Etude d'architectures VLSI numériques parallèles et asynchrones pour la mise en oeuvre de nouveaux algorithmes d'analyse et rendu d'images

Frédéric Robin

TELECOM PARIS

Thèse

présentée pour obtenir le grade de docteur

de l'Ecole nationale supérieure des télécommunications

Spécialité : Electronique et Communications

Frédéric Robin

Etude d'architectures VLSI numériques parallèles et asynchrones pour la mise en œuvre de nouveaux algorithmes d'analyse et rendu d'images

ENST 97 E 021

Soutenue le 27 octobre 1997 devant le jury composé de

Daniel Mlynek	Président
Jean-Michel Jolion	Rapporteurs
Alain Mérigot	
Alain Martin	Examinateurs
Yves Mathieu	
Gilles Privat	
Marc Renaudin	

Ecole nationale supérieure des télécommunications

Thèse

présentée par

Frédéric Robin

pour obtenir le grade de

Docteur de l'Ecole Nationale Supérieure des Télécommunications

Spécialité : Electronique et Communications

Etude d'architectures VLSI numériques parallèles et asynchrones pour la mise en œuvre de nouveaux algorithmes d'analyse et rendu d'images

Soutenue le 27 octobre 1997 devant le jury composé de

Daniel Mlynek	Président
Jean-Michel Jolion	Rapporteurs
Alain Mérigot	
Alain Martin	Examinateurs
Yves Mathieu	
Gilles Privat	
Marc Renaudin	

Thèse préparée au Centre National d'Etudes des Télécommunications de Grenoble

à mes parents

à mes grands-parents

Remerciements

Le travail présenté dans cette thèse a été effectué au Centre National d'Etudes des Télécommunications de Grenoble (CNET Centre Norbert Ségard).

Je tiens à remercier Monsieur Daniel Bois, directeur du CNET Grenoble, Monsieur Jean-Louis Lardy, chef du groupement CIT à mon arrivée, et Monsieur Patrice Senn, chef du département CTS (aujourd'hui laboratoire CET), pour m'avoir permis d'effectuer mon travail de thèse au CNET dans les meilleures conditions, et en particulier pour m'avoir accordé une prolongation d'un mois.

Je tiens tout particulièrement à remercier Monsieur Gilles Privat, ingénieur au CNET, et Monsieur Marc Renaudin, maître de conférences à l'E.N.S.T. de Bretagne, qui m'ont encadré tout au long de cette thèse. Je leur dois des voies de recherche originales et des discussions passionnantes, et les remercie pour tout le temps qu'ils ont bien voulu me consacrer. J'espère pouvoir participer de nouveau avec eux à des projets de recherche aussi intéressants.

Merci à Monsieur Christian Roux, professeur à l'E.N.S.T. de Bretagne, d'avoir accepté d'être le directeur initial de cette thèse.

Je remercie Monsieur Daniel Mlynek, professeur et directeur du laboratoire C3i à l'E.P.F.L., de m'avoir fait l'honneur de présider mon jury de thèse.

Je tiens à remercier Monsieur Jean-Michel Jolion, professeur à l'I.N.S.A. de Lyon, et Monsieur Alain Mérigot, professeur à l'Université Paris Sud, d'avoir accepté d'être les rapporteurs de cette thèse.

Je remercie Monsieur Yves Mathieu, directeur de recherches et responsable du département Electronique à l'E.N.S.T., d'avoir accepté de participer à mon jury.

Je remercie spécialement Monsieur Alain Martin, professeur au California Institute of Technology, qui a effectué le voyage des Etats-Unis jusqu'à Grenoble pour participer à mon jury de thèse et nous présenter ses derniers travaux, qui constituent pour moi une référence unique dans le domaine des circuits asynchrones.

Je tiens enfin à remercier Nadia Van Den Bossche, qui a effectué son stage de DEA au CNET en 1995, et toutes les personnes du CNET qui m'ont aidé lors de la conception du circuit AMPHIN. Je salue tous mes collègues et amis, avec qui j'ai passé des années inoubliables, et leur exprime toute ma sympathie.

Résumé

Le développement des applications de communication visuelle numérique a été rendu possible grâce au succès des normes de compression d'images animées apparues au début des années 90, comme H.261, MPEG1&2, puis H.263. La plupart des applications vidéo actuelles ou envisagées à court terme se basent sur ces normes. Or les techniques de codage d'images, aussi bien que le contexte des applications de communication visuelle, ont considérablement évolué: interactivité, flexibilité, scalabilité, codage basé-objets ou basé-modèles. L'objectif de la future norme MPEG4 est d'établir un nouveau standard de codage prenant en compte ces évolutions et donnant accès aux fonctionnalités correspondantes.

Une brève introduction au codage avancé d'images permet d'entrevoir l'évolution de la puissance de calcul et de la généricité requises pour l'implémentation de cette nouvelle génération de systèmes. L'analyse, la compression, la synthèse de contenus hybrides image/graphique doivent être supportées conjointement par les mêmes architectures matérielles. La présentation de l'évolution des architectures VLSI dédiées à leur mise en oeuvre suscite une réflexion sur les limitations et les perspectives de conception des "processeurs multimédia". Alors que la mise en oeuvre matérielle des premières normes s'est appuyée sur les progrès des technologies microélectroniques, l'implémentation temps réel et faible coût des applications de deuxième génération pose des problèmes qui remettent en cause les choix d'architecture VLSI. Cette thèse propose de combiner le parallélisme massif et l'asynchronisme à grain fin pour apporter de nouvelles perspectives de conception d'algorithmes et d'architectures numériques pour ces applications, pouvant pleinement tirer partie des futures technologies d'intégration.

Une introduction aux différentes notions d'asynchronisme, aux niveaux langage, algorithme, architecture, circuit VLSI, permet de mieux cerner leur sens et les potentiels qu'elles offrent.

L'application d'un asynchronisme fonctionnel à une classe de traitements d'image de bas niveau, basés sur la morphologie mathématique, est le point de départ d'une expérience de conception d'un réseau VLSI cellulaire asynchrone, qui aboutit à la réalisation complète d'un coprocesseur spécifique comprenant 800.000 transistors en technologie CMOS 0.5µ. Cette étude montre comment le concept d'asynchronisme peut être exploité à différents niveaux, élargissant ainsi le spectre de solutions pour la conception conjointe d'algorithmes et d'architectures intégrées.

La combinaison du parallélisme et de l'asynchronisme est finalement généralisée à la définition d'une architecture de coprocesseur programmable pour l'analyse/rendu d'images. L'évaluation de plusieurs primitives algorithmiques, exploitant de manière originale les concepts d'asynchronisme et de contrôle mixte SPMD / cellulaire / associatif / flot de données, illustre comment les architectures cellulaires peuvent exploiter de manière efficace le parallélisme potentiel de nouvelles classes d'algorithmes a priori irréguliers, grâce au relâchement des contraintes de synchronisation et de séquencement, tout en possédant des propriétés structurelles en adéquation avec l'évolution de la technologie.

TABLE DES MATIERES

INTRODUCTION

Nous assistons depuis quelques années à une évolution importante des pratiques de production et consommation d'informations audiovisuelles. Internet, multimédia, réalité virtuelle, interactivité, des leitmotive contemporains dont on ne cerne pas toujours bien le sens, encore moins les fondements. C'est en fait la conjonction des évolutions concurrentes dans les domaines du traitement du signal (au sens large, incluant l'ensemble des techniques liées au son, à l'image, et aux transmissions), des télécommunications, de l'informatique, et de la micro-électronique, qui donne naissance à ces nouveaux supports et applications. C'est la convergence de ces évolutions qui permettra leur développement à grande échelle.

Dans cette thèse, nous nous intéressons en particulier aux évolutions des architectures VLSI numériques, domaine qui s'appuie à la fois sur la micro-électronique et l'informatique, en relation avec l'évolution des techniques de codage d'images et du contexte des applications de communication visuelle. L'objectif n'est pas de trouver des solutions architecturales à des problèmes algorithmiques particuliers. Il s'agit plutôt d'essayer de mieux cerner les évolutions précédentes en termes d'adéquation algorithme-architecture et de convergence, d'en entrevoir les limites (en termes de "scalabilité"), pour proposer de nouvelles directions (architecturales et algorithmiques) qui permettront de concevoir des systèmes tirant mieux parti des futures technologies microélectroniques, seules capables de supporter un niveau de parallélisme (ou plutôt "concurrence") suffisant pour la mise en oeuvre intégrée des systèmes de communication visuelle de deuxième génération.

Mais revenons à la notion de convergence. Elle s'applique aujourd'hui à différents domaines techniques et même applicatifs dans certains cas. La communication multimédia interactive se situe au coeur de la convergence de l'audiovisuel, de l'informatique et des télécommunications [Pere96] [Chia97] [Mess96]. Cela ne signifie pas que les télévisions, les ordinateurs et les téléphones vont servir à la même chose, mais que parce qu'ils s'appuient largement sur des techniques communes, on peut effectivement envisager dans de nouvelles applications une mise en commun ou intégration de plusieurs fonctionnalités

initialement distinctes, permettant par la même occasion d'optimiser les performances, la flexibilité, et le coût des nouveaux systèmes.

D. Messerschmitt explique dans [Mess96] qu'un facteur clé de l'évolution du rapport performance/coût dans les applications de télécommunications a été la programmabilité. L'implémentation d'une application passe en général par trois phases où la programmabilité joue un rôle croissant. Dans la première phase, l'application ne peut souvent être mise en oeuvre qu'en concevant un matériel spécifique, à cause des contraintes de performance et de coût associé. Plus tard, une implémentation programmable définie de manière logicielle devient réalisable, avec un coût de revient acceptable. L'efficacité de la réalisation demeure essentielle mais le gain en temps de conception contrebalance le surcoût dû au matériel programmable. Finalement, les avancées technologiques deviennent telles que l'implémentation purement logicielle devient un standard bon marché, car le coût des solutions spécifiques croît à cause de la baisse de leur volume de production et du coût de migration technologique. Cette dernière phase possède alors la propriété essentielle de permettre la modification des fonctionnalités dans le temps. Un même support (matériel) permet alors la mise en oeuvre de plusieurs applications. Cette possibilité est à la base d'une restructuration radicale de la spécification détaillée d'une application car on peut définir une hiérarchie de supports (matériels ou logiciels) programmables, dont les dépendances sont gérées exclusivement par l'intermédiaire d'interfaces normalisées, appelées API ("Application Programming Interface"), qui permet l'intégration de fonctionnalités et de média divers dans un même modèle. L'intégration verticale des applications tend ainsi à disparaître au profit d'une intégration horizontale des architectures de systèmes. Dans une intégration verticale, une infrastructure spécifique est utilisée pour la réalisation de chaque application. Au contraire, dans une intégration horizontale, toutes les applications sont spécifiées à partir de la hiérarchie de fonctionnalités normalisées, et peuvent donc utiliser la même infrastructure (Figure 1).

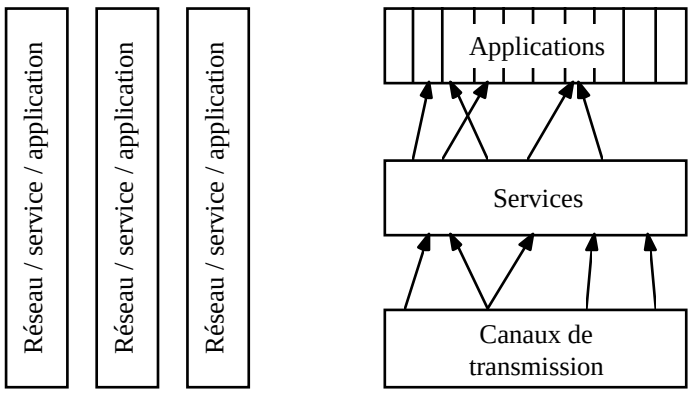

Figure 1: Exemple d'intégrations verticale et horizontale: deux modèles d'architecture pour la mise en oeuvre d'applications sur réseau [Mess96]

Les notions de convergence et d'intégration horizontale s'appliquent en fait de manière identique aux applications informatiques manipulant l'information visuelle (image ou graphique). De nombreuses API standard se sont plus ou moins imposées pour la programmation d'applications multimédia sur ordinateur personnel [Kala97] [Yao96a] [Yao96d].

Au travers d'une brève introduction au codage avancé d'images, nous montrons au chapitre 1 en quoi l'évolution de ces techniques et du contexte de communication visuelle amène à considérer une forme de convergence au niveau de la représentation même de l'information image, qui permet de définir la "communication multimédia" non seulement comme le transport mais aussi comme la restitution et la manipulation interactives et flexibles de contenus hybrides image/graphique. Il s'agit de la convergence "codage / analyse / rendu" (ou encore "compression / traitement / synthèse") dans le domaine du traitement (au sens large) de l'image.

Nous nous intéressons ensuite au chapitre 2 à la mise en oeuvre matérielle des techniques de codage, analyse et rendu d'images, et présentons un certain nombre de classes d'architectures VLSI qui ont été développées jusqu'à maintenant. Nous montrons l'évolution de ces architectures, située comme dans [Mess96] dans une problématique de performance/programmabilité, et qui aboutit aux "processeurs multimédia" apparus récemment. Nous concluons ce chapitre par une réflexion sur les limitations de ces approches et sur certaines perspectives pour la conception des futures générations architecturales. Nous proposons en particulier dans cette thèse de combiner le parallélisme massif et l'asynchronisme à grain fin pour élargir le spectre de conception d'algorithmes et d'architectures numériques pour la communication visuelle, et mieux tirer parti des capacités d'intégration croissantes des technologies VLSI.

Le chapitre 3 constitue une introduction aux différentes notions d'asynchronisme, aux niveaux langage, algorithme, architecture, conception et circuit VLSI, qui permet de mieux cerner leur sens et les potentiels qu'elles offrent, et d'en esquisser une synthèse à travers les notions de dépendances et de synchronisations.

L'application d'un asynchronisme fonctionnel à une classe de traitements d'image de bas niveau, basés sur la morphologie mathématique et utilisés pour la segmentation, est le point de départ d'une expérience de conception d'un réseau VLSI cellulaire asynchrone, présentée au chapitre 4, qui aboutit à la réalisation complète d'un coprocesseur spécifique - le circuit AMPHIN. Cette étude montre comment le concept d'asynchronisme peut être exploité à différents niveaux, apportant ainsi de nouvelles solutions pour la conception conjointe d'algorithmes et d'architectures intégrées.

La combinaison du parallélisme et de l'asynchronisme est finalement généralisée au chapitre 5 à travers la proposition d'un modèle de réseau VLSI cellulaire asynchrone programmable, pouvant servir à la définition d'une architecture de coprocesseur pour l'analyse/rendu d'images. L'évaluation de plusieurs primitives algorithmiques, exploitant de manière originale les concepts d'asynchronisme et de contrôle

mixte SPMD / cellulaire / associatif / flot de données, illustre comment les architectures cellulaires peuvent exploiter de manière efficace le parallélisme potentiel de nouvelles classes d'algorithmes a priori irréguliers, grâce au relâchement des contraintes de synchronisation et de séquencement, tout en possédant des propriétés structurelles en adéquation avec l'évolution de la technologie.

Chapitre 1

EVOLUTION DU CODAGE D'IMAGES

Le développement des applications de communication visuelle numérique comme par exemple la visioconférence, la vidéo à la demande, le "multimédia" sur CD-ROM, a été rendu possible principalement grâce au succès des normes de compression d'images animées apparues au début des années 90. On peut citer en particulier les normes H.261, MPEG1, MPEG2, et plus récemment H.263 [Rijk95]. La plupart des applications vidéo actuelles ou envisagées à court terme se basent sur ces normes. Or les techniques de codage d'images ont considérablement évolué et dépassent largement le cadre de la "simple" compression basée sur le codage hybride par blocs. L'objectif de la norme MPEG4 en cours d'élaboration [IEEE97] [IMCO97a] [IMCO97b] vise ainsi à établir un standard de codage universel pour différentes formes de données audiovisuelles, appelées objets audiovisuels (AVO), donnant accès à de nouvelles fonctionnalités, en particulier l'interactivité et la scalabilité "basés objet" ou "basés contenu". Nous essaierons au terme des deux premiers chapitres de montrer en quoi les traitements nécessaires à la mise en oeuvre de ces nouvelles fonctionnalités remettent en cause les choix d'architecture VLSI. Pour cela, nous présentons d'abord, de manière non exhaustive, un certain nombre de techniques de codage "avancé" d'images.

1.1 Le codage d'images de première génération

1.1.1 Rappels

A titre de référence, rappelons rapidement les principes de codage d'images classique, que l'on peut aujourd'hui appeler "de première génération". La compression vidéo tire partie de deux caractéristiques des séquences d'images animées: la redondance spatiale et la redondance temporelle

(d'où le nom de codage "hybride"). La redondance temporelle exprime le fait que dans une séquence continue d'images, l'information change très peu entre deux images successives. On peut essayer de supprimer cette information redondante en modélisant le changement, puis en appliquant une technique de prédiction. On ne transmet alors que l'erreur de prédiction, et éventuellement les paramètres de la prédiction obtenus par le codeur, pour éviter au décodeur de faire le même calcul ou si on utilise un schéma non causal. La redondance spatiale exprime le fait qu'une image possède en elle-même des propriétés de corrélation très importantes: si nous sommes capables de reconnaître des éléments dans une image, c'est que chacun de ces éléments est constitué de zones de taille significative comportant une certaine "homogénéité" au sens large, qui va par exemple de la couleur constante à la "texture" complexe mais répétitive localement.

Ces deux principes sont relativement généraux et seront utilisables pour toutes les futures approches de codage. Cependant les normes de codage actuelles définissent les techniques précises de décodage vidéo, ce qui impose quasiment les techniques de codage. Bien que laissant peu de place à l'extensibilité, cette approche a permis une familiarisation rapide des utilisateurs à ces normes à l'époque de leur création, qui a débouché sur leur application massive. Ces techniques sont basées sur le traitement de blocs carrés de 8*8 ou 16*16 pixels. La redondance temporelle est minimisée par une estimation de mouvement par appariement de blocs ("block matching") et la compensation de mouvement associée (par translation de blocs). La redondance spatiale est minimisée par une transformée en cosinus discrète (DCT), appliquée à chaque bloc de l'erreur de prédiction, suivie par une quantification adéquate et un codage entropique à longueur variable (VLC). Nous ne donnerons pas davantage de détails sur les normes utilisées, car ces principes suffiront à la comparaison avec de nouvelles structures, et surtout parce que ces normes sont maintenant très répandues. Citons tout de même quelques références: [Arav93], [Chen93], [Dufa95], [Héno93], [Jain81], [LeGa91], [Netr95], [Okub95], [Rena90], [Rijk95], [Schä95].

1.1.2 Limitations

Ces techniques de codage, très utilisées et relativement efficaces en termes de compression, possèdent néanmoins un certain nombre de limitations. Le premier défaut apparent concerne la qualité de l'image décodée, en particulier à bas débit [Ebra95] [Li95] [Yoko95]. Il apparaît ainsi un "effet de blocs" et un "effet moustique" ("blocking and mosquito artefacts") [Oste94a] [Hött94]. Les effets de blocs correspondent à des discontinuités entre blocs de pixels. Ils sont dus au découpage artificiel de l'image en blocs, et au fait que l'on traite les blocs comme des paquets de données indépendants entre eux, en particulier avec un niveau de quantification défini localement. L'effet moustique se manifeste par des "contours fantômes" autour des contours principaux qui séparent les éléments de l'image. Ils sont dûs à une quantification trop "forte" à l'intérieur d'un bloc possédant des variations "rapides" d'intensité, qui

élimine des composantes de hautes fréquences essentielles à la restitution de ces variations. Dans les deux cas, les artéfacts sont dus à l'inadéquation du modèle utilisé pour l'image: le découpage en blocs carrés est purement artificiel et le codage de chaque bloc ne tient pas compte de son contenu en termes d'éléments de l'image mais simplement en termes de statistiques locales. Ce modèle correspond en fait à une approche de type "traitement du signal" ("waveform coding"), où les techniques de base sont définies pour des signaux stochastiques. Par exemple le critère d'erreur pour un bloc est en général une moyenne quadratique des différences locales, ce qui ne correspond pas à un critère de qualité visuelle. Ces artéfacts peuvent être plus ou moins supprimés dans un premier temps à l'aide de techniques de filtrage.

La limitation majeure est due au niveau de représentation de l'information image: le type de données traité est le pixel, c'est-à-dire le plus bas niveau de représentation d'une image numérique. Cela signifie qu'une image est considérée comme un ensemble de pixels indifférenciés, que l'on va coder suivant une statistique de niveau correspondant. Comme la norme spécifie les traitements particuliers à appliquer aux données reçues, l'extensibilité d'un tel schéma est très réduite. Mais surtout, l'information transmise au décodeur n'est véhiculée que dans un but de décompression et d'affichage direct (la visualisation se faisant au niveau pixel, on transmet une information de même niveau), sans aucune possibilité de manipulation, d'interaction avec l'utilisateur.

1.2 Introduction aux codages de deuxième génération

Les codages que l'on peut appeler "de deuxième génération" [Kunt85] cherchent donc à dépasser ces limitations, d'une part en véhiculant de l'information de niveau de représentation plus élevé que celui de blocs carrés de pixels, d'autre part en intégrant des techniques multiples dans un même modèle de flexibilité accrue. Certaines approches consistent à améliorer la qualité des techniques classiques, en tirant mieux partie des propriétés statistiques ou en utilisant des techniques adaptatives. Les approches véritablement novatrices cherchent à analyser le contenu des images en termes de régions homogènes (selon différents critères) ou de modèles d'objets [Li94]. Le niveau de représentation et les possibilités de manipulation du contenu des images, en termes d'objets, si possibles en relation avec un niveau sémantique, peuvent alors offrir de nouvelles fonctionnalités qui dépassent le simple cadre de la compression de données [Bove96].

1.2.1 Amélioration des techniques classiques

1.2.1.1 Techniques de traitement du signal

Bien que les nouvelles techniques de traitement du signal basées pixel ne permettent pas à elles seules d'obtenir des types de représentation de niveau plus élevé, il est utile de prendre en compte les

améliorations qu'elles apportent, du point de vue de l'universalité et de l'évolutivité que devraient présenter un système de codage avancé.

Il ne faut pas oublier que la transformée en cosinus discrète n'a pas été choisie parce qu'elle était la technique la plus performante, mais parce qu'elle présentait le meilleur rapport qualité/complexité au moment de l'élaboration des premières normes. D'autres techniques telles que la transformée en ondelettes, la décomposition en sous-bandes, ou la quantification vectorielle ont permis d'améliorer la qualité et le taux de compression des systèmes de codage d'images. Nous n'allons pas décrire ici ces différentes techniques mais renvoyons le lecteur aux références suivantes: [Come93], [Forc89], [Jaya92], [Kunt85], [Kunt87], [Li97].

1.2.1.2 Partitionnements adaptatifs

Pour diminuer les défauts des systèmes de codage par blocs, et aussi améliorer leurs performances en termes de qualité et compression, la première étape qui vise à affiner le modèle de l'image, consiste à partitionner l'image de manière adaptative, c'est-à-dire à regrouper les pixels non plus par blocs de taille fixe, mais par zones géométriques simples dont on adapte localement un paramètre de taille, ou de position, de manière à traiter des blocs de l'image dimensionnés en fonction de leur corrélation propre. Ceci constitue une étape intermédiaire vers la segmentation en régions de forme quelconque. Les différents types de partitionnements, classés par contraintes décroissantes, sont basés sur le carré, le quadtree, le rectangle, le triangle, les polygones convexes [Davo96].

a) Partitionnement en quadtree

Le "quadtree" ou arbre quaternaire est un découpage hiérarchique par blocs carrés. A partir d'un découpage uniforme par blocs de taille fixe, on autorise la subdivision d'un bloc en quatre sous-blocs de taille identique, si les pixels contenus dans le bloc d'origine ne présentent pas une corrélation considérée comme suffisante. Le découpage possède donc une forme d'adaptation aux données [Dufa95].

Ce découpage est encore fortement contraint car les tailles et les positions des blocs obtenus ne peuvent être égales qu'à un nombre restreint de valeurs, qui dépendent de la méthode de partitionnement et non des données contenues dans l'image.

b) Triangulation

Un partitionnement qui respecte mieux la distribution des données est le partitionnement par triangulation. Il s'agit toujours de rassembler les pixels selon une forme géométrique élémentaire, mais avec plus de degrés de liberté: l'image peut être découpée en un ensemble de triangles sans aucune contrainte sur les positions des sommets, donc sur la taille, la position et l'orientation des structures de base.

La compensation de mouvement peut alors inclure le déplacement des sommets des triangles et donc la déformation de triangles [Aiza95] [Dudo96] [Li93], ce qui permet de prendre en compte des

mouvements autres que la simple translation d'éléments de l'image, comme la rotation ou le changement d'échelle.

Un algorithme de partitionnement assez connu est la triangulation de Delaunay [Dudo96], à laquelle on peut associer les méthodes de codage fractal [Davo95] [Davo96]. Ces dernières ont aussi été étendues au partitionnement polygonal, avec les diagrammes de Voronoï [Robe97].

1.2.2 Codages par analyse-synthèse

Les techniques de codage que l'on peut véritablement qualifier de "deuxième génération", au sens de leurs objectifs en termes de représentations en non pas au sens de l'évolution des techniques mathématiques, sont basées sur l'analyse des images et se rapprochent donc du domaine de la vision par ordinateur. De manière duale, le décodage correspondant se fait par synthèse de l'image à partir de la représentation d'encodage, et peut faire appel aux techniques de rendu graphique pour les "images de synthèse", bien connues par exemple dans les domaines de la conception assistée par ordinateur, des trucages et des jeux vidéos, de la visualisation scientifique et des applications de réalité virtuelle.

L'analyse de l'image consiste à la subdiviser en un ensemble d'objets en mouvement et de décrire chaque objet par un ensemble de paramètres que l'on peut classer en trois sous-ensembles, qui définissent respectivement le mouvement, la forme et la couleur (ou "texture") de l'objet. L'analyse dépend d'un ou plusieurs modèles qui définissent la nature des objets à extraire. Les ensembles de paramètres peuvent être codés de manière indépendante pour chaque objet, et en fonction du mode de codage choisi (qui définit par exemple les types et niveaux de distorsion acceptables). Ces paramètres encodés constituent les données qui sont transmises. Les images sont reconstruites par synthèse à partir de ces paramètres, au niveau du décodeur et aussi au niveau du codeur pour connaître l'erreur de codage et exploiter la redondance temporelle (comme dans les systèmes classiques). La Figure 2 présente un schéma d'ensemble d'un codeur "basé objet" (ou "orienté objet") par analyse-synthèse [Musm89].

On peut distinguer deux grandes classes de techniques d'analyse pour les codages basés objets: celles qui sont basées sur une segmentation de l'image en régions, au sens spatial ou spatio-temporel [Wu96], et celles qui cherchent à faire une correspondance entre l'image et un modèle d'objets de plus haut niveau. Nous ne nous intéressons pas ici aux schémas complets de codage mais présentons plutôt les principes de base de tels schémas. Des études de schémas complets sont présentées par exemple dans [Casa94], [Gerk94], [Hött90], [Hött94], [Li95], [Musm89], [Oste94a], [Oste94b], [Sale95a], [Sale97], [Will91], [Wu96], [Yoko95].

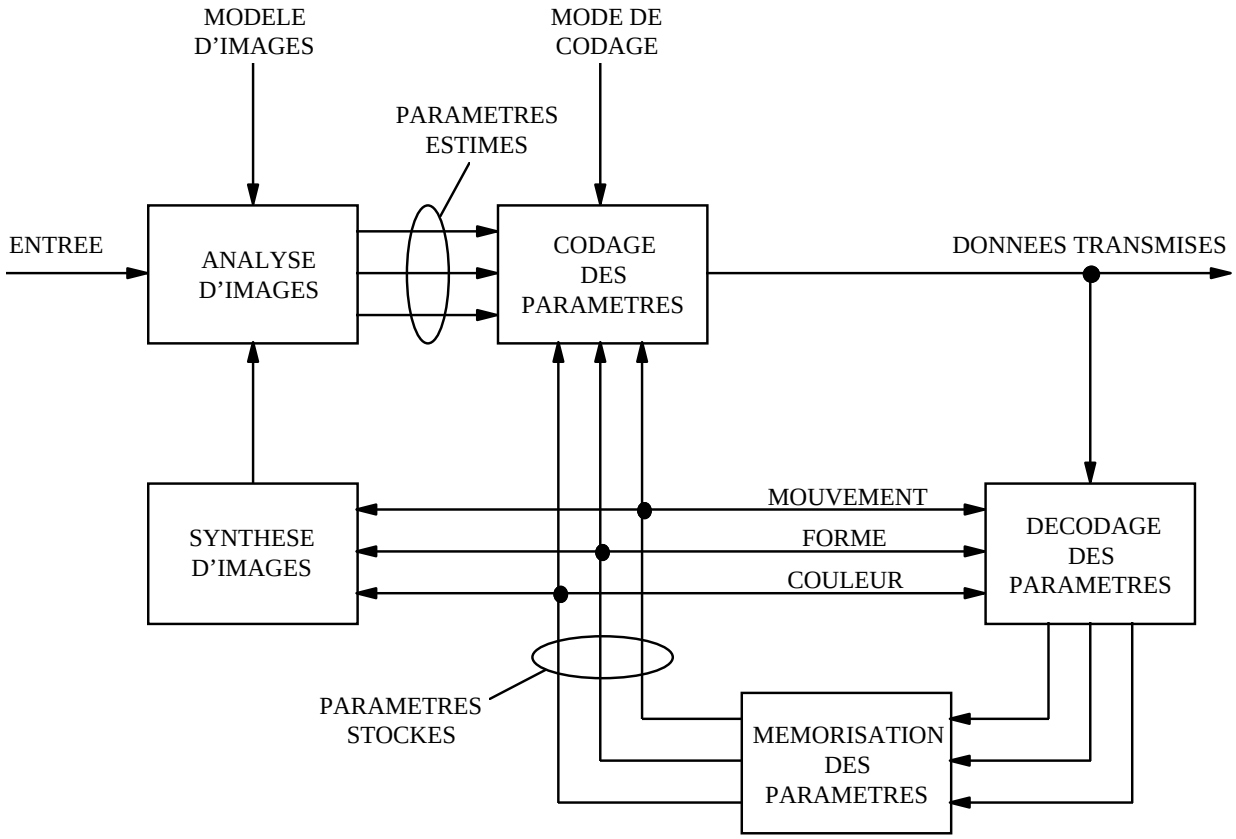

Figure 2: Schéma d'un codeur basé objet par analyse-synthèse [Musm89]

1.2.2.1 Codages basés régions / basés segmentation

Les codages basés segmentation cherchent à découper ("segmenter") une image ou un groupe d'images en un ensemble de composantes disjointes (on obtient ainsi une "partition") de manière à ce que chaque composante possède certaines propriétés d'homogénéité psychovisuelle. Ces idées sont apparues sous le nom de codages "contour/texture" [Kunt85]. La notion de segmentation est très générale et on définit un type de segmentation en spécifiant les propriétés recherchées ainsi que les critères de décision utilisés pour déterminer la partition. Par exemple, on distingue dans un premier temps les segmentations par la couleur, par le contraste ou le contour, par la "texture", par la forme, par le mouvement, ..., ainsi que les segmentations idéales "par sémantique" qui détermineraient une partition correspondant à une interprétation humaine d'une scène en termes d'objets matériels. Ces différents types de segmentation ne sont pas indépendants les uns des autres mais correspondent aux différentes "tactiques" algorithmiques d'analyse. De plus, il existe des techniques très variées pour la mise en oeuvre de chacun de ces types (cf. [Cort95], [Li95], [Will91], [Wu96], [Yoko95],...).

De manière générale, on cherche, comme dans les techniques de codage de première génération, à exploiter les corrélations spatiales, temporelles, ou spatio-temporelles des images, mais sans a priori sur

la position des pixels que l'on va regrouper. La seule segmentation ne constitue pas un codage de l'image et il faut donc utiliser des techniques appropriées au codage de la partition obtenue, en particulier pour coder la forme, le contenu et le mouvement de chaque région [Come93] [Hött90] [Hött94] [Musm89] [Schi93].

a) Segmentation en régions 2D, 2.5D, 3D

Nous nous contenterons ici de donner les caractéristiques générales de quelques approches utilisées dans les schémas de codage "basés segmentation" ou "basés régions". On peut distinguer des types de schémas de codage en fonction de la dimensionnalité des régions obtenues, qui sont soit à deux dimensions (2D), soit à trois dimensions (3D), soit dites à "deux dimensions et demi" (2.5D). Les segmentations 2D fournissent une partition en régions planes pour chaque image, à partir ou non de plusieurs images voisines dans le temps. Les segmentations 3D sont moins utilisées et fournissent une partition "volumique" d'une séquence d'images [Pard94] [Sale94] [Will91]. La demi-dimension supplémentaire des techniques "2.5D" correspond à l'ajout d'une valeur de profondeur à chaque région 2D d'une partition, qui permet une représentation par plans superposés [Bove96] [Lava94].

Les algorithmes de segmentation sont variés mais on peut citer les deux grandes classes d'algorithmes appelées segmentation par croissance de régions ("region-growing") et segmentation par division-fusion ("split and merge") [Come93][Cort95][Kunt85][Kunt87][Will91][Yoko95]. Les techniques de croissance de régions consistent à choisir un ensemble initial de pixels ou de régions auxquels on va rattacher de proche en proche les pixels voisins à partir d'un critère d'appartenance. Ce sont des approches ascendantes ("bottom-up") qui partent d'informations purement locales pour effectuer le regroupement ("clustering") des pixels. Elles sont en général peu robustes dans le sens où le résultat global peut être très sensible aux variations locales. Les techniques de division-fusion consistent à subdiviser d'abord l'image en zones suffisament petites pour qu'un critère d'homogénéité soit respecté dans chacune d'elle, puis à regrouper certaines des zones entre elles en fonction d'un autre ensemble de critères, de manière à ajuster le nombre de classes de la partition en fonction de l'application. Ce sont des approches mixtes "top-down" puis "bottom-up": la partition initiale comprend une seule classe contenant l'image entière, puis cette partition est affinée en fonction d'informations globales et enfin la dernière phase regroupe les régions obtenues. Elles sont peu sensibles aux variations locales mais ne sont pas toujours satisfaisantes car le résultat dépend fortement des méthodes de division qui sont assez arbitraires par rapport au contenu de l'image car elles utilisent des critères statistiques globaux. Il existe bien sûr des approches hybrides qui essayent de combiner l'ensemble des informations locales avec des mesures globales, de manière ascendante, descendante ou mixte.

b) Exemple : utilisation d'opérateurs de morphologie mathématique

Les techniques de segmentation sont à la base de toute analyse d'images et seront donc essentielles dans le développement de nouvelles applications de communication visuelle. Elles font donc partie des

techniques dont nous avons essayé de cerner les caractéristiques principales, de manière à avoir une "vision conjointe algorithme-architecture" suffisamment large pour proposer de nouvelles directions. Nous nous sommes intéressés en particulier à une classe d'algorithmes de segmentation basés sur la morphologie mathématique. Nous introduisons donc ici ces techniques, dont on discutera l'implémentation dans les chapitres 4 et 5.

La morphologie mathématique est une théorie géométrique et ensembliste qui a été développée en partie pour répondre aux problèmes de l'analyse d'images [Serr86] [Serr89]. Les opérateurs d'érosion et de dilatation sont bien connus pour le traitement d'images binaires, ainsi que les notions d'enveloppe convexe ou encore de squelette. Ce type de traitements peut être généralisé pour les images en niveaux de gris ("greyscale mathematical morphology") [Naka78] [Ster86], et on peut définir des "opérateurs en niveaux de gris" à partir d'opérateurs binaires grâce à une décomposition par seuillage des images par exemple [Sale95b] [Sale96] [Vinc93].

Des algorithmes de segmentation basés sur des opérateurs de morphologie mathématique en niveaux de gris sont présentés dans [Casa94] [Lant82] [Meye90] [Pard94] [Sale92] [Sale94] [Sale95a] [Sale95b] [Sale96] [Sale97] [Vinc91]. Ce sont des approches mixtes où l'on utilise à la fois des informations locales et non-locales. Les opérateurs qui sont au coeur de ces approches sont "les filtres morphologiques par reconstruction" et "la ligne de partage des eaux" ("watershed").

Les filtres d'ouverture ou fermeture par reconstruction font partie des opérateurs connexes [Sale95b] [Sale96]. Ce sont des opérateurs non linéaires, la dénomination de filtre morphologique se basant sur les propriétés de croissance (au lieu de la linéarité pour les filtres classiques) et d'idempotence [Serr86]. Ce sont en fait des généralisations des opérateurs de reconstruction (ou "propagation") binaire [Pres84] [Vinc93]. Ils permettent de simplifier les images en créant des "zones plates" dont on peut contrôler par exemple la taille, tout en préservant l'information de contour de l'image originale, contrairement aux filtres linéaires ou aux filtres d'ouverture/fermeture simples [Sale92] [Sale95b] [Sale96]. Ils constituent la première phase de la segmentation et permettent "l'extraction de marqueurs" à partir desquels on déterminera la partition finale de l'image (Figure 3).

(a) (b) (c)

Figure 3: (a) Image originale, (b) Ouverture-fermeture par reconstruction, (c) Image segmentée

L'approche classique de segmentation morphologique consiste à déterminer "la ligne de partage des eaux" du gradient de l'image à partir des marqueurs obtenus préalablement [Jack96] [Lant82] [Meye90] [Sale92] [Sale96] [Vinc91]. La ligne de partage des eaux définit un "bassin d'attraction" ou "bassin versant" pour chaque minimum marqué du gradient. On peut en effet considérer une image en niveaux de gris comme un relief, où l'altitude de chaque pixel est égale à son niveau de gris. Les minima du gradient correspondent à des zones dont le niveau de gris varie peu et on cherche alors à déterminer les lignes de crête de ce gradient, qui correspondent aux contours des différentes zones de l'image (Figure 4).

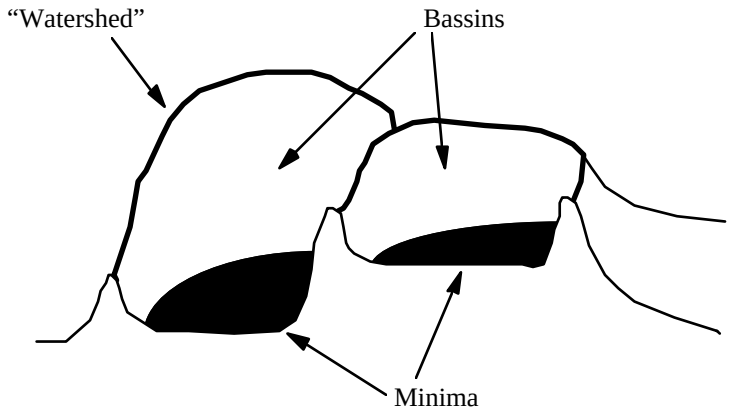

Figure 4: Ligne de partage des eaux [Vinc91]

On peut imaginer pour cela une méthode "d'immersion" ou "de montée des eaux", qui consiste à remplir progressivement d'eau les bassins marqués et à construire des barrages lorsque des eaux d'origines différentes vont se mélanger, jusqu'à ce que la surface soit complètement immergée. Les barrages reposent sur les lignes de crête recherchées et forment les contours de la segmentation, les différents bassins constituant la partition de l'image (Figure 5).

L'algorithme du "watershed" peut aussi être utilisé directement sur l'image plutôt que son gradient, dans une version de type croissance de régions [Sale96], et peut être appliqué à un groupe d'images successives considéré comme un ensemble à trois dimensions pour réaliser une segmentation spatio-temporelle [Pard94] [Sale94].

Ces traitements peuvent être écrits sous une forme parallèle régulière et itérative mais sont alors très inefficaces. Des algorithmes séquentiels à base de files de type FIFO ("First In First Out") sont présentés dans [Sale96] [Vinc91] [Vinc93] et sont eux très efficaces puisqu'ils tentent de n'effectuer que les calculs qui correspondent aux propagations effectives des traitements dans l'image. Ils sont par contre séquentiels et irréguliers par nature. Nous montrerons au chapitre 5 qu'ils peuvent être efficacement parallélisés sur l'architecture proposée.

Ces algorithmes sont simples au sens où ils ne manipulent que des entiers de faible dynamique et peuvent être écrits assez naturellement sous une forme cellulaire, où les traitements se font par propagations locales. Cette simplicité, alliée à une bonne robustesse fonctionnelle comparativement à des méthodes plus classiques de segmentation, ainsi qu'aux possibilités de traitement hiérarchique et interactif, font de ces algorithmes de bons candidats pour la réalisation de systèmes de codages basés régions.

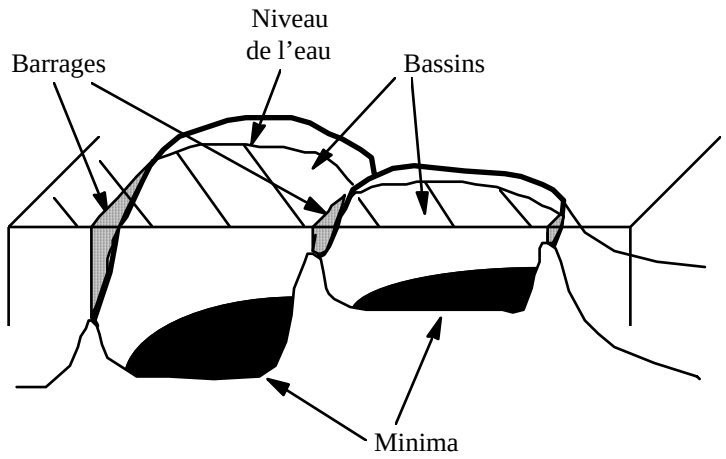

Figure 5: Construction du "watershed" par immersion [Vinc91]

c) Codage de forme

Une fois déterminée la partition de l'image en régions, il faut coder la forme de chacune de ces régions. On peut séparer les techniques de codage de forme en deux classes: les codages "par l'intérieur" et les codages "par le contour" [Come93] [Eden85].

Une première technique de codage par l'intérieur semble avoir été retenue comme algorithme de base dans la norme MPEG4: il s'agit de la "composante alpha" ("alpha channel"). Nous aurions pu la citer dès le paragraphe 1.2.1 puisqu'elle est la seule technique à ne pas proposer de niveau de représentation plus élevé que "le niveau pixel". Elle consiste à transmettre un bloc de pixels formant soit un masque binaire qui indique l'appartenance à la région des pixels correspondants des autres composantes, soit une composante de profondeur indiquant la priorité de la région dans un modèle 2.5D, soit encore une composante de transparence [Blin94] [Port84].

Ce masque binaire peut être analysé ou transformé de manière à obtenir une représentation de plus haut niveau. Une méthode qui a été beaucoup étudiée en morphologie mathématique est la décomposition en motifs élémentaires. Il s'agit de trouver un pavage si possible minimal (disjoint ou recouvrant) de la forme à partir d'un ensemble restreint de surfaces de base [Jean96] [Nils97] [Pita90] [Rein96] [Rons91]. Dans le cas d'une famille de carrés de taille décroissante, on peut alors ne conserver

que les positions du "centre" des carrés ainsi que leur taille, et on obtient ce qui est appelé "l'axe médian" [Rose82]. Des représentations voisines sont basées sur les "squelettes" [Mara86], que l'on obtient par exemple par "amincissement" ("thinning") [Rose82]. Citons en particulier les squelettes géodésiques présentés dans [Brig95] [Sale96].

Les codages de forme par le contour les plus connus sont basés sur "les codes chaînés" ("chain codes" ou "codes de Freeman"), les approximations polygonales, et les courbes paramétrées. Le "chain code" consiste, en parcourant le contour, à noter la succession des déplacements entre pixels [Eden85] [Marq93] [OGor88]. On peut ensuite appliquer un code de type "run-length" ou basé sur les chaînes de Markov et un code entropique, ou encore arithmétique par bloc [Kim91]... Le codage par approximation polygonale transforme le contour en une suite de segments de droite [Heck97] [Oste94a]. Enfin, on peut aussi utiliser des courbes paramétrées de type polynomiale ou "spline", dont il faut quantifier les paramètres.

Les techniques présentées ci-dessus sont initialement définies pour du codage de type "intra", c'est-à-dire sans référence à une image précédente. Les schémas de codage basés régions doivent bien sûr exploiter la redondance temporelle et utilisent donc des techniques différentielles ou par compensation de mouvement [Gu95] [Heck97] [Hött94] [Wu96].

d) Codage de texture

Le contenu d'une région, c'est-à-dire les valeurs des pixels appartenant à cette région, est aussi appelé "texture" car chaque région est considérée comme un objet. Alors que la DCT est bien adaptée aux blocs carrés, les codages basés segmentation doivent traiter des régions de forme quelconque. Un certain nombre d'études ont cherché à se ramener à ce type de transformée. On peut d'abord citer l'utilisation de méthodes de remplissage (appelées "padding" [Siko97] ou extrapolation [Kata97]), qui consistent à choisir la valeur des pixels n'appartenant pas à la région, de manière à augmenter la corrélation du bloc. Ce remplissage peut se faire en propageant et en moyennant les valeurs des pixels situés en bordure de la région [Siko97] ou en prenant la valeur moyenne des pixels appartenant à la région [Kata97]. On peut ensuite utiliser une DCT [Siko97] ou une transformée en ondelettes [Kata97]. Une autre technique consiste à changer les fonctions de base de la transformée pour s'adapter à la forme de la région. Ces algorithmes sont appelés SADCT ("Shape-Adaptive DCT") [Come93] [Oste94a] [Siko95a] [Siko95b] [Siko95c]. Citons aussi l'utilisation de la transformée de Karhunen-Loeve [Care97], ou la combinaison DCT/DPCM [Schi93].

Des techniques moins classiques sont basées sur l'échantillonnage non-uniforme et l'interpolation linéaire ou non-linéaire. Elles consistent à déterminer de manière adaptative un sous-ensemble de pixels de l'image, à partir desquels on la reconstruit par interpolation directe ou itérative. En particulier, des opérateurs de morphologie mathématique ont été proposés pour cela dans [Sale96] et dans [Sapi94] où l'on utilise des squelettes en niveaux de gris.

Les modèles paramétriques de textures permettent d'élever encore le niveau de représentation. Il s'agit par exemple d'obtenir les coefficients d'une fonction de surface de type polynomiale, ou encore d'extraire certaines caractéristiques de la texture en estimant des fonctions de signature [Pika97]. Le codage laisse de nouveau la place à la notion d'analyse-synthèse, et aussi à l'intégration de fonctionnalités, avec par exemple les représentations multirésolutions, qui permettent plusieurs accès à l'information de texture avec différentes résolutions spatiales [Ofek97] [Pika97].

1.2.2.2 Codages basés modèles

a) Analyse et rendu basés modèles

Les codages basés modèles représentent une étape supplémentaire dans la hiérarchie des niveaux de représentation des images [Pear95], par rapport à l'analyse en termes de régions. Il s'agit véritablement d'analyser les images par des techniques empruntées à la vision par ordinateur, en "reconnaissant" un ensemble d'objets dont on possède ou dont on construit un modèle structurel et auquel on associe des paramètres de mouvement, de forme et de texture [Lava94], puis de restituer les modèles obtenus par des techniques de synthèse ou rendu ("rendering") d'images par ordinateur ("computer graphics") [Akim93]. Des techniques de suivi permettent de faire évoluer le modèle pour qu'il imite l'évolution des objets qu'il représente [Aiza95] [Lava94] [Li94] [Pear95] [Wels90]. Dans certaines applications, il suffit de transmettre les paramètres d'évolution du modèle, alors que pour d'autres applications nécessitant un rendu plus conforme à la scène réelle, on transmet aussi une information résiduelle correspondant à l'erreur de modélisation [Pear95].

Les modèles peuvent être partiellement définis avant analyse dans des applications particulières, comme la visiophonie, où l'on cherche par exemple à faire correspondre un modèle tridimensionnel de visage à l'image puis à en adapter les différents paramètres caractéristiques, comme la position et le mouvement global de la tête, le mouvement de la bouche, etc..., [Aiza95] [Forc89] [Jaya92] [Lava94] [Li93] [Li94] [Musm89] [Wels90]. On parle alors parfois de codage "basé connaissance" ("knowledge-based coding") ou "basé sémantique". De telles applications où l'on dispose effectivement d'une grande quantité d'information a priori sont très spécifiques mais permettent en principe d'atteindre de très forts taux de compression. Dans des cas plus généraux, le modèle doit être complètement construit au fur et à mesure de la séquence d'images.

Les termes de codage "basé modèles" ou "basé objets" sont aussi utilisés de manière très générale pour englober quasiment tous les types de codage de deuxième génération, incluant les codages basés régions qui peuvent être vus comme l'utilisation de modèles d'objets 2D à mouvement 2D. Le schéma de la Figure 2 est d'ailleurs valable dans le cas général. Il correspond en fait à l'intégration du codage basé modèles aux techniques de codage prédictif [Bove96] [Gerk94] [Hött90] [Hött94] [Musm89] [Oste94a] [Oste94b]. Il ne semble pas y avoir de consensus très précis sur la terminologie des nouveaux systèmes

de codage. Certaines classifications font intervenir les termes de codages de "zéroième" génération à cinquième génération (codage "sémantique") [Pear95].

b) Modèles de mouvement

On peut aussi classer les schémas de codage en fonction des modèles de mouvement et de déformation utilisés pour l'analyse et le codage des objets. Ces objets peuvent en effet être considérés comme non-déformables (ou "rigides") [Musm89] ou déformables (ou "flexibles") [Hött94] et leur mouvement peut être modélisé soit dans le plan soit dans l'espace (mouvements 2D et 3D), en incluant seulement les translations ou en considérant au contraire l'ensemble des transformations affines [Hött94] [Li93] [Li94] [Oste94a] [Oste94b] [Wels90] [Yoko95].

L'estimation de mouvement dans les schémas de codage basés modèles et basés objets est un sujet tout aussi essentiel que la segmentation, mais que nous ne développerons pas ici. En ce qui concerne le modèle d'objets 2D avec mouvements 2D (pour le codage basé régions), il est possible de généraliser les techniques par blocs utilisées dans les systèmes de première génération, en tenant compte de l'appartenance ou non de chaque pixel du bloc à la région [Siko97].

1.3 Convergence algorithmique pour les applications de communication visuelle

1.3.1 Au delà de la compression

Les techniques de codage basées analyse-synthèse ont d'abord été étudiées en vue d'améliorer les taux de compression. Le processus de normalisation MPEG4 a d'ailleurs été initié dans un objectif de codage à très bas débit [Cort95] [Ebra95] [Li95] [Oddo93] [Pere96] [Yoko95] (ou, de manière équivalente, à très fort taux de compression), en particulier inférieur à 64 kbit/s. Cette norme potentielle a rapidement été réorientée vers l'incorporation de "nouvelles fonctionnalités" aux systèmes de codage, car on s'est rendu compte que les gains en compression obtenus par de nouveaux schémas de codage n'atteindraient pas à court terme l'ordre de grandeur espéré et ne justifiaient pas à eux seuls la création d'une nouvelle norme (au moment même où les normes précédentes commençaient à être exploitées à grande échelle), et que les techniques véritablement novatrices n'étaient pas suffisament mûres. Il s'agissait donc plutôt de mettre en place un cadre de codage extensible et flexible, où de nouvelles techniques pourraient être incorporées a posteriori. Cette idée a eu des conséquences et des développements importants, qui ont conduit au concept de "méta-norme".

Alors que les normes précédentes spécifient à un niveau très fin le format du flux de données transmis ("bitstream") et la succession de traitements composant le système de décodage, une norme "universelle" devrait être suffisament générique pour pouvoir inclure des algorithmes de traitement non encore spécifiés. Le contenu de la norme change alors radicalement de niveau: celle-ci doit indiquer la

manière de spécifier des traitements au lieu des traitements eux-mêmes. Il s'agit d'une première extension du principe de "spécification minimale", introduite en un certain sens dans les normes passées en ne spécifiant que le système de décodage et pas le système de codage, ce qui laisse des degrés de liberté pour améliorer la compression et/ou la qualité des données. Bien que l'extensibilité soit un aspect fondamental de la norme MPEG4, l'apport de cette norme est défini essentiellement en termes de fonctionnalités nouvelles, qui peuvent être réparties en trois classes, correspondant à l'amélioration de l'efficacité du codage, à l'universalité de l'accès, et à l'interactivité basée sur le contenu [Chia97] [Pere96].

L'amélioration de l'efficacité du codage est évidemment requise pour toute nouvelle norme de codage d'images. Outre les gains liés à l'intégration de nouvelles techniques de compression, une fonctionnalité intéressante est le codage de flux concurrents multiples, qui permet par exemple de véhiculer simultanément plusieurs points de vue d'une même scène.

La notion d'accès universel correspond à plusieurs formes d'indépendance entre les systèmes d'acquisition, les systèmes de transmission et les systèmes de restitution (les terminaux) vis-à-vis des caractéristiques de chacun d'entre eux. Tout d'abord, l'interopérabilité consiste à faire en sorte que des systèmes différents puissent convenir d'une configuration commune minimale pour communiquer. L'extensibilité doit permettre la mise à jour d'un système, et les possibilités de reconfigurations dynamiques permettent d'exploiter en temps réel ou presque cette extensibilité. Un terminal peut alors "télécharger" une configuration logicielle (voire matérielle) de manière à gérer des traitements inconnus jusque-là. Le caractère universel doit non seulement s'appliquer aux traitements mais aussi aux formats de données. On parle alors de "scalabilité" spatiale et temporelle lorsqu'il est possible d'accéder à plusieurs résolutions et fréquences d'images et de scalabilité basée sur le contenu lorsque cela peut s'appliquer de manière indépendante sur chacune des composantes du flux de données (par exemple, la scalabilité basée objet permet de choisir une résolution fine pour un objet en gros plan et une résolution faible pour l'arrière plan). La robustesse aux erreurs de transmission doit aussi tenir compte de la diversité des systèmes.

Enfin, les fonctionnalités qui se démarquent vraiment du simple codage sont liées à l'interactivité basée sur le contenu [Bove96] [Chia97], c'est-à-dire qu'elles permettent à l'utilisateur de rechercher et de manipuler l'information en utilisant la représentation de moyen ou haut niveau de celle-ci. L'accès aux données par le contenu comprend entre autres la recherche basée objet par indexation et la navigation basée objet par hyperliens... Par exemple, en "cliquant" sur un objet, on pourrait accéder à l'ensemble des scènes qui ont un lien quelconque avec cet objet. Ce sont enfin la manipulation et l'édition de haut niveau du flux de données, à partir de la représentation basée objets, qui permettront véritablement l'interactivité multimédia puisque l'utilisateur pourra non seulement sélectionner l'information mais aussi la modifier selon son goût, la réutiliser et l'enrichir. L'intégration d'objets synthétiques et naturels (SNHC,

"Synthetic-Natural Hybrid Coding") est essentielle à l'interactivité puisque l'utilisateur doit pouvoir communiquer avec son terminal par le biais de l'information vidéo, textuelle et graphique (2D et 3D). Elle est à la base des applications de réalité virtuelle (VR) et de "réalité augmentée". On peut ainsi penser à des interfaces permettant de modifier la couleur ou la forme des objets, de faire du "copier-coller" d'AVO, de changer le point de vue d'une scène, ou encore d'associer des comportements nouveaux aux objets... Notons que l'ensemble des techniques permettant d'assembler des objets audiovisuels peuvent être regroupées sous le terme de "compositage" (issu de "compositing" [Blin94] [Port84]). Un exemple intéressant d'application est la visioconférence à environnement virtuel (Figure 6), où l'on combine codages basés objets, SNHC et compositage, et interactivité basée contenu.

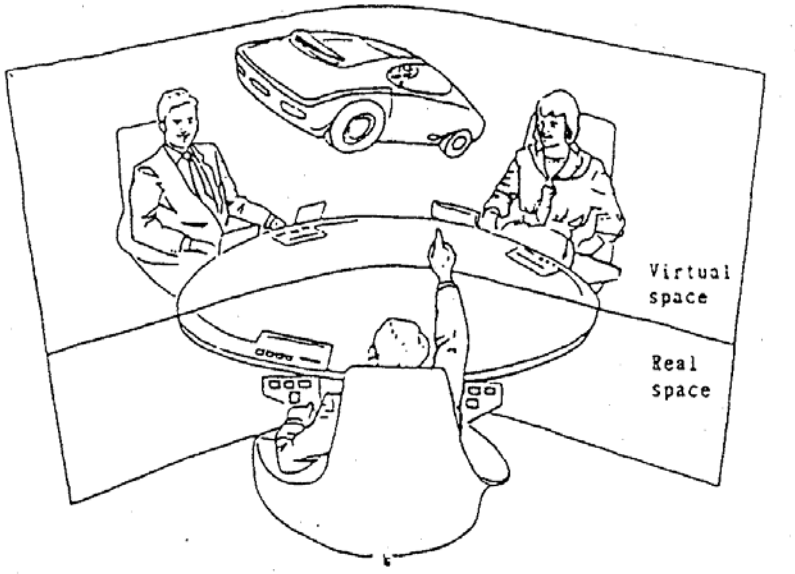

Figure 6: Illustration du concept de visioconférence à environnement virtuel [Aiza95]

1.3.2 Convergence algorithmique et intégration horizontale

Nous avons vu que le codage d'images classique, représenté par exemple par la norme MPEG2, va s'effacer devant le concept de "communication multimédia" [Bove96] [Chia97], que l'on a présenté ci-dessus sans le nommer, et qui est bien plus que la simple intégration de média. Nous allons enfin pouvoir faire le lien avec l'introduction de cette thèse dont le thème était la convergence et l'intégration horizontale. Il s'agit ici d'une convergence algorithmique: les techniques de codage, d'analyse et de rendu d'images, jusque là utilisées séparément dans des applications de transmission, de robotique et de production vidéo par exemple, seront utilisées conjointement dans les futurs systèmes de communication multimédia. C'est d'abord le besoin d'universalité qui nécessite l'intégration de ces différentes techniques. C'est surtout la mise en oeuvre de nouvelles fonctionnalités au delà de la compression, comme l'élévation

du niveau de représentation et l'interactivité basée contenu qui engendre cette convergence. Les algorithmes de codage par analyse-synthèse seront donc basés sur une association de "briques algorithmiques" variées, appelées "outils" dans le cadre de MPEG4. Comme mentionné plus haut, c'est la façon de combiner ces outils qui doit être normalisée et non le contenu algorithmique de ceux-ci. Il faut pour cela définir les interfaces entre les outils, les mécanismes pour les combiner ainsi qu'un mécanisme pour en télécharger de nouveaux. Il s'agit donc d'une interface ouverte. La "glu" qui devrait permettre de combiner les outils se situe à un niveau langage. Elle devrait être gérée par MSDL ("MPEG4 Systems Description Language") dans MPEG4 [Siko97].

La convergence se situe non seulement au niveau des outils, mais aussi au niveau de la structure des données que ces outils manipulent, en particulier les types et les niveaux de représentation des images. MPEG4 introduit la notion d'objet audiovisuel. Les outils devront manipuler de manière la plus transparente possible ces objets d'origines mixtes vidéo et graphique, de résolutions spatiales et temporelles multiples, et de formats variés. Les types de représentation correspondent aux différents modèles d'images, dont quelques uns ont été cités dans le paragraphe 1.2, comme le modèle de régions 2D de forme quelconque en mouvement de translation dans le plan. Ils présentent une hiérarchie partielle, les modèles incluant un mouvement 3D étant par exemple plus généraux que les modèles 2D, ce qui permet de définir des niveaux de représentation. Chaque type peut aussi inclure une classification en sous-types ainsi qu'une hiérarchie, comme par exemple pour le codage de formes de régions 2D où l'on pourrait classer les techniques par masque, par contour, par motifs, par squelette, etc..., suivant des critères de continuité, de scalabilité, ou de fonctionnalités de manipulation. Tous ces types et niveaux de représentation devront pouvoir être traités sous une forme unifiée et générique, à travers une interface, d'une façon similaire aux interfaces de programmation d'application (API, "Application Programming Interface") aujourd'hui répandues en tant que standard de fait pour les applications graphiques. Ces interfaces sont les vecteurs de la programmabilité, et véhiculent une couche de "middleware", que certains ont appelée "mediaware", puisqu'elle intégrait les structures des différents média. La notion d'intégration horizontale s'applique donc aussi à l'évolution du codage d'images.

Un point de vue intéressant consiste ainsi à considérer que l'on peut remplacer le schéma classique de codage/décodage "source", issu du traitement du signal, qui spécifie la chaîne de traitements appliqués au signal de bas niveau (Figure 7), par un schéma généralisé basé sur les représentations et les interfaces entre représentations et processus de traitement (Figure 8). La Figure 7 montre un schéma classique de "communication basée signal", avec un codage de type JPEG [Arav93] [Chen93] [Schä95] ou MJPEG. Le terme signal est employé ici dans le sens où il correspond à la représentation de plus bas niveau de l'image, c'est-à-dire le niveau échantillon ou pixel, que l'on peut qualifier de niveau 0. La Figure 8 montre au contraire un schéma de "communication multimédia", où l'analyse consiste à obtenir des objets (AVO) de niveaux de représentation plus élevés, et où le rendu manipule et transforme cette

représentation jusqu'à un niveau servant à l'affichage. Des interfaces génériques, reposant sur des API multiniveaux, elles-mêmes appelant des outils, remplacent l'enchaînement de traitements fixes et spécifiques.

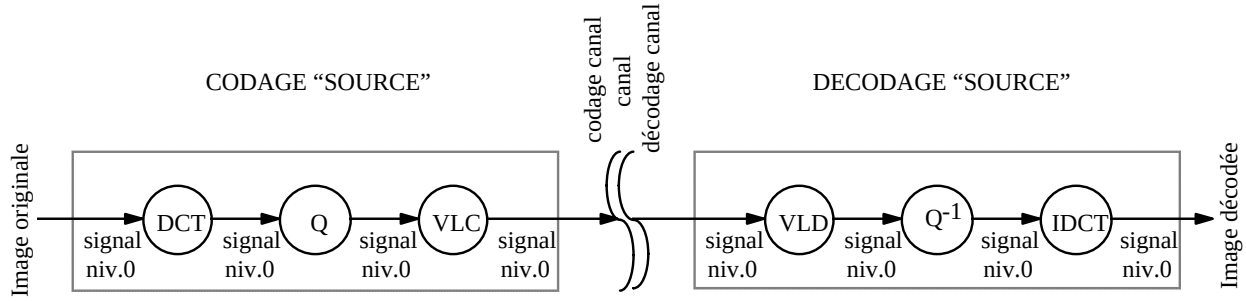

Figure 7: Communication basée signal

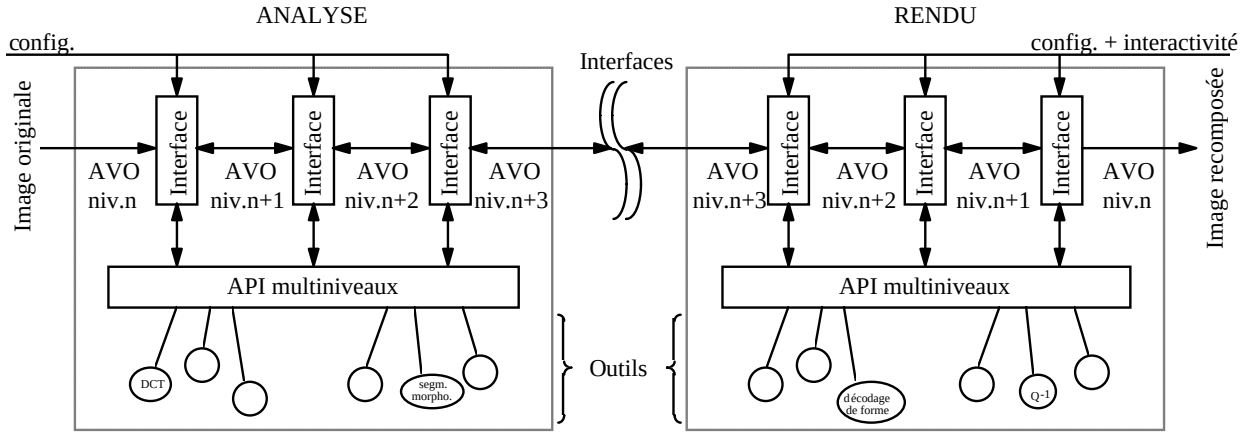

Figure 8: Communication multimédia

1.4 Conclusions sur l'évolution du codage d'images

De manière analogue à la convergence présentée dans l'introduction, on retrouve la problématique du niveau de programmabilité vis-à-vis de la complexité et de la mise en oeuvre en temps réel des systèmes de codage d'images. Essayons de dégager les caractéristiques principales de l'évolution du codage d'images en regard des problèmes de mise en oeuvre associés.

Nous avons introduit plusieurs classes de techniques qui sont au coeur des codages dits de deuxième génération. Nous n'avons pas décrit les algorithmes pour la mise en oeuvre de ces techniques, mais pouvons néanmoins affirmer que la complexité, en termes de nombre d'opérations élémentaires par unité de temps, des futurs systèmes de communication multimédia (au sens de [Chia97]), n'est pas du

même ordre de grandeur que celle des systèmes classiques [Netr95]. Le "simple" codage basé régions peut impliquer des algorithmes de segmentation et d'estimation de mouvement qui ne seront pas exécutables en temps réel dans un avenir proche sur des microprocesseurs à usage général, contrairement aux systèmes classiques, comme par exemple MPEG1 et MPEG2 [Netr95]. L'utilisation d'architectures "spécifiques" paraît donc incontournable.

La diversité et l'intégration des algorithmes est un facteur nouveau dans les systèmes de codage. Il s'agit d'utiliser conjointement et dynamiquement des algorithmes développés jusque-là dans trois domaines distincts: les systèmes de compression vidéo, les systèmes graphiques et de synthèse d'images, et les systèmes de traitements d'images et de vision par ordinateur. Le choix du niveau de programmabilité des différentes composantes d'un tel système n'est plus aussi clair que pour les systèmes classiques à cause de l'antagonisme entre performance et généricité. Des caractéristiques essentielles à prendre en compte pour les algorithmes apparaissant avec les systèmes de deuxième génération sont les niveaux de parallélisme et de régularité (ou d'irrégularité). Il est clair que la plupart des algorithmes liés à l'image présentent un parallélisme potentiel très important (en particulier à grain fin, c'est-à-dire sur les traitements s'appliquant aux représentations niveau pixel). La notion de régularité est plus vague. Elle dépend de la spécification de l'algorithme. On peut parler de régularité spatiale ou temporelle, sur les dépendances de données ou sur l'ordonnancement (ou "séquencement") choisi. Alors que les systèmes de première génération avaient une nette orientation "traitement du signal", c'est-à-dire à base d'opérateurs linéaires s'appliquant globalement et systématiquement à des ensembles réguliers de données, les algorithmes plus sophistiqués d'analyse d'images peuvent posséder des dépendances non-linéaires, dynamiques, globales et locales, s'appliquant à des structures de données irrégulières. Le seul passage du codage par blocs à une analyse en termes de régions de forme quelconque présente une irrégularité assez importante, tout en conservant un fort parallélisme à grain fin. Mais parallélisme massif et irrégularité ont souvent été présentés comme incompatibles. Nous présentons au chapitre suivant l'évolution des architectures VLSI pour les applications de communication visuelle et proposerons au chapitre 5 un modèle permettant de réconcilier performance et programmabilité, parallélisme et irrégularité.

Chapitre 2

EVOLUTION DES ARCHITECTURES VLSI POUR LA COMMUNICATION VISUELLE

2.1 De l'intégration verticale à l'intégration horizontale

Les applications de communication visuelle numérique se sont initialement basées sur la possibilité de stocker ou transmettre des images à partir d'un support à débit restreint. Le développement des techniques de codage d'images ou "compression vidéo", ainsi que leur normalisation, a été à l'origine d'un nouveau domaine d'application pour les circuits intégrés, qui ne cesse de se développer. Les systèmes de visioconférence, de diffusion de télévision numérique, ou de consultation sur CD-ROM, utilisent tous les mêmes principes de codage source, le codage hybride par blocs, combinant en particulier transformée en cosinus discrète et estimation/compensation de mouvement par appariement de blocs. L'importante quantité de calculs nécessaires à la mise en oeuvre en temps réel de ces systèmes a conduit au développement de jeux de circuits VLSI ("chipsets") spécialisés, puis de "processeurs vidéo", qui ont permis l'émergence rapide des applications basées sur le codage d'images.

Comme le chapitre précédent l'a montré, la communication visuelle est aujourd'hui envisagée sous une forme beaucoup plus évoluée, incluant des formes nouvelles de données, de type hybride "synthétique/naturel", et des fonctionnalités d'accès et manipulation basées-objet ou basées-contenu. Cette évolution repose en partie sur l'utilisation de techniques empruntées à deux autres domaines: l'analyse et la synthèse d'images. Ces deux domaines étaient jusqu'alors à la base d'applications disjointes. Comme pour le codage d'images, ils ont aussi la propriété de posséder des besoins intensifs en calcul, et ils ont de même conduit à différentes voies de développement d'architectures VLSI pour leur mise en oeuvre en temps réel. L'analyse d'images, utilisée pour des applications de reconnaissance de

formes ou de vision par ordinateur, a inspiré entre autres la conception des "array-processors" et des processeurs associatifs. La synthèse (ou "rendu") d'images, aujourd'hui répandue dans les systèmes de CAO et de simulation, utilisée pour la réalisation d'effets spéciaux, ou encore à la base des systèmes de réalité virtuelle, a conduit au développement de processeurs de rendu 3D et de mémoires à logique intégrée.

Les sections 2.2 à 2.4 présentent ces différentes catégories d'architectures VLSI pour la mise en oeuvre de primitives de traitement qui devront être supportées conjointement par les nouveaux systèmes de communication visuelle. Nous montrons l'évolution de ces architectures des coprocesseurs spécialisés aux processeurs programmables. La section 2.5 montre le début de convergence entre des architectures qui correspondaient auparavant à plusieurs domaines d'application distincts, en présentant l'apparition récente des "processeurs multimédia". Cette évolution est caractéristique du passage d'une intégration verticale vers une intégration horizontale de l'implémentation matérielle des systèmes de communication visuelle. Notons que la classification suivante ne peut pas être complètement "orthogonale", à cause de l'étendue des compromis possibles entre les différentes structures.

2.2 Architectures VLSI pour la compression d'images

L'implémentation des applications de compression vidéo fait largement appel à des architectures VLSI conçues spécifiquement pour la mise en oeuvre efficace des algorithmes sous-jacents, qui présentent des besoins intensifs en calcul [Pirs95] [Kons92] [IEEE92a] [Rena90]. La Figure 9 rappelle le schéma-bloc d'un codeur-décodeur MPEG-1 ou 2.

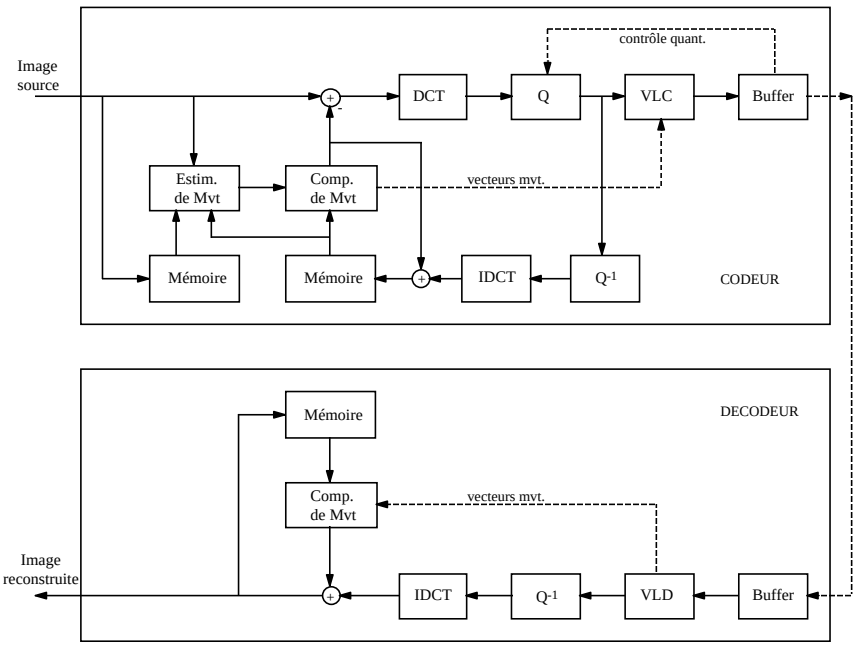

Figure 9: Schéma bloc d'un "codec" MPEG

Les blocs les plus "gourmands" en puissance de calcul sont ceux qui réalisent l'estimation de mouvement et les transformées en cosinus discrètes. Il existe plusieurs façons de décomposer le système de manière à faire apparaître un degré suffisant de concurrence, qui seul peut permettre une implémentation temps réel (i.e. respectant par exemple une fréquence de trame de 30Hz). Les différents blocs fonctionnels peuvent être directement mis en oeuvre par un ensemble de modules spécifiques, formant ainsi une structure en pipeline, avec un module pour chacune des fonctions de base: "block-matching", compensation de mouvement, DCT directe et inverse, quantification directe et inverse, codage et décodage entropique, multiplexage, etc... On peut aussi exploiter un parallélisme de données, en utilisant la division de l'image en macro-blocs, chaque processeur étant affecté à un ou plusieurs macro-blocs et devant effectuer l'ensemble des fonctions de l'algorithme. De plus, chaque fonction peut être réalisée à partir d'opérateurs VLSI spécialisés, ou au contraire peut être exécutée de manière programmable sur des unités de calcul génériques. Ces types de choix, que l'on retrouve d'ailleurs pour les architectures des autres domaines de traitement d'images, ont généré un large spectre de solutions, qui peuvent être comparées en fonction de leur débit maximal et de leur surface de silicium [Pirs95].

2.2.1 Jeux de circuits spécialisés

Dès que les systèmes de codage d'images animées comme H.261 ont été normalisés, la technologie permettant alors l'intégration d'un ou quelques blocs fonctionnels dans un circuit VLSI, toute une gamme de "jeux de circuits" ("chipsets") spécialisés sont apparus pour construire des systèmes de compression. Par exemple, fin 1990, LSI Logic proposait une famille de 7 circuits et GCA une famille de 12 circuits, constituant les briques de base de "codecs" vidéo spécifiques [Ang91]. Une dizaine de solutions commerciales (C-Cube, NEC, SGS-Thomson...) est aussi présentée dans [Kons92]. Comme les premières normes comportaient différentes options d'optimisation ou de configuration, et que de nouvelles normes (comme MPEG2) étaient en cours de préparation, le fait de disposer séparément de ces fonctions de base apportait une certaine flexibilité aux concepteurs de systèmes, qui pouvaient alors facilement réaliser ces différentes configurations, en y incorporant leur savoir-faire (les normes ne spécifiant que la partie décodage). Bien sûr, ces systèmes ont pu être intégrés sur un seul circuit par la suite, grâce à l'évolution permanente de la technologie.

La caractéristique des "chipsets" ou des "monochips" spécialisés est qu'ils correspondent uniquement à l'implémentation d'un algorithme particulier ou d'une configuration particulière d'une norme (qui peut proposer plusieurs modes comme les "profils" et les "niveaux" dans MPEG). Ils sont alors en général composés d'opérateurs spécialisés, qui peuvent être optimisés pour chaque implémentation, en ajustant par exemple le découpage structurel, la taille des chemins de données, le nombre de processeurs élémentaires, la taille des mémoires, la fréquence de fonctionnement, etc... Nous ne présentons pas ici les architectures possibles pour chacun des blocs, comme la DCT ou le "block-

matching". De nombreux travaux ont étudié leur conception jusqu'au niveau le plus fin [Pirs95] [Rena90].

2.2.2 Processeurs vidéo programmables

Le besoin d'expérimenter facilement différentes variantes de systèmes, de pouvoir intégrer les différents modes de fonctionnement spécifiés par une même norme, et de pouvoir faire évoluer une implémentation, a amené au développement de processeurs à la fois programmables et spécifiques à la compression vidéo. La programmabilité apporte bien sûr de nouveaux degrés de flexibilité, mais limite généralement les performances assez fortement vis-à-vis d'une solution spécifique. Ces architectures programmables ont pu tout de même remplacer assez rapidement les circuits spécifiques (sauf pour les configurations les plus complexes), grâce à l'évolution de la technologie qui a permis de presque doubler les performances des dispositifs à chaque nouvelle génération (i.e. environ tous les 18 mois...).

Les processeurs vidéo programmables ne constituent pas véritablement une classe architecturale bien définie. Cela est dû au fait qu'il existe tout un spectre de solutions concernant le partitionnement du système en unités de calcul, de mémorisation et de contrôle plus ou moins spécialisées. Nous pouvons tout de même partager ces processeurs en deux classes plus précises, les "coprocesseurs vidéo" et les "VSP".

Les coprocesseurs vidéo correspondent à une évolution directe des circuits de compression vidéo spécialisés. Ils disposent d'une programmabilité minimale permettant de sélectionner différents modes de fonctionnement ou paramètres de configuration du système (concernant par exemple le mode de prédiction, l'algorithme de "block-matching", la structure des groupes de trames, la résolution des images...). Ils sont constitués d'un ou plusieurs modules spécialisés, pouvant être directement issus des solutions utilisées pour les "chipsets" et se chargeant d'effectuer les tâches de calcul intensif, et d'un module programmable de contrôle (par exemple de type RISC), qui pilote les modules spécialisés et les flux de données qui circulent dans le système, et se charge d'exécuter de manière logicielle les parties moins contraintes et plus irrégulières des algorithmes. Dans certaines variantes, les modules spécialisés peuvent être composés de plusieurs ensembles d'unités arithmétiques banalisées (ALU, multiplieurs-accumulateurs), constituant une structure parallèle de type SIMD, microcontrôlée par un programme stocké dans une ROM par exemple. L'utilisation de tels opérateurs peut permettre de regrouper plusieurs des fonctions de base du système. Une variante encore plus flexible consiste à intégrer un coeur de DSP pour remplacer certains modules spécifiques. Plusieurs architectures de coprocesseurs vidéo sont décrites dans [Pirs95], comme AVP (AT&T), VDSP2, ou HVC. Citons aussi le processeur VRP (C-Cube) [Glas97], qui contient un coeur RISC 32bits, des unités travaillant sur quatre octets en mode SIMD, des unités spécialisées pour le codage à longueur variable et l'estimation de mouvement, et qui permet l'encodage MPEG-1 et MPEG-2.

La classe des "VSP" (Video Signal Processors) est constituée d'architectures encore plus flexibles, ne comprenant pas de modules spécifiques à une fonction particulière d'un système de compression, mais uniquement un ensemble d'unités arithmétiques banalisées formant des chemins de données parallèles, accompagné en général d'une unité de contrôle programmable, de type RISC ou VLIW. Il s'agit en fait d'une adaptation à la compression vidéo des architectures de DSP à usage général. Les architectures de VSP doivent en particulier posséder un système d'accès aux données performant, puisque la quantité de pixels à traiter à chaque trame est très élevée. Comme les DSP, qui sont basés sur les opérations d'addition et de multiplication, les VSP comportent un certain nombre d'opérateurs standard, permettant d'effectuer rapidement les opérations les plus fréquentes des algorithmes de compression. Les transformées en cosinus et les filtrages éventuels utilisent des multiplieurs-accumulateurs. On trouve souvent des opérateurs de valeur absolue et de recherche de maximum ou minimum, qui en sortie d'ALU, permettent de réaliser l'estimation de mouvement. On trouve aussi des décaleurs et des opérateurs de saturation/arrondi, entre autres pour ajuster la dynamique des différentes variables de calcul. De nombreuses architectures de VSP commerciaux sont présentées dans [Kons92] [Pirs95], comme VISP (NEC), DISP (Mitsubishi), IP (Hitachi), IP (Toshiba), RISP (Matsushita), PIP (Vitec), Pipeline IP (Matsushita), ISMP (Matsushita), IDSP (NTT), AxPe640V, VSP3, VP/VC (IIT)... Certaines de ces architectures, peuvent être considérées comme plus générales que les VSP, et peuvent être vues comme des architectures de traitement d'image en temps réel, permettant non seulement la compression vidéo mais aussi l'implémentation de nombreux algorithmes de traitements niveau pixel aussi utilisés en analyse d'images par exemple. C'est véritablement le cas des processeurs DIP [Yate95] [Good95], HiPAR [Rönn96], et PVP (CNET-Grenoble). Ces derniers incorporent tous des unités parallèles de type SIMD. Le processeur DIP incorpore un réseau bidimensionnel de 16*16 processeurs 2 bits, travaillant en mode SIMD ou systolique. Les processeurs HiPAR et PVP possèdent un contrôle de type VLIW, des capacités d'adressage mémoire par motifs, et leur mode SIMD dispose d'une autonomie de sélection d'instruction, qui permet à chaque sous-processeur d'effectuer des traitements conditionnels sans avoir les coûts d'une structure MIMD. Citons enfin le processeur MVP ("Multimedia Video Processor" ou "C80", de Texas Instruments) [Gutt92], qui est un multiprocesseur programmable, incorporant quatre DSP 32 bits à virgule fixe pouvant travailler en mode MIMD, des mémoires SRAM partagées via un réseau "crossbar", un processeur "maître" de type RISC 32 bits à virgule flottante, des mémoires cache, des contrôleurs de transfert et d'affichage... Le MVP permet d'implémenter les normes de compression vidéo (cela constituait un des objectifs majeurs lors de sa conception, et c'est pourquoi nous le mentionnons dans ce paragraphe), mais son champ d'application est en fait beaucoup plus large. Son dérivé C82 peut par exemple se charger de l'ensemble des tâches de la norme H.324 de vidéoconférence [Gols96], ce qui inclut entre autres le codage vidéo H.263, le codage de parole G.723, l'annulation d'écho

acoustique, la fonction modem V.34, les normes de multiplexage et de contrôle H.223 et H.245... Ces processeurs sont en fait les précurseurs des "processeurs multimédia", présentés à la section 2.5.2.

2.3 Architectures VLSI pour la synthèse d'images

La synthèse d'images a pour but de convertir une description de haut niveau d'une scène comportant un ensemble d'objets en une image composée d'un tableau de pixels. Les objets sont généralement définis géométriquement par un assemblage de polygones dans l'espace à trois dimensions, qui définissent une approximation de leur surface. La synthèse ou "rendu 3D" doit calculer la couleur de chaque pixel de l'image, en fonction de la direction d'observation, et de la position, de l'éclairement et de la "texture" des objets. Il faut appliquer pour cela une suite d'opérations qui transforme progressivement la description initiale en une série d'objets élémentaires accompagnés de certains paramètres, qui permettent finalement de déterminer les valeurs des pixels à afficher. Cette suite d'opérations définit le "pipeline graphique" ou "pipeline de rendu", dont les étapes principales sont aujourd'hui classiques et communes à la plupart des systèmes de synthèse d'images (Figure 10) [Fole96] [Chai92] [Croc97] [Yao96a], les détails de chaque étape pouvant par contre présenter de nombreuses variantes, qui influent bien sûr sur le réalisme du rendu.

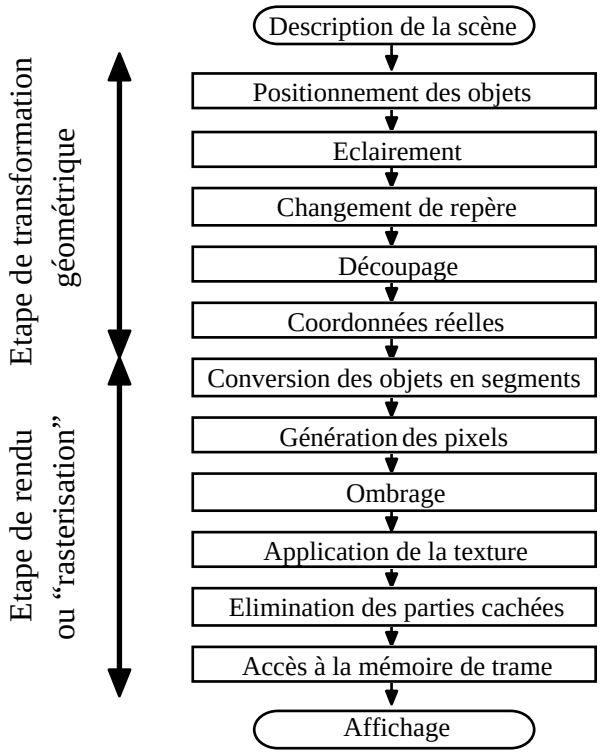

Figure 10: Pipeline de rendu standard

Nous ne détaillons pas ici les différentes étapes élémentaires du pipeline de rendu. Notons cependant qu'elles sont partagées en deux phases ou étapes globales: la transformation géométrique et le

rendu proprement dit ou "rasterisation". La transformation géométrique consiste à transformer le modèle de haut niveau en un ensemble de polygones (souvent des triangles) situés dans le plan image, définis par les coordonnées de leurs sommets (avec une composante de profondeur) et les composantes de couleur associées (ainsi que les composantes des "normales aux sommets" pour un ombrage de Phong) déterminées en fonction des sources d'éclairement. Cette première phase géométrique nécessite une grande quantité de calculs en virgule flottante, de type produit matriciel ou vectoriel. La phase de rendu doit ensuite:

- déterminer l'ensemble des pixels appartenant aux polygones, par conversion en segments horizontaux puis balayage ("rasterisation") des pixels situés entre les extrémités des segments,
- interpoler leur couleur en fonction du type d'ombrage (plat, Gouraud, Phong...), d'application de texture (par échantillonnage direct ou avec filtrage bi/trilinéaire des cartes de textures éventuellement multirésolution et avec correction de perspective), de lissage ou autres effets "atmosphériques",
- ne conserver que les pixels visibles (en général par comparaison des valeurs de profondeur stockées dans un "z-buffer") ou composer des pixels "transparents", le résultat étant écrit dans le "buffer de trame" qui sert à l'affichage.

Les traitements de la phase de rendu se situent au niveau pixel et manipulent des entiers. Bien que moins coûteux que les traitements en virgule flottante, ils sont beaucoup plus nombreux, étant donné le nombre élevé de pixels dans une image. Les deux phases de la synthèse constituent un goulot d'étranglement calculatoire pour le traitement de scènes complexes, la phase de "rasterisation" étant prédominante pour des scènes moyennement complexes ou des images de haute résolution. Le troisième goulot d'étranglement, lié à la phase de rendu, correspond aux accès aux mémoires, qui comprennent le stockage de la texture, le z-buffer et le buffer de trame. La bande passante nécessaire à ces accès est beaucoup plus grande que celle des circuits de mémoires standard. Ces trois barrières de performance ne peuvent être dépassées que par la mise en oeuvre d'architectures parallèles. Le développement d'applications basées sur le rendu 3D pour les marchés des stations de travail puis des ordinateurs personnels a nécessité la conception de processeurs ou "accélérateurs" de rendu 3D, accompagnés de mémoires rapides partitionnées, les microprocesseurs à usage général et les mémoires de données standard étant loin de pouvoir fournir la puissance de calcul et la bande passante nécessaires.

Notons qu'il existe deux grandes catégories d'algorithmes pour la "rasterisation", basées respectivement sur le traitement séquentiel des objets ("object order") ou des pixels ("image order"). Il existe des architectures pipeline et parallèles pour chaque catégorie, ainsi que des architectures hybrides qui combinent les différents types de concurrence [Fole96].

2.3.1 Processeurs de rendu 3D

Selon les marchés (stations ou PC) et les performances visés, de nombreux processeurs, pour la plupart spécialisés, ont été conçus pour accélérer la synthèse ou "graphisme 3D". Il faut distinguer le marché des stations de travail, à haute performance, de celui des ordinateurs personnels, à faible coût.

Le marché des "stations graphiques" a été le premier à se développer. L'utilisation de processeurs spécialisés aussi bien pour l'étape géométrique que pour la rasterisation a permis d'obtenir des performances élevées, à un coût beaucoup plus faible que celui des supercalculateurs. Typiquement (en se basant entre autres sur les stations Silicon Graphics), on trouve d'abord dans ces stations plusieurs processeurs de calcul en virgule flottante, connectés en pipeline et formant l'unité de géométrie. Ces processeurs peuvent être conçus spécifiquement en fonction des algorithmes et des structures de données utilisés ou être constitués d'unités standard de type FPU. Citons entre autres les circuits (plus ou moins récents): Geometry Engine, Transform Engine, Weitek3332 [Chai92], TGPx4 (Fujitsu) [Awag95]. On peut trouver ensuite (dans le pipeline global) une unité de conversion, souvent composée de trois étages d'un ou plusieurs processeurs spécialisés travaillant en mode SIMD, qui découpent les polygones en segments, définissent leurs extrémités, puis balayent et interpolent les valeurs des pixels leur appartenant. La dernière unité est constituée d'une mémoire partitionnée, chaque bloc (de type VRAM il y a quelques années) étant accompagné d'un processeur qui gère l'algorithme du z-buffer, formant par exemple un ensemble d'une vingtaine d'unités travaillant en mode MIMD. Quelques circuits de rendu sont par exemple: PRC, Scan Line Processor [Chai92], SAGE [Fole96], Image [Dunn92], Render Engine [Tan95]. Une structure pipeline objet pour la géométrie et un parallélisme image pour la rasterisation, à base d'unités spécialisées, sont assez classiques pour les stations graphiques.

Pour le marché des ordinateurs personnels, les architectures utilisées visent des performances moins élevées mais à coût faible et à intégration élevée. En général, les accélérateurs de rendu 3D pour les PC ne prennent en charge que la phase de rasterisation, après la conversion des polygones en segments. Le processeur central, qui incorpore couramment une unité FPU, effectue de manière logicielle la phase de géométrie, ainsi que la conversion en segments horizontaux. Les accélérateurs ou coprocesseurs 3D sont pour la plupart des circuits spécialisés, constitués d'un pipeline niveau pixel et d'interfaces mémoire performantes. La mémoire devant être la plus intégrée possible, elle n'est pas partitionnée et les différents buffers (z-buffer, texture, trame) sont souvent stockés ensemble. Des algorithmes simplifiés permettent aussi d'éliminer le z-buffer niveau pixel. La bande passante élevée, correspondant à la vitesse élevée de la structure de calcul pipelinée, est maintenue grâce à l'utilisation de mémoires cache et de nouvelles structures de mémoires comme les EDO, SDRAM, SGRAM, RDRAM [Kuma95] [Saka95] [Przy96] [Floh96]. Toute une gamme de processeurs 3D spécialisés est comparée dans [Floh96] et [Yao96b], comme: Voodoo (3Dfx), Permedia (3Dlabs), Virge (S3), OTI64311 (Oak), T3D9695 (Trident), PowerVR (VideoLogic), et NV1/NV3 (Nvidia). Ce dernier produit incorpore en fait

tout un ensemble de fonctionnalités correspondant au support matériel de l'API DirectX (Microsoft): accélération 2D et 3D, décompression MPEG, synthèse audio, interfaces joystick... Il s'agit donc d'un accélérateur multimédia, qui intègre en une seule puce les fonctions gérées auparavant par l'ajout de plusieurs cartes à la "carte mère" contenant le CPU. Il existe aussi des processeurs programmables qui se chargent du rendu 3D. Citons ici le processeur Verite (Rendition) [Gwen96a] [Floh96] [Yao96b], qui contient non seulement une unité de rendu (toujours de type pipeline niveau pixel) mais aussi un cœur de processeur RISC (de type R3000) avec banc de registres étendu et instructions spéciales permettant de traiter en mode SIMD deux ou quatre pixels par cycle. Le coeur RISC permet en particulier d'effectuer la conversion des polygones en segments. Ce processeur propose une accélération pour les fonctions 2D, 3D et vidéo, et pourrait presque être classé dans la catégorie récemment apparue des "media-processors". Des processeurs comme Mpact (Chromatic) et Trimedia (Philips) permettent donc aussi le rendu 3D mais sont présentés plus loin.

2.3.2 Mémoires à traitement intégré

La bande passante nécessaire entre les étages successifs du pipeline de rendu augmente tout au long de ce pipeline, pour devenir énorme au niveau des accès mémoire. Une approche radicale pour gérer cette bande passante consiste à exploiter un parallélisme pixel massif, en associant un processeur à chaque pixel de l'image, et en l'intégrant directement avec la ligne ou colonne mémoire qui stocke les valeurs associées à ce pixel (composantes R, G, B, alpha, z-buffer...). Il est effectivement possible d'intégrer dans un même circuit une mémoire importante et une ligne de processeurs élémentaires, tous de même largeur qu'une colonne de mémoire et situés directement à côté des "sense amp's". Ces processeurs, de faible complexité (en général basés sur quelques registres et une ALU sur 1 à 8 bits), travaillent en mode SIMD. De tels circuits sont par exemple à la base des machines Pixel Planes 4 et 5, et PixelFlow [Fole96] [Chai92] [Poul92]. Ils constituent une réalisation à grain fin d'une "mémoire à logique intégrée" ("logic-enhanced memory"). Le circuit EMC de la machine PixelFlow [Poul92] est constitué d'une mémoire pixel de 256 colonnes de 2048 bits (soit 512kbit), chaque colonne étant directement connectée à une ALU 8bits, et l'ensemble des ALU recevant les résultats d'interpolation d'un évaluateur parallèle d'expressions linéaires à structure d'arbre binaire. Deux autres architectures de "mémoires à fonctionnalités enrichies" ou "mémoires intelligentes" ("smart memories") ont été proposées dans le cadre de la synthèse d'images: Texram [Schi96], qui permet un "texturage" rapide et de haute qualité, et 3-D-RAM [Inou95], qui est une mémoire de trame incorporant des unités de traitement pour le "z-buffer" et l'"alpha blending" et des mémoires cache de type SRAM. Ces deux circuits intègrent 10Mb de DRAM, partitionnée en huit ou quatre blocs accessibles en parallèle, ce qui correspond à une granularité beaucoup plus élevée que pour EMC, l'approche favorisant surtout l'intégration monochip de toute la mémoire image.

2.4 Architectures VLSI pour l'analyse d'images

La vision par ordinateur constitue toujours un des grands défis de l'informatique et du traitement d'image. Le domaine des architectures parallèles pour l'analyse ou "traitement" d'images est un champ de recherche très vaste et aussi très répandu [Weem91] [Mare88] [Char93] [Méri91] [Cyph89] [IEEE96] [Dula96]. On définit classiquement une architecture pyramidale pour les systèmes de vision, comprenant des traitements de bas niveau mis en oeuvre sur un réseau homogène et régulier de processeurs à grain fin localement interconnectés et travaillant en mode SIMD au niveau pixel, puis des traitements de niveau intermédiaire et de haut niveau mis en oeuvre sur des architectures à grain plus élevé, fonctionnant en mode SPMD ou MIMD, à interconnexions plus complexes (en arbre, hypercube...) et manipulant des structures de données de haut niveau (primitives géométriques, graphes...). L'évolution des technologies VLSI a permis d'intégrer de plus en plus ces systèmes, qui possèdent un parallélisme quasi-illimité par rapport aux applications courantes des ordinateurs. Nous ne considérons pas dans la suite l'ensemble des architectures VLSI conçues pour s'insérer dans un système de traitement d'images mais présentons plutôt deux classes architecturales caractéristiques et génériques, à parallélisme massif: les processeurs associatifs et les "array-processors" SIMD (nous ne présentons ni d'architectures spécifiques à des algorithmes particuliers, ni la classe des réseaux VLSI systoliques [Kung88], qui a surtout été appliquée à la synthèse d'opérateurs spécialisés).

2.4.1 Processeurs associatifs

Les ordinateurs et processeurs associatifs [Lea91] [Chis89] [Pott92] [Fost76] [IEEE92b] [IEEE94] [Krik97] ont été initialement étudiés pour des applications de recherche d'informations et de traitement parallèle dans des bases de données. Ils sont basés sur des mémoires associatives ou "adressables par le contenu" ("content-addressable memory", ou CAM) (les "mémoires cache" qui accompagnent tous les microprocesseurs rapides sont des mémoires associatives). Ces mémoires permettent non seulement de stocker des données mais aussi de les rechercher, non pas à partir d'une adresse, mais à partir d'une valeur (parfois appelée "clé" ou "motif"). Des dispositifs de comparaison sont intégrés à chaque mot ou bit mémoire (chaque point mémoire étant alors composé de 5 à 10 transistors [IEEE92b] [Herr92] [Herr95]) et permettent de sélectionner en parallèle tous les mots qui correspondent à la clé. La recherche peut s'effectuer soit directement sur les données (qui sont stockées dans des cellules de mémoire associative), soit sur un champ réservé au stockage de la clé, associé à chaque donnée (les données pouvant alors être stockées dans des cellules conventionnelles de RAM). La clé peut soit spécifier en entier le mot à rechercher, soit comprendre un masque binaire qui spécifie le sous-ensemble de bits à prendre en compte, les autres bits (de type "don't care") n'intervenant pas dans la comparaison. Une fois la recherche effectuée, on peut extraire séquentiellement les données (voire les clés) sélectionnées ou les remplacer en parallèle par une autre donnée (éventuellement avec un masque, ce qui ne modifie que

certains bits). Des dispositifs particuliers peuvent être ajoutés pour compter le nombre de données sélectionnées.

Dans un processeur associatif, on associe à chaque mot d'une mémoire associative un processeur élémentaire, qui permet d'effectuer des traitements sur les mots sélectionnés, en général en mode "bit-série". Le temps de traitement d'un tel système n'est pas proportionnel au nombre de données (qui peut être très élevé) mais à la taille des données (qui est en général de l'ordre de quelques dizaines de bits), car les opérations de recherche et d'écriture se font en parallèle sur tous les mots (et aussi de manière "bit-parallèle", grâce à l'utilisation de cellules de mémoire associative connectées en "ET câblé" pour la recherche, mais ces opérations sont répétées pour traiter séquentiellement les différentes tranches de bits de poids donné).

Ces processeurs, initialement conçus pour des applications liées aux bases de données ou à l'intelligence artificielle, sont aussi parfaitement adaptés à exploiter le parallélisme massif et à grain fin des traitements d'images de bas niveau. Plusieurs circuits VLSI ont été réalisés dans cette optique. Un processeur associatif parallèle a par exemple été conçu au MIT, comprenant 64 [Herr92] puis 256 processeurs élémentaires 1bit [Herr95] [Geal96], avec 64 "trits" (élements ternaires: 0, 1, X ou "don't care") de CAM chacun. Chaque PE contient un registre d'activité et un générateur de fonction à deux entrées. Le processeur GLiTCH [Stor92] contient 64PE 1bit avec 64(+4)bits de CAM chacun, ainsi qu'un registre à décalage de 8bits pour entrées/sorties vidéo, et une ROM d'instructions. Chaque PE contient trois registres (dont un registre d'activité) et une ALU. Ces processeurs incorporent aussi des mécanismes de communication entre PE voisins. Dans GLiTCH, les registres "Tag" des PE sont connectés en registre à décalage et leur contenu peut ainsi se déplacer d'une ligne à chaque cycle. Dans le processeur du MIT à 256 PE, les processeurs sont reliés en grille bidimensionnelle 16*16, et peuvent échanger 1bit avec chacun de leurs quatre voisins à chaque cycle. De plus, ils peuvent communiquer selon un mode reconfigurable aussi appelé mode de propagation "asynchrone", qui permet en un seul cycle de propager un bit dans un sous-ensemble de PE, correspondant à une région connexe de l'image. Le terme de "propagation asynchrone", aussi utilisé de manière semblable dans [Dula96], à propos de la "Maille Associative" (le terme "associative" étant lui-même utilisé dans un sens différent de celui de "mémoire associative") peut prêter à confusion. Il nous paraît préférable de parler de structure ou de propagation "combinatoire" (voir chapitre suivant). Citons aussi le circuit XIUM (commercialisé par ASP) [Aker95], qui est en fait une mémoire CAM de 1024 mots de 72 bits, dont certains champs sont connectés en registre à décalage.

La structure des processeurs associatifs ressemble fortement à celle des mémoires à logique intégrée présentées plus haut: un processeur élémentaire est associé à chaque pixel et intégré directement à côté des cellules de mémoire correspondantes et tous les PE travaillent en mode SIMD. La différence majeure provient de l'utilisation de mémoire CAM ou RAM. Les processeurs associatifs sont en fait à

l'origine des réseaux VLSI de processeurs ou "array-processors" SIMD à grain fin, initialement conçus pour la réalisation de supercalculateurs massivement parallèles à usage général, et aussi des mémoires à logique intégrée à grain fin, ces deux catégories pouvant être regroupées dans la catégorie des "systèmes de traitement verticaux" ou "VPS" [Fet95], qui sont basés sur un principe de traitement séquentiel par "tranches de bits" ("bit-slice").

2.4.2 Array-processors SIMD

Contrairement aux processeurs associatifs ou aux mémoires à logique intégrée, les array-processors SIMD ne contiennent pas (ou peu) de mémoire. Ce sont typiquement des réseaux de processeurs bidimensionnels connectés en grille, chaque PE pouvant communiquer avec ses quatre plus proches voisins. Le domaine des array-processors SIMD est étroitement lié à celui des automates et réseaux cellulaires et au traitement d'images [Pres84] [Dyer81] [Rose83], pour des raisons structurelles évidentes. Initialement à grain très fin, les PE ne travaillant que sur un bit, ces structures ont évolué vers des réseaux VLSI de processeurs à grain un peu plus élevé, travaillant sur des entiers. En fonction de la technologie, qui a constamment évolué, et de la granularité, les circuits de ce type ont intégré de moins d'une dizaine à quelques centaines de PE. Citons les machines: Staran, ILLIAC4, CLIP3/4/7, DAP, MPP/BLITZEN, CM1/2, GAPP, YUPPIE, SPHINX... Nous ne présentons pas ici les processeurs qui sont à la base de ces machines aujourd'hui relativement connues, et que l'on trouve dans de nombreuses références [Batc80] [Foun87] [Fet95] [Hill88] [Pres84] [Mare88] [Char93] [Méri91] [Dula96] [Kung88] [Elli97].

Les array-processors SIMD ont récemment évolué vers des "array-processors à mémoire intégrée", qui sont des architectures duales (mais très semblables) aux mémoires à traitement intégré. Ces architectures ont plutôt une topologie monodimensionnelle (les PE étant connectés en ligne), sont à grain fin/moyen (de "niveau pixel" et non pas "niveau bit"), et éliminent les problèmes de bande passante entre mémoires et unités de traitement en intégrant les deux. Elles forment un compromis très efficace dans les technologies actuelles. Citons le processeur de traitement d'images IMAP ("Integrated Memory Array Processor", NEC) [Yama94] [Okaz95] et son successeur PIP-RAM ("Parallel Image Processing RAM") [Aimo96], qui intègrent respectivement 64PE 8bits avec 32kbit de SRAM chacun et 128PE 8bits avec 128kbit de DRAM chacun. Les PE de PIP-RAM comprennent une ALU 8bits, un décaleur, 24 registres banalisés et 5 registres spécialisés.

2.5 Architectures VLSI "multimédia"

L'utilisation conjointe de l'information vidéo, graphique (2D et 3D), et audio, dans de nouvelles applications dites "multimédia", a récemment introduit une nouvelle forme de convergence, qui se traduit par l'apparition de nouvelles architectures VLSI, qui sont souvent qualifiées de... "multimédia" (bien que

cela ne renseigne pas beaucoup d'un point de vue architectural !). Cette convergence se traduit effectivement par une évolution du spécifique au programmable, et correspond à une intégration horizontale des différents supports matériels et logiciels auparavant dédiés pour des applications distinctes. Cette intégration (fonctionnelle, structurelle, et physique) va permettre un nouveau degré de banalisation de l'information numérique, qui va pouvoir être exploitée sous de nombreuses formes par l'intermédiaire des ordinateurs personnels ou de terminaux/assistants qui sont encore à inventer... Mais revenons aux architectures VLSI. Nous avons décrit dans les sections précédentes un certain nombre de classes architecturales, relativement spécifiques à des applications basées sur une des formes particulières de traitement de l'information visuelle (compression de séquences vidéo, rendu d'images synthétiques, analyse d'images acquises en temps réel) et avons essayé de montrer l'évolution des circuits VLSI des chipsets dédiés aux coprocesseurs/accélérateurs puis aux processeurs programmables. Cette évolution se complète actuellement par le développement de deux classes de processeurs programmables, qui supportent l'exécution temps réel des algorithmes standard (en particulier normalisés) de restitution à la fois vidéo, graphique et sonore. Il s'agit des microprocesseurs (à usage général) à extension multimédia (ou encore "multimedia-enhanced CPUs") et des "media processors" [Bhas97] [Lee96a].

2.5.1 Extensions multimédia de microprocesseurs à usage général

La solution la plus programmable est bien sûr celle basée sur un microprocesseur à usage général, qui est au coeur de tous les ordinateurs. Elle n'était pourtant pas applicable il y a quelques années à la mise en oeuvre en temps réel des systèmes de compression/décompression vidéo ou audio, car elle ne permettait pas d'atteindre les perfomances nécessaires, contrairement aux solutions spécialisées. Les performances des microprocesseurs ne cessant d'augmenter, grâce à l'évolution de la technologie et des architectures utilisées, ils se situent aujourd'hui à un point où la prise en compte de modifications relativement mineures (du moins en surface de silicium) leur permet d'accélérer substantiellement les traitements spécifiques à la restitution multimédia, et de "rattraper" les solutions spécialisées (au moins pour les configurations de base). Ces modifications consistent en une extension (dite "multimédia") du jeu d'instructions des microprocesseurs, et principalement en l'ajout de degrés de flexibilité dans les unités de calcul entier ou flottant, qui permettent par l'intermédiaire des nouvelles instructions d'exploiter un parallélisme "intra-mot" ("subword") de type SIMD [Bhas97] [Lee96a]. Il faut en effet remarquer que les chemins de données des CPU actuels sont beaucoup plus larges (32 ou 64 bits en entier et 80 bits en flottant) que les données manipulées pour la restitution audio ou vidéo (qui n'utilisent jamais plus de 16 bits, et même 8 bits pour les composantes d'un pixel). L'idée principale consiste donc à effectuer en une seule instruction des traitements (identiques) en parallèle sur plusieurs données, de taille 8, 16 ou 32 bits, dans un chemin de données de 64 bits, ce qui permet théoriquement d'effectuer jusqu'à 8 calculs par cycle. Pour cela, il suffit par exemple dans plusieurs cas d'opérations arithmétiques d'ajouter des

multiplexeurs qui découpent la chaîne de propagation de retenue ou de décalage et créent dynamiquement un ensemble de sous-opérateurs (d'addition/soustraction, de comparaison, de décalage). On peut appeler cela l'arithmétique "sécable" (plutôt qu'"empaquetée" pour "packed arithmetic"). D'autres extensions de l'ALU ou de la FPU permettent d'effectuer plusieurs multiplications ou multiplications-accumulations par cycle, de combiner additions et décalages... L'ajout d'un mode de saturation pour la plupart des nouvelles instructions arithmétiques est aussi une caractéristique intéressante, en particulier pour les traitements sur les pixels (un pixel ne pouvant pas être "plus blanc que blanc" [Gwen96c]...).

La première extension multimédia, MAX-1, comportant 5 types d'instructions (d'addition, soustraction, et décalage, avec différents modes), a été présentée par Hewlett-Packard en 1994 pour le processeur 32bits PA7100LC [Undy94] [Lee95]. Elle a permis de mettre en oeuvre les algorithmes de MPEG-1 en temps réel, à partir d'un logiciel écrit en langage C (utilisant des macros invoquant les instructions MAX-1). Par la suite, une nouvelle extension, MAX-2 [Lee96b], a été introduite avec le processeur 64bits PA8000, permettant de doubler le parallélisme intra-mot. Dans ces processeurs, l'arithmétique sécable est utilisée dans l'unité de calcul entier.

Entre temps, Sun a introduit l'extension VIS ("Visual Instruction Set") [Trem96] pour les processeurs UltraSPARC, qui présente un grand nombre de nouvelles instructions, dont certaines peuvent remplacer plusieurs dizaines d'instructions élémentaires. Citons par exemple une instruction qui calcule la somme des valeurs absolues des différences de 8 paires de pixels (réalisant donc 8 soustractions, 8 valeurs absolues, et 8 additions en un seul cycle), et qui est évidemment prévue pour accélérer l'estimation de mouvement. Contrairement aux processeurs HP, cette extension est mise en oeuvre dans l'unité de calcul flottant, ce qui permet entre autres d'exploiter en parallèle l'unité de calcul entier et de ne pas encombrer le banc de registres principal.

De même, l'extension MMX d'Intel [Gwen96c] [Pele96] [Pele97] pour les processeurs Pentium, est basée sur la modification de la FPU. Cette extension est intermédiaire entre MAX-2 et VIS, en termes de nombre d'instructions et de complexité. Elle permet en particulier des multiplications-accumulations parallèles sur 16bits.

D'autres fabricants de microprocesseurs proposent aussi des extensions multimédia comme MVI (Digital), MIPS V et MDMX (MIPS) [Gwen96d], NEC-V830 [Nade95], ainsi que les fabricants de processeurs compatibles Intel.

2.5.2 Processeurs multimédia

Une classe intermédiaire entre des architectures optimisées pour une application, comme les VSP, ou les coprocesseurs de rendu 3D, et les microprocesseurs à usage général (à extension multimédia) est apparue en 1996, sous le nom de "media processor" [Glas97] [Turl96] [Bhas97], ou "processeurs multimédia". Comme les microprocesseurs à usage général, il s'agit d'architectures pleinement

programmables. Une caractéristique essentielle est qu'ils supportent de manière matérielle tout un ensemble de traitements associés aux applications "multimédia", manipulant différents flux et types de données ("média"). Un media-processor doit pouvoir gérer la compression/décompression vidéo et audio, et l'accélération graphique 2D et 3D, voire aussi la fonction modem. Ils intègrent des interfaces d'entrées/sorties performantes pour les accès à la mémoire (de trame en particulier), l'acquisition et la restitution des signaux audio et vidéo, les communications avec un CPU, la fonction modem... Ils sont utilisables soit comme coprocesseurs de PC multimédia, soit comme processeurs hôtes de terminaux ("set-top box", lecteurs DVD...). Un autre point qui les différencie des coprocesseurs programmables plus classiques est leur capacité à supporter un OS ou noyau temps réel propre, et à partager certaines tâches avec un CPU.

Les media-processors reposent pour la plupart sur des architectures avancées de type VLIW et SIMD. Une architecture VLIW ("Very Long Instruction Word") permet d'exploiter un parallélisme instruction, déterminé statiquement à la compilation, contrairement aux architectures superscalaires. Cette propriété permet d'obtenir des architectures relativement simples et régulières, ce qui augmente leurs performances (car elles sont susceptibles d'être fortement pipelinées). L'utilisation de mémoires à large bande passante est essentielle pour soutenir le débit de calcul potentiel. Plusieurs niveaux de parallélisme SIMD sont aussi utilisés: ces processeurs incorporent de nombreuses unités de calcul entier et flottant de 32 à 128bits, et font un usage généralisé de l'arithmétique sécable, adaptant ainsi dynamiquement la taille des chemins de données au "types de média" pour effectuer le plus d'instructions possibles à chaque cycle. Une ou plusieurs des unités de calcul peuvent être spécialisées pour accélérer certains traitements standard.

Les deux media-processors les plus "média"-tisés sont les processeurs Mpact et Trimedia. Ce sont en fait plutôt des coprocesseurs, dans le sens où ils ont été principalement prévus pour assister un CPU dans un ordinateur personnel. Le processeur Mpact1 (Chromatic Research) [Kala97] [Yao96b] [Bhas97] [Glas97] [Turl96] incorpore entre autres 5 groupes d'ALU 72 bits (pouvant traiter des paquets de données de 9, 18, 24, ou 36 bits), dont un spécialisé pour l'estimation de mouvement. Un total de 19 bus (de 72 bits) véhiculent les données entre les différentes unités avec un entrelacement quelconque (via un "full crossbar"). Une mémoire SRAM contenant 512 entrées de 72 bits sert de banc de registres et de mémoire cache, et possède 4 ports de lecture et 4 ports d'écriture... Le processeur Mpact2 [Purc97] [Yao96d] corrige les faiblesses du premier et apporte en particulier un véritable support de rendu 3D. Outre le fait que sa fréquence d'horloge, la bande passante de ses accès RDRAM, et sa capacité en mémoire cache sont doublés (125MHz, 1200Moctet/s, 8Koctet), il intègre des capacités de calcul flottant dans les quatre premiers groupes d'unités, et ajoute une sixième unité spécialisée pour le rendu 3D, comprenant 35 étages de pipeline et devant permettre le rendu d'environ un million de triangles de 50 pixels par seconde, avec ombrage de Gouraud, z-buffer sur 18 bits, correction de perspective, filtrage

bilinéaire, alpha blending, effet de brouillard... Une caractéristique qui distingue les circuits de Chromatic est qu'ils ne sont pas directement programmables par l'utilisateur ! L'essence de ces produits se trouve au niveau logiciel, Chromatic vendant le noyau temps réel et les modules de "Mediaware" qui résident dans le media-processor, ainsi que les drivers et le reste du Mediaware utilisés par le CPU. Ces produits sont prévus en particulier pour fonctionner avec les processeurs Pentium, et les nouvelles versions de Mediaware devraient exploiter conjointement l'extension MMX et les ressources du media-processor. Le support des API standard est un élément essentiel de ces produits, et correspond effectivement à la notion d'intégration horizontale.

Le processeur Trimedia TM-1 (Philips) [Bhas97] [Glas97] [Turl96], est basé autour d'un coeur VLIW et comprend 27 unités fonctionnelles, dont 5 peuvent recevoir de nouvelles instructions à chaque cycle. Il possède deux unités spécialisées, une pour le codage à longueur variable, et une unité de déplacement de bloc qui gère les flux de données entre les SDRAM externes et les ports audio/vidéo et PCI. Il peut exécuter jusqu'à 38 calculs sur 8bits par cycle, à 100MHz, et fournit des fonctionnalités équivalentes à celles de Mpact-1 (sauf pour l'accélération 2D et 3D).

Deux autres circuits peuvent véritablement porter le nom de processeur multimédia (et non coprocesseur): le MSP (Samsung) [Glas97] [Yao96c] et le MediaProcessor (MicroUnity) [Hans96]. Le MSP incorpore en effet un coeur RISC ARM7, qui peut pleinement gérer les tâches système, ce qui lui permet de fonctionner de manière autonome, pour des applications DVD par exemple. Il inclut aussi un processeur vectoriel SIMD de 288bits permettant les calculs en virgule flottante. Le MediaProcessor est un processeur à part entière et fournit des performances impressionnantes. Il possède un jeu d'instructions original et chaque instruction travaille sur 4 registres de 128bits, divisibles en groupes de 1, 2, 4, 8, 16, 32, et 64 bits. Sa structure permet des bandes passantes très élevées: 512Gbit/s pour les accès aux registres et 128Gbit/s de bande passante mémoire. Il s'agit en fait d'un processeur à entrelacement de contrôle ou "multithreading", qui permet à 5 "threads" d'instructions d'être exécutées simultanément à 200MHz, les instructions étant "lancées" à une fréquence de 1GHz ! Ces deux solutions ne semblent cependant pas connaître le succès des deux premiers processeurs.

Citons ici l'annonce du système Talisman (Microsoft) [Rand97] [Torb96] [Glas96], qui est en premier lieu une architecture innovante pour le rendu 3D. Ce système est basé sur la combinaison d'un media processor (initialement MSP ou Trimedia, seul ce dernier ayant été apparemment retenu) et d'une unité complète de rendu, le media-processor s'occupant de la phase de géométrie et de conversion. Son originalité réside principalement dans l'utilisation d'un compositage de plans image, qui élimine la mémoire de trame. Elle est remplacée par un buffer contenant seulement 32 lignes de l'image, qui est rempli en temps réel par un compositage de différents plans qui sont générés de manière indépendante de l'affichage et seulement lorsque nécessaire, c'est-à-dire lorsque les objets qu'ils contiennent ont subi une modification importante. Le compositage permet de générer les images intermédiaires par transformation

affine, ce qui évite d'avoir à effectuer le rendu de l'ensemble des objets à chaque trame. La bande passante est réduite encore plus par compression/décompression des données lors des accès mémoire. Ce système permet l'utilisation de techniques très avancées de rendu et en particulier de texturage, qui sont généralement beaucoup trop coûteuses à implémenter. De plus, l'utilisation d'un media processor permet de supporter l'accélération 2D, ainsi que le codage/décodage audio et vidéo. Une caractéristique remarquable, qui correspond à une première forme de "convergence multimédia" est la possibilité d'utiliser un plan image contenant des données vidéo comme source de texture pour le compositage, ce qui peut être vu comme une fonctionnalité de type SNHC ("Synthetic Natural Hybrid Coding").

2.6 Limitations des processeurs multimédia

Les media-processors et les microprocesseurs à extension multimédia constituent l'état de l'art en termes de conception de circuits VLSI. Ils procurent des niveaux de performance remarquables et présentent une augmentation de complexité impressionnante, par rapport aux architectures conçues seulement un an ou deux auparavant. Ils permettent aujourd'hui de mettre en oeuvre des systèmes multimédia "de première génération", basés en particulier sur les algorithmes classiques de compression vidéo et le pipeline graphique standard pour le rendu d'objets 3D composés de facettes triangulaires. On peut légitimement se demander quelle est l'adéquation des architectures de processeurs multimédia à la mise en oeuvre des systèmes de communication visuelle de deuxième génération. Nous esquissons ici quelques réflexions sur leurs limitations, et sur leur capacité à évoluer avec la technologie. Ces réflexions peuvent parfois sembler un peu trop partiales, mais cela nous semble nécessaire à la critique constructive...

2.6.1 Niveaux de spécificité

Les media-processors et les extensions multimédia ont été conçus pour accélérer des classes bien particulières d'algorithmes. La compression hybride par blocs requiert des traitements très spécifiques comme la transformée en cosinus discrète, l'appariement de blocs par calcul d'une norme L^1 sur un bloc carré de pixels de taille fixe, le balayage zigzag et la quantification associée, un codage à longueur variable particulier... Tous ces traitements peuvent ne pas figurer parmi les traitements de base d'un algorithme de codage dans un système de deuxième génération. La présence d'unités ou d'instructions spécialisées pour certains de ces traitements est un signe de trop grande spécificité des processeurs multimédia, même s'ils sont fondamentalement programmables. Ils constituent une intégration de différents traitements normalisés plutôt qu'une véritable convergence à interface ouverte, qui prendrait en compte des primitives beaucoup plus génériques, pour la segmentation en régions ou l'analyse en objets à modèle de forme et de mouvement. Ils n'ont pas du tout été conçus pour l'analyse d'images en temps réel, qui peut comprendre des traitements irréguliers, de différents niveaux de parallélisme, et basés sur des

représentations de plus haut niveau que le tableau de pixels. Le modèle de traitement SIMD, prévu pour des opérations régulières sur des blocs de pixels de taille fixe, n'est ainsi pas nécessairement bien adapté. L'adressage séquentiel des pixels ou des blocs de pixels constitue de plus une forme particulière et artificielle de séquencement, alors que le problème peut comporter un parallélisme implicite important, même s'il est spatialement irrégulier. Le flot de contrôle est trop statique et ne s'adapte pas aux valeurs ou à la topologie des données. La durée de vie des media-processors peut de plus sembler faible [Gwen96b], à cause des microprocesseurs à usage général qui rattrapent rapidement leur performance, et du caractère éphémère des API, qui sont rapidement remplacées par de nouveaux standards de fait.

2.6.2 Performances

Bien que performants pour la mise en oeuvre de la génération de traitements pour laquelle ils ont été conçus, les processeurs multimédia ne disposent pas d'une puissance de calcul suffisante pour la nouvelle génération envisagée, en particulier à cause de la complexité du problème de l'analyse.

a) Exploitation du parallélisme

Les types de parallélisme exploités sont soit des parallélismes de contrôle, au niveau instruction (architectures superscalaires et VLIW) ou au niveau des segments de code ("multithreading"), soit des parallélismes de données très contraints et limités, comme les modes SIMD associés au découpage des unités (arithmétique sécable). Le parallélisme instruction, qu'il soit déterminé dynamiquement par le matériel ou statiquement à la compilation, est en général relativement restreint. Il ne correspond d'ailleurs pas au type de parallélisme fondamental des algorithmes de traitement d'image (parallélisme de données niveau pixel). Les architectures superscalaires sont très peu "scalables" car leur complexité croît exponentiellement avec le nombre de voies, sans pour autant permettre un taux d'accélération linéaire. Les architectures VLIW sont limitées par le parallélisme instruction détectable par le compilateur et par le nombre fixe d'instructions à exécuter à chaque cycle. Quant au "multithreading", il est très difficile d'écrire des applications permettant son utilisation. Enfin, le parallélisme SIMD introduit avec l'arithmétique sécable ne dépasse pas huit opérations simultanées, et n'est que peu extensible a priori, puisqu'il s'agit d'un découpage d'unités dont la taille est liée à l'ensemble de l'architecture. Le parallélisme SIMD, s'il pouvait être étendu, serait de toute manière inefficace car les accès aux pixels et leurs traitements doivent respecter un partitionnement trop rigide.

b) Limitations des circuits synchrones

La méthodologie de conception et la structure même des circuits synchrones sont à la base de limitations qui pourraient se révéler très pénalisantes dans quelques générations technologiques. L'utilisation d'une synchronisation globale est en fait très contraignante pour la conception de systèmes à haute vitesse, et ne possède aucune justification fonctionnelle (si l'on s'autorise à imaginer des structures

alternatives). Les temps de propagation de l'horloge dans les différentes parties du circuit, ainsi que la caractérisation des dispositifs et des interconnexions (celle-ci n'étant connue que dans la dernière phase de conception) impliquent la prise en compte d'une marge relative de fonctionnement qui devient de plus en plus importante. C'est d'ailleurs la raison pour laquelle les structures SIMD à large échelle ne connaissent pas une accélération linéaire avec la technologie. De plus le système de distribution de l'horloge, ainsi que la commutation inutile d'une forte proportion de dispositifs, introduisent une consommation énergétique importante, alors que la puissance dissipée semble être justement un facteur critique pour l'utilisation des futures technologies.

2.7 Perspectives de conception des futures architectures VLSI

Nous présentons ici de manière introductive un certain nombre de perspectives d'évolutions et de directions qui sont à la base des contributions architecturales et algorithmiques de cette thèse.

2.7.1 Evolution technologique et choix architecturaux

Les futures technologies VLSI nous promettent d'ici peu des capacités d'intégration et des vitesses de commutation d'un ordre de grandeur supérieures à celles d'aujourd'hui. Alors que cela pourrait sembler apporter des bénéfices immédiats, certains s'inquiètent pourtant de savoir quoi faire avec 100 millions de transistors et comment les faire fonctionner efficacement [Webb96]. Il ne faut pas oublier que l'évolution des architectures VLSI ne correspond pas simplement à une augmentation de la complexité rendue possible par la diminution des surfaces, mais à un ensemble subtil de compromis permettant d'utiliser au mieux un niveau de complexité par un choix de structure adapté. Cela signifie que l'approche "incrémentale" des microprocesseurs standard, plus ou moins imposée par une optimisation des bénéfices à court terme reposant sur la compatibilité ascendante, risque d'être remise en cause par le saut technologique. Des solutions abandonnées il y a longtemps au profit de choix mieux adaptés à une complexité et des performances données, peuvent ainsi devenir de nouveau viables, à partir du moment où elles bénéficient enfin de la technologie qu'elles méritent... Nous proposons de revisiter les concepts de parallélisme massif et d'asynchronisme, apparus dès l'époque des premiers supercalculateurs, mais délaissés momentanément pour cause de trop grande complexité, d'autres modèles de conception s'étant imposés entre temps.

2.7.2 Intégration et convergence mémoire-traitement

Une direction qui semble être largement proposée, parfois à partir d'approches assez différentes au départ, est l'intégration puis la convergence mémoire-traitement. L'intégration mémoire-traitement consiste à associer dans une même technologie une grande quantité de mémoire directement à proximité du coeur de traitement. En particulier, il s'agit d'utiliser une technologie DRAM, plusieurs dizaines de fois plus dense que la SRAM, et de fabriquer la partie logique (le ou les microprocesseurs associés à

cette mémoire) dans cette technologie [Patt97]. Cela a pour effet de diminuer la vitesse intrinsèque des dispositifs, mais procure en contrepartie une augmentation considérable de la bande passante entre la mémoire et le processeur (les évolutions de la bande passante mémoire et de celle de traitement ayant jusque là plutôt eu tendance à diverger en faveur des performances des processeurs). Les structures obtenues sont entre autres dénommées IRAM (pour "Intelligent RAM", ce qui n'est pas forcément la meilleure appellation) [Patt97]. La convergence mémoire-traitement est un concept qui va plus loin. Il consiste à répartir ou distribuer la quantité totale de mémoire parmi un ensemble important de processeurs plus simples (que les CPU standard) qui peuvent travailler concurremment, ce qui multiplie d'autant la bande passante totale, tout en conservant une forte localité, qui garantit la bonne scalabilité de la structure. Une étape intermédiaire comprend les architectures des multiprocesseurs intégrés, qui peuvent être vus comme des mémoires à traitement parallèle ou PPRAM [Mura97].

La convergence mémoire-traitement est au coeur de plusieurs classes d'architectures VLSI, comme les mémoires à logique intégrée, et les processeurs associatifs, introduits plus haut , qui ont déjà montré leurs forts potentiels pour l'analyse et le rendu d'images. Une classe similaire, mais pas nécessairement dédiée au traitement d'images, est constitué des "mémoires computationnelles". Le circuit C-RAM ("Computational RAM") [Elli92] [Elli97] a par exemple été conçu pour réaliser une mémoire de trame "intelligente" pour des applications vidéo, mais présente une structure relativement générique, applicable à différents types de traitement parallèle. Un prototype de mémoire 8kbit intégrant 64PE 1bit avec 128bits de mémoire SRAM chacun est présenté, une version DRAM de 4Mbit avec 2048PE étant envisagée. Cette architecture a été évaluée pour la mise en oeuvre de MPEG2 dans [Le97]. Le processeur PIM ("Processor In Memory") [Gokh95], conçu comme coprocesseur parallèle pour les stations de travail (64PE 1bit avec 2kbit de mémoire SRAM chacun) présente une structure du même type. Le parallélisme pixel basé sur la convergence mémoire-traitement, et pas simplement une structure de réseau de processeurs, peut permettre de définir des modèles génériques et performants pour la mise en oeuvre des algorithmes d'analyse/rendu. Nous y reviendrons au chapitre 5.

2.7.3 Flexibilité de contrôle

Les limitations des architectures massivement parallèles intégrés, liées en particulier à leurs modes de fonctionnement et de contrôle, doivent être dépassées. Il faut y incorporer des niveaux suffisants de flexibilité et de programmabilité, et supprimer certaines contraintes de conception qui ne permettent pas de tirer pleinement partie de leur structure. Il faut tout d'abord se débarasser des limitations des architectures SIMD. La notion d'autonomie des processeurs élémentaires doit être étendue. Nous proposons pour cela de considérer plutôt un contrôle de type SPMD ("Single Program Multiple Data"). Nous verrons en particulier au chapitre 5 comment y associer les modèles de séquencement flot de données et associatif, qui permettent plus de flexibilité que les flots de contrôle classiques. Le mode SPMD prévoit des mécanismes de synchronisation explicite entre processeurs, qui constituent un premier

relâchement des contraintes de synchronisation globale des architectures SIMD. Il s'agit d'une certaine façon de passer du "parallélisme" à la "concurrence", en exploitant de manière générale ce relâchement des contraintes de synchronisation et de séquencement. L'idée principale de cette thèse est de resituer l'ensemble des problématiques de conception d'algorithmes et d'architectures parallèles vis-à-vis de la notion d'asynchronisme, ou synchronisation minimale. L'asynchronisme a déjà été étudié dans l'extension des réseaux systoliques aux réseaux à front d'onde ("wavefront arrays") et dans le mode de fonctionnement de certains réseaux cellulaires [Faur91]. Il n'a cependant jamais été considéré dans ces cas (i.e. combiné au parallélisme massif) à des niveaux de fine granularité, c'est-à-dire au niveau portes logiques, correspondant à la conception de circuits VLSI asynchrones, ni au niveau de l'exploitation algorithmique de dépendances dynamiques. Nous étudions ces différentes directions dans la suite.

Chapitre 3

DE L'ASYNCHRONISME - UNE INTRODUCTION

3.1 Vous avez dit "asynchrone" ?

Nous venons d'utiliser les termes "asynchronisme" et "asynchrone" à la fin du chapitre précédent pour faire référence à certains modèles et certaines structures qui permettent d'envisager des alternatives intéressantes aux systèmes classiques de traitement. Il apparaît cependant que ces termes sont utilisés dans de nombreux contextes, à l'intérieur même des domaines de l'informatique et de l'électronique, avec des significations très diverses (et pas toujours consensuelles). Ils sont souvent source de confusions ou d'incompréhensions. L'objectif principal de cette brève introduction est de discuter de leur définition et de leur sens dans un certain nombre de contextes qui sont soit directement au coeur des chapitres suivants (resituant ainsi nos propres "conventions"), soit liés de manière générale à la conception d'algorithmes et d'architectures parallèles. Un objectif sous-jacent est d'entamer une esquisse de synthèse des différentes notions d'asynchronisme et de différentes formes de spécification d'algorithmes et d'architectures, à partir entre autres des notions de synchronisation et de dépendance.

D'après [Laro92], l'adjectif "asynchrone" signifie: "qui n'est pas synchrone". L'adjectif "synchrone" provient du grec "sunkhronos" (de "sun": avec, et "khronos": temps) et "se dit des mouvements qui se font dans un même temps". Nous interpréterons le mot "mouvement" dans le sens "événement" (nous nous plaçons en effet dans le cadre général des systèmes à événements discrets). Le premier problème concerne bien sûr la notion de temps. On peut d'abord définir le temps comme une dimension, susceptible de permettre des mesures. Le temps est alors un attribut de chaque événement,

que l'on peut appeler "date" ou "instant" et est une propriété globale qui permet de définir une relation d'ordre totale sur l'ensemble des événements. Une autre définition considère le temps comme la succession des événements. Le temps n'est plus un attribut de chaque événement, il est directement la relation d'ordre qui lie les événements. Dans ce cas, cette relation d'ordre est a priori seulement partielle, et définit certaines "dépendances" entre événements. La deuxième difficulté est de définir dans quelles conditions on considère que deux événements se font "dans un même temps". Avec la première définition, on peut considérer les événements ayant strictement la même date, ou encore ceux dont la date appartient à un même intervalle. Dans l'autre cas, la relation d'ordre étant partielle, on peut se ramener à l'existence de dépendances communes. La notion de synchronisme doit donc être définie précisément selon le contexte, en spécifiant en particulier quels sont les événements considérés, et quelle relation d'ordre on leur associe. Elle correspond de manière générale à un ensemble de dépendances ou contraintes entre événements. La notion d'asynchronisme s'en déduit par violation ou "relâchement" d'au moins une de ces contraintes, et correspond à la suppression de certaines dépendances, par rapport à une "référence" synchrone. Elle peut être considérée comme la prise en compte de formes d'indétermination et/ou d'indéterminisme (nous allons y revenir).

3.2 Modèles de calcul parallèle asynchrones

L'étude des propriétés générales d'un algorithme se base sur l'utilisation d'un modèle de calcul, qui définit entre autres les ressources de traitement disponibles et leur mode de fonctionnement, ce qui permet d'obtenir des évaluations asymptotiques de complexité ou performance. Depuis la "machine de Turing", bien d'autres modèles ont été définis, en particulier pour décrire les systèmes de traitement parallèles, concurrents ou distribués. Nous commençons par examiner où intervient la notion d'asynchronisme dans ces modèles, qui sont à l'interface des algorithmes et des architectures.

3.2.1 Modèles et degrés de liberté

Plusieurs modèles de calcul parallèle dits "asynchrones" sont présentés par exemple dans [Gibb93], [Bert89] ou [Lync96]. L'asynchronisme fait référence à l'existence d'un certain nombre de paramètres ou degrés de liberté qui correspondent à des formes d'indétermination liées au temps (au sens soit de la date et/ou de la durée, soit de la succession des événements), contrairement aux modèles plus anciens et plus simples, dits "synchrones".

Dans les modèles parallèles classiquement appelés "synchrones", tous les processeurs sont implicitement synchronisés à chaque étape "atomique" de traitement, c'est-à-dire qu'on considère qu'ils commencent tous une étape en même temps, que la durée d'exécution de l'étape est la même pour tous, et qu'ils terminent tous en même temps. En conséquence, lorsqu'un processeur exécute une étape, il se sert du fait que l'étape précédente a été exécutée par tous les autres processeurs. Dans ce modèle, tous les

événements sont parfaitement déterminés temporellement, aussi bien du point de vue de leurs instants d'occurrence que de leur succession.

Dans les modèles "asynchrones", tout n'est pas déterminé. Un premier degré de liberté concerne la durée d'exécution des étapes élémentaires, qui peut être variable. Bien qu'une synchronisation globale soit encore effectuée implicitement, on tient cependant compte du fait qu'il faut attendre le processeur "le plus lent" à chaque étape.

On peut de plus ne pas synchroniser les processeurs à chaque étape atomique mais seulement après un ensemble déterminé d'étapes, parfois appelé phase. Pendant chaque phase, les processeurs effectuent donc un ensemble de traitements indépendamment les uns des autres et à leur propre vitesse, puis se synchronisent tous à la fin de la phase, par exemple par un mécanisme appelé "barrière de synchronisation", et peuvent échanger explicitement ou pas leurs résultats. Lors de l'exécution d'une phase, chaque processeur utilise ainsi les résulats produits par les autres processeurs à la phase précédente. Les étapes atomiques exécutées lors d'une même phase par des processeurs différents ne sont pas ordonnées.

D'autres degrés de liberté concernent les synchronisations et les communications, qui peuvent ne pas être globales, et dont l'occurrence peut ne pas être complètement déterminée. On peut d'abord définir des barrières de synchronisation ou des échanges de données par sous-ensembles ou groupes de processeurs. Les différentes synchronisations ou communications n'ont pas nécessairement des dates d'occurrence déterminées ni même d'ordre total. Il faut alors les spécifier explicitement. De plus, leur durée peut être variable. Lorsque les processeurs peuvent effectuer des synchronisations ou communications "point à point" (i.e. par groupes de deux), leurs dates et ordre d'occurrence par rapport aux événements liés aux autres processeurs peuvent être complètement indéterminées. La conception d'algorithmes basés sur ces modèles pose un certain nombre de problèmes (de synchronisation), comme l'exclusion mutuelle pour les modèles à mémoire (physiquement ou virtuellement) partagée. Les synchronisations ou communications peuvent se faire par l'intermédiaire de mécanismes de gestion des accès mémoire ou d'ordonnancement de tâches (les notions d'exclusion mutuelle, de sémaphores, de régions critiques, de moniteurs, sont étudiées entre autres dans [Dinn89] [Rayn92] [Lync96] [Chan89] [Bert89]). Elles peuvent aussi se faire par passage de messages ou par mécanismes de rendez-vous, point-à-point ou de manière semi-globale. Un degré de liberté supplémentaire concerne enfin l'exécution conditionnelle de synchronisations ou communications, qui détermine celles qui sont effectivement réalisées. Le fait de ne pas connaître à l'avance le nombre de communications dans un programme pose en particulier des problèmes liés à la détection répartie de la terminaison ou détection de fin globale [Bert89] [Chan89] [Rayn92] [Xu96]. Nous venons ici d'aborder des points qui auraient pu être présentés plus loin, à propos des algorithmes asynchrones, mais ils sont cependant liés à des mécanismes qui font partie de la définition du modèle.

Citons enfin une classe de modèles particuliers, appelés modèles de traitement "flot de données" [Dunc90] [Lee94]. Ces modèles sont basés sur l'exécution des instructions en fonction de la "disponibilité" de leurs opérandes. Il n'y a donc pas de séquencement prédéfini, et l'ordre et la durée d'exécution des traitements élémentaires sont totalement indéterminés, d'où la qualification de modèles asynchrones. Ces modèles tiennent compte directement des dépendances entre données spécifiées dans l'algorithme. Nous ne présentons pas plus en détail les mécanismes de gestion des dépendances et du parallélisme, qui définissent les différents modèles.

3.2.2 Communications synchrones et asynchrones

Nous avons discuté ci-dessus de l'utilisation de mécanismes de communication ou de synchronisation entre processeurs en particulier dans les modèles de calcul dits asynchrones. Il existe cependant plusieurs points de vue pour qualifier une communication elle-même de synchrone ou d'asynchrone.

On peut tout d'abord qualifier un mécanisme de communication en fonction du type de modèle qu'il permet de définir (des modèles synchrones ou asynchrones étant basés sur des communications respectivement synchrones ou asynchrones). Cependant les définitions précises des modes de communication sont surtout importantes dans les modèles asynchrones. Ce premier point de vue n'apporte donc pas beaucoup d'information.

Une autre définition considère les communications comme synchrones si leurs dates ou au moins leurs ordres relatifs d'occurence sont déterminés a priori (par exemple par l'utilisation de barrières de synchronisation globale), et asynchrones sinon (par exemple par l'utilisation de communications point à point conditionnelles). Cette définition correspond bien à l'ajout d'une indétermination s'appliquant sur les communications elles-mêmes (et non à d'autres paramètres qui suffisent à qualifier un modèle d'asynchrone).

Dans [Chan89], on se place dans un modèle asynchrone à communications par passage de messages, qui ne fait pas d'hypothèse sur les relations entre les différentes occurrences de communication ni sur les temps d'exécution des traitements. On considère de manière classique que les messages transitent dans des files de type FIFO ("First-In First-Out"), aussi appelées "buffers". On distingue pourtant dans ce cas des communications synchrones et asynchrones, en fonction de la taille du buffer. Si le buffer est de taille "nulle", la communication correspond directement à un "rendez-vous" ou "point de synchronisation", puisque le récepteur est bloqué en attendant un message, et que l'émetteur est bloqué tant que le récepteur n'est pas en attente (de manière générale, dans le modèle considéré, l'émetteur doit attendre que le buffer ne soit pas plein pour y déposer un message). Dans ce cas, la communication est dite "synchrone". Si l'on considère un buffer de taille non nulle, l'émetteur peut y déposer un message dès qu'il n'est pas plein, puis effectuer un autre traitement, alors que le message ne

sera reçu qu'au bout d'un certain délai non déterminé. Ce cas définit une communication "asynchrone" (cela correspond encore à l'ajout d'une forme d'indétermination). Notons que dans le cas d'un buffer de taille infinie, l'émetteur n'est jamais bloqué. Cette définition nous paraît trop arbitraire car les deux modes sont fonctionnellement équivalents (à partir du moment où les temps d'exécution ne sont pas déterminés, le modèle n'est pas fondamentalement modifié si les messages ont un temps de transit non nul). Il est d'ailleurs présenté dans [Chan89] comment ces deux modes peuvent être simulés l'un par l'autre.

Nous préférons conserver de cette approche la notion de communication bloquante, qui est intéressante parce qu'on peut définir des communications non-bloquantes ! Le buffer étant de taille finie (éventuellement nulle), une communication est bloquante si l'émetteur attend une place vide dans le buffer et le récepteur attend une place "pleine" (un message) dans le buffer. De manière générale, les processus attendent dans ce cas la disponibilité des ressources de communication, et ne continuent leur traitement que lorsque le message a été pris en compte. Dans une communication non-bloquante (ce terme est utilisé par exemple dans [Grop94]), on n'attend pas que les ressources soient disponibles. Si au moment de la demande, un message ne peut pas être envoyé ou reçu, alors le processus demandeur en est informé au plus tôt, et continue son traitement en conséquence. Il y a donc ici un mécanisme de test de l'état des ressources. La notion de communication non-bloquante est généralement introduite pour la réception de messages (on considère souvent qu'il y a suffisamment de ressources de mémorisation pour toujours pouvoir envoyer un message, comme dans les modèles à buffer de taille infinie). On parle alors parfois du "test du canal vide" (cité par exemple dans [Bous92]), où un mécanisme prend la décision d'annoncer qu'il y a ou non un message reçu, alors qu'un nouveau message peut arriver pendant la décision. Ce test introduit une forme d'indéterminisme, liée à l'instabilité de la condition testée. L'issue de l'action de communication dépend de facteurs indéterminés dans le modèle et peut avoir des conséquences fonctionnelles, c'est-à-dire sur le résultat du traitement, contrairement aux indéterminations temporelles qui n'ont que des conséquences sur les performances de l'exécution, mais ceci est en fait lié à l'algorithme qui utilise de telles primitives. Nous pensons que la distinction entre des communications "synchrones" et "asynchrones" pourrait être appropriée en tant qu'alternative terminologique aux communications respectivement "bloquantes "et "non-bloquantes". Nous allons voir plus loin comment la définition d'algorithmes "asynchrones" justifie d'autant plus cette correspondance.

3.2.3 Langages synchrones et asynchrones

Les modèles de calcul permettent de réaliser des évaluations d'algorithmes. Pour définir ou mettre en oeuvre un algorithme, il faut cependant disposer d'un formalisme précis de spécification. On utilise pour cela des langages. Un langage doit avant tout permettre d'écrire une spécification fonctionnelle d'un algorithme, si possible indépendamment d'une architecture particulière de mise en oeuvre ou d'une

exécution particulière de cet algorithme [Chan89]. Comme un modèle de calcul, c'est un support, qui joue le rôle d'intermédiaire entre un algorithme et une architecture. Cependant la définition des langages a longtemps été "renversée" dans le sens où à partir d'une architecture donnée, un langage de "programmation" de cette architecture est défini, ce qui permet ensuite la conception d'algorithmes par composition de ses primitives de base. Les algorithmes reflètent alors fortement les contraintes d'implémentation des systèmes de traitement. Des langages plus récents sont souvent définis conjointement à un modèle de calcul relativement générique. Il semble donc que l'on pourrait parler de manière générale de langages synchrones ou asynchrones, en fonction des modèles auxquels ils sont attachés.

Cependant le terme de "langage synchrone" a été récemment défini de manière particulière, dans le contexte des systèmes réactifs temps réel [IEEE91]. Dans ce contexte, les langages synchrones, pour la plupart d'origine française, comme ESTEREL, LUSTRE, SIGNAL, correspondent à des modèles de calcul basés sur les hypothèses suivantes:

- les événements qui constituent l'environnement externe du système sont totalement ordonnés (ils sont déterminés par la donnée d'une valeur véhiculée et d'un instant ou date d'occurrence),
- tous les événements internes au système sont instantanés, et en particulier toutes les sorties du système sont produites de manière synchrone avec les entrées (i.e. les événements de l'environnement de même date) qui sont à leur origine.

On aboutit ainsi à la définition d'un ordre total sur tous les événements du système, correspondant à une détermination complète des dépendances entre tous les événements. Il s'agit en fait d'une sur-détermination, car la plupart des événements internes d'un système complexe ont en général des dépendances "locales". Elle devra de toute façon être relâchée à l'implémentation, où elle n'a pas de sens strictement équivalent.

Les systèmes décrits par ces modèles peuvent être vus comme des types particuliers de "réseaux de processus réactifs" [Bous92], qui sont opposés aux "réseaux de processus séquentiels". Ces derniers sont essentiellement "asynchrones", au sens où ils sont formés de processus qui s'exécutent indépendamment les uns des autres et qui communiquent par passage de messages dans des canaux de type FIFO et de taille non bornée [Bous92]. Les "langages asynchrones" sont définis de manière un peu similaire dans [Berr89] et [Berr93], à la fois par opposition aux langages synchrones et en considérant l'exécution concurrente de processus "faiblement couplés", l'asynchronisme correspondant effectivement à un relâchement des contraintes des modèles synchrones. Cependant la frontière entre les propriétés de la spécification et celles de l'exécution n'y est pas toujours claire. L'indéterminisme est présenté comme une propriété essentielle des langages asynchrones, ce qui nous paraît être une utilisation incorrecte de ce terme, que nous préférons remplacer par "indétermination" dans la plupart des cas. Il nous semble que le terme de "langage synchrone" a été défini dans un contexte trop restreint, ce qui rend difficile la

définition conjointe des "langages asynchrones". De plus, certains de ces langages sont dits "flot de données", alors que du point de vue des modèles de traitement flot de données, il semblerait naturel de qualifier ces langages d'asynchrones...

3.3 Algorithmes asynchrones

Nous avons déjà cité un certain nombre de problèmes d'algorithmique parallèle dans le cadre des modèles de calcul asynchrones, comme l'exclusion mutuelle, la détection de fin globale, etc... Nous n'allons pas discuter ici de manière générale de l'impact de l'asynchronisme sur la conception d'algorithmes, mais plutôt de l'inverse en quelque sorte, puisque les propriétés de certains algorithmes permettent de définir une forme d'asynchronisme qui leur est propre.

Nous devons cependant commencer par citer une terminologie courante, qui consiste à qualifier d'asynchrones les algorithmes qui sont étudiés ou spécifiés par l'intermédiaire de modèles de calcul asynchrones, comme dans [Gibb93] [Shu95] [Lewa96]... Certaines définitions se restreignent aux algorithmes qui sont basés sur l'utilisation de primitives de communication asynchrone (on peut donc dériver plusieurs définitions en fonction de celle d'une communication asynchrone).

Nous adoptons un point de vue radicalement différent, qui correspond assez bien à celui présenté dans [Bert89]. Un algorithme est dit "synchrone" s'il est fonctionnellement indépendant des paramètres liés au temps qui sont indéterminés a priori, comme les temps d'exécution élémentaires, la durée des communications, l'ordre des traitements... Nous entendons par "fonctionnellement indépendant" le fait qu'il respecte un ensemble de dépendances déterminées et définies a priori. On peut aussi dire qu'il doit être équivalent à un algorithme dont l'exécution est parfaitement déterminée.

Avec un algorithme "asynchrone", les dépendances effectivement prises en compte peuvent délibérément varier à l'exécution, en fonction des paramètres liés au temps (il ne s'agit pas d'algorithmes synchrones qui auraient été incorrectement spécifiés !). On peut parler de dépendances "dynamiques", "relâchées" ou "faiblement contraintes", qui correspondent à un indéterminisme, puisque des facteurs indéterminés a priori peuvent avoir des conséquences fonctionnelles, contrairement aux algorithmes synchrones où ces indéterminations temporelles n'ont que des conséquences sur les performances de l'exécution. Nous parlerons aussi d'asynchronisme fonctionnel ou algorithmique.

Voici à titre d'illustration une définition donnée dans [Üres89] d'un algorithme itératif asynchrone:

• Soit F un opérateur de S^n dans S^n (S étant un ensemble quelconque, fini ou infini, dénombrable ou non). Une itération asynchrone correspondant à l'opérateur F et commençant par un vecteur donné $x(0)$, est une suite $\{(x_1(j),\ldots,x_n(j))\}_{j=0,1,\ldots}$ de vecteurs de S^n définie récursivement par:

$$x_i(j) = \begin{cases} x_i(j-1) & si\ i \notin J_j \\ F_i\left(x_1\left(s_1(j)\right),\ldots,x_n\left(s_n(j)\right)\right) & si\ i \in J_j \end{cases},$$

où $J=\{J_j\}_{j=1,2,...}$ est une suite de sous-ensembles non vides de $\{1,2,...,n\}$, et $C=\{(s_1(j),...,s_n(j))\}_{j=1,2,...}$ une suite d'éléments de N^n. De plus, J et C doivent respecter les conditions suivantes, pour tout $i=1,...,n$:

- $s_i(j) \leq j-1$;
- $\lim_{j\to\infty} s_i(j) = \infty$;
- $\forall K, \exists j > K \,/\, i \in J_j$.

A chaque étape j, les composantes de x dont l'indice i apparaît dans J_j sont mises à jour à partir de composantes "retardées" (première condition), calculées aux étapes spécifiées par les $s_i(j)$. La deuxième condition garantit que la valeur d'une composante calculée à une certaine étape n'est pas utilisée une infinité de fois, et que des valeurs plus récentes de cette composante seront finalement utilisées. La troisième condition garantit que toutes les composantes seront mises à jour une infinité de fois.

Dans [Bert89], la définition des algorithmes asynchrones est un petit peu plus générale, et des algorithmes "partiellement asynchrones" sont définis en remplaçant les deux premières conditions par:

$\exists B > 0 \,/\, j - B \leq s_i(j) \leq j-1$, ce qui signifie que les $x_i(j)$ doivent être mis à jour à partir de valeurs dont le retard est borné.

La plupart des algorithmes étudiés par l'intermédiaire des modèles parallèles "asynchrones" sont en fait "synchrones" dans ce sens, car même si certains temps ou ordres d'exécution sont indéterminés, ils utilisent des mécanismes de communication qui correspondent à des points d'attente ou de synchronisation déterminés du point de vue fonctionnel, c'est-à-dire qui assurent le respect d'un ensemble de dépendances déterminé a priori. Dans les algorithmes asynchrones, il n'y a pas nécessairement de points de synchronisation déterminés. Ils correspondent donc à une suppression ou du moins à un relâchement des synchronisations, ce qui a des avantages à plusieurs niveaux. Ils apportent d'abord une plus grande flexibilité de spécification et une meilleure robustesse aux variations de fonctionnement qui peuvent survenir à l'exécution, en particulier dans les systèmes distribués. Ils permettent ensuite un gain en performance par rapport aux algorithmes synchrones, qui peut provenir d'au moins deux facteurs. Ils éliminent d'abord la fameuse "pénalité de synchronisation" [Bert89], puisque (en considérant que l'algorithme est décomposé en un ensemble de processus) chaque processus peut effectuer des traitements de manière ininterrompue et à sa propre vitesse maximale, sans avoir à attendre d'autres processus plus lents ni être pénalisé par la durée d'échange des données entre processus. Chaque processus essaye de produire au plus tôt de nouveaux résultats intermédiaires, à partir des résultats précédents qui lui sont immédiatement disponibles. Cette propagation au plus tôt constitue le deuxième facteur potentiel d'accélération, qui intervient à un niveau fonctionnel. Il n'existe pas de résultat général sur ce facteur, mais certaines expériences montrent que l'utilisation de telles dépendances "relâchées" permet de s'approcher plus rapidement de la solution finale qu'avec une mise à jour globale.

Le relâchement des synchronisations, qui consiste à ne pas attendre le prochain résultat d'un processus voisin mais à utiliser le résultat le plus récent dont on dispose, peut directement correspondre à une utilisation de communications non-bloquantes. En effet, les communications non-bloquantes permettent aux processus de s'échanger des résultats intermédiaires dès qu'ils ont suffisamment "travaillé" pour que la communication soit possible, sans pour autant imposer de temps d'attente. Un processus peut par exemple effectuer une réception non-bloquante pour voir s'il peut obtenir un nouveau résultat de la part d'un processus voisin. S'il n'y a pas de résultat disponible, le processus peut continuer tout de suite son traitement avec l'ancien résultat. Cela correspond au test du canal vide, et constitue un choix indéterministe entre deux dépendances (avec une communication bloquante, le processus utiliserait de manière déterministe le nouveau résultat). Notons cependant que l'utilisation d'un type de communication ne définit pas le type d'un algorithme. Un algorithme synchrone peut aussi être basé sur des communications non-bloquantes. Après une réception non-bloquante où il n'y a pas eu de message reçu, il peut éventuellement effectuer des traitements indépendants avant d'essayer de nouveau d'obtenir le résultat dont il a nécessairement besoin par ailleurs.

Si les variations des temps d'exécution peuvent influer sur le résultat de l'algorithme, il faut cependant que ce résultat corresponde quand même à une solution du problème ! On peut définir plusieurs types d'algorithmes asynchrones. L'asynchronisme fonctionnel peut d'abord n'influer que sur les dépendances et résultats intermédiaires de calcul, le résultat final étant identique pour toutes les exécutions de l'algorithme, et donc équivalent à celui obtenu par un algorithme synchrone. Il s'agit par exemple d'algorithmes itératifs qui convergent vers un point fixe. Des algorithmes dits de "relaxation asynchrone" ou "chaotique", ou des "méthodes itératives asynchrones" sont étudiés dans [Chaz69] [Baud78] [Luba86] [Mitr87] [Bert89] [Üres89] [Bará96] [Üres96], entre autres pour la résolution de systèmes d'équations linéaires, non-linéaires ou d'équations différentielles. Ces références présentent un certain nombre de résultats concernant les conditions de convergence des algorithmes, l'accélération qu'ils permettent sur certaines architectures, ainsi que leur "taux de convergence" (ce problème restant relativement ouvert).

Certains problèmes ne possèdent cependant pas un unique point fixe mais possèdent plusieurs solutions ou des solutions non optimales, en particulier dans des problèmes d'optimisation. De plus, beaucoup d'algorithmes tolèrent une certaine marge d'erreur sur le résultat. Les algorithmes asynchrones peuvent être intéressants dans ces cas car ils fournissent souvent des résultats aussi bons (voire meilleurs) qu'un algorithme synchrone correspondant, tout en permettant des vitesses d'exécution supérieures. Des algorithmes asynchrones utilisés entre autres pour le recuit simulé ou les réseaux de neurones sont par exemple étudiés dans [Lee96] [Gree90] [Kant90].

Enfin, certains algorithmes cherchent à modéliser des systèmes, naturels ou artificiels, qui évoluent dans le temps de manière indéterminée analytiquement, comme les systèmes dynamiques

complexes ou "chaotiques". L'utilisation d'algorithmes synchrones dans ces domaines peut fortement restreindre l'espace des comportements possibles et conduire à des artéfacts gênants. L'asynchronisme algorithmique peut permettre de supprimer ces artéfacts et de découvrir de nouvelles dynamiques. On le voit en particulier dans les modèles d'automates cellulaires [Hoge88] [Naka81] [Toff87].

3.4 Architectures asynchrones

Comme pour les modèles de calcul, il n'existe pas de définition unique pour les "architectures asynchrones". Une architecture est un modèle "structurel", qui définit une décomposition d'un système en un ensemble d'unités interconnectées, ainsi que les mécanismes de contrôle et de communication qui en permettent le fonctionnement conjoint. Contrairement à un modèle de calcul, une architecture doit prendre en compte un certain nombre de contraintes d'implémentation liées entre autres aux méthodologies de conception des systèmes électroniques numériques ainsi qu'au fonctionnement et à la complexité des dispositifs standard qui peuvent être fabriqués dans une technologie cible.

Toute implémentation particulière d'un modèle de calcul asynchrone peut être appelée "architecture asynchrone", comme les architectures MIMD [Dunc90] ou flot de données [Lee94]. Il semble cependant plus approprié de considérer la nature des mécanismes particuliers qui font l'objet d'un choix architectural. On s'intéresse donc aux mécanismes physiques de synchronisation. La synchronisation est nécessaire pour contrôler le traitement et la circulation des données parmi l'ensemble des unités qui composent l'architecture. Elle permet à ces unités de prendre en compte des données en entrée, de cadencer leur traitement, et de produire des données en sortie, "au bon moment", c'est-à-dire en respectant certaines contraintes temporelles ou de séquencement qui permettent physiquement la coordination.

Dans une architecture synchrone, toutes les unités sont contrôlées par un ensemble de signaux globaux, qui proviennent d'une unité de contrôle centralisée. Il existe en général un signal binaire global, appelé "horloge", qui bascule périodiquement et qui suffit à cadencer l'ensemble des unités. A chaque "cycle d'horloge", chaque unité produit de nouveaux résultats, qui seront utilisés au cycle suivant par les unités qui y sont connectées. Toutes les synchronisations sont totalement déterminées.

Dans une architecture asynchrone, il existe des synchronisations locales, qui sont générées par des unités de contrôle réparties (ou locales vis-à-vis des unités). Ces synchronisations définissent la présence ou la validité des données, qui sont produites au terme d'une quantité de traitement indéterminée a priori, et qui sont donc prêtes à être communiquées. S'il existe par ailleurs une horloge globale (ce qui est le cas dans beaucoup de systèmes), la synchronisation est par exemple effectuée par l'activation d'un signal qui indique la validité des données et qui sera pris en compte au niveau de chaque cycle d'horloge. Une simple signalisation de validité n'est cependant pas suffisante s'il n'y a pas d'hypothèse concernant le temps de "consommation" de la donnée transmise, c'est-à-dire le temps minimum qui sépare la prise en

compte de deux données successives. Par exemple, si le nombre de cycles nécessaire à l'unité réceptrice pour traiter une donnée n'est pas déterminé, une signalisation supplémentaire est nécessaire pour indiquer à l'unité émettrice qu'elle doit attendre avant d'envoyer une nouvelle donnée. Cette indétermination du temps de traitement implique donc une signalisation bidirectionnelle entre unités, qui constitue un "protocole de communication", dit par "poignée de main" ("handshake") ou par "requête-acquittement" ("request/acknowledge"). La signalisation qui circule avec la donnée pour indiquer sa validité constitue la requête, alors que celle qui circule en sens inverse de la donnée est un acquittement qui indique la bonne prise en compte de la donnée et autorise l'activation de la requête suivante.

Cet asynchronisme "structurel" (ou architectural) permet de relâcher les contraintes de "timing" des systèmes synchrones, pour se concentrer sur les contraintes de séquencement. C'est le principe de base des "réseaux à front d'onde" ("wavefront arrays") qui combinent les architectures de réseaux systoliques et le traitement flot de données [Kung87] [Kung88]. Le protocole asynchrone peut être utilisé à différents niveaux de granularité. Il est souvent considéré comme assez coûteux et c'est pourquoi la plupart des architectures asynchrones sont en fait mixtes, de type localement synchrones globalement asynchrones (LSGA) ou localement asynchrones globalement synchrones (LAGS).

3.5 Circuits VLSI asynchrones

Nous abordons enfin le domaine des circuits VLSI numériques, qui constituent l'implémentation physique des systèmes de traitement numérique. La définition d'un circuit synchrone correspond à celle d'une architecture synchrone. Il existe un signal d'horloge global qui synchronise tous les signaux qui doivent être mémorisés. A chaque cycle d'horloge, les signaux mémorisés au cycle précédent sont combinés par des "portes" logiques, puis le résultat est stocké soit dans une mémoire de type RAM, soit dans des dispositifs appelés bascules (comme les "flip-flops"), en écrasant le résultat précédent. Le signal d'horloge est connecté à la totalité des bascules, qui permettent la synchronisation des signaux. Les signaux intermédiaires dans la "logique combinatoire" située entre deux étages de bascules, se propagent sans synchronisation, mais cette logique est conçue de telle sorte que la stabilisation des signaux de résultat est assurée au bout d'un temps borné, correspondant à la "chaîne critique" de la logique (qui en particulier ne comporte pas de rebouclage). L'horloge étant globale, la durée minimale d'un cycle doit être au moins supérieure au temps correspondant à la chaîne critique la plus lente dans l'ensemble du système (il faut aussi ajouter des temps supplémentaires pour le bon fonctionnement des bascules).

Comme pour les architectures asynchrones, on peut appeler asynchrone un circuit qui génère des signaux de validité accompagnant les données, en particulier un circuit utilisant un protocole asynchrone entre certaines de ses unités. C'est le cas des mémoires asynchrones, ou des bus de communication asynchrones, utilisés lorsque le nombre de cycles de traitement d'une donnée est indéterminé a priori (ou

du moins peut prendre plusieurs valeurs). Il ne s'agit cependant pas d'un asynchronisme à grain fin, puisque les structures sous-jacentes correspondent à des circuits synchrones, pilotés par une horloge.

Le terme de "circuit asynchrone" est aujourd'hui consacré aux circuits (séquentiels) qui ne possèdent pas d'horloge globale. Ils utilisent uniquement des synchronisations locales. On parle aussi de circuits "auto-synchronisés" ("self-timed"), mais ce terme est un peu vague. Il existe en fait un éventail de techniques de conception de circuits asynchrones, qui définissent autant de sous-classes de circuits, et qui se différencient par les hypothèses temporelles qui sont utilisées [Hauc95] [ElHa95a] [Rena96a].

Les circuits dits "insensibles aux délais" ("delay-insensitive") ne font aucune hypothèse sur les temps de propagation des signaux, ni dans les portes logiques, ni dans les connexions ou "fils" reliant les portes. Ils correspondent à un modèle à "délais non bornés". Toute transition d'un signal y est considérée comme un événement véhiculant une information. La notion d'acquittement y est poussée à l'extrême, car toute transition en entrée d'un opérateur quelconque doit être acquittée par une transition en sortie, avant qu'il ne puisse y avoir de nouvelle transition sur cette entrée. Cette condition très forte limite en particulier les opérateurs utilisables (à une seule sortie) aux fils, aux inverseurs et aux "portes de Muller" (qui réalisent en quelque sorte un "et logique" entre événements, c'est-à-dire qu'il faut une transition sur chacune des entrées pour générer une transition en sortie) [Mart93] [Hauc95]. Ces circuits ne peuvent donc être utilisés que dans des cas assez particuliers car ils ne permettent pas de réaliser les fonctions logiques courantes.

Les circuits dits "quasi-insensibles aux délais" ("quasi delay-insensitive" ou QDI) permettent de remédier à cette limitation. Ils introduisent une hypothèse minimale qui les distingue des circuits "DI", qui est l'hypothèse dite de "fourche isochrone" ("isochronic fork") [Mart93] [Mart90a] [Hauc95]. Une fourche isochrone est une connexion entre une sortie d'un opérateur et plusieurs entrées d'autres opérateurs, dont on fait l'hypothèse que les transitions qu'elle véhicule arrivent en même temps aux extrémités des différentes branches (ou du moins avec une dispersion négligeable devant les autres délais). Cette hypothèse n'est utilisée pour une connexion donnée que lorsqu'elle est strictement nécessaire, le reste du circuit étant véritablement insensible aux délais. Les circuits QDI sont les circuits à usage général les plus robustes qui existent d'un point de vue temporel. Leur conception n'est encore maîtrisée que par peu de personnes mais ils paraissent très prometteurs vis-à-vis de leurs propriétés et de celles des méthodologies de conception qui leur sont associées. Nous y reviendrons à la fin du chapitre 5.

Les circuits dits "indépendants de la vitesse" ("speed-independent") sont beaucoup plus répandus. Ils sont basés sur un modèle où l'on considère que les délais liés aux connexions sont nuls ou négligeables devant les autres délais, les opérateurs étant par contre à délais non bornés [Hauc95]. Cette hypothèse est plus forte que celle de fourche isochrone et peut être difficile à respecter, d'autant plus que les délais dus aux interconnexions ne peuvent plus être considérés comme négligeables dans les technologies récentes. Elle nécessite donc plus de vérifications pendant la conception d'un circuit, et en

particulier pendant les dernières phases (simulations après placement-routage). Ce modèle a cependant permis la réalisation de nombreux circuits asynchrones relativement complexes et robustes, et a permis d'en démontrer un certain nombre de bénéfices. Il s'applique en particulier assez bien à la conception d'opérateurs arithmétiques [ElHa95a]. Nous nous sommes basés sur ce type de circuits dans le chapitre 4.

Les autres catégories de circuits asynchrones sont partiellement ou totalement basés sur des modèles à délais bornés, ce qui signifie que l'on effectue des hypothèses temporelles relativement fortes, ce qui les rapproche des circuits synchrones.

Les circuits basés sur les "micropipelines" [Suth89] sont constitués de chemins de données conçus à l'aide d'un modèle à délais bornés contrôlés par une structure insensible aux délais. Il est apparu par la suite des variantes où les chemins de données sont indépendants de la vitesse. Ces circuits sont donc plus proches des circuits asynchrones précédents que des circuits synchrones. Leur introduction a d'ailleurs été à la base d'un regain d'intérêt pour les circuits asynchrones, en introduisant entre autres la notion de pipeline élastique à grain fin.

Il existe enfin un certain nombre de catégories de circuits asynchrones basés sur des modèles à délais bornés [Hauc95], qui ont été étudiés dès les années 1950 pour concevoir des automates. Ces circuits sont constitués de portes logiques rebouclées, qui définissent plutôt des horloges locales et n'utilisent pas de protocoles de requête-acquittement. Les hypothèses temporelles qui sont utilisées sont du même ordre que celles des circuits synchrones et leur conception est assez délicate, car il faut éliminer les différents types d'"aléas" ("hazards") qu'ils peuvent présenter, ce problème n'ayant trouvé des solutions exactes qu'assez récemment.

Nous ne présentons pas ici les différentes techniques de conception de circuits asynchrones ni les différents choix de protocoles, de codage des données, ou de types de schémas logiques utilisés (voir [Hauc95] [ElHa95a]).

Les bénéfices des circuits asynchrones sont multiples. Grâce à la suppression partielle ou totale des hypothèses temporelles et à l'utilisation de synchronisations locales, ils présentent entre autres les propriétés et potentiels suivants (plus particulièrement pour les circuits DI et QDI):

- une grande robustesse et une auto-adaptation aux variations des conditions de fonctionnement (tension d'alimentation, température) et des paramètres physiques liés à la fabrication,
- une grande modularité, réutilisabilité et un fort potentiel de migration technologique, puisque le seul respect des protocoles aux interfaces permet de remplacer toute implémentation par une autre fonctionnellement équivalente, indépendamment des caractéristiques de vitesse,
- une accélération des traitements qui présentent une variation des temps de calcul en fonction des données, le calcul pouvant être effectué en temps minimum (ou "temps moyen"), en évitant toute pénalité de synchronisation globale (qui implique un temps "pire cas"),

- la suppression des problèmes liés au déphasage du signal d'horloge ("clock skew"), ce qui permet d'utiliser pleinement les potentiels de vitesse des technologies récentes (contrairement aux circuits synchrones qui doivent prendre en compte une marge relative de plus en plus importante),
- des potentiels d'optimisation de la consommation, car il n'y a pas d'énergie consommée pour distribuer un signal d'horloge global, devant commuter rapidement et continuellement, et toute unité qui n'est pas utilisée est "mise en veille" automatiquement sans mécanisme additionnel (type "gated clock"), et ceci jusqu'au niveau porte logique, supprimant toute commutation inutile ("glitch"),
- une facilité de conception globale, puisque la correction fonctionnelle est indépendante des caractéristiques temporelles, ce qui permet de plus de tirer parti d'optimisations seulement locales,
- des propriétés intéressantes pour la testabilité (puisqu'un fonctionnement basé sur les transitions et non les niveaux logiques, permet souvent de tester facilement les fautes de "collage"),
- des potentiels importants de synchronisation, en particulier en ce qui concerne le traitement de plusieurs signaux qui peuvent évoluer de manière indépendante (i.e. non synchronisée) et qui doivent être combinés (comme des signaux d'interruption, ou des signaux provenant de deux systèmes distants).

Ce dernier point permet d'établir un lien entre les circuits asynchrones et la notion d'indéterminisme introduite avec les algorithmes asynchrones et les communications asynchrones (ou non-bloquantes). Les circuits asynchrones sont les seuls à pouvoir gérér de manière sûre la combinaison de signaux dont les instants de transition sont indéterminés. En effet, toute combinaison de ce type est soumise à des problèmes de métastabilité au niveau des dispositifs de mémorisation [Chan73] [Klee87]. Les circuits asynchrones peuvent gérer ces problèmes sans faire d'hypothèse sur la durée de métastabilité, en utilisant des dispositifs simples appelés "arbitres", "éléments d'exclusion mutuelle" ou "synchroniseurs", qui évitent par ailleurs toute propagation de "glitch" ou de signal à un niveau "non logique". Ces dispositifs présentent un indéterminisme structurel (puisqu'on ne peut prédire l'issue d'un phénomène métastable causé par l'arbitrage entre deux signaux dont les transitions sont quasi-simultanées). Ils permettent donc d'implémenter au niveau le plus fin les "dépendances indéterministes" des algorithmes asynchrones et des communications non-bloquantes.

3.6 Conclusion

Nous avons introduit et avons essayé de définir de manière générale la notion d'asynchronisme dans plusieurs contextes s'étendant de la spécification fonctionnelle à l'implémentation matérielle d'algorithmes et d'architectures. Nous avons montré qu'il n'existe pas de notion unique ni de définition universelle voire simplement consensuelle à l'asynchronisme, et qu'il est plus approprié de définir un ensemble de notions applicables selon le niveau de description d'un système. Nous avons cependant choisi de retenir certaines définitions, parfois peu répandues, qui permettent d'entrevoir une première

forme de synthèse, basée sur les notions de dépendance, de relation d'ordre, de synchronisation, d'indétermination et d'indéterminisme.

L'asynchronisme correspond de manière générale à différentes formes d'indétermination liées aux paramètres "temporels" (au sens de la durée ou de l'ordre des événements), qui permettent la spécification des systèmes. Les dépendances (fonctionnelles) constituent l'essence de tout algorithme, et peuvent être respectées grâce à un ensemble de synchronisations, dont la minimisation est avantageuse. Dans le cas des communications et des algorithmes liés aux systèmes parallèles, la notion plus forte d'indéterminisme, permet de définir un asynchronisme fonctionnel et une notion de "dépendance dynamique". Les architectures et les circuits asynchrones sont basés sur la notion de synchronisation locale [Rena97], qui est une forme structurelle de relâchement des synchronisations, et qui supporte naturellement l'indétermination et avec robustesse l'indéterminisme.

La suite de cette thèse se base en partie sur l'évolution récente du niveau de maturité des circuits asynchrones, pour montrer comment l'asynchronisme peut être exploité à plusieurs niveaux, de la conception de circuits VLSI à celle d'algorithmes parallèles.

3.7 Convergence algorithme-architecture

Ce premier travail de réflexion sur les différentes notions d'asynchronisme et leurs implications au niveau de la spécification et des langages, des algorithmes, des architectures et des circuits, nous a amené à reconsidérer le concept d'adéquation algorithme-architecture et à introduire la notion de "convergence" algorithme-architecture. Partons tout d'abord des techniques actuelles de conception de circuits VLSI numériques. Un circuit numérique complexe est aujourd'hui couramment décrit par l'intermédiaire d'un langage dit "de haut niveau" (comme VHDL), avant d'être "synthétisé" par des outils informatiques qui transforment cette spécification en une représentation de plus bas niveau. La spécification initiale peut en fait être considérée à la fois comme un algorithme, une architecture, ou un circuit, bien que ces notions correspondent habituellement à des niveaux de description distincts. On se rend compte que ce sont les dépendances fonctionnelles qui sont essentielles à la description d'un système, et que ces dépendances peuvent être exprimées plus ou moins explicitement selon le niveau de description. C'est pourquoi on peut penser qu'algorithmes et architectures sont des notions qui peuvent converger par l'intermédiaire de formalismes qui tiennent compte conjointement de contraintes fonctionnelles et structurelles. Cette considération n'est pas nouvelle, cependant il s'agit plus souvent d'une "adéquation" algorithme-architecture, qu'une véritable convergence. En effet, une faiblesse essentielle des méthodologies standard de conception est qu'elles reposent sur des transformations ou "mappings" successifs entre différents modèles de description qui sont fortement contraints et qui constituent des "ruptures" dans la spécification descendante. Par exemple, une description en VHDL "synthétisable" ou "comportemental" repose implicitement sur des structures particulières de circuits synchrones à contrôle centralisé.

L'utilisation de ces structures impose par la suite de vérifier à chaque étape la validité des hypothèses correspondantes, par exemple après le placement-routage des circuits réalisés en cellules standard, où l'on doit vérifier tout un ensemble de temps de propagation (dérive d'horloge, temps de maintien et de prépositionnement, chaîne critique...).

Il semble raisonnable de faire disparaître la notion de temps au niveau de la spécification, pour la remplacer par celle de dépendance, ce qui correspond à un relâchement des contraintes de modélisation. Cela permet de se concentrer sur les aspects fonctionnels et structurels, indépendamment des caractéristiques temporelles des dispositifs élémentaires (les éventuelles contraintes de "temps réel" pouvant être prises en compte au plus tard). Pour mettre en oeuvre des dépendances, on utilise explicitement des mécanismes de synchronisation. Un système peut en particulier être décrit par un ensemble de processus concurrents localement synchronisés, que l'on peut décomposer sans rupture par ajout de synchronisations intermédiaires, jusqu'à obtention de processus élémentaires, dont on dispose d'un équivalent matériel, et ceci à un niveau très fin (quasiment niveau transistor). Nous pensons ainsi que l'asynchronisme permet d'envisager des méthodologies qui offrent une plus grande homogénéité entre les différents niveaux d'abstraction d'un système. La notion de convergence algorithme-architecture commence à prendre sens dans ce cas, car il est possible d'utiliser un formalisme unique, qui supporte le passage d'une spécification initiale de haut niveau à un assemblage de dispositifs élémentaires dont le fonctionnement est correct indépendamment de paramètres physiques. Nous reviendrons sur ce type de méthodologie et en donnerons une illustration à la fin du chapitre 5.

Chapitre 4

LE CIRCUIT AMPHIN - UN RESEAU VLSI CELLULAIRE ASYNCHRONE POUR LE FILTRAGE MORPHOLOGIQUE D'IMAGES

4.1 Le projet AMPHIN

Le projet AMPHIN (Architecture Massivement Parallèle Homogène à grain fIN) est une étude conjointe algorithme-architecture qui a débuté pendant un travail de thèse précédent au CNET-Grenoble [Plan94]. Cette étude portait initialement sur la faisabilité d'architectures VLSI cellulaires à grain fin pour la mise en oeuvre de traitements d'images de niveau pixel appartenant à la classe des algorithmes cellulaires itératifs. Les traitements de bas niveau constituent en effet un goulot d'étranglement calculatoire dans tout système de codage ou d'analyse d'images en temps réel, qui s'accentue avec l'évolution des algorithmes et des normes de codage. Une telle architecture trouve son application comme coprocesseur spécialisé dans des systèmes à besoins intensifs en calcul et qui ont une contrainte d'intégration ou de coût (l'utilisation d'architectures parallèles à plus gros grain étant alors écartée). Dans [Plan94], le modèle cellulaire itératif a été appliqué à l'analyse du mouvement, étendant ainsi son utilisation, longtemps restreinte à différentes formes de filtrage (pour la restauration, la détection de contours...), aux problèmes de codage d'images. Nous nous sommes intéressés par la suite à la segmentation spatiale, autre brique algorithmique des systèmes de codage avancé d'images.

Outre son application à l'image, le projet AMPHIN avait aussi comme objectif de développer de nouveaux concepts architecturaux pour la conception des futurs circuits VLSI à haute complexité. Les deux concepts qui ont été combinés sont le parallélisme massif et l'asynchronisme, en particulier pour

des objectifs de haute performance, faible consommation, et facilité de conception. Alors que l'exploitation du parallélisme est un besoin évident pour les futurs systèmes d'analyse/rendu d'images, du point de vue des performances requises, elle apparaît aussi comme une solution potentielle de réduction de la consommation, que l'on peut appeler "parallélisation de consommation" [Plan94]. Les circuits asynchrones possèdent aussi un certain nombre de potentialités de réduction de consommation, en particulier liées à l'absence d'horloge globale et à la "mise en veille" automatique des unités non utilisées. Mais ce sont surtout des potentialités d'amélioration des performances et du cadre de conception de systèmes intégrés complexes qui peuvent être exploitées à travers la combinaison du parallélisme et de l'asynchronisme, dont les propriétés sont entre autres le calcul en temps minimum, la modularité et la "scalabilité". Nous reviendrons sur ces propriétés à la section 4.3.3.

Les propriétés architecturales d'un modèle cellulaire asynchrone ne peuvent être totalement exploitées que si les algorithmes eux-mêmes incorporent les nouvelles possibilités de "désynchronisation". La prise en compte de l'asynchronisme peut en fait renforcer l'interaction algorithme-architecture et ouvrir de nouvelles perspectives de spécification et de conception conjointe. En effet, l'asynchronisme architectural amène à considérer un relâchement des contraintes habituelles de séquencement ou synchronisation (au sens des dépendances réelles), et inspire ainsi la conception de nouveaux algorithmes plus ou moins équivalents d'un point de vue fonctionnel mais possédant des propriétés plus intéressantes du point de vue de leur mise en oeuvre. C'est ce qui a amené à l'étude de l'asynchronisme "fonctionnel", et à son application à des algorithmes de traitements d'image [Plan94] [Priv93b] [Priv95]. Nous poursuivons ici cette étude en étendant l'exploitation de l'asynchronisme fonctionnel à une autre classe de traitements, les filtres morphologiques (en niveaux de gris) par reconstruction [Robi95] [Robi97a], utilisés pour la segmentation spatiale (et spatio-temporelle). Nous nous intéressons surtout à la concrétisation de ces concepts de conception conjointe, à travers une expérience de conception de circuit VLSI asynchrone complexe, mettant en oeuvre un asynchronisme structurel et fonctionnel, et qui a amené à la réalisation complète d'un prototype de réseau VLSI cellulaire asynchrone à grain fin pour le filtrage morphologique d'images en niveaux de gris [Robi96a] [Robi96b] [Robi96c] [Robi97b]. Ce prototype comprend 16×16 processeurs élémentaires asynchrones paramétrables pour un total de 800.000 transistors, et a été fabriqué fin 1995 dans la technologie CMOS 0.5 µm du centre commun CNET / SGS-THOMSON.

4.2 Relaxation asynchrone d'opérateurs morphologiques

4.2.1 Traitement cellulaire itératif et réseaux VLSI de processeurs

Le modèle de calcul cellulaire itératif a été largement utilisé dans des applications de traitement d'images [Rose78] [Rose83] [Pres84] [Plan94]. Son principe est d'effectuer des traitements globaux ou semi-locaux par l'intermédiaire de la propagation itérative de dépendances purement locales entre les

variables attachées aux différents pixels. Les algorithmes itératifs simples, où le nombre d'itérations est spécifié, réalisent des traitements semi-locaux, correspondant à des formes de filtrage (linéaire ou non-linéaire), les dépendances prises en compte appartenant à un voisinage borné restreint. Les algorithmes dits de "relaxation" permettent des traitements globaux par "coopération" de traitements itératifs locaux, et correspondent à une dynamique d'un système complexe. Les itérations sont effectuées jusqu'à ce que certaines contraintes soient satisfaites localement et/ou globalement. Le problème de la convergence doit être étudié pour de tels systèmes, qui ne possèdent pas nécessairement de "point fixe". Le problème de la détection globale de fin de calcul doit aussi être considéré avec attention, pour ne pas pénaliser les performances globales.

Une ancienne génération de réseaux de processeurs synchrones à grain fin fonctionnant en mode SIMD (Single Instruction Multiple Data), comme MPP et CLIP, a précisément été l'application architecturale directe de ce modèle [Batc80] [Dula96] [Pres84]. L'étude systématique et générale des "array-processors" dans [Kung88] a constitué une synthèse remarquable des aspects algorithmiques et architecturaux de la conception VLSI de systèmes de traitement du signal. En particulier, les réseaux à front d'onde ("wavefront arrays") [Kung88] constituent une extension des architectures systoliques, par l'intégration des notions de synchronisation locale et de modèle flot de données. Ces architectures "asynchrones" sont fonctionnellement équivalentes aux structures synchrones étudiées antérieurement, mais possèdent des propriétés structurelles plus intéressantes. Elles n'ont cependant pas été considérées avec un niveau de granularité aussi fin que pour les réseaux synchrones, car la mise en oeuvre de la synchronisation locale et/ou du fonctionnement flot de données a été considérée comme relativement coûteuse. Les nouvelles machines massivement parallèles MIMD basées sur des processeurs RISC sont de même globalement asynchrones et à gros grain, et permettent l'émulation par des mécanismes de coordination locale d'un mode de fonctionnement SPMD (Single Program Multiple Data). Nous présentons dans ce chapitre un modèle de réseau VLSI cellulaire asynchrone à grain fin, qui va au-delà du modèle classique de réseau à front d'onde, non seulement en montrant comment l'asynchronisme peut être implémenté avec une fine granularité, mais en exploitant aussi le concept d'asynchronisme fonctionnel.

4.2.2 Asynchronisme fonctionnel

Le modèle cellulaire itératif permet de réaliser des traitements globaux à partir de la propagation de dépendances purement locales. La forme canonique d'exécution parallèle d'opérateurs itératifs correspond à un mode de mise à jour globalement synchrone: l'ensemble des variables de l'algorithme sont mises à jour à chaque itération, à partir des valeurs calculées à l'itération précédente. Il existe aussi des modes de mise à jour séquentiels dits récursifs, où la nouvelle valeur d'une variable est utilisée pour la mise à jour d'autres variables dans la même itération, correspondant par exemple au balayage par

lignes d'une image. Ces modes récursifs améliorent en général la vitesse de convergence, car ils favorisent la propagation des résultats intermédiaires, mais sont séquentiels et anisotropes par définition.

Des modes de mises à jour asynchrones, qui allient les avantages des deux modes précédents, ont été étudiés depuis longtemps pour des algorithmes de relaxation [Chaz69] [Baud78] [Üres89]. Un des modèles consiste à autoriser à chaque "étape" (il n'y a plus vraiment d'itération globale) la mise à jour d'un sous-ensemble aléatoire de variables à partir de valeurs éventuellement "retardées", c'est à dire antérieures à l'itération précédente [Baud78]. Ce mode correspond donc à un ordre de mise à jour moins contraint. En fait, l'algorithme peut alors s'écrire sous la forme d'un ensemble de processus non synchronisés à "communications" locales, qui peuvent s'exécuter à des vitesses indépendantes. Plus précisément, un nouveau calcul peut être initié localement pour la mise à jour d'une variable, quels que soient les nombres d'itérations effectuées par les autres processus. Ce principe définit un asynchronisme fonctionnel [Plan94], c'est à dire au niveau des dépendances algorithmiques, en contraste avec l'asynchronisme architectural ou structurel qui fait référence à une structure d'implémentation. D'un point de vue purement algorithmique, l'intérêt de l'asynchronisme fonctionnel est de minimiser les contraintes de dépendances et d'accélérer la convergence par rapport à une mise à jour plus contrainte (c'est une accélération "fonctionnelle", i.e. en termes de nombres d'opérations). Du point de vue de sa mise en oeuvre, il offre des potentiels d'optimisation de la synchronisation dans une architecture parallèle et par suite d'accélération à la fois "fonctionnelle", et "structurelle" (i.e. liée à l'optimisation des temps d'exécution permis par la structure). Nous montrons plus loin comment ces potentiels peuvent être exploités grâce à l'asynchronisme structurel.

4.2.3 Filtres morphologiques pour la segmentation d'images

L'approche basée sur la morphologie mathématique pour la segmentation d'images non supervisée est particulièrement intéressante, car relativement simple, robuste et adaptée à des images quelconques. L'algorithme présenté dans [Sale92] comporte quatre étapes: le pré-traitement, l'extraction d'éléments représentatifs, la décision puis l'estimation de qualité. L'étape de pré-traitement est une phase de simplification dont le but est de générer des régions dont les intensités sont quasi-constantes. Elle est implémentée avec des filtres morphologiques par reconstruction [Vinc93], qui préservent l'information de contour de l'image originale. On s'intéresse en particulier à ces filtres non-linéaires dans la suite.

Les opérateurs de base du filtre de simplification sont l'érosion et la dilatation en niveaux de gris:

- érosion: $y_p = \varepsilon_n\left(x_p\right) = Min\left\{x_{p+q}, q \in M_n\right\}$,
- dilatation: $y_p = \delta_n\left(x_p\right) = Max\left\{x_{p-q}, q \in M_n\right\}$,

où l'indice p est le vecteur de coordonnées du pixel x, M_n un ensemble de coordonnées relatives, appelé élément structurant plat, qui spécifie le motif binaire à combiner avec l'image, et n la taille de cet élément structurant. Ce dernier paramètre contrôle le niveau de simplification. Par exemple, si on utilise

un élément structurant de grande taille, alors le résultat de la segmentation ne fera apparaître que les plus grandes composantes. Pour *n=1*, une dilatation élémentaire consiste donc à réaliser en chaque pixel un maximum (dit "local") sur le voisinage élémentaire.

Le processus de reconstruction qui régénère les contours de l'image originale, tout en maintenant le niveau de simplification désiré, est basé sur la dilatation "géodésique" de taille unitaire:

- dilatation géodésique: $y_p = \delta^{(1)}\left(x_p, r_p\right) = Min\left\{\delta_1\left(x_p\right), r_p\right\}$,

où (r_p) est l'image de référence appelée aussi image "masque", dans laquelle on reconstruit l'image "marqueur". Cet opérateur, qui consiste donc à réaliser un minimum "point à point" ("pointwise") à la suite d'une dilatation élémentaire, est appliqué de manière itérative jusqu'à convergence globale. C'est en fait un opérateur idempotent (i.e. tel que: $\exists k, f \circ f^k = f^k$). L'application de dilatations géodésiques unitaires à la suite d'une érosion de taille *n* (qui constitue la valeur d'initialisation de l'image marqueur), l'image de référence (r_p) étant égale à l'image originale (sur laquelle on a appliqué l'érosion), définit ainsi une ouverture par reconstruction:

- ouverture par reconstruction: $y_p = \delta^{(1)}\left(\ldots\delta^{(1)}\left(\delta^{(1)}\left(\varepsilon_n\left(x_p\right), x_p\right)\right)\ldots, x_p\right)$ (Equation 1)

La fermeture par reconstruction est définie de manière duale (en échangeant les opérations de minimum et de maximum), et le filtre "d'ouverture-fermeture par reconstruction", qui est le filtre de simplification effectivement utilisé pour le pré-traitement, est la composition d'une ouverture et d'une fermeture. La Figure 11 illustre l'utilisation de tels opérateurs morphologiques.

Figure 11: Exemple de filtre morphologique. De gauche à droite: image originale, érosion de taille 32, et ouverture par reconstruction.

4.2.4 Reconstruction parallèle asynchrone

a) Relâchement des contraintes de dépendances

L'application de l'asynchronisme fonctionnel à des algorithmes itératifs de traitement d'images [Plan94] [Priv93b] permet d'élargir l'espace de conception d'algorithmes parallèles. On s'intéresse ici à son application aux filtres morphologiques [Robi95] [Robi97a]. Ce modèle peut être défini à partir d'une transformation de l'algorithme à itération globale en un ensemble de processus à itération locale chacun lié à un pixel, où certaines contraintes sur les indices sont relâchées. Remarquons d'abord que la notation globale de l'Equation 1 est équivalente à celle de l'Equation 2, où (r_p) est l'image originale.

$$x_p(k) = Min\left(Max\left\{ x_{p-q}(k-1), q \in M_1 \right\}, r_p \right), \qquad \text{(Equation 2)}$$

avec $x_p(0) = \varepsilon_n(r_p)$.

Au lieu d'utiliser le même indice d'itération k pour tous les processus-pixels, un indice d'itération locale k_p peut être introduit, conduisant à l'Equation 3, où chaque processus a sa propre vitesse d'évolution.

$$x_p(k_p) = Min\left(Max\left\{ x_{p-q}(k_{p-q}), q \in M_1 \right\}, r_p \right) \qquad \text{(Equation 3)}$$

Alors que k_p s'accroît à la même vitesse pour tous les processus dans l'Equation 2, la nouvelle contrainte est seulement que chaque k_p est une fonction croissante du temps.

Tout en étant une approche complètement parallèle, ce mode de mise à jour tire partie des propriétés du mode récursif, car lorsqu'une variable est mise à jour, sa nouvelle valeur peut être utilisée par un processus voisin dès qu'il commencera un nouveau calcul, ce qui augmente donc la probabilité de propager de l'information utile entre les étapes successives.

b) Convergence

Des simulations ont montré que la reconstruction itérative asynchrone converge et atteint exactement la même solution que l'algorithme de référence synchrone. Même si la fonction itérée n'est pas linéaire, ceci n'est pas réellement une surprise car elle est croissante, bornée et possède un point fixe (elle est même idempotente). Bien qu'il devrait être possible de se ramener à une preuve beaucoup plus générale, nous donnons ci-dessous une démonstration spécifique de l'équivalence synchrone/asynchrone pour la reconstruction positive [Robi97a].

Définition des itérations synchrones et asynchrones

Soit δ la dilatation géodésique de taille unitaire localement définie par:

$\delta(f,r)(p)=Min(r(p),Max(f(p'),p'\in N(p)))$,

où p est un pixel, f(p) sa valeur (niveau de gris), r(p) la valeur de référence correspondant au pixel de même position dans l'image source, et N(p) l'ensemble des pixels qui appartiennent au 5-voisinage de p.

Soient δ_s et δ_a les dilatations géodésiques respectivement synchrone et asynchrone définies récursivement par:

$\forall p, \delta_s^0(p)=\delta_a^0(p)=\varepsilon_n(r)(p)$

(la valeur initiale est égale au résultat d'une érosion de taille n appliquée à l'image de référence) et

$\forall j, \forall p, \delta_s^{j+1}(p)=\delta(\delta_s^j,r)(p)$,

$\forall k, \forall p, \delta_a^{k+1}(p) = \delta(\delta_a^k,r)(p)$ si $p=\pi(k)$

$= \delta_a^k(p)$ si $p\neq\pi(k)$,

où π est une fonction aléatoire qui choisit un pixel p à l'étape k, de telle sorte que chaque pixel soit sélectionné un nombre de fois non borné:

$\forall p, \forall K, \exists k>K / \pi(k)=p$.

Remarquons que δ_s and δ_a sont croissantes par rapport à l'indice d'itération:

$\forall p, j'\geq j \Rightarrow \delta_s^{j'}(p)\geq\delta_s^j(p)$ et $k'\geq k \Rightarrow \delta_a^{k'}(p)\geq\delta_a^k(p)$

et qu'elles sont bornées par l'image de référence:

$\forall j, \forall p, \delta_s^j(p)\leq r(p)$ et $\forall k, \forall p, \delta_a^k(p)\leq r(p)$.

<u>Preuve de l'équivalence entre les deux fonctions</u>

i) Montrons que: $\forall j, \exists k / \forall p, \delta_a^k(p)\geq\delta_s^j(p)$.

On a tout d'abord: $\forall p, \delta_a^1(p)\geq\delta_s^0(p)$.

Faisons maintenant l'hypothèse que: $\exists j,k / \forall p, \delta_a^k(p)\geq\delta_s^j(p)$.

Considérons le pixel p.

On a soit: $\delta_a^k(p)\geq\delta_s^{j+1}(p)$,

ou: $\delta_a^k(p)<\delta_s^{j+1}(p)$.

Considérons ce deuxième cas.

Par définition: $\forall q, \delta_s^{j+1}(q)=\delta(\delta_s^j,r)(q)=\text{Min}(r(q),\text{Max}(\delta_s^j(q'),q'\in N(q)))$,

et: $\exists k'>k / \delta_a^{k'}(p)=\delta(\delta_a^{k'-1},r)(p)=\text{Min}(r(p),\text{Max}(\delta_a^{k'-1}(p'),p'\in N(p)))$.

Donc: $k'-1\geq k \Rightarrow \forall q, \delta_a^{k'-1}(q)\geq\delta_a^k(q)\geq\delta_s^j(q)$

$\Rightarrow \forall q, \text{Max}(\delta_a^{k'-1}(q'),q'\in N(q))\geq\text{Max}(\delta_s^j(q'),q'\in N(q))$

$\Rightarrow \forall q, \text{Min}(r(p),\text{Max}(\delta_a^{k'-1}(q'),q'\in N(q)))\geq\text{Min}(r(p),\text{Max}(\delta_s^j(q'),q'\in N(q)))$

$\Rightarrow \delta_a^{k'}(p) \geq \delta_s^{j+1}(p)$.

Ceci étant vérifié pour tout pixel p, posons: K=Max(k'(p)),

alors: $\forall p, \delta_a^{K}(p) \geq \delta_s^{j+1}(p)$.

Donc: $\exists j,k / \forall p, \delta_a^{k}(p) \geq \delta_s^{j}(p) \Rightarrow \exists k' / \forall p, \delta_a^{k'}(p) \geq \delta_s^{j+1}(p)$,

et par induction : $\forall j, \exists k / \forall p, \delta_a^{k}(p) \geq \delta_s^{j}(p)$.

ii) Montrons que: $\forall k, \exists j / \forall p, \delta_s^{j}(p) \geq \delta_a^{k}(p)$.

On a tout d'abord: $\forall p, \delta_s^{1}(p) \geq \delta_a^{0}(p)$.

Faisons maintenant l'hypothèse que: $\exists j,k / \forall p, \delta_s^{j}(p) \geq \delta_a^{k}(p)$.

Considérons le pixel p tel que: $\pi(k+1)=p$.

Alors, par définition:

$\delta_s^{j+1}(p)=\delta(\delta_s^{j},r)(p)=Min(r(p),Max(\delta_s^{j}(p'),p' \in N(p)))$, et

$\delta_a^{k+1}(p)=\delta(\delta_a^{k},r)(p)=Min(r(p),Max(\delta_a^{k}(p'),p' \in N(p)))$.

Donc: $\forall q, \delta_s^{j}(q) \geq \delta_a^{k}(q) \Rightarrow \delta_s^{j+1}(p) \geq \delta_a^{k+1}(p)$.

De plus: $\forall p' \neq p, \delta_a^{k+1}(p')=\delta_a^{k}(p')$

$\Rightarrow \forall p' \neq p, \delta_s^{j}(p') \geq \delta_a^{k+1}(p')$

$\Rightarrow \forall p' \neq p, \delta_s^{j+1}(p') \geq \delta_a^{k+1}(p')$.

Finalement: $\forall p, \delta_s^{j+1}(p) \geq \delta_a^{k+1}(p)$.

Donc: $\exists j,k / \forall p, \delta_s^{j}(p) \geq \delta_a^{k}(p) \Rightarrow \exists j' / \forall p, \delta_s^{j'}(p) \geq \delta_a^{k+1}(p)$,

et par induction : $\forall k, \exists j / \forall p, \delta_s^{j}(p) \geq \delta_a^{k}(p)$.

De i) et ii), et étant donné que δ_s et δ_a sont bornées, on peut conclure que:

$\forall p, Max_j(\delta_s^{j}(p))=Max_k(\delta_a^{k}(p))$, i.e. $\forall p, \delta_s^{\infty}(p)=\delta_a^{\infty}(p)$,

ce qui signifie que les reconstructions synchrone et asynchrone convergent vers le même résultat.

c) Simulation et accélération fonctionnelle

Les différents algorithmes et modes de mise à jour ont été programmés en langage C. La mise à jour asynchrone a été simulée par une mise à jour séquentielle des pixels dont l'ordre (la position du pixel mis à jour à chaque étape) est tiré aléatoirement. Pour comparer les mises à jour synchrone et asynchrone, on peut définir dans tous les cas une itération comme la mise à jour d'un nombre de pixels égal à la taille de l'image. Dans le mode asynchrone, certains pixels peuvent donc être mis à jour plusieurs fois pendant la même "itération", alors que d'autres pixels n'ont pas été sélectionnés. Des simulations ont confirmé que le mode asynchrone peut améliorer de manière significative la vitesse de

convergence de la reconstruction morphologique, par rapport au mode globalement synchrone. La Figure 12 montre plusieurs images accompagnées d'une ouverture par reconstruction, et le Tableau 1 indique pour chacune d'elles le nombre d'itérations nécessaires pour atteindre l'idempotence, dans différents modes de mise à jour. Le mode synchrone parallèle peut être considéré comme l'algorithme de référence (cf Equation 2). La mise à jour séquentielle ("récursive") par balayage standard permet bien une accélération fonctionnelle, mais n'est pas parallèle par définition (bien qu'elle pourrait être pipelinée). La mise à jour asynchrone permet d'accélérer la convergence de manière semblable à l'algorithme récursif, tout en étant totalement parallélisable. Elle permet de diviser le nombre d'itérations d'environ un facteur 2 par rapport à l'algorithme parallèle synchrone. L'intervalle donné pour le nombre d'itérations en mode asynchrone correspond à différentes initialisations de la fonction pseudo-aléatoire qui détermine l'ordre de mise à jour des pixels.

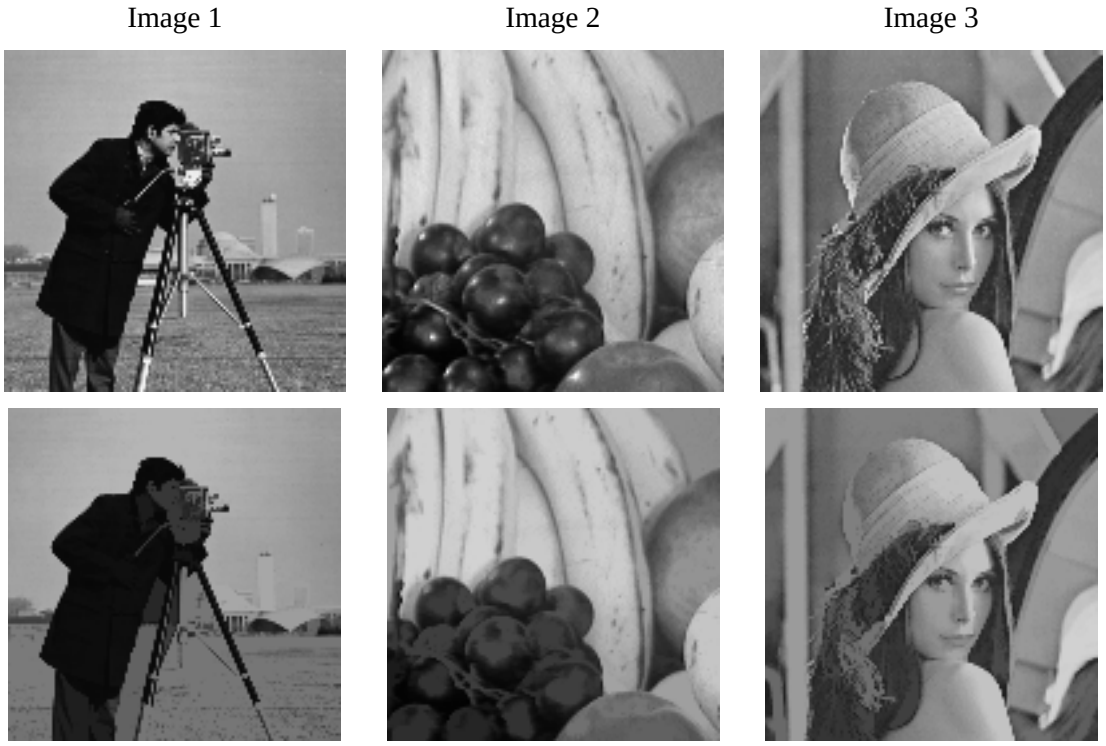

Figure 12: Images originales (en haut) et leur ouverture par reconstruction après érosion de taille 8 (en bas)

De manière à accélérer encore la reconstruction itérative, on pourrait imaginer d'utiliser un mécanisme de contrôle local de convergence, qui arrêterait les itérations correspondant à un pixel dès que plusieurs mises à jour successives de ce pixel n'ont pas modifié sa valeur [Robi95]. On remarque en effet que l'on obtient une solution proche du résultat final, avec un nombre d'itérations assez inférieur à celui nécessaire pour atteindre l'idempotence. Les dernières itérations ne modifient que peu de pixels par rapport au nombre total de pixels contenus dans l'image, ce qui constitue non seulement une "perte de

temps" mais aussi un "coût énergétique" inutile, si l'on considère qu'un résultat intermédiaire est tout aussi valable pour obtenir une segmentation à partir de cette simplification.

Mode de mise à jour	Image 1	Image 2	Image 3
Synchrone parallèle	194	209	250
Synchrone séquentiel	98	140	97
Asynchrone parallèle	105-109	114-121	128-135

Tableau 1: Nombre d'itérations pour atteindre l'idempotence pour une ouverture par reconstruction en fonction du mode de mise à jour

d) Extension aux opérateurs à itération finie

L'idée qu'un résultat intermédiaire ou approximatif de la reconstruction pouvait permettre d'obtenir des résultats équivalents de segmentation, nous a amené à étendre l'application de l'asynchronisme fonctionnel aux opérateurs morphologiques élémentaires, comme la dilatation et l'érosion, qui ne sont pas des processus convergents mais à itération finie comme les filtres linéaires classiques. L'asynchronisme fonctionnel a pour effet dans ce cas de modifier la notion d'élément structurant (plat), qui correspond normalement à un motif binaire donné qui est "combiné" à l'image de manière identique en chaque pixel. Si on considère l'application asynchrone en chaque pixel d'un nombre donné de dilatations ou d'érosions élémentaires, chaque processus associé à un pixel possédant une vitesse aléatoire indépendante des autres, on obtient un ensemble de propagations spatialement bornées mais de forme aléatoire (Figure 13b), alors que l'opérateur "synchrone" fait apparaître de manière homogène l'élément structurant comme motif de propagation unique (Figure 13a). Le résultat peut paraître complètement différent de l'approche classique et assez "artistique". Cependant les deux modes réalisent en fait une tâche identique, qui consiste à obtenir une image marqueur pour la reconstruction, comprenant des zones de niveau de gris constant d'une certaine taille, construites pour éliminer les composantes de taille inférieure. De plus, la forme de l'élément structurant est souvent choisie de façon arbitraire, le paramètre important étant surtout sa taille. Dans ce sens, le mode asynchrone est donc plus général et homogène.

La pertinence d'une telle approche est confirmée par les résultats de simulation de l'algorithme de reconstruction appliqué aux images marqueur obtenues par érosion et dilatation de taille *n* avec les modes synchrone et asynchrone. La Figure 13c montre le résultat d'une ouverture-fermeture basée sur les opérateurs standard (correspondant à la Figure 13a) et la Figure 13d montre le résultat obtenu avec les

opérateurs asynchrones (correspondant à la Figure 13b). On peut remarquer que ces résultats sont très semblables (on pourrait même croire à première vue qu'ils sont identiques). Les itérations asynchrones permettent donc d'améliorer les propriétés algorithmiques en obtenant un résultat tout aussi exploitable.

a) dilatation synchrone b) dilatation asynchrone

c) ouverture-fermeture par recons. synchrone d) ouverture-fermeture par recons. asynchrone

Figure 13: Exemple d'application de l'asynchronisme fonctionnel aux opérateurs à itération finie

e) Partitionnement

Nous avons considéré jusqu'ici la parallélisation totale de la reconstruction, en associant un processus à chaque pixel de l'image. Pour la mise en oeuvre de cet algorithme par un réseau de processeurs intégrés, il va cependant falloir considérer un degré de parallélisme intermédiaire. Pour cela,

on réalise un partitionnement, qui consiste à assigner les calculs du problème initial de grande taille à un ensemble de taille inférieure de processeurs [Kung88]. D'un point de vue purement algorithmique, on peut considérer le partitionnement comme une mise à jour intermédiaire entre le mode totalement parallèle et le mode totalement séquentiel. Un partitionnement LPGS (Localement Parallèle Globalement Séquentiel) classique consiste à effectuer un traitement parallèle par blocs, en balayant séquentiellement les différents blocs. Nous reviendrons sur le partitionnement au chapitre 5, de manière générale et aussi spécifique à l'algorithme de reconstruction. Nous considérons dans la suite une mise à jour parallèle et asynchrone sur un bloc de pixels, l'algorithme étant appliqué à toute l'image par balayage des différents blocs.

4.3 Une architecture VLSI cellulaire structurellement et fonctionnellement asynchrone

Nous avons conçu un processeur asynchrone massivement parallèle qui implémente le modèle fonctionnel décrit plus haut. Un processeur élémentaire (PE) asynchrone est associé à chaque pixel. Les paragraphes suivants montrent comment l'asynchronisme architectural à grain fin (des "circuits asynchrones") est mis au service du concept algorithmique, et décrivent l'architecture du réseau de processeurs et ses principales caractéristiques.

4.3.1 L'asynchronisme architectural au service de l'asynchronisme algorithmique

Le modèle de calcul fonctionnellement asynchrone décrit précédemment n'est effectif en pratique que s'il existe une dispersion entre les temps de mise à jour des différentes variables. Les calculs effectués par les différents PEs se désynchronisent alors dans le temps. C'est la première condition nécessaire qui permet à des variables calculées à des itérations différentes d'être utilisées pour un nouveau calcul local.

La seconde condition nécessaire est que le système de communication entre PEs ne doit pas resynchroniser localement les itérations. Un PE donné doit donc être capable de lire les variables des processeurs voisins sans attendre que ceux-ci aient fini leurs calculs en cours.

L'asynchronisme architectural est la clé de notre implémentation. La première condition est remplie par l'utilisation d'opérateurs asynchrones à détection de fin de calcul, dont le temps de calcul varie en fonction des données, et qui permettent de réaliser les itérations en un temps minimum. La deuxième condition est remplie grâce à l'utilisation d'interfaces de communication inter-PE spécifiques.

4.3.2 Architecture du réseau de processeurs

Le réseau de processeurs élémentaires est organisé suivant une grille bi-dimensionnelle à maille carrée, dans laquelle chaque processeur communique avec ses quatre plus proches voisins (Figure 14).

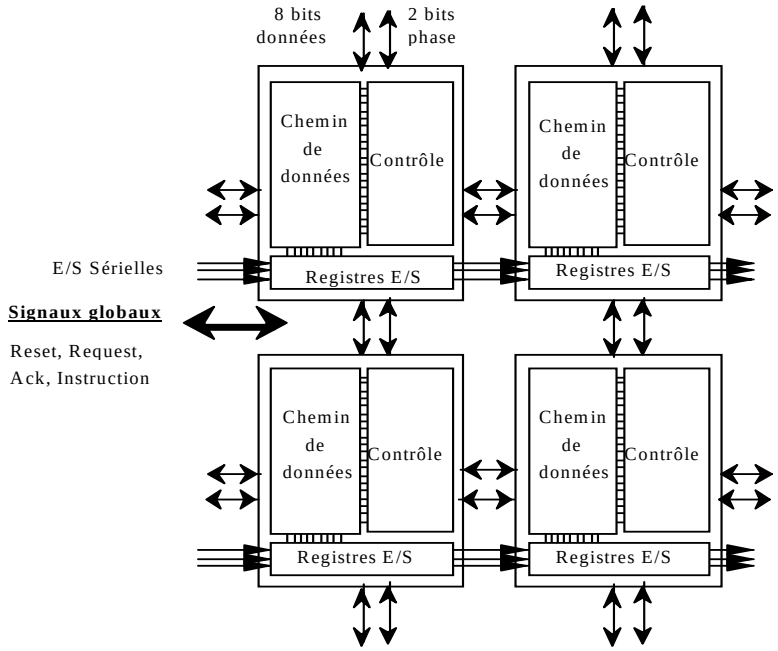

Figure 14: Architecture du coeur du réseau de processeurs

Chaque PE met à jour sa propre variable d'état de manière non synchronisée par rapport aux voisins. Nous avons implémenté un réseau de 16×16 processeurs-pixel dans un circuit VLSI [Robi96b] [Robi96c], composé d'un coeur de 14×14 PEs et de cellules de bord qui contiennent principalement des registres d'E/S. Le réseau de processeurs échange des données avec l'extérieur par le biais de trois liens sériels par ligne de PEs. Les pixels sont introduits dans le réseau par l'intermédiaire de deux liens d'entrée (un pour les données intermédiaires, un autre pour l'image de référence, cf Equation 1), selon un mode bit-série de manière à limiter le nombre de plots. Seize lignes de seize pixels sont stockées dans des registres à décalage inclus dans les PEs pour chacune des deux images d'entrée, sous le contrôle de signaux d'horloge dédiés aux E/S (tout le reste du circuit étant auto-séquencé, c'est-à-dire n'utilisant pas de signal d'horloge global). En parallèle, les données du calcul précédent sortent du réseau par l'intermédiaire du troisième lien série. Ce schéma de pipeline permet aux PEs d'effectuer des calculs pendant que les prochaines données sont introduites et que les résultats précédents sont extraits.

L'instruction de commande peut être initialement stockée dans le réseau à l'aide d'une entrée bit-parallèle qui est distribuée à tous les PEs. Enfin, trois signaux de contrôle sont fournis à l'ensemble des processeurs pour contrôler l'exécution. Le signal "Reset" permet de positionner les PEs dans un état initial déterminé. Le signal "Request" permet au système hôte de lancer un calcul dans le réseau dès que ce dernier a été initialisé avec l'instruction et les pixels d'entrée. Lorsque le calcul est terminé, le signal "Ack" devient actif et un autre calcul peut alors être initié. De manière à optimiser le temps global de calcul, il faut que le temps d'entrée/sortie des données soit égal au temps de calcul moyen correspondant à l'instruction.

Comme il manque des voisins aux PEs des bords du réseau, ceux-ci ne font pas de calcul mais se chargent simplement de répondre correctement aux requêtes de leurs voisins appartenant au coeur. Ces PEs dégradés sont initialisés comme les autres PEs, puis délivrent toujours la même valeur durant les itérations.

4.3.3 Propriétés architecturales

a) Performances

L'implémentation d'un tel réseau de processeurs asynchrones présente un certain nombre de propriétés intéressantes. La propriété majeure est la capacité d'exécuter un ensemble de calculs en temps minimum ou "moyen", plutôt que sur une base de type "pire cas", grâce à l'utilisation d'un additionneur à propagation/génération de retenue et à détection de fin de calcul et d'un schéma spécifique de communication inter-PE. Puisque les PEs n'ont pas à attendre le plus lent d'entre eux à chaque itération, le temps total de calcul est de l'ordre du temps moyen de calcul d'une itération multiplié par le nombre d'itérations. On peut donc clairement tirer parti de la structure itérative de l'algorithme, puisque plus le réseau exécute d'itérations sans aucune synchronisation, plus le temps total de calcul s'approche du cas correspondant au temps moyen. En fait, le temps de calcul est moyenné sur l'ensemble des itérations de l'ouverture ou fermeture par reconstruction. Avec un réseau de processeurs synchrones, une synchronisation globale serait nécessaire à la fin de chaque itération. Au contraire, dans notre réseau asynchrone, une seule synchronisation globale est requise à la fin du calcul de l'ensemble des itérations. Le taux d'accélération théorique de cette implémentation, par rapport à une implémentation synchrone, est donc le produit de deux termes: le temps de calcul du pire cas divisé par le temps de calcul moyen d'une itération, et le taux d'accélération du taux de convergence dû à l'asynchronisme fonctionnel (qui s'exprime comme le nombre total d'itérations nécessaires avec une mise à jour synchrone divisé par le nombre correspondant au mode asynchrone).

b) Consommation

La puissance consommée peut être optimisée de plusieurs façons grâce au parallélisme et à l'asynchronisme. Dans un circuit asynchrone, chaque PE ne consomme que lorsque nécessaire (le fonctionnement flot de données étant mis en oeuvre jusqu'au niveau le plus fin) et s'arrête automatiquement à la fin du calcul, sans coût additionnel. Il n'y a pas de pics de consommation correspondant à des fronts d'horloge, l'activité électrique s'étalant dans le temps, et aucune énergie n'est consommée pour piloter les signaux d'horloge dans tout le circuit. De plus, grâce à la robustesse de l'implémentation asynchrone, la tension d'alimentation peut être diminuée sans modifier le comportement fonctionnel du système. En effet, la suppression des hypothèses temporelles (ou insensibilité aux délais) permet à un circuit asynchrone de fonctionner à la vitesse maximale en

s'adaptant automatiquement aux conditions de fonctionnement (tension, température). En fonction des applications, la puissance consommée peut donc être ajustée en adaptant la tension d'alimentation, ce qui permet de s'approcher de la consommation optimale.

Rappelons l'argument de "parallélisation de consommation" [Plan94]. Plutôt que de considérer une optimisation de la surface de silicium, consistant à utiliser le parallélisme minimal permettant d'atteindre les objectifs de débit de calcul, l'optimisation de la consommation peut consister à faire fonctionner le circuit avec une tension d'alimentation minimale (une technologie CMOS 0.5µ permettant de descendre facilement à la moitié de la valeur nominale) et à utiliser un parallélisme plus important que nécessaire, suffisant pour contrebalancer la diminution de la vitesse du circuit et conserver la même performance. Si le taux d'accélération est linéaire et si la consommation est proportionnelle au nombre d'unités (en première approximation), alors la consommation du circuit lent mais "surparallèle" est diminuée d'un facteur égal au carré du rapport des tensions (permettant ainsi de la diviser facilement par quatre).

c) Facilité de conception et scalabilité

Nous ne détaillerons pas ici une argumentation générale de "facilité de conception" des circuits asynchrones, qui implique de comparer les limitations fondamentales des flots de conception de circuits VLSI. Remarquons tout de même que l'on obtient une implémentation parfaitement locale qui n'utilise pas de signal global d'horloge et qui est basée sur un ensemble de blocs auto-séquencés, ce qui accroît la modularité, la "scalabilité" (c'est-à-dire la conservation des propriétés par extension ou changement d'échelle) et la fiabilité du système. Même pour les registres à décalage synchrones, le schéma de pipeline adopté reste modulaire et scalable, car les signaux de données et d'horloge sont "bufferisés" dans chaque PE et propagés le long des lignes de processeurs. La synchronisation globale est effectuée en tenant compte des signaux de fin de calcul de l'ensemble des PEs. De manière à préserver la modularité de la structure, les informations de fin de calcul sont assimilées par lignes, de PE en PE, et finalement combinées pour générer le signal d'acquittement global. La diffusion de l'instruction de commande dans tous les PEs n'est faite qu'une fois pendant la phase d'initialisation, et ne constitue pas un point bloquant.

4.4 Conception d'un processeur élémentaire

4.4.1 Approche "standard cells"

Depuis les travaux de thèse présentés dans [ElHa95a], différents styles de "logiques" et de conception ont été étudiés dans le groupe de Marc Renaudin pour implémenter des circuits asynchrones. A partir de cette étude et de l'expérience acquise à travers la conception et la fabrication de plusieurs circuits asynchrones [Rena94a] [Rena94b] [ElHa95b], nous avons choisi (en 1995) d'utiliser des portes logiques à précharge basées sur la logique DCVS (Differential Cascode Voltage Switch Logic) [Erde84]

pour concevoir les chemins de données, et d'utiliser des portes logiques standard CMOS pour concevoir les parties de contrôle.

De manière à être capables de concevoir rapidement des circuits indépendants de la vitesse basés sur des portes logiques à précharge, et pour ne pas avoir à concevoir à chaque fois les mêmes cellules, une librairie de cellules standard asynchrones avait été spécifiée et conçue [ElHa95a]. Il s'agissait de disposer de fonctions logiques à précharge qui pouvaient être directement associées à des portes CMOS standard dans un même circuit. Les portes logiques à précharge ont donc été conçues sous forme de "standard cells" en respectant le format des bibliothèques de cellules CMOS disponibles au CNET, basées sur la technologie CMOS 0.5µm à trois niveaux de métal du centre commun CNET/SGS-THOMSON. Cette approche permet de concentrer les moyens de conception sur les seules cellules asynchrones spécifiques, en laissant la tâche de maintenir la librairie de cellules standard CMOS à la fonderie.

Une porte logique à précharge est constituée de deux parties, un arbre fonctionnel et une charge [Rena94a]. La Figure 15 montre la structure d'une porte logique DCVS à détection de fin [ElHa95a] [VanD95]. Les entrées et les sorties sont en double rail, et sont accompagnées d'un signal de requête et d'un signal d'acquittement, le bloc fonctionnant selon le protocole à quatre phases.

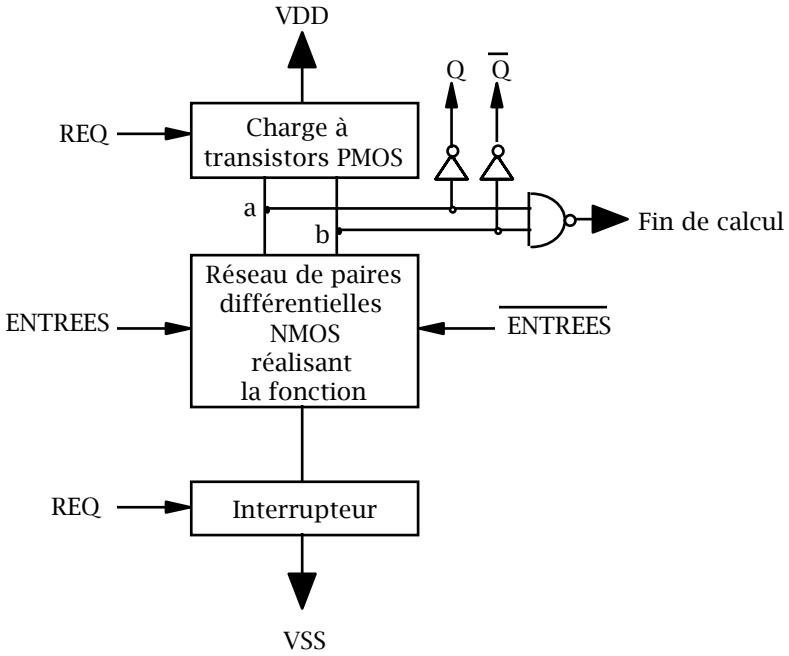

Figure 15: Structure d'une porte logique DCVS avec génération d'un signal de fin [VanD95]

Deux types de charges ont été conçues, une charge pseudo-statique, couramment utilisée dans les cellules DCVS, et une charge avec "latch". Cette dernière permet de tirer partie de la possibilité de

mémorisation d'une donnée, comme expliqué dans [Rena96]. Cette structure s'appelle LDCVSL (Latched DCVS) et peut être exploitée pour concevoir des structures en anneaux très compactes.

Certaines cellules qui sont utilisées très souvent dans les parties de contrôle de circuits asynchrones ont été ajoutées: la porte de Muller (parfois appelée "C-element"), la porte de Muller généralisée, une cellule d'exclusion mutuelle, une bascule Q-Flop, une bascule RS.

La conception complète d'un circuit indépendant de la vitesse à partir des librairies se fait de la manière suivante. Les parties opératives sont spécifiées au niveau schématique à partir des blocs logiques à précharge basés sur les logiques DCVS ou LDCVS. Les parties de contrôle sont spécifiées à partir de graphes de transitions de signaux (STG) [Chu94] [Meng89] [Hauc95]. Les schémas logiques correspondants basés sur des portes CMOS sont dérivés à la main à partir des graphes (cf 4.4.5) et entrés dans notre environnement de conception. Le circuit peut alors être simulé aux niveaux logique et électrique. Dès que le circuit est validé, il peut être routé avec des outils standard. Le layout obtenu est un mélange de cellules standard CMOS et asynchrones. Des simulations post-layout peuvent être effectuées pour vérifier les éventuelles hypothèses de timing et la vitesse. La Figure 16 présente le flot de conception complet.

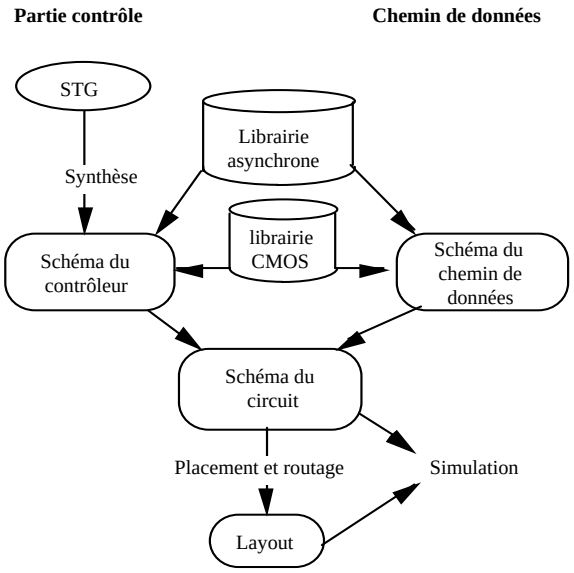

Figure 16: Flot de conception

4.4.2 Spécification et programmabilité des processeurs élémentaires

Le processeur élémentaire est configuré par un mot d'instruction (Figure 17), pour permettre le calcul de différents opérateurs morphologiques par le réseau. Un bit indique si on effectue une opération de minimum ou de maximum entre les variables voisines, ce qui correspond respectivement à une érosion et une dilatation. Un autre bit spécifie si on utilise l'opérateur géodésique, qui effectuera en fait l'opération duale avec le pixel de référence à la suite de l'opération normale (cf Equation 3). Un champ

de 5 bits spécifie les valeurs voisines à prendre en compte (le voisinage comprend au plus les 5 valeurs notées Nord, Est, Sud, Ouest et Local). Un autre champ de 5 bits donne le nombre d'itérations à effectuer localement. Le mot d'instruction est donc équivalent à un petit programme, qui contient des boucles et des exécutions conditionnelles, ce qui correspond à un modèle SPMD où il n'y pas d'attente entre les processeurs pendant toute l'exécution de l'instruction. L'ouverture par reconstruction se programme alors avec deux instructions, correspondant à l'érosion et à la dilatation géodésique itérée.

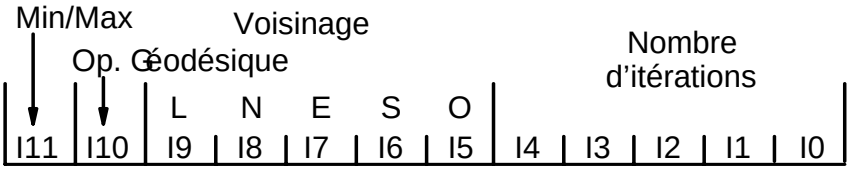

Figure 17: Format de l'instruction

4.4.3 Interface de communication inter-PE

Comme expliqué plus haut, l'asynchronisme fonctionnel ne requiert aucune synchronisation entre les calculs effectués par les différents PEs. Il faut donc le mettre en oeuvre au niveau du réseau de processeurs en utilisant un mécanisme spécifique d'échange entre PEs. Dans cette structure, chaque processeur élémentaire doit être capable de lire les valeurs les plus récemment mises à jour par les PEs voisins à n'importe quel moment, et ceci indépendamment de leur déphasage mutuel et sans signal de synchronisation préalable. Au niveau matériel, ceci correspond à autoriser un PE à échantillonner les variables de ses voisins à n'importe quel instant. Le schéma de communication adopté n'utilise pas les règles classiques de séquencement de type requête-acquittement, qui ne sont en fait pas strictement nécessaires. Un échantillonnage direct des valeurs données par les registres de sortie des processeurs voisins est donc mis en oeuvre de manière inconditionnelle.

Le problème majeur dans la conception matérielle d'une telle interface est l'absence d'une quelconque relation de phase entre l'activation du signal de lecture et la mise à jour de la variable qui doit être lue. Si un conflit de lecture/écriture survient, il se peut qu'il y ait une tentative d'échantillonnage d'un signal électrique transitoire. Alors il se peut que des états électriques non-logiques se propagent dans le circuit, ou que l'opérateur d'échantillonnage entre dans un état métastable [Klee87], ce qui la plupart du temps conduit à un comportement imprévisible. Pour éliminer ce problème, nous avons conçu une solution à bases de "Q-Flops" [Rose88]. Un circuit Q-Flop est une "latch" qui est capable d'échantillonner un signal d'entrée de manière asynchrone par rapport au signal de requête (Figure 18).

Après l'activation du signal de requête ("clock"), ce circuit produit de manière sûre une sortie à un niveau logique valide (c'est-à-dire de manière monotone et avec des fronts rapides, les sorties "output" et "output barre" ne pouvant de plus être à "un" simultanément), cependant en un temps théoriquement non

borné (s'il y a métastabilité). On peut générer ensuite un signal d'acquittement qui indique que l'échantillonnage est terminé et que la sortie est à un niveau valide, à l'aide d'un "et" logique (comme en DCVSL). Les transistors pilotés par les entrées doivent être "faibles" devant la boucle constituée par les deux inverseurs, de telle sorte qu'un changement des entrées n'a aucun effet sur le circuit à partir du moment où cette boucle a commencé à "basculer" (le transistor piloté par le signal de requête l'ayant préalablement positionnée dans un état d'équilibre instable).

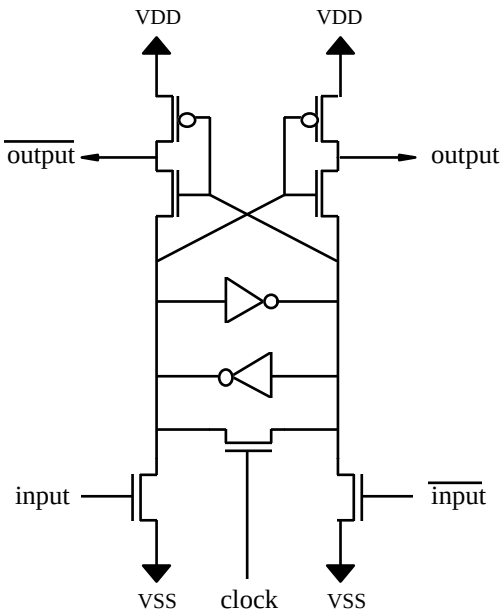

Figure 18: Schéma transistor d'une "Q-Flop"

Considérons le cas le plus simple où deux processeurs adjacents doivent échanger un unique bit de donnée. L'émetteur écrit alors simplement le nouveau bit de donnée dans une latch ou un registre standard, alors que le récepteur recopie la sortie de ce registre à n'importe quel instant, dans un registre Q-Flop. Comme la Figure 19 le montre, aucun signal de contrôle supplémentaire n'est nécessaire entre les PEs. Un protocole standard à quatre phases est utilisé entre les parties de contrôle et leurs registres d'entrée/sortie associés. Côté réception, le processeur peut commencer un calcul avec la nouvelle valeur dès que le signal d'acquittement de la Q-Flop devient actif. Ce mécanisme très simple est suffisant pour implémenter l'asynchronisme fonctionnel décrit plus haut, lorsqu'un seul bit ou un ensemble de bits indépendants doivent être échangés entre PEs.

Dans le cas présent, nous devons considérer le problème général où les processeurs s'échangent des données parallèles multi-bits. Ceci implique qu'une forme de synchronisation doit être imposée entre les bits. En fait, à cause de délais assymétriques dans les portes et les connexions, et des temps de réponse variables des Q-Flops, des bits appartenant à deux mots consécutifs pourraient être mélangés lors de l'échantillonnage côté récepteur. De manière à éviter ce type d'aléas, et toujours en

s'affranchissant d'un protocole de requête-acquittement entre les PEs, deux bits de phase peuvent être associés aux données (Figure 20).

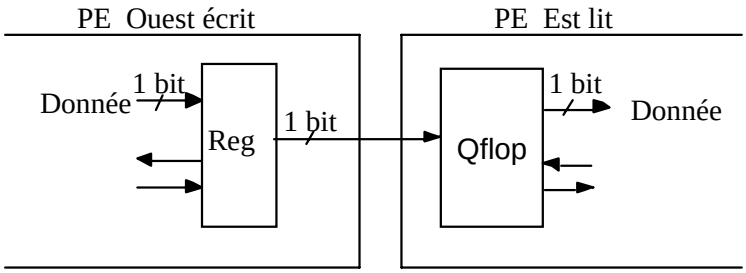

Figure 19: Interface mono-bit

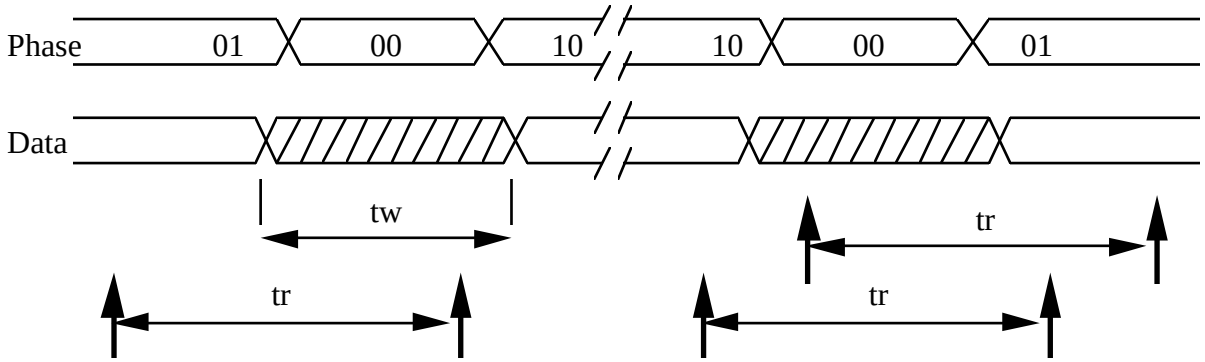

Figure 20: Deux séquence d'écriture successives. Les flèches verticales représentent les dates d'échantillonnage des bits de phase pendant une séquence de lecture. Les trois cas présentés correspondent aux cas où une autre séquence de lecture est nécessaire. Les autres situations ne nécessitent pas d'autre séquence de lecture à cause des hypothèses temporelles.

Le protocole de communication peut être décrit en considérant les actions de communication effectuées par l'émetteur et le récepteur. Considérons d'abord le côté émetteur. Supposant que les bits de phase ont une valeur initiale "01", l'émetteur commence par écrire "00" dans ces bits de phase, signifiant ainsi que la variable va être mise à jour. Il écrit alors la nouvelle valeur puis écrit enfin "10" dans les bits de phase, signalant que l'opération d'écriture est terminée. Ces opérations définissent une séquence d'écriture (Figure 20). Le fait d'utiliser deux bits de phase au lieu d'un seul est nécessaire puisque cela permet au récepteur non seulement de détecter que la variable va être modifiée mais aussi de s'apercevoir si elle a changé pendant la lecture. En fait, les bits de phase sont différents avant et après l'opération d'écriture. Le récepteur procède de la manière suivante. Il lit les bits de phase, puis lit les données, et finalement relit les bits de phase. Ces opérations définissent une séquence de lecture. Si les bits de phase ont changé au cours de l'opération, cela signifie que la valeur lue est peut-être fausse car elle a peut-être

changé pendant la lecture. Dans ce cas, le récepteur effectue une autre séquence de lecture. La Figure 21 présente un synoptique de l'interface.

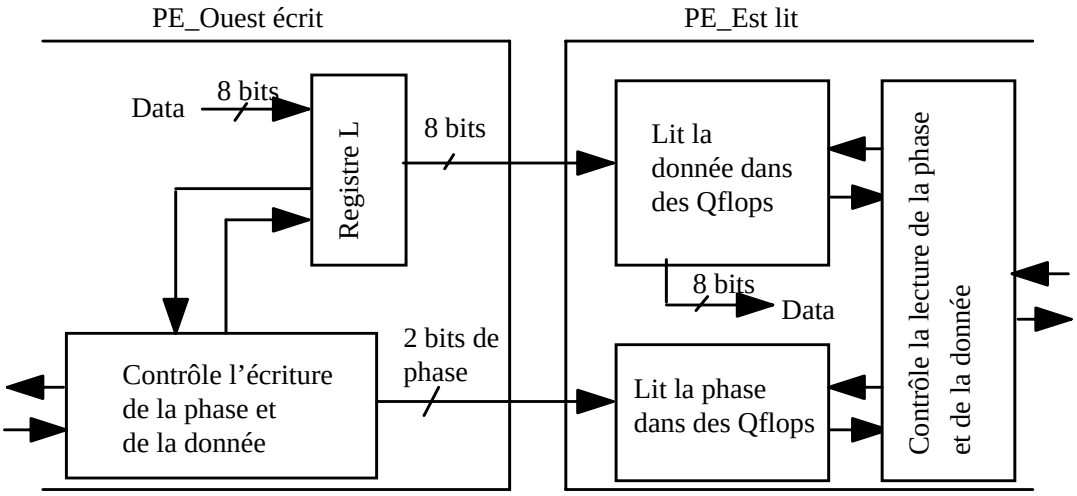

Figure 21: Synoptique de l'interface multi-bits

Cependant cette implémentation n'est pas insensible aux délais et nous devons donc spécifier les hypothèses temporelles qui doivent être respectées pour assurer un fonctionnement correct. D'abord, la durée d'une séquence d'écriture (notée "tw" sur la Figure 20), qui inclut la durée d'écriture des bits de phase et des données, de même que les délais dus aux connexions, doit être plus courte que la durée d'une séquence de lecture (notée "tr"). Cette hypothèse temporelle garantit que le récepteur ne peut pas échantillonner une variable deux fois de suite pendant une même séquence d'écriture. Ceci est facilement garanti par l'implémentation matérielle de l'interface. Deuxièmement, l'intervalle de temps qui sépare deux mises à jour d'une même variable doit être plus long que le temps nécessaire à la lecture et à la re-lecture de cette variable. Cette seconde hypothèse temporelle garantit que le récepteur ne peut pas observer deux valeurs identiques de la phase alors que la variable a été mise à jour deux fois de suite. C'est en fait toujours le cas car le temps de calcul qui sépare deux écritures est bien plus grand que le temps passé à effectuer deux séquences de lecture. En dépit de ces deux hypothèses temporelles, l'implémentation du contrôleur d'interface s'est montrée très robuste. Nous avons finalement un mécanisme qui implémente l'asynchronisme fonctionnel en évitant un arbitrage trop important, au coût de deux phases de lecture et d'une séquence de lecture supplémentaire lorsqu'une tentative de lecture se produit pendant une écriture.

4.4.4 Chemin de données

Le processeur élémentaire doit pouvoir stocker trois pixels sur 8 bits, un pour la valeur d'itération courante qui est visible à partir des voisins, un autre pour une valeur temporaire utilisée à l'intérieur

d'une itération, et un troisième pour le pixel de référence. Le chemin de données comprend les registres correspondants (respectivement L, B, et I), de même que des registres série/parallèle pour les données d'entrée/sortie qui se propagent indépendamment des calculs du réseau (E et R pour les pixels intermédiaires d'entrée et les pixels de référence respectivement, et S pour la sortie). Un multiplexeur sélectionne une valeur parmi le pixel de référence et les quatre valeurs des voisins. La partie opérative doit pouvoir calculer un minimum ou un maximum entre la valeur temporaire stockée dans B et la valeur voisine sélectionnée. Puisque la valeur temporaire B doit être initialisée avec la valeur du pixel d'entrée stocké en E, un multiplexeur est utilisé pour sélectionner la sortie du registre E ou la sortie de la partie opérative. En fait, la Figure 22 montre que B n'est pas stocké dans un registre mais dans le multiplexeur lui-même, qui est conçu en logique DCVS avec latch [Rena96]. Comme l'ALU est aussi implémentée en LDCVSL, l'ensemble constitue un anneau auto-séquencé avec seulement deux étages [ElHa95b].

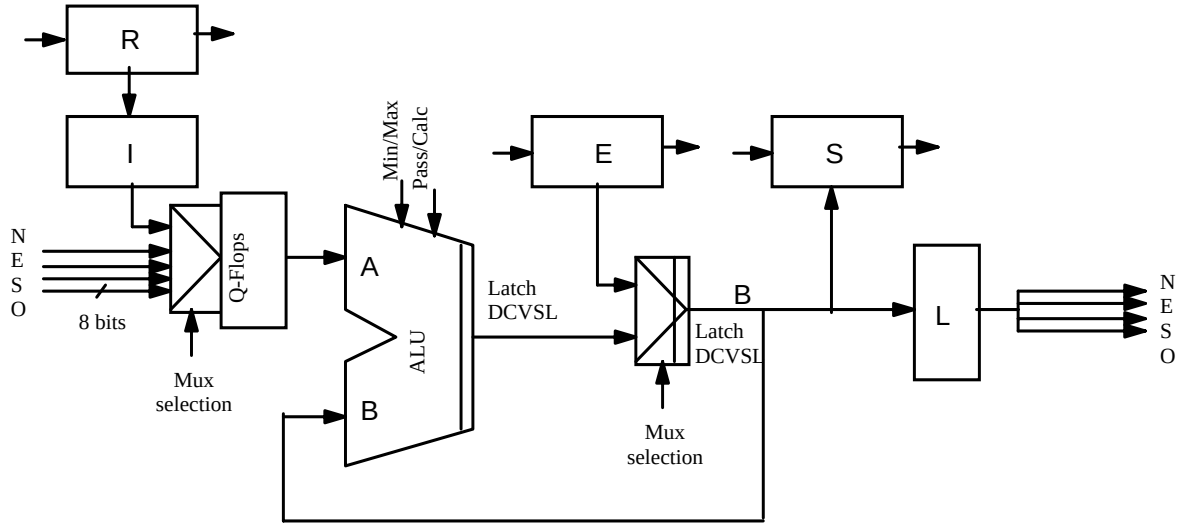

Figure 22: Chemin de données

L'ALU est contrôlée par deux signaux qui spécifient si la sortie doit être égale au minimum ou au maximum des entrées A et B, ou encore égale à l'entrée A. Des signaux de requête et d'acquittement (qui ne figurent pas sur la Figure 22) sont connectés à l'ALU, le multiplexeur B et les Q-Flops. Les registres reçoivent aussi des signaux locaux d'activation.

L'ALU est implémentée avec un additionneur et des multiplexeurs [VanD95]. Un additionneur asynchrone à propagation/génération de retenue et détection de fin de calcul est utilisé, de telle sorte que l'ALU puisse tirer partie de l'existence de chemins critiques dynamiques et puisse donner lieu à des dispersions de temps de calcul significatives, conduisant à un temps de calcul moyen en $O(\log_2 n)$ pour une addition sur n bits [Rena94a] [ElHa95a]. Dans un additionneur élémentaire (ou "full adder"), on peut en effet générer la retenue sortante, indépendamment de la retenue entrante dans les cas où les deux

entrées sont égales (0+0+cin donne une retenue à 0 et 1+1+cin donne une retenue à 1). Ceci permet de couper dynamiquement la chaîne de propagation de la retenue, en fonction des valeurs des opérandes, et donc d'obtenir un temps de calcul minimum, contrairement aux structures synchrones qui doivent considérer le pire cas (ou "chemin critique").

4.4.5 Contrôle asynchrone

Le schéma de contrôle consiste principalement à attendre une requête externe, copier le dernier résultat B dans le registre parallèle/série de sortie S, copier les valeurs d'entrée des registres série/parallèle E et R dans les registres de second niveau B et I, et démarrer le calcul itératif. Ce calcul contient deux boucles imbriquées: la boucle externe compte le nombre d'itérations, la boucle interne balaye les différents PEs appartenant au voisinage spécifié. Une fois que le résultat de l'itération précédente a été copié dans le registre d'itération courante L, la boucle interne sélectionne séquentiellement les valeurs voisines, et les combine à la valeur temporaire B.

Une approche modulaire a été utilisée pour implémenter le contrôle asynchrone. Il est relativement simple de spécifier un ensemble de modules de contrôle de base, qui peuvent être assemblés pour concevoir le contrôle global du chemin de données. Une fois que cet ensemble de modules correspond effectivement à la séquence de contrôle principale, des automates asynchrones plus complexes peuvent être ajoutés comme des modules "feuilles" de l'arbre de contrôle, par exemple les interfaces de lecture et d'écriture qui échangent des signaux de contrôle entre les PEs.

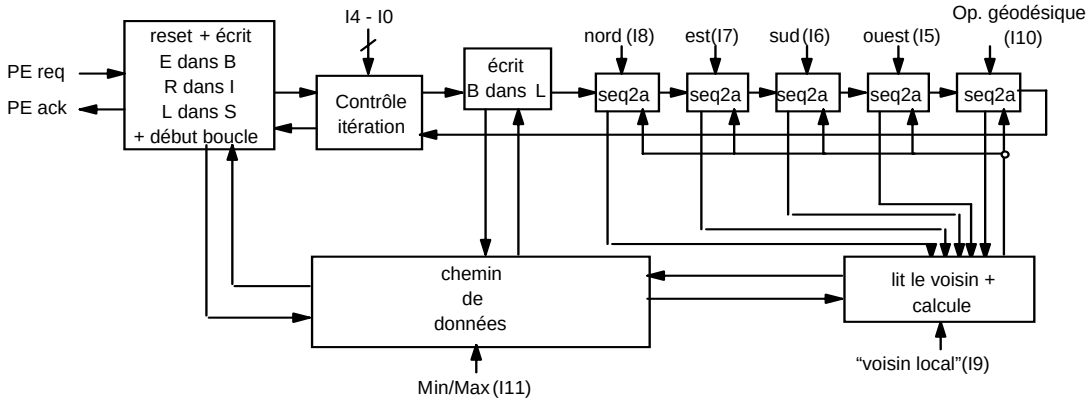

Figure 23: Vue simplifiée de l'architecture du contrôle

La Figure 23 montre l'architecture du contrôle. Toutes les flèches correspondent à des signaux de requête/acquittement qui connectent entre eux les sous-modules de contrôle et le chemin de données. Le module de contrôle d'itération génère autant de cycles requête-acquittement à quatre phases que le spécifie le champ <I4-I0> de l'instruction. Chaque module "seq2a" est un automate asynchrone simple qui génère de manière conditionnelle un cycle "req/ack" vers le contrôle du chemin de données, en

fonction de la valeur du bit de l'instruction qui lui est attaché. Si ce bit (I8 par exemple) est inactif, alors le voisin correspondant (Nord dans ce cas) n'est pas pris en compte (car l'instruction spécifie qu'il n'appartient pas à l'élément structurant) et le signal de requête est immédiatement propagé au module "seq2a" suivant. Lorsque toutes les valeurs voisines ont été prises en compte et que l'opération géodésique duale a été éventuellement effectuée (en fonction du bit I10 de l'instruction), le module de contrôle d'itération peut commencer la prochaine itération. Finalement un signal d'acquittement est activé par l'entité correspondant au PE complet.

Chaque module de contrôle a été spécifié sous la forme d'un automate asynchrone à l'aide d'un graphe de transitions de signaux à choix possible sur les sorties (STG/NC pour "Signal Transition Graph with Non-input Choice" [Hauc95]). Un STG peut être vu comme une représentation duale d'un réseau de Pétri, où les transitions sont notées explicitement avec les noms des signaux (par exemple "s+" ou "x-") et où les "places" sont remplacées par des arcs entre les transitions. Les choix sont seulement permis lorsqu'une variable statique est impliquée, par exemple les informations issues du mot d'instruction. L'étape d'assignation des états (qui passe du STG au graphe d'état SG) utilise directement les valeurs des signaux d'entrée/sortie du module pour encoder les différents états de l'automate. Certains signaux internes doivent parfois être ajoutés pour garantir que les états ont des codes différents (propriété d'assignation unique des états). La valeur logique de chaque signal de sortie s'écrit alors comme une fonction Booléenne des signaux d'entrée et de sortie. L'élaboration des équations Booléennes à partir des STG/NC a été effectuée à la main, en s'assurant que les aléas logiques éventuels ont été éliminés. Notre procédure de synthèse s'est inspirée des techniques présentées dans [Chu94] et [Meng89]. Les fonctions combinatoires sont directement "mappées" sur un ensemble de portes logiques CMOS. Les fonctions séquentielles sont transformées en signaux "set" et "reset" qui contrôlent des bascules RS, composées de deux portes "NOR". Pour illustrer cette procédure, les figures suivantes montrent les étapes correspondantes pour le module "seq2a" mentionné plus haut (Figure 24a), en particulier la description STG/NC (Figure 24b), les équations Booléennes (Figure 24c) et l'implémentation niveau portes (Figure 24d). Le signal interne "BID" a été introduit pour résoudre un problème d'assignation d'état.

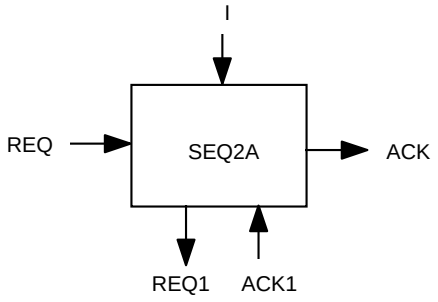

Figure 24: Exemple de conception d'un bloc de contrôle. a) Vue externe du module "seq2a"

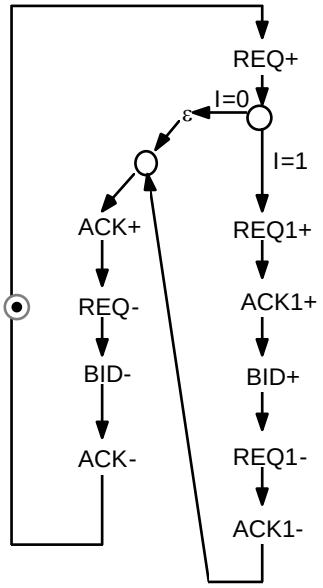

Figure 24. b) Spécification STG

$$REQ1 = \overline{\overline{REQ} + \bar{I} + BID}$$

$$SET\_ACK = \overline{\overline{REQ}.\bar{I}.\overline{BID}.\overline{ACK1}}$$

$$RESET\_ACK = \overline{BID + REQ}$$

$$SET\_BID = ACK1$$

$$RESET\_BID = \overline{REQ}$$

Figure 24. c) Equations Booléennes

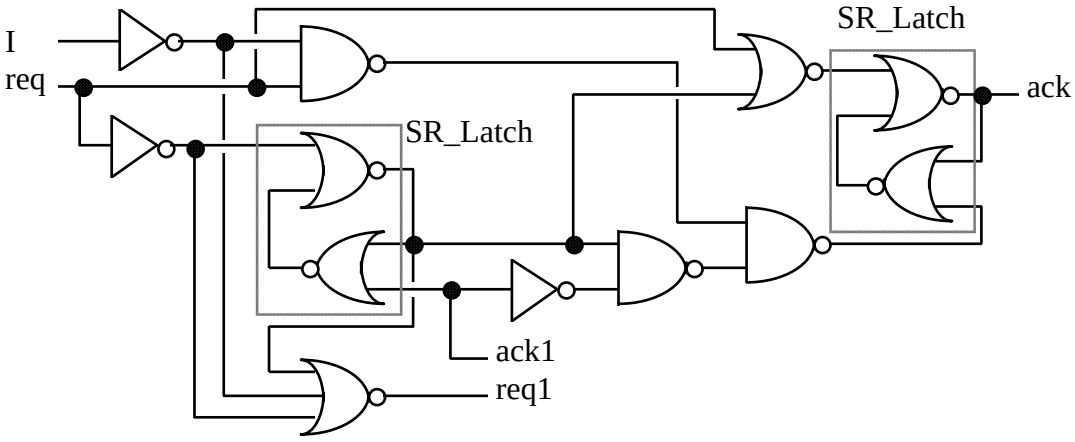

Figure 24. d) Schéma de l'implémentation CMOS

L'ensemble des modules élémentaires de contrôle, ainsi que tous les automates asynchrones complexes du PE, comme le contrôle d'itération ou les interfaces de communication, sont décrits dans

[VanD95]. L'implémentation des composantes du PE est pour la plupart d'entre elles de type "indépendante de la vitesse", mais quelques hypothèses temporelles de même niveau, largement vérifiées à la simulation et basées sur les nombres relatifs de portes traversées entre certains points du circuit, ont parfois été utilisées.

4.5 Circuit test basé sur un réseau de 16×16 processeurs-pixel

4.5.1 Conception "back-end" du circuit

Le processeur élémentaire a été routé de manière à obtenir un bloc de niveau layout, dans lequel les positions relatives des ports d'entrée/sortie ont été fixées. Ceci permet ensuite de générer le layout global du réseau de processeurs, par simple aboutement de 14×14 blocs de layout élémentaires. Les cellules de bord sont routées séparément puis ajoutées au coeur. Les plots du circuit sont finalement connectés aux plots de bord du layout.

Le processeur élémentaire contient environ 3500 transistors pour une surface de 500×500 μm^2. La complexité du réseau de processeurs atteint 800.000 transistors et une surface globale de 8×9 mm^2, dans une technologie CMOS 0.5 µm à trois niveaux de métal (Figure 25) [Robi96b].

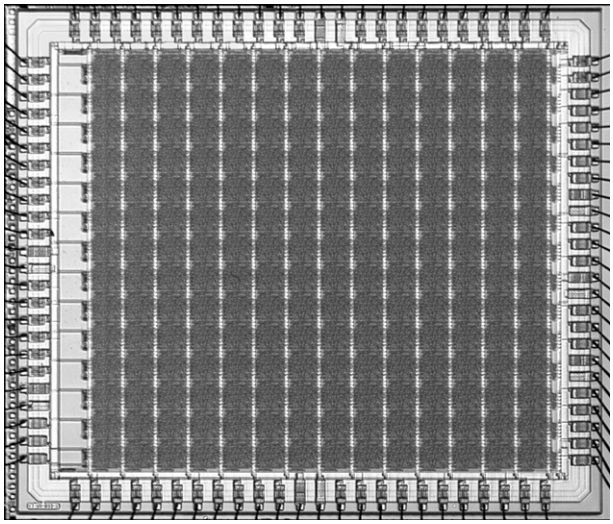

Figure 25: Photomicrographie du circuit

4.5.2 Intégration système

Ce réseau de processeurs VLSI sera intégré dans un prototype de traitement temps réel. Puisque le circuit ne peut seulement traiter qu'un morceau de l'image complète, un schéma de traitement partitionné localement parallèle globalement séquentiel doit être utilisé. Une image doit ainsi être parcourue bloc par bloc. De plus, pour propager les résultats entre blocs, l'image doit être parcourue avec un certain recouvrement, d'au moins un pixel entre blocs adjacents. Enfin, l'image doit être parcourue plusieurs fois

jusqu'à ce que la convergence du traitement soit atteinte. Pour démontrer les capacités du circuit, des séquences d'images seront acquises en temps réel, traitées et visualisées. Deux interfaces spécifiques seront conçues pour interfacer le circuit avec un système standard d'acquisition d'images, comme le montre la Figure 26.

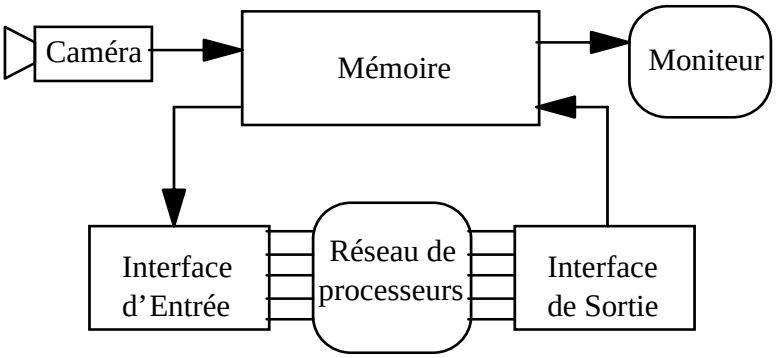

Figure 26: Intégration système.

4.5.3 Résultats de test

Le circuit a été fabriqué au centre commun CNET / SGS-THOMSON, et le test sur HP82000 a montré qu'il était totalement fonctionnel. Les fichiers utilisés par le testeur ont été générés à partir d'un modèle VHDL structurel du circuit. Nous donnons ci-dessous un certain nombre de mesures concernant la vitesse et la consommation du circuit.

A 3.3 V, l'exécution par un seul PE d'une instruction correspondant à une seule étape d'itération, avec le voisinage complet et l'opération géodésique duale, prend entre 200 ns et 260 ns, en fonction des données. Ce calcul correspond à cinq additions sur huit bits, cinq accès aux registres voisins et une opération d'écriture pour stocker le résultat. Le temps de calcul total correspondant à 31 itérations atteint 9 µs, ce qui inclue tous les temps dus au contrôle (liés aux itérations multiples, aux communications inter-PE, et au protocole de communication externe).

Selon ces chiffres, des images de 256×256 pixels peuvent donc être traitées à une fréquence maximale théorique de 43 Hz avec un seul circuit. En effet, avec un recouvrement de un pixel, 17×17 blocs doivent être traités. D'après des simulations algorithmiques, la convergence globale est atteinte après huit parcours de l'image. En surestimant le temps de calcul d'un bloc à 10 µs, le temps total de traitement d'une image est égal à 23 ms.

Le circuit est fonctionnel dans une plage de tension d'alimentation variant de 1.75 V à 4.75 V. La Figure 27 et la Figure 28 donnent respectivement le temps de calcul d'un bloc et la consommation en fonction de la tension d'alimentation. A 3.3 V, la consommation du circuit est d'environ 900 mW. A

1.8 V, elle est seulement de 100 mW, c'est-à-dire que la consommation est réduite d'un facteur 9, alors que le temps de calcul est seulement multiplié par 2 (20 ms).

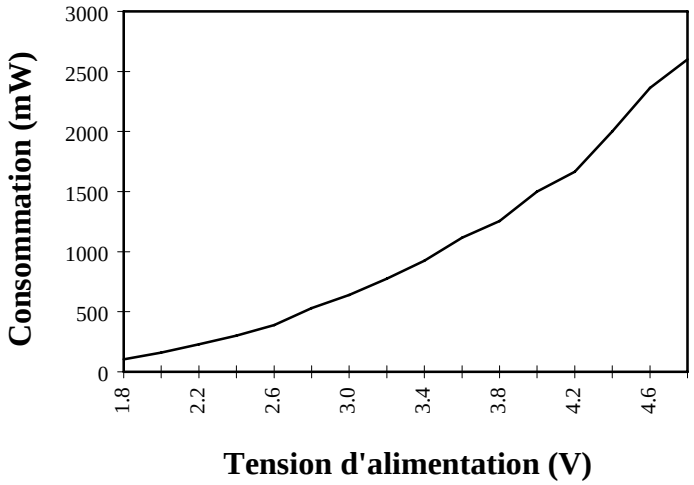

Figure 27: Consommation en fonction de la tension d'alimentation

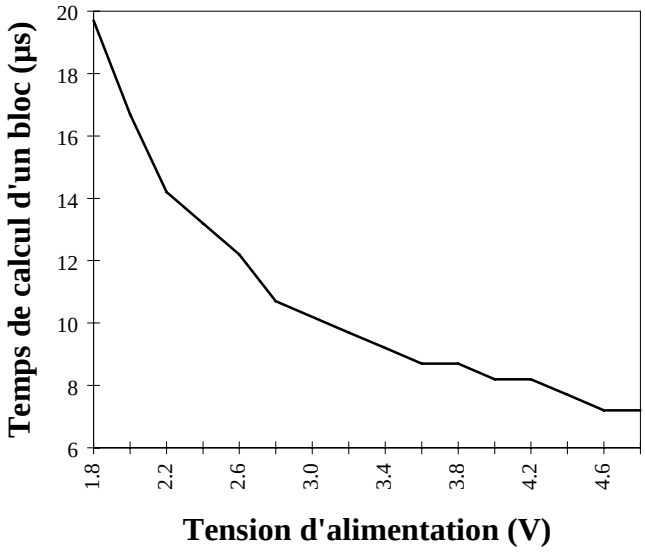

Figure 28: Temps de calcul d'un bloc en fonction de la tension d'alimentation

4.6 Conclusion

Partant des concepts d'architecture cellulaire à grain fin et d'asynchronisme, dont la combinaison constituait l'idée de départ d'un projet d'étude conjointe algorithme-architecture, nous avons réalisé l'étude et la conception complète d'un prototype de coprocesseur VLSI spécifique pour une classe de

traitements d'images de bas niveau. D'un point de vue algorithmique, cette étude constitue une extension des travaux présentés dans [Plan84], par l'application d'un asynchronisme fonctionnel au problème du filtrage morphologique, qui est à la base d'algorithmes de segmentation pour le codage d'images basé-régions. Nous en avons présenté ensuite une implémentation VLSI originale basée sur un asynchronisme structurel, ce qui illustre la façon dont le concept d'asynchronisme peut être exploité à différents niveaux, algorithmique pour améliorer la vitesse de convergence, et structurel pour accroître la modularité, la scalabilité et la robustesse, et optimiser les performances d'une architecture massivement parallèle intégrée. Le circuit AMPHIN a constitué le premier (et le plus gros!) circuit asynchrone à contrôle complexe réalisé dans le groupe de Marc Renaudin, et aussi la première implémentation VLSI d'un algorithme de relaxation asynchrone.

Les solutions et le style de circuits asynchrones présentés plus haut correspondent à notre première expérience de mise en oeuvre de telles spécifications. Nous avons acquis depuis un nouveau point de vue, correspondant à la mise en place d'un cadre et d'un style de conception de circuits radicalement différents (qui sont introduits à la fin du chapitre suivant). Cette nouvelle optique nous amène bien sûr à regarder de manière assez critique cette première réalisation. Bien que les structures de base du circuit AMPHIN n'avaient pas pour objectif d'être optimales, le prototype devant démontrer la faisabilité de mise en oeuvre des principes algorithmiques et architecturaux, nous pensons que de nouvelles solutions pourraient largement améliorer les performances, la robustesse et la facilité d'utilisation du circuit, en particulier en ce qui concerne les interfaces de communication inter-PE, les entrées/sorties globales, et le contrôle interne.

De manière générale, nous avons néanmoins démontré sur une application de traitement d'images que les circuits asynchrones élargissent le spectre de solutions possibles pour la conception conjointe d'algorithmes et d'architectures. A partir de notre application, nous avons ainsi montré que l'approche asynchrone autorise l'implémentation d'une classe plus étendue d'algorithmes parallèles, dont les contraintes de synchronisation sont relâchées. Nous pensons que cette étude ouvre un nouveau champ d'application pour les architectures VLSI asynchrones, qui devrait encourager le développement d'algorithmes basés sur des ensembles de processus concurrents communicants faiblement synchronisés.

Chapitre 5

PROPOSITION D'UN MODELE DE RESEAU VLSI CELLULAIRE ASYNCHRONE PROGRAMMABLE

5.1 Vers une architecture de coprocesseur pour l'analyse / rendu d'images

Le processeur AMPHIN nous a permis de valider l'intégration des concepts architecturaux de parallélisme massif et d'asynchronismes structurel et fonctionnel à grain fin. On peut le classer dans la catégorie des "réseaux computationnels" spécifiques ("computational arrays" [Seit84]). Tout comme les nombreuses architectures VLSI spécifiques qui ont été conçues pour répondre à des contraintes de performance, par exemple des architectures parallèles pour l'estimation de mouvement, il trouverait son application dans une chaîne particulière d'analyse / traitement d'images comme coprocesseur, dans le sens où un processeur à usage général ne permettrait pas, à génération technologique équivalente, de respecter les mêmes contraintes de performance. D'un point de vue économique, il faut aussi prendre en compte le coût d'une telle solution spécifique par rapport à celui de l'utilisation de plusieurs processeurs "sur étagère" ("off-the-shelf") qui atteindraient une performance équivalente. L'évaluation de ce coût dépend fortement du créneau d'utilisation du coprocesseur et est lié aux problèmes d'économies d'échelle [Tred95] [Tred96]. En fait, dans les applications télécom, ce sont plutôt les contraintes d'intégration et de consommation qui rendront viable l'utilisation d'un coprocesseur spécifique, jusqu'au jour où la solution programmable respecte l'ensemble des contraintes, grâce à l'évolution de la technologie. Le positionnement et la durée de vie d'une architecture VLSI sont des critères aussi importants que l'innovation structurelle ou fonctionnelle qu'elle présente...

Nous souhaitons dans ce chapitre prendre en compte l'évolution du codage d'images, présentée au chapitre 1, vers la communication "multimédia", dans le sens de la généricité, des fonctionnalités et des

représentations de "haut" niveau liées à l'image. Nous pensons que l'évolution de la technologie VLSI doit remettre en cause les choix architecturaux dans le domaine de la communication visuelle. Alors que certains considèrent que la mise en oeuvre de MPEG2 de manière "logicielle" est le signe de l'avènement des microprocesseurs à usage général, nous estimons que le nouveau paradigme introduit par MPEG4 doit s'appuyer sur des architectures radicalement différentes de l'approche "incrémentale" classique, qui consiste à ajouter des unités de traitement à une architecture à flot de contrôle et à mémoire centralisés. Bien qu'apparaissent certaines évolutions sous le nom très commercial de "media processors", nous pensons qu'elles sont trop spécifiques et/ou qu'elles présentent des limitations équivalentes aux architectures des microprocesseurs. Alors que certains se demandent ce que l'on pourra bien faire avec cent millions de transistors dans seulement quelques générations technologiques, sinon augmenter la taille des chemins de données et de la mémoire intégrée, il nous semble que les architectures VLSI parallèles et "concurrentes" [Seit84] peuvent devenir une réalité à grande échelle, en particulier dans les applications de communication visuelle. La mise en oeuvre de la convergence "analyse / codage / rendu" de l'image doit se baser sur le respect de contraintes de performance et de généricité. En termes d'architecture, les propriétés essentielles à prendre en compte sont le parallélisme et la programmabilité. Performance et parallélisme ont souvent impliqué la notion d'architecture spécifique, au niveau VLSI. Cependant, les notions d'architecture spécifique et d'architecture programmable ne sont pas nécessairement incompatibles. Tout dépend de la granularité considérée. L'évolution de la technologie permet d'envisager de nouveaux degrés de granularité et c'est pourquoi la notion de coprocesseur peut changer de niveau, en passant de la mise en oeuvre d'une fonction calculatoire dédiée à celle d'un domaine complet de fonctionnalités liées à l'analyse / rendu d'images, pour les applications qui nous concernent. C'est la notion de "coprocesseur d'API" qui remplace celle de coprocesseur spécialisé (Figure 29, cf [Tred96]).

Un coprocesseur à la fois programmable et spécifique à l'image repose sur une granularité intermédiaire entre le réseau computationnel et les systèmes multiprocesseurs. C'est en fait une alternative aux architectures pyramidales pour la vision, qui font intervenir un empilement de différents niveaux de granularité (le grain fin pour les traitements de bas niveau et le gros grain pour ceux de haut niveau). Nous proposons de combiner le parallélisme à grain fin (niveau pixel) d'une architecture cellulaire, dont les caractéristiques sont la localité, l'homogénéité et la régularité (spatiale), à la flexibilité (temporelle) de séquencement des architectures à flots de contrôle et de données concurrents, que l'on base sur un principe généralisé de synchronisation minimale (ou "asynchronisme(s)"). Nous présentons dans ce chapitre un modèle d'architecture cellulaire asynchrone programmable, ainsi qu'un certain nombre d'évaluations d'algorithmes, qui généralisent la combinaison du parallélisme et de l'asynchronisme.

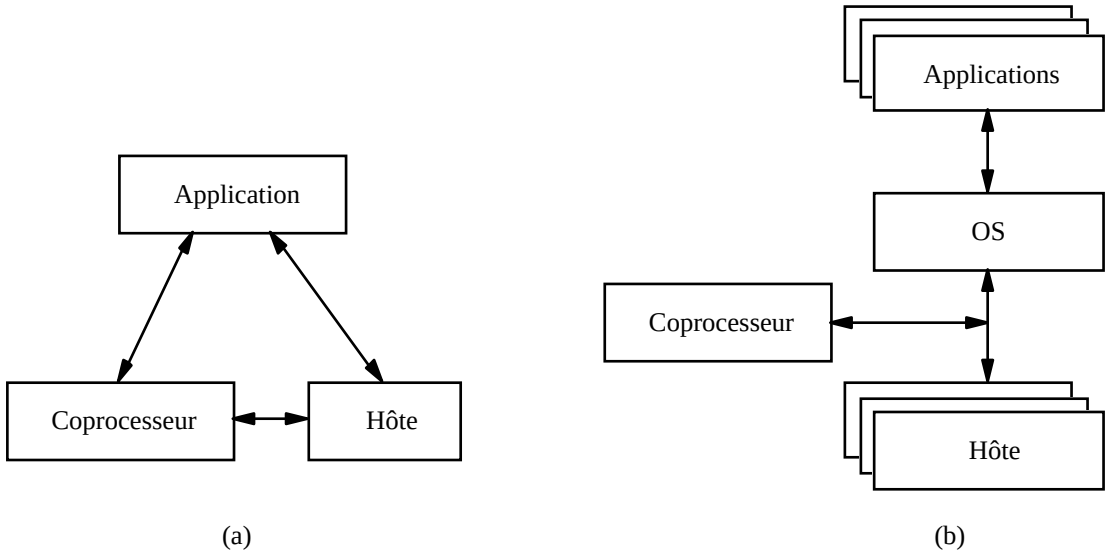

Figure 29: Evolution du coprocesseur au sein des systèmes:
(a) coprocesseur spécialisé, (b) coprocesseur d'API

5.2 Présentation du modèle architectural

5.2.1 Un réseau VLSI cellulaire toroïdal asynchrone à contrôle SPMD

L'architecture globale s'appuie sur une structure classique en grille à maille carrée (Figure 30a), qui correspond directement à la structure en pixels des images. Cela correspond initialement à considérer une architecture à grain fin, où l'on associe un processeur élémentaire à chaque pixel. Comme nous considérons une granularité intermédiaire, on associe un processeur à un ensemble de pixels, après partitionnement de l'assignation ("mapping") initiale. Nous discuterons du choix du partitionnement au paragraphe 5.3.3, mais pouvons introduire dans tous les cas la structure quasiment aussi classique de grille toroïdale, par un simple argument de symétrie. La grille étant de taille finie (ce qui n'est pas toujours le cas dans les modèles computationnels d'automates cellulaires), les processeurs élémentaires situés au bord du réseau présentent une singularité due au fait que leur voisinage est structurellement incomplet. Un "rebouclage" du réseau permet de rétablir la régularité topologique (Figure 30b). Cependant l'intégration directe de ce schéma au niveau "layout" ne respecte pas le principe de localité, puisque la longueur des liens de rebouclage augmente avec la taille du réseau. Pour rétablir la localité, c'est-à-dire obtenir des liens de taille minimale et indépendante de la taille du réseau, il suffit d'effectuer un double "repliement" (Figure 30c), qui ne modifie pas la topologie du réseau. Ce schéma est aussi très classique [Seit84], mais introduira quatre types de processeurs élémentaires ne différant que par les positions de leurs ports d'entrée / sortie, si l'on veut réaliser le réseau au niveau "layout" par simple aboutement des processeurs.

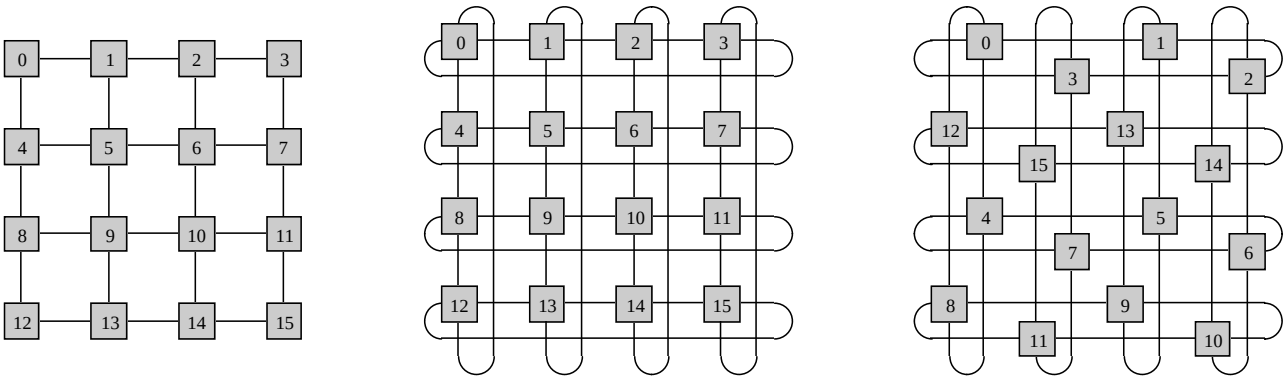

Figure 30: Structures de réseau (a) en grille, (b) en grille toroïdale, (c) en grille toroïdale "repliée"

Le principe fondamental de l'architecture proposée est basé sur le fait que la régularité spatiale doit être "contrebalancée" par la flexibilité temporelle. Le point de départ est l'utilisation d'un mode de contrôle de type SPMD, intermédiaire entre les modes SIMD et MIMD, où chaque processeur peut exécuter son propre flot d'instructions de manière indépendante des autres processeurs, ce flot étant identique et dupliqué dans tout le réseau. Cette architecture fait partie de la catégorie des systèmes concurrents faiblement couplés ("loosely coupled concurrent systems" [Seit84]), où le "couplage", c'est-à-dire les synchronisations, est explicitement spécifié au niveau programme par des actions de communications locales. Etant donné que nous ne faisons aucune hypothèse de synchronisation globale du réseau, le principe de synchronisation minimale peut être généralisé au niveau structurel le plus fin. Nous nous intéressons donc à une implémentation totalement asynchrone, c'est-à-dire où chaque processeur élémentaire travaille à sa propre vitesse maximale, indépendamment des autres processeurs ou de tout signal global de cadencement, comme le signal d'horloge utilisé dans la quasi-totalité des circuits numériques aujourd'hui. Nous baserons la conception VLSI sur des techniques de conception asynchrone, donc sans signal d'horloge, et en particulier sur une méthode de conception de circuits quasi-insensibles aux délais (QDI) à partir d'une spécification sous forme de processus concurrents communiquants.

Regardons maintenant de plus près la brique de base du réseau. Chaque processeur élémentaire est un coeur de microprocesseur asynchrone de type RISC, muni d'une mémoire de programme et d'une mémoire de données accessibles par lui seul. Les communications et synchronisations entre processeurs élémentaires et avec l'extérieur du réseau se font uniquement par l'intermédiaire d'interfaces spécifiques.

5.2.2 Coeur RISC asynchrone d'un processeur élémentaire

En contraste avec la spécificité et la fine granularité généralement associées au réseau en grille, l'architecture est complètement programmable, grâce à l'utilisation d'un coeur de microprocesseur à usage général dans chaque processeur élémentaire (PE). Contrairement à une approche

"multiprocesseurs", basée sur l'intégration de plusieurs processeurs à usage général, initialement conçus pour maximiser la performance de traitement associée à un flot de contrôle unique, la prise en compte d'une granularité intermédiaire consiste à concevoir le PE autour d'une architecture minimale de microprocesseur. Le processeur élémentaire est basé pour cela sur les architectures RISC [Henn92] [Henn94] [Heud90] [Stal88], qui ont démontré que la simplicité permettait de surcroît de hautes performances, grâce à l'utilisation d'un jeu d'instructions simple, de l'exploitation de bancs de registres étendus et d'une mise en oeuvre optimale du "pipeline", ce qui correspond en fait à des principes de localité et de régularité. De plus, nous pouvons minimiser la complexité du processeur de manière spécifique aux problèmes à traiter. Les composantes des pixels d'une image sont couramment codés sur une dynamique de 8 bits, soit 256 niveaux. Ces "octets" correspondent aussi à la taille des données manipulées par les premières générations de microprocesseurs et par beaucoup de microcontrôleurs. Beaucoup de traitements d'image de bas et moyen niveau ne nécessitent pas une précision plus importante, et de nombreuses classes d'algorithmes avancés, de morphologie mathématique par exemple, sont basées ou peuvent être spécifiées uniquement avec des nombres entiers de faible précision. En continuité avec les principes des architectures RISC, nous écarterons ici des solutions à granularité plus élevées basées sur des processeurs superscalaires ou VLIW (au profit d'architectures "superpipelinées", cf 5.3.4), le parallélisme essentiellement visé étant un parallélisme de données au niveau réseau, ayant la propriété de scalabilité quasi-parfaite.

D'autres paramètres architecturaux, comme les modes d'adressage, la taille du banc de registres ou les autres dimensionnements, sont entre autres influencés par la granularité visée, la facilité de mise en oeuvre, la complexité des primitives de traitements visées ou encore par le codage du jeu d'instructions. Certains des choix architecturaux ou de dimensionnement effectués dans ce chapitre ne doivent pas être considérés de manière absolue, mais comme donnant des ordres de grandeur cohérents pour la réalisation d'évaluations. Ainsi, nous considérerons dans la suite, à titre d'exemple, un coeur RISC scalaire de type "registre-registre", à 16 registres, travaillant sur des entiers non signés de 8 bits.

Nous présentons dans le Tableau 2 la liste des instructions ou pseudo-instructions non spécifiques à l'utilisation des mémoires ou des interfaces de communication. Les opérandes sont soit des valeurs "immédiates", précédées du symbole *#*, soit des valeurs de registres. Le registre numéro *i* ainsi que sa valeur sont notés r_i. Les fonctions associées aux instructions sont décrites avec des notations issues du langage C (rappelons que la notation "*x = cond ? a : b;*" est équivalente à "*if (cond) x = a; else x = b;*"). L'utilisation du modulo *%256* est équivalente à celle du "et logique" *&255*, qui rappelle en particulier que les registres "destination" ne possèdent que 8 bits. Les valeurs immédiates notées *valeur* ont 8 bits, celles notées *décalage* ont 3 bits et celles notées *saut* sont codées sur 8 bits signés. Les instructions marquées du symbole (*) peuvent n'être que des pseudo-instructions, en particulier pour *subi* qui est

équivalente à *addi*. Les instructions *min* et *max* pourraient être remplacées par *sltu*, *brz* et *move*, mais il est préférable de les implémenter directement car elles sont assez souvent utilisables.

Mnémonique	Opérandes	Fonction
move	r_i , r_k	$r_k = r_i$
movei	#valeur , r_k	r_k = valeur
add	r_i , r_j , r_k	$r_k = (r_i + r_j) \% 256$
addi	#valeur , r_k	$r_k = (r_k$ + valeur) % 256
sub	r_i , r_j , r_k	$r_k = (r_i + \sim r_j + 1) \% 256$
subi (*)	#valeur , r_k	$r_k = (r_k + \sim$valeur + 1) % 256
min (*)	r_i , r_j , r_k	$r_k = (r_i < r_j ? r_i : r_j)$
max (*)	r_i , r_j , r_k	$r_k = (r_i > r_j ? r_i : r_j)$
sltu	r_i , r_j , r_k	$r_k = (r_i < r_j ? 1 : 0)$
and	r_i , r_j , r_k	$r_k = r_i \& r_j$
andi	#valeur , r_k	r_k &= valeur
or	r_i , r_j , r_k	$r_k = r_i \mid r_j$
ori	#valeur , r_k	r_k |= valeur
xor	r_i , r_j , r_k	$r_k = r_i \wedge r_j$
xori	#valeur , r_k	r_k ^= valeur
shrui	#décalage , r_k	r_k >>= décalage
shli	#décalage , r_k	$r_k = (r_k$ << décalage) % 256
mulu	r_i , r_j , r_k (k pair)	$r_k = (r_i * r_j) / 256$ $r_{k+1} = (r_i * r_j) \% 256$
brz	#saut , r_i	PC = (r_i == 0 ? PC + saut : PC + 1)
brnz	#saut , r_i	PC = (r_i !=0 ? PC + saut : PC + 1)
bra	#saut	PC += saut

Tableau 2: Instructions non spécifiques

Le processeur élémentaire a pour particularité d'être totalement asynchrone, c'est-à-dire sans horloge et auto-séquencé. La notion d'asynchronisme est donc présente non seulement au niveau réseau, entre processeurs, mais aussi au coeur même de chaque processeur. Chaque PE exécute une nouvelle instruction dès que l'instruction précédente est terminée. En fait, étant donné que l'architecture du PE est de type RISC et est donc pipelinée, chaque étage de pipeline entame une nouvelle opération dès que les étages adjacents le permettent. Le modèle ne fait ainsi aucune hypothèse sur la durée d'exécution des instructions, qui est a priori non bornée mais finie. Les programmes doivent être conformes à ce modèle,

c'est-à-dire que leur spécification doit être indépendante de toute hypothèse liée à la vitesse d'exécution de chaque processeur. Certaines évaluations de performance du modèle seront par contre basées sur une distribution des temps d'exécution élémentaires, mais sans influencer la spécification des programmes.

5.2.3 Mémoire partitionnée associative

Chaque PE est muni d'une mémoire de données, pour stocker les pixels des images et l'ensemble des variables intermédiaires nécessaires aux calculs. Nous considérons dans la suite des images de résolution 256*256 pixels, et prenons un réseau de taille 16*16 PE (soit 256 processeurs). Les pixels étant répartis dans l'ensemble des processeurs, chaque processeur doit contenir (256*256)/(16*16)=256 pixels, pour stocker une image complète dans le réseau.

Chaque processeur va en fait être muni de plusieurs plans mémoire de 256 octets, et chaque plan mémoire pourra par exemple être associé à une variable ou à une image intermédiaire. L'organisation des données en mémoire est laissée au choix de l'utilisateur, mais on utilisera a priori les cases mémoires de même adresse des différents plans pour stocker des informations se rapportant à un même pixel du plan image. Lorsque les variables à stocker ne nécessitent pas un octet entier, il est bien sûr possible de considérer plusieurs champs dans chaque octet. Le nombre de plans mémoire dépend encore de la complexité des algorithmes, de la granularité visée, des possibilités d'intégration... Les algorithmes présentés plus bas pour l'évaluation de l'architecture nécessitent par exemple moins de 8 plans mémoire (pour des images en niveaux de gris). Il semble que l'intégration de 16 plans soit assez confortable pour bon nombre de traitements. Cela représenterait donc 16*256*8=32 kbits de mémoire de données par processeur, et un total de 8 Mbits pour tout le réseau.

La mémoire considérée dans cette architecture n'est pas une mémoire classique à accès par adressage. Il s'agit d'une mémoire associative (cf chapitre 2), qui permet à la fois un accès par adressage et un accès associatif ou "par le contenu". L'accès par adressage consiste à spécifier un numéro de plan et une adresse, et à transférer un octet entre un registre et un mot mémoire (instructions *load* et *store*, cf Tableau 3). L'accès associatif se fait en deux étapes. On commence par rechercher dans un plan donné un "motif", composé d'une valeur accompagnée d'un "masque", c'est-à-dire un autre octet indiquant les bits à prendre en compte (et "masquant" donc les autres bits de la mémoire). Cette opération a pour effet de positionner en parallèle dans la mémoire une colonne d'indicateurs, que l'on dénommera "tags", comportant un bit associé à chaque mot mémoire et qui réalise un marquage de l'ensemble des mots correspondant au motif recherché (instruction *search*, cf Tableau 3). La colonne de tags est commune à tous les plans mémoire et n'est modifiée que par l'instruction de recherche. Une autre instruction permet de compter le nombre de mots marqués, en effectuant la somme des tags (instruction *count*, cf Tableau 3). La deuxième étape utilise la valeur des tags pour effectuer l'accès en lui-même. L'opération de lecture associative permet d'obtenir la valeur d'un mot mémoire d'un plan donné dont le tag correspondant est activé, ainsi que son adresse dans le plan. Ce mot peut être choisi de manière arbitraire puisque seule

compte sa propriété "d'association". Nous considérons dans la suite que l'opération de lecture associative sélectionne le premier mot dont le tag est activé, c'est-à-dire celui dont l'adresse est la plus faible (instruction *read*, cf Tableau 3). Enfin, l'opération d'écriture associative modifie en parallèle tous les mots d'un plan donné dont le tag est activé, en remplaçant les bits des mots mémoire sélectionnés par le masque spécifié, par les bits correspondants de la valeur spécifiée (instruction *write*, cf Tableau 3).

Le plan est sélectionné par une valeur immédiate (sur 4 bits pour un modèle à 16 plans), alors que les valeurs recherchées ou à écrire, les masques et les adresses sont stockés dans des registres (notés respectivement r_v, r_m et r_a dans le Tableau 3). En effet, une allocation dynamique des plans ne paraît pas nécessaire pour les algorithmes considérés. L'octet stocké en mémoire à l'adresse *a* dans le plan *p* est noté *mem(p,a)* et le tag qui est associé à tous les mots d'adresse *a* est noté *tag(a)*.

Mnémonique	Opérandes	Fonction
search	#plan , r_v , r_m	$\forall i$, tag(i) = ((mem(plan,i) & r_m) == (r_v & r_m) ? 1 : 0)
count	r_n , r_t	r_n = ($\sum_i$ tag(i)) % 256 , r_t = ($\sum_i$ tag(i) == 256 ? 1 : 0)
read	#plan , r_v , r_a	r_a = inf { i , tag(i) == 1 } , r_v = mem(plan,r_a)
write	#plan , r_v , r_m	$\forall i$, mem(plan,i) = (tag(i) == 1 ? (r_v & r_m) | (mem(plan,i) & ~r_m) : mem(plan,i))
load	#plan , r_v , r_a	r_v = mem(plan,r_a)
store	#plan , r_v , r_a	mem(plan,r_a) = r_v

Tableau 3: Instructions liées à la mémoire de données

5.2.4 Interfaces de communications locales

Le modèle étant basé sur un réseau cellulaire et un asynchronisme à grain fin, chaque processeur communique avec ses quatre voisins de manière totalement asynchrone, c'est-à-dire sans aucune hypothèse sur les dates auxquelles les différents processeurs vont exécuter une instruction de communication. Les communications entre processeurs sont basées sur un mécanisme de type "passage de messages", emprunté aux systèmes concurrents à grain plus gros, et adapté dans une implémentation à grain fin. Chaque PE est muni d'une interface spécifique qui lui permet d'envoyer un message à un ou plusieurs de ses voisins et de recevoir un message en provenance d'un voisin particulier ou parmi un ensemble de voisins. Chaque message effectivement envoyé correspond à la création d'un jeton, et chaque lecture effective d'un message consomme un jeton et supprime l'information au niveau de l'interface. On parle de communication effective lorsque le contenu d'un message a pu être transféré entre le coeur d'un processeur et son interface. Nous considérons en effet des interfaces à mémoire bornée, et

des communications soit bloquantes, soit non-bloquantes. Certaines actions de communication peuvent donc se terminer sans avoir réussi le transfert.

Une communication bloquante est par définition effective puisqu'elle ne se termine qu'après la mémorisation du message dans l'interface (pour l'envoi) ou la lecture d'un message (pour la réception). Les interfaces étant pourvues d'une mémoire bornée, une mémorisation ou écriture ne peut se faire que si la mémoire n'est pas pleine (on ne peut écraser un message correspondant à un jeton non consommé). La lecture d'un message ne peut se faire que si la mémoire n'est pas vide (donc s'il existe un jeton). Lorsque la condition n'est pas remplie, le processeur est bloqué jusqu'à ce qu'un autre processeur modifie l'état des interfaces par une action de communication duale. Une communication bloquante a donc une durée non bornée, mais finie à partir du moment où le programme est correct (sans interblocage).

Une communication non-bloquante doit pouvoir se terminer indépendamment des actions de communication des autres processeurs. Elle commence par une prise de décision concernant l'état de l'interface (qui implémente le "canal" de communication), puis effectue ou non l'écriture ou la lecture d'un message en fonction de cet état. Si le résultat de cette décision est négatif, il n'y a pas transfert de données. Dans tous les cas, la communication (le terme englobant alors la décision et la transmission éventuelle d'une donnée) se termine en un temps fini et le résultat de son exécution est signalé au processeur correspondant, qui peut continuer son traitement, en commençant éventuellement par gérer ce résultat. Une communication effective correspond encore à la création puis la consommation d'un jeton. Cependant, dans le cas général, la création et la consommation du jeton sont indéterministes, dans le sens où l'on ne peut prédire en principe la présence du jeton. La création d'un jeton peut être impossible lorsque la décision indique qu'il n'y a pas de place libre et la consommation peut de même être impossible si la décision indique qu'il n'y a pas de jeton. Une communication non-bloquante par passage de messages a pour caractéristique de retourner une information indiquant si elle a été effective ou non.

La mémoire de l'interface, c'est-à-dire la capacité de mémorisation transitoire de messages, est basée sur des files de type FIFO ("First In First Out"), qui conservent l'ordre des messages d'un même canal. Ces files sont aussi appelées "buffers" ou "queues". La taille de ces files, qui correspond au nombre maximal de jetons pouvant être présents à un même instant, est bornée mais non précisée a priori. Cela implique que la spécification des programmes doit être indépendante de cette taille, et donc que les programmes soient corrects avec une taille minimale, correspondant à la mémorisation d'un seul message. Lors de l'évaluation du modèle, nous pourrons faire varier cette taille et observer l'évolution du comportement du programme, conçu pour fonctionner dans tous les cas. Augmenter la taille des FIFOs permet à une communication qui aurait été bloquée en écriture avec une taille inférieure, d'être effectuée sans attente et permet à une communication non-bloquante de devenir éventuellement effective. Nous verrons plus loin que l'influence de ce paramètre sur les performances d'exécution de certains algorithmes peut cependant s'avérer inattendue. On pourrait imaginer de faire varier dynamiquement ce

paramètre, par exemple avec une instruction qui modifie avant ou pendant l'exécution d'un programme le nombre de places disponibles dans l'interface pour mémoriser les messages.

Nous proposons de baser les communications entre processeurs sur des instructions de communication non-bloquantes, aussi bien pour l'envoi que pour la réception de messages. L'instruction *send* permet un envoi non-bloquant et l'instruction *get* permet une réception non-bloquante (cf Tableau 4). Les opérandes de ces deux instructions sont lues dans trois registres banalisés. Un premier registre, noté r_d, spécifie le voisinage à qui s'adresse la communication. Chaque PE est lié à quatre PE voisins dans le réseau considéré, et nous ajoutons un lien de communication avec un contrôleur global (cf paragraphe 5.2.5), considéré comme un cinquième voisin. Les communications avec les différents voisins sont indépendantes, et l'interface comporte un buffer pour chaque canal. Les bits r_d*<4>* à r_d*<0>* correspondent respectivement au contrôleur global, et aux voisins "Nord", "Est", "Sud" et "Ouest" (ces derniers pouvant respectivement se voir attribuer les coordonnées (x,(y-1)%16), ((x+1)%16,y), (x,(y+1)%16), et ((x-1)%16,y), si le processeur considéré a pour coordonnées (x,y)). Chaque message est constitué de la concaténation des valeurs de deux registres r_i et r_j, ainsi que des bits restants du registre qui indique le voisinage, contenus dans le champ r_d*<7:5>*, qui permettent par exemple de différencier plusieurs types de messages. L'instruction *send* essaye d'envoyer un message vers tous les voisins spécifiés dans r_d et modifie les bits de r_d*<4:0>* correspondant aux voisins à qui le message a pu être envoyé. L'instruction *get* essaye de recevoir un message d'un des voisins spécifiés dans r_d et modifie le bit de r_d*<4:0>* correspondant au voisin d'origine du message, si un message a été reçu. Si des messages ont été reçus de plusieurs des voisins spécifiés, le choix du voisin est a priori indéterministe.

Une communication bloquante peut être réalisée à partir d'une communication non-bloquante. Il suffit qu'elle soit précédée d'une synchronisation, qui attend que l'état de l'interface, en termes de jetons correspondant à des messages, permette d'effectuer l'ensemble des passages de messages prévus par cette communication. L'instruction *wait_b4_send* permet ainsi d'attendre que l'interface dispose d'une place pour envoyer un message à tous les voisins spécifiés dans r_d (cf Tableau 4). De même, l'instruction *wait_b4_get* attend qu'au moins un message ait été reçu d'un des voisins spécifiés dans r_d. Ces deux instructions ne modifient aucun registre. L'instruction *send* précédée de l'instruction *wait_b4_send* et l'instruction *get* précédée de l'instruction *wait_b4_get*, avec une valeur identique dans le champ r_d*<4:0>* de leurs opérandes, réalisent respectivement un envoi bloquant et une réception bloquante.

Bien que l'instruction *set_idle* ne fasse pas partie des instructions de communication entre processeurs, elle figure dans le Tableau 4 car elle est gérée par l'interface de communication. Elle est discutée au paragraphe 5.2.6.

Mnémonique	Opérandes	Fonction
send	r_d , r_i , r_j	Envoi non bloquant du message (r_d<7:5> , r_i , r_j) à tous les voisins spécifiés par un bit valant 1 dans r_d<4:0>. Les bits de r_d<4:0> correspondant aux voisins à qui le message a pu être envoyé sont positionnés à 0.
get	r_d , r_i , r_j	Réception non bloquante d'un message dans (r_d<7:5> , r_i , r_j) en provenance d'un des voisins spécifiés par un bit valant 1 dans r_d<4:0>. Si un message a été reçu, le bit de r_d<4:0> correspondant au voisin d'origine est positionné à 0.
wait_b4_send	r_d	Attente bloquante de la possibilité d'envoi d'un message à tous les voisins spécifiés par r_d<4:0> (r_d n'est pas modifié).
wait_b4_get	r_d	Attente bloquante de la présence d'un message reçu d'un des voisins spécifiés par r_d<4:0> (r_d n'est pas modifié).
set_idle	#flag	Positionnement d'un drapeau pour signaler un état d'activité (flag=0) ou d'inactivité (flag=1) au contrôleur global.

Tableau 4: Instructions liées aux communications

5.2.5 Entrées / sorties et contrôle global

Un principe de base concernant le fonctionnement global du réseau est que les problèmes dont on vise la mise en oeuvre deviennent, grâce à l'architecture, de type "compute bound", c'est-à-dire de temps d'exécution borné par le traitement et non par les entrées/sorties ("I/O bound"), ces dernières comprenant les flux globaux de données et de contrôle. Ce principe nécessite la validité d'au moins deux conditions. La granularité et l'autonomie de l'architecture doivent d'abord être telles que la quantité d'opérations nécessaires à ces entrées/sorties soit petite devant la quantité d'opérations de calcul effectuées dans la même partie du problème. Nous discutons ce point au paragraphe 5.3.1. La deuxième condition concerne l'indépendance des entrées/sorties vis à vis de l'exécution de la partie calculatoire. La circulation des flux de données et de contrôle doit pouvoir se faire concurremment à l'exécution et indépendamment de celle-ci, c'est-à-dire avec une influence minimale sur sa vitesse. Nous considérerons pour cela des solutions basées de nouveau sur le parallélisme et la notion de synchronisation minimale. Les principes de localité et de régularité nous amènent à utiliser une structure pipelinée, distribuée par lignes ou par colonnes dans tout le réseau. Chaque PE contient un élément de ce pipeline, et est relié à deux autres processeurs. On peut par exemple choisir de faire circuler le flux de données selon les lignes de processeurs et le flux de contrôle selon les colonnes, pour répartir les ports du système autour du réseau (cf Figure 31). Les éléments de ces pipelines fonctionnent la plupart du temps de manière totalement indépendante des

coeurs de processeurs et permettent d'acheminer les flux vers l'ensemble des processeurs du réseau. Après acheminement, la prise en compte de ces flux par les PE, qui correspond à la synchronisation entre les entrées/sorties et le coeur de traitement, peut se faire par exemple par un mécanisme d'interruption. Cette interruption peut être déclenchée par l'arrivée de la donnée (ou de l'instruction) elle-même (ou d'une requête du canal de données sortantes) ou par l'arrivée d'une commande provenant du flux de contrôle entrant.

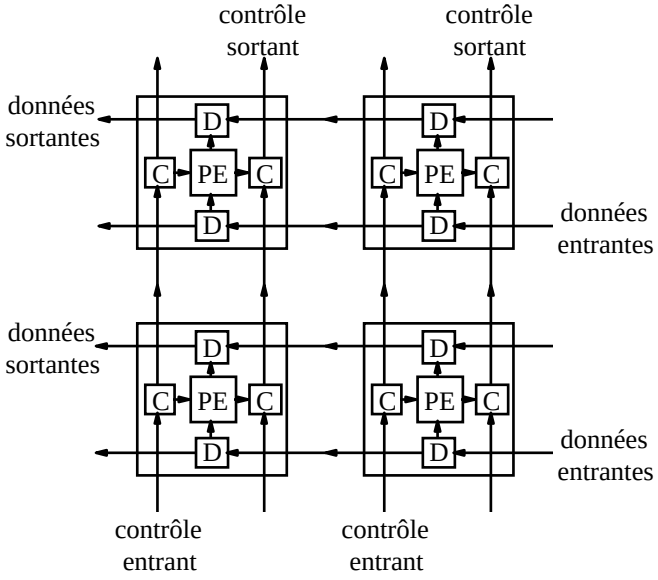

Figure 31: Pipeline par lignes et colonnes des flux d'entrée/sortie

Explicitons les différents flux d'entrée/sortie. Ce que nous appelons flux de données correspond aux entrées/sorties, a priori régulières, de pixels dans le réseau. Il existe de nombreuses possibilités pour l'acheminement et la mémorisation de ces flux de données. L'acheminement des données dépend de la spécification de l'environnement extérieur du réseau et du choix du partitionnement pour leur distribution dans le réseau. Un environnement classique traite les flux de données de manière séquentielle et selon le balayage vidéo, qui consiste à parcourir une image ligne par ligne. Le partitionnement, discuté au paragraphe 5.3.3, selon qu'il est figé ou laissé au choix de l'utilisateur, impliquera l'utilisation ou non d'un mécanisme de routage dans les pipelines, pour pouvoir adresser une donnée directement à un PE quelconque, ou au contraire pour distribuer simplement un ensemble de données consécutives vers une ligne de processeurs grâce à une structure de type "convertisseur série-parallèle". Nous avons déjà indiqué que la mémorisation ou l'émission d'une donnée peut être déclenchée par interruption, mais il faut aussi spécifier la position de la donnée dans la mémoire de chaque PE. Notons d'abord qu'on réservera a priori un plan mémoire pour le stockage des données d'entrée qui correspondent par exemple à une nouvelle image ou trame, et un plan mémoire pour la génération du flux de données sortantes, ce qui permet de rendre indépendants et concurrents les flux d'entrée et de sortie. L'adresse d'une donnée

pourra soit être transmise avec celle-ci dans le flux de données, soit spécifiée par une commande du flux de contrôle entrant, soit calculée par une fonction dans une routine de gestion d'interruption incluse dans chaque programme...

Le flux de contrôle entrant est bien sûr principalement constitué du flux d'instructions, qui est dupliqué vers l'ensemble des processeurs, et qui compose les programmes exécutés en mode SPMD. Une spécificité des éléments de pipeline qui acheminent ce flux est précisément d'assurer la duplication des instructions, qui du point de vue de l'environnement extérieur du réseau constituent un flux unique et centralisé, comme pour une structure de contrôle de type SIMD. Chaque PE possède sa propre mémoire programme, dont la taille doit correspondre au niveau de granularité intermédiaire déja évoqué. Il s'agit de pouvoir stocker et exécuter en mode SPMD un programme correspondant à une primitive de traitement, qui peut être par exemple une primitive de base utilisée pour la mise en oeuvre d'une API. Nous considérerons par exemple des programmes comportant au plus 256 instructions, ce qui permet de stocker des primitives relativement complexes, comme le montrent les exemples donnés au paragraphe 5.4. De la même manière que des plans mémoire sont réservés aux flux de données, on peut utiliser un mécanisme de "double buffer" pour rendre concurrents et indépendants la mémorisation et la lecture des instructions, ce qui double la taille de la mémoire programme, soit 512 mots d'instruction (la taille des mots d'instruction dépend du codage choisi, mais devrait être comprise entre 16 et 18 bits).

Le flux de contrôle entrant est aussi constitué de commandes et de messages. Les commandes peuvent être considérées comme des interruptions matérielles, qui peuvent servir à l'initialisation du système et à la prise en compte des flux d'entrée ou à la génération des flux de sortie. Une commande pourra en particulier réinitialiser le pointeur d'instruction (ou "compteur programme") et annuler les éventuelles instructions de synchronisation bloquante, pour exécuter un nouveau programme, préalablement chargé dans la partie réservée de la mémoire programme. Ces commandes, de même que la mémorisation des instructions, ne peuvent être influencées par l'exécution du programme. Les messages sont par contre reçus par les PE au niveau de leur interface de communication locale, comme un message en provenance d'un cinquième voisin correspondant à un contrôleur global (cf paragraphe 5.2.4). Ces messages ne seront pris en compte que par une instruction de communication contenue dans le programme. Ils peuvent servir à passer des paramètres aux programmes. Il n'est pas toujours intéressant que les commandes et les messages du flux de contrôle entrant soient dupliqués vers tous les processeurs, ni qu'ils soient acheminés vers un seul processeur précis. Une solution intermédiaire consiste à pouvoir spécifier une colonne dans laquelle ils seront dupliqués.

Les trois types d'informations constituant le flux de contrôle entrant peuvent circuler sur le même réseau de communication. Il suffit de discriminer les instructions, les commandes et les messages par le choix de l'encodage. Les éléments du pipeline devront pouvoir effectuer cette distinction et propager l'information en conséquence.

Le "flux de contrôle sortant" est constitué des messages envoyés à l'environnement extérieur du réseau (que nous avons parfois appelé "contrôleur global") par les processeurs via le "cinquième voisin" de leur interface de communication locale. Ces messages peuvent transporter des résultats autres que des pixels (qui sont a priori transmis dans le flux de données sortant), et peuvent correspondre en particulier aux paramètres calculés par des algorithmes de moyen niveau dits "d'extraction de caractéristiques" ("feature extraction") ou encore à une représentation de "haut" niveau d'un objet image (c'est-à-dire de niveau supérieur à la représentation par pixels). Les processeurs n'étant pas nécessairement synchronisés entre eux, ces messages peuvent être produits à n'importe quel instant et peuvent donc ne pas respecter de relation d'ordre à la sortie du système. Si une telle relation d'ordre est nécessaire, il est possible de synchroniser localement et explicitement les processeurs au niveau du programme avant l'émission des messages de résultat, ou il peut être suffisant d'indiquer dans le message les coordonnées du processeur émetteur.

Les différents flux d'entrée/sortie circulent en parallèle à l'intérieur du réseau, selon les lignes et colonnes de processeurs. Il n'est cependant pas toujours possible d'utiliser un tel nombre de ports du point de vue extérieur, à cause de la spécification ou des limitations d'entrée/sortie de l'environnement du réseau. Ceci ne constitue pas une limitation de l'architecture proposée puisque toutes les entrées/sorties considérées ont des contraintes de bande passante relativement faibles (par rapport à la bande passante totale interne du réseau, qui répond aux problèmes de mémorisation des algorithmes liés à l'image grâce à la distribution et au parallélisme). A l'extérieur du réseau, on pourra donc ajouter un système d'aiguillage (avec duplication conditionnelle) des flux entrants vers les différentes lignes ou colonnes, ainsi qu'un système dual de concentration pour les flux sortants (cf Figure 32).

Nous avons discuté dans ce paragraphe d'un certain nombre de généralités concernant les entrées/sorties du réseau. Il apparaît que de nombreuses solutions sont possibles, en fonction de la spécification et des contraintes de l'environnement du réseau, ainsi que du niveau de flexibilité du contrôle global. Nous ne proposerons pas ici de spécification plus fine de ce problème, et concentrerons l'étude sur le fonctionnement calculatoire du réseau, l'argumentation générale nous semblant suffisante pour démontrer la propriété "compute bound". L'évaluation des performances du réseau ne dépend alors que des caractéristiques fonctionnelles et architecturales locales.

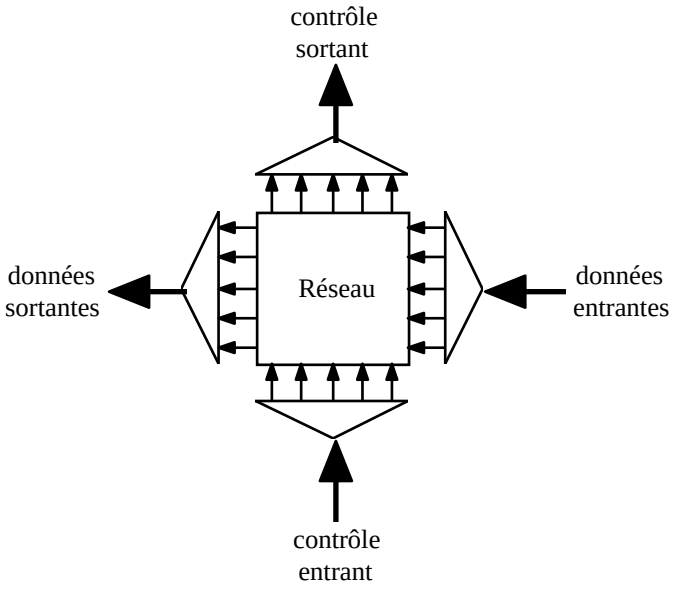

Figure 32: Diffusion et concentration des flux d'entrée/sortie

5.2.6 La détection globale de fin de calcul

Regardons tout de même un dernier point concernant le contrôle global. Bien que nous nous intéresserons principalement dans la suite aux propriétés d'exécution des programmes dans le réseau, il reste un problème relativement difficile à traiter dans un contexte purement asynchrone: la détection globale de fin de calcul. Nous avons en effet indiqué que le contrôleur global envoie une commande de "démarrage" d'un nouveau programme, lorsque le programme précédent a terminé sa tâche (le nouveau programme ayant été chargé dans la deuxième partie de la mémoire programme pendant l'exécution du précédent). La difficulté provient précisément de la détection de l'arrêt global du réseau, c'est-à-dire de signaler de manière sûre à l'environnement du réseau que tous les processeurs sont inactifs et vont le rester.

Nous ne développons pas ici une solution complète pour ce problème, mais souhaitons donner quelques indications sur l'approche envisagée. Il existe divers types d'algorithmes de détection de fin de calcul pour les architectures distribuées. On peut distinguer le problème de détection centralisée du problème de détection distribuée. Dans le premier cas, on peut considérer que la "terminaison" doit être détectée uniquement par un contrôleur global, ou par un processeur précis appartenant au système. La détection distribuée consiste au contraire à mettre en oeuvre un mécanisme qui permet à chaque processeur du réseau de savoir si tous les processeurs ont terminé leur traitement. Pour l'approche SPMD considérée, la détection centralisée est suffisante et a priori moins coûteuse. Il est suffisant de détecter l'inactivité globale seulement au niveau du contrôleur externe, puisque c'est ce contrôleur qui effectue la

synchronisation globale (minimale) du réseau et qui déclenche les exécutions successives des différents programmes. On considère donc que chaque processeur est capable d'indiquer son état local, soit actif soit inactif, sans que cela ne donne nécessairement d'indication sur la fin du traitement. Il existe en effet deux types d'algorithmes du point de vue de ce problème de terminaison: soit l'algorithme permet à chaque processeur de savoir qu'il n'a plus de traitement à effectuer, soit ce n'est pas possible car les traitements restants peuvent dépendre des autres processeurs. Les dépendances entre processeurs ne s'exprimant que par l'intermédiaire des canaux de communication locaux, cela revient à distinguer les algorithmes où un processeur peut savoir qu'il ne recevra plus de messages de la part de son voisinage, de ceux où cette connaissance n'est pas disponible. Nous souhaitons pouvoir traiter les deux types d'algorithmes.

Le premier type d'algorithme ne pose pas de problème particulier puisque l'activité (ou l'inactivité) globale suit un fonctionnement "monotone" au cours de l'exécution, c'est-à-dire que tous les processeurs passent une fois et une seule de l'état actif à l'état inactif, dès qu'ils savent qu'ils ne recevront plus de message et qu'ils ont terminé leur traitement. La combinaison des états de l'ensemble des processeurs permet trivialement de détecter l'inactivité globale. Dans le deuxième type d'algorithme, les processeurs peuvent seulement indiquer qu'à un instant donné, ils n'ont pas de message à traiter et qu'ils n'ont plus de traitement à effectuer. Ils décident alors d'indiquer un état inactif, mais repasseront à l'état actif s'ils reçoivent par la suite un message d'un des processeurs voisins. Il peut se poser dans ce cas un problème d'aléas, ou d'erreur transitoire, pour l'évaluation de l'état global, dans les modèles asynchrones à délais non bornés. En effet, le processeur voisin qui a émis un message pour "réveiller" le processeur inactif, peut décider de devenir lui-même inactif avant que le voisin en question n'ait effectivement changé d'état. La détection globale de fin s'appuie donc non seulement sur les états indiqués individuellement par les processeurs d'un point de vue algorithmique, mais aussi sur la présence de messages non consommés au niveau des interfaces de communication locale. Lors de l'envoi d'un message, il faut donc s'assurer du transfert d'activité qu'il représente.

Nous avons ajouté l'instruction *set_idle* (cf Tableau 4) pour permettre à chaque processeur d'indiquer son état d'activité à un niveau algorithmique, c'est-à-dire qui peut être explicité dans un programme. Le problème de détection de fin étant lié aux communications entre processeurs, nous pouvons considérer que l'instruction est gérée par les interfaces de communication. Cette instruction a pour effet de positionner la valeur d'un drapeau binaire, qui indique ce que l'on appellera l'(in)activité "algorithmique". Ce drapeau devra être combiné localement à cinq autres drapeaux, correspondant à l'état des interfaces de communication avec les cinq voisins, qui indiquera la présence d'un message en attente. L'envoi d'un message devra donc s'assurer de la modification du drapeau correspondant dans l'interface du processeur destinataire, avant d'être considéré comme terminé. La combinaison locale des drapeaux, indiquant à la fois l'(in)activité algorithmique et la présence (absence) de messages dans les

interfaces, sera appelée "(in)activité logique". C'est la combinaison des (in)activités logiques locales qui constituera la détection globale d'(in)activité. Ces deux niveaux de combinaisons de signaux représentent le point délicat du problème de terminaison.

Les deux scénarios typiques envisagés pour écrire la fin des programmes, correspondant aux deux types d'algorithmes cités plus haut, sont les suivants. Dans le premier cas d'algorithme, il suffit de mettre une instruction " *set_idle #1* " à la fin du traitement, qui indique l'inactivité algorithmique du processeur, suivie d'une instruction *wait_b4_get*, dans laquelle on spécifie un voisinage vide (r_d*<4:0>=00000*). Le processeur est alors bloqué pour de bon et ne pourra être réveillé que par une commande envoyée par le contrôleur global, après que celui-ci ait détecté que l'ensemble des processeurs sont dans un état d'inactivité logique. Dans le deuxième cas, on utilise aussi ces deux instructions, mais avec un voisinage a priori complet (r_d*<4:0>=11111* ou *01111*). Ces deux instructions sont exécutées lorsque le processeur n'a plus de traitement à effectuer et ne sait pas s'il recevra de nouveaux messages. Il est alors bloqué en attente soit d'un message d'un voisin ou du contrôleur global, qui lui indique d'effectuer un traitement complémentaire, soit d'être réinitialisé par une commande du contrôleur global, pour commencer un nouveau programme. La première instruction qui suit est normalement " *set_idle #0* ", qui indique le retour à l'état d'activité algorithmique, et enfin le programme effectue la lecture du message reçu et reprend une phase de traitement, avant de "repasser" au bout d'un certain temps par les deux instructions de retour à l'inactivité algorithmique puis logique.

5.3 Caractéristiques du modèle

Les paragraphes suivants présentent une synthèse des caractéristiques générales et des propriétés originales du modèle.

5.3.1 Convergence mémoire / traitement et granularité intermédiaire

Pour exploiter le parallélisme potentiel des algorithmes de traitement d'images, il faut bien sûr considérer leur mise en oeuvre sur des architectures elles-mêmes fortement parallèles. Cependant, l'utilisation de multiples chemins de données n'est pas le seul facteur à prendre en compte. En effet, un autre facteur limitant, après le nombre d'unités de calcul, est lié à la mémorisation des données intermédiaires du traitement. Les progrès technologiques ayant permis l'intégration monobloc de mémoires toujours plus grandes, à faible coût, les architectures à mémoire partagée ont été considérées comme l'évolution naturelle des architectures séquentielles. Cependant, les contraintes de complexité et de vitesse imposées par l'évolution et la convergence des traitements liés à l'image sont telles que la structure de mémoire centralisée est indéniablement un goulot d'étranglement. C'est pourquoi le passage du réseau computationnel au multiprocesseur à mémoire distribuée est essentiel à la mise en oeuvre des nouvelles générations d'applications. L'idée est bien sûr de répartir la bande passante totale, en en donnant une partie à chaque unité de traitement associée à un bloc mémoire, et de minimiser les

échanges résiduels entre le réseau qui effectue les traitements niveau pixel et le reste du système. Toutes les opérations de mémorisation sont donc confinées avec les unités de traitement et effectuées de manière parallèle, comme les traitements. Alors que le réseau computationnel, principalement constitué d'unités de traitements, souffrait des limitations dues aux entrées/sorties des données et du contrôle (le système étant dit "I/O-bound"), l'intégration d'un multiprocesseur à mémoire distribuée permet de mieux exploiter le parallélisme potentiel (le système devenant "compute-bound", i.e. borné par le calcul), grâce à la localité des accès mémoire. Pour que le taux de parallélisme reste important, il faut bien sûr que la granularité des unités de traitement (i.e. leur complexité relative, par rapport à l'ensemble des unités et des mémoires) soit plus faible que dans la notion classique de multiprocesseur, où chaque processeur est a priori une unité à haute performance, donc complexe. De même, il faut que le coût associé aux communications entre processeurs puisse être réduit au même niveau que le coût élémentaire de traitement, ce qui n'est généralement pas le cas dans les systèmes non-intégrés (i.e. où chaque processeur nécessite au moins un circuit intégré). Enfin, il faut aussi confiner le flux de contrôle, en donnant suffisamment d'autonomie aux unités de traitement, grâce à l'intégration de mémoires programme de faible taille qui permettent aux unités de rester actives sans limitation vis-à-vis des flux d'entrée, ce qui constitue de même une granularité plus fine que la notion classique de multiprocesseur, où le contrôle est soit centralisé, soit dupliqué dans des mémoires cache relativement importantes.

On peut aussi considérer le problème de la granularité dans la perspective opposée: partant d'un ensemble constitué uniquement d'unités de mémorisation, on peut envisager des architectures où l'on associe à chaque bloc mémoire une capacité de traitement minimum, ce qui définit une "mémoire augmentée" ou à traitement intégré ("logic-enhanced memory") (cf chapitre 2). On se rend alors compte qu'il existe un spectre de solutions allant de la mémoire partitionnée (tout est mémoire) au réseau computationnel (tout est traitement), en passant justement par les mémoires augmentées et les multiprocesseurs à mémoire distribuée, et que la mise en oeuvre efficace de traitements niveau pixel nécessite un niveau de granularité "intermédiaire". Ce niveau se justifie pleinement par les nouvelles possibilités d'intégration VLSI, qui permettent ainsi ce que l'on peut appeller la "convergence mémoire/traitement". Une caractéristique essentielle d'un tel modèle, vis-à-vis des systèmes de codage/traitement/synthèse d'images, est l'unification des accès mémoire. Cette convergence permet en effet d'intégrer dans une même structure les accès nécessaires aux calculs (données intermédiaires, Z-buffer...) et la fonctionnalité de "buffer vidéo", tout en permettant l'entrée/sortie concurrente de données de format variable, les entrées pouvant utiliser un format de représentation "pivot" et les sorties pouvant permettre la scalabilité spatiale (i.e. le choix de la résolution ou de la fenêtre) de l'affichage.

Il ne s'agit pas simplement de profiter de la technologie pour intégrer des structures complexes existantes, mais de redéfinir une architecture adaptée à la complexité et aux caractéristiques des traitements. C'est pourquoi l'architecture d'un processeur élémentaire d'un réseau intégré ne doit pas

suivre l'évolution des architectures des microprocesseurs, qui sont utilisés comme systèmes à part entière. Il faut au contraire favoriser de manière globale la capacité de mémorisation et de parallélisation, plutôt que la complexité locale. Les architectures VLSI massivement parallèles n'ont cependant pas connu un énorme succès, non seulement à cause des anciennes limitations d'intégration, mais aussi à cause de leur efficacité et scalabilité limitées, liées à l'utilisation de modèles de fonctionnement et de conception trop contraints. C'est pourquoi nous proposons d'associer plusieurs types de contrôle, qui permettent de relâcher les contraintes de séquencement, et de baser le modèle sur un asynchronisme à grain fin, qui relâche les contraintes de conception et améliore aussi les performances. La structure de réseau cellulaire toroïdal (cf 5.2.1) est quant à elle relativement naturelle.

5.3.2 Contrôle mixte SPMD, cellulaire, associatif et flot de données

Puisque la structure de l'information image et le principe de localité amènent à considérer une architecture cellulaire (à partitionnement par blocs), la généricité et la flexibilité doivent être introduits au niveau des possibilités de séquencement et de synchronisation, pour "contrebalancer" les contraintes introduites par la régularité spatiale. Le jeu d'instructions du processeur élémentaire, bien que réduit, permet une programmabilité importante, puisqu'il est basé sur celui d'un microprocesseur RISC et n'est pas propre à une application particulière. Le flot de contrôle est distribué, par duplication de manière asynchrone, ce qui permet l'autonomie et la localité de contrôle de tous les processeurs, qui fonctionnent en mode SPMD. Le séquencement de chaque processeur est alors dynamique, dans le sens où il dépend des données locales et de la position du processeur. Les communications asynchrones non-bloquantes permettent d'introduire des dépendances dynamiques indéterministes, qui forment un asynchronisme fonctionnel (cf 5.3.5). Cette possibilité n'est cependant pas toujours exploitable pour n'importe quel traitement. D'une manière générale, le fonctionnement SPMD permet de minimiser les synchronisations au niveau instruction, puisqu'elles doivent être explicitées par les instructions de communication. Cet asynchronisme au niveau instruction permet de minimiser le temps total de calcul, puisque chaque processeur peut enchaîner sans attente l'exécution de ses instructions, indépendamment des autres processeurs.

Le fonctionnement SPMD permet, par l'intermédiaire des instructions de communication, la génération d'un contrôle interne entre processeurs, à partir du contrôle global, c'est-à-dire qu'un processeur peut modifier ou même définir le contrôle des processeurs voisins. On obtient alors un "contrôle cellulaire", où les traitements à effectuer dans chaque processeur sont définis, non pas directement par le programme global, mais indirectement par l'effet du résultat de traitement de chaque processeur sur son voisinage, le contrôle étant ainsi défini localement par le réseau lui-même. Ce contrôle cellulaire peut correspondre simplement à un passage de paramètres entre processeurs, ou encore on peut imaginer que le programme exécuté par chaque processeur inclut la détermination du

contrôle des voisins en fonction des résultats de traitement locaux, ainsi que l'interprétation des messages de commande envoyés par ces voisins.

La combinaison du mode SPMD et de l'utilisation de mémoires associatives permet de mettre en oeuvre un contrôle à la fois associatif et flot de données, localement et entre processeurs voisins. En effet, on dispose localement d'un contrôle associatif (via les instructions liées aux mémoires associatives), qui permet à chaque processeur de traiter directement les données associées à une caractéristique liée à leur valeur. Un processeur peut par exemple accéder directement aux pixels appartenant à un objet donné, et définir le séquencement d'un algorithme en fonction d'une telle relation d'appartenance. Le contrôle associatif permet la mise en oeuvre d'un contrôle flot de données, où les données à traiter sont déterminées en fonction de la résolution de leurs dépendances mutuelles. Il suffit pour cela que la caractéristique d'association corresponde précisément à la résolution des dépendances, et que le programme mémorise et mette à jour cette information avec chaque donnée traitée. De cette manière, un processeur peut par exemple parcourir les pixels à traiter dans un ordre qui correspond à la propagation des mises à jour entre pixels, ou envoyer un message au processeur voisin, dont le contenu permettra une telle propagation et influera sur le séquencement de ce processeur pour réaliser un séquencement flot de données au niveau du réseau. Une propriété de cette combinaison de formes de contrôle est que le traitement des différents pixels va être effectué de manière dynamique dans un ordre quelconque, en fonction des dépendances intrinsèques de l'algorithme, qui tout en étant locales peuvent être irrégulières. Cette forme de contrôle mixte permet d'exploiter un parallélisme implicite (contenu dans une spécification flot de données du séquencement), tout en supprimant les synchronisations inutiles et les séquencements arbitraires et parfois inefficaces comme le traitement par blocs. Une application immédiate est le traitement efficace de régions de forme quelconque et d'algorithmes flot de données définis par des propagations locales (qui sont très courants en traitement d'images).

5.3.3 Partitionnement et entrelacement

Le principe de granularité intermédiaire, justifié par les possibilités d'intégration VLSI et l'optimisation globale du temps de calcul lié à l'activité effective d'une architecture parallèle (le taux d'accélération atteignant souvent une limite pour un trop grand nombre d'unités), implique le partitionnement des algorithmes et des données, pour être utilisés sur une architecture à degré de parallélisme limité. Les deux types de partitionnement classiques (par blocs) sont les partitionnements LSGP (localement séquentiel globalement parallèle) et LPGS (localement parallèle globalement séquentiel) [Kung88] [Priv93a] [Rose83]. Ces termes définissent en général à la fois une répartition des données dans la structure parallèle et un séquencement des processeurs pour effectuer l'ensemble des traitements du problème initial. Le partitionnement LSGP signifie que les différents blocs de données consécutives sont traités en parallèle, chaque bloc de l'ensemble initial étant associé à un processeur, et que les données d'un même bloc sont traitées séquentiellement. Le partitionnement LPGS signifie de

manière duale que les différents blocs de données sont traités séquentiellement alors que les données d'un même bloc sont traitées en parallèle, ces dernières étant réparties parmi l'ensemble des processeurs. Le séquencement lié à chaque répartition est implicitement basé sur un fonctionnement SIMD de l'architecture parallèle. La Figure 33 représente les deux types de partitionnement sur l'ensemble initial de données (une matrice 4*4), les numéros indiquant le processeur auquel est associée la donnée et les cases grisées montrant un ensemble de données qui sont traitées en parallèle par les différents processeurs.

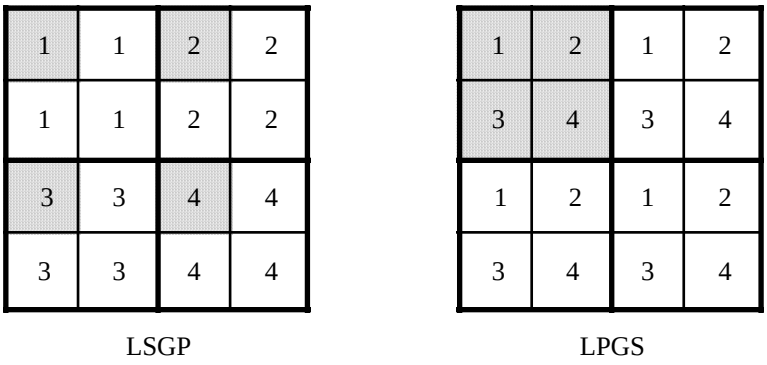

Figure 33: Illustration des partitionnements classiques par blocs

Le modèle de réseau présenté peut bien sûr utiliser les deux types de partitionnements du point de vue du traitement. Plusieurs types de partitionnement et séquencement sont en particulier étudiés dans la section 5.4.3, au travers d'exemples de mise en oeuvre d'un algorithme de filtrage morphologique. Du point de vue des entrées/sorties de données, le type LPGS est préférable si l'on considère que le système requiert par ailleurs un accès séquentiel aux pixels par balayage, la répartition nécessitant alors l'accès à des processeurs successifs pour le stockage de données successives, ce qui correspond à des entrées/sorties par lignes du réseau (alors que le type LSGP nécessite des accès successifs à un même processeur pour stocker un segment de données successives). Le type LPGS présente en fait un avantage essentiel, du point de vue du traitement, qui est confirmé par les évaluations présentées plus loin. Il permet un bon "équilibrage de charge" ("load balancing"), qui correspond à une répartition homogène des données et des calculs dans le réseau, en particulier pour des algorithmes irréguliers. En effet, cette répartition et la localité des dépendances entre pixels impliquent la décorrélation des données associées à un même processeur et permettent de maximiser l'activité du réseau pour des traitements ne concernant qu'une partie de l'image, en particulier pour les traitements basés-régions (contrairement au type LSGP, où seuls les processeurs associés à la région participent au traitement). En contrepartie, l'efficacité de ce partitionnement repose sur l'efficacité des communications entre processeurs voisins, qui gèrent les

dépendances locales entre pixels, cette propriété étant rendue possible par l'intégration et la mise en oeuvre asynchrone à grain fin du parallélisme.

Le fonctionnement SPMD et le contrôle mixte étant utilisés pour éliminer les contraintes du mode SIMD, la notion de partitionnement classique, comportant en particulier un traitement par blocs, ne s'applique ici que pour définir la répartition des données dans la structure parallèle, et ne contraint pas nécessairement le séquencement de l'algorithme "partitionné". Elle devient un cas particulier de séquencement, et est remplacée par la notion d'entrelacement, où chaque PE traite séquentiellement les pixels qui lui sont associés dans un ordre quelconque et possiblement de manière indépendante des autres PE, sauf lors de la prise en compte des messages envoyés par les PE voisins. L'entrelacement correspond à un séquencement faiblement contraint, déterminé dynamiquement par le contrôle associatif et flot de données. Il autorise un traitement à la fois localement et globalement parallèle, c'est-à-dire où le parallélisme peut s'appliquer à un instant donné à la fois à des pixels appartenant à une même région et à des pixels répartis de manière quelconque dans toute l'image. Il permet l'exploitation à grain fin du parallélisme implicite du modèle flot de données et illustre l'exploitation de l'asynchronisme au niveau du partitionnement, en relâchant les contraintes de séquencement classiques liées au traitement par blocs.

5.3.4 Asynchronisme structurel à grain fin

L'exploitation conjointe du parallélisme et de l'asynchronisme (niveau instruction ou opération) ne prend véritablement son sens que lorsque l'architecture est basée sur un asynchronisme structurel à grain fin (niveau portes logiques). Il s'agit de conception VLSI asynchrone. Rappelons que les propriétés principales des circuits asynchrones sont la localité des synchronisations, l'insensibilité aux délais, le fonctionnement flot de données, le calcul en temps minimum, la modularité, la robustesse, et la scalabilité architecturale et technologique (i.e. l'évolution linéaire des performances avec le degré de parallélisme ou la vitesse des dispositifs). La localité permet de s'affranchir de la limitation des performances et de la scalabilité, ainsi que de la difficulté de conception, dues à l'utilisation d'un signal d'horloge global dans les méthodologies standard de conception VLSI. Cette localité, associée à une technique de conception qui permet l'exploitation de chaînes critiques dynamiques, permet l'utilisation de structures à temps de calcul minimum, qui sont capables de détecter la fin d'un calcul et d'enchaîner les traitements au plus tôt (le temps effectif de calcul des opérateurs arithmétiques, entre autres, varie en effet fortement en fonction des données). Cette propriété donne tout son intérêt au contrôle SPMD, où chaque processeur peut exécuter son traitement au plus tôt indépendamment des autres PE, en fonction de ses propres données et de leurs dépendances, les synchronisations entre PE étant par ailleurs minimisées et explicitées au niveau instruction. Les seules synchronisations globales qui sont effectuées, pour la détection de fin et la diffusion de la commande de démarrage, ont un coût négligeable devant l'exécution d'un programme.

Les circuits asynchrones ont un fort potentiel de synchronisation et permettent la mise en oeuvre à grain fin, et avec de hautes performances, de mécanismes généralement considérés comme très coûteux, en particulier concernant les "communications asynchrones" entre unités (qu'il s'agisse de points mémoire ou de microprocesseurs). La gestion d'un protocole de requête/acquittement n'est pas réservée aux architectures à gros grain, et peut au contraire être réalisée au niveau porte logique, le surcoût d'une telle implémentation par rapport à une structure RTL synchrone étant peut-être même inférieur à la pénalité introduite par l'horloge. Les circuits asynchrones permettent en effet une meilleure exploitation du concept de pipeline, introduisant entre autres de nouvelles possibilités de "pipeline élastique" et de superpipeline [Suth89] [Rena97]. Les interfaces et les primitives de communication entre processeurs, dont l'efficacité est primordiale dans l'architecture présentée, peuvent être implémentées en matériel à l'aide de structures peu complexes et très rapides, validant ainsi l'hypothèse que le coût d'une instruction de communication locale est du même ordre que le côut de n'importe quel autre opérateur constituant un processeur élémentaire. Les circuits asynchrones permettent en particulier une implémentation efficace et robuste des communications non-bloquantes, à l'aide de portes élémentaires appelées "arbitres" ou "synchroniseurs", qui autorisent la "détection du canal vide" au plus tôt.

5.3.5 Asynchronisme algorithmique, séquencement dynamique et indéterminisme

Les communications non-bloquantes sont à la base de modes de séquencement qui améliorent l'efficacité du réseau. Elles permettent en effet de maximiser la coopération entre processeurs tout en minimisant les synchronisations liées aux communications. Elles sont de plus indispensables pour les algorithmes où le nombre de messages échangés entre processeurs n'est pas connu a priori, car la localité des synchronisations peut entraîner des interblocages, à cause de la taille finie des buffers de mémorisation des messages et de la présence possible de cycles dans le schéma de communication. Elles permettent enfin la mise en oeuvre d'algorithmes présentant un asynchronisme "fonctionnel" ou "algorithmique", où certaines dépendances entre données peuvent varier dynamiquement au cours de l'exécution, en fonction d'un paramètre externe à la spécification de l'algorithme, autre que la valeur des données. Il s'agit d'une généralisation de l'asynchronisme fonctionnel exploité dans le circuit AMPHIN, où le paramètre externe était le temps d'exécution des opérations élémentaires, qui n'est pas connu a priori et peut dépendre des conditions de fabrication ou de fonctionnement du circuit. L'asynchronisme fonctionnel peut être spécifié de manière générale par l'ajout d'un choix indéterministe aux formalismes de description flot de données, dont les extensions incluant la notion de choix sont généralement déterministes et prennent uniquement en compte la valeur des données. Dans le cas d'une communication non-bloquante, le choix indéterministe réside dans la "détection de canal vide", où l'interface de communication doit décider au plus tôt de la présence ou de l'absence de message, sachant qu'il n'existe strictement aucune synchronisation ni hypothèse temporelle entre les deux unités qui communiquent. Le résultat de ce choix peut, si le programme exploite cette possibilité, influencer le séquencement et les

dépendances effectivement prises en compte, car les dépendances entre données qui sont transférées entre processeurs par les communications peuvent être éventuellement supprimées, ce qui constitue une nouvelle forme de relâchement des contraintes de séquencement. En pratique, il est possible d'effectuer un calcul dans un processeur en utilisant les seules données disponibles localement à cet instant, c'est-à-dire sans attendre que toutes les dépendances initialement spécifiées soient effectivement résolues par réception de messages des autres processeurs. Soit un calcul intermédiaire est effectué à partir de ces données disponibles, soit le calcul est retardé et est remplacé par un autre calcul possible localement. Une autre possibilité est de supprimer un message reçu pour ne pas tenir compte de la dépendance correspondante. L'exploitation de l'asynchronisme fonctionnel dans une optique de calcul "au plus tôt" s'intègre naturellement avec le mode de fonctionnement SPMD et le contrôle associatif et flot de données, pour peu que l'algorithme s'y prête. Nous illustrons cette exploitation dans la section 5.4.3, avec l'exemple de la reconstruction en niveaux de gris, et montrons l'accélération rendue possible par la combinaison de ces différents modes. L'asynchronisme fonctionnel peut être bénéfique à la fois par la minimisation des synchronisations au niveau instruction et aussi par une accélération à un niveau global, correspondant à une minimisation des propagations de dépendances intermédiaires (entre pixels dans des régions étendues de l'image).

5.4 Evaluations

Nous présentons dans cette partie des évaluations du modèle, basées sur un simulateur au niveau instruction et la mise en oeuvre originale de plusieurs primitives algorithmiques (dont les descriptions en assembleur sont données en annexe).

5.4.1 Simulation du modèle

Un simulateur du modèle de réseau SPMD asynchrone a été programmé en langage C. Il s'agit d'un simulateur événementiel au niveau instruction, comportant un tirage pseudo-aléatoire des temps d'exécution. Il contient une routine d'analyse grammaticale et syntaxique simplifiée, qui permet d'écrire des programmes en langage assembleur dans un fichier texte séparé, dont le nom est donné en paramètre d'entrée du simulateur. Cet assembleur intégré permet en particulier de gérer des labels pour les instructions de branchement. Une autre entrée de la simulation peut être le nom d'un fichier contenant une image, qui sera lue puis mémorisée de manière adéquate selon le partitionnement utilisé. Il est possible de comparer le résultat de l'exécution avec le résultat d'une routine en langage C, ou avec le contenu d'un fichier de référence. Le simulateur peut aussi générer au cours de la simulation une série de fichiers image, qui correspondent à l'évolution du contenu des mémoires des PE, par exemple à intervalles réguliers de temps virtuel. Ces fichiers peuvent par la suite être concaténés et visualisés sous la forme d'une animation vidéo, ce qui permet de voir l'évolution des calculs au cours d'un algorithme. Enfin, le simulateur peut fournir un certain nombre de mesures liées à l'exécution de l'algorithme,

comme le temps de calcul (estimé en se basant sur les temps de calcul élémentaires introduits), le nombre d'instructions exécutées par chacun des processeurs (en particulier les nombres minimum, maximum et total d'instructions), la courbe d'activité logique temporelle du réseau (i.e. l'évolution du nombre de processeurs à l'état d'inactivité logique), le nombre maximum de messages effectivement présents à un même instant dans un des liens locaux entre PE au cours de l'exécution... Ce dernier nombre est borné par la capacité de mémorisation des interfaces de communications locales, que l'on appellera aussi taille des "buffers", qui est le seul paramètre lié au dimensionnement de l'architecture dont nous ayons étudié l'effet sur les performances, en particulier dans l'algorithme présenté au paragraphe 5.4.3.c.

Pour effectuer la simulation, on mémorise un certain nombre d'informations pour chaque processeur, comme le contenu des mémoires associées, le pointeur d'instruction, la valeur des registres, l'état des interfaces de communication... Une structure de queue est utilisée pour mémoriser les événements à exécuter. Un événement contient en particulier une date et un numéro de processeur. Les pointeurs d'instruction sont tous initialisés de manière à pointer sur la première instruction du programme à exécuter. Ensuite, un événement est ajouté dans la queue pour chacun des processeurs, pour démarrer l'exécution. La date de ce premier événement n'est pas nécessairement la même pour tous les processeurs (elle est d'ailleurs arbitraire a priori puisque le modèle est complètement asynchone) et peut être décalée de manière à prendre en compte un temps de propagation des commandes dans le réseau. La simulation est constituée d'une boucle qui s'arrête lorsque la queue d'événements est vide. Cette condition doit aussi correspondre à celle de détection globale de fin de calcul. A chaque étape de la boucle, un événement est retiré de la queue, ce qui identifie un processeur du réseau. Le pointeur d'instruction de ce processeur désigne une instruction, qui est exécutée par mise à jour des informations liées au processeur. On associe normalement à cette action un temps d'exécution (ou délai), qui dépend a priori non seulement du type d'instruction mais aussi de la valeur des opérandes, voire même du numéro du processeur et du temps (ces deux derniers facteurs permettant par exemple de prendre en compte des variations spatiales et temporelles de l'alimentation ou des dispersions technologiques et donc des vitesses dans le réseau...). Nous avons choisi pour l'instant d'utiliser un modèle de délai simplifié, qui consiste à faire un tirage aléatoire uniforme dans un intervalle de nombres entiers. Le simulateur permet de définir cet intervalle pour chaque type d'instruction, mais nous avons choisi en première approximation de prendre l'intervalle [10,15] pour toutes les instructions, sauf pour l'instruction *mulu*, à laquelle a été associé l'intervalle [40,60] à titre d'exemple. La valeur "absolue" de ces nombres, considérés comme des temps de cycle en nanosecondes, ne sert pour le moment qu'à l'évaluation d'ordres de grandeur basés sur une technologie environ équivalente à celle du circuit AMPHIN, et donc sous-estimés pour les technologies d'intégration visées qui permettront des temps de cycle bien inférieurs à 10ns (disons autour de 2ns...). En additionnant la date contenue dans l'événement et le temps

d'exécution de l'instruction, on détermine enfin la date du prochain événement qui sera exécuté pour ce processeur. Cet événement est inséré dans la queue par tri selon la date, de manière à ce que le premier élément de la queue soit toujours le prochain événement à exécuter (les événements de même date étant exécutés dans un ordre arbitraire).

Une exception concerne les instructions *wait_b4_get* et *wait_b4_send* utilisées pour les communications bloquantes. Dans le cas où la condition de réception ou d'émission d'un message n'est pas vérifiée, le pointeur d'instruction n'est pas modifié et aucun événement n'est remis dans la queue. Le processeur correspondant "disparait" ainsi de la simulation tant qu'il est bloqué. Il sera débloqué par l'exécution d'une instruction *send* ou *get* par un processeur voisin, qui introduira dans la queue un événement pour ce processeur. A la date de "réveil", l'instruction d'attente bloquante sera de nouveau exécutée, et se terminera normalement cette fois-ci.

Le simulateur permet pour l'instant de réaliser des évaluations correspondant à l'exécution d'un seul programme et ne traite pas les flux d'entrée/sortie. La structure et la gestion des événements permettra facilement d'inclure la simulation de plusieurs types de processus concurrents. Plusieurs niveaux de fonctionnement en mode "trace" ou "pas à pas" permettent la mise au point des programmes (et du simulateur !).

Avant de passer aux exemples de programmes, déterminons la puissance de calcul "théorique" du réseau. La puissance crête correspond à l'exécution d'une instruction par chacun des processeurs pendant le temps de cycle minimal, soit 256 instructions en 10ns, ce qui donne 25600 MIPS soit environ 25 GIPS (giga instructions par seconde). On considérera plutôt la puissance crête "moyenne", calculée à partir du temps moyen d'exécution des instructions (en toute rigueur, dans un programme donné, c'est-à-dire en tenant compte de la quantité d'instructions de chaque type). Si le programme ne contient pas l'instruction *mulu*, la puissance crête moyenne, correspondant à l'exécution de 256 instructions en 12,5ns, est de 20480 MIPS soit environ 20 GIPS. La puissance de calcul moyenne correspondant à l'exécution d'un programme est obtenue en divisant le nombre total d'instructions exécutées dans le réseau par le temps total d'exécution estimé. Elle correspond à l'activité physique moyenne ou encore à la puissance consommée moyenne du réseau (le nombre total d'instructions exécutées correspondant à une énergie consommée, à une pondération par instruction près). Elle ne peut être sensiblement inférieure à la puissance crête moyenne qu'à cause des instructions de communications bloquantes (plus précisément des temps d'inactivité dûs aux instructions *wait_b4_get* et *wait_b4_send*), de l'utilisation de l'instruction *mulu* (dont le temps d'exécution est supérieur à la moyenne), de la dispersion des dates de terminaison locale (tous les processeurs n'étant plus actifs à la fin du traitement), ou des décalages dans les dates des événements de démarrage du programme (ce dernier facteur étant négligeable). Dans la suite, certains résultats de simulation seront arrondis pour tenir compte des variations liées à l'initialisation de la fonction pseudo-aléatoire.

5.4.2 Produit matriciel

Description du programme

Commençons par un programme simple, consistant à effectuer le produit de deux matrices de taille 16*16, c'est-à-dire de la taille du réseau. Le programme est donné en annexe (fichier "produit.asm"). Les coefficients des matrices sont des entiers signés codés sur 8 bits. Les coefficients du produit sont des entiers signés codés sur 19 bits, avec une extension de signe automatique qui donne en fait un résultat sur 24 bits, stocké dans trois registres de 8 bits.

Le produit $C=A.B$ s'écrit: $\forall i,j, \quad c_{ij} = \Sigma_k \, a_{ik}.b_{kj}$,

où a_{ij}, b_{ij} et c_{ij} sont des coefficients situés sur la ligne i et la colonne j.

On suppose que chaque processeur de coordonnées $(x,y)=(j,i)$ (l'abscisse correspondant au numéro de colonne et l'ordonnée au numéro de ligne) contient initialement les coefficients a_{ij} et b_{ij}, dans ses registres r_0 et r_1, et a pour objectif de calculer c_{ij} dans le triplet de registres (r_{10},r_{11},r_{12}), où r_{10} contient les poids forts et r_{12} les poids faibles. On suppose aussi que chaque processeur connaît ses coordonnées et qu'elles sont stockées dans les registres $r_{14}=x=j$ et $r_{15}=y=i$. Notons que ces coordonnées peuvent être indiquées aux processeurs à l'initialisation, alors qu'ils sont encore indifférenciés, en envoyant au réseau les deux matrices $x_{ij}=j$ et $y_{ij}=i$.

Le calcul d'un coefficient c_{ij} nécessite de connaître tous les coefficients de la matrice A situés sur la même ligne (notés a_{ik}) et tous les coefficients de la matrice B de la même colonne (notés b_{kj}). L'algorithme peut s'écrire en deux étapes. La première étape consiste à faire parvenir les familles de coefficients a_{ik} et b_{kj} au processeur (j,i), et ceci pour tous les couples (i,j). Cette étape est donc constituée de communications et de mémorisations. La deuxième étape consiste à effectuer le calcul de c_{ij} à partir des a_{ik} et b_{kj} stockés localement, et se fait donc de manière totalement indépendante entre processeurs.

La première étape de communication des coefficients (lignes 20 à 47 du fichier "produit.asm") s'écrit de manière simple et régulière grâce à la structure toroïdale du réseau, avec $4n$ actions de communication bloquantes par PE (n étant la taille de la matrice, soit $n=16$ dans notre étude). Il suffit de faire circuler les coefficients a_{ij} par rotation le long des lignes du réseau et les coefficients b_{ij} par rotation le long des colonnes du réseau. A chaque étape (locale), chaque PE de coordonnées (j,i) stocke les deux coefficients a_{ip} et b_{qj} courants, propage a_{ip} vers son voisin EST et b_{qj} vers son voisin SUD (on pourrait évidemment choisir un autre sens de rotation), puis reçoit un nouvel a_{ip} de son voisin OUEST et un nouveau b_{qj} de son voisin NORD. Au bout de n étapes, les coefficients ont "fait le tour". La Figure 34 illustre la circulation des coefficients dans le réseau (avec $n=4$). Toutefois, cette figure ne doit pas être vue comme une évolution temporelle de l'exécution de l'algorithme, mais comme une illustration "fonctionnelle" équivalente car l'algorithme ne comporte que des synchronisations locales.

A chaque fois qu'un coefficient est reçu par un processeur, il doit être mémorisé à une adresse correspondant aux indices du coefficient. Les coefficients a_{ik} sont stockés à l'adresse k dans le plan

mémoire numéro 0, et les coefficients b_{kj} à l'adresse k dans le plan mémoire numéro 1, du point de vue de chacun des processeurs de coordonnées (j,i). On utilise un pointeur d'adresse pour les a_{ik}, stocké dans r_5, et initialisé à j, dont la valeur est contenue dans r_{14} et correspond à l'abscisse du processeur, et un pointeur d'adresse pour les b_{kj}, stocké dans r_6, et initialisé à i, dont la valeur est contenue dans r_{15} et correspond à l'ordonnée du processeur. Les deux pointeurs sont décrémentés d'une unité à chaque nouveau stockage, modulo n, ce qui correspond à une "rotation" complète de chaque pointeur dans l'intervalle $[0,n-1]$.

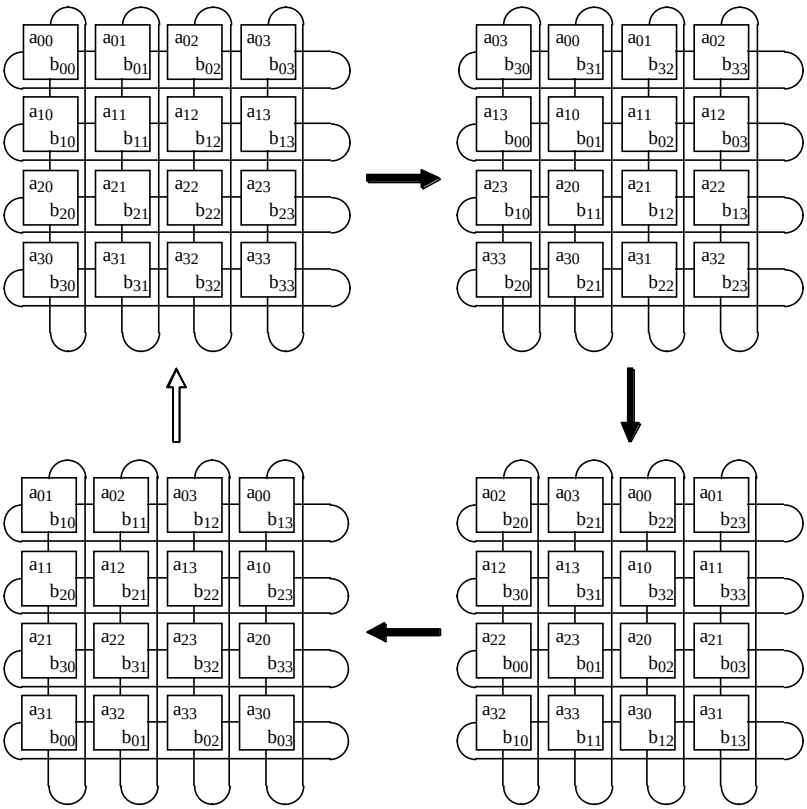

Figure 34: Circulation des coefficients dans le réseau

Dans la deuxième étape de l'algorithme (lignes 49 à 84), il suffit de parcourir séquentiellement les deux plans mémoire et d'accumuler le produit des coefficients de même adresse. Nous ne détaillerons pas la mise en oeuvre de l'arithmétique multiprécision, c'est-à-dire qui gère des opérandes de taille supérieure à celle des registres, en les stockant par morceaux dans plusieurs registres consécutifs. Le programme proposé présente de plus une complexité supplémentaire de gestion d'opérandes signés à partir d'une instruction de multiplication non signée (ceci à titre d'exercice, l'instruction *mulu* pouvant être remplacée par une multiplication signée).

Résultats de simulation

Le Tableau 5 résume les évaluations obtenues par simulation du programme donné en annexe.

Taille du programme	66 instructions	
Init. de la fonction pseudo-aléatoire (coeff. et temps)	seed1	seed2
Taille des buffers des interfaces	>1	>1
Nombre max. de messages dans un buffer	1	1
Nombre min. d'instructions exécutées par les PE	662	662
Nombre max. d'instructions exécutées par les PE	688	694
Nombre total d'instructions exécutées par le réseau	172928	172860
Temps total d'exécution estimé (ns)	9291	9304
Puissance moyenne de calcul (GIPS)	18,6	18,6

Tableau 5: Evaluations du programme "produit.asm"

Ce programme comporte 66 instructions, mais pourrait en comporter moins de 50, en introduisant dans le modèle une instruction de multiplication signée. Le Tableau 5 donne les résultats de deux simulations ne différant que par l'initialisation de la fonction pseudo-aléatoire qui détermine la valeur des coefficients des matrices et les temps d'exécution élémentaires. Le temps d'exécution du programme indique une puissance de plus de 100000 produits matriciels à 16*16 coefficients de 8 bits, par seconde. La dispersion des nombres d'instructions exécutées par les différents processeurs (i.e. l'écart relatif entre les nombres maximum et minimum, parmi l'ensemble des PE, d'instructions exécutées par chacun des PE) est de l'ordre de 5% et provient uniquement de la gestion du signe des opérandes avec une multiplication non signée, le reste de l'algorithme étant parfaitement déterministe (c'est-à-dire de séquencement indépendant des données et du temps). La puissance moyenne de calcul, qui est de 18,6 GIPS, est inférieure à la puissance crête moyenne calculée plus haut (20 GIPS) à cause du temps d'exécution des instructions de multiplication et de la dispersion relative des temps de calcul, et donc des dates de terminaison, des différents processeurs (qui n'est pas complètement négligeable dans ce cas car le nombre total d'instructions exécutées n'est pas très grand). Il est peu probable dans ce programme que les communications bloquantes engendrent des temps d'attente notables ou simplement non nuls car dans la phase de communication des coefficients, le séquencement est complètement déterministe et identique pour tous les processeurs, et la dispersion des temps d'exécution des instructions ne désynchronise pas beaucoup les processeurs. On note d'ailleurs que le nombre maximum de messages présents dans un buffer d'interface à un instant donné ne dépasse pas 1, avec une taille de buffers supérieure à ce nombre maximum, ce qui signifie qu'il n'y a pas eu deux envois successifs de messages sans qu'il n'y ait eu de

réception correspondante entre temps, et donc qu'il n'y a pas d'attente pendant un envoi bloquant, même avec une taille de buffer minimale.

L'activité logique temporelle du réseau est représentée Figure 35. Le programme correspond au premier type d'algorithme (cf 5.2.6) du point de vue de la détection globale de fin de calcul. Il commence par l'instruction " *set_idle #0* " et se termine par les instructions " *set_idle #1* " et " *wait_b4_get* r_7 ", sachant que la valeur du registre r_7 correspond à un voisinage vide. L'évolution de l'activité logique est de type "monotone", c'est-à-dire que tous les processeurs deviennent actifs au démarrage du programme (on voit sur la Figure 35 la prise en compte d'un temps de propagation des commandes du contrôleur global dans le réseau) et redeviennent définitivement inactifs à la fin du traitement local qui est complètement déterministe (la Figure 35 montre la dispersion des temps de calcul, qui tout en étant assez faible, n'est pas complètement négligeable devant le temps total de traitement).

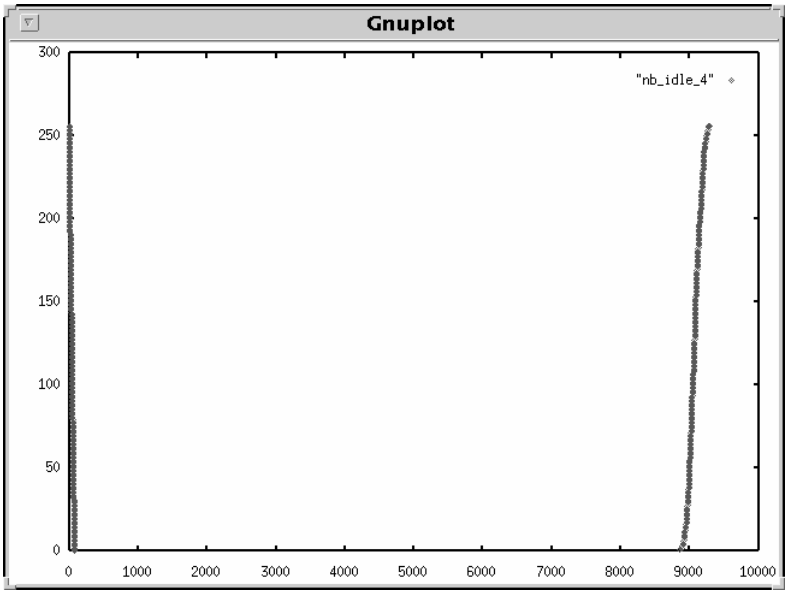

Figure 35: Activité logique du réseau (nombre de PE inactifs au cours du temps)

5.4.3 Filtres morphologiques par reconstruction

Considérons maintenant la mise en oeuvre de filtres morphologiques par reconstruction, primitives de traitement d'image très utiles pour la segmentation et l'analyse d'images, déjà introduites aux chapitres 1 et 4. Nous présentons ici trois versions originales d'algorithmes de reconstruction en niveaux de gris. Les algorithmes classiques de type parallèle itératif, séquentiel (récursif) par balayage, et séquentiel par propagation à base de queues de pixels, sont présentés dans [Sale92] [Sale96] [Vinc93].

a) Reconstruction LPGS itérative

Rappel de l'algorithme parallèle itératif

L'algorithme de reconstruction parallèle classique consiste à appliquer de manière itérative en chaque pixel une dilatation élémentaire, soit un maximum sur le voisinage, suivie d'un minimum avec l'image de référence (ces deux opérations étant parfois appelées "dilatation géodésique de taille unitaire") [Sale92] [Vinc93]. L'image sur laquelle est effectué ce traitement est appelée "image marqueur" et sa valeur initiale est une érosion de l'image de référence, pour réaliser une ouverture par reconstruction (Figure 36). Les itérations sont répétées jusqu'à idempotence, c'est-à-dire stabilité (ou "convergence") de l'image marqueur (l'idempotence signifie qu'une nouvelle itération ne modifie pas l'image).

La parallélisation à grain fin d'un tel algorithme est immédiate. En associant un processeur à chaque pixel, l'algorithme local consiste alors à:

- envoyer la valeur courante du pixel de l'image marqueur à tous les processeurs voisins,
- recevoir la valeur que chaque processeur voisin a envoyé,
- calculer le maximum de ces valeurs et de la valeur courante puis le minimum de ce résultat et du pixel de l'image de référence,
- mettre à jour la valeur courante avec le résultat de cette dilatation géodésique,
- et recommencer une nouvelle itération.

Figure 36: Exemple de filtre morphologique par reconstruction. De gauche à droite: image source ("camera"), érosion de taille 32, ouverture par reconstruction

Partitionnement et séquencement

La principale transformation de cet algorithme pour sa mise en oeuvre sur l'architecture étudiée consiste à effectuer un partitionnement, c'est-à-dire assigner un ensemble de pixels à chaque processeur. Dans notre cas, chaque PE doit s'occuper de 256 pixels. Considérons un partitionnement de type LPGS (cf 5.3.3) et un traitement itératif par blocs. On découpe l'image en blocs de taille 16*16 pixels et on applique l'algorithme parallèle itératif sur chaque bloc, avec un nombre n_{ite_bloc} fixe d'itérations à chaque

étape, et on effectue ceci sur les différents blocs de l'image en les parcourant séquentiellement et a priori selon un balayage standard. Choisissons de plus un traitement séquentiel de type récursif au niveau bloc, c'est-à-dire où le traitement d'un bloc utilise les valeurs calculées dans les blocs déjà mis à jour dans le balayage (ou "itération image") en cours (en fait seulement les valeurs des pixels situés sur certains bords des blocs adjacents déjà traités, puisque les dépendances sont purement locales). Ces deux transformations modifient a priori l'algorithme. Cependant, nous avons vu au chapitre 4 que n'importe quel ordre de mise à jour des pixels conduit au même résultat pour l'algorithme de reconstruction. Notons quand même que pour $n_{ite\_bloc}=1$, un balayage complet de l'algorithme LPGS itératif par blocs et non récursif est "algorithmiquement équivalent" (i.e. du point de vue des dépendances) à une itération de l'algorithme initial (parallèle non partitionné). Nous utilisons finalement un balayage alternatif, souvent associé au traitement récursif [Vinc93], qui consiste à alterner un balayage descendant standard ("raster scan") avec un balayage montant inverse "anti-raster" (ce qui permet d'accroitre encore la vitesse de convergence). Rappelons que le programme ne définit qu'un séquencement local et que le séquencement global n'est pas nécessairement totalement défini. Tous les séquencements globaux permis par le programme sont cependant algorithmiquement équivalents lorsque seules des communications bloquantes sont utilisées, ce qui est le cas du programme considéré (contrairement aux algorithmes de reconstruction présentés dans les paragraphes suivants et à l'algorithme mis en oeuvre par le circuit AMPHIN).

Effets de bord

Une partie souvent négligée dans la spécification ou l'évaluation d'algorithmes est la prise en compte des irrégularités de traitement dues à la taille finie de l'ensemble des données à traiter et plus précisément aux traitements des données situées aux "bords" (les dépendances étant locales dans les algorithmes considérés). Le partitionnement introduit évidemment des effets de bord importants, qui peuvent avoir une influence considérable sur les performances et la complexité d'une architecture. Ces effets de bord sont très variables et dépendent fortement des algorithmes. Il est parfois possible de les faire disparaître en intégrant le problème à traiter dans un problème identique mais de taille supérieure, dans lequel il suffit de supprimer les résultats obtenus sur les nouveaux bords introduits. Cette solution n'est cependant pas satisfaisante car elle augmente les besoins de mémorisation et diminue l'efficacité de l'architecture (en termes de quantité de traitements utiles).

Nous pouvons conserver le principe de régularité et de simplicité, en exploitant la flexibilité de séquencement et la structure de tore. Le réseau torique permet de rétablir certaines régularités puisqu'il supprime topologiquement la notion de bord. La notion de voisinage peut ainsi rester régulière au niveau de tous les processeurs. Par contre, le partitionnement, en particulier la structure d'adressage des données qui en résulte, implique la discrimination des processeurs associés aux données situées aux bords des partitions. Nous pouvons reporter la totalité des irrégularités au niveau du contrôle de l'architecture. Le

contrôle SPMD permet à la fois d'unifier complètement le traitement réalisé par l'ensemble des processeurs, qui exécutent tous le même programme et qui sont donc parfaitement indifférenciés au niveau de la spécification du traitement, et de disposer d'un séquencement qui s'adapte dynamiquement aux irrégularités de ce traitement. La gestion unifiée des processeurs de bord dans l'ensemble du réseau est rendue possible en indiquant à chaque processeur ses coordonnées, ce qui peut être réalisé facilement à l'initialisation du système en envoyant une matrice de valeurs au réseau, cette opération ne nécessitant pas de différenciation.

Dans le cas d'un partitionnement LPGS, seuls les processeurs de "bord", c'est-à-dire d'abscisse ou d'ordonnée *0* ou *n-1*, où *n* représente la taille du réseau et des blocs, sont associés à des données situées aux bords des partitions. On peut commencer par écrire un programme simplifié, qui correspond au point de vue d'un processeur quelconque qui n'est pas situé sur un bord, puis ajouter ensuite une ou plusieurs instructions de branchement conditionnel et une section supplémentaire pour gérer le traitement des processeurs de bord. Pour réaliser complètement un traitement par blocs, c'est-à-dire où tous les processeurs mettent à jour des valeurs associées à un même bloc de pixels à chaque étape, on peut faire en sorte que les valeurs obtenues lors de la réception d'un message par un PE correspondent, y compris pour les PE de bord, au bloc en cours de traitement. La gestion des effets de bord se fait alors uniquement lors des envois de messages. Le voisinage est alors régulier en réception, c'est-à-dire que les PE de bord n'ont pas besoin de différencier de manière particulière leurs voisins lors de la réception de messages. Lors de l'émission d'un message, chaque PE doit par contre tenir compte de sa position pour différencier certains de ses voisins et leur envoyer des valeurs appropriées. On utilise bien sûr la structure torique pour que les processeurs de bord puissent recevoir des valeurs liées à des pixels d'un bloc adjacent. Il n'est alors pas nécessaire d'utiliser un recouvrement entre blocs lors du balayage de l'image.

Prenons un exemple. Numérotons les blocs selon le balayage standard. Un pixel de coordonnées *(x,y)*, l'origine étant située dans l'angle supérieur gauche de l'image, est alors dans le bloc numéro *(y/n)*n+(x/n)*, avec des divisions entières. Lors du traitement du bloc numéro *b*, le PE "de coin" de coordonnées *(0,0)* envoie des valeurs liées à son pixel courant, de coordonnées *((b%n)*n,(b/n)*n)* dans l'image, à ses voisins SUD et EST. Par contre, il doit lire dans sa mémoire les valeurs liées à son pixel du bloc *(b+n)* pour les envoyer à son voisin NORD, qui est en fait le PE de coordonnées *(0,n-1)* et qui reçoit ainsi de son voisin SUD des valeurs liées au pixel d'ordonnée immédiatement supérieure à celle de son pixel courant. De même, ce PE de coordonnées *(0,0)* prend en compte les valeurs liées à son pixel du bloc *(b+1)* pour les envoyer à son voisin OUEST, qui est en fait le PE de coordonnées *(n-1,0)*.

Il faut en fait prendre en compte deux types d'effets de bord. Le premier type d'effet, illustré ci-dessus, est lié au partitionnement et plus précisément aux bords des blocs. Le deuxième correspond aux irrégularités initiales liées aux bords de l'image. Pour conserver la régularité en réception, il faut que les

PE de bord qui traitent un pixel de bord de l'image envoyent une valeur particulière à leurs voisins singuliers, par exemple un élément neutre s'il en existe un.

Détection de fin

L'algorithme LPGS itératif présenté se termine lorsque l'idempotence est atteinte, c'est-à-dire lorsqu'un nouveau balayage ne modifie pas le résultat. On ne connait pas a priori le nombre de balayages (itérations image) correspondant. Le critère de détection locale de fin de calcul utilisé consiste à vérifier à la fin de chaque balayage si un pixel a été mis à jour par le processeur au cours de ce balayage. Un registre utilisé comme drapeau est donc initialisé au début de chaque balayage, et modifié dès qu'une valeur a été mise à jour. Si le drapeau n'a pas changé d'état à la fin du balayage, le processeur devient inactif et se met en attente d'un éventuel message en provenance de ses voisins (deuxième type d'algorithme vis-à-vis du paragraphe 5.2.6). Quand un processeur voit au contraire que le traitement n'est pas terminé, il peut commencer une nouvelle itération sans se soucier de l'état de ses voisins et sans envoyer de message particulier, mais en envoyant simplement la valeur de son pixel courant, comme à n'importe quelle étape de l'algorithme. C'est ce message qui réveillera les voisins qui se sont éventuellement arrêtés, qui propageront eux-mêmes l'exécution de la nouvelle itération. Il suffit qu'un seul processeur envoie une valeur pour que tout le réseau effectue une nouvelle itération.

Description du programme

Le programme est donné en annexe (fichier "recons_iter.asm"). On suppose que les registres r_0 et r_1 contiennent les coordonnées du processeur et que le registre r_2 est initialisé à 0 uniquement pour les processeurs "de bord", c'est-à-dire d'abscisse ou d'ordonnée 0 ou 15. Les plans mémoire 0 et 1 sont réservés pour les entrées/sorties (nous n'avons pas écrit le transfert des données d'entrée et de sortie vers ou à partir des plans réservés aux traitements). Le plan mémoire 2 contient l'image de référence et le plan mémoire 3, initialisé avec le résultat de l'érosion, stocke la reconstruction courante. On utilise l'adressage canonique lié au partitionnement LPGS, c'est-à-dire que l'adresse *a* de la mémoire du processeur de coordonnées *(i,j)* contient le pixel de coordonnées *(i,j)* dans le bloc numéro *a* de l'image, soit le pixel de coordonnées *((a%n)*n+i,(a/n)*n+j)* dans l'image.

Le programme principal (lignes 21 à 56) est constitué de trois boucles imbriquées, correspondant aux itérations image, au balayage des blocs et aux itérations bloc. Le nombre d'itérations effectuées sur un bloc à chaque étape est spécifié ligne 23. Le "coeur" des boucles correspond exactement à l'algorithme parallèle initial, qui consiste à envoyer la valeur courante aux voisins, à recevoir les valeurs des voisins, puis à calculer la "dilatation géodésique". Les communications sont bloquantes. On peut noter une optimisation rendue possible grâce aux spécificités des instructions de communication, plus précisément dans le cas présent à la gestion du voisinage pour l'instruction de réception. Au lieu de lire les messages en spécifiant séquentiellement les différents voisins dans un ordre prédéfini, on spécifie le voisinage complet dans un registre (registre r_4, ligne 33), puis chaque réception ne tiendra compte que

des voisins non encore traités dans l'itération et supprimera un de ces voisins, correspondant au message reçu. Cela permet de n'utiliser qu'une seule instruction pour spécifier le voisinage, de relâcher des contraintes de synchronisation liées à l'ordre de traitement des voisins (qui jouent ici un rôle symétrique) et donc de faire un traitement au plus tôt, puisque la réception se termine dès qu'un message a été reçu parmi l'ensemble des voisins spécifiés et non pas d'un voisin particulier, tout en garantissant qu'un message et un seul sera lu pour chacun des voisins. Cette optimisation, même si elle n'apporte pas de gain important dans ce programme particulier (6% en vitesse et 12% en énergie tout de même), est intéressante du point de vue d'un modèle d'exécution à temps variable, puisqu'elle permet d'exploiter dynamiquement la dispersion des temps de calcul et apporte une contribution supplémentaire à l'approche "calcul en temps minimum".

Le traitement par bloc n'impose pas de coût particulier de synchronisation au niveau du changement de bloc. Il n'y a aucune synchronisation globale ou message particulier (de même qu'au niveau des itérations image). Tous les processeurs utilisant uniquement des communications bloquantes et envoyant tous le même nombre de messages, les synchronisations locales des actions de communications suffisent à respecter le séquencement de l'algorithme, quels que soient les temps d'exécution élémentaires.

La gestion des effets de bord est ici confinée dans la section située lignes 65 à 117. Comme expliqué plus haut, cette gestion se fait uniquement par modification des envois de messages par les processeurs "de bord". On initialise un registre à 0 pour discriminer uniquement les processeurs de bord et on ajoute une instruction de branchement conditionnel (ligne 30), qui teste ce registre, juste avant l'instruction d'envoi située dans la partie régulière du programme. Les processeurs de bord exécutent ainsi une partie de code supplémentaire qui se charge d'envoyer les valeurs adéquates aux voisins "singuliers", puis reprennent l'exécution de la partie régulière au niveau de l'instruction d'envoi normale (les branchements inconditionnels des lignes 103 et 117 pointant sur la ligne 31), le voisinage ayant été réduit entre temps aux seuls voisins "réguliers" restants. Un processeur de bord possède au plus deux voisins singuliers (dans le cas des processeurs "de coin"). Le test des registres r_0 et r_1 permet de distinguer les différents cas de position des processeurs. Le compteur utilisé pour le balayage des blocs dans le programme principal (registre r_{15}) est testé pour distinguer les deux types d'effet de bord. Lorsqu'il faut gérer le bord de l'image, on envoie la valeur 0, qui est élément neutre pour l'opération de maximum. Pour gérer un bord de bloc "normal", il faut calculer le numéro de bloc correspondant à la donnée attendue par le voisin singulier, lire cette donnée en mémoire et la transmettre à la place de la valeur courante (voir l'exemple donné plus haut pour le processeur de coordonnées *(0,0)*). Des opérations de masquage par un "et logique" permettent de déterminer le voisinage restant (lignes 76, 90, 102, 116).

La détection locale de fin de calcul se base sur le positionnement d'un drapeau, contenu dans le registre r_{14}. Au début de chaque itération bloc, la valeur du pixel à traiter est lue en mémoire et stockée

dans le registre r_3, puis transférée dans le registre r_6 (ligne 25), qui est ensuite mis à jour à chaque itération. A la fin du traitement du bloc, les deux registres sont comparés (ligne 49) pour savoir si la valeur du pixel a été modifiée. Si c'est le cas, la nouvelle valeur est stockée en mémoire et le drapeau est positionné. A la fin de l'itération image, le drapeau est testé (ligne 56), et le processeur devient éventuellement inactif et attend un message de la part d'un de ses voisins qui pourrait lui signaler d'exécuter une nouvelle itération (lignes 58 à 62).

Notons enfin que le balayage alterné est simplement réalisé en faisant osciller le registre r_{11} entre les deux valeurs 1 et 255, grâce à un "ou exclusif" avec la valeur 254 (ligne 55), ce registre étant utilisé comme incrément du compteur de bloc (ligne 53). On balaye ainsi les blocs 0 à 255 en ordre croissant la première fois. Si une nouvelle itération image est exécutée, on traite alors le bloc 0, puis on balaye les blocs 255 à 1 en ordre décroissant, puis le balayage est de nouveau inversé pour la suite.

Résultats de simulation

Le Tableau 6 résume les évaluations obtenues par simulation du programme donné en annexe. La valeur initiale de l'image marqueur correspond à une érosion de l'image de référence. Nous avons appliqué ici 32 érosions élémentaires (l'élément structurant étant le 4-voisinage classique). Dans les algorithmes de segmentation morphologique hiérarchique étudiés [Sale92], cette taille d'érosion est suffisamment grande par rapport à la taille des images traitées pour être utilisée dans le niveau initial de segmentation. Les niveaux suivants utilisent des tailles décroissantes d'éléments structurants, la phase de reconstruction nécessitant alors de moins en moins d'itérations.

Taille du programme	94 instructions							
Taille de l'érosion initiale	32 érosions élémentaires							
Image source	camera.pic							face.pic
Taille des buffers	1						>2	>2
Nombre max. de messages	1						2	2
Nb. ité. par balayage (n_{ite_bloc})	1	2	4	8	16	32	8	8
Nb. de balayages image	242	106	50	25	14	8	25	26
Nb. total d'itérations par pixel	242	212	200	200	224	256	200	208
Nb. ité. avec l'algo. parallèle	254							291
Nb. min. d'instr. par PE (Mega)	1,740	1,307	1,129	1,077	1,177	1,328	1,077	1,120
Nb. max. d'instr. par PE (Mega)	3,514	2,860	2,595	2,543	2,818	3,204	2,543	2,644
Nb. total d'instr. exécut. (Mega)	513,3	393,8	345,0	331,7	363,8	411,5	331,7	345,0
Temps total d'exécution (ms)	44,0	35,8	32,5	31,9	35,3	40,2	31,9	33,1
Puiss. moy. de calcul (GIPS)	11,7	11,0	10,6	10,4	10,3	10,2	10,4	10,4

Tableau 6: Evaluations du programme "recons_iter.asm"

Le paramètre principal est le nombre d'itérations appliquées sur chaque bloc lors d'une itération image (n_{ite_bloc}). Les simulations montrent qu'il existe une valeur optimale pour ce nombre d'itérations bloc vis-à-vis du nombre total d'itérations en chaque pixel (qui est le produit du nombre d'itérations bloc par le nombre de balayages image requis pour atteindre l'idempotence). Ce résultat s'explique assez simplement par le fait qu'un nombre d'itérations trop faible diminue "l'effet récursif" entre blocs (qui accélère les propagations utiles), alors qu'un nombre d'itérations trop important introduit systématiquement des mises à jour inutiles, la plupart ou la totalité des pixels d'un bloc ayant déjà "localement convergé", pénalisant en particulier la dernière itération image. Rappelons qu'un bloc de taille *n* atteint nécessairement l'idempotence au bout d'au plus *2n-1* itérations dans le cas de la dilatation géodésique. Le temps total d'exécution est principalement fonction du nombre total d'itérations, mais il augmente aussi lorsqu'on diminue le nombre d'itérations bloc, à cause des instructions qui ne sont pas situées dans le coeur des boucles imbriquées et qui sont exécutées plus de fois lorsque le nombre d'itérations image augmente (ce surcoût est visible dans le Tableau 6 en comparant les résultats obtenus avec $n_{ite\_bloc}=4$ et $n_{ite\_bloc}=8$ où le nombre total d'itérations est identique mais pas le nombre d'instructions exécutées). Bien que la valeur optimale dépende des données et de plusieurs autres facteurs, la valeur $n_{ite\_bloc}=n/2$ semble être un bon choix a priori pour obtenir un résultat quasi-optimal. Nous avons mentionné dans le Tableau 6 le nombre d'itérations nécessaires avec l'algorithme parallèle initial (non récursif), pour montrer l'accélération de la convergence due au traitement récursif au niveau bloc (on voit en particulier que, même pour $n_{ite\_bloc}=1$, le nombre total d'itérations est inférieur à celui de l'algorithme parallèle).

La valeur du temps total d'exécution (qui est surestimée pour ramener les résultats à une technologie 0.5µm) semble raisonnable. Elle est un peu supérieure au temps de traitement du circuit AMPHIN, ce qui parait normal, ce dernier étant un circuit moins pipeliné mais plus spécifique. Elle reste tout de même trop importante pour considérer cet algorithme de reconstruction comme une primitive de base d'un algorithme de segmentation à effectuer en temps réel.

La dispersion du nombre d'instructions exécutées par chacun des processeurs correspond uniquement aux effets de bord (à quelques instructions près liées à la détection de fin). Les processeurs de bord exécutent ainsi environ 2 à 2,5 fois plus d'instructions que les processeurs "de coeur". Ce sont les processeurs de bord et donc les effets de bord, qui déterminent la vitesse de traitement dans tout le réseau, puisqu'on utilise des communications bloquantes. La puissance moyenne de calcul, environ égale à la moitié de la puissance crête, correspond directement à la pondération des nombres d'instructions exécutées par les deux types de processeurs (de bord et de coeur), divisée par le temps de calcul imposé par les processeurs les plus lents. Elle augmente quand n_{ite_bloc} diminue parce que le surcoût lié aux instructions qui ne sont pas au coeur des boucles est plus faible en proportion pour les PE de bord, qui

fixent le temps d'exécution (ce dernier augmente donc moins que le nombre total d'instructions exécutées dans le réseau).

L'avant-dernière colonne du Tableau 6 montre qu'il n'y a jamais plus de 2 messages présents simultanément dans une interface au cours de l'exécution. Cela est imposé par les synchronisations de l'algorithme. Il est en effet impossible qu'un processeur ait plus d'un envoi d'avance sur un de ses voisins (ce qui n'empêche pas qu'il y ait plusieurs itérations d'écart entre des processeurs distants à un instant donné), car dans ce programme chaque PE consomme un message dans une interface de réception avant d'envoyer un nouveau message dans l'interface d'envoi correspondant au même voisin. On peut noter que l'utilisation d'une taille de buffers égale à 2 n'a aucun effet perceptible sur le temps total d'exécution, qui est toujours fixé par les processeurs de bord.

La dernière colonne donne les résultats obtenus avec une image différente et montre simplement que, bien que le nombre d'itérations dépende des données, l'ordre de grandeur du temps d'exécution reste identique.

L'activité logique temporelle du réseau correspondant à la simulation effectuée avec $n_{ite\_bloc}=8$ est représentée Figure 37. Comme expliqué plus haut, l'algorithme correspond au deuxième type de détection de fin considéré au paragraphe 5.2.6. Cependant, la variation de l'activité est particulière dans ce programme, puisque les processeurs ne décident éventuellement de s'arrêter qu'à la fin d'une itération image, et redémarrent immédiatement après si au moins un des processeurs a décidé de continuer. Elle ne peut donc se produire qu'autour d'intervalles réguliers et avec une dispersion relative très faible, et ressemble donc à une variable discrète à l'échelle du traitement complet. On voit ici que quelques processeurs n'ont pas mis à jour de pixels aux itérations 21, 22 et 23, qu'il en a été de même pour plus d'un tiers des processeurs à l'itération 24, et que tous les processeurs ont finalement terminé à l'itération 25.

Ce premier programme de reconstruction, bien qu'utilisant certaines spécificités de l'architecture, est néanmoins assez classique du point de vue du séquencement utilisé, correspondant à un traitement parallèle itératif par blocs. Passons maintenant à la mise en oeuvre du même problème avec des séquencements originaux utilisant les modes de contrôle SPMD, cellulaire, associatif, flot de données et basés sur les communications non-bloquantes.

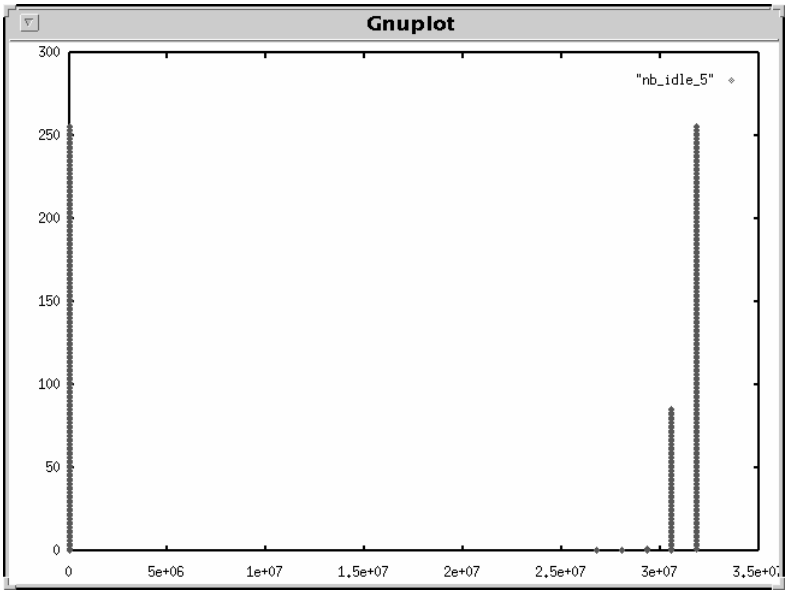

Figure 37: Activité logique du réseau

b) Reconstruction LSGP flot de données

Algorithme séquentiel par propagation

Il existe des algorithmes beaucoup plus efficaces que l'algorithme parallèle itératif pour la mise en oeuvre de la reconstruction sur des architectures séquentielles [Sale96] [Vinc93]. L'idée est d'essayer de ne considérer que les pixels qui doivent être modifiés au cours de la reconstruction, plus précisément de parcourir les pixels dans un ordre correspondant aux propagations des valeurs dans l'image, plutôt que d'effectuer systématiquement des itérations sur tous les pixels, ce qui engendre beaucoup de calculs inutiles. Pour cela, on commence par déterminer un ensemble minimum de pixels "initiateurs" du processus de reconstruction, qui vérifient les conditions locales impliquant une mise à jour de certains de leurs pixels voisins. A partir de ces pixels, l'information utile est propagée de proche en proche dans les régions qui doivent être reconstruites. On essaye ainsi de minimiser les dépendances fonctionnelles en envisageant l'algorithme sous une forme flot de données.

Pour déterminer les pixels initiateurs, on rejette les pixels dont on sait qu'ils seront mis à jour par propagation de la valeur d'un de leurs voisins. Dans le cas de la reconstruction positive, on essaye d'abord de ne conserver que les maxima de l'image marqueur qui ne peuvent être modifiés au cours de la reconstruction. Ce sont en fait les maxima régionaux, définis de la manière suivante [Vinc93]: un maximum régional *M* d'une image en niveaux de gris est une composante connexe de pixels de même valeur *h* (un plateau d'altitude *h*) telle que tous les pixels du voisinage de *M* aient une valeur strictement inférieure. On peut ensuite ne conserver que les bords des maxima régionaux (Figure 38). Cependant la détermination des maxima régionaux possède un coût non négligeable puisque ce n'est pas un problème local (elle peut entre autres s'exprimer elle-même sous la forme d'une reconstruction!). Nous proposons

d'accélérer la phase d'initialisation en utilisant un sur-ensemble de ces pixels initiateurs, déterminés de façon purement locale: les maxima (stricts) locaux, c'est-à-dire les pixels qui sont à la fois égaux au maximum et strictement supérieurs au minimum des valeurs du 4-voisinage (Figure 38). Nous comparerons dans la suite les performances de la reconstruction à partir des deux types d'initialisation.

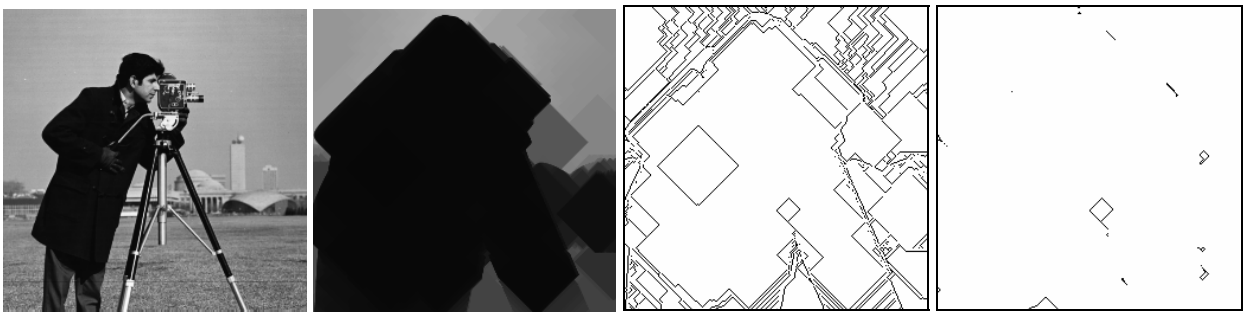

Figure 38: Exemple d'initialisation pour la reconstruction flot de données. De gauche à doite: image source, image marqueur (érosion de taille 32), maxima stricts locaux, et bords des maxima régionaux.

La reconstruction se fait ensuite par propagation, ce qui nécessite de mémoriser les positions des pixels susceptibles de propager leur valeur à leurs voisins. Les algorithmes séquentiels utilisent pour cela une structure de queue de type FIFO. L'algorithme séquentiel de reconstruction consiste alors à:

- initialiser la queue avec les pixels de bord des maxima régionaux de l'image marqueur,
- puis tant que la queue n'est pas vide, répéter les opérations suivantes:
- retirer le premier pixel *x* de la queue,
- pour chacun de ses voisins *y* (dans l'image marqueur) dont la valeur est strictement inférieure à la sienne, calculer le minimum entre la valeur de *x* et la valeur du pixel de l'image de référence de même position que *y*,
- si ce minimum est strictement inférieur à la valeur de *y*, mettre celle-ci à jour et ajouter *y* dans la queue de pixels.

Partitionnement et séquencement

Nous considérons ici un partitionnement de type LSGP et un séquencement flot de données. On découpe l'image en blocs de 16*16 pixels, et chaque bloc est associé à un processeur du réseau. La mémoire du processeur de coordonnées *(x,y)* contient le bloc numéro *(x+16*y)*, soit les pixels de coordonnées *(16*x+p,16*y+q)* dans l'image, avec $p,q \in [0,15]$. Le pixel de coordonnées *(p,q)* dans le bloc est mémorisé à l'adresse *(p+16*q)*. L'algorithme séquentiel par propagation est appliqué par chaque processeur à chaque bloc de l'image. Notons qu'un partitionnement par blocs de l'algorithme de reconstruction a été présenté dans [Laur97] pour sa mise en oeuvre sur le processeur MVP. Nous n'utiliserons pas de structure mémorisant les adresses de la suite de pixels à traiter, mais nous nous servirons directement des fonctionnalités des mémoires associatives. Il suffit en effet d'utiliser un bit

associé à chaque pixel, pour marquer les pixels à propager. Une instruction de lecture associative peut choisir un des pixels marqués et renvoyer la valeur et l'adresse de ce pixel. Ceci constitue un relâchement des contraintes de séquencement de l'algorithme initial. En effet, la structure FIFO de la queue de pixels impose un ordre particulier de traitement, alors que cela n'est pas nécessaire pour la reconstruction, tous les pixels devant propager une valeur pouvant être traités dans un ordre arbitraire.

Avec un partitionnement de type LSGP, tous les processeurs sont associés à des données situées aux bords des partitions et doivent donc gérer les effets de bord. Comme dans le programme précédent, cette gestion des irrégularités peut être confinée au niveau des envois de messages entre PE, de manière à ce que les réceptions de messages soient régulières, c'est-à-dire identiques pour tous les PE et indépendantes du voisin d'origine. Contrairement à l'algorithme LPGS itératif par blocs, où tous les processeurs s'échangeaient des pixels de même adresse locale à une étape donnée, les pixels sont ici mis à jour dans un ordre qui dépend des données locales. Pour que les processeurs ne s'échangent que les pixels mis à jour, il faut envoyer les coordonnées du pixel à propager en plus de sa valeur, lorsqu'un pixel de bord de bloc doit être propagé dans un bloc voisin (d'où le choix d'une taille de deux registres pour les messages). Pour réaliser les propagations, qu'elles soient inter-blocs ou intra-blocs, on va en fait calculer l'adresse du pixel "destination". Il suffit d'effectuer un calcul particulier lorsqu'un pixel doit être transmis à un processeur voisin. Par exemple, le pixel de coordonnées locales *(15,q)* sera propagé vers le processeur correspondant à la direction EST, en envoyant un message contenant la valeur du pixel et l'adresse *0+16*q*, correspondant aux coordonnées *(0,q)*. Pour gérer les bords de l'image, il suffit de détecter les cas où un PE "de bord" (du réseau) essaye d'envoyer un pixel du même bord (de bloc), et de ne pas envoyer de message. A la réception d'un message, il suffit de traiter le pixel dont l'adresse est donnée dans le message, et de marquer ce pixel s'il a été mis à jour.

L'algorithme proposé est basé sur l'utilisation de communications non-bloquantes. Il ne comporte aucune contrainte sur les instants de communication, ni sur le nombre de communications entre PE. Les communications non-bloquantes sont nécessaires pour éviter des interblocages, sachant qu'il n'y a aucune synchronisation globale. Un premier principe de l'algorithme est d'effectuer les propagations entre PE au plus tôt (compte tenu de la taille finie des buffers d'interfaces), tout en essayant de maintenir l'activité des PE au maximum, ceci en minimisant les synchronisations. Si un PE essaye de propager un pixel vers un processeur voisin mais que la communication n'est pas effective, il suffit que le pixel reste marqué pour que la propagation puisse être réalisée ultérieurement, et le processeur peut effectuer un autre traitement, sans attendre qu'une place soit libérée en émission dans l'interface. Pour minimiser les communications non effectives, un deuxième principe consiste à traiter en priorité les réceptions de message, par rapport aux autres propagations. A la fin de chaque propagation, on effectue donc une lecture non-bloquante sur le 4-voisinage. Si un message a été reçu, on traite le pixel correspondant et s'il vérifie les conditions de mise à jour, il est modifié et marqué. Au lieu de propager ce pixel

immédiatement, on effectue une nouvelle lecture. C'est seulement lorsqu'il n'y a plus de message à traiter que la phase de propagation recommence et que l'on effectue une recherche associative d'un pixel marqué.

La mise à jour d'un pixel qui suit la réception d'un message introduit une forme d'asynchronisme fonctionnel car on ne teste pas si le pixel était déjà marqué avant de le mettre à jour. Cela signifie que l'on peut supprimer une valeur intermédiaire du pixel, qui aurait pu donner lieu à une série de propagations, pour la remplacer par une valeur plus proche de la solution finale, ce qui minimise les propagations intermédiaires et accélère les propagations "utiles". L'asynchronisme fonctionnel consiste ainsi à supprimer dynamiquement certaines dépendances, de manière indéterministe puisque cela dépend des données, des temps d'exécution élémentaires et de l'ordre de traitement des pixels (qui dépend d'une opération de recherche associative, a priori indépendante de l'adressage).

Nous avons enfin optimisé la gestion des propagations restantes en mémorisant pour chaque pixel les directions de propagation utiles. Il est en effet inutile d'essayer de propager systématiquement un pixel marqué vers la totalité de son 4-voisinage. D'abord, lorsqu'un pixel de bord n'a pu être propagé vers le(s) processeur(s) voisin(s) correspondant(s), il a tout de même été propagé vers ses pixels voisins appartenant au même bloc et il serait donc inutile d'effectuer de nouveau ces propagations. De plus, lorsqu'un pixel est propagé vers un de ses voisins et qu'il provoque une mise à jour, on sait que la nouvelle valeur de ce voisin ne peut pas en retour provoquer une mise à jour du premier pixel, puisque cette valeur ne peut pas être supérieure à la première. En dehors des pixels initiateurs, il suffit donc de propager chaque pixel vers au plus trois de ses voisins, c'est à dire au 4-voisinage diminué du voisin qui a provoqué la dernière mise à jour du pixel. En résumé, à chaque propagation réalisée, on supprime la direction correspondante pour le pixel "source", et si le pixel "destination" est mis à jour, on remplace son voisinage restant par le 3-voisinage correspondant à la propagation.

La détection locale de fin de calcul correspond à la situation où il ne reste plus de message à traiter ni de pixel à propager (cela correspond au deuxième type d'algorithme au sens du paragraphe 5.2.6, puisque le nombre de messages échangés entre processeurs n'est pas connu). Le processeur devient alors inactif et se met en attente d'un message éventuel d'un processeur voisin. Ce message provoquera le retour à l'état actif, la mise à jour éventuelle d'un pixel de bord et une nouvelle "vague" de propagations le cas échéant.

Concluons ce paragraphe en notant que l'algorithme est bien basé sur un contrôle mixte: SPMD (tous les processeurs exécutant le même programme), cellulaire (un PE envoyant un message vers un voisin pouvant influencer le séquencement de celui-ci), associatif (le séquencement étant basé sur une recherche associative parmi un ensemble de données marquées), flot de données (le séquencement étant déterminé à partir de la résolution des dépendances entre données), à asynchronisme fonctionnel (les dépendances étant partiellement indéterministes et pouvant être dynamiquement modifiées).

Description du programme

Le programme est donné en annexe (fichier "recons_bloc.asm"). Il s'agit uniquement de l'algorithme de reconstruction, ce qui n'inclut pas la détermination des pixels initiateurs.

On suppose que les registres r_0 et r_1 contiennent les coordonnées du processeur. Le plan mémoire 2 contient l'image de référence et le plan mémoire 3, initialisé avec le résultat de l'érosion, stocke la reconstruction courante. Le plan mémoire 5 sert au marquage des pixels à propager, la valeur associée à chaque pixel (dénommée "tag") étant supposée initialisée à 1 pour les pixels initiateurs. Le plan mémoire 4 sert à stocker les directions restantes pour les pixels à propager (les tags et les directions auraient pu être mémorisés ensemble dans le plan mémoire 4). Les directions sont initialisées avec le 4-voisinage complet pour les pixels initiateurs, le format utilisé étant le même que celui des instructions de communication (un bit par voisin).

L'algorithme est le suivant:

- chercher un pixel marqué à propager (lignes 25 et 120) et lire la valeur de l'image marqueur et les directions de propagation associées (lignes 32 et 33) (s'il n'y a pas de pixel marqué, devenir inactif jusqu'à réception d'un message)
- pour chacune des directions de propagation restantes associées à ce pixel source (les quatre sections correspondantes sont situées lignes 35 à 60, 62 à 90, 92 à 118, 129 à 157):
  - s'il s'agit d'un pixel du bord de bloc correspondant à la direction de propagation, calculer l'adresse du pixel destination dans le bloc voisin, essayer d'envoyer un message contenant cette adresse et la valeur de l'image marqueur à propager, et si la communication a été effective, supprimer la direction parmi les directions restantes (s'il s'agit d'un pixel de bord de l'image, aucun envoi n'est effectué et la direction est supprimée)
  - dans le cas "normal" où la propagation ne sort pas du bloc, déterminer l'adresse du pixel destination, lire en mémoire la valeur courante de l'image marqueur en ce pixel ainsi que la valeur de l'image de référence, tester les conditions de propagation (la valeur de l'image marqueur du pixel destination doit être strictement inférieure à celle du pixel à propager et à la valeur de l'image de référence), et si le test est positif, mettre à jour l'image marqueur, ainsi que le tag et les directions restantes associés à ce pixel destination (pour que ce pixel puisse à son tour être propagé vers le 3-voisinage restant), enfin supprimer la direction traitée quel que soit le résultat du test
- stocker les directions restantes et effacer le tag du pixel source si toutes les propagations ont été effectuées (lignes 159 à 162)
- essayer de recevoir un message (lignes 163 à 166)
  - s'il n'y a pas de message, chercher un nouveau pixel à propager

- si un message a pu être reçu, lire en mémoire les valeurs de l'image marqueur et de l'image de référence du pixel dont l'adresse est donnée dans le message, tester les conditions de propagation (la valeur de l'image marqueur du pixel source provenant du reste du message), et si le test est positif, mettre à jour l'image marqueur et passer au stockage des directions restantes puis à la réception d'un autre message, sinon passer directement à la réception d'un autre message (lignes 167 à 175).

Résultats de simulation

Le Tableau 7 résume les évaluations obtenues par simulation du programme donné en annexe. Comme pour l'évaluation du programme précédent, l'image marqueur a pour valeur initiale une érosion de taille 32 de l'image de référence. Nous avons observé les effets du choix de la détermination des pixels initiateurs, ainsi que de la taille des buffers d'interface, qui peut jouer un rôle a priori assez important lorsque l'on utilise des communications non-bloquantes pour maximiser l'activité des processeurs.

Taille du programme	150 instructions					
Taille de l'érosion initiale	32 érosions élémentaires					
Image source	camera.pic			face.pic		
Détermination des pixels initiateurs	max. locaux		région.	max. locaux		région.
Taille des buffers des interfaces	1	>2	>2	1	>2	1
Nombre max. de messages dans un buffer	1	2	2	1	2	1
Nb. min. d'instructions exécutées par les PE (kilo)	16,4	16,4	16,4	16,6	16,6	16,5
Nb. max. d'instructions exécutées par les PE (kilo)	147,9	149,0	155,1	245,5	246,6	168,9
Nb. total d'instr. exécutées par le réseau (Mega)	15,1	15,0	13,2	17,7	17,7	14,7
Temps total d'exécution estimé (ms)	1,963	1,960	2,027	3,069	3,083	2,439
Puissance moyenne de calcul (GIPS)	7,7	7,7	6,5	5,8	5,7	6,0

Tableau 7: Evaluations du programme "recons_bloc.asm"

Notons tout d'abord que sur les images testées, le temps d'exécution total est de l'ordre de 2 à 3 millisecondes, c'est-à-dire que cet algorithme flot de données est de 10 à 15 fois plus rapide que l'algorithme itératif précédent, pour une énergie consommée 20 fois plus faible (ce qui va dans le même sens que pour la mise en oeuvre sur une architecture séquentielle). Cependant la puissance moyenne de calcul est assez faible, alors que le programme utilise des communications non-bloquantes. Cela est entièrement dû à l'utilisation du partitionnement LSGP sur un algorithme flot de données dont les dépendances sont spatialement réparties de façon très inhomogène et variables au cours de l'exécution, ce qui cause un mauvais équilibrage de charge ("load balancing"). On peut le voir directement soit à partir

des nombres minimum et maximum d'instructions exécutées par les différents PE, qui diffèrent d'un ordre de grandeur alors qu'il ne peut y avoir de dispersion notable due aux effets de bord, soit à partir de la Figure 39 qui représente l'activité logique du réseau et qui montre que la plupart des processeurs deviennent vite inactifs au cours de l'exécution. Le partitionnement LSGP peut donc limiter fortement le taux de parallélisme effectif, à cause de la structure régionale des dépendances présentes dans le problème de la reconstruction. On peut noter que les performances de l'algorithme varient assez fortement en fonction des données, en comparant les résultats obtenus sur les deux images testées.

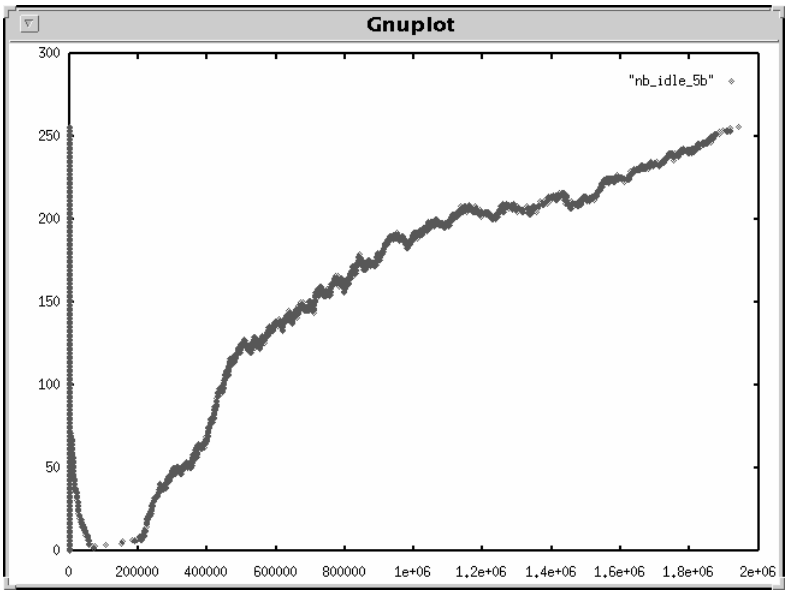

Figure 39: Activité logique du réseau

Il peut être un peu surprenant a priori de voir que la taille des buffers d'interface a une influence négligeable sur les performances de l'exécution, et qu'il n'y a jamais plus de deux messages simultanément présents dans une interface, alors que les instants de communication et le nombre de communications peuvent fortement varier. Cela est dû à la gestion prioritaire des réceptions de messages par rapport aux autres propagations, au fait que le nombre relatif de propagations pouvant générer un message est faible à cause du partitionnement, et enfin au fait que dans cet algorithme l'envoi successif de deux messages à une même interface nécessite plus d'instructions et de temps que la prise en compte de la réception d'un message et sa consommation par un PE voisin.

Quant à la détermination des pixels initiateurs, bien que son temps d'exécution n'ait pas été évalué par simulation, les résultats montrent que l'utilisation des maxima locaux (a priori non optimale) semble être préférable au calcul des (bords des) maxima régionaux, puisque cela ne semble pas pénaliser beaucoup la phase de reconstruction proprement dite, tout en étant bien plus simple et rapide à réaliser. Pour l'image "camera", l'utilisation des maxima régionaux ralentit en fait la reconstruction, parce que les propagations utiles atteignent moins vite les régions concernées à cause du très faible nombre de pixels

initiateurs (cf Figure 38). Pour l'autre image, elle permet au contraire d'éviter tout de même un bon nombre de propagations inutiles.

c) Reconstruction entrelacée flot de données

Passons enfin à la description de l'algorithme de reconstruction le plus performant parmi ceux présentés dans cette section. Il reprend les principes de l'algorithme flot de données précédent, en utilisant une répartition duale des pixels et un séquencement qui entrelace des propagations réparties de manière quelconque dans l'image.

Partitionnement et séquencement

Considérons une répartition des pixels dans le réseau correspondant à un partitionnement LPGS, comme dans l'algorithme itératif par blocs. Les pixels associés à un même PE sont répartis dans toute l'image, et lorsqu'un PE traite un pixel, tout pixel voisin dans l'image est associé à un processeur voisin. Le pixel de coordonnées *(i,j)* dans le bloc numéro *b* de l'image est stocké à l'adresse *b* dans la mémoire du processeur de coordonnées *(i,j)* dans le réseau.

On peut facilement adapter l'algorithme flot de données précédent à cette répartition. Ainsi, une lecture associative permet d'abord de choisir un des pixels marqués. Pour chacune des directions restantes associées au pixel, il faut aussi essayer de propager la valeur de l'image marqueur vers le pixel voisin correspondant. Avec une "répartition LPGS", ce pixel voisin est toujours associé à un processeur voisin. Toute propagation se fait donc par un envoi de message. Comme il faut indiquer au processeur voisin de quel pixel il s'agit, on inclut dans le message l'adresse du pixel destination. Il se trouve que pour la plupart des pixels, l'adresse d'un pixel voisin dans la mémoire d'un processeur voisin est identique (puisqu'elle est égale au numéro du bloc de l'image dans lequel se trouve le pixel). Seuls les pixels de bord de bloc peuvent avoir un pixel voisin d'adresse différente. Pour les autres pixels, les propagations vers les directions restantes peuvent être effectuées avec une seule instruction d'envoi, puisque le même message peut être utilisé et qu'il suffit d'utiliser le registre contenant les directions restantes comme voisinage pour l'envoi. A la fin de l'instruction d'envoi, ce registre contient les nouvelles directions restantes, c'est-à-dire celles pour lesquelles la communication n'a pas été effective, et il suffit de stocker cette valeur en mémoire pour mettre à jour les directions de propagation. De même, pour recevoir un pixel, on spécifie un 4-voisinage complet pour l'instruction de réception, et si un message a été reçu, le registre de voisinage contient précisément le 3-voisinage qui doit être stocké comme directions de propagation restantes. Comme dans l'algorithme précédent, les communications sont non-bloquantes et les réceptions de message sont traitées en priorité, ce qui est d'autant plus important ici que toutes les propagations de pixels se font par un envoi de messages. L'algorithme présente le même asynchronisme fonctionnel, qui permet l'accélération des propagations utiles.

La notion de partitionnement laisse ici sa place à celle d'entrelacement du point de vue du séquencement, et ne correspond plus qu'à la répartition des données dans la structure parallèle. En effet, la notion habituelle de partitionnement LPGS comprend à la fois le découpage de l'image en blocs et le fait que tous les processeurs vont traiter les pixels d'un même bloc à une même étape de l'algorithme. Dans cet algorithme flot de données, le séquencement, non pas du point de vue d'un processeur mais du point de vue des pixels traités par le réseau à une étape donnée, est plutôt du type LSGP, c'est-à-dire que le réseau a la possibilité de traiter en parallèle des pixels répartis sur toute l'image. En effet, il n'y a aucune contrainte sur les numéros de bloc des pixels traités par les différents PE. Le terme "entrelacement" signifie simplement que chaque PE peut traiter des pixels situés dans n'importe quel bloc, dans un ordre quelconque et indépendamment de ses voisins.

Les effets de bord sont dus au traitement des pixels de bord de bloc, qui sont associés uniquement à des processeurs du "bord" du réseau. Comme dans le premier algorithme LPGS, il suffit de détourner l'exécution de l'algorithme vers une section particulière, lorsqu'il s'agit d'un processeur de bord. Cette section va traiter les propagations singulières (dont la direction correspond au bord auquel appartient le PE) en calculant l'adresse du pixel destination pour tenir compte du changement de bloc, éliminer les messages pour les pixels de bord de l'image, et envoyer un message normal vers les autres directions restantes. Après que les directions de propagation aient été supprimées pour les communications qui ont été effectives, l'exécution se poursuit avec le traitement normal. Le calcul de l'adresse destination consiste par exemple à remplacer le bloc b par le bloc $b+1$ pour la direction EST ou $b+n$ pour la direction SUD.

Description du programme

Le programme est donné en annexe (fichier "recons_new.asm"). Il est relativement simple puisque le traitement est régulier pour la plupart des processeurs. Les initialisations sont identiques à celles du programme précédent. Le nouvel algorithme consiste à:

- chercher un pixel marqué à propager (lignes 26 et 54) et lire la valeur de l'image marqueur et les directions de propagation associées (lignes 33 et 34) (s'il n'y a pas de pixel marqué, devenir inactif jusqu'à réception d'un message)
- si ce n'est pas un PE de bord, envoyer un message vers le voisinage égal aux directions restantes et contenant la valeur et l'adresse du pixel (ligne 36), sinon sauter à la section traitant les effets de bord (nous ne détaillons pas plus cette section, située lignes 65 à 128, dont la fonction est déja décrite plus haut)
- stocker les directions restantes et effacer le tag si toutes les propagations ont été effectuées (lignes 37 à 40)
- essayer de recevoir un message (lignes 42 à 44)
  - s'il n'y a pas de message, chercher un nouveau pixel à propager

- si un message a été reçu, lire en mémoire les valeurs de l'image marqueur et de l'image de référence du pixel dont l'adresse est donnée dans le message, tester les conditions de propagation, et si le test est positif, mettre à jour l'image marqueur et passer au stockage des directions restantes puis à la réception d'un autre message, sinon passer directement à la réception d'un autre message (lignes 45 à 53).

Résultats de simulation

Le Tableau 8 résume les évaluations obtenues par simulation du programme donné en annexe. Nous avons effectué des simulations équivalentes à celles du programme précédent. Le temps total d'exécution est de l'ordre de 0,6 à 0,8 milliseconde, soit 3 à 4 fois moins que pour le programme précédent, pour une diminution d'énergie consommée (correspondant au nombre total d'instructions exécutées) de 10 à 15%. La puissance moyenne de calcul est cette fois beaucoup plus élevée et proche de la puissance crête, ce que l'on peut aussi observer à partir de la Figure 40, qui représente l'activité du réseau. On peut constater que presque tout le réseau est actif durant toute l'exécution. Il y a ainsi un bon équilibrage de charge, qui peut aussi être remarqué à partir de la faible dispersion des nombres d'instructions exécutées, par rapport à l'algorithme précédent. La répartition LPGS et l'entrelacement permettent une parallélisation efficace d'un problème dont les dépendances peuvent être localement irrégulières et globalement inhomogènes.

Taille du programme	105 instructions					
Taille de l'érosion initiale	32 érosions élémentaires					
Image source	camera.pic				face.pic	
Détermination des pixels initiateurs	max. locaux			région.	locaux	région.
Taille des buffers	1	2	>127	1	1	1
Nombre max. de messages	1	2	127	1	1	1
Nb. min. d'instructions par PE (kilo)	44,2	49,5	59,3	36,1	53,8	39,6
Nb. max. d'instructions par PE (kilo)	55,9	62,6	78,7	47,8	64,5	50,9
Nb. total d'instructions exécutées (Mega)	13,3	14,7	17,4	11,1	15,6	12,0
Temps total d'exécution (ms)	0,698	0,784	0,984	0,602	0,807	0,637
Puissance moyenne de calcul (GIPS)	19,0	18,7	17,7	18,4	19,4	18,8

Tableau 8: Evaluations du programme "recons_new.asm"

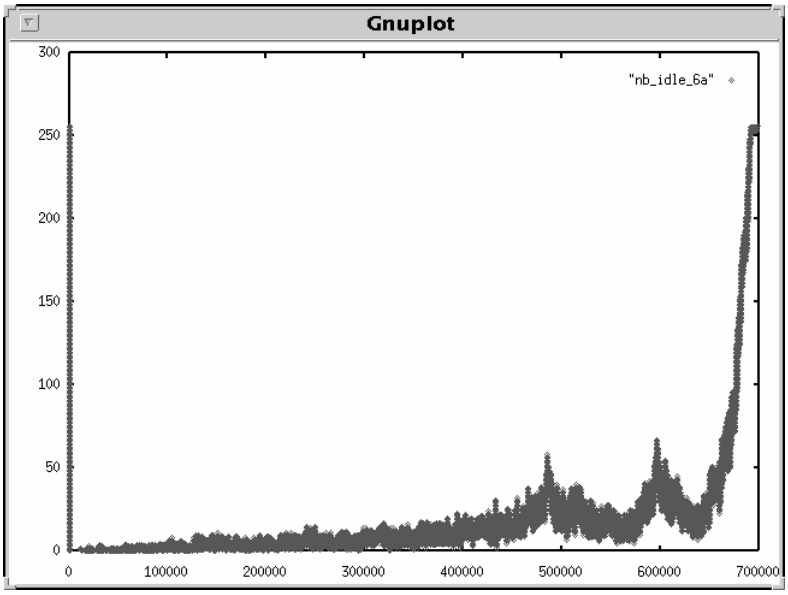

Figure 40: Activité logique du réseau

L'utilisation des maxima régionaux comme pixels initiateurs permet effectivement d'optimiser le temps de reconstruction, mais comme pour l'algorithme précédent, il faudrait évaluer le surcoût de ce choix par rapport à l'utilisation des maxima locaux.

Le comportement de l'algorithme vis-à-vis de la taille des buffers d'interface est par contre très différent de celui des autres algorithmes et assez surprenant à première vue. En effet, on observe que plus la taille des buffers est grande, plus le temps de calcul et le nombre d'instructions exécutées sont importants, alors que cela devrait minimiser le nombre de communications non effectives et de tentatives de propagations de pixels entre PE. De plus, on constate qu'un nombre très important de messages peuvent être stockés simultanément dans une interface, si on utilise une taille de buffer "infinie" (127 pendant une des simulations avec l'image "camera"), ce qui signifie qu'avec une taille de buffer réduite, il peut y avoir localement beaucoup de communications non effectives. Cela est dû au fait que chaque message reçu par un PE va potentiellement générer trois nouveaux messages dans le réseau, et que les PE de bord gèrent les réceptions de messages moins rapidement que les autres. L'effet a priori paradoxal de la taille des buffers permet en fait d'observer directement les bénéfices de l'asynchronisme fonctionnel. Cela signifie que la limitation des capacités de communication permet de minimiser le nombre de propagations intermédiaires (qui a une "tendance exponentielle") et de favoriser les propagations "utiles", qui de toute façon viendraient "écraser" le résultat des propagations intermédiaires. Il est plus pénalisant d'effectuer toutes les propagations (i.e. à chaque mise à jour d'une valeur) que de perdre du temps à cause des communications non effectives, parce que l'envoi "au plus tard" et la réception "au plus tôt" permettent d'augmenter la probabilité que les données soient de nouveau mises à jour avant d'être propagées, faisant ainsi disparaître des valeurs intermédiaires qui auraient généré de nombreuses propagations inutiles. Notons que ce phénomène n'existerait pas si l'algorithme était capable de ne

générer qu'un seul message par pixel (ce qui n'est pas possible, même avec un ensemble minimum de pixels initiateurs, puisque cela reviendrait à déterminer un ensemble minimum de chemins de propagation, soit encore plus d'information que la reconstruction elle-même).

Notons enfin que le meilleur équilibrage de charge de cet algorithme semble diminuer la dispersion des résultats obtenus avec des images différentes, c'est-à-dire la dépendance des performances vis-à-vis des données.

5.4.4 Rotation rapide d'objets de forme quelconque

Nous souhaitons présenter un dernier algorithme, pour illustrer l'utilisation du modèle architectural dans le contexte du compositage d'objets, et donc du rendu d'images, alors que le filtrage morphologique sert principalement à la segmentation, utilisée pour l'analyse. Nous présentons un algorithme de rotation d'objets, qui est une primitive de base du compositage, et qui pourtant nécessite une quantité non négligeable de traitements.

L'algorithme de rotation rapide

L'algorithme de rotation rapide est basé sur la décomposition d'une rotation en une série de translations horizontales et verticales [Unse95]. Une matrice de rotation peut en effet être factorisée de la manière suivante:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} 1 & -\tan\theta/2 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ \sin\theta & 1 \end{bmatrix} \times \begin{bmatrix} 1 & -\tan\theta/2 \\ 0 & 1 \end{bmatrix}.$$

Une rotation peut donc être effectuée en trois passes, chaque passe ne comportant que des translations de lignes ou colonnes de pixels. Le premier terme correspond en effet à une translation de chacune des lignes d'une quantité proportionnelle à l'ordonnée de la ligne. La deuxième passe consiste en une translation de chaque colonne d'une quantité proportionnelle à l'abscisse de la colonne et la troisième passe est identique à la première. Les coefficients ne sont bien sûr pas entiers dans le cas général, mais nous allons nous contenter d'un résultat arrondi et n'effectuer que des translations entières de pixels (des algorithmes basés sur des convolutions permettant d'améliorer la qualité du résultat [Unse95]). La Figure 41 illustre une telle rotation en trois passes basée uniquement sur des translations entières.

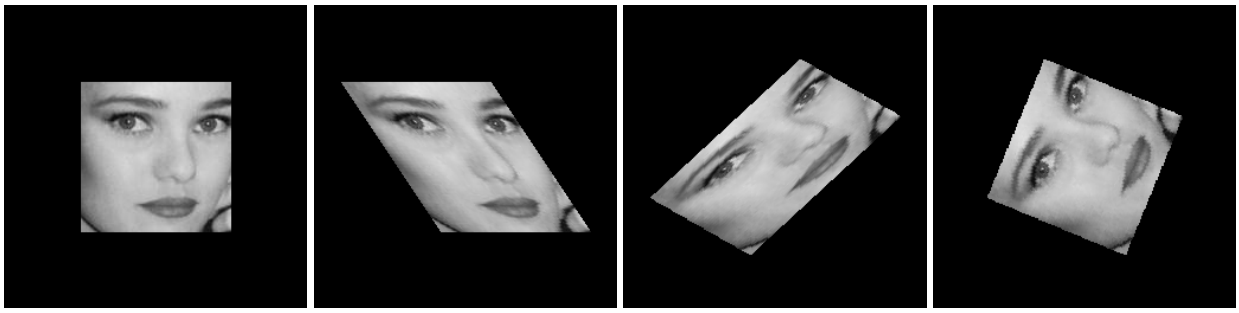

Figure 41: Exemple de rotation rapide en trois passes avec l'image "face"

Partitionnement et séquencement

Considérons une répartition de type LPGS des pixels en mémoire. Puisque chaque passe ne fait intervenir que des translations de lignes (colonnes) de pixels, les lignes (colonnes) de processeurs du réseau peuvent effectuer ces translations de manière totalement indépendante et donc non synchronisée. Par contre, tous les processeurs appartenant à la même ligne (colonne) vont utiliser entre eux des communications bloquantes pour faire circuler les pixels. Les pixels associés à un processeur appartiennent tous à un ensemble de 16 lignes et 16 colonnes de l'image.

Chaque passe peut être décomposée en deux étapes. La première étape consiste, pour chaque pixel devant être translaté, à calculer la position finale du pixel, et à le translater partiellement en le déplaçant dans son bloc destination, sans modifier sa position dans le bloc, ce que chaque PE peut réaliser localement et indépendamment des autres PE (puisqu'il suffit de stocker le pixel à l'adresse correspondant au bloc). La deuxième étape consiste à effectuer la translation restante à l'intérieur du bloc destination, en faisant circuler les pixels sur la ligne (colonne) jusqu'au processeur pouvant stocker le pixel à sa position finale. Il suffit de remarquer que la deuxième étape doit en fait effectuer une permutation circulaire de chaque segment correspondant à une ligne (colonne) de pixels d'un même bloc, ce qui est facilement réalisé par une suite de décalages entre PE en utilisant la structure de tore (en modifiant la direction de décalage, on peut limiter le nombre de décalages à la moitié de la taille du bloc).

L'algorithme peut être adapté à la manipulation d'objets de forme quelconque, en utilisant un séquencement basé sur la recherche associative des seuls pixels appartenant au masque binaire de l'objet. Les performances de l'algorithme dépendent alors de la morphologie de l'objet. Pour éviter des irrégularités inutiles, la deuxième étape de l'algorithme effectue systématiquement des permutations circulaires sur l'ensemble des 16 pixels d'un segment, dès que le segment correspondant contient au moins un pixel de l'objet, ce qui correspond à augmenter légèrement la granularité du traitement associatif. Chaque ligne (colonne) de processeurs doit réaliser la translation de pixels situés sur 16 lignes (colonnes) de l'image, mais comme le séquencement associatif n'impose pas d'ordre de traitement et que tous les pixels d'un segment n'appartiennent pas nécessairement à l'objet, on va définir un séquencement en donnant le contrôle de toute la ligne (colonne) de PE à un PE particulier, et en propageant ce contrôle séquentiellement aux différents processeurs. Au début de la deuxième étape de la première passe, le processeur situé à l'origine de la ligne devient ainsi "maître" de la ligne. Par une recherche associative d'un pixel de l'objet, il détermine un segment à permuter et un message contenant le numéro de bloc correspondant est propagé à tous les autres PE, qui peuvent alors participer à la série de décalages des pixels du segment. Lorsque le PE maître a traité tous les pixels "source" qui lui sont associés, il envoie un message particulier à son voisin pour qu'il devienne maître à son tour. Lorsque le PE situé à l'autre extrémité de la ligne est passé maître et a lui-même terminé, un autre message particulier est propagé à

tous les PE de la ligne pour leur indiquer de changer de passe. Chacun de ces PE peut alors effectuer la première étape de la deuxième passe, indépendamment du reste du réseau. Lorsqu'ils commencent la deuxième étape, qui consiste à effectuer des permutations de segments de colonne, la synchronisation entre PE appartenant à des lignes différentes se fait de manière purement locale et sans mécanisme particulier, par l'utilisation de communications bloquantes avec un voisinage adapté (i.e. tous les PE ne communiquement qu'horizontalement dans une passe impaire et verticalement dans une passe paire). Il y a donc un entrelacement entre passes, c'est-à-dire que certains processeurs peuvent effectuer une partie d'une passe alors que d'autres n'ont pas terminé la passe précédente. Il n'y a ainsi aucune synchronisation entre lignes (colonnes) mais un ensemble de synchronisations locales à l'intérieur de chaque ligne (colonne) de processeurs qui effectuent la même passe, et le réseau n'effectue que les opérations nécessaires au traitement des pixels de l'objet grâce au contrôle associatif et flot de données, tout ceci correspondant à une minimisation des synchronisations et des dépendances de l'algorithme.

Description du programme

Le programme est donné en annexe (fichier "rotate.asm"). Notons d'abord que deux simplifications (non limitatives ni incontournables) ont été introduites: l'objet doit être contenu dans la zone centrée dont la taille est égale au quart de l'image (la Figure 41 utilisant un objet de taille maximale), et au lieu d'utiliser un masque binaire, on considère que les pixels appartenant à l'objet sont ceux dont la valeur est différente de zéro.

Les plans mémoire 2 et 3 stockent les coefficients permettant de calculer les translations pour 256 angles compris dans l'intervalle]0,pi/2], car dans ce programme, un octet est envoyé par le contrôleur global pour spécifier l'angle choisi (en fait neuf bits, pour effectuer aussi les rotations dans l'intervalle [-pi/2,0[). La mémorisation de tous ces coefficients n'est bien sûr pas nécessaire, si le contrôleur global envoie directement à chaque rotation les deux coefficients correspondants. Le plan 4 contient initialement l'image source. Les plans 4 et 5 servent aux calculs intermédiaires et contiennent finalement le résultat de la rotation. Les plans 6 et 7, servent respectivement à marquer des pixels et à stocker des valeurs de translation. On suppose que les registres r_{14} et r_{15} contiennent les coordonnées du processeur. Les différents types de messages sont différenciés par l'utilisation des bits restants (appelés "tags" du message) du registre de voisinage (cf paragraphe 5.2.4).

L'algorithme est le suivant:

- attendre un message du contrôleur global, qui contient l'adresse des coefficients à utiliser (lignes 25 à 27), et initialiser certains registres en fonction de la passe
- initialiser le plan mémoire destination et le plan marquant les pixels de l'objet
- calculer les translations correspondant à chacune des 16 lignes (colonnes) associées à un processeur, et mémoriser la valeur de la translation complète ainsi que la valeur de la

translation restante une fois les pixels déplacés dans leur bloc destination (sous la forme d'un bit de direction et d'un nombre de décalages compris dans [0,8]) (lignes 54 à 93)

- tant qu'il y a des pixels marqués comme appartenant à l'objet (première étape d'une passe, lignes 95 à 136):
  - lire dans le plan mémoire source la valeur d'un pixel marqué
  - calculer le numéro de son bloc destination (en fonction de la passe)
  - stocker le pixel dans le plan mémoire destination à l'adresse correspondante
  - effacer la marque du pixel (tag_objet) et positionner une autre marque (tag_envoi) pour les pixels à envoyer (i.e. dont la translation restante est non nulle)
- si l'abscisse (l'ordonnée) du processeur est égale à 0, devenir le PE "maître" (ligne 163)
- pour le processeur maître (deuxième étape, lignes 164 à 176, 178 à 205, 207 à 212, 138 à 144):
  - chercher un pixel à envoyer (lignes 164 et 165)
  - s'il n'en reste plus,
    - si l'abscisse (l'ordonnée) du processeur est égale à 15 (lignes 207 à 209), envoyer un message de tag "011" au processeur voisin, pour propager une information de fin de passe, attendre "le retour" du message du côté opposé (lignes 138 à 144), et passer à la section de fin de passe (lignes 146 à 160)
    - sinon, envoyer un message de tag "000" au processeur voisin pour lui donner le contrôle (lignes 210 à 212), et devenir un PE "non maître"
  - sinon, lire dans le plan mémoire destination la valeur du pixel, envoyer au processeur voisin un message de tag "001" contenant le numéro de bloc du pixel, pour propager ce numéro aux autres PE, attendre "le retour" du message du côté opposé (lignes 169 à 176), et passer à la section qui effectue la permutation du segment (lignes 178 à 205)
- pour un processeur "non maître" (lignes 214 à 228, 178 à 205):
  - devenir inactif jusqu'à réception d'un message
  - si le message a pour tag "000", devenir PE "maître"
  - sinon, propager le message vers le voisin opposé, et
    - si le message a pour tag "011", passer à la section de fin de passe
    - sinon (tag "001"), lire dans le plan mémoire destination le pixel dont l'adresse est contenue dans le message, et passer à la section qui effectue la permutation du segment
- effectuer la permutation du segment (lignes 178 à 205), pour cela:
  - effacer la marque du pixel (tag_envoi)

  - déterminer à partir du numéro de bloc du pixel, le nombre de décalages (en lisant en mémoire la translation restante correspondante) et la direction de décalage (en fonction de la passe et du bit de direction mémorisé avec la translation restante)
  - envoyer et recevoir un pixel autant de fois que spécifié par le nombre de décalages
  - stocker le pixel final dans le plan mémoire destination
  - retourner au traitement correspondant à un PE "maître" ou "non maître"
- avant d'effectuer la passe suivante ou de finalement devenir inactif, recopier le plan mémoire destination dans le plan mémoire source (les lignes 146 à 154 étant un exemple typique de "traitement associatif", le nombre d'instructions pour traiter tout un ensemble de pixels ne dépendant pas de la longueur du plan mémoire mais de sa largeur).

Résultats de simulation

Le Tableau 9 résume les évaluations obtenues par simulation du programme donné en annexe. Le temps total d'exécution est de l'ordre de 0,5 à 0,6 milliseconde pour un objet de taille 127*127 pixels (le temps d'exécution dépendant de l'angle de rotation mais surtout de la taille de l'objet). La puissance moyenne de calcul est de l'ordre de 8 à 9 GIPS, ce qui est principalement dû aux synchronisations indirectes des processeurs par ligne ou colonne. La Figure 42 et la visualisation du plan mémoire destination au cours de l'exécution (cf annexe) permettent d'observer les différentes passes, leurs étapes, ainsi que l'entrelacement. Ce programme, qui comporte le séquencement le plus complexe parmi tous ceux qui ont été présentés, montre que l'on peut mettre en oeuvre de manière efficace des algorithmes relativement complexes, irréguliers, à dépendances non locales, à partir d'un fonctionnement purement cellulaire, et avec assez peu d'instructions (moins de 200 pour cet algorithme de rotation flot de données).

Taille du programme	195 instructions				
Image source	face.pic				
Angle de rotation (]0,256] pour]0,pi/2])	10	192		256	
Taille des buffers	>4	1	>8	1	>8
Nombre max. de messages	4	1	8	1	8
Nb. min. d'instructions par PE (kilo)	13,8	16,3		5,2	
Nb. max. d'instructions par PE (kilo)	15,0	22,4		26,3	
Nb. total d'instructions exécutées (Mega)	3,7	5,3		5,0	
Temps total d'exécution (ms)	0,460	0,595	0,595	0,626	0,625
Puissance moyenne de calcul (GIPS)	7,97	8,88	8,88	7,93	7,95

Tableau 9: Evaluations du programme "rotate.asm"

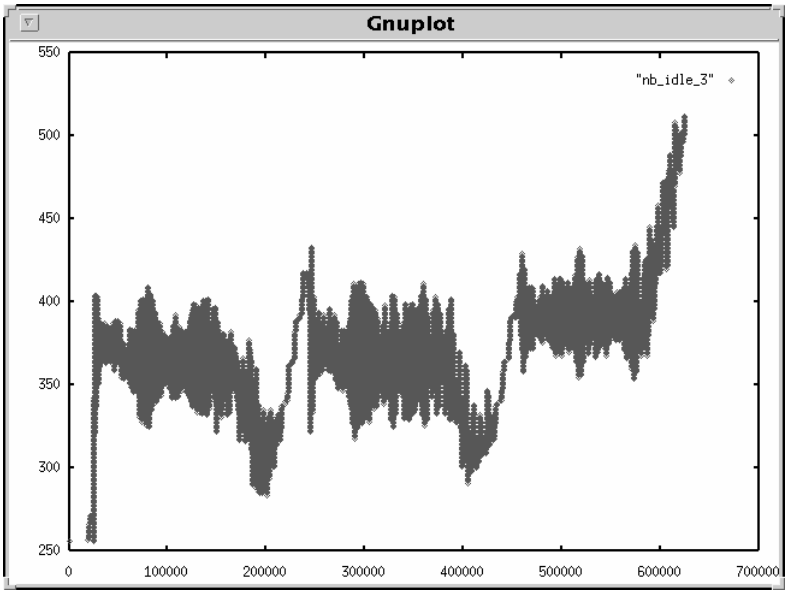

Figure 42: Activité logique du réseau

5.5 Introduction à la conception de circuits VLSI asynchrones quasi-insensibles aux délais

Le circuit AMPHIN a constitué une première expérience de conception de "processeur" asynchrone, comportant non seulement des opérateurs arithmétiques mais aussi un contrôle paramétrable et "complexe" par rapport à une simple structure combinatoire ou en anneau, comme les opérateurs présentés dans [ElHa95a], sur lesquels nous nous sommes basés. Nous ne pouvons développer ici la conception du processeur élémentaire qui est à la base de l'architecture décrite dans ce chapitre, mais souhaitons introduire le cadre de conception asynchrone que nous étudions actuellement, de manière à montrer le lien qui existe entre la spécification de ce modèle architectural et son implémentation VLSI. La conception d'un microprocesseur RISC asynchrone (dont celle d'un PE pourrait en partie être inspirée) fait actuellement l'objet des travaux du groupe de Marc Renaudin (Telecom Bretagne), au CNET-Grenoble, et démontre la faisabilité d'un tel circuit, un prototype en technologie 0.25 µm devant être envoyé en fabrication fin 1997.

5.5.1 Concevoir des circuits QDI en programmant

Nous étudions actuellement une méthodologie de conception de circuits quasi-insensibles aux délais utilisant un formalisme basé sur les processus concurrents communiquants. Cette méthodologie a été créée par le groupe d'Alain Martin à Caltech [Mart93] [Mart90] [Hauc95]. Nous avons développé à partir de celle-ci un style d'implémentation basé sur une approche "cellules standard" ("standard cells"), correspondant à l'étude d'un certain nombre des "choix subtils" invoqués dans [Hauc95] vis-à-vis des

différentes étapes de "compilation". Contrairement à la conception du circuit AMPHIN, qui était basée sur une approche architecturale consistant à assembler des blocs de contrôle élémentaires obtenus à partir de STG avec des chemins de données en logique DCVS, la méthodologie que nous utilisons dorénavant est fondée sur la transformation d'une spécification programmée dans un langage inspiré du CSP (Communicating Sequential Processes) [Hoar85] et des commandes gardées [Dijk75] vers le schéma transistor d'un circuit équivalent. La spécification est ainsi programmée en langage CHP (Communicating Hardware Processes), sous la forme de processus concurrents qui communiquent exclusivement par l'intermédiaire de canaux locaux ("point-à-point"). Cette spécification est transformée par décomposition jusqu'à obtention d'un ensemble de processus suffisamment simples, ce qui permet ensuite d'effectuer la phase de "compilation" qui aboutit au circuit proprement dit. La compilation comprend deux étapes appelées "expansion des communications" et "génération des règles de production", qui prennent en compte les styles de protocole et de cible technologique. Cette méthodologie de conception est basée sur un formalisme qui permet une programmation aisée, tout en favorisant l'étude des dépendances fonctionnelles et des synchronisations présentes dans une spécification. Elle permet une progression continue entre le niveau fonctionnel et le niveau structurel, par l'intermédiaire d'une sémantique homogène de la spécification jusqu'à la réalisation, et peut ainsi facilement préserver la correction au cours de toutes les transformations. Elle apporte aussi une bonne lisibilité conjointe de l'algorithme et de l'architecture, permet de prendre en compte des contraintes à la fois fonctionnelles et matérielles, et offre une transposition directe de la spécification décomposée en circuits logiques. Nous pensons que cette méthodologie et le type de circuits associé forment le cadre de conception le plus prometteur pour la réalisation de circuits asynchrones.

5.5.2 Exemples sur la mise en œuvre d'interfaces de communication

Nous illustrons brièvement dans cette section la conception de circuits quasi-insensibles aux délais, sur un exemple d'implémentation d'interfaces de communication bloquante et non-bloquante.

a) Communications bloquantes

Soit une interface de communication unidirectionnelle comportant un canal d'entrée E et un canal de sortie S, qui s'insère entre deux processus devant communiquer de manière bloquante (Figure 43), et dont la spécification en langage CHP sous la forme d'un processus est la suivante:

*[E?x ; S!x]

La structure *[...] est une structure de répétition qui indique que le processus effectue sa tâche une infinité de fois (il s'agit en fait d'une notation abrégée de la structure *[true→...], qui comporte une garde toujours vraie). Les opérations E?x et S!x sont des actions de communication, respectivement de lecture et d'écriture. L'action E?x signifie qu'une communication est effectuée en lecture sur le canal E, et que la valeur transmise est stockée dans la variable locale x. L'action S!x signifie de manière duale que le

contenu de la variable x est envoyé dans le canal S. Le point virgule est un opérateur de séquentialité, qui indique la précédence fonctionnelle entre la lecture et l'écriture, c'est-à-dire le fait que l'action de communication sur E doit être terminée avant d'effectuer celle sur S. Cette interface réalise bien une communication bloquante parce que les actions de communications du CHP constituent par définition une synchronisation entre deux processus par l'intermédiaire d'un canal.

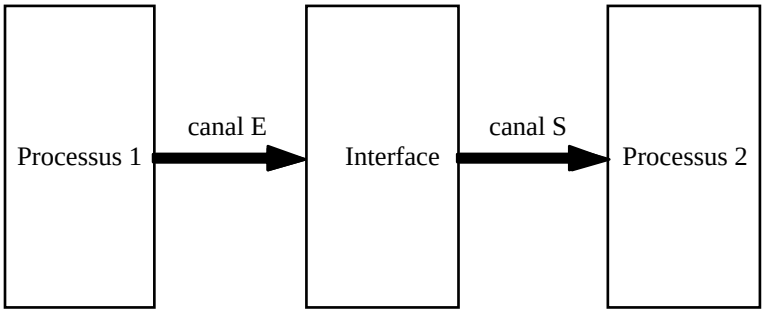

Figure 43: Interface de communication bloquante unidirectionnelle

Supposons que les canaux ne transmettent qu'un seul bit. Chaque canal utilise alors un codage "double rail" et est composé de trois signaux, deux pour la donnée (un par rail) et un pour l'acquittement. En utilisant un protocole à quatre phases et certaines conventions de polarité (entrées "actives" et sorties "passives") et d'entrelacement ("reshuffling"), l'expansion des communications peut s'écrire:

[Si^Ei] ; So+ ; Eo- ; [¬Si^¬Ei] ; So- ; Eo+

[Si^Eib] ; Sob+ ; Eo- ; [¬Si^¬Eib] ; Sob- ; Eo+

où les crochets indiquent une attente de la validité d'une expression booléenne, le point virgule étant encore l'opérateur de séquentialité. Ei et Eib constituent le double rail de donnée entrant et Eo le signal d'acquittement sortant du canal d'entrée E. So et Sob constituent le double rail de donnée sortant et Si le signal d'acquittement entrant du canal de sortie S. La première ligne correspond à la transmission de la valeur "1", entre les rails Ei et So, la deuxième à celle de la valeur "0" entre les rails Eib et Sob. La génération des règles de production conduit facilement au schéma d'implémentation de la Figure 44, comportant deux "portes de Muller" (ou "C-elements") et une porte "nor".

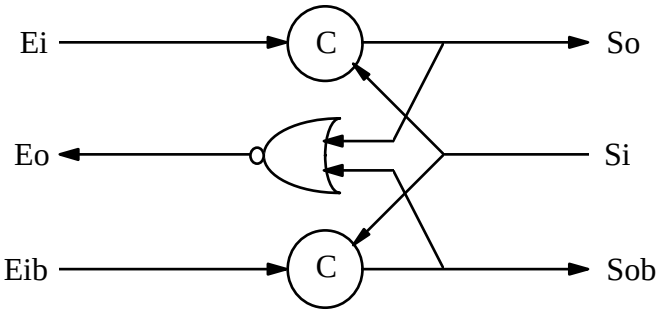

Figure 44: Schéma niveau porte logique d'une interface de communication bloquante ("buffer")

En fait, cette interface n'est pas nécessairement utile d'un point de vue fonctionnel, puisque les actions de communication spécifient directement des communications bloquantes. Elle constitue cependant un élément de mémorisation intermédiaire et peut toujours être insérée entre deux processus. En cascadant plusieurs de ces interfaces, on obtient une file de mémorisation de type FIFO, qui correspond précisément aux "buffers" des interfaces de communication, dont le rôle a été discuté dans les sections précédentes (une cellule de FIFO équivalente à la Figure 44 se trouve par exemple dans [Mart87] [Hulg95]).

b) Communications non-bloquantes

Considérons maintenant une interface de communication qui s'insère entre deux processus devant communiquer de manière non-bloquante, toujours unidirectionnelle avec un canal d'entrée E et un canal de sortie S. La communication étant non-bloquante du point de vue des deux processus, il faut ajouter pour chacun d'eux un canal permettant de signaler si la communication a été effective ou non. Associons au canal E un canal de sortie Eok, sur lequel un "1" est transmis en retour d'une communication effective et un "0" sinon, et associons de même au canal S un canal de sortie Sok, avec une signalisation identique (Figure 45).

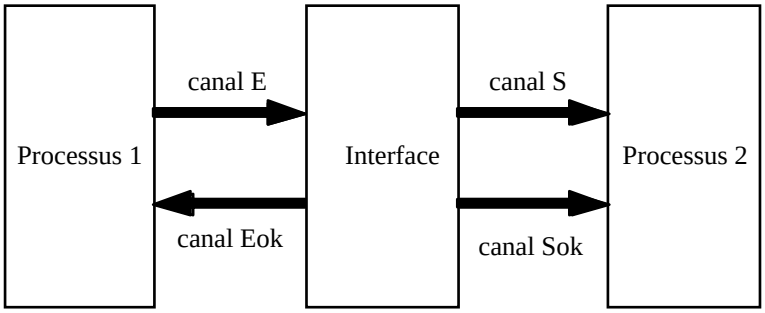

Figure 45: Interface de communication non-bloquante unidirectionnelle

La spécification de l'interface en langage CHP est la suivante:

```
*[ #E→  [ sema=0 → E?x , Eok!1 , sema:=1
        @ sema=1 → E? , Eok!0
        ]
 | #S→  [ sema=1 → S!x , Sok!1 , sema:=0
        @ sema=0 → S!x , Sok!0
        ]
 ]
```

La structure [G_1→... | ... | G_n→...] est une structure de choix indéterministe entre plusieurs gardes non-exclusives, c'est-à-dire qui peuvent être vraies simultanément. La notation #, appelée "sonde"

("probe"), est un opérateur booléen qui s'applique à un canal, et qui teste si une action de communication est en cours sur ce canal. Dans cette interface, on teste donc si un des deux processus entame une action de communication (sur E ou S) et la commande gardée correspondante est exécutée dès la détection de l'action (si les deux gardes #E et #S deviennent vraies simultanément, une des gardes est choisie de manière indéterministe et l'autre est prise en compte après exécution de la première commande). Le processus de l'interface contient deux variables locales. La variable x stocke la valeur à transmettre entre les deux processus et la variable sema (pour "sémaphore") mémorise l'état de l'interface et permet de définir les actions de communication sur les canaux Eok et Sok, qui indiquent si la communication est effective.

La structure [$G_1 \rightarrow ...$ @ ... @ $G_n \rightarrow ...$] est une structure de choix déterministe, les gardes étant exclusives. Supposons que le processus 1 entame une action de communication sur E (avec une instruction E!var). Si la variable sema vaut "0", ce qui indique que l'interface peut prendre en compte la valeur transmise sur E, alors cette valeur est lue dans la variable x, l'interface envoie la valeur "1" sur Eok pour signaler que la communication a été effective, et la variable sema est positionnée à "1" pour mémoriser le fait qu'une valeur a été écrite dans l'interface. Si la variable sema vaut déjà "1", alors un "0" est envoyé sur Eok pour signaler que la communication n'est pas effective et le canal E est acquitté par E?, sans modifier la variable x (dont le contenu n'a pas encore été consommé par le processus 2). Les différentes instructions qui constituent la commande gardée sont séparées par des virgules, qui désignent l'opérateur de concurrence, car elles peuvent être exécutées en parallèle, la commande se terminant lorsque toutes les instructions sont terminées. La gestion des canaux S et Sok se fait de manière duale. Si la variable sema vaut "1", le contenu de la variable x est envoyé dans le canal S, un "1" est envoyé sur Sok pour signaler que la communication est effective, et la variable sema est positionnée à "0" pour indiquer que la valeur a été consommée. Si la variable sema vaut "0", un "0" est envoyé sur Sok pour signaler que la communication n'est pas effective et l'interface transmet tout de même sur le canal S le contenu de la variable x, c'est-à-dire la dernière valeur qui a été écrite dans l'interface (qui a déjà été consommée auparavant, mais qui peut éventuellement être utilisée à nouveau par le processus 2).

Cette interface a été "compilée" et simulée. Nous ne présentons pas les étapes de conception, mais le schéma niveau portes est donné en annexe. Les portes "rondes" notées CZ2 ou CZ3 sont des portes de Muller. Le symbole noté ME est un élément d'exclusion mutuelle. La simulation électrique a montré que cette interface est très rapide (temps de cycle de 1.5ns pour une suite d'écritures, 1.2ns pour une suite de lectures, et 2ns en entrée et en sortie pour des lectures/écritures alternées, en technologie CMOS 0.25µm à base de cellules standard, les simulations étant effectuées avant routage). Cet exemple illustre le fait que les communications asynchrones peuvent être implémentées avec une fine granularité, et sans aucun surcoût, le temps de cycle des interfaces ne pouvant constituer un goulot d'étranglement.

5.6 Conclusion

Dans ce chapitre, nous avons généralisé et illustré la combinaison du parallélisme et de l'asynchronisme à grain fin, dans le cadre de la conception VLSI d'un coprocesseur programmable pour l'analyse / rendu d'images. Grâce au relâchement des contraintes de synchronisation et de séquencement, les architectures cellulaires peuvent exploiter de manière efficace le parallélisme potentiel de nouvelles classes d'algorithmes, présentant des irrégularités qui les rendent incompatibles ou qui en limitent fortement les performances avec les modèles de conception et de fonctionnement classiques. La notion d'asynchronisme au sens large est exploitée aux niveaux algorithmique, architectural et matériel, intégrant aussi les modes de contrôle associatif et flot de données. Cette exploitation est illustrée par l'évaluation de primitives algorithmiques originales, qui mettent en évidence l'élargissement du spectre de solutions rendu possible par l'ensemble des concepts architecturaux introduits, ainsi que l'adéquation du modèle vis-à-vis de l'évolution du codage d'images et de la technologie (le principe de localité devenant essentiel pour la facilité de conception, la robustesse et les performances des futurs circuits intégrés).

CONCLUSION

Les systèmes de communication visuelle de deuxième génération présentent des caractéristiques inédites en termes de besoins en puissance de calcul, de généricité, d'interactivité, qui correspondent à une convergence des représentations et des traitements liés à l'information image. Les architectures VLSI pour la mise en oeuvre des systèmes de première génération ont déjà évolué vers des solutions programmables telles que les "media-processors", qui permettent d'atteindre des performances remarquables et d'implémenter une première forme de convergence ou plutôt d'intégration des différents média. Nous pensons cependant que cette approche ne possède pas de bonnes propriétés de "scalabilité", c'est-à-dire que le seul "changement d'échelle" de la structure avec celui de la technologie n'est pas suffisant pour espérer pouvoir mettre en oeuvre en temps réel les systèmes de deuxième génération, ou tout du moins que des approches alternatives pourraient se révéler mieux adaptées aux possibilités d'intégration que nous promettent les futures technologies VLSI.

La voie étudiée dans cette thèse est basée sur la combinaison du parallélisme massif et de l'asynchronisme à grain fin. De manière plus générale, elle s'inscrit dans une évolution des architectures VLSI vers la convergence mémoire-traitement et vers des modes de fonctionnements moins contraints. Le principe de base est en fait un principe de localité, qui se dérive par la suite en concepts de traitement concurrent et de synchronisation minimale. Nous avons d'abord essayé de montrer les différentes facettes de la (ou des) notion(s) d'asynchronisme. Alors que certaines de ses propriétés bénéfiques ont été souvent remarquées mais jamais véritablement exploitées, nous avons tenté de concrétiser l'exploitation de l'asynchronisme à différents niveaux, en relation avec la conception de circuits VLSI et l'algorithmique des systèmes de communication visuelle. Il nous semble que l'expérience du circuit AMPHIN démontre la faisabilité de mise en oeuvre des concepts étudiés et ouvre de nouvelles perspectives de conception conjointe algorithme-architecture. La combinaison du parallélisme et de l'asynchronisme a ensuite été généralisée et rattachée à un ensemble de modèles de traitement et de propriétés ou problèmes liés aux systèmes concurrents, comme les traitements flot de données, associatif, cellulaire, le fonctionnement SPMD, le partitionnement, la détection de fin globale, les communications

bloquantes et non-bloquantes... Cet ensemble de concepts n'est pas une simple compilation de modèles revisités, mais forme un tout cohérent, qui définit de nouvelles formes de conception et de programmation d'architectures et d'algorithmes ("algotectures" ?!), appliquées à la spécification d'un coprocesseur programmable pour l'analyse/rendu d'images. En résumé, il nous semble que le relâchement des contraintes de synchronisation et de séquencement permet à la fois d'élargir le spectre de solutions (à la fois algorithmiques et architecturales) et de mieux exploiter la technologie VLSI.

De nombreux points pourraient être approfondis sur l'expérience AMPHIN et le modèle du dernier chapitre. Comme nous l'avons souligné, il existe des techniques de conception de circuits asynchrones plus performantes que celles utilisées dans AMPHIN. Il serait intéressant d'utiliser ces techniques pour concevoir une version étendue et moins spécifique du processeur, en intégrant par exemple une ALU générale, quelques registres, un automate de séquencement véritablement programmable, et en améliorant les interfaces entre processeurs et d'entrées/sorties globales. Les entrées/sorties et le contrôle global, ainsi que la détection de fin de calcul, sont aussi à approfondir pour le modèle de coprocesseur programmable. Il serait de plus intéressant de compléter les évaluations algorithmiques de ce modèle, en particulier par la mise en oeuvre de primitives de rendu 2D et 3D, par l'application de l'asynchronisme fonctionnel à d'autres opérateurs, comme le "watershed", ou encore par l'évaluation complète du problème de segmentation. L'application des techniques de conception de circuits "QDI" à la réalisation des différents constituants des PE constitue une étude en cours, qui permettra d'évaluer la complexité de la structure (thèse de P. Vivet, ENST de Bretagne, CNET-Grenoble).

Une perspective naturelle d'élargissement de cette étude serait la prise en compte d'un niveau supplémentaire de convergence, la convergence capteur-traitement (voire même capteur-traitement-actuateur). Le principe de localité est alors appliqué au niveau de l'acquisition de l'information image, et de son lien avec la structure de traitement. L'intégration de photodiodes dans les processeurs élémentaires d'un réseau cellulaire est une étape complémentaire permettant d'éliminer les goulots d'étranglement d'une chaîne de traitement globale destinée à la vision artificielle. Il existe déjà un tel domaine de recherche, mais qui part d'une approche duale, consistant à intégrer des capacités de traitement à un coût minimum dans une matrice de capteurs. On parle alors de "capteurs intelligents" ("smart sensors"), de rétines artificielles, ou de traitement d'image au voisinage du capteur ("near-sensor image processing") [Bern93] [Forc94] [Åstr96] [Eklu96] [Morr94] [Dick95]. Ces réalisations sont basées sur la superposition d'un réseau analogique ou logique dont la complexité doit garder des proportions "raisonnables" vis-à-vis de celle de la matrice de capteurs (par exemple en termes de surface relative). Il s'agit peut-être de déterminer un niveau de granularité intermédiaire qui permettrait de combiner les différentes approches, en incorporant une matrice de photodétecteurs, un réseau de prétraitement analogique paramétrable pour le filtrage spatio-temporel, un réseau de traitement logique programmable localement synchronisé, et une certaine quantité de mémoire associative par exemple. Le développement des technologies d'interconnexions 3D et optiques sera certainement à prendre en compte.

REFERENCES

[Aimo96] Y. Aimoto et al., "A 7.68GIPS 3.84GB/s 1W parallel image-processing RAM integrating a 16Mb DRAM and 128 processors", Digest of Technical Papers, International Solid-State Circuits Conference, IEEE, 1996, pp. 372-373, 476.

[Aiza95] K. Aizawa, T. Huang, "Model-based image coding: advanced video coding techniques for very low bit-rate applications", Proceedings of the IEEE, Vol. 83, No. 2, February 1995, pp. 259-271.

[Aker95] A. Akerib, R. Adar, "Associative approach to real time color, motion and stereo vision", Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, 1995, pp. 3291-3294.

[Akim93] T. Akimoto, Y. Suenaga, R. Wallace, "Automatic creation of 3D facial models", IEEE Computer Graphics and Applications, September 1993, pp. 16-22.

[Ang91] P. Ang, P. Ruetz, D. Auld, "Video compression makes big gains", IEEE Spectrum, October 1991, pp. 16-19.

[Arav93] R. Aravind, G. Cash, D. Duttweiler, H.-M. Hang, B. Haskell, A. Puri, "Image and video coding standards", AT&T Technical Journal, January-February 1993, pp. 67-89.

[Åstr96] A. Åström, R. Forchheimer, J.-E. Eklund, "Global feature extraction operations for near-sensor image processing", IEEE Transactions on Image Processing, Vol. 5, No. 1, January 1996, pp. 102-110.

[Awag95] M. Awaga, T. Ohtsuka, H. Yoshizawa, S. Sasaki, "3D graphics processor chip set", IEEE Micro, December 1995, pp. 37-45.

[Bará96] B. Barán, E. Kaszkurewicz, A. Bhaya, "Parallel asynchronous team algorithms: convergence and performance analysis", IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 7, July 1996, pp. 677-688.

[Batc80] K. Batcher, "Design of a massively parallel processor", IEEE Transactions on Computers, Vol. 29, 1980, pp. 836-840.

[Baud78] G. Baudet, "Asynchronous iterative methods for multiprocessors", Journal of the ACM, Vol. 25, No. 2, April 1978, pp. 226-244.

[Bern93] T. Bernard, B. Zavidovique, F. Devos, "A programmable artificial retina", IEEE Journal of Solid-State Circuits, Vol. 28, No. 7, July 1993, pp. 789-798.

[Berr89] G. Berry, "Real time programming: special purpose or general purpose languages", Information Processing (IFIP89), Elsevier Science Publishers, 1989, pp. 11-17.

[Berr93] G. Berry, S. Ramesh, R. K. Shyamasundar, "Communicating reactive processes", Proceedings of the 20th ACM Conference on Principles of Programming Languages, Charleston, 1993.

[Bert89] D. Bertsekas, J. Tsitsiklis, Parallel and Distributed Computation - Numerical Methods, Prentice-Hall, 1989.

[Bhas97] V. Bhaskaran, K. Konstantinides, B. Natarajan, "Multimedia architectures: from desktop systems to portable appliances", Proceedings SPIE Multimedia Hardware Architectures, Vol. 3021, 1997, pp. 14-25.

[Blin94] J. Blinn, "Compositing, Part I: Theory", IEEE Computer Graphics and Applications, September 1994, pp. 83-87.

[Bous92] F. Boussinot, Réseaux de processus réactifs, rapport de recherche No. 1588, INRIA, Sophia-Antipolis, France, Janvier 1992.

[Bove96] V. Bove, "Multimedia based on object models: some whys and hows", IBM Systems Journal, Vol. 35, No. 3-4, 1996, pp. 337-348.

[Brig95] P. Brigger, M. Kunt, "Morphological shape representation for very low bit-rate video coding", Signal Processing: Image Communication, Vol. 7, 1995, pp. 297-311.

[Care97] D. Carevic, T. Caelli, "Region-based coding of color images using Karhunen-Loeve transform", Graphical Models and Image Processing, Vol. 59, No. 1, January 1997, pp. 27-38.

[Casa94] J. Casas, L. Torres, "Coding of details in very low bit-rate video systems", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 4, No. 3, June 1994, pp. 317-327.

[Chai92] C. Chaillou, Architectures des systèmes pour la synthèse d'images, Dunod informatique, 1992.

[Chan73] T. Chaney, C. Molnar, "Anomalous behavior of synchronizer and arbiter circuits", IEEE Transactions on Computers, Vol. 22, No. 4, April 1973, pp. 421-422.

[Chan89] K. M. Chandy, J. Misra, Parallel Program Design - a foundation, Addison-Wesley, 1989.

[Char93] F. Charot, Architectures parallèles spécialisées pour le traitement d'image, Publication interne No. 722, IRISA, Rennes, France.

[Chaz69] D. Chazan, W. Miranker, "Chaotic relaxation", Linear Algebra and its Applications, Vol. 2, 1969, pp. 199-222.

[Chen93] C.-T. Chen, "Video compression: standards and applications", Journal of Visual Communication and Image Representation, Vol. 4, No. 2, June 1993, pp. 103-111.

[Chia97] L. Chiariglione, "MPEG and multimedia communications", IEEE Transactions on Circuits and Systems for Video Technology, special issue on MPEG-4, Vol. 7, No. 1, February 1997, pp. 5-18.

[Chis89] L. Chisvin, R. J. Duckworth, "Content-addressable and associative memory: alternatives to the ubiquitous RAM", IEEE Computer, July 1989, pp. 51-64.

[Chu94] T. Chu, "Synthesis of hazard-free control circuits from asynchronous finite state machines specifications", Journal of VLSI Signal Processing, Vol. 7, 1994, pp. 61-84.

[Cich97] J. Cichosz, F. Meyer, "Segmentation morphologique multi-échelle en vue du codage", Actes des Journées d'études et d'échanges: COmpression et REprésentation des Signaux Audiovisuels, CNET, Issy-les-Moulineaux, France, Mars 1997.

[Come93] S. Comes, S. Maes, M. Van Droogenbroeck, "Recent and prospective decorrelation techniques for image processing", Annales des Télécommunications, Vol. 48, No. 7-8, 1993, pp. 390-403.

[Cort95] D. Cortez, P. Nunes, M. Menezes de Sequeira, F. Pereira, "Image segmentation towards new image representation methods", Signal Processing: Image Communication, Vol. 6, 1995, pp. 485-498.

[Croc97] T. Crockett, "An introduction to parallel rendering", Parallel Computing, Vol. 23, 1997, pp. 819-843.

[Cyph89] R. Cypher, J. Sanz, "SIMD architectures and algorithms for image processing and computer vision", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 37, No. 12, December 1989, pp. 2158-2173.

[Davo95] F. Davoine, Compression d'images par fractales basée sur la triangulation de Delaunay, Thèse de Doctorat de l'I.N.P.G., Grenoble, France, 1995.

[Davo96] F. Davoine, J.-M. Chassery, "Compression d'images par fractales", Actes des Journées d'études et d'échanges: COmpression et REprésentation des Signaux Audiovisuels, CNET, Grenoble, France, Février 1996, pp. 56-63.

[Dick95] A. Dickinson, B. Ackland, E.-S. Eid, D. Inglis, E. Fossum, "A 256x256 CMOS active pixel image sensor with motion detection", Digest of Technical Papers, International Solid-State Circuits Conference, IEEE, 1995, pp. 226-227.

[Dijk75] E. Dijkstra, "Guarded commands, nondeterminancy and formal derivations of programs", Communications of the ACM, Vol. 18, August 1975.

[Dinn89] A. Dinning, "A survey of synchronization methods for parallel computers", IEEE Computer, July 1989, pp. 66-77.

[Dudo96] M. Dudon, G. Eude, C. Roux, "Treillis actif et codage optimal", Actes des Journées d'études et d'échanges: COmpression et REprésentation des Signaux Audiovisuels, CNET, Grenoble, France, Février 1996, pp. 10-17.

[Dufa95] F. Dufaux, F. Moscheni, "Motion estimation techniques for digital TV: a review and a new contribution", Proceedings of the IEEE, Vol. 83, No. 6, June 1995, pp. 858-875.

[Dula96] D. Dulac, Contribution au parallélisme massif en analyse d'image: une architecture SIMD fondée sur la reconfigurabilité et l'asynchronisme, Thèse de Doctorat de l'université Paris XI, Orsay, France, Janvier 1996.

[Dunc90] R. Duncan, "A survey of parallel computer architectures", IEEE Computer, February 1990, pp. 5-16.

[Dunn92] G. Dunnett, M. White, P. Lister, R. Grimsdale, F. Glemot, "The Image chip for high performance 3D rendering", IEEE Computer Graphics and Applications, November 1992, pp. 41-52.

[Dyer81] C. Dyer, A. Rosenfeld, "Parallel image processing by memory-augmented cellular automata", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 3, No. 1, January 1981, pp. 29-41.

[Ebra95] T. Ebrahimi, E. Reusens, W. Li, "New trends in very low bitrate video coding", Proceedings of the IEEE, Vol. 83, No. 6, June 1995, pp. 877-891.

[Eden85] M. Eden, M. Kocher, "On the performance of a contour coding algorithm in the context of image coding - Part I: contour segment coding", Signal Processing, Vol. 8, 1985, pp. 381-386.

[Eklu96] J.-E. Eklund, C. Svensson, A. Åström, "VLSI implementation of a focal plane image processor - a realization of the near-sensor image processing concept", IEEE Transactions on VLSI Systems, Vol. 4, No. 3, September 1996, pp. 322-335.

[ElHa95a] B. El Hassan, Architecture VLSI asynchrone utilisant la logique différentielle à précharge: application aux opérateurs arithmétiques, Thèse de Doctorat de l'I.N.P.G., Grenoble, France, Septembre 1995.

[ElHa95b] B. El Hassan, A. Guyot, M. Renaudin, V. Levering, "New self timed rings and their application to division and square root extraction", Proceedings of the European Solid-State CIRcuits Conference, Lille, France, September 1995, pp. 226-229.

[Elli92] D. Elliott, W. M. Snelgrove, M. Stumm, "Computational RAM: a memory-SIMD hybrid and its application to DSP", Proceedings of the Custom Integrated Circuits Conference, IEEE, 1992, pp. 30.6.1-30.6.4.

[Elli97] D. Elliott, M. Snelgrove, C. Cojocaru, M. Stumm, "Computing RAMs for media processing", Proceedings SPIE Multimedia Hardware Architectures, Vol. 3021, 1997, pp. 66-77.

[Erde84] C. Erdelyi, W. Griffin, R. Kilmoyer, "Cascode Voltage Switch Logic Design", VLSI Design, October 1984, pp. 78-86.

[Faur91] B. Faure, S. Karabernou, G. Mazaré, E. Payan, P. Rubini, "Une architecture massivement parallèle intégrée", Annales des Télécommunications, Vol. 46, No. 1-2, 1991, pp. 90-100.

[Fet95] Y. Fet, "Vertical processing systems: a survey", IEEE Micro, Vol. 15, No. 1, February 1995, pp. 65-75.

[Floh96] U. Flohr, "3-D for everyone", Byte, October 1996, pp. 76-88.

[Fole96] J. Foley, A. van Dam, S. Feiner, J. Hughes, Computer Graphics: Principles and Practice, second edition in C, Addison Wesley, 1996.

[Forc89] R. Forchheimer, T. Kronander, "Image coding - from waveforms to animation", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 37, No. 12, December 1989, pp. 2008-2023.

[Forc94] R. Forchheimer, A. Åström, "Near-sensor image processing: a new paradigm", IEEE Transactions on Image Processing, Vol. 3, No. 6, November 1994, pp. 736-746.

[Fost76] C. Foster, Content Addressable Parallel Processors, Van Nostrand Reinhold, 1976.

[Foun87] T. Fountain, Processor Arrays - Architectures and Applications, Academic Press, 1987.

[Geal96] J. Gealow, F. Herrmann, L. Hsu, C. Sodini, "System design for pixel-parallel image processing", IEEE Transactions on VLSI Systems, Vol. 4, No. 1, March 1996, pp. 32-41.

[Gerk94] P. Gerken, "Object-based analysis-synthesis coding of image sequences at very low bit rates", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 4, No. 3, June 1994, pp. 228-235.

[Gibb93] P. Gibbons, "Asynchronous PRAM algorithms", in Synthesis of Parallel Algorithms, Morgan Kaufmann Publishers, 1993, pp. 957-997.

[Glas96] P. Glaskowsky, "Talisman redefines 3D rendering", Microprocessor Report, Vol. 10, No. 11, August 26, 1996, pp. 5, 10-11.

[Glas97] P. Glaskowsky, "First media processors reach the market", Microprocessor Report, January 27, 1997, pp. 10-15.

[Gokh95] M. Gokhale, B. Holmes, K. Iobst, "Processing in memory: the Terasys massively parallel PIM array", IEEE Computer, April 1995, pp. 23-31.

[Gols96] J. Golston, "Single-chip H.324 videoconferencing", IEEE Micro, August 1996, pp. 21-33.

[Good95] J. Goodenough, R. Meacham, J. Morris, N. Luke Seed, P. Ivey, "A single chip video signal processing architecture for image processing, coding, and computer vision", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 5, No. 5, October 1995, pp. 436-445.

[Gree90] D. Greening, "Parallel simulated annealing techniques", Physica D, Vol. 42, 1990, pp. 293-306.

[Grop94] W. Gropp, E. Lusk, A. Skjellum, Using MPI - Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1994.

[Gu95] C. Gu, M. Kunt, "Contour simplification and motion compensated coding", Signal Processing: Image Communication, Vol. 7, 1995, pp. 279-296.

[Gutt92] K. Guttag, R. Gove, J. Van Aken, "A single-chip multiprocessor for multimedia: the MVP", IEEE Computer Graphics and Applications,November 1992, pp. 53-64.

[Gwen96a] L. Gwennap, "Verite: a programmable 3D chip", Microprocessor Report, Vol. 10, No. 6, May 6, 1996, pp. 1-10.

[Gwen96b] L. Gwennap, "Media processors may have short reign", Microprocessor Report, Vol. 10, No. 13, October 7, 1996, pp. 3.

[Gwen96c] L. Gwennap, "Intel's MMX speeds multimedia", Microprocessor Report, Vol. 10, No. 3, March 5, 1996, pp. 1, 6-10.

[Gwen96d] L. Gwennap, "Digital, MIPS add multimedia extensions", Microprocessor Report, November 18, 1996, pp. 24-28.

[Hans96] C. Hansen, "MicroUnity's MediaProcessor architecture", IEEE Micro, August 1996, pp. 34-41.

[Hauc95] S. Hauck, "Asynchronous Design Methodologies : An Overview", Proceedings of the IEEE, Vol. 83, No. 1, January 1995, pp. 69-93.

[Heck97] G. Heckner, "Redundancy reducing coding of moving object shapes", Signal Processing: Image Communication, Vol. 9, 1997, pp. 91-98.

[Henn92] J. Hennessy, D. Patterson, Architecture des ordinateurs - Une approche quantitative, Ediscience international, 1992 (traduction de Computer Architecture - A Quantitative Approach, Morgan Kaufmann Publishers, 1990).

[Henn94] J. Hennessy, D. Patterson, Organisation et conception des ordinateurs - L'interface matériel / logiciel, Dunod, 1994 (traduction de Computer Organization and Design - The Hardware / Software Interface, Morgan Kaufmann Publishers, 1994).

[Héno93] J.-P. Hénot, J. Mau, V. Thomas, "Systèmes de codage pour l'image de télévision", L'écho des Recherches, No. 151, 1er trimestre 1993, pp. 5-16.

[Herr92] F. Herrmann, C. Sodini, "A dynamic associative processor for machine vision applications", IEEE Micro, Vol. 12, No. 3, June 1992, pp. 31-41.

[Herr95] F. Herrmann, C. Sodini, "A 256-element associative parallel processor", IEEE Journal of Solid-State Circuits, Vol. 30, No. 4, April 1995, pp. 365-370.

[Heud90] J.-C. Heudin, C. Panetto, Les architectures RISC - Théorie et pratique des ordinateurs à jeu d'instructions réduit, Dunod, 1990.

[Hill88] W. D. Hillis, La machine à connexions, Masson, 1988 (traduction de The Connection Machine, MIT Press, 1985).

[Hoar85] C.A.R. Hoare, Communicating Sequential Processes, International series in computer science, Prentice-Hall, 1985.

[Hoge88] P. Hogeweg, "Cellular automata as a paradigm for ecological modeling", Applied Mathematics and Computation, Vol. 27, 1988, pp. 81-100.

[Hött90] M. Hötter, "Object-oriented analysis-synthesis coding based on moving two-dimensional objects", Signal Processing: Image Communication, Vol. 2, 1990, pp. 409-428.

[Hött94] M. Hötter, "Optimization and efficiency of an object-oriented analysis-synthesis coder", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 4, No. 2, April 1994, pp. 181-194.

[Hulg95] H. Hulgaard, S. Burns, G. Borriello, "Testing asynchronous circuits: a survey", Integration - the VLSI Journal, Vol. 19, 1995, pp. 111-131.

[IEEE91] Proceedings of the IEEE, special section on Another Look at Real-Time Programming, Vol. 79, No. 9, September 1991, pp. 1268-1336.

[IEEE92a] IEEE Micro, special issue on Hardware for Video Coding, Vol. 12, No. 5, October 1992.

[IEEE92b] IEEE Micro, special issue on Associative Memories and Processors, Parts 1&2, Vol. 12, No. 3/6, June/December 1992.

[IEEE94] IEEE Computer, special issue on Associative Processing, Vol. 27, No. 11, November 1994.

[IEEE96] Proceedings of the IEEE, special issue on Parallel Architectures for Image Processing, Vol. 84, No. 7, July 1996.

[IEEE97] IEEE Transactions on Circuits and Systems for Video Technology, special issue on MPEG-4, Vol. 7, No. 1, February 1997.

[IMCO97a] Image Communication, special issue on MPEG-4, Part 1: Invited papers, Vol. 9, No. 4, Elsevier, May 1997.

[IMCO97b] Image Communication, special issue on MPEG-4, Part 2: Submitted papers, Vol. 10, Nos. 1-3, Elsevier, July 1997.

[Inou95] K. Inoue, H. Nakamura, H. Kawai, "A 10 Mb frame buffer memory with Z-compare and A-blend units", IEEE Journal of Solid-State Circuits, Vol. 30, No. 12, December 1995, pp. 1563-1568.

[Jack96] P. Jackway, "Gradient watersheds in morphological scale-space", IEEE Transactions on Image Processing, Vol. 5, No. 6, June 1996, pp. 913-921.

[Jain81] A. Jain, "Image data compression: a review", Proceedings of the IEEE, Vol. 69, No. 3, March 1981, pp. 349-389.

[Jaya92] N. Jayant, "Signal compression: technology targets and research directions", IEEE Journal on Selected Areas in Communications, Vol. 10, No. 5, June 1992, pp. 796-818.

[Jean96] R. Jeannot, D. Wang, V. Haese-Coat, "Binary image representation and coding by a double-recursive morphological algorithm", Signal Processing: Image Communication, Vol. 8, 1996, pp. 241-266.

[Kala97] P. Kalapathy, "Hardware-software interactions on Mpact", IEEE Micro, March-April 1997, pp. 20-26.

[Kant90] I. Kanter, "Synchronous or asynchronous parallel dynamics. Which is more efficient?", Physica D, Vol. 42, 1990, pp. 273-280.

[Kata97] H. Katata, N. Ito, T. Aono, H. Kusao, "Object wavelet transform for coding of arbitrarily shaped image segments", IEEE Transactions on Circuits and Systems for Video Technology, special issue on MPEG-4, Vol. 7, No. 1, February 1997, pp. 234-237.

[Kim91] K. Kim, J. Kim, T. Kim, "Block arithmetic coding of contour images", Proceedings SPIE Visual Communications and Image Processing: Visual Communication, Vol. 1605, 1991, pp. 851-862.

[Klee87] L. Kleeman, A. Cantoni, "Metastable behavior in digital systems", IEEE Design & Test of Computers, December 1987, pp. 4-19.

[Kons92] K. Konstantinides, V. Bhaskaran, "Monolithic architectures for image processing and compression", IEEE Computer Graphics and Applications, November 1992, pp. 75-86.

[Krik97] A. Krikelis, C. Weems, Associative Processing and Processors, IEEE Computer Society, 1997.

[Kuma95] M. Kumanoya, T. Ogawa, K. Inoue, "Advances in DRAM interfaces", IEEE Micro, December 1995, pp. 30-36.

[Kung87] S. Y. Kung, S. C. Lo, S. N. Jean, J. N. Hwang, "Wavefront array processors - concept to implementation", IEEE Computer, July 1987, pp. 18-33.

[Kung88] S. Y. Kung, VLSI Array Processors, Prentice Hall, 1988.

[Kunt85] M. Kunt, A. Ikonomopoulos, M. Kocher, "Second-generation image-coding techniques", Proceedings of the IEEE, Vol. 73, No. 4, April 1985, pp. 549-574.

[Kunt87] M. Kunt, M. Bénard, R. Leonardi, "Recent results in high-compression image coding", IEEE Transactions on Circuits and Systems, Vol. 34, No. 11, November 1987, pp. 1306-1336.

[Lant82] C. Lantuéjoul, "Geodesic segmentation", in Multicomputers and Image Processing - Algorithms and Programs, K. Preston, L. Uhr, Academic Press, 1982, pp. 111-124.

[Laro92] Le Petit Larousse illustré, Larousse, 1992.

[Laur97] C. Laurent, C. Bouville, "Algorithme parallèle efficace pour la reconstruction morphologique sur architecture MVP", Actes des Journées d'études et d'échanges: COmpression et REprésentation des Signaux Audiovisuels, CNET, Issy-les-Moulineaux, France, Mars 1997.

[Lava94] F. Lavagetto, S. Curinga, "Object-oriented scene modeling for interpersonal video communication at very low bit-rate", Signal Processing: Image Communication, Vol. 6, 1994, pp. 379-395.

[Le97] T. M. Le, W. M. Snelgrove, S. Panchanathan, "Computational RAM implementation of MPEG-2 for real-time encoding", Proceedings SPIE Multimedia Hardware Architectures, Vol. 3021, 1997, pp. 182-192.

[Lea91] R. M. Lea, I. Jalowiecki, "Associative massively parallel computers", Proceedings of the IEEE, Vol. 79, No. 4, April 1991, pp. 469-478.

[Lee94] B. Lee, A. R. Hurson, "Dataflow architectures and multithreading", IEEE Computer, August 1994, pp. 27-39.

[Lee95] R. Lee, "Accelerating multimedia with enhanced microprocessors", IEEE Micro, April 1995, pp. 22-32.

[Lee96] S.-Y. Lee, K. G. Lee, "Synchronous and asynchronous parallel simulated annealing with multiple Markov chains", IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 10, October 1996, pp. 993-1008.

[Lee96a] R. Lee, M. Smith, "Media processing: a new design target", IEEE Micro, August 1996, pp. 6-9.

[Lee96b] R. Lee, "Subword parallelism with MAX-2", IEEE Micro, August 1996, pp. 51-59.

[LeGa91] D. Le Gall, "MPEG: a video compression standard for multimedia applications", Communications of the ACM, Vol. 34, No. 4, April 1991, pp. 46-58.

[Lewa96] G. Lewandowski, A. Condon, E. Bach, "Asynchronous analysis of parallel dynamic programming algorithms", IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 4, April 1996, pp. 425-438.

[Li93] H. Li, Low bitrate image sequence coding, Linköping Studies in Science and Technology, Dissertation No. 318, Linköping University, Sweden, 1993.

[Li94] H. Li, "Image sequence coding at very low bitrates: a review", IEEE Transactions on Image Processing, Vol. 3, No. 5, September 1994, pp. 589-608.

[Li95] W. Li, V. Bhaskaran, M. Kunt, "Very low bit-rate video coding with DFD segmentation", Signal Processing: Image Communication, Vol. 7, 1995, pp. 419-434.

[Li97] W. Li, H. Cao, S. Li, F. Ling, S. Segan, H. Sun, J. Wus, Y.-Q. Zhang, "A video coding algorithm using vector-based techniques", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 7, No. 1, February 1997, pp. 146-157.

[Luba86] B. Lubachevsky, D. Mitra, "A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius", Journal of the ACM, Vol. 33, No. 1, January 1986, pp. 130-150.

[Lync96] N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, 1996.

[Mara86] P. Maragos, R. Schafer, "Morphological skeleton representation and coding of binary images", IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 34, No. 5, October 1986, pp. 1228-1244.

[Mare88] M. Maresca, M. Lavin, H. Li, "Parallel architectures for vision", Proceedings of the IEEE, Vol. 76, No. 8, August 1988, pp. 970-981.

[Marq93] F. Marqués, J. Sauleda, A. Gasull, "Shape and location coding for contour images", Proceedings of the Picture Coding Symposium, 1993.

[Mart87] A. Martin, "Self-timed FIFO: an exercise in compiling programs into VLSI circuits", in From HDL descriptions to guaranteed correct circuit designs, edited by D. Borrione, Elsevier Science Publishers, North-Holland, 1987, pp. 133-153.

[Mart90] A. Martin, "Programming in VLSI: from communicating processes to delay-insensitive circuits", in Developments in Concurrency and Communication, edited by C.A.R. Hoare, University of Texas Year of Programming Series, Addison-Wesley, 1990, pp. 1-64.

[Mart90a] A. Martin, "The limitations to delay-insensitivity in asynchronous circuits", Proceedings of the MIT Conference on Advanced Research in VLSI, MIT Press, 1990, pp. 263-278.

[Mart93] A. Martin, Synthesis of asynchronous VLSI circuits, Technical Report CALTECH-CS-TR-93-28, California Institute of Technology, 1993.

[Meng89] T. Meng, R. Brodersen, D. Messerschmitt, "Automatic synthesis of asynchronous circuits from high-level specifications", IEEE Transactions on Computer-Aided Design, Vol. 8, No. 11, November 1989, pp. 1185-1205.

[Méri91] A. Mérigot, Y. Ni, F. Devos, "Architectures massivement parallèles pour la vision artificielle", Annales des Télécommunications, Vol. 46, No. 1-2, 1991, pp. 78-89.

[Mess96] D. Messerschmitt, "The convergence of telecommunications and computing: what are the implications today ?", Proceedings of the IEEE, Vol. 84, No. 8, August 1996, pp. 1167-1186.

[Meye90] F. Meyer, S. Beucher, "Morphological Segmentation", Journal of Visual Communication and Image Representation, Vol. 1, No. 1, September 1990, pp. 21-46.

[Mitr87] D. Mitra, "Asynchronous relaxations for the numerical solution of differential equations by parallel processors", SIAM J. Sci. Stat. Comput., Vol. 8, No. 1, January 1987, pp. s43-s58.

[Morr94] T. Morris, S. DeWeerth, "Analog VLSI arrays for morphological image processing", Proceedings of the International Conference on Application Specific Array Processors, 1994, pp. 132-142.

[Mura97] K. Murakami, S. Shirakawa, H. Miyajima, "Parallel processing RAM chip with 256Mb DRAM and quad processors", Digest of Technical Papers, International Solid-State Circuits Conference, IEEE, 1997, pp. 228-229.

[Musm89] H. Musmann, M. Hötter, J. Ostermann, "Object-oriented analysis-synthesis coding of moving images", Signal Processing: Image Communication, Vol. 1, 1989, pp. 117-138.

[Nade95] K. Nadehara, I. Kuroda, M. Daito, T. Nakayama, "Low-power multimedia RISC", IEEE Micro, December 1995, pp. 20-29.

[Naka78] Y. Nakagawa, A. Rosenfeld, "A note on the use of local min and max operations in digital picture processing", IEEE Transactions on Systems, Man, and Cybernetics, Vol. 8, No. 8, August 1978, pp. 632-635.

[Naka81] K. Nakamura, "Synchronous to asynchronous transformation of polyautomata", Journal of Computer and System Sciences, Vol. 23, 1981, pp. 22-37.

[Netr95] A. Netravali, A. Lippman, "Digital television: a perspective", Proceedings of the IEEE, Vol. 83, No. 6, June 1995, pp. 834-842.

[Nils97] F. Nilsson, P.-E. Danielsson, "Finding the minimal set of maximum disks for binary objects", Graphical Models and Image Processing, Vol. 59, No. 1, January 1997, pp. 55-60.

[Nodi91] M. Nodine, D. Lopresti, J. Vitter, "I/O overhead and parallel VLSI architectures for lattice computations", IEEE Transactions on Computers, Vol. 40, No. 7, July 1991, pp. 843-852.

[Oddo93] C. Oddou, P. Riglet, "MPEG-4: le futur standard pour le codage à très bas débit", Revue Annuelle LEP, 1993, pp. 17-21.

[Ofek97] E. Ofek, E. Shilat, A. Rappoport, M. Werman, "Multiresolution textures from image sequences", IEEE Computer Graphics and Applications, March-April 1997, pp. 18-29.

[OGor88] L. O'Gorman, "Primitives chain code", Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, 1988, pp. 792-795.

[Okaz95] S. Okazaki, Y. Fujita, N. Yamashita, "A compact real-time vision system using integrated memory array processor architecture", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 5, No. 5, October 1995, pp. 446-452.

[Okub95] S. Okubo, "Reference model methodology- a tool for the collaborative creation of video coding standards", Proceedings of the IEEE, Vol. 83, No. 2, February 1995, pp. 139-150.

[Oste94a] J. Ostermann, "Object-based analysis-synthesis coding based on the source model of moving rigid 3D objects", Signal Processing: Image Communication, Vol. 6, 1994, pp. 143-161.

[Oste94b] J. Ostermann, "Object-based analysis-synthesis coding (OBASC) based on the source model of moving flexible 3-D objects", IEEE Transactions on Image Processing, Vol. 3, No. 5, September 1994, pp. 705-711.

[Pard94] M. Pardàs, P. Salembier, "3D morphological segmentation and motion estimation for image sequences", Signal Processing, Vol. 38, 1994, pp. 31-43.

[Patt97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick, "A case for intelligent RAM", IEEE Micro, March/April 1997, pp. 34-43.

[Pear95] D. Pearson, "Developments in model-based video coding", Proceedings of the IEEE, Vol. 83, No. 6, June 1995, pp. 892-906.

[Pele96] A. Peleg, U. Weiser, "MMX technology extension to the Intel architecture", IEEE Micro, August 1996, pp. 42-50.

[Pele97] A. Peleg, S. Wilkie, U. Weiser, "Intel MMX for multimedia PCs", Communications of the ACM, Vol. 40, No. 1, January 1997, pp. 25-38.

[Pere96] F. Pereira, R. Koenen, "Very low bit-rate audio-visual applications", Signal Processing: Image Communication, Vol. 9, 1996, pp. 55-77.

[Pika97] A. Pikaz, A. Averbuch, "An efficient topological characterization of gray-levels textures, using a multiresolution representation", Graphical Models and Image Processing, Vol. 59, No. 1, January 1997, pp. 1-17.

[Pirs95] P. Pirsch, N. Demassieux, W. Gehrke, "VLSI architectures for video compression - a survey", Proceedings of the IEEE, Vol. 83, No. 2, February 1995, pp. 220-246.

[Pita90] I. Pitas, A. Venetsanopoulos, "Morphological shape decomposition", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 12, No. 1, January 1990, pp. 38-45.

[Plan94] P. Planet, Algorithmes et architectures cellulaires pour le traitement d'images, Thèse de Doctorat de l'I.N.P.G., Grenoble, France, Décembre 1994.

[Port84] T. Porter, T. Duff, "Compositing digital images", Computer Graphics, Vol. 18, No. 3, July 1984, pp. 253-259.

[Pott92] J. Potter, Associative Computing - a programming paradigm for massively parallel computers, Plenum Press, 1992.

[Poul92] J. Poulton, J. Eyles, S. Molnar, H. Fuchs, "Breaking the frame-buffer bottleneck with logic-enhanced memories", IEEE Computer Graphics and Applications, November 1992, pp. 65-74.

[Pres84] K. Preston, M. Duff, Modern Cellular Automata - Theory and Applications, Plenum Press, 1984.

[Priv93a] G. Privat, Algorithmique et architecture parallèles en VLSI, Notes de cours, ENST de Bretagne, Option Microélectronique et Conception de Systèmes, 1993.

[Priv93b] G. Privat, P. Planet, M. Renaudin, "Asynchronous relaxation of locally-coupled automata networks, with application to parallel VLSI implementation of iterative image processing algorithms", Proceedings of the International Conference on Application Specific Array Processors, October 1993.

[Priv95] G. Privat, F. Robin, M. Renaudin, B. El Hassan, "A fine-grain asynchronous VLSI cellular array processor architecture", Proceedings of the International Symposium on Circuits And Systems, Seattle, April 1995.

[Przy96] S. Przybylski, "SDRAMs ready to enter PC mainstream", Microprocessor Report, May 6, 1996, pp. 17-23.

[Purc97] S. Purcell, "Mpact 2 media processor, balanced 2X performance", Proceedings SPIE Multimedia Hardware Architectures, Vol. 3021, 1997, pp. 102-108.

[Rand97] M. Randall, "Talisman: multimedia for the PC", IEEE Micro, March-April 1997, pp. 11-19.

[Rayn92] M. Raynal, Synchronisation et état global dans les systèmes répartis, Collection de la Direction des Etudes et Recherches d'Electricité de France, Eyrolles, 1992.

[Rein96] J. Reinhardt, W. Higgins, "Efficient morphological shape representation", IEEE Transactions on Image Processing, Vol. 5, No. 1, January 1996, pp. 89-101.

[Rena90] M. Renaudin, Architecture VLSI pour le codage d'images, Thèse de Doctorat de l'I.N.P.G., Grenoble, France, Octobre 1990.

[Rena94a] M. Renaudin, B. El Hassan, "The design of fast asynchronous adder structures and their implementation using DCVS logic", Proceedings of the International Symposium on Circuits And Systems, London, 1994.

[Rena94b] M. Renaudin, B. El Hassan, "A minimum power, 100 MHz , 12x18+30-b Multiplier-Accumulator operating in asynchronous and synchronous mode", Proceedings of the European Solid-State CIRcuits Conference, Ulm, Germany, September 1994.

[Rena96] M. Renaudin, B. El Hassan, A. Guyot, "A new asynchronous pipeline scheme: application to the design of a self-timed ring divider", IEEE Journal of Solid-State Circuits, Vol. 31, No. 7, July 1996, pp. 1001-1013.

[Rena96a] M. Renaudin, "AAAA: asynchronisme et adéquation algorithme-architecture", Conférence invitée aux Journées Adéquation Algorithme Architecture en traitement du signal et images, CNES Toulouse, France, Janvier 1996.

[Rena97] M. Renaudin, F. Robin, P. Vivet, "AAAA: asynchronisme et adéquation algorithme-architecture", soumis à la revue Traitement du Signal.

[Rijk95] K. Rijkse, "ITU standardisation of very low bitrate video coding algorithms", Signal Processing: Image Communication, Vol. 7, 1995, pp. 553-565.

[Robe97] G. Robert, J.-M. Chassery, "Détection de mouvement par maillages polygonaux pour une extension des I.F.S. à la vidéo", Actes des Journées d'études et d'échanges: COmpression et REprésentation des Signaux Audiovisuels, CNET, Issy-les-Moulineaux, France, Mars 1997.

[Robi95] F. Robin, G. Privat, M. Renaudin, "Asynchronous relaxation of morphological operators: a joint architecture-algorithm perspective", Proceedings of the International Workshop on Parallel Image Analysis, Lyon, France, December 1995.

[Robi96a] F. Robin, G. Privat, M. Renaudin, N. Van Den Bossche, "Une architecture VLSI cellulaire massivement parallèle pour la relaxation asynchrone d'opérateurs morphologiques", Actes des Journées Adéquation Algorithme Architecture en traitement du signal et images, CNES Toulouse, France, Janvier 1996, pp. 91-97.

[Robi96b] F. Robin, M. Renaudin, G. Privat, "An asynchronous 16*16 pixel array-processor for morphological filtering of greyscale images", Proceedings of the European Solid-State Circuits Conference, Neuchâtel, Switzerland, September 1996, pp. 188-191.

[Robi96c] F. Robin, M. Renaudin, G. Privat, N. Van Den Bossche, "Functionally asynchronous array-processor for morphological filtering of greyscale images", IEE Proceedings on Computers and Digital Techniques, special section on Asynchronous Architecture, Vol. 143, No. 5, September 1996, pp. 273-281.

[Robi97a] F. Robin, G. Privat, M. Renaudin, "Asynchronous relaxation of morphological operators: a joint algorithm-architecture perspective", International Journal on Pattern Recognition and Artificial Intelligence, Vol. 11, No. 7, World Scientific Publishing, 1997.

[Robi97b] F. Robin, M. Renaudin, G. Privat, N. Van Den Bossche, "Un réseau cellulaire VLSI fonctionnellement asynchrone pour le filtrage morphologique d'images", soumis à la revue Traitement du Signal.

[Rönn96] K. Rönner, J. Kneip, "Architecture and applications of the HiPAR video signal processor", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 6, No. 1, February 1996, pp. 56-66.

[Rons91] C. Ronse, B. Macq, "Morphological shape and region description", Signal Processing, Vol. 25, 1991, pp. 91-105.

[Rose78] A. Rosenfeld, "Iterative methods in image analysis", Pattern Recognition, Vol. 10, Pergamon Press, 1978, pp. 181-187.

[Rose82] A. Rosenfeld, A. Kak, Digital Picture Processing, Vol. 2, Academic Press, 1982.

[Rose83] A. Rosenfeld, "Parallel image processing using cellular arrays", IEEE Computer, January 1983, pp. 14-20.

[Rose88] F. Rosenberger, C. Molnar, T. Chaney, T. Fang, "Q-modules : internally clocked delay-insensitive modules", IEEE Transactions on Computers, Vol. 37, No. 9, September 1988.

[Saka95] M. Sakamoto, S. Ishikawa, Y. Saiki, T. Kizaki, "Graphics memories for multimedia", Hitachi Review, Vol. 44, No. 6, 1995, pp. 325-328.

[Sale92] P. Salembier, J. Serra, "Morphological multiscale image segmentation", Proceedings SPIE Visual Communications and Image Processing, Vol. 1818, 1992, pp. 620-631.

[Sale94] P. Salembier, M. Pardàs, "Hierarchical morphological segmentation for image sequence coding", IEEE Transactions on Image Processing, Vol. 3, No. 5, September 1994, pp. 639-651.

[Sale95a] P. Salembier, L. Torres, F. Meyer, C. Gu, "Region-based video coding using mathematical morphology", Proceedings of the IEEE, Vol. 83, No. 6, June1995, pp. 843-857.

[Sale95b] P. Salembier, J. Serra, "Flat zones filtering, connected operators, and filters by reconstruction", IEEE Transactions on Image Processing, Vol. 4, No. 8, August 1995, pp. 1153-1160.

[Sale96] P. Salembier, P. Brigger, J. Casas, M. Pardàs, "Morphological operators for image and video compression", IEEE Transactions on Image Processing, Vol. 5, No. 6, June 1996, pp. 881-898.

[Sale97] P. Salembier, F. Marqués, M. Pardàs, J. Ramon Morros, I. Corset, S. Jeannin, L. Bouchard, F. Meyer, B. Marcotegui, "Segmentation-based video coding system allowing the manipulation of objects", IEEE Transactions on Circuits and Systems for Video Technology, special issue on MPEG-4, Vol. 7, No. 1, February 1997, pp. 60-74.

[Sapi94] G. Sapiro, D. Malah, "Morphological image coding based on a geometric sampling theorem and a modified skeleton representation", Journal of Visual Communication and Image Representation, Vol. 5, No. 1, March 1994, pp. 29-40.

[Schä95] R. Schäfer, T. Sikora, "Digital video coding standards and their role in video communications", Proceedings of the IEEE, Vol. 83, No. 6, June 1995, pp. 907-924.

[Schi93] H. Schiller, M. Hötter, "Investigations on colour coding in an object-oriented analysis-synthesis coder", Signal Processing: Image Communication, Vol. 5, 1993, pp. 319-326.

[Schi96] A. Schilling, G. Knittel, W. Strasser, "Texram: a smart memory for texturing", IEEE Computer Graphics and Applications, May 1996, pp. 32-41.

[Seit84] C. Seitz, "Concurrent VLSI architectures", IEEE Transactions on Computers, Vol. 33, No. 12, December 1984, pp. 1247-1265.

[Serr86] J. Serra, "Introduction to mathematical morphology", Computer Vision, Graphics, and Image Processing, Vol. 35, 1986, pp. 283-305.

[Serr89] J. Serra, Image Analysis and Mathematical Morphology, Vol. 1, Academic Press, 1989.

[Shu95] W. Shu, M.-Y. Wu, "Asynchronous problems on SIMD parallel computers", IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 7, July 1995, pp. 704-713.

[Siko95a] T. Sikora, "Low complexity shape-adaptive DCT for coding of arbitrarily shaped image segments", Signal Processing: Image Communication, Vol. 7, 1995, pp. 381-395.

[Siko95b] T. Sikora, B. Makai, "Shape-adaptive DCT for generic coding of video", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 5, No. 1, February 1995, pp. 59-62.

[Siko95c] T. Sikora, S. Bauer, B. Makai, "Efficiency of shape-adaptive 2-D tranforms for coding of arbitrarily shaped image segments", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 5, No. 3, June 1995, pp. 254-258.

[Siko97] T. Sikora, "The MPEG-4 video standard verification model", IEEE Transactions on Circuits and Systems for Video Technology, special issue on MPEG-4, Vol. 7, No. 1, February 1997, pp. 19-31.

[Stal88] W. Stallings, "Reduced instruction set computer architecture", Proceedings of the IEEE, Vol. 76, No. 1, January 1988, pp. 38-55.

[Ster86] S. Sternberg, "Grayscale morphology", Computer Vision, Graphics, and Image Processing, Vol. 35, 1986, pp. 333-355.

[Stor92] R. Storer, M. Pout, A. Thomson, E. Dagless, A. Duller, A. P. Marriott, P. Hicks, "An associative processing module for a heterogeneous vision architecture", IEEE Micro, Vol. 12, No. 3, June 1992, pp. 42-55.

[Suth89] I. Sutherland, "Micropipelines", Communications of the ACM, Vol. 32, No. 6, June 1989, pp. 720-738.

[Tan95] W.-C. Tan, T. Meng, "A low-power high performance polygon renderer for computer graphics", Journal of VLSI Signal Processing, Vol. 9, 1995, pp. 233-255.

[Toff87] T. Toffoli, N. Margolus, Cellular Automata Machines, MIT Press, 1987.

[Torb96] J. Torborg, J. Kajiya, "Talisman: commodity realtime 3D graphics for the PC", Proceedings SIGGRAPH, ACM, 1996, pp. 353-363.

[Tred95] N. Tredennick, "Technology and business: forces driving microprocessor evolution", Proceedings of the IEEE, Vol. 83, No. 12, December 1995, pp. 1641-1652.

[Tred96] N. Tredennick, "Microprocessor-based computers", IEEE Computer, October 1996, pp. 27-37.

[Trem96] M. Tremblay, J. M. O'Connor, V. Narayanan, L. He, "VIS speeds new media processing", IEEE Micro, August 1996, pp. 10-20.

[Turl96] J. Turley, "Multimedia chips complicate choices", Microprocessor Report, February 12, 1996, pp. 14-19.

[Undy94] S. Undy, M. Bass, D. Hollenbeck, W. Kever, L. Thayer, "A low-cost graphics and multimedia workstation chip set", IEEE Micro, April 1994, pp. 10-22.

[Üres89] A. Üresin, M. Dubois, "Sufficient conditions for the convergence of asynchronous iterations", Parallel Computing, Vol. 10, 1989, pp. 83-92.

[Üres96] A. Üresin, M. Dubois, "Effects of asynchronism on the convergence rate of iterative algorithms", Journal of Parallel and Distributed Computing, Vol. 34, 1996, pp. 66-81.

[VanD95] N. Van Den Bossche, Etude et conception d'un array-processor cellulaire asynchrone, Tomes I et II, Rapport de stage, CNET, Grenoble, France, Juin et Septembre 1995.

[Vinc91] L. Vincent, P. Soille, "Watersheds in digital spaces: an efficient algorithm based on immersion simulations", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 13, No. 6, June 1991, pp. 583-597.

[Vinc93] L. Vincent, "Morphological grayscale reconstruction in image analysis: applications and efficient algorithms", IEEE Transactions on Image Processing, Vol. 2, No. 2, April 1993, pp. 176-201.

[Webb96] M. Webb, "Integration challenges PC architects - as transistor counts head for 100 million, new designs are needed", Microprocessor Report, July 8, 1996, pp. 13-15.

[Weem91] C. Weems, "Architectural requirements of image understanding with respect to parallel processing", Proceedings of the IEEE, Vol. 79, No. 4, April 1991, pp. 537-547.

[Wels90] W. Welsh, S. Searby, J. Waite, "Model-based image coding", British Telecom Technology Journal, Vol. 8, No. 3, July 1990, pp. 94-106.

[Will91] P. Willemin, T. Reed, M. Kunt, "Image sequence coding by split and merge", IEEE Transactions on Communications, Vol. 39, No. 12, December 1991, pp. 1845-1854.

[Wu96] L. Wu, J. Benois-Pineau, P. Delagnes, D. Barba, "Spatio-temporal segmentation of image sequences for object-oriented low bit-rate image coding", Signal Processing: Image Communication, Vol. 8, 1996, pp. 513-543.

[Xu96] C. Xu, F. Lau, "Efficient termination detection for loosely synchronous applications in multicomputers", IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 5, May 1996, pp. 537-544.

[Yama94] N. Yamashita, T. Kimura, Y. Fujita, Y. Aimoto, T. Manabe, S. Okazaki, K. Nakamura, M. Yamashina, "A 3.84 GIPS integrated memory array processor with 64 processing elements and a 2-Mb SRAM", IEEE Journal of Solid-State Circuits, Vol. 29, No. 11, November 1994, pp. 1336-1343.

[Yao96a] Y. Yao, "PC graphics reach new level: 3D", Microprocessor Report, January 22, 1996, pp. 14-19.

[Yao96b] Y. Yao, "Competition heats up in 3D accelerators", Microprocessor Report, March 5, 1996, pp. 16-23.

[Yao96c] Y. Yao, "Samsung launches media processor", Microprocessor Report, Vol. 10, No. 11, August 26, 1996, pp. 1, 6-9.

[Yao96d] Y. Yao, "Chromatic's Mpact 2 boosts 3D", Microprocessor Report, Vol. 10, No. 15, November 18, 1996, pp. 1, 6-10.

[Yate95] R. Yates, N. Thacker, S. Evans, S. Walker, P. Ivey, "An array processor for general purpose digital image compression", IEEE Journal of Solid-State Circuits, Vol. 30, No. 3, March 1995, pp. 244-249.

[Yoko95] Y. Yokoyama, Y. Miyamoto, M. Ohta, "Very low bit rate video coding using arbitrarily shaped region-based motion compensation", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 5, No. 6, December 1995, pp. 500-507.

ANNEXES

- photomicrographie du circuit AMPHIN (§4.5)
- programmes en assembleur (§5.4)
    - "produit.asm"
    - "recons_iter.asm"
    - "recons_bloc.asm"
    - "recons_new.asm"
    - "rotate.asm"
- visualisation du plan mémoire destination au cours de l'exécution de l'algorithme de rotation (§5.4.4)
- schéma niveau portes de l'interface de communication non-bloquante (§5.5.2.b)