre contient une étude théorique concernant les traitements d'images ainsi que les résultats obtenus après application des algorithmes sur des images permettant de montrer les performances du système en termes de traitements.

Les structures séquentielles et parallèles des algorithmes sont décrites par le quatrième chapitre. Nous avons opté pour une description algorithmique qui se rapproche le plus possible de la programmation en Occam. Ceci a pour but de décrire d'une manière rigoureuse la façon avec laquelle sont implantés les différents algorithmes parallèles qui exploitent des structures de programmation propres à ce langage. Les algorithmes parallèles seront testés sur un réseau de Transputers formé par quatre Transputers sous le contrôle d'un Transputer maître. Les performances en termes de gain en temps seront donc par la suite évaluées.

Le dernier chapitre est entièrement dédié à l'application du traitement d'images à l'étude de la morphologie des surfaces de Silicium métallisées. Cette étude permet de noter l'apport que peut procurer le traitement d'images dans ce type d'applications scientifiques.

Chapitre I

ARCHITECTURES PARALLELES ET TRANSPUTERS

1.1 Introduction

Durant les cinq dernières décades, l'industrie des ordinateurs a connu quatre générations marquées par un progrès très rapide de la technologie : relais et tubes à vide (1940 - 1950), diodes et transistors (1950 - 1960), circuits SSI et MSI (*small, medium scale integration*) (1960 - 1970) et composants LSI et VLSI (*large, very large scale integration*) [1].

Ces avancées technologiques considérables obéissaient aux besoins des scientifiques de disposer d'ordinateurs de plus en plus performants, nécessaires à diverses applications scientifiques et autres.

Cependant, les composants rapides et très peu encombrants ne sont pas les seuls critères de performance des ordinateurs. Car depuis le concept de la machine de Von Neumann, un ordinateur est plus qu'un problème hardware. La performance d'un système moderne dépend aussi bien de l'architecture que des techniques de traitement.

1.2 Description des structures parallèles de base

Les ordinateurs parallèles sont divisés en trois configurations architecturales de base [1] :

- Ordinateurs *pipeline*.
- *Array* processeurs.
- Systèmes multiprocesseurs.

Un ordinateur *pipeline* réalise des traitements simultanés en exploitant le **parallélisme temporel**. Un *array* processeur utilise plusieurs unités arithmétiques et logiques (ALU) pour exploiter le **parallélisme spatial**. Enfin, un système multiprocesseurs réalise des traitement parallèles d'une manière asynchrone, grâce à un ensemble de processeurs et ressources partagées.

Ces trois approches, dans la conception de systèmes parallèles, ne sont pas mutuellement exclusives. En fait, la majorité des ordinateurs existant intègrent le parallélisme pipeline, et certains possèdent en plus une structure array ou multiprocesseurs.

La différence fondamentale entre un array processeur et un système multiprocesseurs est que tous les éléments de traitement d'un array processeur opèrent d'une manière synchrone, tandis que les unités de traitement d'un multiprocesseurs peuvent réaliser des traitements asynchrones.

1.2.1 Les ordinateurs *pipeline* :

L'exécution d'une instruction sur un processeur nécessite quatre cycles : recherche de l'instruction (IF) dans la mémoire; décodage de l'instruction ou identification de l'instruction à exécuter (ID); recherche de l'opérande (OF); et enfin l'exécution de l'opération arithmétique ou logique (EX). Dans un processeur non pipeline, tous ces cycles doivent être terminés avant l'exécution de la prochaine instruction, alors que dans un processeur pipeline, des instructions successives s'exécutent simultanément. La figure 1.1 montre la structure d'un processeur pipeline linéaire à quatre étages.

Dans ce type de processeurs, une instruction est exécutée en quatre cycles. Un cycle est défini comme le temps que met l'étage le plus lent dans la chaîne pour

terminer son traitement. Une fois le pipeline rempli, un résultat est obtenu à chaque cycle. Un processeur pipeline linéaire à **k** étages peut être au plus **k** fois plus rapide qu'un processeur non pipeline.

Les processeurs pipeline sont mieux adaptés pour réaliser les mêmes opérations d'une manière répétitive, car si une opération, dans un étage change (par exemple d'une soustraction à une multiplication), le pipeline doit être reconfiguré, ce qui cause des pertes considérables du point de vue temps [1].

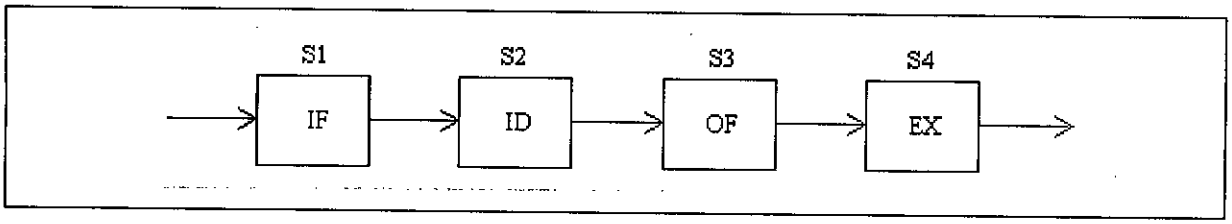


Fig. 1.1 Processeur pipeline.

1.2.2 Les *array* processeurs :

Un array processeur est un ordinateur parallèle, synchrone, constitué de plusieurs unités arithmétiques et logiques dites **éléments de traitement** (PE : *processing elements*). Tous ces éléments sont synchronisés pour réaliser la même fonction sur des données différentes. Ceci leur permet donc de réaliser un parallélisme spatial. Les array processeurs peuvent avoir plusieurs configurations topologiques (linéaire, en étoile, hypercubique, pyramidale etc...). La figure 1.2 montre quelques exemples de configurations [1].

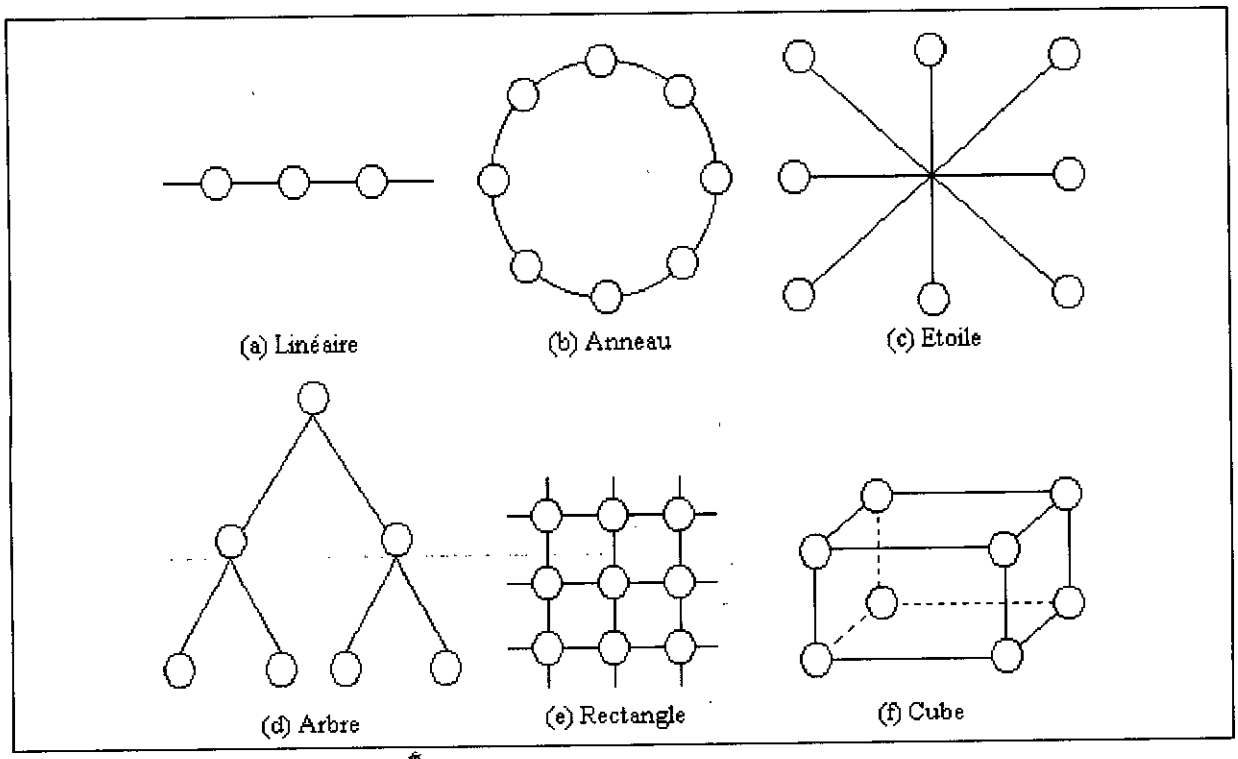


Fig. 1.2 Exemples de topologies des array processeurs.

1.2.3 Les systèmes multiprocesseurs :

Ces systèmes sont formés de plusieurs processeurs qui se partagent de la mémoire, des canaux d'entrée sortie, des périphériques et qui communiquent entre eux par l'intermédiaire d'un réseau d'interconnexion qui peut être :

- Un bus partagé.
- Un réseau de commutateurs (*Crossbar switch*).
- Des mémoires multi-ports.

Les processeurs exécutent simultanément plusieurs tâches différentes, ce qui leur permet d'assurer une grande diversité de calcul. En augmentant le nombre d'unités de traitement, dans les limites raisonnables, on arrive à augmenter d'une façon considérable, la puissance de la machine.

Cependant, l'inconvénient principal de cette architecture est le conflit d'accès aux ressources communes. La performance de ces systèmes dépend donc de l'occurrence de ces conflits qui nécessite la mise en place d'unités d'arbitrage, très coûteuses en temps, d'où la limitation du nombre de processeurs [1][2].

1.3 Classification des architectures d'ordinateurs

Selon la classification de Flynn (1966), les ordinateurs peuvent être répartis en quatre catégories, suivant la multiplicité des flux des instructions et des données [1] :

SISD : (*Single Instruction stream - Single Data stream*)

Instruction unique - donnée unique.

Cette catégorie représente la majorité des machines séquentielles qui existent. Les instructions sont exécutées d'une manière séquentielle ou en pipeline. Le schéma de principe est illustré par la figure 1.3.a.

SIMD : (*Single Instruction stream - Multiple Data stream*)

Instruction unique - données multiples.

Cette classe correspond en particulier aux array processeurs. Comme le montre la figure 1.3.b, tous les éléments de traitement exécutent la même opération sur des données différentes sous la gestion de l'unité de contrôle.

MISD : (*Multiple Instruction stream - Single Data stream*)

Instructions multiples - donnée unique.

Cette organisation est illustrée par la figure 1.3.c. Il y a *n* processeurs, chacun recevant des instructions différentes et opérant tous sur le même flux de données. Le résultat (la sortie) d'un processeur est l'opérande (l'entrée) du suivant.

MIMD : (*Multiple Instruction stream - Multiple Data stream*)

Instruction multiples - données multiples.

La majorité des systèmes multiprocesseurs sont classés dans cette catégorie (figure 1.3.d). Une architecture MIMD est dite intrinsèque lorsque le système est à mémoire partagée. Dans le cas où la mémoire est divisée en sous espaces dont chacun est réservé à un processeur, l'architecture est dite MSISD, équivalente à une architecture de plusieurs systèmes uniprocesseurs SISD indépendants.

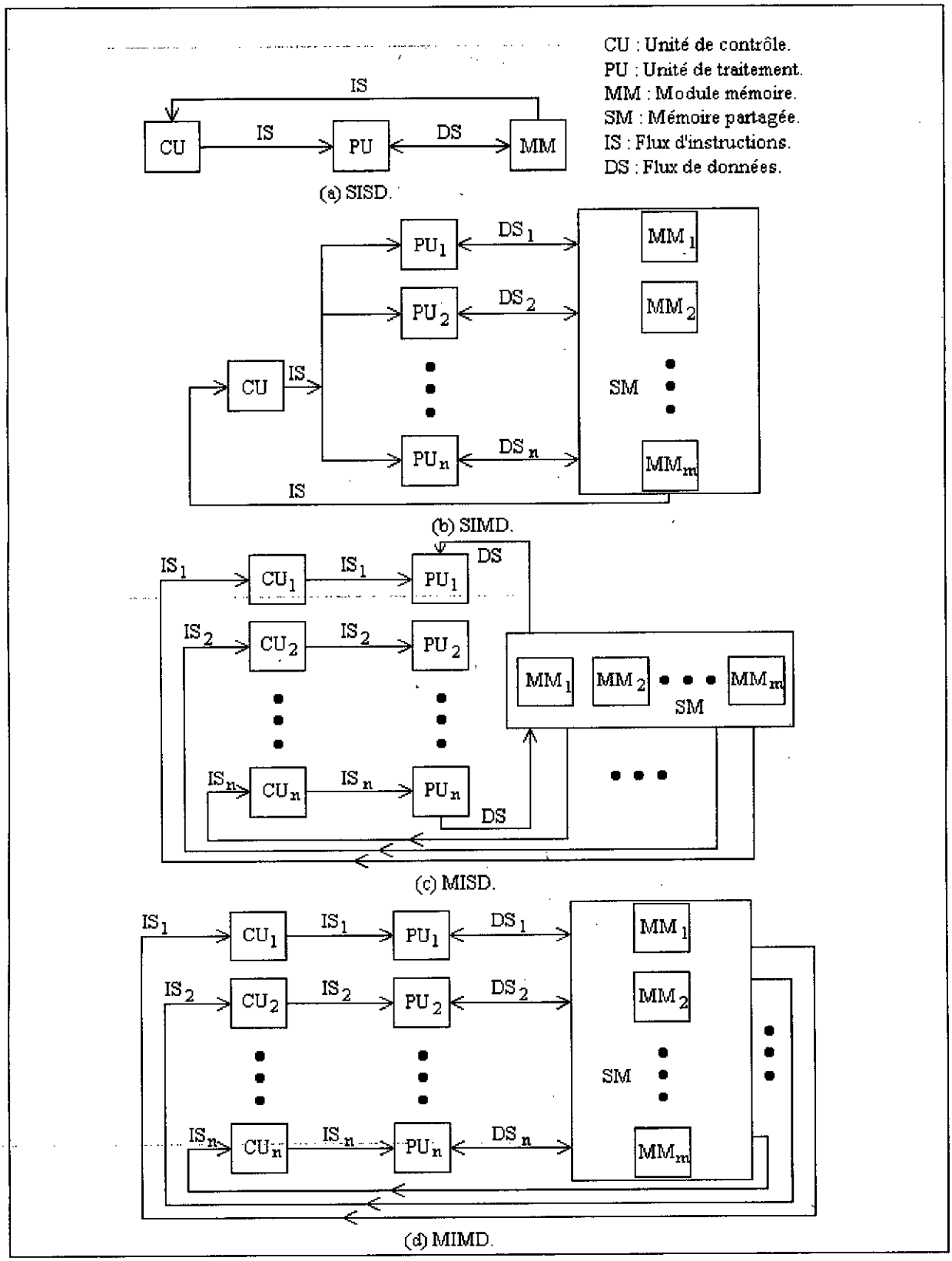


Fig. 1.3 Classification de Flynn des architectures d'ordinateurs.

1.4 Les Transputers (Etude du Transputer IMS T800 de INMOS)

1.4.1 Introduction :

La famille de Transputers de INMOS (membre de CGS THOMSON) est une gamme de composants VLSI intégrant une unité de traitement, de la mémoire et des liens de communication. Le premier produit de cette gamme est le IMS T414, sorti en Septembre 1985. Le IMS T800 est sorti en 1987, c'est un Transputer 32 bits offrant en plus (par rapport au T414) une unité de traitement en virgule flottante, de nouvelles instructions destinées à des applications graphiques et plus de mémoire intégrée.

Le T800 fait partie du projet *P1085 European ESPRIT* dont le but est de concevoir un ordinateur parallèle constitué d'un réseau reconfigurable de Transputers, et destiné à des applications en physique, CAO et traitement d'images [3].

1.4.2 Le Transputer : architecture et concepts

. Un composant programmable :

Le Transputer est un composant VLSI intégrant une unité de traitement, une mémoire locale et des liens de communications (*links*) pour le connecter à d'autres Transputers. Les systèmes à Transputers peuvent être conçus et programmés en utilisant l'**Occam** (langage de programmation parallèle décrit dans le chapitre II) qui permet de décrire une application comme un ensemble de processus concurrents qui communiquent entre eux grâce à des canaux.

. Processeur et mémoire sur la même puce :

L'intégration d'une mémoire locale permet de réduire le temps d'accès par rapport à l'utilisation d'une mémoire externe. Ce qui permet donc d'accélérer les traitements sur le processeur.

. Les liens de communication sérielle :

Les systèmes multiprocesseurs, où la communication se fait à travers un bus partagé, sont ralentis et nécessitent une logique de contrôle additionnelle pour le partage du bus. Des liaisons sérielles sont donc utilisées dans les réseaux de Transputers, dans le but de réduire le nombre de connexions et d'accélérer les communications.

. Jeu d'instructions :

Le jeu d'instructions des Transputers contient une partie destinée à l'implémentation des programmes séquentiels sur le processeur, en plus de groupes d'instructions spécialisées pour le calcul arithmétique et l'allocation du processeur, dans le cas de l'exécution de processus concurrents sur un seul Transputer.

1.4.3 L'architecture du Transputer IMS T800 :

Le Transputer IMS T800 intègre un processeur (CPU) 32 bits s'inspirant des concepts RISC (*Reduced Instruction Set Computer*), une unité de traitement en virgule flottante (FPU) sur 64 bits, 4 K.octets de mémoire interne et dispose de quatre liens série ayant chacun un débit de 20 M bits / sec. Tous ces éléments sont reliés entre eux par un bus 32 bits permettant également d'adresser 4 G octets de mémoire externe [3][7]. Le schéma bloc du T800 est illustré par la figure 1.4.

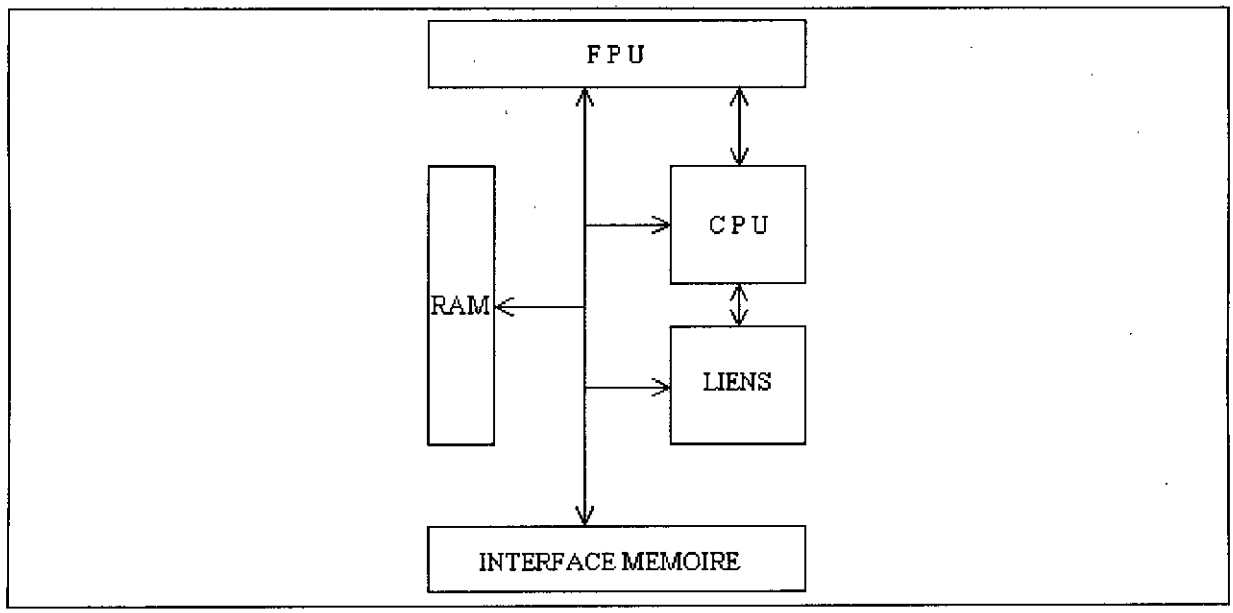


Fig. 1.4 Schéma bloc du Transputer IMS T800.

La CPU du Transputer possède un nombre réduit de registres (6) utilisés pour l'exécution des programmes séquentiels. Ces registres sont :

- Le registre pointeur du champ de travail qui pointe la zone mémoire contenant les variables d'un programme.
- Le registre pointeur d'instructions contenant l'adresse de la prochaine instruction à exécuter.
- Le registre opérande.

- Trois registres A B et C qui sont les sources et destination pour la plupart des opérations arithmétiques et logiques. Ces registres forment une pile : charger une valeur dans A transfère le contenu de B dans C et celui de A dans B, et de la même manière, stocker une valeur à partir de A transfère le contenu de B dans A et celui de C dans B [3].

. Codage des instructions :

Tous les Transputers ont le même jeu d'instructions de base. Toutes ces instructions ont un même format, choisi pour donner une représentation compacte aux opérations les plus fréquentes dans les programmes. Chaque instruction est codée par un seul octet divisé en deux parties. Les quatre bits de poids fort représentent le code de la fonction, ceux de poids faible constituent la donnée. Les seize fonctions ainsi codées incluent les opérations de chargement, de stockage et de sauts. Ce codage ne permet que quatre bits pour l'opérande, d'autres fonctions disponibles, utilisent le registre opérande pour étendre la taille des données manipulées. De plus, des fonctions manipulent des opérandes contenus dans les registres du processeur, ce qui permet d'avoir plus de seize fonctions toutes codées par un seul octet. Le T800 possède des instructions additionnelles pour le calcul en virgule flottante ainsi que des instructions destinées à des applications graphiques.

Les instructions de base se répartissent en trois classes :

- Les fonctions directes, au nombre de treize, agissent sur un opérande appartenant à l'intervalle 0-15.
- Les fonctions préfixées, au nombre de deux, sont dédiées à l'extension du format des opérandes.
- Les fonctions indirectes, sont obtenues au moyen du mot clé *operate* et interprètent l'opération comme une fonction agissant sur les valeurs de la pile de registres.

. Les processus :

On appelle processus une séquence d'instructions exécutant une tâche donnée. Un processus peut s'arrêter à la fin de la séquence ou sans avoir accompli toutes les instructions qui le composent. Le Transputer peut traiter plusieurs processus en même temps. A un instant donné un seul processus peut disposer du processeur, les autres sont placés dans une liste chaînée (chaque processus de la liste contenant l'adresse du suivant).

A un instant, un processus peut être actif (en exécution ou en attente d'exécution) ou inactif (en attente de données externes ou pour un temps spécifié). Le grand avantage du Transputer est d'intégrer un ordonnanceur (*scheduler*) micro codé, supprimant ainsi l'addition d'une couche logicielle supplémentaire du type noyau temps réel. L'ordonnanceur partage le temps du processeur entre plusieurs processus (*time sharing*).

Tout processus subsiste tant que sa séquence d'instructions n'a pas été entièrement exécutée. Dans le Transputer, on dispose d'un élément matériel appelé allocateur de ressources qui réalise la commutation d'un processus avec un autre. L'utilisation de l'allocateur est déterminée par le code compilé obtenu à partir du programme source. L'occupation du processeur par un processus peut être interrompue de deux manières :

- Le processus peut être désactivé par mise en attente d'une communication ou par attente de l'expiration d'un délai.
- La fin du nombre de tranches de temps processeur qui lui sont allouées (*time slicing*).

. Opérations en virgule flottante :

L'unité de traitement en virgule flottante (FPU) du T800 opère en parallèle avec la CPU. C'est à dire que la CPU peut effectuer un calcul d'adresse alors que la FPU exécute une opération en virgule flottante. Le jeu d'instruction de la FPU a été rigoureusement choisi. Pour aboutir à celui-ci, plusieurs possibilités ont été proposées, puis testées sur des programmes de calcul numérique et ce dans le but d'obtenir un maximum de performances (en temps et en code). La table 1.1 indique les temps nécessaires à l'exécution de différentes opérations pour les versions 20 et 30 MHz du T800.

	IMS T800-30		**IMS T800-20**	
Opération	**Simple précision**	**Double précision**	**Simple précision**	**Double précision**
Addition	233 ns	233 ns	350 ns	350 ns
Soustraction	233 ns	233 ns	350 ns	350 ns
Multiplication	367 ns	667 ns	550 ns	1000 ns
Division	567 ns	1067 ns	850 ns	1600 ns

Table 1.1

Les temps d'opérations ne sont pas très appropriés pour la mesure des performances. Pour cette raison celles-ci sont mesurées par des programmes spécialisés (*Benchmarks*). Le ***Whetstone*** est le plus utilisé car il procure un grand nombre d'opérations et inclut des appels de procédures et des manipulations de tableau. Il constitue dans un certain sens le programme scientifique typique.

La table 1.2 compare les performances du T414 et du T800 avec celles d'autres processeurs en utilisant le *Whetstone*.

Processeur	**Whetstones / seconde Simple précision**
Intel 80286/80287 8 MHz	300 K
IMS T414-20 20 MHz	663 K
NS 32332-32081 15 MHz	728 K
MC 68020/68881 16/12 MHz SUN 3	755 K
VAX 11/780 FPA UNIX 403 BSD	1083 K
IMS T800-20 20 MHz	4000 K
IMS T800-30 30 MHZ	6000 K

Table 1.2

. Les liens de communication :

Les communications sérielles inter-Transputers se font octet par octet, ceci a pour avantage de ne nécessiter qu'un buffer d'un octet dans le Transputer récepteur. Chaque octet transmis est précédé par un bit de start (à 1) et un bit à 1 puis est suivi d'un bit de stop (à 0). Pour synchroniser les communications, chaque message transmis doit être accompagné en retour d'un accusé de réception. L'émetteur attend donc cet accusé de réception qui consiste en un bit de start et un bit de stop. La réception de cet accusé signifie que le Transputer récepteur a bien reçu l'octet émis et que la ligne est prête à véhiculer un autre octet. Le protocole de communication permet qu'un accusé de réception puisse être envoyé dès que l'acquisition de l'octet commence, ainsi les communications sont continues et il n'y a pas de temps mort entre l'envoi de deux octets [3][7].

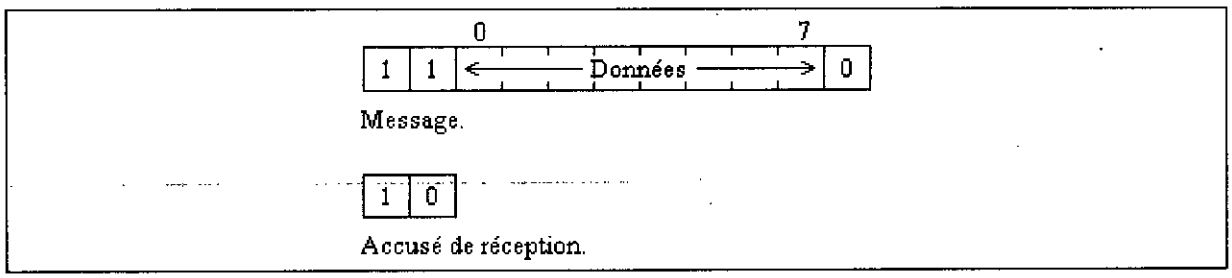


Fig. 1.5 Protocole de communication entre Transputers.

1.5 Les réseaux de Transputers

Il existe à l'heure actuelle de nombreuses machines multiprocesseurs à base de Transputers sur le marché. On y trouve des réalisations à configuration statique (à connectique fixe) et d'autres reconfigurables.

Toutefois, choisir une topologie d'interconnexion fixe signifie bâtir des systèmes plus ou moins bien adaptés à certains types d'applications. Pour avoir donc des machines capables de traiter toutes sortes d'algorithmes avec la même efficacité, mieux vaut ne pas relier directement les processeurs par des liens physiques inamovibles, mais par l'intermédiaire de commutateurs configurables par programme. La machine *Multi Cluster* de PARSYTEC est un exemple de réseau reconfigurable contenant 16 Transputers T800-20 disposant chacun d'une mémoire locale de 1 M octets.

Pour la réalisation d'une interconnexion reconfigurable INMOS propose le circuit IMS C004 (*Crossbar switch*) de 32 commutateurs permettant de relier d'une manière programmable 32 liens de Transputers.

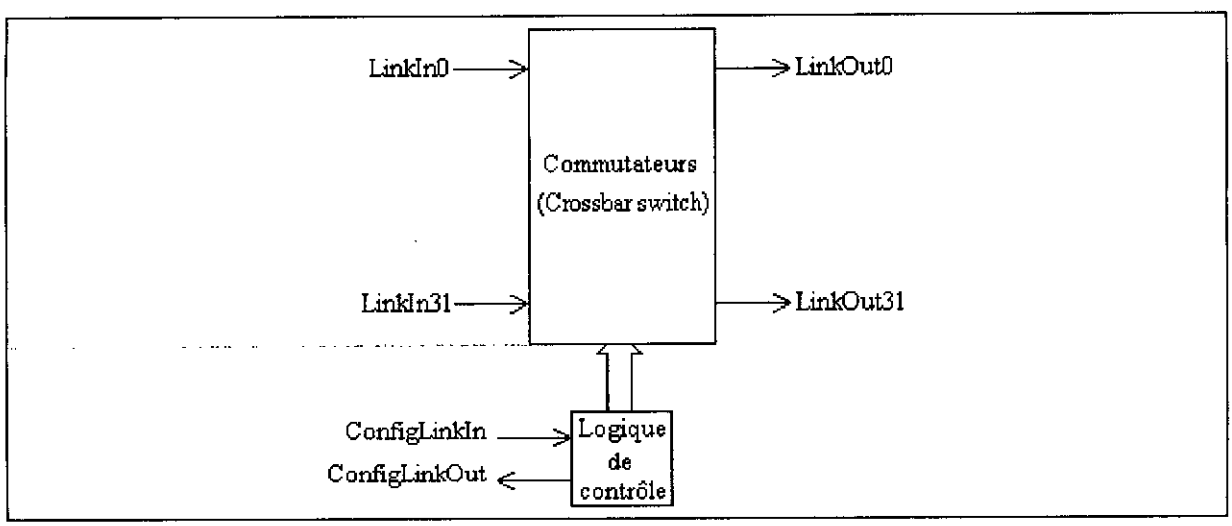


Fig. 1.6 Schéma bloc du IMS C004.

Le IMS C004 est constitué de 32 multiplexeurs 32 x 1. A chaque multiplexeur est associé un registre de six bits dont cinq permettent de sélectionner l'une des entrées comme source pour la sortie désirée et le sixième permet de connecter ou de déconnecter la sortie. Ces registres sont manipulés (en lecture et en écriture) grâce aux liens de configuration **configLinkOut** et **configLinkIn**. Les entrées et les sorties sont identifiées par un nombre compris entre 0 et 31. Le message de configuration consiste en un, deux ou trois octets réalisant les fonctions de la table 1.3 ci-dessous :

Message de configuration	**Fonction**
[0] [input] [output]	- Connecter **input** à **output**.
[1] [link1] [link2]	- Connecter l'entrée de **link1** à la sortie de **link2** et l'entrée de **link2** à la sortie de **link1**.
[2] [output]	- Demander quelle entrée est connectée à **output**.
[3]	- Message de fin de configuration.
[4]	- *Reset* : toutes les sorties sont déconnectées.
[5] [output]	- Déconnecter **output**.
[6] [link1] [link2]	- Déconnecter les sorties de **link1** et **link2**.

Table 1.3

Plusieurs circuits IMS C004 peuvent être combinés pour réaliser un réseau d'interconnexion de plus de 32 voies (taille supérieure à 32).

Un réseau reconfigurable de Transputers est illustré par la figure 1.7.

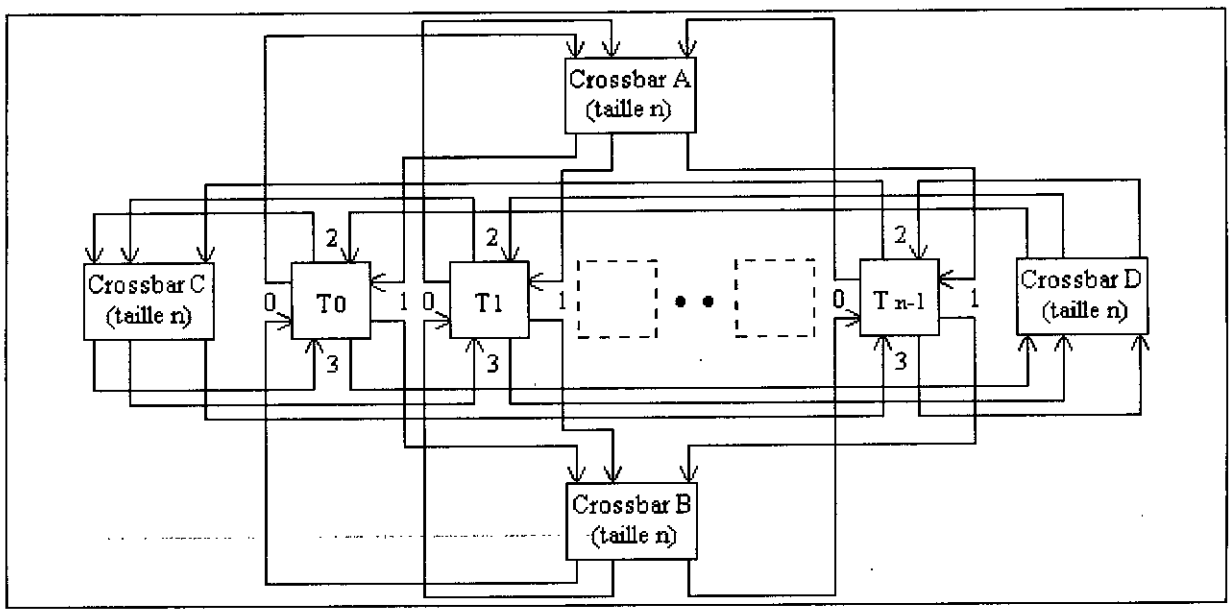


Fig. 1.7 Réseau reconfigurable de Transputers.

La carte à Transputers dont nous disposons est une *TMB 04* de TRANSTECH pour IBM PC XT ou AT et compatibles. Elle dispose d'un Transputer (IMS T800 20MHz) avec une mémoire locale de 4 M octets (extensible jusqu'à 16 M octets). Cette carte peut recevoir 4 TRAM (*TRAnsputer Modules* : Transputer et mémoire locale) reliés en pipeline; le lien 2 de chaque TRAM est connecté au lien 1 du TRAM suivant dans la chaîne. Le reste des liens est relié à un connecteur pour permettre une liaison avec une autre carte à Transputers [8].

1.6 Conclusion

L'étude menée dans ce chapitre a permis de décrire, d'une manière brève, les architectures de base des ordinateurs parallèles. On a pu aussi mettre le point sur les performances des Transputers et les facilités qu'ils offrent pour la conception de systèmes multiprocesseurs MIMD reconfigurables. Cet aspect de reconfigurabilité permet aux Transputers d'être adaptés à un grand nombre d'applications.

Il est à noter enfin que le Transputer est le premier processeur conçu pour exécuter avec efficacité des programmes écrits en Occam. Le Transputer peut également être programmé en utilisant d'autres langages. Toutefois c'est l'Occam qui offre le meilleur rendement, que ce soit en puissance de calcul ou en taille du code exécutable, comme le montre la table suivante [7] :

Langage	**Whetstones / seconde (T800-20)**	**Taille du code en K.Octets**
Occam2	3.47 Méga	5.3
PAR C 1.3	2.11 Méga	21.5
C 3L 1.3	2.06 Méga	71.6

Table 1.4

Chapitre II

LE LANGAGE DE PROGRAMMATION PARALLELE OCCAM

2.1 Le langage Occam

2.1.1 Introduction :

Les premiers ordinateurs étaient programmés grâce aux instructions de base du logiciel d'un processeur, c'est la programmation en langage machine. Une telle programmation est très difficile à utiliser; c'est pour cela que les spécialistes ont conçu des langages de programmation de haut niveau (Fortran, Basic, Pascal, C, Modula 2, Ada, Lisp, etc...).

Ces langages offrent plus de facilités, puisqu'ils permettent d'exprimer la logique d'un programme en une suite d'instructions utilisant des mots d'une langue quelconque (Anglais, Français, etc...). Un programme appelé compilateur, permet par la suite de translater ces notations en des instructions de base d'un processeur.

Pour la plupart de ces langages, le résultat de la compilation est une séquence d'instructions à exécuter une à la fois par le processeur. Ces langages sont donc destinés à des machines séquentielles.

Pour pouvoir exploiter les performances des ordinateurs parallèles, d'autres langages, dits parallèles ou concurrents, sont nécessaires. Ces langages permettent de décrire un programme comme un ensemble de tâches dont chacune est prise en charge par un élément de traitement. L'**Occam** fait partie de ces langages.

C'est un langage concurrent issu du modèle de programmation parallèle CSP (*Communicating Sequential Processes*) développé initialement à l'université d'Oxford.

L'Occam permet d'exprimer une application comme une collection de processus concurrents pouvant s'exécuter sur un ou plusieurs Transputers. Un processus étant une entité qui échange des informations avec son environnement, formé d'autres entités du même type, par l'intermédiaire de messages transitant à travers des canaux qui relient les processus deux à deux. Les canaux de communication entre processus Occam sont reproduits au niveau matériel par les liens des Transputers.

Chaque processus Occam a une existence dans le temps. Il commence à un instant précis, exécute un certain nombre d'opérations à une vitesse propre, puis se termine. Une opération peut consister en un ensemble d'instructions s'exécutant les unes après les autres comme dans les langages séquentiels classiques, ou en un ensemble de processus s'exécutant séquentiellement, en parallèle ou en exclusion mutuelle. Le canal est le seul élément de communication entre deux processus, l'un joue le rôle d'émetteur, l'autre celui de récepteur. C'est au cours d'un rendez-vous qu'ils se synchronisent et échangent un message. Dans ce qui suit, nous expliquerons l'essentiel des principes de la programmation en Occam [3][7][9].

2.1.2 Concepts fondamentaux :

a) Les processus élémentaires :

Tous les programmes Occam sont construits en utilisant trois processus élémentaires :

. **L'affectation :** Ce processus permet d'assigner une valeur à une variable. Le symbole de l'affectation en Occam est **:=**. L'instruction :

x := 2

permet d'affecter la valeur 2 à la variable x.

. L'émission : Permet d'envoyer un message sur un canal. Elle est représentée par le symbole **!**. Par exemple :

chan ! e

envoie la valeur de e sur le canal chan.

. La réception : Exprime l'attente d'un message sur un canal. Elle est représentée par le symbole **?**. Par exemple :

chan ? x

reçoit une valeur du canal chan et l'affecte à la variable x.

b) La communication :

La communication à travers un canal ne peut se faire que si les processus émetteur et récepteur sont tous les deux prêts. Si par exemple, dans un programme, une opération de réception (ou émission) est atteinte en premier, le processus devra se mettre en attente jusqu'à ce que le processus émetteur (ou récepteur) correspondant soit prêt. La communication est donc synchronisée. Un processus peut alors être non activable, s'il a terminé son exécution, ou bien s'il attend l'occurrence d'un événement.

c) Les structures :

Plusieurs processus élémentaires peuvent être combinés pour former des structures. Une structure est elle-même un processus qui peut être un composant d'une autre structure. Il existe quatre classes de structures :

. Structure séquentielle : Elle est représentée de la manière suivante :

```
SEQ
  P1
  P2
  P3
  ...
```

Les processus P1, P2, P3, ... sont exécutés l'un après l'autre. La structure s'arrête lorsque le dernier processus est terminé.

. Structure parallèle :

```
PAR
  P1
  P2
  P3
  ...
```

Les processus P1, P2, P3, ... sont exécutés en même temps et sont appelés des processus concurrents. La structure s'arrête lorsque tous les processus sont terminés.

. Structure conditionnelle :

```
IF
  condition 1
    P1
  condition 2
    P2
  ...
```

Indique que si la condition 1 est vraie, P1 sera exécuté; sinon si la condition 2 est vraie, P2 sera exécuté, et ainsi de suite. La structure s'arrête après que l'un des processus ait été exécuté.

. Structure alternée :

```
ALT
  input1
    P1
  input2
    P2
  input3
    P3
```

Attend que l'une des conditions input1, input2, input3, ... soit réalisée. Si par exemple la condition input2 est réalisée la première, alors c'est le processus p2 qui sera exécuté. Lorsque l'une des conditions est réalisée et son processus correspondant exécuté, la structure s'arrête.

L'Occam permet de définir des niveaux de priorité dans des structures parallèles ou alternées. **PRI PAR** et **PRI ALT** permettent d'attribuer la priorité la plus haute au processus noté en premier. Par exemple, dans la structure parallèle :

```
PRI PAR
  P1
  P2
  P3
```

l'ordre de priorité va de P1 à P3.

NOTE : Pour définir le début et la fin d'une structure, on utilise une indentation de deux espaces vers la droite, comptés depuis la position de la première lettre du mot réservé de cette structure.

d) Les déclarations et les types :

L'Occam exige une déclaration des variables, des constantes ainsi que des canaux de communication tout en spécifiant leurs types. Une déclaration peut se faire à n'importe quelle position dans le programme, mais ne sera reconnue que par le processus qui la contient. Il existe plusieurs types de données parmi lesquels on peut citer :

INT : type entier.
BYTE : entiers compris entre 0 et 255.
BOOL : type booléen.

Ainsi que **INT32**, **INT64**, **REAL32** et **REAL64** qui sont des types d'entiers et de réels de 32 et 64 bits respectivement.

```
REAL32  x , y :          -- x et y variables réelles sur 32 bits.
VAL INT year IS 365 :    -- déclaration de la constante year=365.
CHAN OF INT chan :       -- déclaration d'un canal d'entiers : chan.
[20] [20] INT mat :      -- mat est un tableau 20 x 20 d'entiers.
```

e) Les opérations arithmétiques et logiques :

L'Occam possède des opérations arithmétiques dont les opérandes sont des entiers ou des réels. Il permet aussi de réaliser des opérations logiques sur des variables du type booléen. Tous les opérateurs ont le même niveau de priorité, donc des parenthèses doivent être utilisées dans les expressions complexes pour déterminer l'ordre d'évaluation.

2.1.3 Les procédures et les fonctions :

a) Les procédures :

Une procédure débute par le mot réservé **PROC** et se termine par **:** sur la même colonne que le P de PROC. Exemple :

```
PROC exemple ( INT x , VAL INT y , z)
  SEQ
    y := z * z
    x := y + z
:
INT a , b , c :
  SEQ
    b := 2
    c := 5
    exemple (a , b , c)
```

Lorsqu'une variable est passée à une procédure comme paramètre actuel, son contenu ne peut changer que si le paramètre formel n'est pas précédé par **VAL** dans la déclaration de la procédure.

b) Les fonctions :

La forme générale d'une fonction est :

```
type FUNCTION ( paramètres formels )
  déclarations :
  VALOF
    corps de la fonction
    RESULT expression
:
```

Par exemple une fonction qui permet de déterminer le plus grand entre deux entiers peut s'écrire :

```
INT FUNCTION max ( VAL INT a , b )
  INT answer :
```

```
VALOF
  SEQ
    IF
      b > a
        answer := a
      TRUE
        answer := b
    RESULT answer
:
```

2.1.4 Les canaux et protocoles de communication :

Les canaux de communication sont les seuls éléments pouvant relier des processus Occam. Il est très important donc de pouvoir véhiculer à travers ces canaux des donnés de n'importe quel type ou des données de types différents. L'Occam présente un certain nombre de possibilités pour regrouper ensemble différents types de données sous la forme d'un **protocole** de communication. Ce protocole permet de spécifier la séquence de types de données qui peuvent être envoyées sur un canal. Chaque émission et réception sur un canal doit être compatible avec le protocole de celui-ci. Il existe plusieurs manières de définir un protocole :

a) Protocole simple : C'est le protocole ne contenant qu'un type de données :

```
PROTOCOLE word IS INT :
...
CHAN OF word comm :
```

Une communication sur le canal **comm** doit être compatible avec le protocole **word** (seuls des entiers peuvent être transmis à travers **comm**).

b) Protocole séquentiel : Ce protocole spécifie qu'une séquence de valeurs, de types différents, peut être transmise à travers un canal.

Par exemple, la définition de protocole et la déclaration de canal :

```
PROTOCOLE message IS BYTE; INT; INT:
...
CHAN OF message chan :
```

spécifient que le canal **chan** peut véhiculer seulement des messages formés d'un octet suivi par deux entiers. Par exemple :

```
chan ! 40(BYTE) ; 250 ; 505
```

c) Protocoles pour tableaux : Ce type de protocole permet de transmettre sur un canal des tableaux d'un type déterminé et de taille variable. Par exemple :

```
CHAN OF INT : : [ ] BYTE chan :
```

Sur le canal **chan** on peut transmettre des tableaux de bytes de taille entière variable :

```
CHAN OF INT : : [ ] BYTE chan :
[20] BYTE vec1 :
[40] BYTE vec2 :
SEQ
  chan ! 20 : : vec1
  chan ! 30 : : [ vec2 FROM 0 FOR 30 ]
```

d) Protocoles variants : Les protocoles séquentiels permettent de définir le format des messages qui peuvent être transmis entre processus Occam. Souvent, il est plus intéressant de transmettre des messages de formats différents sur le même canal. Ceci est possible grâce au protocole variant.

Un protocole variant est, en fait, un ensemble de protocoles différents dont chacun est identifié par un nom, et peut être utilisé pour la communication sur le canal.

```
PROTOCOL message
  CASE
    prt1 ; INT
    prt2 ; BYTE ; INT
    prt3 ; [20] REAL32
:
...
CHAN OF message comm :
```

Une émission sur le canal **comm** doit se faire en spécifiant le nom de l'un des trois protocoles :

```
comm ! prt1; 1
...
comm ! prt2; 100 ( BYTE ) ; 3
```

Pour une réception sur le canal **comm**, deux cas peuvent se présenter :

- La valeur à lire est connue : Dans ce cas le processus de réception peut s'écrire par exemple :

```
[20] REAL32 vec :
...
  comm ? CASE prt3 ; vec
```

- La valeur à lire est inconnue : Dans ce cas il faudrait prévoir une action pour chaque variante du protocole :

```
INT x , y :
BYTE b :
[20] REAL32 vec :
...
  comm ? CASE
    prt1 ; x
    prt2 ; b ; y
    prt3 ; vec
```

2.1.5 Les boucles :

En plus de la boucle *while* classique, l'Occam permet la duplication de processus avec les constructions **SEQ**, **PAR**, **IF** et **ALT**.

- Avec la structure **SEQ**, il nous est possible d'obtenir une boucle *for* conventionnelle. Par exemple, la structure séquentielle suivante :

```
SEQ i = 0 FOR n
  P
```

permet d'exécuter le processus P, **n** fois.

- Avec la structure **PAR**, il est possible d'exécuter un ensemble de processus concurrents similaires. Par exemple, la structure :

```
PAR i = 0 FOR n
  Pi
```

entraîne l'exécution des **n** processus similaires : P_0, P_1, ..., P_{n-1}.

2.1.6 La configuration :

Pour pouvoir exécuter un programme Occam sur un réseau de Transputers, une configuration de celui-ci est nécessaire. Elle consiste à :

- Associer à chaque canal Occam un lien de Transputer :

```
PLACE link0in AT 4 :
```

permet de placer le canal **link0in** sur le lien 4.

- Allouer un Transputer à chaque processus d'une structure parallèle, en remplaçant **PAR** par **PLACED PAR** suivi d'un placement, qui consiste en le numéro du processeur et le processus qu'il doit exécuter :

```
PLACED PAR
  PROCESSOR 0
    P0
  PROCESSOR 1
    P1
  ...
```

2.2 Le système de développement pour Transputers TDS d'INMOS

2.2.1 Introduction :

Le TDS est un environnement intégré conçu par INMOS pour permettre la programmation en Occam des réseaux de Transputers. Il comprend un éditeur intégré, un gestionnaire de fichiers, un compilateur et un debuggueur. Le TDS tourne sur des

cartes à Transputers connectées à un IBM PC/AT ou XT et compatibles; ceci permet d'interfacer le clavier, l'écran et les unités de stockage au Transputer.

Des programmes Occam peuvent être écrits, compilés et exécutés à partir du système de développement. Ces programmes peuvent être configurés pour être exécutés aussi bien sur un Transputer unique que sur un réseau de plusieurs centaines de Transputers [5].

2.2.2 L'éditeur :

En plus des fonctions classiques qu'elle intègre, l'interface d'édition du TDS est basée sur un concept très important appelé *folding* (mettre en plis). Cette opération permet de donner au texte en cours d'édition une structure hiérarchique (structure en *folds* ou plis) qui reflète la structure du programme. L'éditeur permet de cacher un bloc de lignes d'un document, ce bloc est appelé *fold* (pli).

L'exemple ci-dessous montre un programme dont certains blocs sont cachés. Un pli est marqué par trois points (...). L'éditeur permet d'écrire un commentaire sur la ligne du pli pour résumer son contenu. Ce contenu peut être visualisé soit en entrant à l'intérieur du pli (*enter fold*) soit en l'ouvrant (*open fold*). Les trois accolades ouvrantes marquent le début, alors que les trois accolades fermantes marquent la fin du pli.

Exemple 1 : Programme avec des plis fermés.

```
... Déclarations
SEQ
  ... initialisation
  WHILE go.on
      process ( ch , go.on )
```

Exemple 2 : Programme avec un pli ouvert.

```
... Déclarations
SEQ
  {{{ initialisation
  go.on := TRUE
  input ? ch
  }}}
  WHILE go.on
      process ( ch , go.on )
```

Chaque pli a une indentation qui lui est associée; la marque de début du pli commence au niveau de cette indentation. Aucun texte ne peut être inséré à gauche par rapport à la marque de début du pli. La structure en plis permet donc de modifier facilement l'indentation d'une partie d'un programme Occam.

En entrant à l'intérieur d'un pli, seul son contenu est affiché. Exemple 3 :

```
{{{ Déclarations
INT ch :
BOOL go.on
PROC process ( )
    ... corps de la procédure
:
}}}
```

L'éditeur permet d'écrire le contenu d'un pli dans un fichier (*file fold*). Celui-ci est ouvert chaque fois que l'on ouvre le pli, et il est mis à jour après fermeture du pli.

2.2.3 Le compilateur :

Des programmes Occam peuvent être compilés et exécutés sans devoir quitter l'environnement de développement du TDS. Pour ce faire les utilitaires de compilation peuvent être chargés dans la mémoire locale du Transputer maître. Ces utilitaires permettent de réaliser la compilation et l'édition de liens de programmes Occam, ainsi que la configuration et le chargement du code sur un réseau de Transputers.

Avant qu'un programme Occam puisse être compilé, deux opérations doivent être réalisées au préalable. La première consiste à écrire la source du programme dans un fichier, la seconde consiste à créer un **pli de compilation** (*make foldset*) auquel le compilateur sera appliqué.

2.2.4 Exemple d'exécution d'un programme Occam sous TDS :

Cet exemple permet de suivre les étapes nécessaires à l'exécution d'un programme Occam sous TDS. On a présenté après chaque étape les messages renvoyés par le TDS.

- Créer un pli (*create fold*) et lui donner le nom : **exemple**

```
...  exemple
```

- Entrer à l'intérieur du pli (*enter fold*) et écrire le texte du programme.

```
{{{  exemple
# USE userio
VAL  message  IS  " Bonjour ! "  :
INT  key  :
SEQ
     write.full.string (screen , message)
     read.char (keyboard , key)
}}}
```

- Sortir du pli exemple puis écrire son contenu dans un fichier (*file fold*).

```
Filed ok as exemple.tsr
...  F exemple
```

- Créer un pli de compilation contenant le pli du programme source (*make foldset*) avec le paramètre **EXE** pour créer un exécutable.

```
...  EXE  program
{{{  EXE  program
...  F exemple
}}}
```

- Se placer sur le pli **program** puis lancer la compilation (*compile*).

```
program linked ok
```

- Charger le code du programme en mémoire (*get code*).

```
code got ok
```

- Exécuter le programme (*run EXE*).

Le programme affiche simplement le message (Bonjour !) sur l'écran puis se met en attente d'une touche du clavier pour revenir à l'écran d'édition. Ce programme utilise la librairie d'entrées / sorties **userio** du TDS à laquelle appartiennent les deux procédures **write.full.string** et **read.char**. **screen** et **keyboard** sont des canaux prédéfinis.

L'exemple détaillé ci-dessous est le programme source en Occam permettant de réaliser le calcul d'histogramme ou le filtrage spatial d'une image de résolution 512 x 512. Le principe de son exécution reste le même que pour l'exemple précédent.

```
{{{ Déclarations
#USE  userio
[512][512] BYTE image, c.image:
[256] REAL32 hist:
TIMER clock:
INT s.time,e.time:
[3][3] REAL32 mask:
}}}

... procédure eval.time ( )

{{{ procédure histogram ( )
PROC histogram(VAL [ ][ ] BYTE h.mg, [256] REAL32 his)
   INT g :
   SEQ
      SEQ i = 0 FOR 256
         his[i] := 0.0 (REAL32)
      SEQ x = 0 FOR 512
         SEQ y = 0 FOR 512
            SEQ
               g := INT ( h.mg[x][y] )
               his[g] := his[g] + 1.0 (REAL32)
      SEQ i = 0 FOR 256
         his[i] := ( his[i]  /  262144.0 (REAL32) )
:--histogram
}}}
{{{ procédure filter ( )
PROC filter( )
   REAL32  value :
   SEQ --filter procedure
     SEQ i = 1 FOR 510
       SEQ j = 1 FOR 510
         SEQ
           value:=REAL32((REAL32 ROUND ( INT image[i-1][j-1]))*mask[0][0])
           value:=value+ ((REAL32 ROUND (INT image[i-1][j]))*mask[0][1])
           value:=value+ ((REAL32 ROUND (INT image[i-1][j+1]))*mask[0][2])
           value:=value+ ((REAL32 ROUND (INT image[i][j-1]))*mask[1][0])
           value:=value+ ((REAL32 ROUND (INT image[i][j]))*mask[1][1])
```

```
                value:=value+ ((REAL32 ROUND (INT image[i][j+1]))*mask[1][2])
                value:=value+ ((REAL32 ROUND (INT image[i+1][j-1]))*mask[2][0])
                value:=value+ ((REAL32 ROUND (INT image[i+1][j]))*mask[2][1])
                value:=value+ ((REAL32 ROUND (INT image[i+1][j+1]))*mask[2][2])
                c.image[i][j] := BYTE ( ( INT32 ROUND value) )
:--filter
}}}

{{{ procédure monitor.          Programme principal
PROC monitor( )
   INT key.char :
   BOOL activ :
   SEQ
      write.text.line(screen,"                ** MAIN MENU **   ")
      newline(screen)
      write.text.line(screen,"        h : histogram          f : s.Filter ")
      write.text.line(screen,"        t : time               q : quit ")
      activ:=TRUE
      WHILE  activ
         SEQ
            read.char (keyboard , key.char)
            CASE key.char
               INT ' q '
                  activ := FALSE
               INT ' h '
                  histogram (image , hist)
               INT ' f '
                  filter ( )
               INT ' t '
                  eval.time ( )
               ELSE
                  beep (screen)
:--monitor
}}}

monitor ( )          -- exécution du programme principal.
```

Chapitre III

LES TRAITEMENTS D'IMAGES

3.1 Introduction

La multitude d'informations contenues dans une image fait de celle-ci un support privilégié dans de nombreux domaines scientifiques :

- Recherche biomédicale : Radiographie X ou γ, microscopie optique ou électronique, thermographie, etc.
- Recherche spatiale : télescopes, images de planètes, télédétection.
- Robotique : vision robotique, détection et localisation d'objets, vision tridimensionnelle.
- Audiovisuel : Simulation, conception assistée par ordinateur, animation d'images.

Le développement de l'acquisition et du traitement d'images a été longtemps ralenti du fait de l'absence de capteurs performants d'une part et de l'absence de grandes capacités mémoires d'une autre part. Les traitements d'images, du fait de leur coût élevé, étaient réservés à des applications stratégiques : recherche spatiale et applications militaires.

Le développement de capteurs simples et fiables et l'augmentation des performances des équipements digitaux (vitesse de calcul et capacité de mémorisation) permettent d'imaginer et de concevoir des systèmes de traitement d'images de plus en plus performants, tant au niveau des applications industrielles que dans la recherche appliquée, en liaison avec des problèmes de reconnaissance de formes et d'intelligence artificielle [10][11].

3.2 Les différentes disciplines de traitement d'images

Les techniques de traitement d'images sont destinées à l'exploitation des informations contenues dans une image. Le but recherché est l'amélioration de la qualité visuelle de l'image, ou la mise en évidence de certaines informations particulières pour l'application envisagée.

Les problèmes rencontrés en traitement d'images sont abordés de différentes manières selon le type des images, leur origine et le but précis recherché.

Parmi les principales disciplines, nous pouvons distinguer :

- La restauration d'images.
- L'amélioration d'images.
- Le codage et la compression d'images.
- L'analyse descriptive des images.
- La synthèse d'images.

3.2.1 La restauration d'images :

La restauration d'images [10][11][12] consiste en la reconstitution d'une image ayant subi des dégradations plus ou moins importantes lors de son processus de formation par différents facteurs qui peuvent être :

- Déformations géométriques.
- Flous divers.
- Turbulences atmosphériques.
- Défauts dûs au système de prise de vue.

Les techniques de restauration d'images nécessitent la connaissance du phénomène de dégradation. Cette connaissance peut provenir soit d'un modèle

analytique ou statistique, en relation avec le phénomène physique lié au capteur, soit d'informations à priori en relation avec la structure de l'image.

Ces techniques, fondées sur les méthodes classiques du traitement du signal, consistent en l'application de fonctions de transfert sur les images, ou en des traitements supprimant des défauts dûs à des parasites divers.

La figure 3.1 montre le processus de dégradation modélisé par un système (ou opérateur) **H**, qui avec un bruit additif **n(x,y)**, opère sur une image d'entrée **f(x,y)** pour produire l'image dégradée **g(x,y)**.

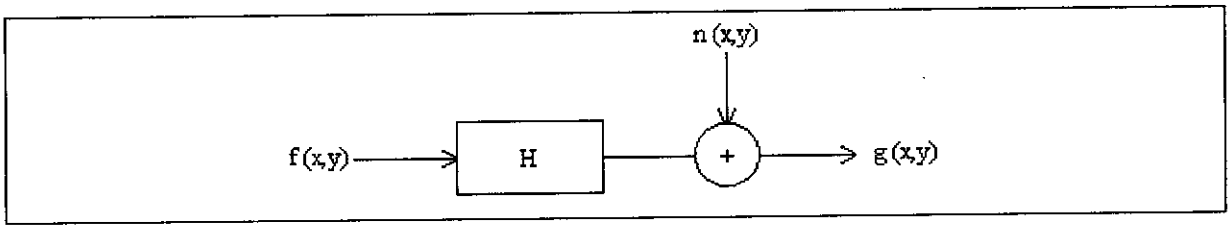


Fig. 3.1 Modèle de dégradation d'une image.

Le problème de restauration d'images peut être vu comme l'obtention d'une approximation de **f(x,y)**, étant donnée **g(x,y)** et une connaissance sur la dégradation (opérateur **H**). La connaissance sur **n(x,y)** est limitée à une information de nature statistique.

$$g(x,y) = f(x,y) * h(x,y) + n(x,y) \qquad 3.1$$

g(x,y) est le résultat du produit de convolution de **f(x,y)** et **h(x,y)** additionné au terme **n(x,y)** représentant le bruit. L'expression 3.1 s'écrit dans le domaine fréquentiel :

$$G = F \,.\, H + N \qquad 3.2$$

où G, F, H et N sont les transformées de Fourrier respectives de g, f, h et n.

f(x,y) pourra donc être obtenue par l'une des deux expressions de transformée de Fourrier inverse :

- En absence du bruit additif :

$$f(x,y) = TF^{-1} \{ G / H \} \qquad 3.3$$

- En présence de bruit :

$$f(x,y) = TF^{-1} \{ (G - N) / H \} \qquad 3.4$$

3.2.2 Le rehaussement d'images :

L'amélioration d'images [10][11][12], quant à elle, est une discipline étroitement liée à la subjectivité de l'observateur. Elle consiste à fournir une image, parfois différente de l'image initiale, permettant la mise en évidence, pour l'oeil de l'observateur, de certains détails ou traits caractéristiques peu ou pas apparents.

Les principaux traitements liés au rehaussement d'images peuvent être :

- La modification d'intensité des points de l'image.
- Le renforcement du contraste.
- La réduction du bruit.

Ces traitements consistent en des filtrages, élimination ou atténuation de certaines données en faveur des informations utiles.

3.2.3 Le codage et la compression d'images :

Il peut arriver que les moyens de stockage ou les canaux de transmission disponibles ne permettent pas de mémoriser (capacité mémoire insuffisante) ou de transmettre (bande passante étroite) l'intégralité des informations contenues dans une image.

Dans ce cas, il est utile de chercher à réduire le nombre de bits nécessaire à la représentation de l'image par un codage approprié, c'est à dire tenant compte de la structure de l'image, de ses propriétés statistiques et de sa connaissance à priori.

Les transformations orthogonales (Fourrier, Haar, Hadamard, Karhunen Loeve, etc.) [10][13] sont particulièrement adaptées pour ce type de codage. Elles utilisent les corrélations importantes qui existent sur un support visuel (continuité, dépendance des échantillons adjacents) en éliminant une partie des informations qui ne sont pas nécessaires.

3.2.4 L'analyse descriptive des images :

Le besoin d'analyser un grand nombre d'images a entraîné l'apparition de systèmes automatiques de détection et d'utilisation d'informations complexes contenues dans les images.

Cette analyse d'images a des buts multiples, notamment :

- Les mesures statistiques.
- Les opérations de comptage.
- La classification (objets, zones, ...etc.).
- La reconnaissance de formes.

L'analyse descriptive en tant que discipline, fait appel à diverses techniques associant le traitement d'images (extraction de contours) à l'intelligence artificielle (description structurée de l'état d'une scène).

Ces techniques d'analyse et de reconnaissance de formes sont très utilisées dans des domaines d'application parfois très différents tels que la physique, la recherche biomédicale, la robotique, la reconnaissance de caractères, etc.

3.2.5 La synthèse d'images :

Dans cette discipline, il s'agit de la génération d'images par calculateur à partir d'informations contenues dans la mémoire de celui-ci et constituée par des formes élémentaires spécifiques à l'application envisagée.

Les systèmes incluants des techniques de synthèse d'images apparaissent dans des domaines très divers :

- Domaine audiovisuel : films, génériques, spots publicitaires, enseignement, ...etc.
- Domaine de la conception assistée par ordinateur (CAO) : jeux, architecture, ...etc.
- Domaine de la simulation : identification d'attitude d'objets en mouvement, fonctionnement de machines, ...etc.

3.3 Opérations de traitement d'images

Les opérations utilisées en traitements d'images sont complexes mais peuvent se décomposer en des opérations élémentaires. Les différents traitements qui peuvent être effectués sur une image sont regroupés dans les catégories suivantes :

- Manipulations ponctuelles d'intensité : Ces manipulations font appel généralement à des opérations non linéaires qui consistent à remplacer l'intensité g d'un point de l'image par f(g), où f est une fonction ne dépendant que de g.

- Opérations localisées : Dans une opération localisée, la détermination de la nouvelle valeur du niveau de gris d'un point est fonction des niveaux de gris des points voisins. Ces points sont contenus dans une fenêtre rectangulaire de dimensions m x n. L'opérateur est un masque de mêmes dimensions. Ce type d'opérations est appelé convolution discrète.

- Transformations globales : Dans une opération globale, chaque élément de l'image résultat dépend de la totalité des éléments de l'image originale.

- Traitements géométriques : Ces traitements consistent à modifier l'image en déplaçant ses points constitutifs.

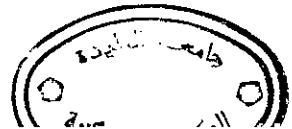
- Paramètres et statistiques : Ces fonctions permettent d'avoir des informations sur l'image ou d'établir des statistiques. Les résultats obtenus permettent le calcul de paramètres intervenant dans des traitements ultérieurs.

- Traitements multi-images : Ces traitements nécessitent l'accès simultané à plusieurs images en vue d'une comparaison (mise en évidence d'une évolution dans le temps) ou d'une moyenne sur plusieurs images identiques (amélioration du rapport signal sur bruit).

Dans ce qui suit, nous allons détailler quelques traitements d'images fondamentaux.

3.3.1 Manipulations de l'histogramme de l'image :

Le contenu en niveau de gris d'une image peut être facilement manipulé pour améliorer son interprétation visuelle. Parmi ces manipulations, on peut citer le seuillage, l'égalisation et la spécification d'histogramme.



a) Calcul d'histogramme :

L'histogramme d'une image est une fonction de N dans N (N ensemble des entiers naturels), qui à toute valeur ou mesure dans l'image, associe son nombre d'occurrence. L'histogramme résume au mieux l'information globale que contient l'image et donne donc un aperçu sur sa structure statistique. La forme de l'histogramme est significative de la forme de la distribution des niveaux de gris dans l'image. Elle donne des indications précises sur le caractère et la nature de l'image dont elle est issue. Par exemple si l'histogramme est étroit, l'image a un faible contraste lumineux. La figure 3.2 montre une image dont l'histogramme est représenté sur la figure 3.4 [2][14].

b) Egalisation d'histogramme :

L'égalisation permet de redistribuer les valeurs des niveaux de gris des pixels d'une image de telle sorte à avoir le même nombre de pixels pour chaque niveau de gris. En d'autres termes, l'histogramme égalisé d'une image tend vers un histogramme uniforme [12][14][17].

Soit **hi** l'histogramme non égalisé, le but est de trouver une transformation **T** qui, appliquée à **hi**, donne un histogramme uniforme **hu**.

La transformation qui permet d'égaliser l'histogramme est donnée par :

$$gi = T(fi) = (m-1)\sum_{j=0}^{i}\frac{nj}{n} \qquad 3.5$$

gi : i^{eme} niveau de gris de l'image égalisée.
fi : i^{eme} niveau de gris de l'image originale.
m : nombre total de niveaux de gris.
nj : nombre d'occurrences du j^{eme} niveau de gris.
n : nombre total de pixels dans l'image.

Pour égaliser donc une image, la première étape consiste en le calcul de l'histogramme, puis par la suite la détermination des niveaux de gris **gi** par l'équation 3.5, et enfin, affecter à chaque pixel de l'image sa nouvelle valeur de niveau de gris.

Les figures 3.2 et 3.3 montrent l'image d'origine et l'image égalisée, les figures 3.4 et 3.5 montrent leurs histogrammes respectifs.

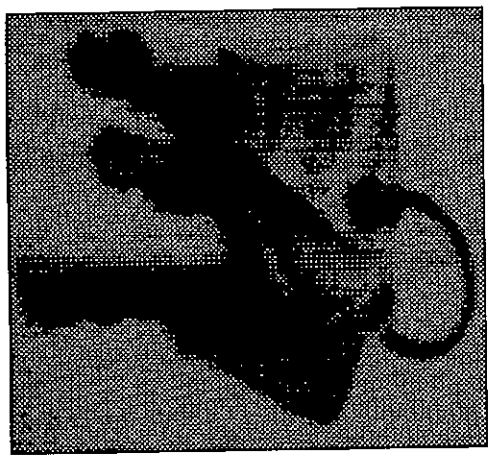


Fig. 3.2 Image originale sombre.

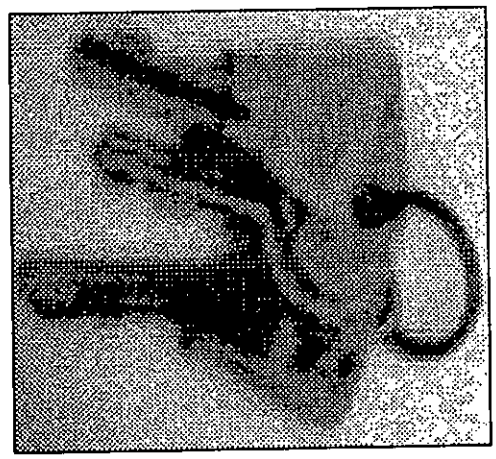


Fig. 3.3 Image après égalisation d'histogramme.

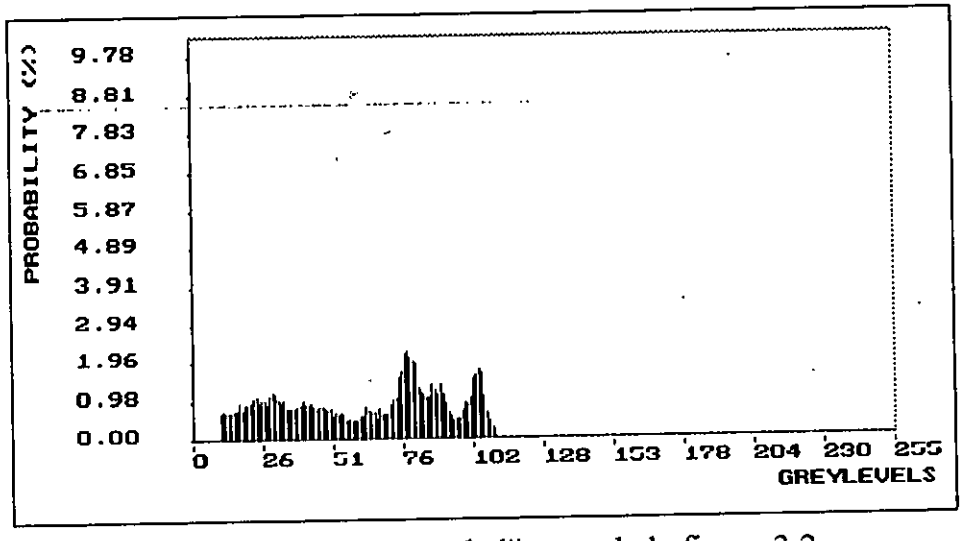


Fig. 3.4 Histogramme de l'image de la figure 3.2.

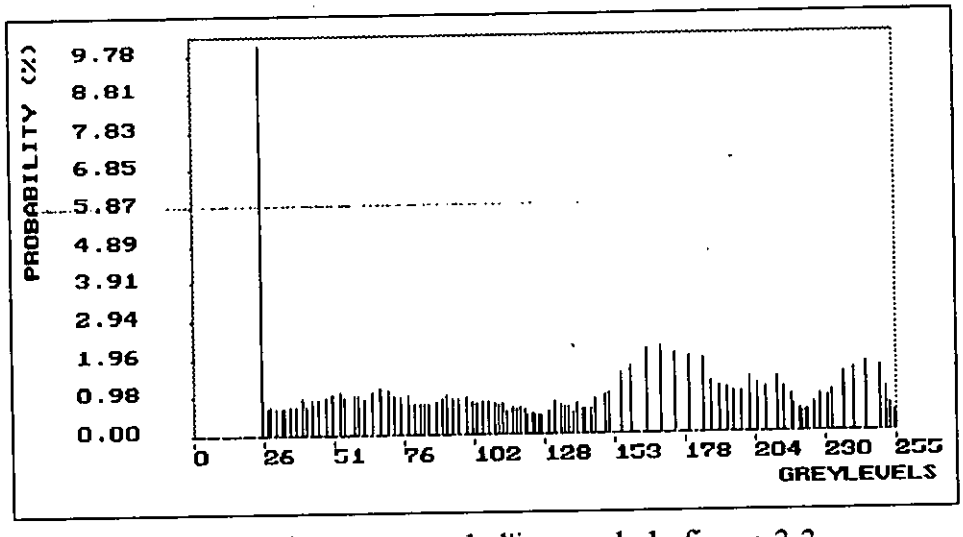


Fig. 3.5 Histogramme de l'image de la figure 3.3.

c) Le seuillage :

Dans de nombreuses applications, il est nécessaire de segmenter les niveaux de gris d'une image dans des régions particulières. Une binarisation, par exemple, permet de passer d'une image à 256 niveaux de gris à une image à 2 niveaux de gris [14][16].

La figure 3.7 est l'image binaire correspondant à l'image de la figure 3.6. Le seuillage à 2 niveaux de gris d'une image f(x,y) peut être décrit par la fonction de discrimination suivante :

$$g(x,y)=\begin{cases} 0 & si \quad f(x,y)<S \\ 255 & si \quad f(x,y)\geq S \end{cases} \qquad 3.6$$

où S est le seuil de discrimination.

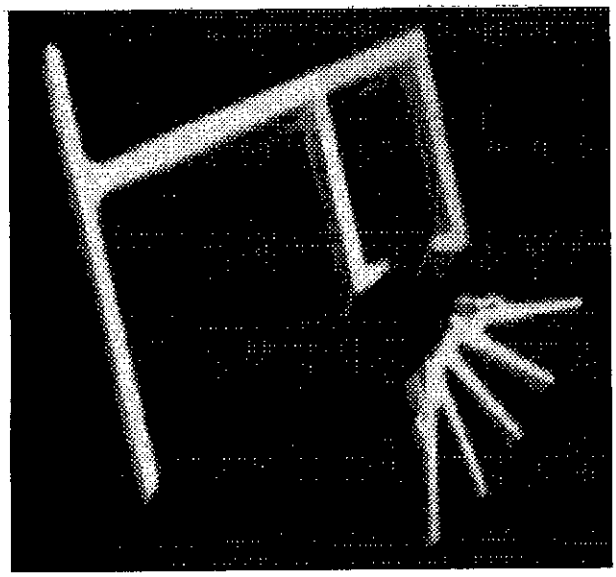


Fig. 3.6 Image originale.

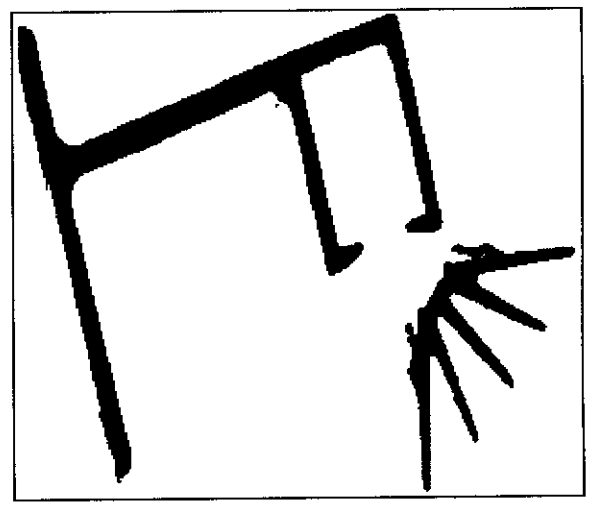


Fig. 3.7 Image binaire correspondant à l'image de la figure 3.6.

3.3.2 Le filtrage spatial :

Le filtrage spatial [12][14][15][16] consiste en la convolution discrète d'une image avec une autre (représentant le filtre). Le filtre est une image de taille très réduite, il est appelé masque de convolution. Le filtrage permet d'extraire certaines informations contenues dans l'image, d'éliminer du bruit ou de diminuer les effets des distorsions.

Un masque 3x3, par exemple, s'écrit :

$$M = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \qquad 3.7$$

A un point f(x,y) de l'image originale correspond le point g(x,y) de l'image filtrée donnée par :

$$g(x,y) = \sum_{r=-1}^{1}\sum_{s=-1}^{1} f(x+r, y+s).M(r,s) \qquad 3.8$$

a) Filtres détecteurs de fronts et de contours :

La détection de contours constitue une étape très importante dans tout processus d'analyse d'images. Ces filtres sont des filtres passe-haut (opérateurs différentiels), car les fronts et les contours sont constitués par des transitions brusques des niveaux de gris dans l'image, donc compris dans les hautes fréquences. Ces filtres calculent soit le gradient soit le Laplacien de l'image.

Le gradient est un opérateur qui peut être défini en chaque point f(x,y) de l'image par les deux composantes :

$\frac{\delta f(x,y)}{\delta x}$ approximée par $\Delta f_x = f(x,y) - f(x-1,y)$ et
$\frac{\delta f(x,y)}{\delta y}$ approximée par $\Delta f_y = f(x,y) - f(x,y-1)$.

Ces approximations sont très sensibles au bruit. Une des façons pour éviter ce problème consiste à étendre le voisinage du point considéré. Par exemple, on remplace f(x,y) - f(x-1,y) par

$$\frac{1}{\alpha}\left[\sum_{n=1}^{\alpha} f(x+n,y) - \sum_{n=1}^{\alpha} f(x-n,y)\right],$$

où α est un entier quelconque.

Différents opérateurs ont été développés faisant intervenir des voisinages et des coefficients divers. Sobel propose par exemple le masque

$$\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix},$$

qui permet de détecter les fronts verticaux dont la transition des niveaux de gris se fait de gauche à droite dans l'image. En permutant les colonnes 1 et 3 du masque, les fronts dont la transition se fait de droite à gauche sont détectés. Le gradient est un opérateur directionnel [16].

Le Laplacien est un opérateur différentiel non directionnel utilisé pour la détection de contours, il est donné par l'expression :

$$\nabla^2 = \frac{\delta^2 f(x,y)}{\delta x^2} + \frac{\delta^2 f(x,y)}{\delta y^2} \qquad 3.9$$

Le Laplacien en un point f(x,y) est approximé par :

$$L = f(x,y) - \frac{1}{4}[f(x,y+1) + f(x,y-1) + f(x+1,y) + f(x-1,y)] \qquad 3.10$$

D'où le masque 3 x3 :

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

La figure 3.9 montre le résultat de la détection de contours en utilisant le Laplacien sur l'image de la figure 3.8. La figure 3.10 est le résultat de l'application de l'opérateur vertical gauche de Sobel.

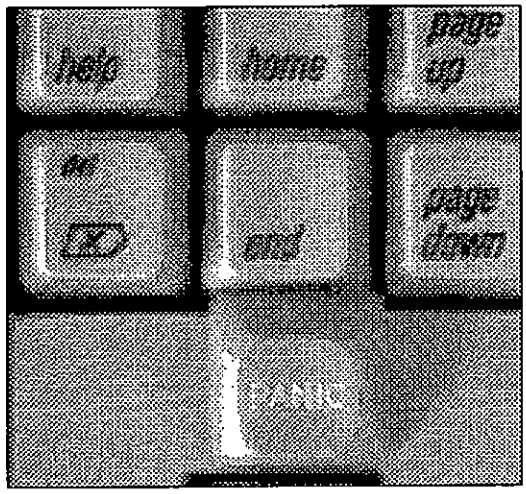


Fig. 3.8 Image originale.

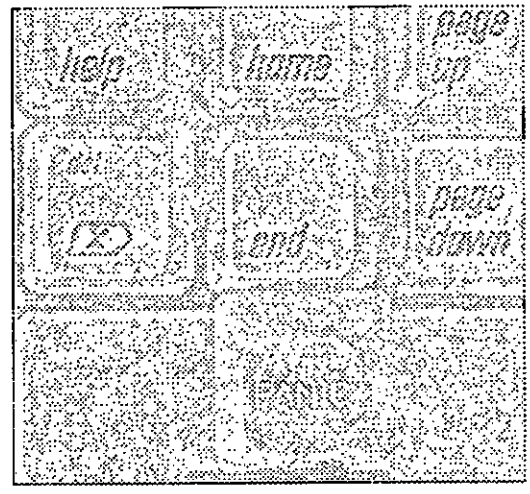


Fig. 3.9 Résultat de l'application du Laplacien à l'image de la figure 3.8

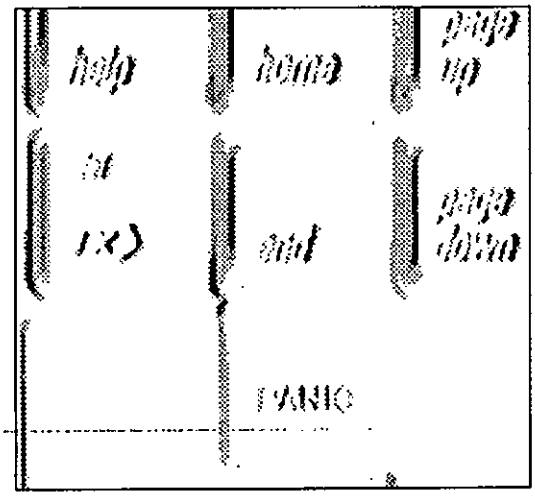


Fig. 3.10 Résultat de l'application du filtre de Sobel vertical gauche.

b) Filtres moyen et médian :

Le filtre moyen remplace la valeur du niveau de gris d'un pixel par la valeur moyenne des niveaux de gris des pixels de son voisinage. Ce filtre permet d'éliminer du bruit de fond, mais a tendance à adoucir les fronts. L'image de la figure 3.11 représente le résultat du filtrage moyen appliqué à l'image de la figure 3.3. On remarque l'élimination du bruit de fond, et l'apparition d'un effet de flou sur l'image.

Le filtre médian remplace la valeur du niveau de gris d'un pixel par la valeur médiane dans son voisinage. Il permet d'éliminer des points isolés dans l'image (bruit impulsionnel). La figure 3.13 montre une image après addition d'un bruit "*salt and pepper*" de probabilité 0.02% sur l'image originale de la figure 3.12. ce bruit est éliminé par un filtrage médian dont le résultat est illustré par la figure 3.14.

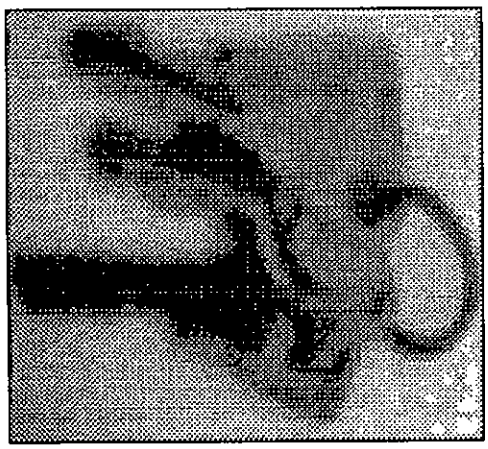


Fig. 3.11 Résultat du filtrage moyen.

Fig. 3.12 Image originale.

Fig. 3.13 Image additionnée à un bruit "salt and pepper" de probabilité 0.02%.

Fig. 3.14 Image après filtrage médian de l'image de la figure 3.13.

c) Filtres de rehaussement :

Ces filtres sont de deux types : filtres passe-bas et filtres passe-haut. Les premiers, comme le filtre moyen, permettent de réaliser un lissage de l'image pour éliminer le bruit, mais ont tendance à rendre l'image floue. Les seconds n'éliminent pas le bruit, bien au contraire, mais permettent l'amélioration du contraste et la mise en évidence des contours. La figure 3.15 est le résultat du filtrage passe-haut appliqué à l'image de la figure 3.8.

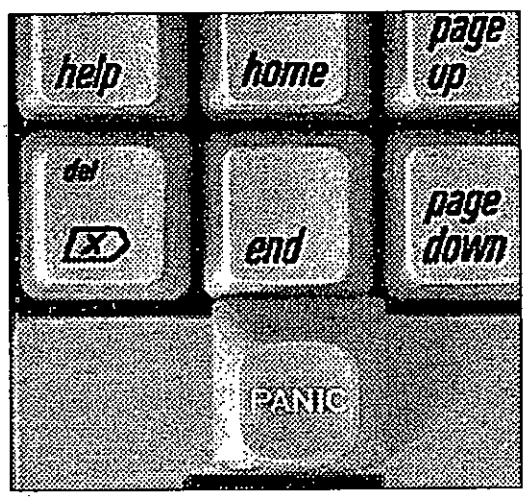


Fig. 3.15 Résultat du filtre passe-haut appliqué à l'image de la figure 3.8

3.3.3 Filtrage morphologique :

Une autre approche pour le filtrage en traitement d'images consiste à considérer l'image non pas comme une fonction bi-dimensionnelle mais plutôt comme un espace de représentation d'objets ayant des propriétés ensemblistes. Cette approche, appelée **morphologie mathématique**, permet, en définissant les environnements mathématiques appropriés, de conserver les propriétés algébriques, géométriques, topologiques et ensemblistes des objets.

La morphologie mathématique est née, vers les années 60, sous l'impulsion de G. Matheron, au centre de géostatistique et de morphologie mathématique de l'Ecole des Mines de Paris.

Le terme **morphologie mathématique** est apparu, quant à lui, pour la première fois en 1967, dans l'article de A. Haas, G. Matheron et J. Serra, intitulé : "Morphologie mathématique et granulométrie en place".

La morphologie mathématique est une description ensembliste de l'image, elle considère celle-ci comme un ensemble d'objets ou de sous-ensembles définis dans un espace donné. Elle opère sur ces objets en manipulant leur propriétés géométriques, dans le but de lisser les contours et de décomposer l'image en ses formes géométriques fondamentales [2][14][15][16][18].

Le filtrage morphologique est différent selon qu'il s'agisse d'une image binaire ou d'une image en niveaux de gris. Une image binaire peut être représentée comme un ensemble de points dans l'espace qui peuvent prendre l'état actif ou l'état inactif. Un point actif appartient à un objet de l'image, tandis qu'un point inactif appartient au fond.

Des images en niveaux de gris sont souvent représentées comme des images binaires dans un espace tridimensionnel, avec la troisième dimension représentant la luminance des points. De cette manière, les mêmes concepts d'analyse morphologique développés pour des images binaires peuvent être utilisés pour des images en niveaux de gris [19].

Il existe quatre types d'opérations de base du filtrage morphologique d'images binaires : l'érosion, la dilatation, l'ouverture et la fermeture.

Chacune de ces opérations utilise un masque dit **élément structurant** pour déterminer le processus de filtrage géométrique. Il existe deux autres opérations morphologiques importantes qui sont la détection de contour et la squelettisation.

a) 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 peut avoir la forme d'un cercle ou d'un segment.

b) L'érosion :

L'érosion est l'opération qui permet de réduire la taille géométrique d'un objet. Pour définir cette opération, nous allons nous situer dans un espace $\mathbf{R}^2$ partiellement occupé par un ensemble **A**. Prenons un élément structurant **B** représentant une figure géométrique simple, par exemple un cercle. Cet élément **Ba** est repéré par son centre placé en un point **a** dans l'espace $\mathbf{R}^2$. On appelle érodé de **A** par l'élément structurant **B** l'ensemble noté par $\mathbf{A\Theta B}$:

$$A\Theta B = \{a / Ba \subset A\} \qquad 3.11$$

La figure 3.16 montre l'érosion d'un objet rectangulaire avec, dans le premier cas un bipoint (segment) comme élément structurant, et un cercle dans le second cas.

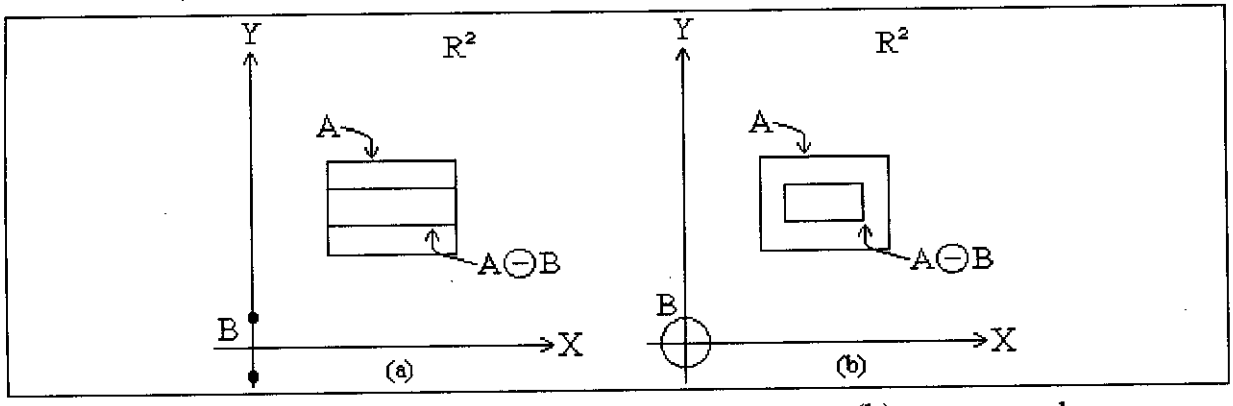


Fig. 3.16 Opération d'érosion : (a) par un segment, (b) par un cercle.

c) La dilatation :

La dilatation est l'opération duale de l'opération d'érosion; elle s'écrit :

$$A \oplus B = (A^C \Theta B)^C \qquad 3.12$$

où A^C est le complémentaire de A dans R^2.

La dilatation est basée sur l'addition ensembliste de Minkowski, elle permet d'élargir la taille géométrique d'un objet. Le dilaté d'un ensemble **A** de $\mathbf{R}^2$ par un élément structurant **B** s'exprime par :

$$A \oplus B = \{a / Ba \cap A \neq \varnothing\} \qquad 3.13$$

La figure 3.17 montre la dilatation d'un objet rectangulaire par un bipoint et un cercle.

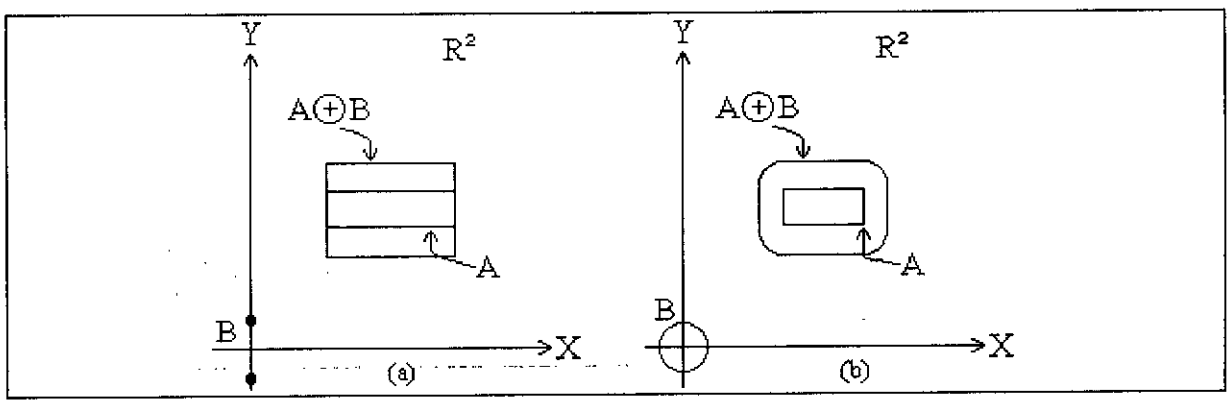


Fig. 3.17 Opération de dilatation : (a) par un segment, (b) par un cercle.

d) L'ouverture et la fermeture :

Ces deux opérations morphologiques sont utilisées pour le filtrage géométrique des contours d'objets dans une image.

Une opération d'ouverture consiste en une érosion suivie d'une dilatation.

$$O(A,B) = (A \Theta B) \oplus B \qquad 3.14$$

L'ouverture a tendance à supprimer les détails non significatifs de la forme en enlevant les isthmes et les caps, et à arrondir la forme (figure 3.18).

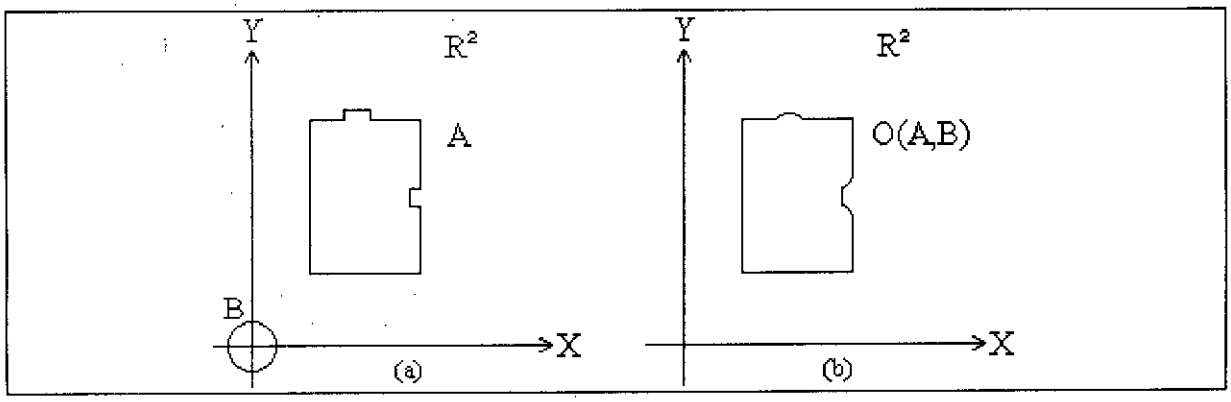


Fig. 3.18 Ouverture morphologique:(a) ensemble A original, (b) ouverture de A par B.

La fermeture est l'opération duale de l'ouverture; c'est une dilatation suivie d'une érosion.

$$F(A,B)=(A\oplus B)\Theta B \qquad 3.15$$

Elle a pour effets d'élargir les isthmes, de colmater les baies étroites, de boucher les trous et les détroits, en arrondissant les formes (figure 3.19).

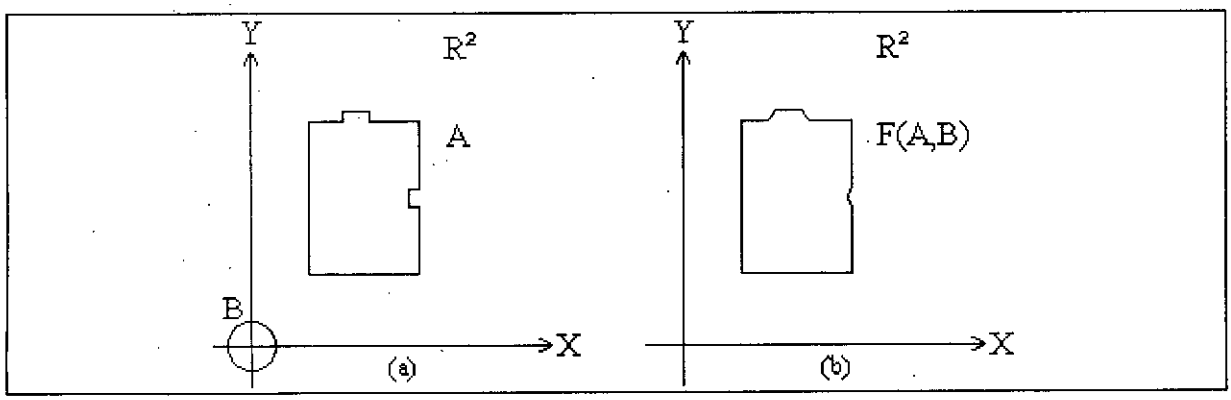


Fig. 3.19 Fermeture morphologique:(a) ensemble A original, (b) ouverture de A par B.

e) La détection de contours morphologique :

Il existe deux détecteurs de contours morphologiques, ils permettent d'extraire soit les contours externes soit les contours internes des objets. L'avantage de la détection de contours morphologique est qu'elle est moins sensible au bruit par rapport à la détection de contours par les opérateurs différentiels.

Considérons un objet **A** et un élément structurant **B**, le contour interne **Ci** et le contour externe **Ce** sont donnés par :

$$Ci = A - (A \ominus B) \quad 3.16$$

$$Ce = (A \oplus B) - A \quad 3.17$$

La largeur du contour obtenu dépend de la taille de l'élément structurant. Un élément structurant carré **m** x **m** (m ≥ 3 et impair) permet d'avoir une largeur de contour égale à **(m-1)/2**. La figure 3.20 illustre la détection de contours de l'image de la figure 3.7.

f) La squelettisation :

Pour analyser la structure des objets d'une image, il est souvent nécessaire de donner à ces objets une représentation compacte approximative. En effet, l'identification d'un objet en utilisant tous ses points constituants est très complexe. Pour remédier à ce problème, on utilise une transformation qui permet d'obtenir le squelette de l'objet. La squelettisation permet de générer une représentation en lignes découlant de la forme générale de l'objet.

Cette transformation, appelée transformation de l'axe médian (*medial axe trannsform*), consiste en l'élimination successive de points externes de l'objet jusqu'à l'obtention du squelette [14][15][16][20].

Pour réaliser cette opération, les dix (10) masques de la table 3.1 sont définis [20]. Les masques (a) - (d) permettent d'éliminer les points du contour dans les directions nord, ouest, sud et est respectivement. Alors que les masques (e) - (h) éliminent les points dans les directions nord-est, nord-ouest, sud-ouest et sud-est respectivement. Les deux autres masques (i) et (j) sont utilisés pour préserver la connexité du squelette. La figure 3.21 représente les squelettes des objets de l'image de la figure 3.7.

0	0	0
1	**1**	1
x	1	x

(a)

0	1	x
0	**1**	1
0	1	x

(b)

x	1	x
1	**1**	1
0	0	0

(c)

x	1	0
1	**1**	0
x	1	0

(d)

x	0	0
1	**1**	0
x	1	x

(e)

0	0	x
0	**1**	1
x	1	x

(f)

x	1	x
0	**1**	1
0	0	x

(g)

x	1	x
1	**1**	0
x	0	0

(h)

x	x	x	x
0	1	1	0
x	x	x	x

(i)

x	0	x
x	1	x
x	1	x
x	0	x

(j)

x = état indifférent

Table 3.1

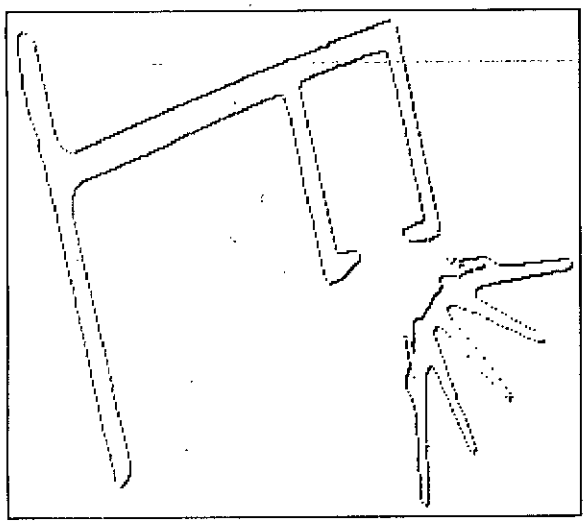


Fig. 3.20 Détection de contours morphologique appliquée à l'image 3.7.

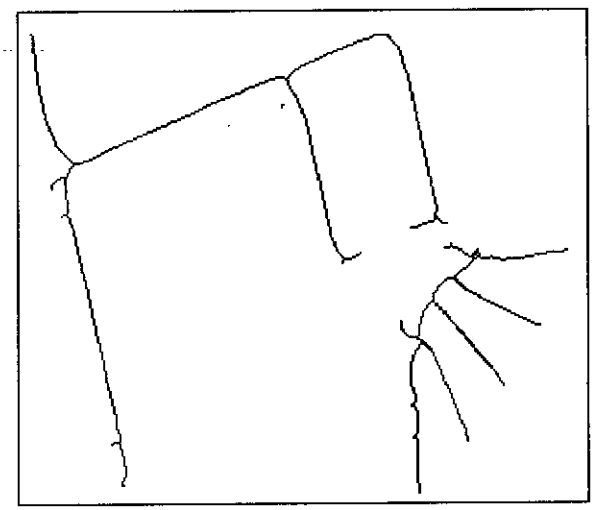


Fig. 3.21 Squelettes des objets de l'image 3.7.

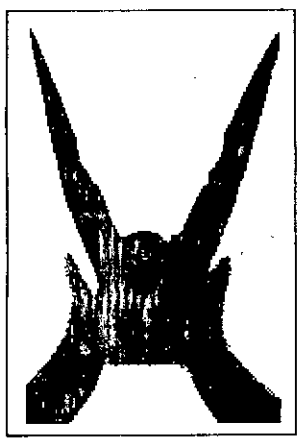


Fig. 3.22.

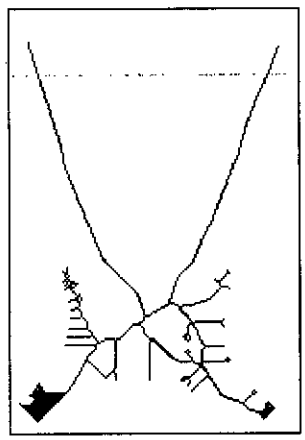


Fig. 3.23.

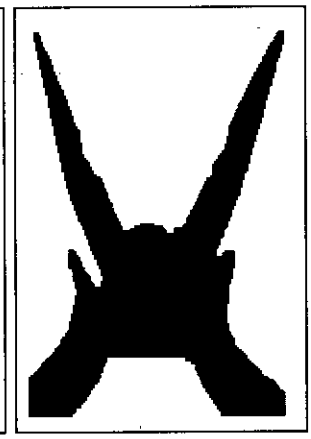


Fig. 3.24.

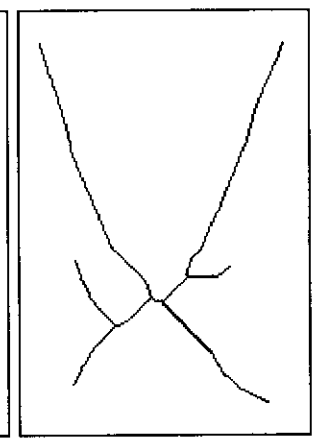


Fig. 3.25.

Fig. 3.22 Image originale.

Fig. 3.23 Résultat de l'opération de squelettisation appliquée à l'image 3.22.

Fig. 3.24 Lissage des contours par une fermeture suivie d'une ouverture (élément structurant carré 3x3).

Fig. 3.25 Résultat de l'opération de squelettisation appliquée à l'image 3.24.

3.4 Conclusion

Ce chapitre nous a permis de décrire quelques disciplines et opérations en traitement d'images, et de mettre en évidence les possibilités offertes. Ces possibilités constituent une étape nécessaire pour tout processus d'analyse ou de reconnaissance. Nous avons pu aussi introduire la notion de morphologie mathématique qui constitue un outil très puissant de plus en plus utilisé dans les systèmes de vision artificielle. Un exemple de l'utilité et de la puissance des opérations morphologiques est illustré par les figures 3.22 à 3.25.

Chapitre IV

PARALLELISATION D'ALGORITHMES DE TRAITEMENT D'IMAGES

4.1 Introduction

La puissance de calcul exigée par les programmes scientifiques a poussé au développement et à l'industrialisation de machines parallèles souvent appelées super-ordinateurs. Cependant, les capacités théoriques annoncées par les constructeurs sont rarement atteintes dans la pratique. L'exploitation efficace de ces machines directement par les utilisateurs, est difficile et exige une bonne connaissance de l'architecture.

Le traitement d'images reste parmi les disciplines les plus ciblées par le développement de machines parallèles spécialisées et d'algorithmes parallèles.

Dans ce chapitre, nous introduirons dans la section 4.2 quelques notions sur la programmation parallèle. Dans la section 4.3, sont décrits les types de parallélisme qui peuvent exister dans les algorithmes de traitement d'images. Le paragraphe 4.4 est consacré à deux processeurs parallèles de traitement d'images. Enfin, dans le paragraphe 4.5, est détaillée l'implantation des algorithmes parallèles sur un réseau de Transputers [1][2][21][22][23].

4.2 Eléments de programmation parallèle

4.2.1 Concepts de la programmation parallèle :

L'utilisation optimisée des architectures parallèles nécessite une bonne distribution du travail sur les processeurs. Ceci reste la tâche essentielle du logiciel et ce à travers deux moyens [2] :

- La détection du parallélisme.
- L'expression du parallélisme.

a) Détection du parallélisme :

Cette approche consiste à transformer un programme séquentiel en un programme parallèle et en assurer l'exécution. 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 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 mieux la structure du programme.

b) 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 (*Think parallel*), leur 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.

4.2.2 Parallélisation des boucles :

Un programme passe la plus grande partie de son temps dans l'exécution des boucles. Une exécution efficace de celles-ci ferait gagner un temps considérable. Pour cette raison, beaucoup de travaux de recherche se sont focalisés sur la parallélisation de boucles dans le but de réduire les temps d'exécution des programmes.

Trois sortes de boucles peuvent être identifiées dans un programme : le **forserial**, le **forall** et le **foracross** (doserial, doall et doacross).

Le forserial est une boucle **for** classique dont les itérations doivent s'exécuter séquentiellement pour conserver la sémantique du programme. Le forall est une boucle dont toutes les itérations peuvent s'exécuter simultanément. Le foracross est, quant à lui, une boucle dont les itérations peuvent être exécutées en pipeline. Les itérations sont lancées dans l'ordre, mais l'itération **i+1** peut commencer avant la fin de l'itération **i** de telle sorte qu'il y ait, à chaque instant, plusieurs itérations en cours d'exécution.

Le parallélisme qu'on se propose d'exploiter se situe au niveau des instructions du programme. Outre le parallélisme entre une ou plusieurs instructions (ou blocs d'instructions) non englobées par des boucles communes, le parallélisme peut aussi exister entre des itérations différentes d'une même boucle. Le parallélisme le plus facile à exploiter, et le plus rentable, correspond à celui entre les itérations d'une même boucle et plus particulièrement le forall. Pour cette raison, beaucoup de travaux se sont focalisés sur la transformation de boucles dans le but de faire apparaître le maximum possible de forall. La technique le plus utilisée est la distribution de boucles [22][23].

L'autre cas de parallélisme entre les itérations d'une même boucle est celui du foracross. L'exploitation du parallélisme des foracross est plus difficile que celui du cas particulier du forall. D'une part, la détection de ces boucles nécessite une analyse de dépendance plus fine. D'autre part, le coût de parallélisation est plus élevé que celui des forall. En effet, sur une machine synchrone (pipeline), l'ordonnancement des itérations d'un foracross peut s'obtenir par une écriture appropriée du programme, alors que sur une machine asynchrone, il faut prévoir des opérations de synchronisation exécutées à chaque tour de boucle, ce qui engendre en général un coût prohibitif [22][23].

4.2.3 Allocation des processeurs :

Pour pouvoir exécuter efficacement un programme sur une machine multiprocesseur, il faut décider de la façon de distribuer le code sur les différents processeurs disponibles ; ce problème est désigné par **l'allocation de processeurs**.

Souvent, on n'arrive pas à distribuer le code afin que les processeurs puissent travailler de façon asynchrone. Les raisons peuvent être dues à l'insuffisance du parallélisme dans le programme, mais aussi à la difficulté de trouver la bonne

allocation qui nous permette d'obtenir ce résultat optimal. Les processeurs doivent donc se synchroniser pour garantir le respect des contraintes de précédence. Ce problème est désigné par **l'ordonnancement sur multiprocesseur**.

Pour trouver une Parallélisation efficace, deux approches d'allocation de processeurs sont envisageables : l'approche dynamique et l'approche statique [22].

a) L'allocation dynamique :

La parallélisation dynamique d'un programme est justifiée quand on ne connaît pas à priori le nombre de processeurs à allouer au programme ou à une composante du programme. L'approche dynamique est aussi intéressante quand les durées d'exécution des composantes ne peuvent être estimées au moment de la compilation avec une bonne approximation. Le choix de l'allocation des processeurs et leur ordonnancement doit être effectué rapidement. La mise en oeuvre d'un algorithme complexe de décision n'est pas justifiée, car elle ralentirait considérablement le temps d'exécution global du programme.

b) L'allocation statique :

Dans ce cas, le compilateur se charge de préparer l'exécution parallèle. La mise en oeuvre d'algorithmes plus coûteux en temps est possible. Le nombre de processeurs alloués au programme doit être connu au moment de la compilation. La phase de compilation sera, évidemment, plus longue. Par contre, le coût de gestion des tâches parallèles doit diminuer, puisque le code de chacun des processeurs est préparé d'avance.

4.3 Parallélisme dans les algorithmes de traitement d'images

4.3.1 Le parallélisme des pixels :

Les opérations ponctuelles modifient la valeur de chaque pixel de l'image de manière indépendante des autres pixels. Cette indépendance de calcul entre les points et la répétition de la même opération élémentaire sur tous les points favorisent le parallélisme SIMD (cas des array processeurs).

4.3.2 Le parallélisme des voisinages :

Un processeur spécialisé est utilisé pour réaliser les mêmes opérations sur tous les voisinages de pixels d'une image. Le processeur accède de manière parallèle à tous les points du voisinage, calcul une valeur (de manière parallèle ou séquentielle) et passe ensuite au voisinage suivant.

4.3.3 Le parallélisme des tâches :

Dans ce cas, le traitement est réparti en tâches indépendantes. Chaque processeur exécute sa propre tâche. C'est le parallélisme MIMD de la classification de Flynn [1].

4.4 Exemples de processeurs parallèles de traitement d'images

Plusieurs processeurs parallèles ont été conçus dans le but de réaliser des opérations de traitement d'images. Dans ce paragraphe, nous en citerons deux. La suite du chapitre est consacrée à une architecture, différente des deux premières, et que nous avons choisie pour l'implantation d'algorithmes parallèles de traitement d'images.

4.4.1 Processeur pipeline à flot de données :

Conçu pour traiter une grande quantité de données au moyen de programmes relativement courts, le μPD 7281 de la société NEC permet d'augmenter considérablement le rendement et l'efficacité des systèmes d'analyse d'images. Ce composant VLSI de type MIMD intègre simultanément les concepts de type pipeline et d'architecture à flot de données.

Dans les architectures à flot de données, le compteur ordinal est supprimé. Le cadencement par les données consiste à déléguer le contrôle de l'exécution des instructions aux données elles-mêmes. Naturellement, ceci ne peut se faire qu'en modifiant la structure de ces données et en introduisant par exemple, la notion de champ d'une instruction. On définit de la sorte une structure de données à plusieurs champs. La donnée propre ne présente qu'un de ces champs, tandis que les autres sont réservés au contrôle du flot d'instructions. Dans cette architecture, une instruction n'est exécutée que lorsque tous ses opérandes sont présents. Dans ce cas, plusieurs instructions peuvent être exécutées simultanément, pourvu que tous leurs opérandes soient disponibles [21].

4.4.2 Processeur matriciel systolique :

Le *Geometric Arithmetic Parallel processor* est un composant VLSI micro-programmable de type SIMD. Développé par la société NCR, ce macro-composant plus connu sous le nom de GAPP, est constitué d'une grille de 6 x 12 = 72 processeurs élémentaires.

Chacune des 72 cellules possède une structure systolique et peut communiquer avec ses quatre voisins. Le terme systolique a été employé à l'origine pour évoquer les contractions du coeur qui aspirent et refoulent le sang. La structure systolique peut être considérée comme une généralisation de la structure pipeline. Ce concept architectural a été introduit par H.T. Kung [21]. Une architecture systolique se compose d'un ensemble de cellules interconnectées, chacune pouvant réaliser une opération élémentaire.

Dans cette structure, les informations circulent entre les cellules selon le technique pipeline, les cellules des extrémités étant dévouées aux communications d'entrée/sortie. Une opération peut être effectuée sur la donnée présente dans chaque cellule, ceci maintient un flux de données régulier à travers le réseau. Tous les processeurs sont synchronisés par une horloge de référence, afin que les données soient traitées et transmises périodiquement à travers le réseau.

4.5 Implantation d'algorithmes de traitement d'images sur un réseau de Transputers

4.5.1 Etude de l'architecture cible :

Le Transputer doit s'intégrer dans des environnements matériels existants (PC, stations de travail, systèmes d'acquisition / restitution, ...etc.). Dans une structure standard comportant un réseau de Transputers et un système hôte, celui-ci dialogue avec le premier Transputer du réseau qui joue le rôle de Transputer de contrôle. Le rôle de l'ordinateur hôte est passif vis à vis du déroulement de l'application. Du fait de son architecture à liens série, le Transputer de contrôle échange des informations avec le système hôte via un circuit intermédiaire appelé adaptateur de liens (IMS C011, IMS C012). Ces circuits permettent la connexion des Transputers aux bus parallèles standards par conversions série-parallèle et parallèle-série [21][26].

La carte à Transputers dont nous disposons est une *TMB 04* de TRANSTECH pour IBM PC XT ou AT et compatibles. Elle dispose d'un Transputer maître (IMS T800 20MHz) avec une mémoire locale de 4 M octets (extensible jusqu'à 16 M). Cette carte peut recevoir 4 TRAM (TRAnsputer Modules : Transputer et mémoire locale) reliés en pipeline ; le lien 2 de chaque TRAM est connecté au lien 1 du TRAM suivant dans la chaîne. Le reste des liens est relié à un connecteur pour permettre une liaison avec d'autres cartes à Transputers. Une boucle est réalisée grâce à la liaison entre le lien 2 du dernier module TRAM et le lien 1 du Transputer maître. Le lien 0 de ce dernier est relié au bus du système hôte via l'adaptateur de liens IMS C012 (figure 4.1) [8].

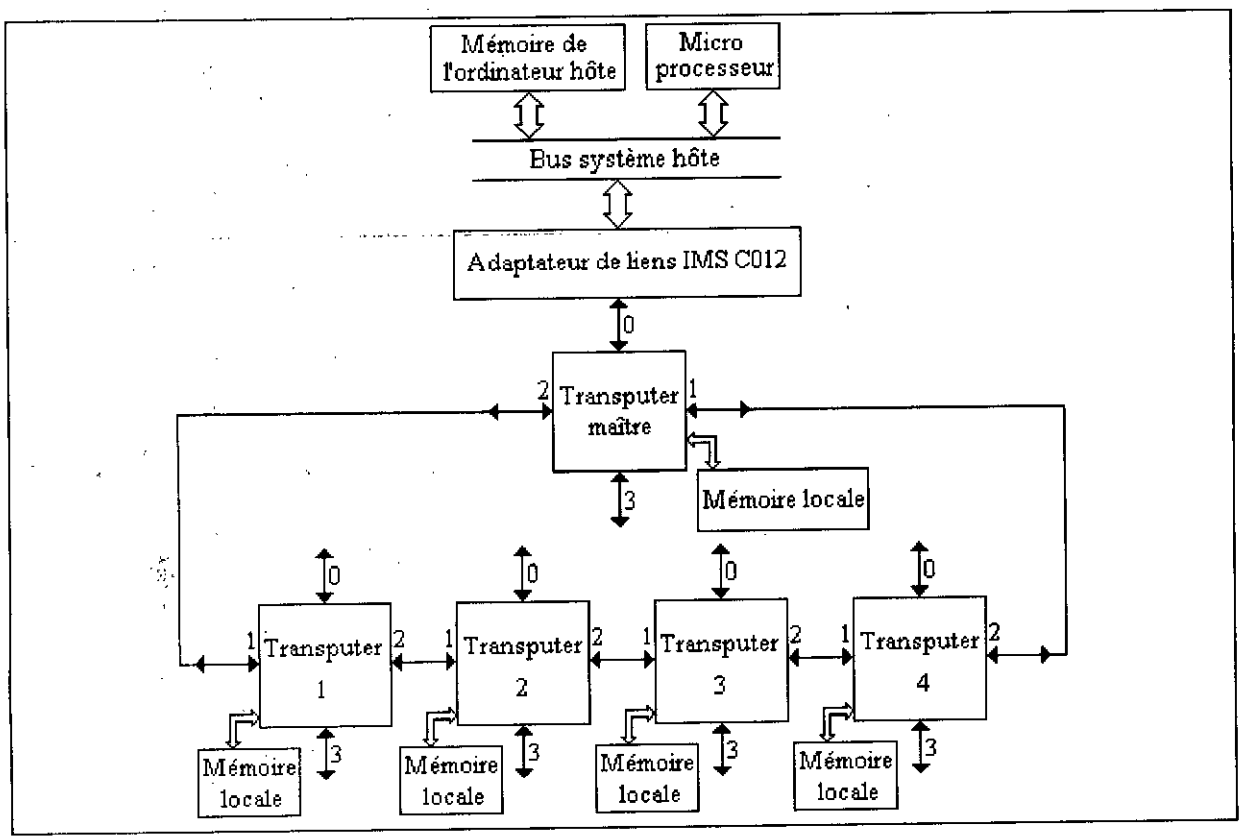


Fig. 4.1 Synoptique de la machine parallèle.

Le mécanisme d'exécution d'une application sur le réseau de Transputers est le suivant :

- Configuration du réseau par le Transputer maître.
- Distribution, par le Transputer maître, des codes et des données aux divers processeurs du réseau (la configuration doit être telle que tous les processeurs soient accessibles à partir du Transputer maître).
- Démarrage de l'exécution.
- Collecte des résultats.

Nous disposons de cette manière, d'une architecture de type MIMD où chaque processeur est téléchargé avec un programme propre. Ceci nous amène directement aux problèmes à résoudre pour chaque algorithme :

- Quelle topologie de réseau choisir ?
- Comment répartir au mieux les tâches ?

4.5.2 Topologies pour traitement d'images :

Dans beaucoup d'applications, la configuration optimale du réseau n'est pas évidente à trouver. Ce domaine d'intérêt fait d'ailleurs l'objet de travaux de recherches et de développement. On peut distinguer schématiquement deux grandes classes de topologies :

- Les constructions régulières reflétant le parallélisme géométrique ou spatial.
- Le type ferme de processeurs, utilisant des structures en arbre ou linéaires (pipeline) plus adaptées à une décomposition de l'algorithme en sous tâches.

Dans la première approche, le parallélisme est introduit en exploitant la structure spatiale régulière découlant de la symétrie du problème en termes de données. Plus simplement, un algorithme contient une zone programme et une zone données. L'idée est de découper l'algorithme en parties de plus en plus petites jusqu'à ce que chaque partie soit indépendante des autres et possède ses propres données. L'exemple le plus répandu est celui de **l'hypercube** de degré **n**. On illustre un hypercube en le projetant dans l'espace. Par exemple, un hypercube de degré 2 est représenté par un carré, un hypercube de degré 3 par un cube, un hypercube de degré 4 par deux cubes décalés et reliés par leurs sommets, ...etc (figure 4.2). Remarquons qu'on peut associer un Transputer à chaque sommet de la structure jusqu'au degré 3. Pour les degrés supérieurs, les quatre liens du Transputer sont insuffisants pour permettre l'association un noeud-un processeur. On peut alors employer l'idée qui consiste à considérer le noeud comme une entité composée de plusieurs Transputers. Avec deux Transputers par exemple, on dispose de six liens externes au lieu de quatre.

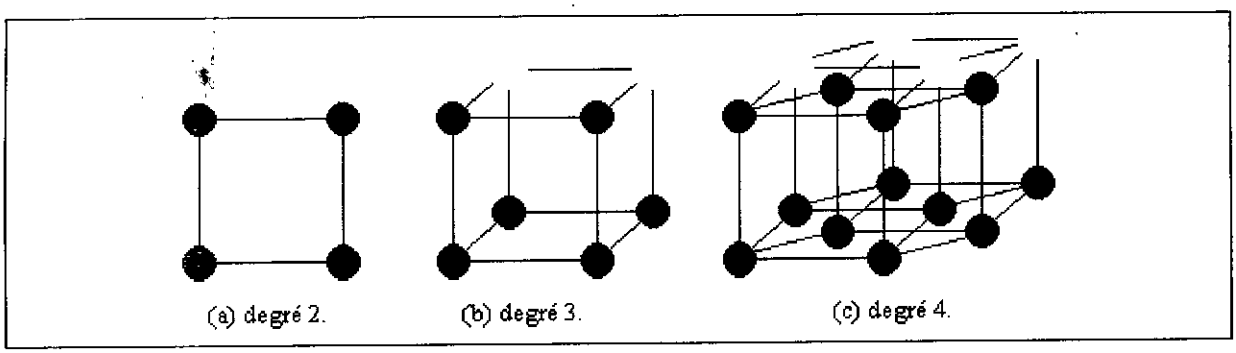


Fig. 4.2 Structures en hyprecubes.

La structure hypercubique réduit au minimum le nombre de liens de communication entre 2^n processeurs. Cette propriété est fondamentale dans les algorithmes où les flux de données échangés sont importants [21].

Lorsque l'algorithme est décomposable en tâches indépendantes, on applique le concept de ferme de processeurs (*process farming*) [21][24][25]. Les tâches étant indépendantes, elles peuvent être traitées simultanément par des processeurs distincts. La ferme est modélisée par un ensemble de processus Occam. Il existe un processus maître (*farmer process*) et un ensemble de processus esclaves (*worker processes*). Le processus maître contrôle l'organisation du travail et l'allocation des tâches aux processus ouvriers. Il distribue le travail aux ouvriers, recueille les résultats et redonne du travail aux ouvriers inactifs. Du point de vue topologique, on distingue deux configurations :

- La ferme linéaire ou pipeline.
- La ferme en arbre.

Le choix d'une topologie plutôt qu'une autre est délicat, et dépend fortement de l'application traitée. Les figures 4.3 et 4.4 montrent comment peut on réaliser les deux topologies, linéaire et en arbre, sur notre machine parallèle.

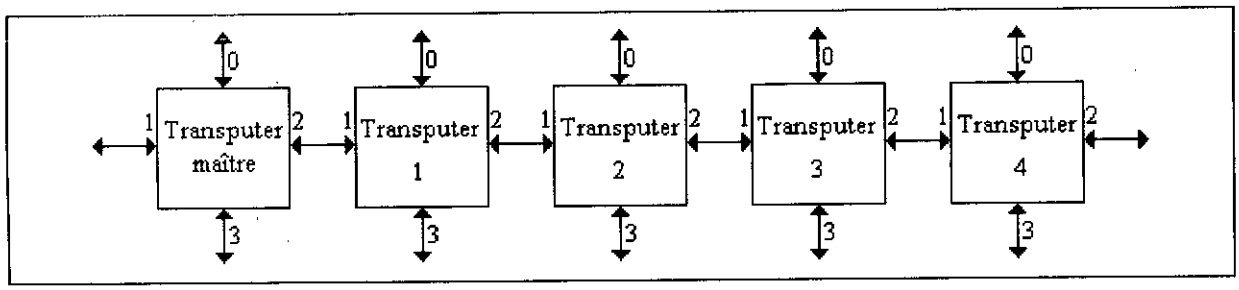


Fig. 4.3 Ferme de processeurs linéaire (pipeline).

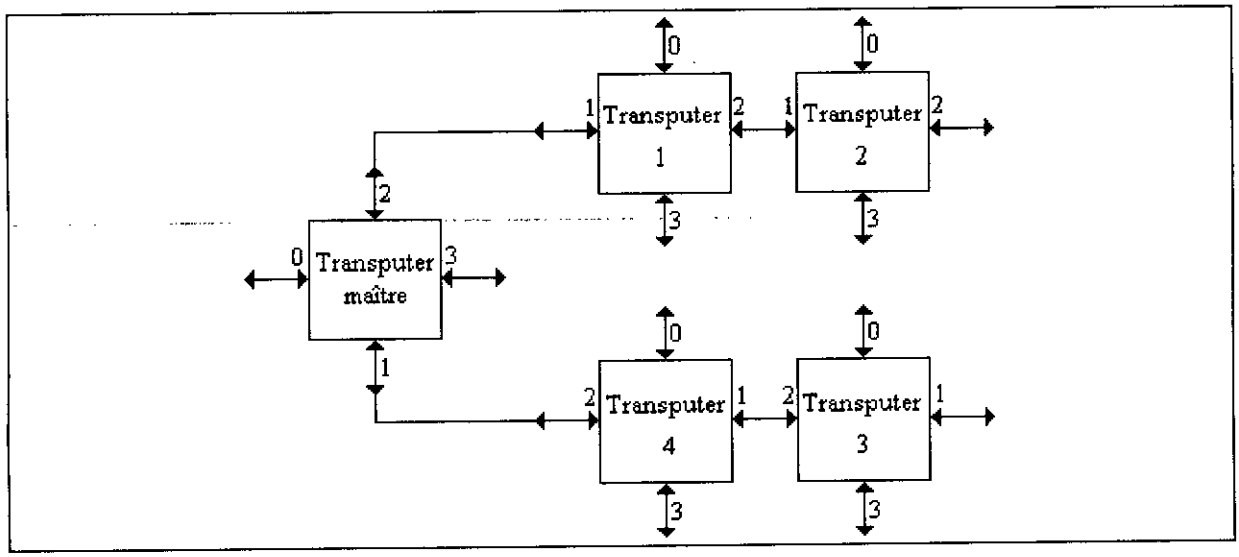


Fig. 4.4 Ferme de processeurs en arbre.

Dans ce qui suit, nous allons décrire quelques algorithmes de traitement d'images. Nous donnerons leur structure séquentielle, puis leur implantation parallèle. La parallélisation de ces algorithmes comporte deux étapes successives :

- Duplication de l'image sur l'ensemble des Transputers.
- Division de l'image en p parties correspondant à p Transputers, puis affectation de chaque partie à chaque Transputer.

Il est à noter que nous avons opté pour une allocation statique des processeurs. De plus, pour une meilleure exploitation de l'architecture, le processeur maître réalise en premier lieu la configuration et le routage du réseau, puis prend en charge, comme les processeurs ouvriers, une partie de l'image pour réaliser le traitement désiré.

Les algorithmes étant divisés en processus indépendants, il est évident que c'est la topologie en arbre qui doit être adoptée, car c'est elle qui procure le coût de communication le moins important (par rapport à la configuration linéaire).

Dans un réseau de Transputers, les messages émis par certains processeurs doivent traverser plusieurs processeurs du réseau pour se rendre au Transputer concerné. Il est donc nécessaire de définir un processus supplémentaire dit de routage. De ce fait, on classe les Transputers d'un réseau en deux types (figure 4.5) :

- Les Transputers noeuds qui traitent leur propres données et assurent le routage des données des autres (Transputers **1** et **4**).

- Les Transputers de calcul qui ne font que traiter les données et retourner des résultats (Transputer **2** et **3**).

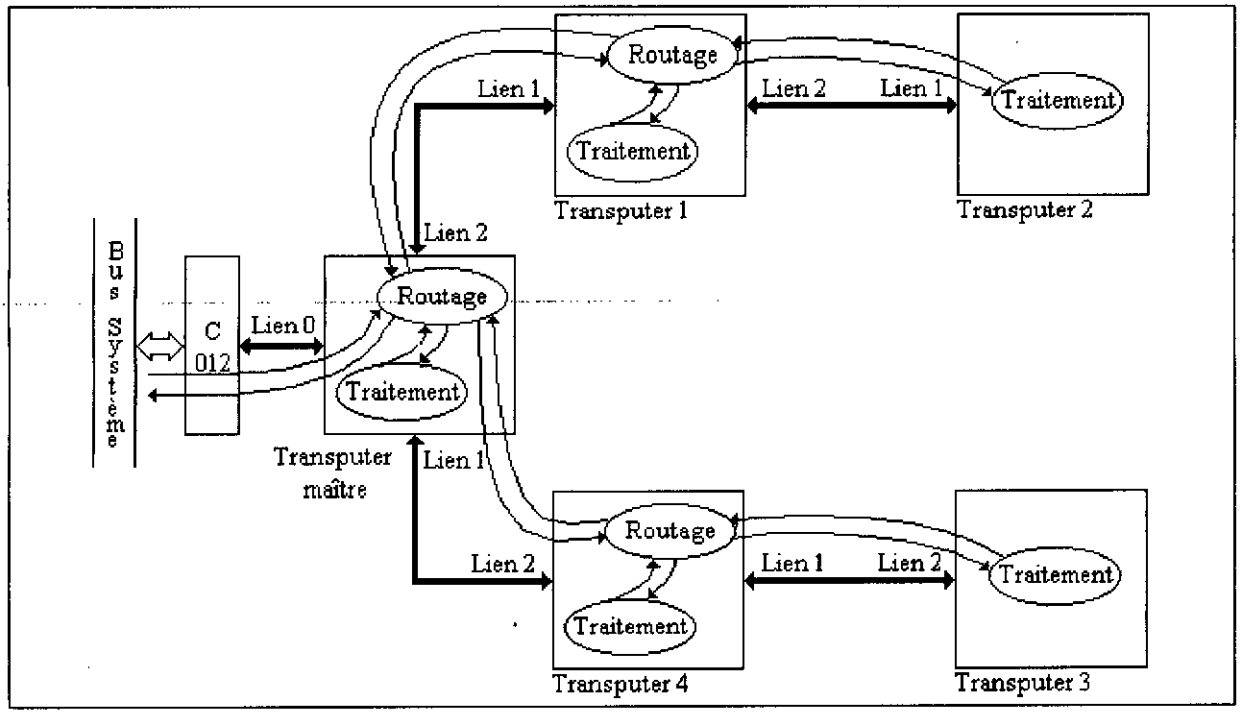


Fig 4.5 Définition des processus et des chemins des données.

Pour tous les algorithmes, on notera par **traite**, la procédure qui définit le processus à exécuter sur chaque Transputer ouvrier, recevant ses données sur le canal **linkin** et envoyant ses résultats sur le canal **linkout**. Les Transputers noeuds doivent exécuter le processus supplémentaire de routage, qui consiste à recevoir des données à partir du Transputer maître et à les envoyer sur les Transputers de calcul. Puis après exécution du traitement, ils récupèrent les résultats pour les envoyer au Transputer maître (figure 4.5). L'image est divisée en cinq (5) blocs, chacun étant affecté à un Transputer du réseau (figure 4.6).

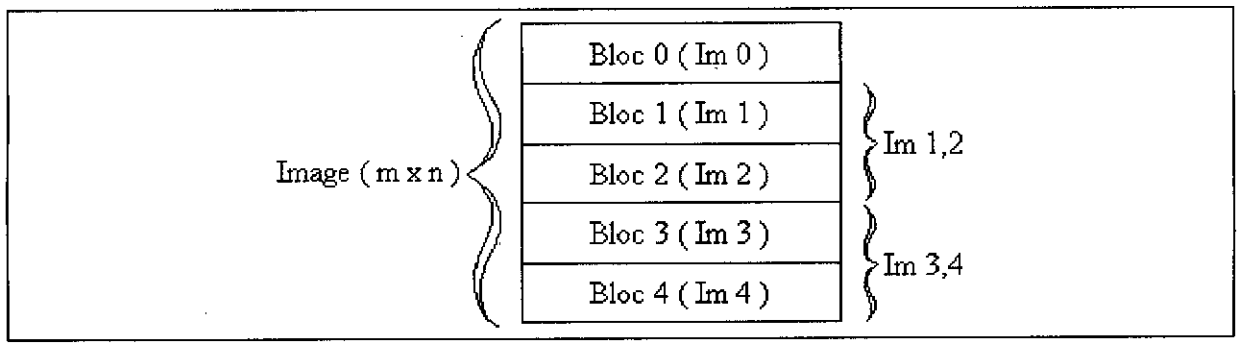


Fig 4.6 Division de l'image en blocs.

4.5.3 Parallélisation d'algorithmes de base :

a) Calcul de histogramme d'une image :

Soit une image représentée par un tableau de dimensions m x n. L'algorithme séquentiel qui permet de calculer l'histogramme, décrit en Occam est le suivant :

```
[m][n] BYTE image :
[256] INT hist :
SEQ
   hist := 0                          --initialisation des compteurs.
   SEQ i = 0 FOR m
      SEQ j = 0 FOR n
         hist[ image [i][j] ] := hist[ image [i][j] ] + 1
```

L'algorithme procède ligne par ligne dans l'évaluation de l'histogramme. Ayant m lignes dans l'image, on peut partitionner celle-ci en p segments, chaque segment possède m/p=s lignes. On peut ainsi diviser le programme en p processus parallèles, dont chacun calcule l'histogramme d'un segment de l'image. La contrainte cependant, est que tous les processus coopèrent pour former l'histogramme de l'image complète. Ils partagent le tableau **hist[256]** pour la mise à jour des compteurs d'occurrence des niveaux de gris.

Pour remédier à ce problème, une première approche, utilisée sur les machines à mémoire partagée [1], consiste à déclarer un tableau pour l'histogramme sur la mémoire locale du Transputer maître. Ce tableau sera partagé par les processus concurrents. La section critique est réalisée grâce à la structure alternée (ALT) de l'Occam. Le nombre de processus concurrents est égal au nombre de processus ouvriers qui ont une liaison directe sur un lien du Transputer maître. La mise à jour des éléments du tableau représentant l'histogramme doit se faire en exclusion mutuelle, ce qui constitue avec le nombre restreint de processeurs les désavantages de cette méthode.

Ces problèmes peuvent être détournés en adoptant une autre approche qui utilise un histogramme local pour chaque segment de l'image. La somme de tous les histogrammes est réalisée par le processeur maître pour donner l'histogramme final.

Chaque processeur ouvrier devra exécuter la procédure suivante :

```
PROC traite
  [m][n] BYTE image :
  [256] INT hist :
  INT nl :                    -- nl = nombre de lignes.
  INT de :                    -- de = ligne de départ.
  SEQ
    hist := 0                 -- initialisation.
    SEQ i = de FOR nl
      SEQ j = 0 FOR n
        hist[ image[i][j] ] := hist[ image[i][j] ] + 1
:
```

Pour un réseau constitué par un Transputer maître et quatre Transputers ouvriers, l'implantation de l'algorithme sera la suivante : (on désignera par processus **x**, le processus qui s'exécute sur le Transputer **x** du réseau).

```
[m][n] BYTE image :
[256] INT hist , lhist :
PAR
  SEQ                                   -- Processus maître.
    PAR
      link2out ! im1,2
      link1out ! im3,4
    traite
    SEQ i = 0 FOR 2                     -- collecte des résultats.
      link2in ? lhist
      hist := hist + lhist
      link1in ? lhist
      hist := hist + lhist
  SEQ                                   -- Processus 1
    link1in ? im1,2
    link2out ! im2
    traite
    link1out ! lhist
    link2in ? lhist
    link1out ! lhist
```

```
SEQ                                    -- Processus 2
    link1in ? im2
    traite
    link1out ! lhist
SEQ                                    -- Processus 3
    link2in ? im3
    traite
    link2out ! lhist
SEQ                                    -- Processus 4
    link2in ? im3,4
    link1out ! im3
    traite
    link2out ! lhist
    link1in ? lhist
    link2out ! lhist
```

b) Egalisation d'histogramme :

L'algorithme d'égalisation d'histogramme est composé de trois tâches :

- Calcul de l'histogramme de l'image originale.
- Calcul de l'histogramme égalisé.
- Modification de l'image originale pour obtenir l'image égalisée.

Sa structure séquentielle est la suivante :

```
[m][n] BYTE image , e.image :
[256] INT hist , e.hist :
INT sum :
SEQ
   histogram (image, hist)             -- calcul de l'histogramme.
   sum := 0
   SEQ i = 0 FOR 256
      sum := sum + hist[i]
      e.hist[i] := 255 * sum
   SEQ i = 0 FOR m
      SEQ j = 0 FOR n
         e.image[i][j] := e.hist[ image[i][j] ]
```

Les trois tâches citées sont nécessairement séquentielles. Le calcul de l'histogramme égalisé dépend de tout le tableau de l'histogramme original, et de même, la modification de l'image ne peut se faire qu'après évaluation de tous les éléments du tableau de l'histogramme égalisé.

A partir de ceci, il est évident que le parallélisme de tâches ne peut être utilisé. Il faut alors chercher le parallélisme au niveau des boucles. La première est celle qui permet de calculer l'histogramme original. La seconde boucle, dont la fonction est d'évaluer l'histogramme égalisé, est une boucle du type **foserial** : l'itération **i+1** ne peut être réalisée qu'après la fin de l'itération **i**. la seule alternative reste donc dans la parallélisation de la boucle de la modification de l'image, qui pour sa part est une boucle du type **forall**. L'image peut être divisée en segments, chaque segment est pris en charge par un processeur, pourvu qu'il dispose du tableau **hist**. Le code à exécuter par chaque processeur ouvrier est décrit ci-dessous :

```
PROC traite
    [m][n] BYTE image :    -- image originale.
    [nl][n] BYTE e.image :-- segment de l'image égalisée (nl: nombre de lignes)
    [256] INT hsit , e.hist :
    INT de :                    -- ligne de départ.
    SEQ
        sum := 0
        SEQ i = 0 FOR 256
            sum := sum + hist[i]
            e.hist[i] := 255 * sum
        SEQ i = de FOR nl
            SEQ j = 0 FOR n
                e.image[i][j] := e.hist[ image[i][j] ]
:
```

L'implantation parallèle de l'algorithme sur un réseau de cinq Transputers est :

```
PAR
    [m][n] BYTE image , r.image :
    SEQ                                 -- processus maître.
        histogram               -- calcul de l'histogramme original.
```

```
PAR                    -- duplication de l'histogramme.
   link2out ! hist
   link1out ! hist
traite
link2in ? r.im1        -- collecte des résultats.
link1in ? r.im4
link2in ? r.im2
link1in ? r.im3
SEQ                    -- processus 1.
histogram
link1in ? hist
link2out ! hist
traite
link1out ! e.image
link2in ? e.image
link1out ! e.image
SEQ                    -- processus 2.
histogram
link1in ? hist
traite
link1out ! e.image
SEQ                    -- processus 3.
histogram
link2in ? hist
traite
link2out ! e.image
SEQ                    -- processus 4.
histogram
link2in ? hist
link1out ! hist
traite
link2out ! e.image
link1in ? e.image
link2out ! e.image
```

c) Le seuillage :

L'algorithme séquentiel est le suivant :

```
[m][n] BYTE image , r.image :
INT th :      -- seuil.
SEQ i = 0 FOR m
  SEQ j = 0 FOR n
    IF
      image[i][j] ≥ th
        r.image[i][j] := 255
      ELSE
        r.image[i][j] := 0
```

Il est constitué d'une boucle du type **forall** qui permet de balayer tous les points de l'image. Le parallélisme de pixels est donc évident; c'est un cas typique de parallélisation sur un array processeur. Dans notre approche, nous remplaçons un bloc d'éléments de traitement du processeur matriciel par un Transputer. Chacun permet alors de réaliser le seuillage d'une partie de l'image.

```
PROC traite
  [m][n] BYTE image :
  [nl][n] BYTE r.image :
  INT th :
  SEQ i = de FOR nl
    SEQ j = 0 FOR n
      IF
        image[i][j] ≥ th
          r.image[i][j] := 255
        ELSE
          r.image[i][j] := 0
:
```

d) Le filtrage spatial :

Algorithme séquentiel :

```
[m][n] BYTE image , r.image :
[3][3] INT mask :
SEQ i = 1 FOR m-2
  SEQ j = 1 FOR n-2
    r.image[i][j] := conv ( image[i][j] , mask )
```

Pour tous les algorithmes de filtrage spatial, **conv** permet de réaliser la convolution discrète entre le voisinage du point **image[i][j]** et le masque de convolution (voir section 3.3.2 du chapitre 3). Le type de filtre est spécifié en modifiant les valeurs du masque, sauf dans le cas du filtrage médian, où ce n'est pas une convolution qui est réalisée, mais une recherche de la valeur médiane dans le voisinage.

Le parallélisme qui existe dans ce cas est un parallélisme de voisinages. L'idée reste la même que celle pour l'algorithme de seuillage, sauf qu'il faudrait tenir compte de l'effet de bord. Cet effet existe car le filtrage de la première et la dernière ligne de tous les blocs (sauf le premier et le dernier), nécessite respectivement la dernière ligne du bloc précédent et la première ligne du bloc suivant (le premier et le dernier bloc n'ont besoin que d'une seule ligne supplémentaire dans le cas d'un masque 3 x 3) (figure 4.6).

```
PROC traite
    [m][n] BYTE image :
    [nl][n] BYTE r.image :
    SEQ i = de FOR nl
        SEQ j = 0 FOR n-2
            r.image[i][j] := conv ( image[i][j] , mask )
:
```

e) Le filtrage morphologique :

Les deux principales opérations morphologiques sont la dilatation et l'érosion. Leurs algorithmes séquentiels respectifs sont décrits ci-dessous :

Dilatation

```
[m][n] BYTE image , r.image :
[5][5] INT stru :                          -- élément structurant.
INT smax :
SEQ y = 2 FOR m-4
   SEQ x = 2 FOR n-4
      smax := 0
      SEQ i = -2 FOR 5
         SEQ j = -2 FOR 5
```

```
                IF
                  stru[i+2][j+2] = 1
                    IF
                      image[y+i][x+j] > smax
                        smax := image[y+i][x+j]
                r.image[y][x] := smax
```

Erosion

```
[m][n] BYTE image , r.image :
[5][5] INT stru :                          -- élément structurant.
INT smin :
SEQ  y = 2  FOR  m-4
   SEQ  x = 2  FOR  n-4
      smin := 255
      SEQ  i = -2  FOR  5
         SEQ  j = -2  FOR  5
            IF
              stru[i+2][j+2] = 1
                IF
                  image[y+i][x+j] < smin
                    smin := image[y+i][x+j]
            r.image[y][x] := smin
```

Le parallélisme qui existe dans ces deux algorithmes est identique à celui dans les algorithmes de filtrage spatial. C'est le parallélisme de voisinages, sauf que dans ce cas le masque représente l'élément structurant. L'effet de bord est toujours à prendre en considération, mais le nombre de lignes supplémentaires est plus grand. Ceci est dû au fait que les deux autres opérations morphologiques (ouverture et fermeture) sont réalisées grâce à une érosion suivie d'une dilatation ou une dilatation suivie d'une érosion. Ce qui fait que la première opération doit traiter deux lignes supplémentaires (masque 5 x 5) de chaque bord du bloc pour que la seconde opération puisse traiter correctement les lignes du bord du bloc.

La structure parallèle de ces deux algorithmes, ainsi que les algorithmes de seuillage et de filtrage spatial, implantée sur un réseau de cinq Transputers configurés selon la topologie en arbre est la suivante :

```
PAR
  [m][n] BYTE image , r.image :
  SEQ                              -- processus maître.
    PAR                            -- duplication de l'image.
      link2out ! im1,2
      link1out ! im3,4
    traite
    link2in ? r.im1                -- collecte des résultats (quatre blocs).
    link1in ? r.im4
    link2in ? r.im2
    link1in ? r.im3
  SEQ                              -- processus 1.
    link1in ? im1,2
    link2out ! im2
    traite
    link1out ! r.im1               -- transfert d'un bloc de l'image résultat.
    link2in ? r.im2
    link1out ! r.im2
  SEQ                              -- processus 2.
    link1in ? im2
    traite
    link1out ! r.im2
  SEQ                              -- processus 3.
    link2in ? im3
    traite
    link2out ! r.im3
  SEQ                              -- processus 4.
    link2in ? im3,4
    link1out ! im3
    traite
    link2out ! r.im4
    link1in ? r.im3
    link2out ! r.im3
```

f) La squelettisation :

L'algorithme de squelettisation est un algorithme itératif, qui dans chaque itération, permet d'éliminer les points des contours d'objets n'appartenant pas au

squelette. Ceci est réalisé en comparant le voisinage de chaque point avec plusieurs masques (section 3.3.3.f du chapitre 3). L'approche séquentielle de l'algorithme est la suivante :

```
[m][n] BYTE image :
BOOLEAN go.on , test :
SEQ
   go.on := TRUE
   WHILE go.on
      go.on := FALSE
      SEQ i = 1 FOR m-2
         SEQ j = 1 FOR n-2
            test := comp ( image[i][j] )
            IF
               test = TRUE
                  r.image[i][j] := 0
                  go.on := TRUE
               test = FALSE
                  r.image[i][j] := image[i][j]
```

La fonction **comp** permet de comparer le voisinage d'un point **image[i][j]** avec les différents masques et est vraie si le point n'appartient pas au squelette.

La parallélisation de cet algorithme ne peut pas se faire au niveau de la boucle la plus externe (boucle while), car une itération de cette boucle ne peut démarrer qu'après la fin de l'itération qui la précède.

Deux méthodes peuvent être testées pour paralléliser l'algorithme. La première consiste, comme pour les algorithmes précédents, à diviser l'image à chaque itération en blocs dont chacun est pris en charge par un processeur. La seconde méthode consiste à réaliser un pipeline dont chaque étage reçoit un voisinage de point pour le comparer avec un masque. Une boucle est réalisée permettant au dernier processeur de la chaîne d'envoyer une information au processeur maître et ce dans le but de décider si oui ou non le point appartient au squelette.

L'algorithme parallèle s'écrit donc :

```
[3][3] BYTE vois :                    -- voisinage d'un point de l'image.
INT code :
BOOLEAN go.on :
go.on := TRUE
WHILE go.on
    go.on := FALSE
    PAR
        SEQ x = 2 FOR m-4             -- processus maître.
          SEQ y = 2 FOR n-4
            Link2out ! vois
            Link1in ? code
            IF
              code = 1
                image[x][y] := 0
                go.on := TRUE
        SEQ                           -- processus 1.
          Link1in ? vois
          comp ( vois , mask )
          Link2out ! code ; vois
        SEQ                           -- processus 2
          Link1in ? vois
          comp ( vois , mask )
          Link2out ! code ; vois
        SEQ                           -- processus 3
          Link1in ? vois
          comp ( vois , mask )
          Link2out ! code ; vois
        SEQ                           -- processus 4
          Link1in ? vois
          comp ( vois , mask )
          Link2out ! code
```

La procédure **comp** permet de comparer le voisinage du point avec l'un des masques, si le point doit être éliminé, elle met la variable code à 1.

4.5.4 Mesure des performances :

Dans le cas idéal, le temps d'exécution **T(n)** d'un programme sur un système à **n** processeurs est donné par :

$$T(n) = \frac{T(1)}{n} \qquad 4.1$$

où T(1) est le temps d'exécution sur un seul processeur.

Dans le cas réel **T(n)** est donné par :

$$T(n) = \frac{T(1)}{n} + T(c) \qquad 4.2$$

T(c) est le temps des communications entre processeurs. Ce temps peut être calculé connaissant le débit des liens des Transputers (20 M bits / sec).

Le tableau suivant résume les temps d'exécution des différents algorithmes sur un seul processeur, l'estimation du temps dans le cas d'un réseau de 5 Transputers et l'accélération correspondante. Cette accélération est donnée par :

$$S = \frac{T(1)}{T(n)} \qquad 4.3$$

Traitement	1 Transputers	5 Transputers	S (Accé.)
Histogramme	0 : 0 : 94	0 : 0 : 29	3.24
Egalisation	0 : 1 : 99	0 : 0 : 61	3.26
Laplacien	0 : 9 : 25	0 : 2 : 05	4.51
médian	0 : 50 : 05	0 : 11 : 04	4.53
dilatation	0 : 36 : 82	0 : 7 : 57	4.85
Erosion	0 : 37 : 54	0 : 7 : 72	4.86
Ouverture / Fermeture	1 : 14 : 37	0 : 15 : 30	4.86
Squelettisation	13 : 42 : 83	4 : 24 : 97	3.1

Table 4.1

4.6 Conclusion

On remarque que l'accélération est d'autant plus importante que le temps d'exécution sur un Transputer est grand (sauf dans le cas de l'algorithme de squelettisation, où le temps de communication est très important). On atteint une accélération moyenne de 4.15.

Chapitre V

CROISSANCE EPITAXIQUE DE SILIÇURE DE CUIVRE SUR LA FACE (100) D'UN MONOCRISTAL DE SILICIUM

5.1 Introduction

En vue d'illustrer les possibilités de notre système de traitement d'images, nous l'avons utilisé pour étudier la morphologie de la surface d'un échantillon de Silicium sur lequel on a évaporé une couche de Cuivre d'environ 1000 Å d'épaisseur. Cette surface a été recuite sous vide pendant 2 heures à 700 °C et examinée au microscope électronique à balayage.

L'analyse EDX montre qu'elle est formée de plages sombres correspondant au Silicium du substrat, de plages claires correspondant à la formation de cristallites d'alliage CuSix et enfin de plages grises correspondant au Cuivre pur.

La photo prise a été numérisée en utilisant un scanner (format A4, résolution 1024 DPI). La figure 5.1 représente le résultat de la numérisation.

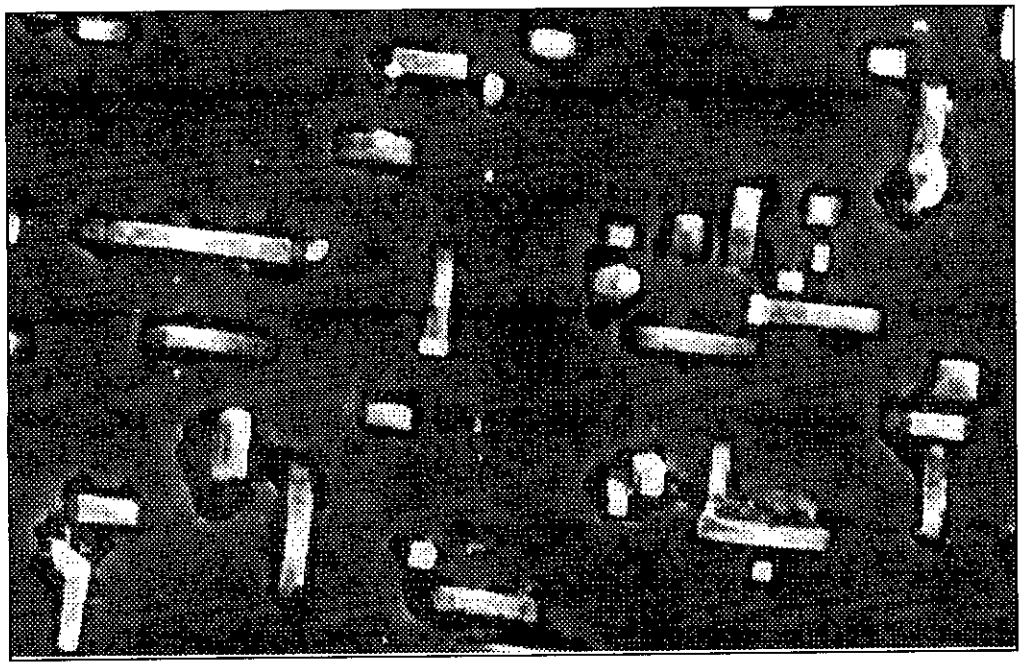


Fig. 5.1 Image numérisée.

5.2 Histogramme des différentes populations apparaissant sur l'image

Le Silicium, ayant un faible coefficient d'émission secondaire, apparaît sur l'image avec des niveaux de gris faibles (voisins de zéro). La partie de l'image qui correspond à la couche étalée de Cuivre se traduit par des niveaux de gris plus grands que ceux correspondants au Silicium. Les cristallites de Siliçure de Cuivre (CuSix) possèdent un coefficient d'émission secondaire plus important, et donc correspondent aux niveaux de gris les plus clairs.

D'autre part, on a remarqué que sur certaines parties de l'image acquise, le Cuivre possède un niveau de gris inférieur à celui du Silicium, ce qui justifie la présence de deux régions uniquement au niveau de l'histogramme de la figure 5.2. Donc, le calcul des surfaces des trois populations présentes sur l'image, en réalisant directement un seuillage à trois niveaux de gris sera entaché de beaucoup d'erreurs. C'est pour cela nous avons procédé à une correction des niveaux de gris du Silicium grâce a un seuillage local (par fenêtres), en plus du filtrage médian qui permet d'éliminer les points isolés (bruit). La figure 5.3 montre le résultat de ces deux derniers traitements.

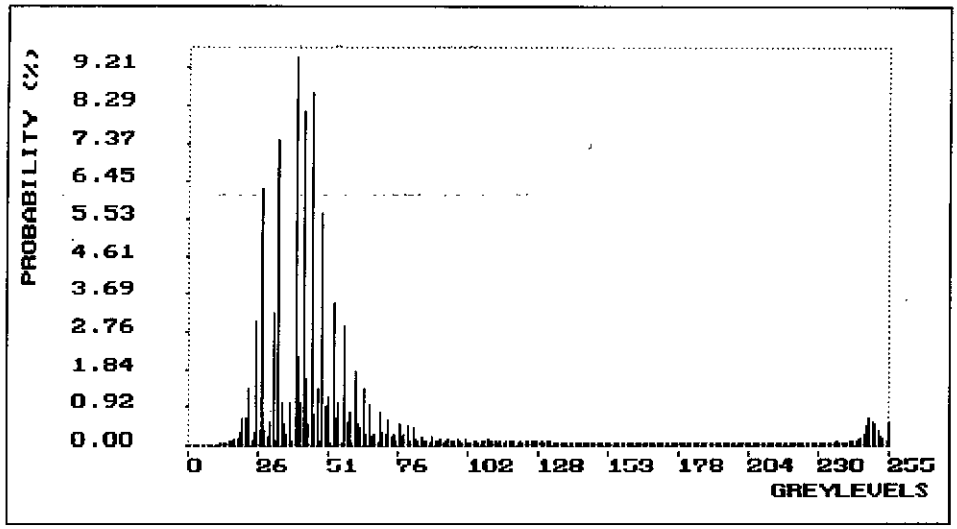


Fig. 5.2 Histogramme de l'image de la figure 5.1.

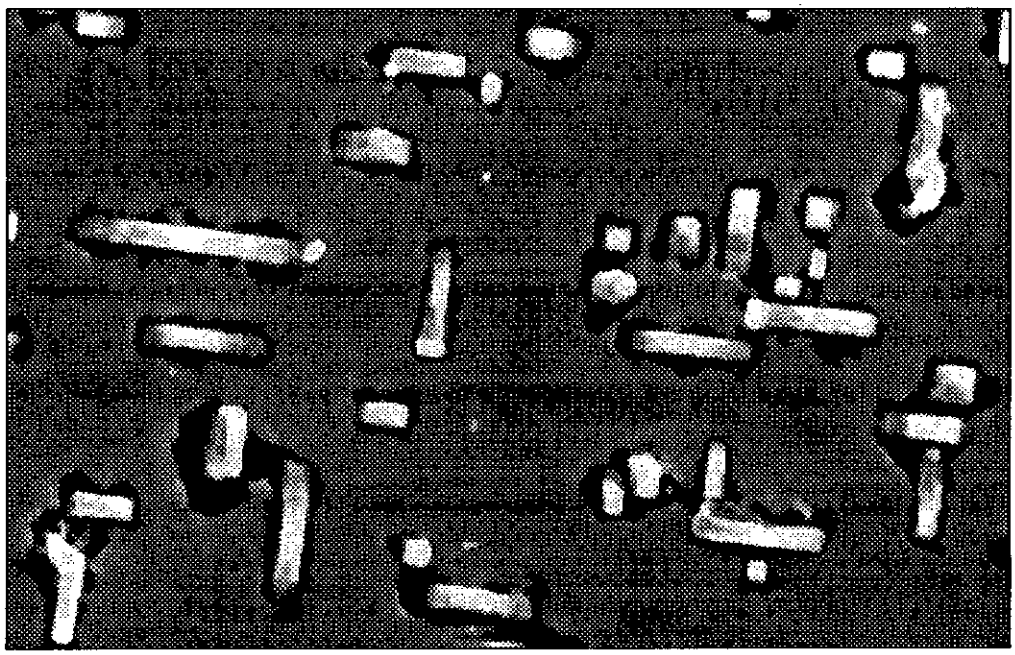


Fig. 5.3 Résultat du traitement de l'image numérisée.

La figure 5.4 représente l'histogramme correspondant à l'image de la figure 5.3. On distingue sur l'histogramme trois régions distinctes correspondant respectivement aux surfaces de Silicium, de Cuivre et des cristallites. Ces trois surfaces ont trois niveaux de gris qu'on peut estimer comme suit :

Silicium : ses niveaux de gris correspondent à la partie 1 de l'histogramme.
Cuivre : ses niveaux de gris correspondent à la partie 2 de l'histogramme.
Cristallites : leurs niveaux de gris correspondent à la partie 3 de l'histogramme.

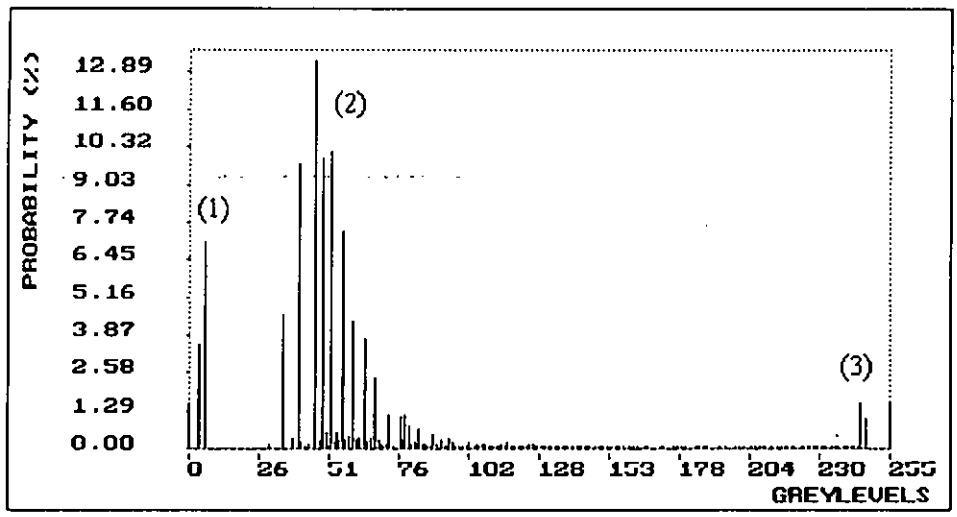


Fig. 5.4 Histogramme de l'image de la figure 5.3.

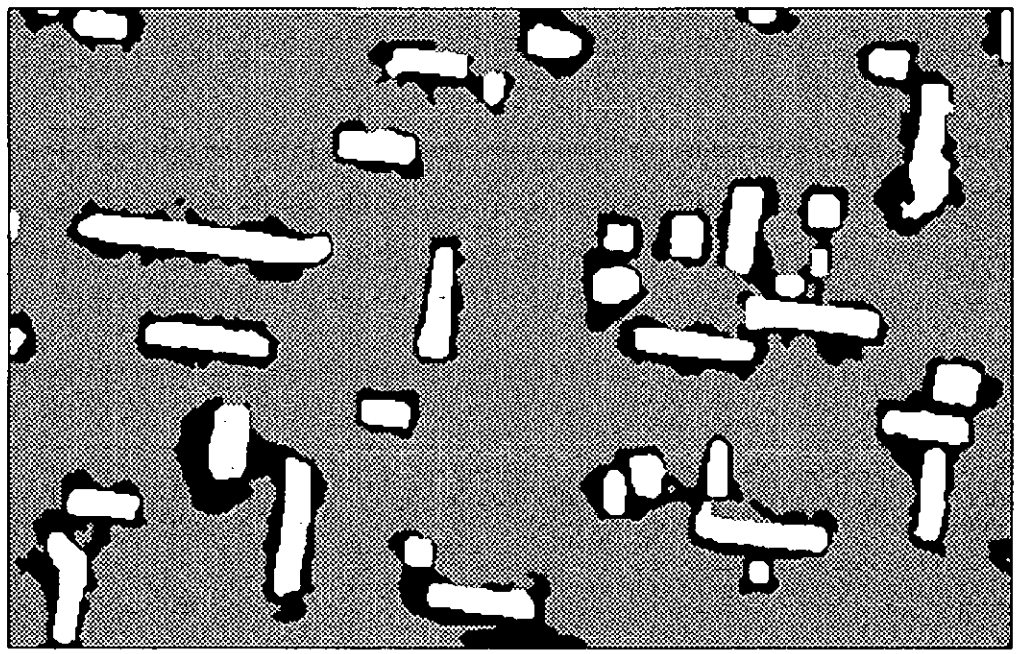


Fig. 5.5 Seuillage à trois niveaux de gris de l'image de la figure 5.3.

Pour l'évaluation des surfaces occupées par chaque classe, un seuillage à 3 niveaux de gris est réalisé (figure 5.5), les niveaux des deux seuils choisis sont 15 et 170. Il permet de compter le nombre de pixels qu'occupe chaque classe. La correspondance entre le nombre de pixels et les surfaces correspondantes a été ensuite établie. A partir de l'histogramme de cette image, on peut déduire les informations suivantes :

- Nombre total de pixels sur l'image : **159000**
- Nombre de pixels au niveau 0 (Si) : **19944**
- Nombre de pixels au niveau 100 (Cu) : **121812**
- Nombre de pixels au niveau 255 (Crist.) : **17244**

- Echelle : 10 µm ⟶ 139 pixels

Une surface de 100 μm^2 correspond à 139 x 139 = 19321 pixels.

- Surface totale : **822.93** μm^2
- Surface du Silicium : **103.22** μm^2
- Surface du Cuivre : **630.46** μm^2
- Surface des Cristallites : **89.25** μm^2

Conclusion :

Environ 11% (10.84%) de la surface est formée de cristallites. D'autre part, la surface de Silicium découvert est environ égale à celle des cristallites formés; il en résulte que l'épaisseur des cristallites est environ double de celle de la couche de Cuivre, car le Cuivre est très peu soluble dans le Silicium.

5.3 Dimensions des cristallites

Dans ce qui suit, nous nous intéresserons aux formes et dimensions des cristallites. Les opérations se feront donc sur l'image binaire de la figure 5.6, obtenue par seuillage à deux niveaux de gris de l'image de la figure 5.5. Cette opération permet de ramener, les niveaux de gris des régions 1 et 2 de l'histogramme de la figure 5.4 au niveau du noir, et les niveaux de gris de la région 3 au niveau du blanc.

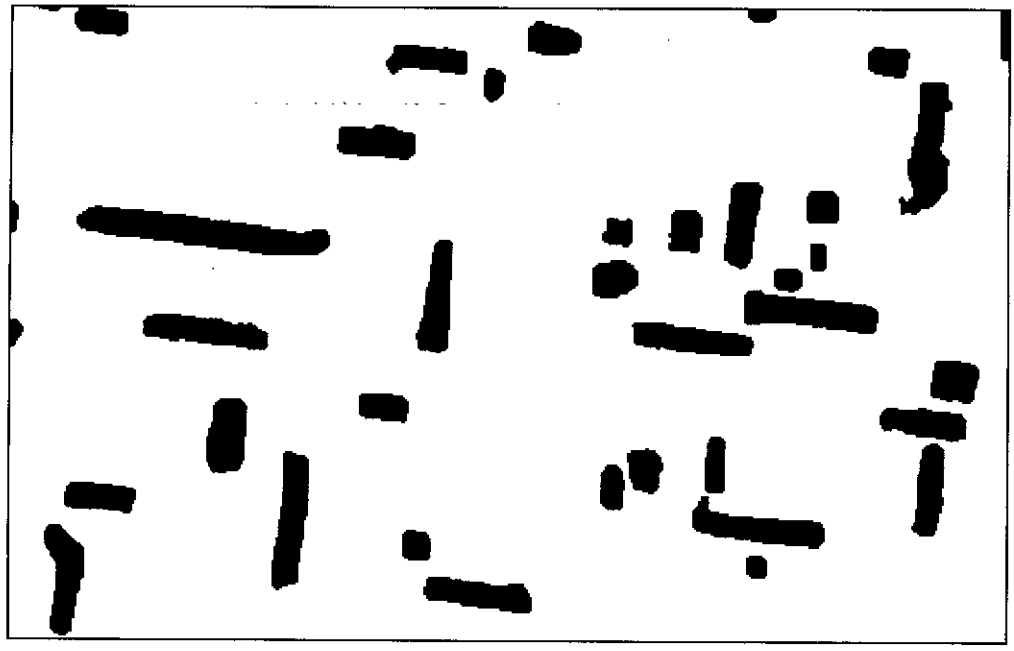


Fig. 5.6 Image binaire représentant les cristallites.

Nous avons utilisé, par la suite, un algorithme qui permet de compter le nombre de cristallites, ainsi que d'évaluer leurs surfaces. Le résultat du comptage donne un nombre total de cristallites égal à 39 et qui se répartissent comme suit :

- 8 cristallites carrés (à 8 pixels près)
- 31 cristallites rectangulaires.

L'histogramme des surfaces des cristallites est illustré sur la figure 5.7.

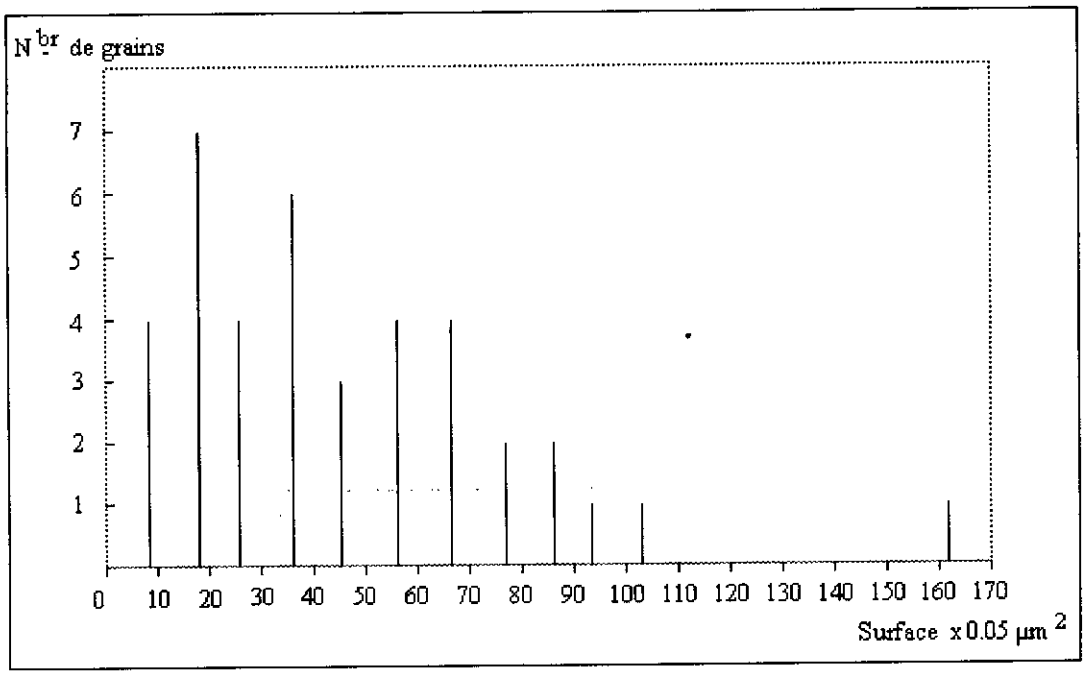


Fig. 5.7 Histogramme des surfaces des cristallites.

Conclusion :

L'histogramme de la figure 5.7 permet d'extraire les indications suivantes :

- La surface minimale d'un cristallite est d'environ 0.5 μm^2.
- Les surfaces maximales varient de 5 à 8 μm^2.
- La surface la plus probable : 1 μm^2.

5.4 Orientation des cristallites

Pour déterminer l'orientation des cristallites, on réalise une opération de squelettisation, qui permet de manipuler non pas tous les points constituants de l'objet mais une représentation en lignes de celui-ci (squelette). Cette opération sera suivie de deux opérations d'érosion : verticale et horizontale. L'érosion verticale de l'image aura

pour effet de préserver les squelettes des cristallites qui se sont développés suivant la direction verticale. L'érosion horizontale de l'image aura pour effet de préserver les squelettes des cristallites qui se sont développés dans la direction horizontale.

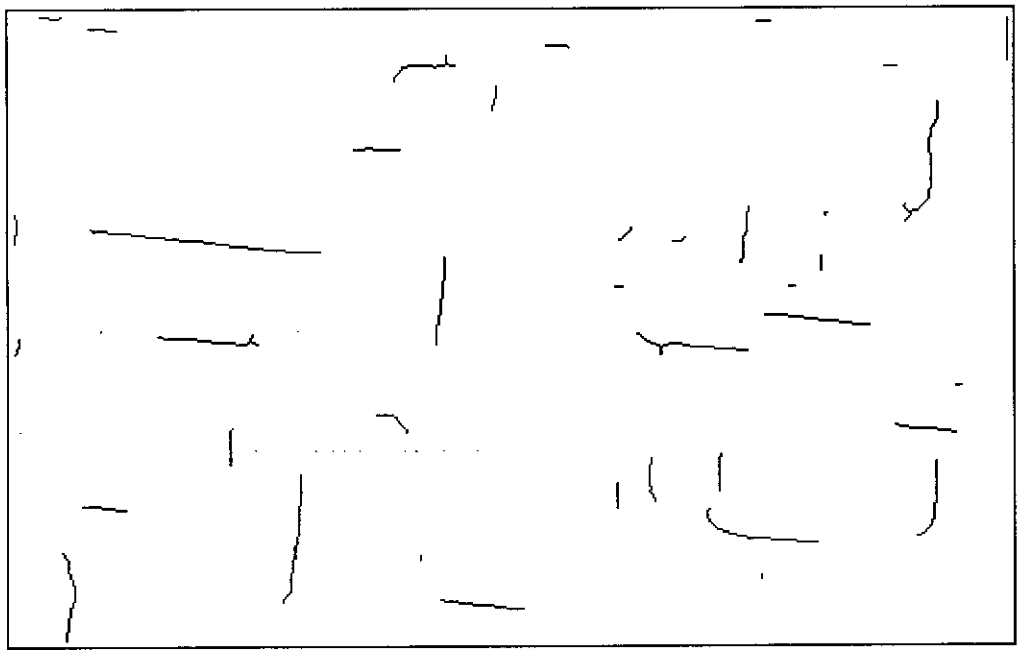


Fig. 5.8 Résultat de la squelettisation opérée sur l'image de la figure 5.6.

La figure 5.8 montre le résultat de la squelettisation opérée sur l'image binaire de la figure 5.6. On remarque que les éléments ne sont pas parfaitement horizontaux ou verticaux. Donc, avant de procéder aux opérations d'érosion, il est nécessaire de ramener les squelettes à des formes régulières (rectangulaires ou carrées) qui les englobent (figure 5.9). Une autre opération de squelettisation est réalisée pour obtenir enfin des squelettes orientés dans l'un ou l'autre sens (figure 5.10). La première opération d'érosion (en utilisant un élément structurant vertical) donne le résultat de la figure 5.11. Le comptage d'objets indique 17 cristallites dans le sens vertical. La seconde opération d'érosion (horizontale) donne le résultat de la figure 5.12. Le comptage d'objets indique 20 cristallites dans le sens horizontal.

Conclusion :

D'une part, l'orientation des cristallites suivant deux directions privilégiées perpendiculaires montre que les cristallites sont épitaxiés sur le substrat 100. D'autre part, si on enlève les cristallites carrés, il apparaît qu'il y a à peu près autant de cristallites épitaxiés suivant les deux directions : ce qui est normal.

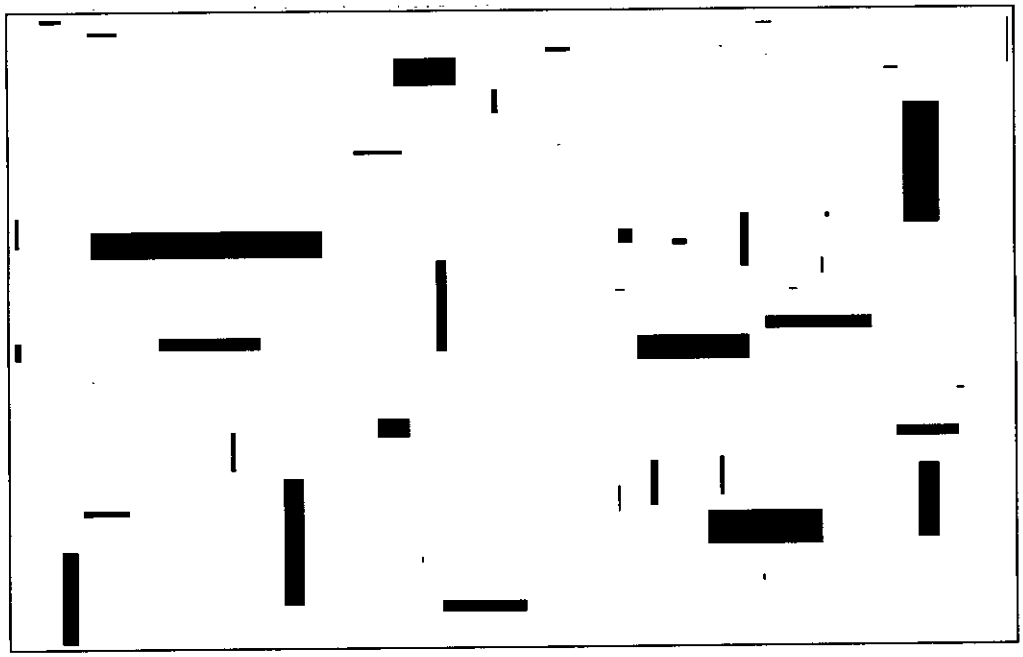


Fig. 5.9 Transformation en formes régulières.

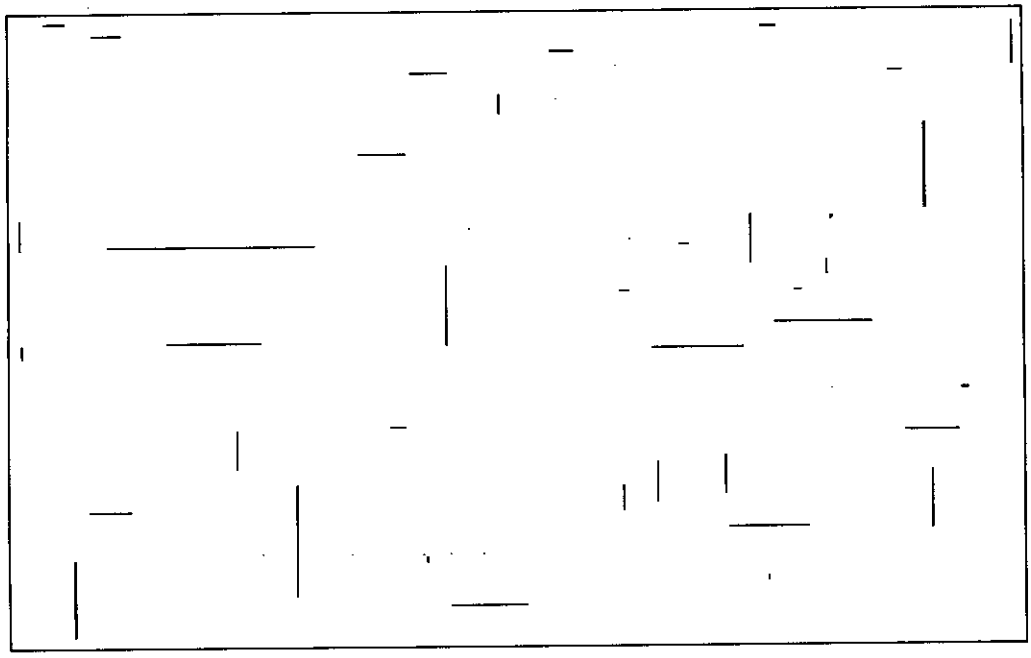


Fig. 5.10 Squelettes des objets de l'image de la figure 5.9.

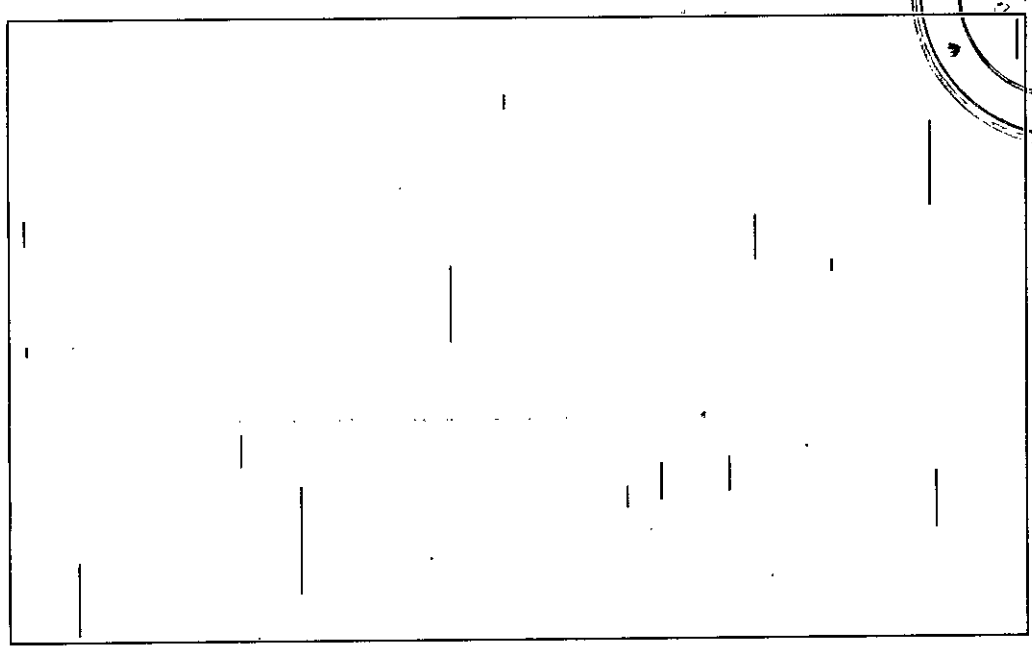


Fig. 5.11 Résultat de l'érosion verticale.

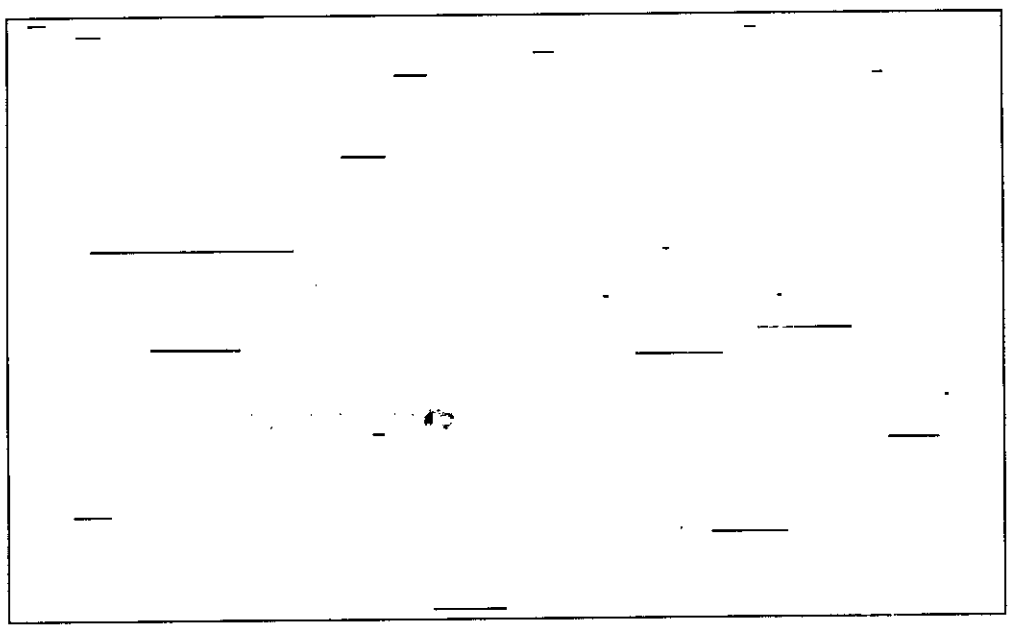


Fig. 5.12 Résultat de l'érosion horizontale.

5.5 Conclusion

L'application du traitement d'images à l'analyse des surfaces de Silicium métallisées s'est révélée intéressante. Cependant, le fait que les images soient tridimensionnelles pose le problème des effets d'ombre, qui se traduit par des chevauchements des niveaux de gris du Cuivre et ceux des cristallites. De plus, pour pouvoir tirer le meilleur profit des traitements d'images, il serait plus intéressant d'avoir un système d'acquisition d'images qui évite le passage par les photos et leur numérisation souvent source de problèmes.

CONCLUSION

Dans ce travail, nous nous sommes proposés de paralléliser des algorithmes de traitement d'images sur un réseau de Transputers. Pour ce faire, nous avons d'abord implanté les algorithmes fondamentaux (histogramme, filtrage et morphologie mathématique), grâce à un langage de programmation séquentielle (Turbo Pascal 6.0 de BORLAND), sur un système de traitement d'images à base d'un PC/AT et d'une carte d'acquisition d'images MATROX. Ces algorithmes ont été testés avec succès sur des images acquises à partir d'une caméra CCD.

Ensuite, nous avons parallélisé ces algorithmes séquentiels en vue de leur implantation sur un réseau à cinq (5) Transputers. Les algorithmes décrits en Occam et exécutés sous TDS ont donné entière satisfaction.

Enfin, pour valider les programmes et les tester, nous avons choisi une application qui consiste en l'étude de la morphologie des surfaces de Silicium métallisées. Ainsi, nous avons pu appliquer les différents traitements pour mettre en évidence la croissance épitaxiale du Cuivre sur le Silicium (100).

Par ailleurs, notre étude a donné un aperçu sur l'architecture des réseaux de Transputers ainsi que quelques notions sur la programmation en Occam. L'étude des

algorithmes usuels de traitement d'images a permis de développer un certain nombre d'outils qui constituent des prétraitements nécessaires à toutes les applications de vision artificielle. Il nous a été également possible de développer des algorithmes de type image-liste exploités au niveau de notre application du traitement d'images.

Enfin, cette étude nous a permis de découvrir le Transputer en tant que processeur très performant. Elle nous permettra dans le futur de concevoir des systèmes d'acquisition de données performants à base de Transputers.

BIBLIOGRAPHIE

[1] K. Hwang, F. A. Briggs.
Computer architecture and parallel processing.
Mc Graw-Hill, 1984.

[2] A. Belaid.
Méthodes et outils de programmation de systèmes de traitement d'images.
Thèse de Doctorat. INP. Lorraine, Juin 1987.

[3] *The transputer applications notebook. Architecture and software.*
INMOS, May 1989.

[4] *The transputer applications notebook. Systems and performances.*
INMOS, June 1989.

[5] *The transputer developement system. Delivery manual.*
Prentice Hall, 1988.

[6] J. Boreddy, A. Paulray.
On the performance of transputer arrays for dense linear systems.
Parallel computing. Vol. 15, No 1-3, September 1990.

[7] *Parallélisation de l'algorithme du lancé de rayons.*
Mémoire d'ingénieur. ENSPS. Strasbourg.

[8] *Transtech TMB 04 user manual.*
Transtech, 1991.

[9] D. Pountain.
A tutorial introduction to occam programming.
Billings & Sons, 1987.

[10] Y. Smara.
Conception et réalisation d'un système de traitement numérique d'images.
Thèse de Magister. USTHB. Alger, Juin 1985.

[11] W. K. Pratt.
Digital image processing.
Wiley-Interscience, 1978.

[12] R. C. Gonzales, P. Wintz.
Digital image processing.
Addison-Wesley, 1987.

[13] R. Chellapa, A. Sawchuk.
Digital image processing and analysis. volume 1 : Digital image processing.
IEEE Computer Society, 1985.

[14] H. R. Myler, A.R. Weeks.
Computer imaging recipes in C.
Prentice Hall, 1990.

[15] M. Coster, J. L. Chermant.
Précis d'analyse d'images.
Presses du CNRS, 1989.

[16] A. Belaid, Y. Belaid.
Reconnaissance de formes. Méthodes et applications.
Inter-Edition, 1992.

[17] R. A. Hummel.
Histogram modification techniques.
Computer vision, graphics and image processing, Vol. 4, 1975.

[18] F. Preteux.
Eléments de morphologie mathématique.
Cours de DEA de génie biologique et médical. Université Paris XII, 1992/1993.

[19] D. L. McCubbrey, R. M. Lougheed.
Morphological image analysis using a raster pipeline processor.
IEEE, 1985.

[20] R. T. Chin, Hong-Khoon Wan, D. L. Stover, R. D. Iverson.
A one pass thining algorithm and its parallel implementation.
Computer vision, graphics and image processing, Vol. 40, No 1, October 1987.

[21] R. Dapoigny.
Traitement d'images et architectures parallèles.
Addison-Wesley, 1995.

[22] N. Tawbi.
Parallélisation automatique : Estimation des durées d'exécution et allocation statique de processeurs.
Thèse de Doctorat. Université Paris VI, Septembre 1991.

[23] U. Banerjee, R. Eigenmann, A. Nicolau, D. A. Padua.
Automatic program parallelization.
IEEE. Vol. 81, No 2, February 1993.

[24] C. P. Ravikumar, A. K. Cupta.
Genetic algorithm for mapping tasks onto a reconfigurable parallel processor.
IEE Proceedings. Computer & digital techniques. Vol. 142, No 2, March 1995.

[25] A. C. Downton, R. W. S. Tregidgo, A. Cuhadar
Top-down structured parallelisation of embedded image processing applications.
IEE proceedings Image and signal processing, Vol. 141, No 6, December 1994.

[26] *IMS C 012 data sheet.*
INMOS, 1987.