HAL
open science

SMAL_A31 pour une architecture SIMD reconfigurable

Belkacem Zerrouk

To cite this version:

Belkacem Zerrouk. SMAL_A31 pour une architecture SIMD reconfigurable. [Rapport de recherche] RR-1509, INRIA. 1991. inria-00075053

HAL Id: inria-00075053

https://inria.hal.science/inria-00075053v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports de Recherche

N° 1509

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

SMAL_A31 POUR UNE ARCHITECTURE SIMD RECONFIGURABLE

belkacem ZERROUK

Septembre 1991

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

RR_1509

SMAL_A31

POUR UNE ARCHITECTURE SIMD RECONFIGURABLE

SMAL_A31

FOR A RECONFIGURABLE SIMD ARCHITECTURE

Belkacem ZERROUK

Projet ARCHI
INRIA-Rocquencourt
Domaine de Voluceau
BP. 105 78153 LE CHESNAY CEDEX

Résumé

L'autonomie d'opération et la reconfigurabilité topologique constituent une bonne alternative pour l'amélioration des performances d'un réseau SIMD en grille: diminution de l'oisiveté des processeurs élémentaires, et choix topologiques mieux adaptés à l'application à réaliser.

Nous présentons dans ce rapport une nouvelle version de processeur élémentaire d'une architecture SIMD massivement parallèle, qui implémente d'une manière originale ces deux mécanismes. Ce processeur élémentaire offre aussi un moyen de recouvrir les communications avec les calculs locaux.

Abstract

Operation autonomy and reconfigurability of the interconnection network are two key factors enhancing the performance of a SIMD grid architectures: well mapped topologies can be established for a given application, and idleness of the processing elements can be reduced.

A new processing element of a massively parallel SIMD architecture, with original implementation of these mechanisms is presented. Communication and local computations overlap is also possible.

1. INTRODUCTION

De nombreux chercheurs se sont penchés sur les moyens d'améliorer les architectures SIMD en grille ou en tore dans le but d'atteindre des performances plus élevées pour une large gamme d'applications. Le point critique dans de telles architectures est purement connectique. Certaines conceptions proposées dans la littérature, ont pour point commun de vouloir préserver certains avantages d'une architecture matricielle: A savoir son aspect topologique planaire qui facilite énormément la réalisation de telles machines. En effet, la conception d'une machine parallèle est assujettie à plusieurs critères de choix et en termes de VLSI, il est intéressant de prendre en compte le coût en surface, donc le coût du support matériel de l'architecture visée [1]. Si l'on compare de ce point de vue une architecture cubique, la Connection Machine [2] par exemple , à une architecture matricielle, la machine BLITZEN [3], le coût des connexions est un facteur dominant, dans le premier cas.

Parmi les améliorations proposées pour une architecture en grille, on suggère l'adjonction de un ou de plusieurs bus [4,5] ou encore une imbrication hiérarchique de bus [6], afin de répondre aux besoins des communications globales. Des algorithmes ont été proposés pour ces architectures dites "augmentées". L'efficacité de l'introduction de tels bus est souvent liée dans ce cas au principe "diviser pour régner". Une solution qui nous semble plus intéressante, aussi bien en implantation VLSI qu'en algorithmique, serait l'introduction d'une autonomie pour les interconnexions. Le réseau "polymorphique" de Maresca et al [7,8,9] en est l'exemple type. Les mécanismes de base sont aussi présents dans l'architecture FAMA, proposée par Alcolea et al [10].

L'autonomie d'interconnexion consiste à donner une possibilité à chaque processeur élémentaire d'envoyer et/ou recevoir des données, depuis/vers des directions déterminées, ou encore de court-cicuiter ses liens périphériques en fonction de "l'état" local de ce processeur. Nous pouvons dégager deux types d'autonomie pour l'interconnexion :

Le premier est présenté dans YUPPIE [7]. Chaque processeur élémentaire possède son propre registre de décision qui permet de choisir la configuration de sa connectique locale, c.à.d liaison Nord, Sud, Est ou Ouest, grâce à deux champs "configuration" de l'instruction.

Le second consiste à positionner des registres de configuration des connexions locales, par des opérations de chargement. Ce second type ne présente pas de handicaps temporels, du fait que l'on utilise des opérateurs sériels 1bit.

SMAL_A31*, une version étendue de SMAL_X31 [11], est un processeur élémentaire conçu dans l'optique de fournir une meilleure flexibilité. Ceci est possible grâce à l'introduction de l'autonomie pour l'interconnexion, en utilisant des registres de configuration internes, et une extension du mode de fonctionnement que l'on retrouve dans SMAL_X31. Le choix de l'autonomie pour les connexions paramètrées localement, nous offre plusieurs degrés de liberté supplémentaire. De plus, cette

* SMAL est acronyme de SIMD Multiprocessor Architecture with a Long instruction word.

solution est une bonne approche à la tolérance aux défaillances et plus particulièrement au sens de Trotter et Moore [12].

L'implantation d'une autonomie pour l'interconnexion, implique l'utilisation d'éléments de court-circuit ("switchs") qui sont temporellement pénalisant. En effet, le temps d'envoi d'une donnée d'un processeur à un autre dépend du nombre de couches logiques traversées (nombre de "switchs"), ou de la distance entre les deux processeurs (émeteur-récepteur). Dans le cas ou l'on utilise des boîtiers regroupant un nombre donné de processeurs, le temps de transmission dépend de la propagation locale dans chaque boîtier et de la propagation inter-boîtiers.

Dans le calcul des temps de propagation, c'est le temps de propagation inter-boîtier qui représente la composante principale. Afin de remédier à ce problème, nous avons deux possibilités :

L'une consiste à faire une intégration totale du réseau en utilisant une technologie WSI assez rapide, à coûts élevés.

La seconde consiste à se limiter à des réseaux de taille raisonnable, de manière à ce que la dispersion des temps de propagation d'une configuration à une autre soit minime et à des coûts acceptables.

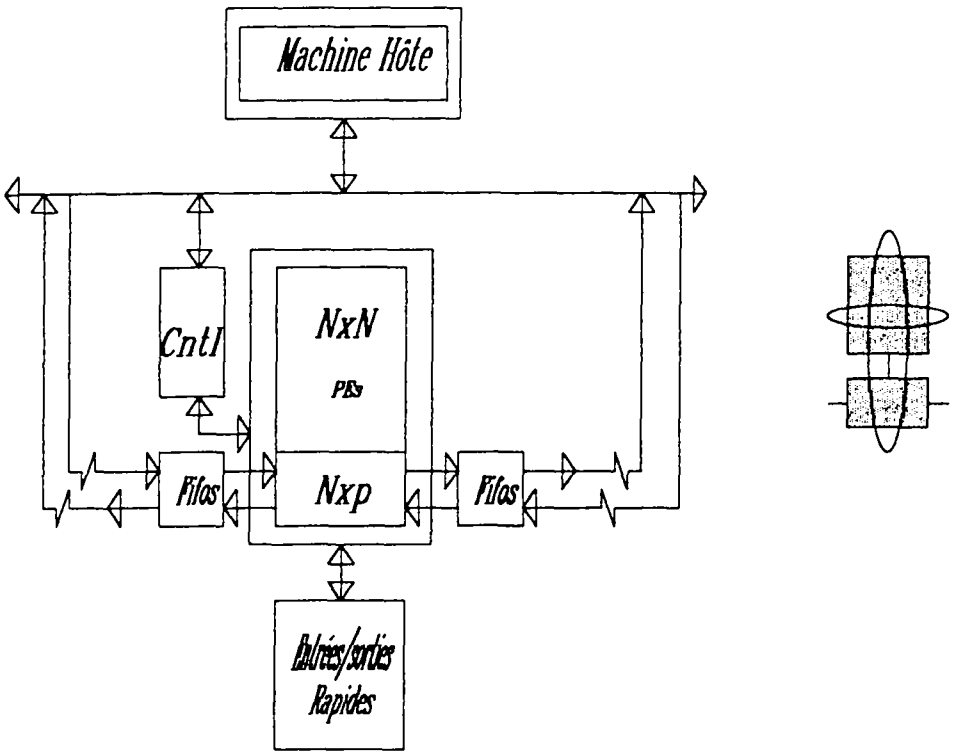

Figure 1: l'architecture SMAL.

Dans l'optique de l'architecture SMAL [13] nous avons choisi une approche résultant d'un compromis entre différents points de conception à savoir:

-Le coût de la machine.
-La flexibilité et l'extensibilité architecturale.
-Les performances de la machine.

Les deux premiers points sont directement liés à la possibilité de fournir un outil de recherche à un prix raisonnable. Le facteur extensibilité jouant un rôle important dans ce cas. Par conséquent, il est nécessaire d'avoir une conception modulaire. SMAL répond à ces deux exigeances en mettant en avant l'utilisation de modules matriciels de "faibles" tailles, pouvant être chainés de manière à former des architectures linéaires (fig.1). Ainsi, nous gagnons en simplicité, conjuguée à une grande souplesse d'utilisation offerte par les réseaux en grille et les réseaux linéaires.

Les performances de la machine dépendent de l'architecture des éléments et de la technologie de fabrication. L'architecture du processeur élémentaire SMAL_A31 (comme son prédécésseur SMAL_X31) vise à rentabiliser le réseau par la cohabitation de plusieurs opérateurs arithmétiques et logiques dans le même chemin de données. Dans l'article [13], nous avons abordé la possibilité d'une optimisation du code par cette approche, vu l'existence d'un parallélisme local dans chaque processeur élémentaire. Ainsi nous débouchons sur une architecture peu famillière, qui est l'application du concept **VLIW** pour des réseaux SIMD sériels.

La suite de ce rapport est composée de 4 parties. La partie 2 donne une vue architecturale du processeur SMAL_A31, la partie 3 traitera l'instruction du processeur élémentaire, la partie 4 cerne les éléments liés à l'autonomie pour les opérations, la partie 5 explicite l'autonomie pour les connexions. Nous donnerons dans la partie 6 un aspect de modélisation sur l'autonomie d'interconnexion et de ce fait nous donnerons certaines possibilité de l'architecture SMAL. Des exemples d'applications sont donnés dans la partie 7. Nous explicitons l'influence temporelle de l'autonomie d'interconnexion sur l'exécution d'une application dans la partie 8.

2. ARCHITECTURE DE SMAL_A31

2.1. INTERFACE

SMAL_A31 constitue la brique de base d'un réseau de processeurs élémentaires. Vu de l'extérieur et en considérant un seul processeur élementaire, SMAL_A31 se présente comme une boîte noire ayant 37 broches d'entrées/sorties, en plus des alimentations (fig.2).

Les broches sont affectées comme suit:

.Une entrée d'horloge (CLK),
.Un signal de remise à zéro (RESET),
.Une sortie phase d'exécution (RWB),
.Seize lignes d'instructions,
.Quatre entrées voisinage (WRP,ELP,SRP et NLP),
.Quatre sorties voisinage (West,East,North, South),

.Une entrée plan d'E/S (Din),
.Une sortie plan d'E/S (Dout),
.Une entrée de contrôle du plan d'E/S de données (Prop),
.Une sortie OU global (Sum_or),
.Deux entrées de données globales (GLR et GLL),
.Deux ports de lecture mémoire (MBR et MBL),
.Deux ports de lecture/ecriture mémoire (MAR et MAL).

2.2. ARCHITECTURE INTERNE

En considérant un processeur élémentaire par boitier, SMAL_A31 est composé principalement de 6 sous-circuits, comme le montre la figure 2. Les différents sous-blocs sont respectivement : une unité de contrôle, une partie opérative inférieure (POI), une partie opérative supérieure (POS), une unité de décision (DU), un circuit d'entrées/sorties de données (IOU), et une unité de communication (EU). Nous donnons dans ce qui suit une description de chacun de ces sous-blocs.

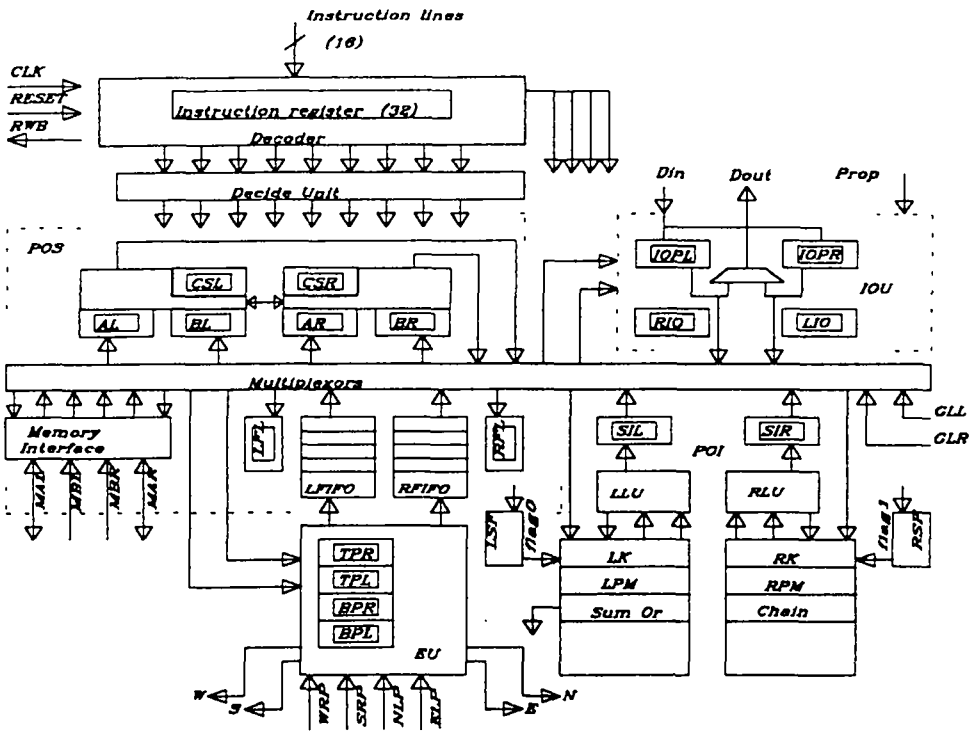

Figure 2: SMAL_A31.

2.2.1. L'UNITE DE DECODAGE

Le circuit de décodage est composé d'un registre instructions sur 32 bits et d'un décodeur. En sortie du bloc, 68 micro-commandes assurent le contrôle des autres sous-blocs cités ci-dessus.

2.2.2. LA PARTIE OPERATIVE INFERIEURE

La POI est constituée de deux parties, gauche et droite, identiques. La partie de gauche (resp. droite) est formée d'un banc de 8 registres LB (resp. RB), d'une unité logique LLU (resp.RLU), et d'un circuit de sélection d'indicateurs LSF (resp. RSF). S'ajoute à cela, une interface avec la partie opérative supérieure (POS).

Les bancs de registres ont deux ports de lecture et un port d'écriture chacun. Dans chaque banc, les trois premier registres numéros 0 à 2 sont spécialisés. Nous les décrivons brièvement ci-dessous:

.Dans le banc de gauche nous avons un bit de condition/masque d'exécution (LK), un bit pour le OU_global et un masque d'écriture de la file de gauche de la POS (LPM).

.Dans le banc de droite nous avons un bit de condition/masque d'exécution (RK), un bit de chainage des deux unités arithmétiques et logiques de la POS (CHAIN) et un masque d'écriture de la file de droite de la POS (RPM).

Pour plus de clarté nous reviendrons sur l'interface dans le paragraphe suivant.

2.2.3. LA PARTIE OPERATIVE SUPERIEURE

La partie opérative supérieure est composée de deux parties gauche et droite identiques, regroupant deux unités aritmétiques et logiques (RALU et LALU), deux files LFIFO et RFIFO, et d'une interface mémoire. Chaque partie peut exécuter sa propre opération.

L'environnement de chaque unité arithmétique et logique est constitué de deux registres d'entrées opérandes (AR et BR pour RALU et AL et BL pour LALU), de multiplexeurs d'entrées et d'un registre retenue (CSR pour RALU et CSL pour LALU).

Sachant que les files sont principalement spécialisées dans la sérialisation de données émises par les voisins, nous avons associé à chacune d'elle un paramètre de rebouclage, permettant une sérialisation locale. Ce paramètre est matérialisé par un registre (LFL pour la file de gauche et RFL pour la file de droite).

L'interface mémoire est un simple mécanisme qui sert d'une part, aux opération d'entrées de données, et d'autre part à masquer l'écriture mémoire lors de l'apparition d'une opération "vide" (NOP).

2.2.4. LE CIRCUIT D'ENTREES/SORTIES

L'unité d'entrées/sorties comprend deux registres, IOPR et IOPL. Ces registres forment les composantes du plan d'entrées/sorties que l'on retrouve dans les architectures habituelles.

Les opérations d'entrées données s'effectuent depuis les registres IOPR et IOPL vers les ports mémoires, MAR et MAL, correspondants. Les sorties données s'effectuent via les unités arithmétiques et logiques.

Une opération d'entrée/sortie peut être masquée: à chaque opération d'entrée/sortie, nous avons associé un registre servant de masque, RIO pour l'opération de droite et LIO pour l'opération de gauche.

2.2.5. L'UNITE DE COMMUNICATION

L'unité de communication à pour rôle de multiplexer les entrées depuis les voisins vers les files locales d'une part, et d'autre part d'envoyer une donnée vers les voisins. L'envoi de chaque bit d'une donnée est précédé par un bit de validation. Chaque file peut recevoir une donnée de l'un des deux ports "voisinage" qui lui sont associés. En effet, la file de gauche peut recevoir soit depuis WRP ou bien depuis SRP, la file de droite peut recevoir soit depuis NLP ou bien depuis ELP.

Le mécanisme d'échange, comme dans la version SMAL_X31, est tel que chaque processeur élémentaire peut émettre simultanément deux données (pouvant être différentes) sur le port nord (ou est) et le port sud (ou ouest). Et, moyennant les files, les communications peuvent s'effectuer en **recouvrement avec les calculs**.

L'unité de communication est paramétrable de façon à assurer une autonomie pour l'interconnexion. Nous avons deux types de paramètres dans cette partie: des paramètres de topologies locales concernant le choix des proches voisins, et des paramètres de "court-circuit". Pour cela, nous avons résérvé deux registres, TPR et TPL pour les premiers et deux registres, BPR et BPL pour les seconds. Dans le paragraphe 6, nous reviendrons en détail sur ce point.

D'après ce qui a précédé, nous pouvons définir deux couples de voisins: les voisins de gauches (Sud et Ouest) et les voisins de droite (Nord et Est).

2.2.6. L'UNITE DE DECISION

L'unité de décision assure l'autonomie pour la réalisation d'opérations du processeur élémentaire. Le mécanisme de décision dans SMAL_A31, est une version étendue de celui implémenté dans le processeur SMAL_X31. Les entrées principales de cette unité sont les lignes décodées des opérations POS, les lignes du champ mode d'exécution de l'instruction et le contenu des registres masques/conditions de la POI.

3. L'INSTRUCTION

Après avoir présenté les ressources de base du processeur élémentaire SMAL_A31, nous consacrons cette partie à la description de l'instruction.

3.1. LE FORMAT DE L'INSTRUCTION ET SEQUENCEMENT

Le fonctionnement du processeur élémentaire est basé sur deux phases d'exécution (RWB=1 et RWB=0). Chaque phase correspond à une période d'horloge.

Sachant que l'instruction SMAL_A31 tient sur 32 bits et, du fait que nous ne disposons que de 16 lignes d'instructions, une décomposition en deux champs de 16 bits est nécessaire. Chaque champ est introduit à la fin d'une phase donnée. Cette décomposition tient compte de la conformité de réalisation des opérations et de l'optimisation temporelle. La taille de l'instruction n'inclut pas les adresses mémoires locales. Ces adresses sont supposées générées par le contrôleur du réseau constitué par des processeurs élémentaires SMAL_A31 [13].

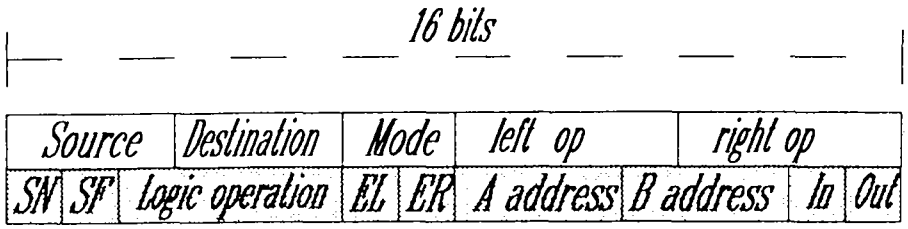

Figure 3: l'instruction SMAL_A31.

3.2. LES DIFFERENTS CHAMPS DE L'INSTRUCTION

L'instruction est composée de 13 champs comme indiqué par la figure 3. Sur cette même figure, la plus petite des cases correspond à un bit. Les champs en blanc désignent les bits que l'on introduit en phase écriture (RWB=0) alors que les champs de couleur désignent les bits que l'on introduit en phase Lecture (RWB=1).

La signification de chacun de ces champs est comme suit:

Le champ **S** désigne les sources d'opérandes pour les opérateurs POS (RALU et LALU). Le champ **D** désigne les destinations des résultats des opérateurs POS. **CC** indique le mode d'exécution des opérations. **OPL** et **OPR** indiquent les opérations à réaliser dans la partie gauche et la partie droite de la POS (cela dépend du mode). **SN** contribue dans la sélection du voisinage et sert aussi de bit d'adresse de destination. **SF** indique l'opération de chargement des registres conditions/masques POI par les valeurs des indicateurs sélectionnés. **OP** désigne l'opération logique commune aux deux unités logiques de la POI. Une opération logique POI peut être inhibée par le bit **EL** ou **ER**, selon que la partie est gauche ou droite. Ces derniers servent aussi à complémenter ou non les indicateurs respectifs séléctionnés. Les champs **AA** et **BB** servent d'une part à l'adressage commun des deux bancs de registres de la POI et d'autre part à sélectionner les indicateurs, au nombre de 8, dont les valeurs peuvent, éventuellement, être chargées dans les registres conditions/masques de la POI. La correspondance est telle que **AA** (resp. **BB**) sélectionne l'indicateur à charger dans le registre condition/masque de gauche (resp. droite). Enfin, le champ **IO** spécifie les opérations d'entrées/sorties de données.

3.3. LES OPERATIONS POS

Les opérations POS, données par les champs **OPR** et **OPL**, se résument en un ensemble d'opérations arithmétiques et logiques, une opération vide (**Nop**) et une

opération de remise à zéro d'une file donnée (**Clr**). Les codes opérations **Nop** et **Clr** sont en recouvrement avec deux opérations UAL ineffectives.

La présence d'un **NOP** inhibe toute altération du contenu des zones destinataires, y compris les registres de retenues. Le même effet se produit pour **Clr** en dehors de la remise à zéro du pointeur de file.

Les opérations POS sont de type:

destination=source1 op source2

Pour chaque opération, les unités arithmétiques et logiques de la POS fournissent deux sorties chacune: la sortie résultat et la sortie retenue. La table 1 donne l'ensemble des opérations ainsi que leur code.

Les sources et destinations de chacune des ALUs de la POS sont données par les tables 2 et 3.

Les sources **rm** et **mr** correspondent aux dérniers résultats de l'opération logique de la POI. Les éléments destinataires dans le cas de **WR** et **IWR** sont le couple des registres de gauche et de droite de la POI adressés par le champ **A**. Le rôle de l'interface POI/POS est, de stocker les résultats de la POI de l'instruction en cours afin d'être utilisés durant l'instruction suivante par la POS et de régler matériellement l'accès aux bus d'écriture des bancs de registres. La façon dont est résolu ce dérnier problème dépend de l'apparition ou non d'un **Nop**: dans le cas de **Nop**s ce sont les résultats des unités logiques (LLU et RLU) qui sont mis sur les bus résultats correspondants.

3.4. LES OPERATIONS POI

Parmis les opérations POI, nous avons les opérations logiques et l'opération de chargement des registres conditions/masques (**SF**).

Les opérations logiques sont de type:

aa=(aa) op (bb)

Lors de l'apparition d'une commande **SF** une inhibition des opérations logiques par les champs **ER** et **EL,** ou la spécification d'une affectation **a=(a)**, est souvent nécessaire.

La table 4 donne l'ensemble des codes opérations POI. La table 5 donne l'ensemble des indicateurs ainsi que leurs adresses de sélection.

3.5. LES OPERATIONS D'ENTREES/SORTIES

Les opérations d'entrées/sorties sont données par le champ **IO**, comme indiqué précédemment. La table 6 montre le cheminement des données lors de la réalisation de telles opérations. Nous faisons remarquer qu'une opération d'entrée, lorsqu'elle

n'est pas masquée, est prioritaire par rapport à un rangement mémoire (cas ou le type de destinations est **wm** ou **iwm**).

4. MODES D'OPERATION

Le processeur SMAL_A31 est muni d'un mécanisme efficace pour assurer "l'autonomie SIMD". Nous avons implémenté trois modes de fonctionnement principaux:

-le mode normal
-le mode "conditions séparées"
-le mode commun

.Le mode normal (**nc**) correspond à la réalisation sans condition des différentes opérations d'une instruction: la partie gauche (resp. droite) de la POS réalise l'opération OPL (resp. OPR).

.Le mode "conditions séparées" est de deux types, comme indiqué par la table 7: **sm** et **sm2**.

Dans le premier mode nous retrouvons le même mécanisme que celui de SMAL_X31, où chaque partie de la POS réalise une opération pouvant être masquée par le registre condition/masque (POI) correspondant. Cela revient à la formalisation suivante:

if ((lk)==0) then opl else nop;

pour la partie gauche, et

if ((rk)==0) then opr else nop;

pour la partie droite.

Le second mode séparé est une extension du premier pour l'opération de chargement des registres conditions/masques. Nous avons en plus ce qui suit:

if ((lk)==0 && sf) then load(lk);

pour le masque de gauche et,

if ((rk)==0 && sf) then load(rk);

pour le masque de droite.

.Dans le cas du mode commun, chaque partie de la POS réalise l'une des opérations spécifiées dans le couple (**OPL,OPR**). Ce mode est une extension du mode "condition commune" que l'on retrouve dans SMAL_X31. Les opérations se déroulent comme suit:

if (lk) then opr else opl;

pour la partie de gauche de la POS, et

if (rk) then opr else opl;

pour la partie de droite de la POS.

En résumé les modes opérations de SMAL_A31 répondent bien au modes de fonctionnement que l'on trouve dans les machines SIMD massivement parallèle existantes; A savoir le mode masqué et le mode conditionnel (**sm**) de type ***if_then..*** De plus, il offre le mode conditionnel étendu **if_then_else** (**cm**). Le mode **sm2** permet de réaliser facilement des opérations telles que les comparaisons, où la condition peut être changée au niveau d'un **then**.

5. AUTONOMIE D'INTERCONNEXIONS

Dans un réseau maillé, chaque processeur élémentaire peut se trouver dans une certaine configuration d'interconnexions avec ses voisins directs (Nord, Sud, Est et Ouest) et/ou avec des voisins éloignés. Dans le premier cas nous parlons de liaisons locales alors que dans le second nous parlons de liaisons globales.

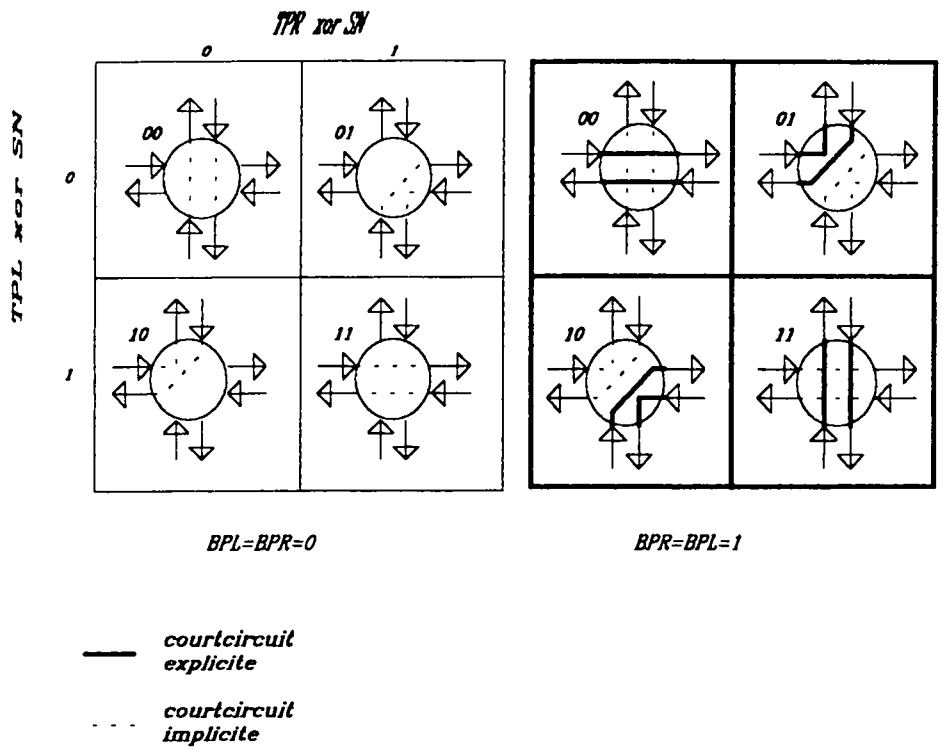

Figure 4: L'autonomie d'interconnexion.

Les registres **TPR** et **TPL** de la circuiterie d'échanges (**EU**) du processeur élémentaire fixent les liens effectifs avec le voisinage, et de manière plus précise le couple de

liens entrées/sorties de la périphérie correspondant à un couple (voisin de gauche, voisin de droite).

Les registres **BPR** et **BPL** fixent les "courtcircuits" (by_pass) de manière explicite sur les liaisons effectives. La réception de données depuis les voisins est toujours possible suivant qu'elles soient masquées ou non.

Le second type de courtcircuits est implicite et ne dépend que de la configuration topologique du processeur élémentaire. Ces courtcircuit sont définis sur les liens ineffectifs et servent d'éventuels ponts de connexions pour d'autres processeurs élémentaires du réseau.

Le bit **sn** de l'instruction, s'il est à 1, sert à complémenter la configuration fixée par **TPR** et **TPL**.

La partie de gauche de la figure 4 montre les différentes configurations possibles, en fonction de **TPR** et **TPL** et **sn** ainsi que les courtcircuit implicites correspondants. A droite, nous donnons des exemples de courtcircuits explicites.

6. FLEXIBILITE TOPOLOGIQUE

La question qui se pose lors de l'utilisation d'une architecture parallèle est de savoir quelles sont les possibilités d'émulation de réseaux pouvant être mises en oeuvre. L'intérêt qui découle de cela est surtout algorithmique. En effet si un réseau de processeurs **R1** peut émuler avec une complexité temporelle **Te**, un réseau de processeurs de type **R2**, alors tout algorithme dédié au réseau **R2** s'exécutant avec une complexité temporelle **Ta** peut être exécuté avec une complexité temporelle au plus égale au produit **TexTa** sur l'architecture **R1**. De ce fait, il en découle la notion de coût de portabilité entre architectures parallèles de différentes topologies, et une influence sur la définition d'un langage parallèle pour la machine cible.

Le but de cette partie est de donner certains exemples d'émulation, et d'opérations qu'on trouve dans certains langages parallèles notamment le langage C^* de la Connection Machine [2], que l'on peut mettre en oeuvre sur une architecture SMAL.

6.1. MODELE DE CONNEXIONS

Le réseaux SMAL peut être vu comme un ensemble de deux composantes topologiques: La composante globale qui consiste à rassembler les processeur élémentaires dans un réseau torique, et une composante locale définie par la fonction de connexion d'un processeur élémentaire avec sa périphérie.

Un processeur élémentaire SMAL_A31 peut être vu plus simplement, comme étant un élément de calcul avec deux ressources d'entrées données **r1** et **r2**, représentant les FIFOs, deux sources de données locales **p1** et **p2**, et quatre ports d'entrées/sorties nord, sud est et ouest. Nous noterons **n**,**s**, **e** et **o** (resp. **N**,**S**,**E** et **O**) les points d'entrées (resp. sorties). Nous rajoutons à ces éléments deux registres de topologie (**t1** et **t2**) sur 1 bit chacun, deux registres (sur 1 bit chacun) de courtcircuits (**b1** et **b2**) et deux registres 1 bit de masques d'entrées aux ressources **r1** et **r2** qui sont respectivement

m1 et **m2**. De plus nous avons une commande de complémentation de la configuration topologique que l'on nomme **c**. La figure 5 rassemble toutes ces données.

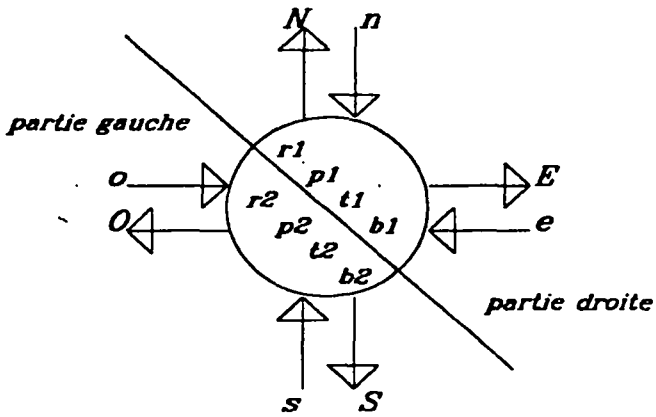

Figure 5: modèle (connectique) de SMAL_A31.

Le réseau global est matériellement figé. Nous pouvons le voir comme une matrice carrée **LxL** (pour simplifier), de processeurs élémentaires chacun ayant ses coordonnées **i** et **j**. Finalement tout processeur élémentaire est lié avec ses voisins avec la fonction connectique suivante:

o(i,j)=E(i-1[L],j)
e(i,j)=O(i+1[L],j)
s(i,j)=N(i,j-1[L])
n(i,j)=S(i,j+1[L])

La fonction réseau local de SMAL_A31 se ramène à la formulation suivante:

Nous définissons une fonction de connexion **connect(p,{c0,c1,c2..c8})** ou **p** est une valeur allant de 0 à 8 et **ci** i=0..8 des connexions d'entrée. La sortie de cette fonction est la connexion ayant comme numéro d'ordre **p**.

Si l'on considère que **p** est formé par la concaténation des 3 bits $b_1t_2t_1$ ou $b_2t_2t_1$ alors les sorties du processeur élémentaire **PE(i,j)**, conformément à l'autonomie adoptée, s'écriront:

$E(i,j)=connect_{(i,j)}(b_1t_2t_1,\{p1,s,p1,w,w,s,s,w\})$
$N(i,j)=connect_{(i,j)}(b_1t_2t_1,\{s,p1,w,p1,s,w,w,s\})$
$O(i,j)=connect_{(i,j)}(b_2t_2t_1,\{p2,p2,n,e,e,n,n,e\})$
$S(i,j)=connect_{(i,j)}(b_2t_2t_1,\{n,e,p2,p2,n,e,e,n\})$

Pour simplifier nous utiliserons deux variables $x=b_2b_1$ et $y=t_2t_1$. Chaque processeur élémentaire **PE(i,j)** à sa propre fonction de connexions locales définie par le couple $(x,y)_{(i,j)}$.

6.2. RESEAUX LINEAIRES

SMAL offre la possibilité d'implémenter des réseaux linéaires pouvant englober tous les processeurs élémentaires. Cela revient à chainer tout les processeurs élémentaires de SMAL de manière à former une grande table de processeurs, chacun ayant un successeur et un prédécesseur.

En utilisant les fonctions de connexion données dans le paragraphe précédant, nous pouvons établir la conséquence suivante:

Tout processeur $PE_{(i,j)}$ peut avoir une connexion directe avec un processeur $PE_{(i+1[L],j+1[L])}$ et réciproquement.

Une possibilité serait de configurer les processeurs $PE_{(i,j+1[L])}$, $PE_{(i,j+1[L])}$ et $PE_{(i,j+1[L])}$ comme suit:

$(\mathbf{x},\mathbf{y})_{(i,j)}=(\mathbf{?0},\mathbf{?1})$
$(\mathbf{x},\mathbf{y})_{(i+1[L],j+1[L])}=(\mathbf{0?},\mathbf{0?})$
$(\mathbf{x},\mathbf{y})_{(i,j+1[L])}=(\mathbf{??},\mathbf{01})$

où "?" indique un état pouvant être à zéro ou à 1.

Pour avoir un réseau linéaire global il suffit de configurer la matrice toroidale de processeurs élémentaires comme suit:

$(\mathbf{x},\mathbf{y})_{(L,j)}=(\mathbf{00},\mathbf{01})$ j=0..L
$(\mathbf{x},\mathbf{y})_{(i,j)}=(\mathbf{00},\mathbf{00})$ j=0..L et i=0..L-1

Nous pouvons remarquer que les éléments de la Lième colonne vérifient la conséquence donnée ci-dessus.

6.3. EMULATION D'UN ARBRE BINAIRE

Un arbre binaire s'implémente facilement sur les architectures à autonomie d'interconnexion. Nous présentons ici un modèle classique d'émulation que nous montrons par la figure 6.

Nous prenons comme exemple de problème adapté à ce genre d'émulation, une opération commutative et associative ("semi-group operation") que l'on note "**o**". Nous proposons alors d'évaluer une opération sur une ligne donnée du réseau,où chaque processeur possède une donnée de départ locale notée *v*. Pour simplifier nous supposons que L est une puissance de 2. L'évaluation de "**o**" peut s'effectuer sur un réseau à base de SMAL_A31, en n'utilisant qu'une seule direction (gauche vers la droite), avec l'algorithme simplifié suivant:

```
Tree

begin

comment: initialisation endcomment

(x,y)=(?0,00):

for t=1 to t<=log2(L) do
        if bit_eq_0(t,My_address) then
        send_p1(v);(x,y)=(?1,00);
        else
        receive_r2(v*);v=vov*;
        endif
        enddo
end
```

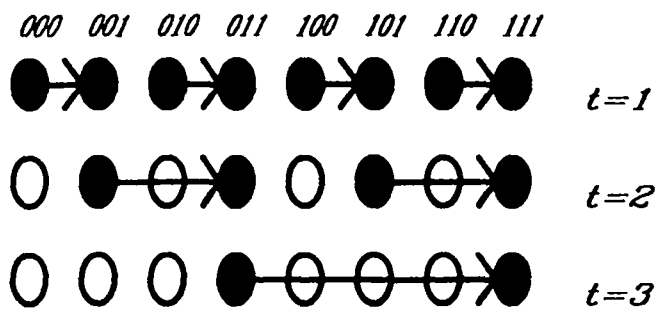

Figure 6: émulation d'un arbre binaire.

Dans cet algorithme, nous précisons que les actions **send_p1**() (émission sur le port **p1**) et **receive_r2**() (réception sur ressource **r2**) se font simultanément. En effet, quand un processeur envoie une donnée sur un port elle peut être automatiquement sérialisée au niveau de la file du voisin. La fonction **bit_eq_0** donne une valeur vraie si le $t^{ème}$ bit de l'adresse physique du processeur (sachant qu'elle est préchargée) est égal à zéro. Le résultat est récupéré sur l'élément d'adresse L-1 de la ligne considérée.

Dans la procédure **receive** il peut y avoir un masquage des entrées aux resources.

Enfin ce genre d'algorithme peut être étendu pour tout les éléments du réseau et peut servir pour les opérations de réduction.

6.4. ARCHITECTURE A BUS MULTIPLES

Les architectures matricielles munies d'autonomie d'interconnexion peuvent émuler des architectures matricielles "augmentées". Nous faisons référence à une grille de processeurs avec des bus de transmission. Les cas usuels d'architectures augmentées sont:

-Grille avec un bus global [4]
-Grille avec des bus multiples lignes et colonnes [5][17]
-Grille avec bus hiérarchiques [6]

Le principe employé dans ce type de topologie est commun aux trois cas. **En considérant une ligne i (anneau)** de processeurs SMAL_A31, l'idée et de fixer un seul processeur émetteur et un ou plusieurs récépteurs. Ces éléments sont définis soit durant un traitement donné ce qui nous mène à une configuration dynamique, ou bien fixés dès le début du traitement et dans ce cas nous sommes dans un cas d'émulation topologique. Pour voir la mise en oeuvre de ce genre de problème, considérons l'exemple suivant:

Soit un processeur "maître", qui possède une adresse **A** d'un autre processeur issue d'un traitement quelconque. Nous cherchons à établir une connexion bidirectionnelle entre ce processeur et le processeur d'adresse **A**. Pour une direction gauche/droite, nous procédons alors comme suit:

comment: *initialisation* ***endcomment***

if *Im_master* ***then*** *(x,y)=(00,00)*
else *(x,y)=(01,00)*
endif
comment: *on suppose que l'adresse A est différente de celle du processeur maître*
endcomment

if *Im_master* ***then send_p1(A)***
else receive_r2(v)
if *my_address==v* ***then*** *(x,y)=(00,00)*
else if not *Im_master* ***then*** *(x,y)=(11,00)*

endif
endif

La variable *Im_master* indique qu'un processeur est "maître" ou non. La variable *my_address* locale à chaque processeur élémentaire sert de stockage d'adresse pour celui-ci, *v* est la variable devant contenir la donnée émise par l'émetteur. Cette variable peut être stockée physiquement dans une file (gauche) dans le cas de SMAL_A31. L'algorithme est un simple vote où tous les processeurs recoivent l'adresse envoyée par le maître et après comparaison, un lien direct est établi entre le maître et le processeur élu.

6.5. OPERATIONS LOGIQUES

Nous considérons la classe des opérations "**o**" logiques de base du fait qu'il est possible de les traiter différemment vu que l'on peut tirer avantage de l'autonomie de connexion.

L'évaluation d'une opération logique sur un ensemble de données réparties peut se faire à l'aide de l'algorithme décrit précédemment. Cependant si un processeur est

considéré comme un courtcircuit, nous pouvons avoir un raisonnement à base d'interrupteurs à la manière de la logique à transistors de passage que l'on trouve en technologie MOS. Nous prenons comme exemples les opérations "." (ET logique) et "+" (OU logique).

Pour montrer le principe de la mise en oeuvre, nous considérons toujours un anneau de processeurs numérotés de 0 à L-1. Sachant que l'on veut récupérer le résultat au niveau du processeur 0 (processeur maître). Chaque processeur possède une variable booléenne que l'on note *v*. L'algorithme d'évaluation de l'opération "." ou "+" devient:

comment: *pour une opération OU vop=0, pour l'opération ET vop=1* ***endcomment***

if *(v=vop & not Im_master)* ***then*** *(x,y)=(?0,01)*
else *(x,y)=(?0,00)*
endif

send_p1*(v);* ***receive_r2****(result);*

comment: *le résultat significatif est donné par la variable result locale au processeur maître* ***endcomment***

La figure 7 explique schématiquement l'évaluation de telles opérations.

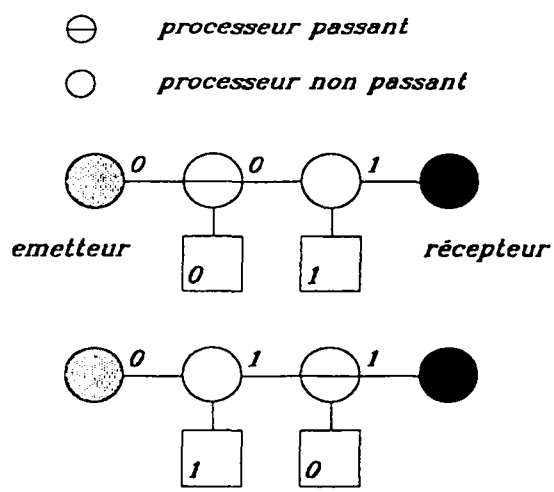

Figure 7: exemple d'évaluation d'un ou logique.

Ainsi les opérations logiques peuvent être évaluées en un temps constant. L'extension peut être facilement faite pour les opérations de calcul du **max**ima et **min**ima d'un ensemble de données réparties [9]. Le calcul du **max** et **min** necessite un temps proportionnel à la longueur en bits des opérandes.

7. APPLICATIONS

Dans cette partie, nous présentons quatre exemples d'applications. Nous reprenons, en premier lieu, les deux applications décrites dans [18] utilisant des communications locales, afin de donner une comparaison entre leur mise en oeuvre respective dans SLiM dont l'architecture est décrite ci-dessous, et SMAL (PE sériel 1bit). Ces applications sont le calcul de convolutions bidimentionnelles et le filtrage médian. Les deux autres exemples, donnés en second lieu, utilisent des communications globales et mettent en oeuvre l'autonomie d'interconnexions.

L'architecture de SLiM est donnée par la figure 8. Le découpage du chemin de données est tel que les recouvrements entre les communications, les entrées sorties et les calculs sont possibles. Le point que l'on retient d'une grille SLiM est l'utilisation d'un plan de registres S glissant (ang. sliding plane). Ce plan est utilisé pour le stockage d'une image qui peut alors être décalée dans une direction donnée (Nord, Sud, Est ou Ouest).

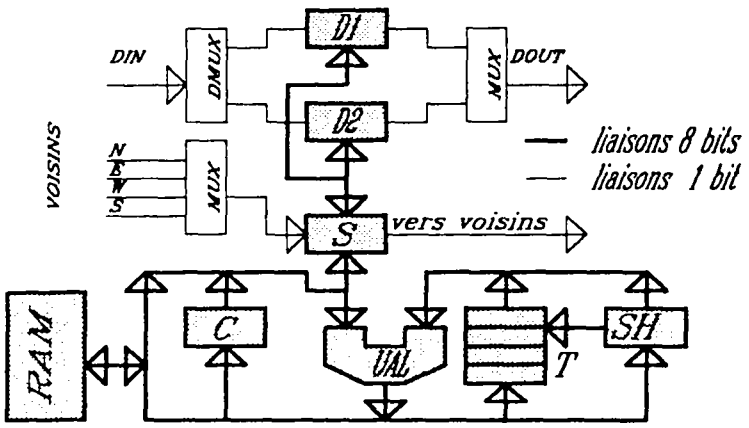

Figure 8: Architecture du processeur élémentaire SLiM.

7.1. CONVOLUTION

Le calcul de convolutions d'une image, est un exemple que l'on retrouve souvent dans les publications traitant d'architectures parallèles étant donné son importance en traitement d'images [19][20][21]. Une publication concernant ce sujet, et spécialement la convolution 2D, a été faite par Lee et Aggarwal [22]. Ce dernier a en effet introduit la notion de chemin hamiltonien. L'existence d'un chemin hamiltonien selon la fenêtre, permet de gagner aussi bien en performance de calcul sur le rapport calcul/communication, qu'en espace mémoire pour la sauvegarde des résultats intermédiaires.

Le principe du chemin hamiltonien est tel qu'une donnée (ou un résultat intermédiare) doit visiter un ensemble de processeurs connexes à raison d'une visite seulement par processeur. Les figures 9a1 et 9b1 donnent des exemples de chemins hamiltoniens respectivement pour une fenêtre carré et une fenêtre en diamant.

Le calcul d'une convolution revient à une multiplication/ accumulation, de la forme $y_i=y_{i-1}+w_i*x_i$. Cependant pour un calcul bidimensionnel deux approches sont possibles, tenant compte de l'existence d'un chemin hamiltonien:

Si l'on considère l'architecture SLiM, la première méthode consiste en la propagation de pixels ce qui donnerait l'algorithme suivant [18]:

```
pardo % faire parallèle %
T<-s*w0; s<- s_voisin(0);
enddo
for t=1 to window_size-1
pardo
        T<-T+s*wi;
        s<- s_voisin(t);
enddo
```

T représente un registre interne à un processeur élémentaire et s le registre faisant partie du plan glissant. Toute l'image est contenue dans ce plan glissant. Les calculs et les transferts sont dans cet exemple parallélisés.

s_voisin(t) est utilisée pour spécifier un décalage de l'image au temps *t*. Par exemple, pour une fenêtre 3x3 pour t=0..7 nous devons avoir respectivement la succesions de décalage E, N, O, O, S, S, E et E pour réaliser le chemin de convolution de la figure 9a1.

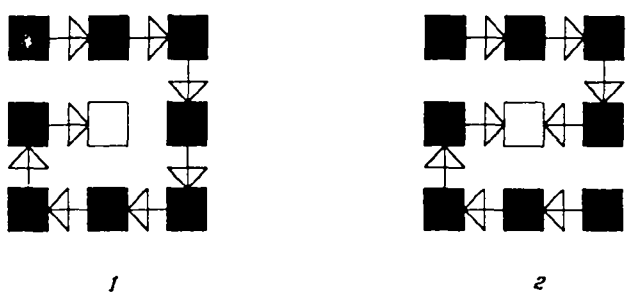

Figure 9a: Chemin hamiltonien pour une fenêtre carré 3x3 1)SLiM 2)SMAL.

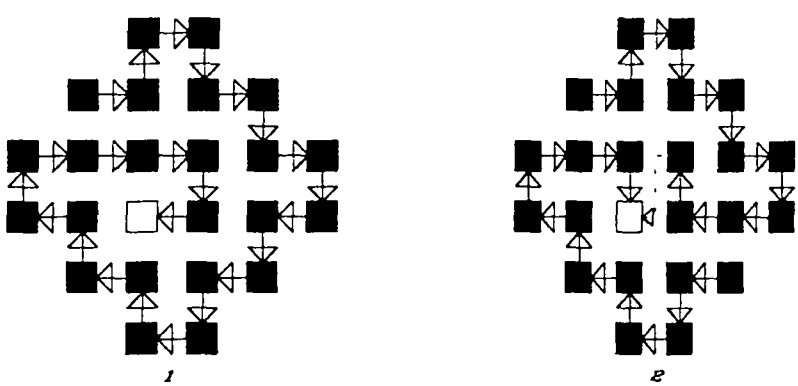

Figure 9b: Chemin hamiltonien pour une fenêtre en diamant 6x6 1)SLiM 2)SMAL.

La seconde méthode séquentielle (par rapport à un processeur élémentaire), consiste à propager les résultats intermédiaires:

```
T<-w0*p;
for t=1 to window_size-1
begin
        s<- T_voisin(t);
        T<-s+wi*p;
end
```

Où **p** est cette fois-ci la valeur du pixel de l'image, correspondant à un processeur donné et s ne sert ici qu'en simple registre. *T_voisin(t)* est similaire à *s_voisin(t)* sauf que le graphe de transfert (de résultat intermédiaire) est directement décrit par le graphe du chemin de convolution.

Le premier algorithme est plus adéquat à l'architecture SLiM, du fait que localement le chemin de données est sur 8 bits. Le deuxième algorithme est plus adéquat pour SMAL. En effet, avec la sérialisation des opérations, et une communication à base de files (FIFOs), **l'absorption des communications** par le calcul s'obtient comme précisé dans l'article [23].

Dans SMAL, il est possible de parler d'algorithmes de calcul à double chemins hamiltoniens géométriquement symétrique par rapport au centre d'une fenêtre, comme le montre la figure 9a2. Cette symétrie est adoptée pour utiliser les liaisons bidirectionnelles et les deux axes de communications (horizontalement ou verticalement selon le contrôle) présents dans SMAL_A31.

Dans le cas ou il n'existe pas de chemins hamiltoniens, des étapes de communications supplémentaires, montrées en pointillés sur la figure 9b2, apparaissent. Quant au nombre de ces étapes, il dépend de la construction topographique de la fenêtre.

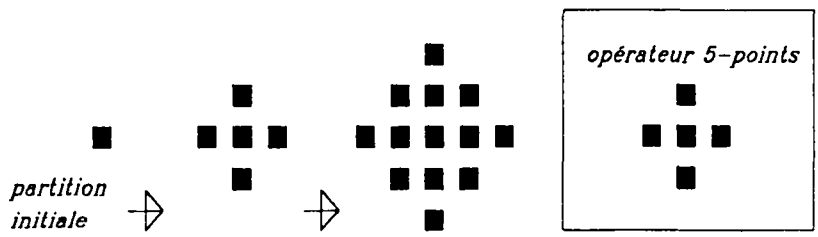

Figure 10: Extension d'une partition par un opérateur 5_points.

Une fenêtre peut être vue comme le résultat d'une suite d'applications successives d'opérateurs d'extension sur une partition noyau [24]. C'est ainsi qu'une fenêtre en diamant impaire [22] d'ordre 2n+1, peut être obtenue par 2n applications de l'opérateur 5-points sur un point (voir figure 10). Pour ce même exemple le nombre d'étapes supplémentaires est de 2n.

La notion de chemin hamiltonien peut être étendue pour des liaisons en diagonales. D'après le modèle d'une matrice SMAL_A31, nous pouvons constater que ce type de

liaisons est possible dans un sens. Cela permet de gagner du temps pour certaines applications de traitement d'images de bas niveau comme l'opérateur de Sobel ou le Laplacien. La figure 11 montre le graphe de calcul d'un laplacien 3x3 sous SMAL.

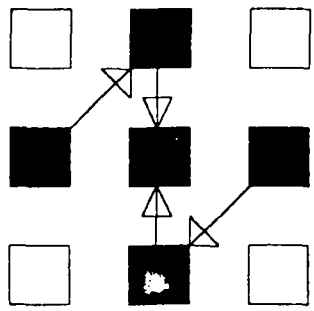

Figure 11: chemins hamiltoniens pour un laplacien 3x3.

7.2. FILTRE MEDIAN

Le filtrage par médiane consiste à remplacer un point d'une image (pixel) par la médiane de son voisinage. La médiane d'un ensemble de nombres est un nombre choisi de telle sorte que la moitié des nombres de l'ensemble lui soit inférieure et l'autre moitié lui soit supérieure. Le calcul de la médiane d'un pixel nécessite alors un tri sur le voisinage défini par une fenêtre. Le tri local à chaque processeur élémentaire d'une machine SIMD est une tâche coûteuse en temps surtout pour des machines ne disposant pas d'autonomie d'adressage.

La méthode décrite dans [18], consiste à construire une liste unidirectionnelle ordonnée sous la forme d'une zone contigüe. Avec le mécanisme de l'image glissante (chemin hamiltonien unique) nous obtenons l'algorithme suivant:

```
for i=1 to window_size-1
pardo
        insertion(s,l);
        s<- pixel_voisin;
enddo
```

où **l** est la liste (chaque PE a sa propre liste locale). Le résultat est tel que dans tout processeur élémentaire nous avons un tableau de valeurs ordonnées. La mediane en chaque point serait alors la valeur se trouvant au milieu de ce tableau. Si l'on considère qu'un décalage (pour l'opération d'insertion) et une comparaison sont temporellement équivalents alors l'algorithme est réalisable en $O(n^2/2)$, **n** étant le nombre de points de la fenêtre.

La construction d'une liste nécessite un espace mémoire proportionnel au nombre de voisins. Dans le cas de SliM par exemple, 9 mots de 8bits sont nécessaires pour une fenêtre 3x3 sachant qu'un pixel est codé sur 8bits. Pour une architecture à base de PEs 1bit, cela demanderait 72 bits.

Nous décrivons ci-dessous un algorithme qui tient compte de l'espace mémoire. Le principe de l'algorithme est comme suit:

Soient **E** un ensemble de **2n** valeurs $\{e_1,..,e_{2n}\}$, **A** et **B** deux parties de **E** de taille **n**, tel que **E=A+B**. Considérons $E'=\{r_1,..,r_{2n}\}$ comme résultat du tri de **E**. Alors nous cherchons à trouver les éléments successifs r_n et r_{n+1}. La recherche de ces éléments est équivalente à décomposer **E** en deux partitions **A** et **B** telles que tout élément de **A** est inférieur à tout élément de **B**:

```
a<-max(A);b<-min(B);
while(a>b)
begin
        A<-A-a+b;B<-B-b+a;
        a<-max(A);b<-min(B);
end
```

Le filtrage par médiane sur une image, doit tenir compte de la répartition spatiale des données, sachant que l'on a un pixel par processeur élémentaire. Nous proposons le squelette algorithmique suivant:

```
pardo
MAX(A,a);MIN(B,b);
enddo
if a>b then begin
                a'=b;
                b'=a;
                end
        else begin
                a'=b;
                b'=a;
                end
while((MAX(A,a)>MIN(B,b)) & ((A+B) not empty))
begin
pardo
% phase élimination %
        A<-A-a;
        B<-B-b;
enddo
parddo
        a'<-max(a',b);
        b'<-min(b',a);
enddo
end
```

MAX(X,x) (resp. MIN(X,x)) est une fonction qui renvoie l'élément maximum (resp. minimum) de X (voisinage). La valeur renvoyée est aussi sauvegardée dans x.

a, b, a' et b' sont des zones locales à chaque processeur élémentaire. Autrement dit, ceux sont des variables parallèles.

L'opération d'élimination d'un élément d'un ensemble n'est qu'un simple marquage. Chaque processeur élémentaire a une zone de taille **window_size-1** bits ou chaque bit correspond à un voisin. L'élimination d'un point par rapport au voisin revient à positionner à une valeur donnée le bit correspondant.

Nous constatons dans l'algorithme que l'on doit parcourir deux fois un ensemble à chaque pas de boucle: une fois pour la procédure d'élimination et une autre pour la recherche du "**min**" ou du "**max**". A la sortie de la boucle, il ne reste qu'à comparer l'élément central à **a** et **b** pour en déduire la médiane.

N'ayant pratiquement que des opérations de comparaisons et de sélections cet algorithme a une complexité temporelle de $O(n^2/2)$, **dans le pire des cas**. De plus il ne consomme pas beaucoup d'espace mémoire.

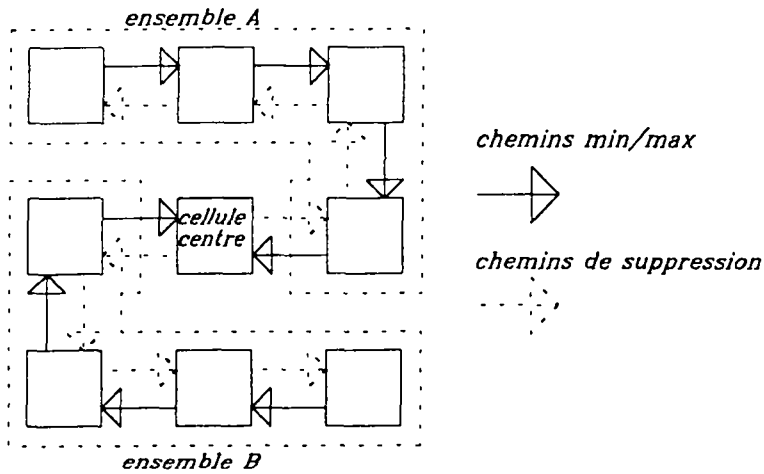

Figure 12: chemins hamiltoniens pour le calcul de médiane.

La décomposition en deux ensembles est parfaitement adaptée au réseau SMAL. Le principe du double chemins hamiltoniens disjoints, et les capacités d'autonomie du PE répondent directement aux besoins de la mise en oeuvre de l'algorithme. La figure 12 montre un exemple sur une fenêtre carré 3x3.

7.3. PROBLEME DU VOISIN LE PLUS PROCHE

Soit une image binaire I où un pixel peut être à 0 ou à 1. Nous nous proposons de calculer le voisin le plus proche (au sens de la distance de Manhattan) de chaque pixel, ayant pour valeur 1. Nous supposons que chaque pixel est affecté à un processeur.

L'algorithme que nous proposons est assez simple en comparaison avec ce qui a été proposé dans la litérature pour une grille augmentée [6]. Nous proposons par la suite

une esquisse de cet algorithme implémentable sur une architecture à base de SMAL_A31.

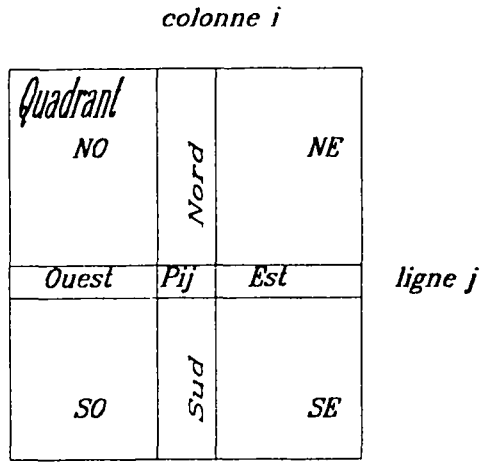

Figure 13: zones de recherche du voisin le plus proche.

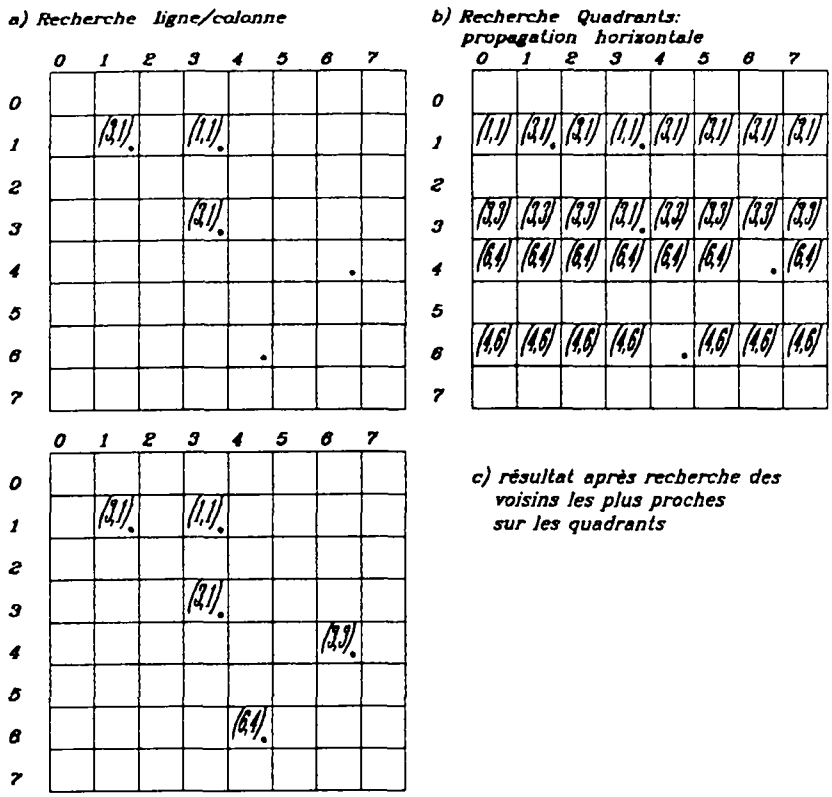

Figure 14: exemple de recherche du voisin le plus proche.

Chaque pixel voit sa colonne, sa ligne et les quatres quadrants qui l'entourent, comme le montre la figure 13. Les étapes de l'algorithme se résument dans ce qui suit:

Sachant qu'un processeur peut être complétement ***court-circuité*** *et que l'on dispose d'une communication bidirectionnelle, tout processeur ayant comme valeur de pixel égale à 1, peut facilement connaitre les coordonnés de ses voisins les plus proches (de valeur 1) situés sur sa colonne et sa ligne. Du fait que l'on ne peut avoir que quatre voisins à 1 au plus, on peut trouver, localement, le pixel vérifiant la distance minimale et cela en un temps O(1).*

La seconde étape consiste à trouver le plus proche voisin appartenant à l'un des quatre quadrants:

A l'aide d'une segmentation, chaque processeur contenant un pixel à 1, peut envoyer ses coordonnées à ses voisins de gauche et de droite contenant un pixel à 0. Cela peut être effectué en adoptant le même principe donné au §6.4. Après cette propagation, chaque pixel (processeur) dispose sur sa colonne tous les pixels à 1, pouvant être éligibles. Après un calcul de distance et en appliquant toujours une segmentation, on peut facilement trouver le minimum comme indiqué au §6.5. Enfin l'algorithme se termine par la recherche locale du pixel le plus proche. La complexité de cette seconde étape est de O(1).

La figure 14 donne un exemple de recherche du voisin le plus proche.

Remarque: La segmentation consiste dans le cas de cet algorithme à délimiter un ensemble de pixels successifs éteints d'une même ligne (ou colonne), par au plus 2 pixels à 1.

7.4. RESEAU DE HOPFIELD

Ce second exemple est consacré au réseau de neurones et plus particulièrement aux réseaux de type Hopfield. Nous donnons le principe de leur implémentations du point de vue topologique.

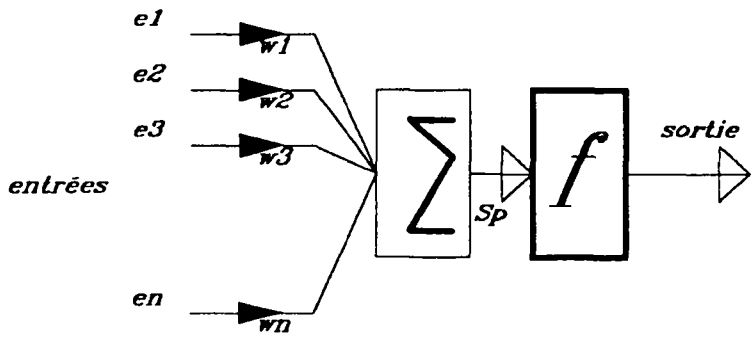

Figure 15: neurone formel.

Nous rappelons qu'un neurone formel est généralement considéré comme un opérateur à n entrées et une sortie. La sortie est obtenue moyennant une fonction f de décision, sur la somme pondérée de ses entrées. La sortie est aussi appellée état du neurone. Les coefficients de pondération sont appelés coefficients synaptiques. La

figure 15 schématise un neurone formel typique. Pour des précisions sur ce sujet le lecteur peut se reférer aux publications (exemple [25]).

Prenons un réseau à n neurones complètements connectés. Nous nous intéressons à la phase de relaxation du réseau, ce qui revient à calculer un produit matrice/vecteur. Nous considérons alors une matrice toroïdale de *nxn* processeurs SMAL_A31, où une diagonale est affectée à l'évaluation du vecteur état des neurones. Les autres processeurs sont affectés à l'évaluation des différentes pondérations. Dans ce cas chaque colonne *i* est affectée à un neurone *i*, et la tâche de chaque processeur Pij est de calculer l'entité *CijxSj* où *Cij* est le coefficient synaptique liant le neurone *j* au neurone *i* et *Sj* l'état du neurone *j*.

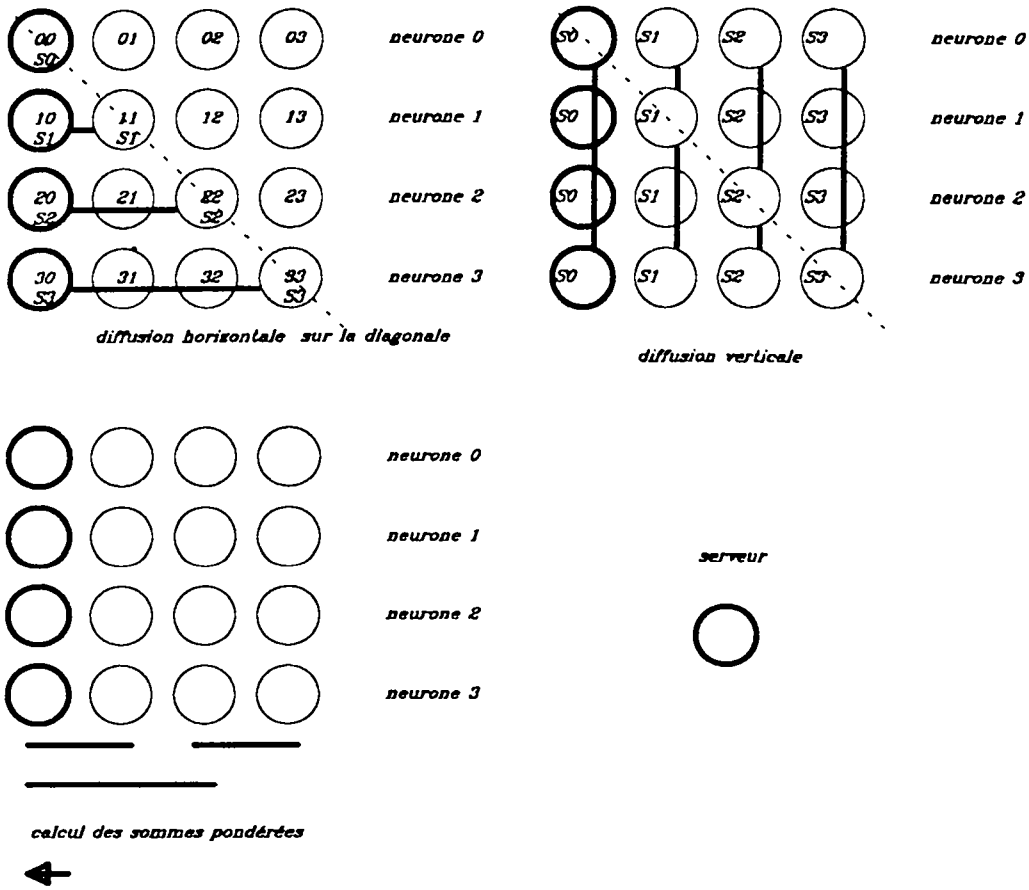

Figure 16: relaxation dans un réseau neuronal.

Une itération du réseau revient à l'algorithme suivant:

1.Evaluation du vecteur états des neurones.
2.Chaque processeur de la diagonale (choisie) propage l'état du neurone correspondant sur toute sa ligne (broadcasting).
3.Calcul des produits CijxSj.
4.Calcul de la somme pondérée par arbre binaire.
5.Aller à 1.

Dans l'étape 4, le calcul peut être effectué de manière à ce qu'un processeur diagonal soit la racine de l'arbre binaire (virtuel), correspondant à sa colonne. La figure 16 explique l'algorithme sur un petit réseau.

La complexité temporelle d'une itération est de l'ordre *O(log(n))*.

8. COMPLEXITE TEMPORELLE

Dans ce paragraphe nous essayons de donner une évaluation assez claire de ce que peut être une complexité temporelle, pour des architectures à autonomie de connexion. Nous nous positionnons dans un cadre WSI afin d'uniformiser certains facteurs. On s'interesse au coût des communications d'un algorithme. Nous noterons **H** la période de l'horloge de base du système et **u** la durée d'une communication entre deux processeurs voisins. Nous considérons **u** comme unité temporelle et **H=g.u**, **g** étant une constante. Généralement **H** est détérminé par le chemin critique (mémoire/opérateurs) du processeur élémentaire.

Un algorithme peut être vu comme un ensemble d'étapes successives E_t **t=0,..,m**. Dans le contexte d'architecture à autonomie de connexions, à chaque étape **t** est associé un coût "topologique" noté $c_t=d_t.u$, où d_t représente le lien le plus long entre deux processeurs élémentaires. Il est bien évident que tant que $d_t<=g$ l'horloge est conforme: la réalisation de l'étape t est exacte. Maintenant si $d_t>g$ alors l'horloge de base doit être ajustée de manière à assurer un fonctionement correct. Rappelons qu'on a deux types d'algorithmies: les algorithmes à coût calculable, donc régis par une fonction connue par le contrôleur du réseau de processeurs, et les algorithmes dont la fonction de configuration n'est pas connue à priori. Dans les deux cas on peut envisager un mécanisme d'horloge réglable.

Pour optimiser le temps des communications, la solution idéale est d'avoir à tout moment, dans le cas où $d_t>g$, une horloge $H'=g'.u$ où $g'=d_t$. Dans le cas approximatif nous pouvons nous contenter d'assurer **H'=k.g.u** tel que $k.g>=d_t$. Dans le cas le plus simple **k** est choisi égal à $[d_t/g+1]$ ([x] symbolise la partie entière de x). Pour des réseaux de faible taille on peut s'attendre à des facteurs multipicatifs de l'ordre de $\log^s(d_{max})$ où d_{max} [7] est la distance maximale, et s un nombre compris entre 0 et 1.

Le problème de mise en oeuvre d'un système d'horloge réglable se pose surtout pour les applications où la fonction de reconfiguration n'est pas connue. Cependant nous pouvons imaginer un système permettant de trouver dynamiquement la fréquence d'horloge adéquate à une étape donnée du traitement. Cela nous confronte à une complexité matérielle.

Afin de pallier à ce problème de complexité materielle, une véritable solution se baserait plutôt sur un appui logiciel. Cela nous conduit au choix d'une mise en oeuvre matérielle non coûteuse (taille de la matrice raisonnable et simple réglage de

l'horloge), et de ramener la problèmatique au partitionnement et à la répartition de données (à titre indicatif le lecteur peut consulter les références [14],[15] et [16]).

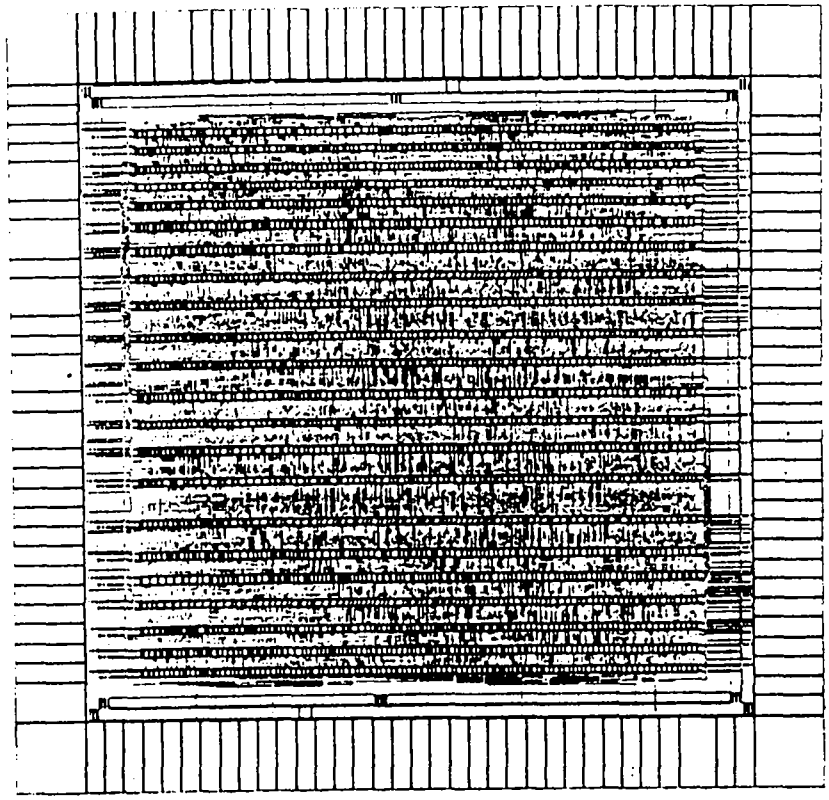

Figure 17: implantation de 4 processeurs élémentaires.

9. CONCLUSION

Des prototypes "Standard Cell" de processeurs élémentaires SMAL_A31, avec 1 et 4 (figure 17) processeurs élémentaires, ont été réalisés au laboratoire CAO&VLSI de Paris IV. Les jeux de masques ont été envoyés au CMP français. Le circuit à 4 processeurs présente les caractéristiques suivantes:

technologie	: CMOS 1.5µm
Surface CMP	: 7x7 mmsq
Lambda	: 1.4µ
dimensions	: 4830x4600 lamba-sq
chemin critique	: 20 ns
nombre de portes	: 2600

L'autonomie d'interconnexion est une bonne alternative pour l'exploitation "optimale" des réseaux matriciels comme suggéré par Maresca et al [9]. Dans SMAL_A31, nous avons adopté une solution flexible.

L'adoption d'un modèle VLIW pour SMAL, est une bonne solution à l'autonomie d'opération des processeurs élémentaires. Ce même modèle ouvre une voie vers la conception de réseaux SIMD à grain reconfigurable.

L'introduction des FIFOs pour les communications permet un recouvrement calcul/communication. L'adoption de chemins hamiltoniens, en particulier pour des applications de traitement d'images de bas niveau, permet d'exploiter de manière efficace ce mécanisme.

Remerciements

Je tiens à remercier le Professeur Alain Greiner et Madame Anne Derieux du laboratoire CAO&VLSI_MASI de m'avoir laissé disposer des moyens nécessaires à la réalisation du circuit. Je remercie aussi, mon Directeur de projet, Dimitri Tusera, et Monsieur François Schwaab pour leurs conseils.

Bibliographie

[1].C.E.Leiserson, "VLSI Theory and Parallel Supercomputing", advanced research in VLSI, Proc. of 1989 decennial Caltech conference, edited by C.Seitz .

[2]."Connection Machine Model CM-2 Technical Summary", by Thinking Machines Corporation, Cambridge, Massachusetts, May 1989.

[3].D.W.Blevins et al, "Blitzen: a highly integrated massively parallel machine", Journal of parallel and distributed computing 8,p102-118 1990.

[4].Q.F.Stout, "Mesh-Connected Computers with Broadcasting", IEEE trans. on computers C-32, september 1983.

[5]. V.K.P.Kumar and C.S.Ragharvendra, "Array Processors with Multiple Broadcasting", Journal of parallel and distributed computing 4, 1987.

[6].S.B.Chalasani and C.S.Raghavendra, "Geometric algorithms on HMESH architecture", in proceedings of IEEE Workshop on computer architectures for pattern analysis and machine intelligence, 1987.

[7].M.Maresca and H.Li, "Toward connection autonomy of fine-grain SIMD parallel architecture", in Parallel Processing for Computer Vision and Display, edited by P.M.Dew et al, Addisson Wesley 1989.

[8].M.Maresca and H.Li, "Connection Autonomy in SIMD Computers: A VLSI Implementation", Journal of Parallel and Distributed Computing 7,p302-320 1989.

[9].M.Maresca, H.Li and M.M.C.Sheng, "Parallel Computer Vision on Polymorphic Torus Architecture", in Machine Vision and Applications p215-230, Springer-Verlag New York inc. 1989.

[10].A.Alcolea et al,"A Cost-Effective Architecture For Vision", DEECS University of Zaragoza Spain.

[11].B.Zerrouk, "A SIMD Multiprocessor Architecture based on a fine grain processor for image processing and neural networks emulation", IFIP Workshop On Parallel Architectures On Silicon, Grenoble, December 1989.

[12].J.A.Trotter and W.R.Moore, "Mesh: an architecture forimage processing", in Parallel Processing for Computer Vision and Display, edited by P.M.Dew et al, Addisson Wesley 1989.

[13].B.Zerrouk et A.Derieux, "Simd Multiprocessor Architecture with a Long instruction word", rapport de recherche 1413, INRIA 1991.

[14]. K.Knobe et al, "Data optimization: allocation of arrays to reduce communication on SIMD machines", Journal of parallel and distributed computing 8,p102-118 1990.

[15].H.Umeo et al, "Broadcasting-Bus elimination without any loss of time Efficiency in iterative (Cellular or Systolic) Arrays", VLSI & Computer peripherals, 3rd Annual European computer conference.

[16].P.Arvin and K.Balasubramanian, " Reducing communication costs for sorting on Mesh connected and linearly connected parallel computers, Journal of parallel and distributed computing 9, p318-322 1990.

[17].Y.C.Chen et al, "Designing efficient parallel algorithms on Mesh-connected computers with multiple broadcasting", IEEE trans. on parallel and distributed systems, april 1990.

[18].M.H.Sunwoo and J.K.Aggarwal,"A Sliding Memory Plan Array Processor", IEEE Second Symposium On frontiers of massively parallel computations, 1988.

[19].P.Quinton et Y.Robert, "Algorithmes et Architectures Systoliques", Edition Masson, 1989.

[20].P.M.Dew et al, "Parallel Processing for Computer Vision and Display", Addison Wesley 1989.

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

[22].S.Y.Lee and J.K.Aggarwal, "Parallel 2-D Convolution on Mesh Connected Array Processor", IEEE Trans. On Patern Analysis And Machine Intelligence, Vol. PAMI-9,NO.4, July 1987.

[23].B.Zerrouk et A.Derieux, "SMAL, une architecture parallèle pour le traitement d'images", 3ième symposium sur les architecture nouvelles de machines, Ecole Polytechnique, juin 1991.

[24].F.F.Lee," Partitioning of Regular Computation on Multiprocesseor Systems", Journal of Parallel and Distributed Computing 9,1990.

[25].A.Johannet, "Réseaux de neurones formels: étude d'un simulateur à architecture parallèle et conception d'un circuit intégré", Thèse de doctorat de l'université de Paris 6, décembre 1990.

TABLEAUX

Tableau 1: opérations arithmétiques et logiques POS.

code	Mném	résultat	retenue		
0000	add	a xor b xor cin	and(cin,or(a,b)) or and(a,b)		
0001	gt	and(a,not(b))	b		
0010	lt	and(not(a),b)	a		
0011	c	cin	or(a,b)		
0100	nand	nand(a,b)	and(a,b)		
0101	nb	not(b)	b		
0110	na	not(a)	a		
0111	z	0	1		
1000	suba	cin xor not(a) xor b	and(cin,or(not(a),b)) or and(not(a),b)		
1001	subb	cin xor a xor not(b)	and(cin,or(not(b),a)) or and(not(b),a)		
1010	nor	nor(a,b)	b		
1011	and	and(a,b)	not(b)		
1100	nxor	nxor(a,b)	and(not(a),b)		
1101	b	b	not(b)		
1110	#	a	not(a)	%NOP	%
1111	clr	0	or(a,not(b))	%Reset FIFO	%

Tableau 2: sources d'opérandes des opérateurs POS.

S	Mné	AR	AL	BR	BL
000	mm	mar	mal	mbr	mbl
001	xm	mar	mal	mbl	mbr
010	mf	mbr	mbl	rfifo	lfifo
011	fm	mbr	mbl	lfifo	rfifo
100	mr	mbr	mbl	sir	sil
101	rm	mbr	mbl	sil	sir
110	gm	mbr	mbl	glr	gll
111	pm	mbr	mbl	iopr	iopl

Tableau 3: destinations des résultats POS.

D	Mné	résultat de droite	résultat de gauche
000	wr	BR(aa)	BL(aa)
001	iwr	BL(aa)	BR(aa)
010	wm	mar	mal
011	iwm	mal	mar
100	wp	voisin droite	voisin gauche
101	iwp	voisin gauche	voisin droite
110	wfp	sn?ior:rfl	sn?iol:lfl
111	wtp	sn?tpr:bpr	sn?tpl:bpl

Tableau 4: opérations logiques POI.

code	Mném	résultat
0000	z	0
0001	nor	nor(a,b)
0010	lt	and(not(a),b)
0011	na	not(a)
0100	gt	and(a,not(b)
0101	nb	not(b)
0110	xor	xor(a,b)
0111	nand	nand(a,b)
1000	and	and(a,b)
1001	nxor	not(xor(a,b))
1010	b	b
1011	le	or(not(a),b)
1100	#	a
1101	ge	or(a,not(b))
1110	or	or(a,b)
1111	one	1

Tableau 5: les indicateurs.

adresse	indicateur
000	sortie résultat de l'opérateur de droite (POS)
001	sortie résultat de l'opérateur de gauche (POS)
010	sortie retenue de l'opérateur de droite (POS)
011	sortie retenue de l'opérateur de gauche (POS)
100	indicateur file-droite vide
101	indicateur file-gauche vide
110	indicateur file-droite pleine
111	indicateur file-gauche pleine

Tableau 6: les entrées/sorties.

IO[1]	operation
0	NOP
1	si(iol) alors iopl= résultat gauche si(ior) alors iopr=résultat droite

IO[0]	operation
0	NOP
1	si(iol) alors forcer en sortie mar à iopr si(ior) alors forcer en sortie mal à iopl

Tableau 7: modes conditions.

cc	Mné	commentaire
00	sm2	second mode séparé
01	sm	premier mode séparé
10	nc	mode normal
11	cm	mode commun

ISSN 0249 - 6399