HAL
open science

Définition fonctionnelle, évaluation et programmation d'une architecture massivement parallèle

Pascal Rubini

To cite this version:

Pascal Rubini. Définition fonctionnelle, évaluation et programmation d'une architecture massivement parallèle. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG, 1992. Français. NNT : . tel-00342041

HAL Id: tel-00342041

https://theses.hal.science/tel-00342041v1

Submitted on 26 Nov 2008

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.

316571
c1
c2
TS20185.

INSTITUT IMAG
Informatique, Mathématiques Appliquées de Grenoble
CNRS-INPG-USMG
MÉDIATHÈQUE
B.P. 53 X
38041 GRENOBLE CEDEX
FRANCE
Tél. 76.51.46.36

THÈSE

présentée par

Pascal RUBINI

pour obtenir le titre de DOCTEUR

de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 23 novembre 1988)

Spécialité : Informatique

USUEL
EXCLU DU PRÊT

Définition fonctionnelle, évaluation et programmation d'une architecture massivement parallèle

---

Date de soutenance : 29 juin 1992

Composition du jury :

Messieurs	Jacques Mossière	Président
	Guy Mazaré	Examinateurs
	Alain Guyot	
	Patrice Quinton	Rapporteurs
	Jean-Paul Sansonnet	

Thèse préparée au sein du Laboratoire de Génie Informatique

Je tiens à remercier :

Monsieur Jacques Mossières, Directeur de l'ENSIMAG, de l'honneur qu'il me fait en acceptant de présider le jury de cette thèse

Monsieur Guy Mazaré, Professeur à l'ENSIMAG, qui m'a accueilli au sein de son équipe, m'a donné l'occasion de travailler sur un sujet passionnant, et m'a guidé de ses conseils pendant les quelques années qu'aura duré cette recherche

Messieurs Jean-Paul Sansonnet, Directeur de recherches au CNRS et Directeur de du PRC "Architectures de machines nouvelles"et Patrice Quinton, Directeur de Recherches au CNRS et Directeur du C3, pour avoir accepté de faire partie du jury de cette thèse et d'en être les rapporteurs

Monsieur Alain Guyot, pour avoir accepté de faire partie du jury de cette thèse, ainsi que pour les conseils qu'il a bien voulu me prodiguer tout au long de ce travail

Tous les membres anciens et nouveaux du groupe "Circuits intégrés" qui par idées, avis et critiques m'ont aidé à avancer, ainsi que tous ceux qui pendant deux ans et des poussières m'ont supporté, dans tous les sens du terme

Et enfin tout spécialement le Ministère de la Défense qui, malgré son inclinaison naturelle à être arrangeant avec les jeunes étudiants, a su comprendre le parti que je pouvais tirer d'une date "butoir" pour accélerer la rédaction de ce mémoire et m'a gratifié, en final, d'un séjour dans un club plein de gentils organisateurs toujours très soucieux d'organiser.

Table des matières

Introduction 5

Chapitre I : Le parallélisme massif 9

1. Les différentes formes de parallélisme. 10
2. Granularité. 13
3. Grandes options architecturales. 14
   3.1. SIMD, MIMD. 14
   3.2. Synchronisme, asynchronisme. 15
   3.3. Topologie. 15
4. Quelques machines fortement parallèles. 15
   4.2. "Connection machine" et autres machines SIMD. 15
   4.2. Transputers, architecture Supernode. 18
   4.3. Cosmic Cube, iPSC et NCUBE. 20
   4.4. Projets Mosaic et MEGA. 21
5. Quels moyens pour quelles fins ? Méthodologie. 22

Chapitre II : Le réseau de communication 31

A. Généralités sur les réseaux. 32
1. Topologies. 32
   1.1. Structures à bus. 32
   1.2. Structures en réseaux de liens 34
      1.2.1. Interconnexion complète 35
      1.2.2. Structures en grilles. 36
      1.2.3. Hypercubes. 38
      1.2.4. Structures arborescentes. 39
      1.2.5. Autres topologies. 39
2. Stratégies de routage. 42
3. Mode de commutation. 43
   3.1. La commutation de circuit virtuel. 43
   3.2. Le passage de message. 44
   3.3. Le Wormhole routing. 45
   3.4. Virtual cut-through. 46

B. Le cas du réseau cellulaire. 47
1. Caractérisation des applications. 47
1.1. Etude statique. 47
1.1.1. Types d'algorithmes. 49
1.1.2. Topologies logiques. 49
1.1.3. Types de données. 50
1.1.4. Niveaux de recouvrement. 50
1.2. Etude dynamique. 51
1.2.1. Influence de la latence de communication sur le ralentissement. 51
1.2.2. Analyse des comportements. 54
1.2.3. Taux d'émission et charge du réseau. 57
2. Solutions envisagées et évaluation. 59
2.1. Transfert parallèle. 59
2.2. Transfert série 61
2.2.1. Présentation. 61
2.2.2. Prévention d'interblocage. 69
2.2.3. Prévention de famine. 71
3.2.4. Evaluation. 72
2.3. Transfert wormhole 77
2.3.1. Définition de variantes. 77
2.3.2. Absence d'interblocage. 81
2.3.3. Réalisation. 82
2.3.4. Evaluation. 85
2.4. Transfert par bus. 87
2.4.1. Présentation. 87
2.4.2. Réalisation. 88
2.4.3. Evaluation. 92
3. Comparaison et conclusions. 93

Chapitre III : Partie traitement 97
1. Les différents processeurs. 98
1.1. Processeurs actuels. 98
1.1.1. Processeurs CISC. 98
1.1.2. RISC 99
1.2. Processeurs 8 bits. 101
1.3. Processeurs pour machines MIMD. 101
1.3.1. Transputer. 101
1.3.2. MDP. 102
2. Fonctionnalités requises. 103
2.2. Multiprogrammer ? 106
2.3. Structures de contrôle. 108
2.4. Modes d'adressage. 110
2.5. Opérations. 111

3. Structure retenue. 111
3.1. Taille du processeur. 111
3.2. Premier jet. 113
3.2.1. Communication. 113
3.2.2. Structure générale. 118
3.2.3. Jeu d'instructions initial. 119
3.3. Améliorations introduites. 121
3.3.1. Modes d'adressage courts. 121
3.3.2. Modes d'adressage indirects. 124
3.3.4. Calcul 132
3.3.5. Communication. 134
3.3.6. Instructions de contrôle. 135
3.4. Définition de la version 1. 137
3.4.1. Chemin de données. 137
3.4.2. Jeu d'instructions. 137
4. Conclusions. 138

Chapitre IV : Programmation 139

1. Langages parallèles. 141
2. LUSTRE. 144
2.1. Description. 144
2.2. Mise en œuvre. 145
2.3. Résultats. 146
3. Approche manuelle 147
3.1. Tri de chaînes de caractères. 148
3.1.1. Tri par insertions. 148
3.1.2. Tri par interclassement. 151
3.2. Recherche de plus courts chemins dans un graphe. 153
3.3. Réseaux de neurones 160
3.4. Automates cellulaires. 164
4. Génération automatique. 172
4.1. Générateurs spécifiques. 172
4.2. Générateurs généraux. 173
4.3. Compilateurs spécifiques. 173
5. Conclusions. 174

Conclusion 175

Bibliographie 179

Annexe 1 : Applications 185

1. Conway1D et 2D 186
2. Distance 187
3. Crible d'Eratosthène. 191
4. Tris avec données en place. 195
5. Multiplication de matrices creuses. 201
6. Simulation logique à échéancier. 207
7. Problème des huit reines. 208

Annexe 2 : Environnement de programmation et de simulation 213

1. Un outil modulaire d'aide à la conception et à l'expérimentation. 214
2. L'assembleur parallèle 215
3. Le simulateur 225

Annexe 3 : Communication entre le réseau cellulaire et l'extérieur 231

1. Interface hôte / réseau. 232
2. Chargement. 236
   2.1. Utilisation de relais. 237
   2.2. Utilisation de signaux globaux. 238
   2.3. Performances. 240
3. Conclusion. 242

Introduction

L'histoire de la technologie informatique peut d'une certaine manière se décrire comme une alternance de modes. La problématique fondamentale de l'ourlet au dessus ou au dessous du genou est remplacée par celle non moins fondamentale du matériel et du logiciel, mais en fin de compte le mouvement reste le même : les concepteurs comme les grands couturiers ne font jamais qu'explorer à l'infini les variantes des intuitions géniales des pères fondateurs de leurs disciplines respectives. Parmi les modes informatiques, citons en vrac les machines langages, les RISC, les réseaux neuro-mimétiques, les automates cellulaires, l'intelligence artificielle, les langages orientés objets, les réseaux systoliques, etc. Certaines ont fait école, d'autres se sont diluées dans la culture générale et sont mises un peu à toutes les sauces, mais, avec le recul, pouvons-nous affirmer que l'une d'entre elles ait été véritablement révolutionnaire ? Quelle que soit la réponse, il est certain que la grande majorité des ordinateurs effectue encore, et pour sans doute longtemps, la fameuse boucle *fetch-execute* de von Neumann.

La présente thèse ne fera pas exception. Elle s'inscrit dans la mode dite du *parallélisme massif*, appellation un peu pompeuse qui pourrait se résumer par la question : "est-il possible de réaliser une architecture dont la puissance de calcul effective serait extensible à l'infini par addition d'éléments réguliers ?".

Dans cette formulation, deux termes sont particulièrement importants. Le premier est *puissance effective*. Il ne s'agit pas simplement de juxtaposer n processeurs de puissance p et d'affirmer que nous obtenons une machine de puissance $n.p$, encore faut-il que celle-ci puisse être mise en œuvre pour concourir au calcul de problèmes réels dont l'implémentation sur machine séquentielle ne donne pas des performances satisfaisantes. Il faut prouver que la machine peut être *programmée* et cette opération n'a rien de trivial. Les problèmes doivent receler du parallélisme, c'est à dire comporter des tâches exécutables simultanément (l'une ne dépendant pas du résultat de l'autre). Il est évident que tous n'en comporteront pas en quantité suffisante pour alimenter un grand nombre de processeurs, mais ce sera en général le cas de ceux pour lesquels une machine séquentielle est trop lente. Les tâches, les *processus*, ne sont pas toutes indépendantes entres elles et l'expression de ces dépendances prend la forme d'opérations de *communication*. La machine parallèle doit permettre ces communications.

Le second terme important est *éléments réguliers*. En effet, s'il est raisonnable de concevoir quelques dizaines d'éléments différents pour réaliser un calculateur, la conception de plusieurs milliers d'éléments différents sera toujours d'un coût rédhibitoire et de ce fait, une bonne régularité des composants est indispensable.

Notre équipe étudie depuis plusieurs années une architecture, baptisée *réseau cellulaire*, qui appartient à la classe des machines massivement parallèles. Dès la naissance de ce projet, la simplicité de réalisation a été placée au centre de nos préoccupations puisqu'un réseau cellulaire est une interconnexion d'unités de calcul, les cellules, en tout point identiques entre elles. Les cellules sont disposées dans un plan et connectées avec leurs voisines immédiates selon un schéma en grille orthogonale NEWS (*North, East, West, South*); elles calculent de façon asynchrone, l'architecture entre donc dans la catégorie des machines MIMD (Multiple Data Multiple Instruction). La communication, elle aussi, est asynchrone puisqu'elle se fait par envoi de messages. Les premiers réseaux cellulaires étudiés étaient des machines dédiées à des applications données (simulation logique [BER85] [OBJ88], placement par recuit simulé [COR88], reconstruction d'images en tomographie [LAT89], réseaux neuro-mimétiques [FAU90]) et un certain nombre de circuits VLSI ont été réalisés. Toutefois, le coût de développement de ces architectures dédiées, leur manque de généralité, ainsi qu'une taille des cellules obtenues (entre 10000 et 20000 transistors) comparable à celle d'un microprocesseur, nous a poussé à envisager l'étude d'un réseau cellulaire programmable non dédié.

La présente étude qui se centre sur la définition fonctionnelle de cette architecture généraliste et sur sa programmation, se donne comme but la réalisation d'un équilibre

entre les différentes composantes (réseau de communication, partie traitement, capacité à être programmée, interface avec l'extérieur) pour obtenir un rapport coût/performance optimum. Elle s'insère dans le cadre d'un travail d'équipe :

- Eric Payan a principalement centré son attention sur l'implémentation du langage data-flow LUSTRE sur le réseau [PAY91], mais a aussi activement participé aux premières étapes de la définition fonctionnelle du processeur
- Mahmoud Karabernou, après avoir effectué l'analyse de bas niveau de la cellule, s'attache à mener à bien une première version sur silicium du circuit
- Youssef Latrous étudie l'implémentation de langages d'acteurs sur une structure cellulaire
- Khalid Khoumsi a évalué une implémentation de la structure de routeur à bus introduite ici (chap. II) et a ainsi permis une comparaison avec le routeur actuellement utilisé.
- Chaouki Aktouf s'intéresse actuellement au test du réseau cellulaire [ATK91].
- Marc Robichon et Frédéric Boiteux [BOI91] se sont intéressé lors de leur stage de troisième année d'école d'ingénieur, à la réalisation matérielle de l'interface hôte/réseau et au chargement des programmes avec, à la clef, une maquette matérielle et logicielle opérationnelle pour Macintosh II. Ils se sont aussi penchés sur la programmation avec le codage d'un tri de chaînes par interclassement.
- Olivier Jacquiot a travaillé sur le problème de la simulation logique à échéancier répartie [JAC91].

Dans ce travail de définition fonctionnelle, un souci de cohérence de l'ensemble des différentes composantes fait que les analyses des différentes parties sont interdépendantes et ont du être menées de front. Il n'est malheureusement pas possible de les présenter ici d'une façon entrelacée qui serait fidèle à la démarche mais compromettrait la clarté de l'exposé. Nous devrons donc les ordonner arbitrairement, tout en faisant apparaître, lorsque c'est nécessaire, leurs interactions. L'ordre de présentation choisi est ascendant, non par choix méthodologique, mais parce qu'il nous semble être le plus facile à suivre pour le lecteur. Il nous a aussi paru souhaitable de scinder les études bibliographiques parallèlement au découpage en parties de l'étude fonctionnelle, car ces études ne se recoupent généralement pas. Nous espérons que cette structure de document ne déroutera pas trop le lecteur, généralement plutôt habitué à l'ordre global traditionnel *étude bibliographique - analyse - réalisation -résultats.*

Le chapitre I reprendra pour les expliciter les notions de parallélisme et de grain de parallélisme et situera sous cet angle notre architecture par rapport aux autres architectures fortement parallèles. Nous expliciterons les interdépendances entre programmation, communication et partie traitement et en déduirons une méthodologie adaptée pour la suite.

Par le chapitre II débutera l'étude fonctionnelle proprement dite, avec l'analyse du "cahier des charges" du système de communication et des problèmes présentés par sa réalisation. Plusieurs techniques de commutation et plusieurs structures de routeur seront envisagées et comparées. Débuter par la communication est aussi arbitraire que l'ordre général ascendant, mais nous pensons que la conception du processeur étant assez classique (8 bits de type Motorola 6800) et celle du réseau beaucoup moins, cet ordre sera plus facile à appréhender que l'ordre inverse.

Le chapitre III traitera de la partie traitement et plus spécialement du jeu d'instructions et du modèle de programmation du processeur. Là aussi plusieurs options architecturales seront comparées, sur la base des caractéristiques des programmes.

Le chapitre IV abordera le délicat problème de la conception de programmes parallèles pour le réseau cellulaire, des méthodologies et langages à mettre en œuvre. Mais en tout état de cause nous n'épuiserons pas le sujet qui déborde largement du cadre d'une thèse. Les exemples les plus intéressants y seront détaillés, les autres étant regroupés en annexe 1.

Que ce soit pour le système de routage, pour le processeur ou pour la programmation, une approche expérimentale à base de simulations a été utilisée. Un assembleur "parallèle" et un simulateur (niveau cycle machine) ont été écrits dans le cadre de ce travail de recherche et sont la source des résultats présentés ici. Ces outils ont aussi servi de base à d'autres travaux menés au sein de notre équipe. Ils sont décrits en annexe 2.

Enfin, l'annexe 3 rassemble quelques considérations portant sur l'interfaçage du réseau cellulaire à un calculateur hôte et sur le problème du chargement des programmes que nous n'avons pas jugé possible de faire figurer dans le corps du document.

Chapitre I : Le parallélisme massif

"Pour répondre à des besoins toujours croissants en puissance de calcul..." — incantation rituelle.

La réalisation matérielle d'un ensemble d'unités de traitement interconnectées n'est pas à l'heure actuelle une tâche spécialement ardue, mais elle pose un problème général de dimensionnement qui n'existe pas à un degré aussi élevé dans les machines séquentielles. Dans ces dernières, il incombe au concepteur de choisir avec une certaine cohérence la puissance des différents éléments, mais les contraintes technologiques restent prépondérantes en général. Dès lors que nous admettons le principe de multiplier les unités de traitement, le facteur de duplication devient un paramètre qui entre en concurrence directe avec la complexité de l'unité de traitement. Il s'agit de déterminer la taille des unités de traitement qui donne le meilleur rapport performance/coût, avec l'espoir que ce rapport restera indépendant de la surface totale de silicium disponible. L'objectif final est de pouvoir jouer librement sur la surface, c'est à dire le nombre de processeurs, pour construire des machines de puissance arbitrairement grandes.

Les unités de traitement séquentielles que nous savons concevoir exploitent avec un rendement sans cesse moins bon les nouvelles capacités d'intégration offertes par la technologie. Ainsi, la structure des processeurs étant à peu près stabilisée, l'excédent de surface est utilisé pour réaliser des multiplieurs parallèles, des mémoires caches, des bancs de registres, ou encore mettre plusieurs unités de fonctionnelles spécialisées (intel i860). Tous ces éléments permettent un gain de performances intéressant, mais ponctuel, et complexifient notablement les compilateurs. En fait leur rendement général est assez faible et les performances de crête mirifiques ne sont jamais approchées par les performances réelles. Quitte à faire du parallélisme, autant le faire franchement pour pouvoir l'exploiter correctement, avec des outils appropriés.

La politique visant à favoriser de petites mais nombreuses unités de traitement semble donc promise à un plus grand avenir. Mais, de même qu'il ne suffit pas de multiplier les transistors pour calculer plus vite, la mise en œuvre d'un nombre important d'unités de traitement est conditionnée par la présence d'un parallélisme suffisant, nous pourrions dire en quantité et en qualité, pour les alimenter en travail.

Nous rappellerons les différentes formes de parallélisme qu'il est possible d'extraire d'une application et comment elles peuvent être exploitées.

Toutes les applications présenteront bien entendu des caractéristiques propres, du point de vue des opportunités de parallélisme, entre lesquelles les architectes doivent faire un compromis. Ces derniers raisonnent non sur les applications, qui ne peuvent être connues de façon exhaustive, mais sur les caractéristiques qu'ils leurs supposent. Ces hypothèses peuvent être mises en relations avec les grandes options de conception, que nous présenterons ainsi que quelques machines qui les mettent en œuvre.

Nous finirons ce chapitre par un exposé de l'acquis propre à notre projet de réseau cellulaire, ainsi qu'un examen des problèmes méthodologiques auxquels nous avons dû faire face.

1. Les différentes formes de parallélisme.

Historiquement, la plus ancienne forme de parallélisme exploitée résulte de la décomposition des opérations de base en circuits logiques binaires : l'additionneur 8 bits est formé de huit additionneurs 1 bits fonctionnant en parallèle (plus ou moins, d'ailleurs, à cause des propagations de retenues). La concurrence induite était fortement liée à la capacité d'intégration disponible, 8, puis 16 et enfin 32 bits. Actuellement, nous pourrions aller bien plus loin, mais le besoin ne s'en fait pas sentir : l'expression de plus grands nombres est rarement nécessaire. Il s'agit d'un parallélisme interne qui concerne très peu le programmeur à partir du moment où celui-ci dispose d'une précision satisfaisante, puisqu'il raisonne généralement au niveau du mot plutôt qu'au niveau du bit.

Avec les premiers calculateurs multitâches en temps partagé des années 60, il a été possible de mettre en œuvre un parallélisme de contrôle, c'est à dire exécuter en concurrence des séquences indépendantes. Bien sûr, le parallélisme ne pouvait être réel ; seul le respect de sa sémantique était exploité pour simplifier l'expression de certains problèmes.

La panoplie d'outil de contrôle de la concurrence développée à cette époque (sémaphores, moniteurs, etc.) ne s'applique guère qu'à des architectures à mémoire centralisée et intéresse peu le parallélisme massif, mais cette génération de matériel a permis aux programmeurs de se familiariser avec l'idée même de concurrence d'exécution, qui fait maintenant parti de la culture générale informatique. Actuellement ncore, l'essentiel de l'enseignement de la multiprogrammation est centré sur cet aspect, les autres gardant un certain parfum d'exotisme dans les cursus.

En raison du coût encore prohibitif du matériel et en dépit de quelques tentatives héroïques comme l'ILLIAC IV, l'étape décisive suivante n'a pas été le passage à des structures multiprocesseurs. Le point d'orgue des années 70 a été la naissance des premiers calculateurs "vectoriels", avec le CRAY-1. Le parallélisme exploité par ces architectures est le parallélisme de recouvrement, plus connu sous le nom de *pipeline*. Son principe est de diviser les opérateurs comme l'additionneur en virgule flottante en plusieurs sous-opérations implémentées par des étages matériels distincts et de faire fonctionner ces étages à la chaîne, chacun traitant partiellement à un instant donné un jeu de données différent avant de passer le résultat de son travail à l'étage suivant. Associé à une technologie de pointe (AsGa cadencé à 1 GHz pour le Cray-3), les performances obtenues sont spectaculaires, au point que vingt ans après , cette technique est encore à la base des calculateurs les plus performants et les plus opérationnels.

Conceptuellement, le parallélisme exploité est très limité (par la profondeur des pipelines) et de caractère rudimentaire. Par analogie avec le vocabulaire de la compilation nous pourrions dire qu'il s'agit là d'un parallélisme *peep-hole* : les opportunités de recouvrement sont repérées sur une base strictement locale, en ignorant totalement la structure algorithmique du programme. Toute dépendance entre deux opérations trop proches l'une de l'autre est susceptible de créer un "trou" dans les pipelines, de sorte qu'une réorganisation des opérations doit être faite par le compilateur pour rendre plus adéquat le flux d'opérations par rapport à l'architecture des opérateurs. Il y a là tout un ensemble de difficultés qui rend la réalisation des compilateurs très délicate.

La technique du pipeline est maintenant complètement généralisée, que les microprocesseurs soient RISC ou CISC, car elle permet des gains substantiels de performances à un coût matériel assez léger, mais en fin de compte, ce gain reste ponctuel. Les processeurs vectoriels sont arrivés à maturité, ils n'évolueront plus beaucoup. Les nouvelles machines des compagnies comme Cray Technology sont un

aveu implicite de ce fait : les nouveaux produits sont obtenus par multiplication des unités vectorielles des monoprocesseurs (le Cray X-MP est constitué de 4 processeurs Cray-1) ou par changement de technologie d'intégration (le Cray-3 est un Cray-2 en AsGa).

Dans le même temps, la baisse spectaculaire du coût d'intégration dans les technologies grand-public (VLSI) a remis à l'ordre du jour, au début des années 80, l'étude des machines multiprocesseurs de grande dimension, et de nombreuses voies on été explorées durant la décennie. La plus immédiate, mais aussi la plus limitée à terme, est celle qui consiste à multiplier les processeurs autour d'une mémoire centralisée. La mémoire permet une coopération facile et une programmation relativement peu modifiée, exploitant les vieilles techniques de contrôle de concurrence ; toutefois elle est aussi un goulot qui, bien qu'élargi par divers artifices (caches, mémoire multi-bancs...), reste incontournable. A l'heure où les processeurs sont tellement rapides qu'il devient difficile d'en alimenter un seul, la voie des machines à mémoire partagée n'est sous doute pas la plus prometteuse.

La contention d'accès à la mémoire peut être maîtrisée en la distribuant au niveau de chaque processeur et en interconnectant ces derniers. Les organisations possibles seront examinées dans le chapitre suivant. La solution matérielle la plus praticable pour un grand nombre de processeurs est une mémoire privée distribuée, mais la disparition d'une structure de données commune pose le problème de sa répartition : chaque processeur doit avoir dans sa mémoire toutes les données nécessaires à sa tâche sans passer par une duplication de la structure de donnée au niveau de chaque processeur qui poserait des problèmes de coût, d'encombrement et surtout de maintien de la cohérence entre les divers exemplaires. Néanmoins, les échanges de données entre processeurs doivent être limités le plus possible car la bande passante du réseau reste une ressource contraignante. Il est donc nécessaire d'analyser et de comprendre la nature du parallélisme sous-jacent des applications.

Il est généralement admis de distinguer le parallélisme de contrôle (algorithmique), où plusieurs tâches hétérogènes sont exécutées simultanément et le parallélisme de données (géométrique), où le même traitement est appliqué simultanément à chaque élément d'un ensemble de données. Cette classification est assez floue, le parallélisme géométrique pouvant être vu comme une forme particulière de parallélisme de contrôle où les tâches sont identiques et les données différentes. D'autre part, le parallélisme pipeline est en général de type algorithmique, mais, lorsqu'il s'applique sur des boucles de corps très courts, il devient incontestablement géométrique. En excluant le parallélisme de données du parallélisme de contrôle et en réservant ce dernier terme pour les cas où les tâches sont différentes mais portent sur des données plus ou moins communes, il apparaît que le parallélisme de données est en nombre de processus supérieur de plusieurs ordres de grandeur.

2. Granularité.

Si on s'intéresse maintenant à la taille des processus, à travers la notion de *granularité*, la situation est assez obscure car personne ne parle de la même chose. Désigner les programmes à gros processus comme relevant d'un parallélisme à gros grain et les autres comme relevant d'un grain moyen ou fin n'avance pas à grand-chose : quelle grandeur doit-on mesurer ? le nombre de d'instructions, le temps d'exécution ? où poser la frontière ? Nous pensons que la notion de granularité est plutôt relative à la méthode d'analyse employée pour extraire le parallélisme de l'application.

Prenons l'exemple d'une boucle simple :

FOR I=1 TO 100 DO T(I)

Avec 10 processeurs U1 à U10, le découpage à gros grain va produire, par tronçonnage de la boucle :

P_1 : T(1) ... T(10) $\rightarrow U_1$
P_2 : T(11) ... T(20) $\rightarrow U_2$
...
P_{10} : T(91) ... T(100) $\rightarrow U_{10}$

Le découpage à grain fin cherche dans une première phase à produire le maximum de processus P_1 à P_{100}, puis multiplexe ceux-ci en fonction du nombre de processeurs disponibles.

$$\left.\begin{array}{l} T_1 : P_1 \\ T_2 : P_2 \\ \ldots \\ T_{10} : P_{10} \end{array}\right\} \rightarrow U_1 \quad \left.\begin{array}{l} T_{11} : P_{11} \\ T_{12} : P_{12} \\ \ldots \\ T_{20} : P_{20} \end{array}\right\} \rightarrow U_2 \ \ldots \ \left.\begin{array}{l} T_1 : P_{91} \\ T_{92} : P_{92} \\ \ldots \\ T_{100} : P_{100} \end{array}\right\} \rightarrow U_{10}$$

A nombre égal de processeurs, ces deux approches donnent la même assignation des traitements, mais on mettra en œuvre l'une ou l'autre, suivant le rapport entre le nombre de processeurs et le nombre de processus, et la régularité de ces derniers. Le découpage à grain fin s'adapte à n'importe quel nombre de processus, mais le regroupement a posteriori peut être parfois problématique : si la structure de processus est simple, comme dans l'exemple précédent, le regroupement est trivial et le sur-coût lié à la multiprogrammation interne au processeur peu être évité, mais cette structure n'est pas toujours simple. Le découpage à gros grain ne présente pas tels sur-coûts, mais la construction de gros processus est parfois une tâche impossible à mener à bien.

3. Grandes options architecturales.

Le type de parallélisme visé (géométrique ou algorithmique), la régularité spatiale et temporelle de la structure de processus vont induire des architectures différentes que nous pouvons ramener à quelques grandes options : SIMD ou MIMD, topologie de connectivité faible ou forte, statique ou dynamique, communication synchrone ou asynchrone, parallélisme statique ou dynamique.

3.1. SIMD, MIMD.

Introduite par la classification de Flynn [FLY72], cette distinction concerne l'origine d'un flux d'instructions. Dans le mode SIMD (Single Instruction Multiple Data) le même flux d'instructions est distribué par un séquenceur central vers tous les processeurs qui l'exécutent sur des données différentes. Dans le mode MIMD (Multiple Instructions Multiple Data), chaque processeur dispose de son propre séquenceur et de son propre programme. Cette définition est matérielle, et non fonctionnelle car, comme le note W.D. Hillis, le concepteur de la Connection Machine, il est toujours possible d'émuler une machine MIMD avec une machine SIMD en faisant exécuter par chaque processeur un interpréteur qui va transformer les données en programmes et, inversement, d'émuler une machine SIMD avec une MIMD en donnant le même programme à exécuter à chaque processeur [HIL85, p. 43].

Traditionnellement, la souplesse est obtenue dans les machines SIMD non par interprétation, mais à l'aide d'un inhibiteur local d'exécution. Les divergences dans les flux d'instructions des processeurs sont traitées séquentiellement et non pas en parallèle comme en MIMD. Par exemple une structure IF donne les chronogrammes d'exécution suivants :

IF condition THEN T_1 ELSE T_2

SIMD :		proc. où la condition est vérifiée	proc. où la condition n'est pas vérifiée
	1)	T_1	*rien* (T_1 inhibé)
	2)	*rien* (T_2 inhibé)	T_2
MIMD :		proc. où la condition est vérifiée	proc. où la condition n'est pas vérifiée
	1)	T_1	T_2

Cela relativise la portée de la remarque de Hillis : l'interprétation étant une sorte de grand aiguillage, la programmer par inhibition reviendrait à avoir une activité inversement propotionnelle au nombre d'instructions comprises dans le jeu.

3.2. Synchronisme, asynchronisme.

Outre l'asynchronisme d'exécution que nous venons de voir, il faut prendre en compte un éventuel asynchronisme dans la communication. Lorsque deux processus, l'un émetteur, l'autre récepteur, veulent entrer en communication, ils peuvent procéder de deux façons. La plus simple consiste à imposer un *rendez-vous* entre les deux acteurs : les données ne seront échangées que lorsque l'émetteur sera sur l'instruction d'émission et le récepteur sur l'instruction de réception, le premier arrivé devant attendre l'autre. C'est la solution adoptée dans le transputer d'Imnos (§ 4.2). Elle présente le risque d'une restriction de la concurrence, est difficile à mettre en œuvre lorsque les deux dispositifs ne sont pas directement connectés (voir chap. II, § 3.1, commutation de circuits virtuels).

L'autre méthode est celle du passage de message. L'émetteur envoie ses données dès qu'il est prêt à la faire, indépendamment de l'état du récepteur ; il peut ensuite vaquer à ses occupations. Le récepteur reste par contre potentiellement bloqué s'il devance l'émetteur. Le découplage temporel entre les deux opérations nécessite une mémorisation intermédiaire et, en conséquence, un contrôle de flux pour limiter un émetteur trop productif.

3.3. Topologie.

Le choix d'un graphe d'interconnexion des unités de traitement est l'un des plus difficiles à faire car de nombreux paramètres entrent en jeu. Nous les étudierons en détail dans le chapitre suivant. Les topologies actuellement utilisées se divisent en trois groupes : grilles, hypercubes et réseaux reconfigurables par commutateurs.

4. Quelques machines fortement parallèles.

Loin de faire ici un inventaire complet des projets se revendiquant du parallélisme massif, nous avons voulu donner un aperçu des principales classes de machines afin de donner au lecteur des points de repère. Nous porterons une attention particulière aux travaux menés au California Institute of Technology (Caltech) dans ce domaine afin de rendre justice à leur caractère précurseur, de montrer leur influence sur notre projet.

4.2. "Connection machine" et autres machines SIMD.[1]

Les machines SIMD sont par excellence destinées à l'exploitation d'un parallélisme géométrique. Le synchronisme d'exécution renvoie à la similitude des tâches à exécuter, la simplicité de l'unité de traitement (pas de séquenceur ni de

---

1 bibliographie : [TRE91, HIL85, TUC88, DEL90]

mémoire de programme) qui permet la réalisation à un faible coût de machine à grande échelle répond à l'ordre de grandeur de concurrence permis par ce type de parallélisme.

L'une des premières architectures de ce type commercialisée est le Distributed Array Processor (DAP) d'ICL puis d'Active Memory Technology (AMT). Le DAP, dans sa version actuelle, est constitué de 4096 processeurs 1 bit interconnecté en grille 2D et à une grille de bus. Chaque processeur est muni d'une mémoire privé de 32 Kbits à 1Mbits et, dans certaines versions, possède un coprocesseur 8 bits destiné au calcul numérique.

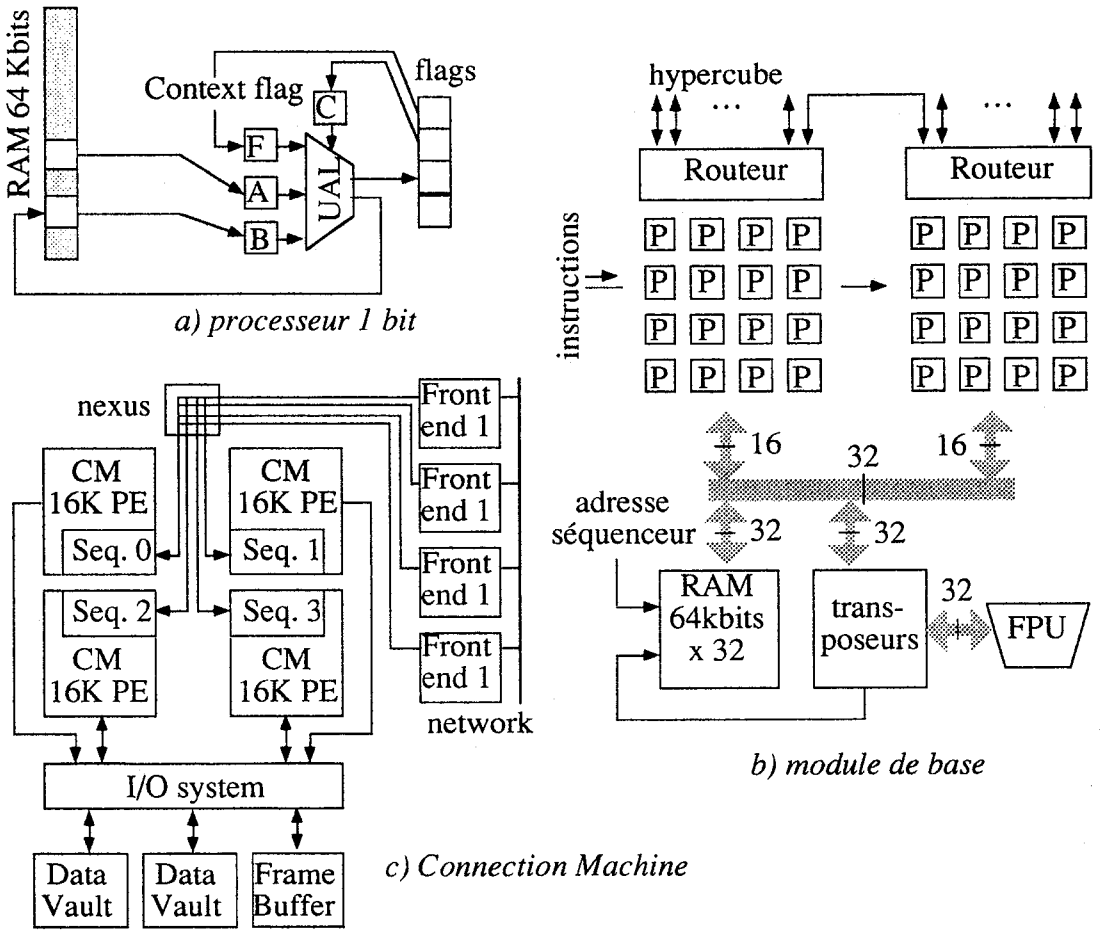


Fig. 1 — *Structure de la Connection Machine 2.*

La "Connection Machine" de Thinking Machine Corp., initialement développée au MIT, est une architecture qui intègre jusqu'à 64K processeurs 1 bits, interconnectés en hypercube (la CM-1 possédait un autre niveau de communication, en grille 2D ; il a été supprimé dans la CM-2 au profit d'un niveau en grille de dimension quelconque bâtie sur le réseau hypercubique). La figure 1 montre les différents niveaux architecturaux de la Connection Machine. Le "processeur" *(fig. 1a)* est constitué d'une UAL 1 bits capable de réaliser n'importe quelle fonction logique de $\{0,1\}^3$ dans $\{0,1\}^2$,

avec une éventuelle inhibition par un flag de contexte. Les opérandes sont pris dans la mémoire (A et B) et parmi les flags du processeur ; les résultats sont rangés dans un flag et à l'adresse de l'une des opérandes (éventuellement). Les processeurs sont regroupés par 16 sur un circuit VLSI custom où tous sont interconnectés directement et se partagent un routeur. Les routeurs sont reliés entre eux pour former un hypercube de degré 12. Ces groupes de 16 couples processeur-routeur sont à leur tour regroupés par deux pour se partager une mémoire et un coprocesseur en virgule flottante Weitek *(fig. 1b)*. La mémoire possède un front de 32 bits, chaque bit est distribué à un processeur différent. Le FPU étant un circuit opérant sur des arguments reçus en parallèle, il faut une interface pour passer d'une mémoire série à des arguments parallèles : le transposeur. Au niveau le plus global, les modules de bases sont organisés en machines de 16K processeurs alimentés par un séquenceur. Une Connection Machine complète en comprend quatre, reliées à jusqu'à quatre calculateurs hôtes, ainsi qu'à des disques rapides et à une mémoire d'écran, par un système d'entrées/sorties *(fig. 1c)*.

La communication se fait par envois de messages, ce qui est nécessaire dans la topologie hypercubique adoptée pour permettre à toute paire de processeurs de communiquer, mais relativement surprenant dans une machine SIMD : comment concilier une communication asynchrone (on ne connaît pas à l'avance le temps d'acheminement d'un message) et une exécution synchrone ? Il est nécessaire de planifier à l'avance comment se dérouleront les acheminements, quelles seront les collisions, pour connaître le temps global de communication et ainsi revenir dans le cadre du synchronisme. Notons que la communication de processeur à processeur se double d'un système de diffusion et d'un système de OU global destiné à la détection d'erreur ou de terminaison.

La Connection Machine permet de s'abstraire du nombre de processeurs grâce à un procédé logiciel : les processeurs virtuels (VP). Chaque processeur physique peut recevoir un nombre quelconque de processeurs virtuels limité seulement par la taille de la mémoire. Chaque processeur virtuel apparaît au programmeur comme un processeur réel.

Initialement conçue pour des applications dans le domaine de l'intelligence artificielle (d'où le nom de la compagnie), la Connection Machine était d'abord programmée à partir d'un environnement *LISP, supporté par un hôte Symbolics. On peut maintenant utiliser des dialectes de divers langages procéduraux classiques (C*, FORTRAN-90) ou programmer directement avec un assembleur évolué (ParIS). Tous ces langages recouvrent plus ou moins la même philosophie qui sera décrite et analysée dans le chapitre IV. La qualité de l'environnement logiciel est sans doute à l'origine du succès de cette machine SIMD qui est de loin la plus connue de sa classe.

Pourtant, lorsqu'on considère avec un peu de recul l'architecture de la CM, on ne peut qu'être frappé par la disproportion entre la simplicité extrême du processeur et la complexité du système de communication, par la faiblesse du taux d'activité des FPU, limité à entre 5 et 10% par l'accès sériel à la mémoire.

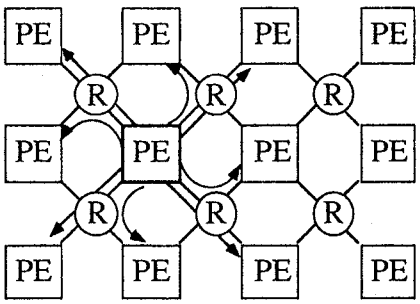


Fig. 2 — *Topologie de la MP1 (X-Net).*

Le processeur de la MP-1, de Maspar Computer corp., est très différent de celui des deux précédentes machines. Il contient une unité logique 1 bit, une unité arithmétique 4 bits et deux unités destinées au calcul en virgule flottante : 64 bits (mantisse) et 16 bits (exposant). La mémoire associée à chaque processeur est de 16 Ko. La topologie est un peu inusitée : chaque PE est connecté à ses 4 voisins diagonaux, avec un point de routage à chaque intersection de liens *(fig. 2)*. La topologie logique vue par le programmeur est alors une connexion aux huit voisins immédiats. Un système de routage global permet des communications distantes.

Ces architectures partagent certains points communs. Elles sont toutes réalisées dans des technologies éprouvées, associant des composants VLSI spécifiques (regroupant plusieurs processeurs) et des composant du commerce (mémoires RAM). Les machines obtenues ont généralement des fréquences de fonctionnement assez faibles qu'elles compensent par un grand nombre de processeurs et une réalisation bon marché. Autre point de convergence remarquable, elles semblent toutes hésiter entre le domaine du calcul numérique intensif en virgule flottante et le domaine du calcul booléen (traitement d'image), ce qui donne des architectures de PE un peu hybrides.

4.2. Transputers, architecture Supernode.

Étudié dans le cadre du projet européen ESPRIT de supercalculateur massivement parallèle de faible coût, le Supernode est l'une des nombreuses architectures construites à base de transputers. Le transputer, conçu et fabriqué par la société britannique INMOS, est un processeur RISC associé à une mémoire locale et doté de fonctionnalités de communication novatrices : quatre liens bit-série qui permettent d'interconnecter directement entre eux plusieurs transputers. Le processeur, bien que généralement qualifié de RISC[1], fourni des instructions de haut niveau pour supporter plusieurs processus en temps partagé (par tranche de temps), ce qui permet d'exécuter un même programme parallèle indifféremment sur un ou plusieurs transputers. Plusieurs modèles de transputers existent , qui vont du modeste 16 bits T212 au dernier né, le

[1] RISC : *Reduced Instruction Set Computer* (cf. chapitre 3).

T9000, crédité (par le constructeur) d'une puissance de crête de 200 Mips[1] et 25 MFlops[2]. Le langage de programmation OCCAM, qui est à la base de cette famille de circuits, est brièvement présenté dans le chapitre 4.

On comprend aisément pourquoi ce composant intéresse au plus haut point les architectes de machines puisqu'il constitue une sorte de brique de base (traitement, communication) qu'il suffit d'assembler pour obtenir un système minimum. C'est selon cette méthode où la topologie d'interconnexion est considérée comme fixe que les premiers calculateurs à base de transputers ont été construits (série T de Floating Point Systems). La topologie, généralement plane ou hypercubique, n'est malheureusement pas cohérente avec les capacités de communication incluses dans le transputer, car rien n'y est prévu pour assurer la communication entre deux transputers non directement connectés, le dialogue étant synchrone (mécanisme de rendez-vous). C'est pourquoi on a développé la possibilité d'interconnecter directement toute paire de transputer via un commutateur reconfigurable (*switch*).

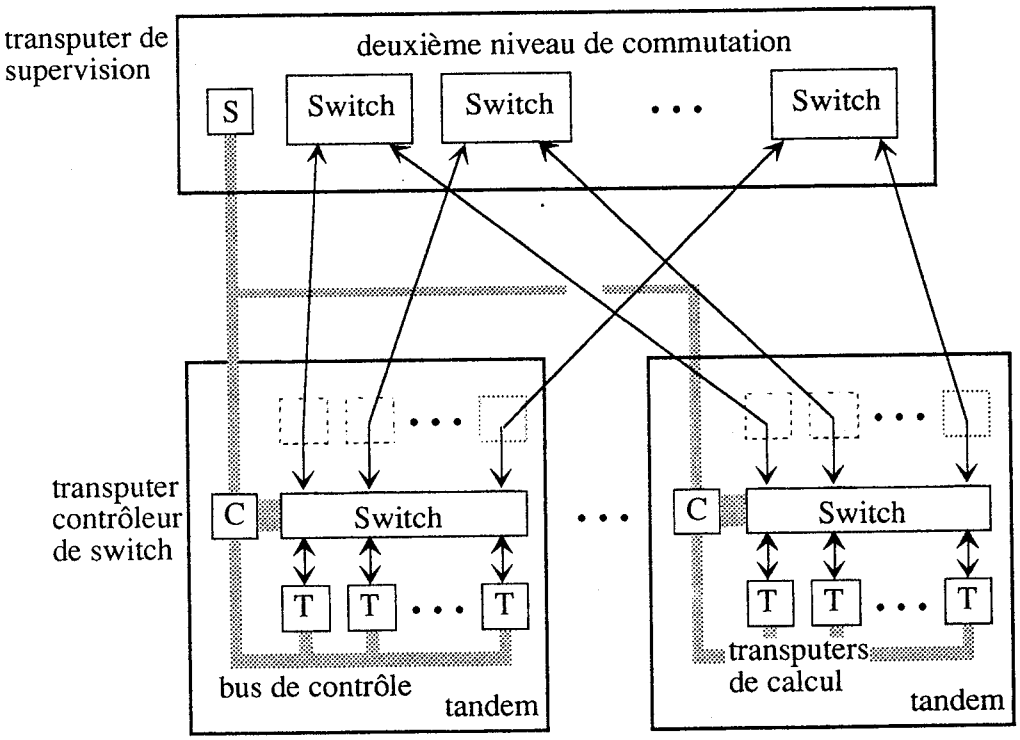


Fig. 3 — *Architecture hiérarchique du Supernode.*

L'architecture Supernode [HAR86, MUN90] est de ce type *(fig. 3)*. Elle est constituée par module de base, le *tandem node*, qui regroupe entre 32 et 64 transputers T800 reliés entre eux par l'intermédiaire d'un commutateur réalisé à l'aide de réseaux de Clos et commandé par un transputer dédié. Le tandem le plus petit représentant de la famille supernode. Des machines de plus grande taille peuvent être réalisées en

---

1 MIPS : millions d'instructions exécutées par seconde.

2 MFLOPS : millions d'opérations en virgule flottante par seconde.

interconnectant des tandems au moyen d'un commutateur de second niveau, ce qui autorise des ensembles de 1024 transputers au plus. Les machines Supernode sont actuellement commercialisées par les sociétés Telmat et Thorn-EMI.

4.3. Cosmic Cube, iPSC et NCUBE.[1]

Développé au Caltech au début des années 80, le Cosmic Cube est une machine MIMD à topologie hypercubique comprenant jusqu'à 128 PE. Les processeurs élémentaires sont constitués par un couple 8086/8087 associé à quelques K de mémoire RAM. En plus du réseau d'interconnexion, tous les nœuds sont couplés à un bus Ethernet qui permet la communication avec l'hôte.

La communication est faite par passage de message. Dans la première version de Cosmic Cube, le mode de commutation utilisé est un banal *store-and-forward* et le routage est entièrement assuré par logiciel. L'iPSC/1, d'Intel Scientific Computers, en est une version commerciale où le couple 8086/8087 a été réactualisé avec les 16 bits 80286/80287 et une mémoire de 512 Ko. Une carte d'extension pour le calcul vectoriel peut être ajoutée à chaque nœud.

L'indigence de la vitesse d'acheminement induit sur cette première génération des délais de communication trop importants pour permettre l'exploitation de la puissance de la machine. La seconde génération (iPSC/2), outre un passage au couple 80386/80387, apporte un système de routage entièrement matériel : un module de routage "Direct-Connect Module" implémente la technique de commutation *wormhole* (chap. II, § 3.3) sur silicium. La dernière version, l'iPSC/860, remplace les processeurs par l'i860 qui permet des performances accrues en matière de calcul numérique, et gonfle la mémoire jusqu'à 16 Mo.

Le NCUBE, de NCUBE corp., s'inspire également du Cosmic Cube, mais a été conçu indépendamment. Les petits processeurs d'Intel ont été délaissés au profit d'un processeur spécifique 32 bits proche de celui du VAX, associé à une FPU 64 bits. Le transfert entre nœuds de l'hypercube se fait grâce à un mécanisme de DMA. Le NCUBE-2, version actuelle, utilise jusqu'à 8192 processeurs VAX-like 64 bits et offre la puissance de calcul fabuleuse, du moins sur le papier.

Le Cosmic Cube et ses descendants ont été riches d'enseignements, tant sur le plan logiciel que matériel :

- une latence de communication faible est l'un facteur les plus fondamentaux dans ce type de système, c'est d'elle que dépend le degré de parallélisme utilisable ;
- la topologie en hypercube n'est pas la meilleure, les projets actuels du Caltech (Mosaic) et d'Intel (TouchStone) tablent tout deux sur des topologies de bidimensionnelles ;

---

[1] bibliographie : [TRE91, LUT84, SEI90, FLA87]

- ce type d'architectures semble incapable de pénétrer le marché du calcul numérique malgré la puissance offerte, essentiellement à cause de la difficulté de parallélisation des applications actuelles, plus coûteuse qu'en mode SIMD ;
- elle peut manifestement être programmée, mais la parallélisation est à la charge de l'utilisateur ; comme les langages pour machines SIMD incluent des appels à des procédures parallèles, les langages pour machines MIMD incluent des procédures de passage de message ;
- des langages à sémantique parallèle ont été expérimentés avec succès (CANTOR).

4.4. Projets Mosaic et MEGA.

Le projet Mosaic [SEI90], toujours du Caltech, prend le contre-pied de l'évolution vers de gros processeurs qui caractérise le Cosmic Cube. Ce dernier était handicapé par la taille des réseaux réalisables sous forme d'hypercube. Le Mosaic revient à une topologie bi- ou tri-dimensionnelle, permettant une extension simple par juxtaposition, et cherche à intégrer l'ensemble du nœud dans un seul circuit, voire d'en mettre plusieurs. Le PE du Mosaic C contient donc :

- un processeur RISC 16 bits
- un routeur *wormhole* [FLA87]
- 8 Ko de ROM
- 16 Ko de RAM dynamique

L'ensemble, en technologie CMOS 1.6 µm, occupe environ 50 mm^2 de silicium, ce qui reste tout à fait raisonnable dans la perspective d'intégration de plusieurs PE par chip, de réalisation de machines MIMD à plusieurs dizaines de milliers de PE.

Nous pouvons noter une sévère réduction de la taille de la mémoire qui ne va pas sans poser un problème logiciel : le code résidant des systèmes d'exploitation distribués mis au point pour les cubes ne tient plus dans la mémoire ! Le grain de parallélisme exploité dans le Mosaic est beaucoup plus fin et peut entraîner une révision des méthodes de parallélisation.

Le projet MEGA [GER89], de l'université Paris-Sud, s'inscrit dans le même mouvement que le projet Mosaic, mais vise d'emblée une échelle encore plus grande : un million de processeurs 16 bits, associés chacun à 64K de mémoire vive et capable de supporter chacun 4096 processus. L'ensemble de l'unité de traitement, processeur, mémoire et système de communication, est intégré dans un seul circuit. Les PE sont interconnectés en grille 3D et communiquent par messages. L'une des originalités de ce projet est l'introduction d'une technologie de packaging tridimensionnelle, réalisant par simple emboîtement la topologie logique du réseau.

5. Quels moyens pour quelles fins ? Méthodologie.

Les tableaux 1 et 2 résument les caractéristiques des projets que nous venons de présenter, au niveau global et au niveau du processeur élémentaire. On observe un mouvement de partitionnement entre machines extensibles en nombre de PE (SIMD et Mosaic) et machines à grosses unités de traitement (machines à base de transputer, Cosmic Cube et descendants) où l'extensibilité est un objectif secondaire et seule compte la performance de crête des machines effectivement construites. D'un point de vue programmation, ces dernières se revendiquent plus d'un parallélisme à grain *moyen* qu'à grain fin, ce terme exprimant sans doute l'incapacité de ces architectures à exploiter réellement une granularité fine...

architecture	mode	taille maximale (PE)	topologie	comm.
DAP	SIMD	4096	Grille 2D	synchrone
Connection Machine 2	SIMD	65536	hypercube	asynchrone
MasPar-1	SIMD	16384	X-Net (grille 2D) + switch global	synchrone
Supernode	MIMD	1024	Reconfigurable	synchrone
NCUBE-2	MIMD	8192	Hypercube	asynchrone
iPSC/860	MIMD	128	Hypercube	asynchrone
Mosaic C	MIMD	16384	Grille 2D ou 3D	asynchrone
MEGA	MIMD	10^6	Grille 3D	asynchrone

Tab. 1 — *Comparaison des architectures au niveau global.*

architecture	processeur	mémoire / processeur	PE / chip	taille chip processeur
DAP	1 bit, coprocesseur 8 bits	de 32 Kbits à 1 Mbits	64	50000 portes
Connection Machine 2	1 bit, coprocesseur Weitek 3132 par 32 PE	de 64 Kbits à 1Mbits	16	14000 portes
MasPar-1	logique 1 bit, arithmétique 4 bits, mantisse 64 bits, exposant 16 bits	16 Ko	32	450000 T par chip
Supernode	Transputer T800 (32 bit + FPU)	jusqu'à 4 Mo	1	
NCUBE-2	32 bits VAX-like + FPU	jusqu'à 64 Mo	1	
iPSC/860	i860 et 80386	jusqu'à 16 Mo	1	
Mosaic C	16 bits	16 Ko	1	
MEGA	16 bits	64 Ko	1	

Tab. 2 — *Comparaison des processeurs élémentaires.*

Ces machines ont toutes un point commun, leur caractère non-dédié. Elles sont souvent issues de motivations plus précises, l'intelligence artificielle pour la CM et MEGA, le traitement d'image pour le DAP, les langages orientés objets pour le Cosmic Cube, mais leurs structures se sont avérées suffisamment générales pour être utilisées dans tous les domaines.

Elles sont ainsi tiraillées entre des besoins contradictoires, particulièrement en ce qui concerne les architectures SIMD. Si une ressource comme une unité de calcul en virgule flottante manque, les performances sur les applications scientifiques seront de un ou deux ordres de grandeur inférieures mais, si elle existe, de nombreuses applications ne l'utiliseront pas, d'où un certain gaspillage. Le même problème peut être transposé pour la mémoire, pour la topologie d'interconnexion, pour le débit des liens, etc.

L'idéal serait de prendre l'ensemble des applications auxquelles la machine est destinée, de les analyser et d'en déduire les valeurs optimales des différents paramètres. La démarche logique pourrait alors se décomposer de la façon suivante :

- niveau applicatif : identification des besoins, effort de formalisation produisant des modèles, des langages, des algorithmes-types
- niveau architectural : choix d'une architecture adaptée, détermination qualitative des composantes, dimensionnement de ces diverses composantes en taille, en performance, en nombre
- niveau circuit : analyse logico-fonctionnelle, implémentation.

Cette démarche descendante est invalidée par l'existence d'une forte rétroaction de l'architecture et de la technologie sur les besoins. Des coûts trop élevés vont réduire le domaine d'application (généralement en faveur du calcul numérique). Au contraire, des coûts raisonnables, des modèles puissants, des structures pratiques, ainsi que la simple disponibilité de la machine vont favoriser des utilisations originales et enrichir le domaine d'applications, induisant de nouveaux besoins. Ces besoins souvent hétérogènes traduits au niveau conception vont renchérir la machine et nous ramener au premier cas...

D'autre part, la technologie impose des contraintes économiques ou parfois de simple faisabilité. Elles vont rendre plus rentables certaines solutions, soit parce qu'elles ont déjà été étudiées et validées, soit parce qu'elles sont plus simples. Les besoins exprimés au niveau architectural vont aussi influencer l'évolution de la technologie, mais sur une échelle de temps assez longue.

Enfin, les différents éléments architecturaux ne sont pas indépendants les uns des autres. Outre un problème de dimensionnement absolu de l'ensemble qui est lié aux applications, il existe un problème de dimensionnement relatif des éléments (mémoire, système de communication, processeur, E/S) ainsi que la nécessité d'assurer une cohérence d'ensemble des mécanismes mis en jeu.

Nous devons donc avoir une démarche dialectique dans notre prise en compte des besoins externes (applicatifs), internes (dimensionnement relatif) et de celle des contraintes externes (technologiques) et internes (cohérence d'ensemble)? Nous devons aussi intégrer les diverses rétroactions, qui ont lieu sur des échelles de temps différentes. Comment maîtriser ce processus ? Il n'est pas envisageable qu'un concepteur, aussi doué soit-il, puisse y arriver d'un seul coup, ne serait-ce qu'à cause de la dimension temporelle et de la présence d'acteurs sur lesquels il n'a aucune prise.

L'approche adoptée jusqu'à présent semble donc bien la seule possible. De nature itérative, elle consiste à se fixer une cible au niveau applicatif et quelques grandes lignes d'implémentation et à réaliser avec une machine aussi cohérente que possible. Ces choix s'enracinent dans ce que nous pourrions nommer "l'état de l'art", état que viennent enrichir les enseignements tirés du cycle de conception et du cycle d'utilisation de la machine. Ces correctifs peuvent alors donner lieu à un nouvel ensemble de choix initiaux et s'actualiser dans le projet suivant.

Les diverses architectures du Caltech illustrent à merveille cette démarche. En partant de l'idée de base de calculateur à mémoire distribuée (concept hérité de la constatation des limitations des calculateurs à mémoire partagée), de l'approche acteur (pour laquelle on ne disposait pas de machine très appropriée), et d'utilisateur à petits budgets, la première génération de Cosmic Cubes a été construite. L'expérimentation de modèles *réactifs* a mis en évidence la latence de communication comme facteur limitatif de la concurrence dans les langages d'acteurs. D'autres utilisateurs qui ont trouvé intéressant d'essayer des codes numériques ont constaté une certaine faiblesse des PE en ce domaine. La seconde génération de cubes a donc introduit un système de communication beaucoup plus performant, ainsi que des accélérateurs vectoriels optionnels. La contradiction entre un modèle de calcul à grain fin tirant vers un grand nombre de PE et le calcul numérique tirant plutôt vers l'augmentation de puissance des PE a conduit à scinder la descendance du Cosmic Cube en deux lignées :

- l'une a figé l'architecture et s'est contentée d'une évolution purement technologique vers des PE très puissants, tout en développant des systèmes d'exploitation distribués appropriés,
- l'autre (projet Mosaic), fidèle à l'approche acteur, a modifié l'architecture pour permettre d'accéder à une échelle supérieure de nombre de PE et a simplifié ceux-ci pour les intégrer entièrement dans un seul boîtier. Dans la même voie, le MDP du MIT [DAL87b] va encore plus loin en dédiant pratiquement le processeur à l'exécution de langages d'acteur et orientés objets.

Le réseau cellulaire.

Notre équipe a suivi une voie un peu différente. Nous sommes partis avec l'objectif de concevoir des accélérateurs matériels parallèles spécifiques pour certaines applications de CAO comme la simulation logique ou le placement. Toutefois, sous des dehors très ciblés, notre approche avait déjà une portée plus générale. L'idée de base qui

sous-tendait tous les développements effectués est qu'une architecture unifiée pouvait servir de squelette de départ pour la conception rapide d'accélérateurs pour toute sorte de problèmes. L'architecture en question, le réseau cellulaire, n'a pas beaucoup bougé depuis son introduction [BER85], contrairement à la façon dont nous l'utilisions.

Le réseau cellulaire est une architecture massivement parallèle par le nombre d'unités de traitement qu'elle envisage d'intégrer, au minimum de l'ordre du millier. Comme le qualificatif "cellulaire" le laisse supposer, la régularité est l'un des objectifs fondamentaux que nous avons visé. Nous voulions pouvoir construire une machine de taille et de puissance incrémentable, basée sur des éléments tous identiques.

En fait, les aspects qui ont prévalu lors de la conception de l'architecture cellulaire ont concerné essentiellement la facilité de réalisation, le coût de conception ainsi que celui de fabrication : les briques de base devaient être mono-circuit (la technologie VLSI nous le permettait), et s'assembler par simple juxtaposition . Les critères d'incrémentabilité matérielle sont des contraintes initialement posées comme prioritaires par rapport au caractère fonctionnel du réseau. L'essentiel du travail de l'équipe aura consisté à montrer qu'une machine conçue selon cette optique peut être effectivement fonctionnelle pour une grande variété de problèmes.

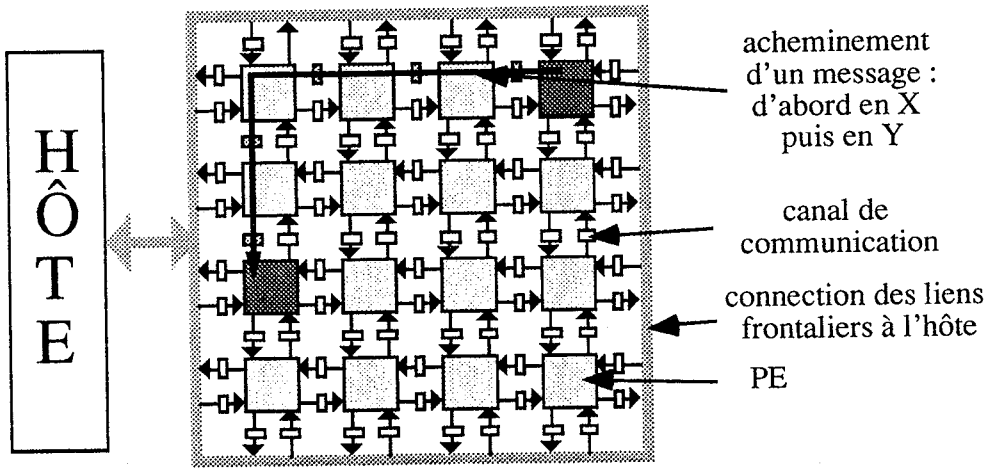


Fig. 4 — *Le réseau cellulaire.*

Pour le présenter brièvement, sans en justifier pour l'instant les choix, disons simplement que le réseau cellulaire est constitué d'un ensemble d'unités de traitement interconnectées en grille bidimensionnelle. Chaque unité de traitement fonctionne de façon autonome, à sa propre cadence (asynchronisme d'exécution). Une unité peut communiquer directement avec ses quatre voisines immédiates grâce à des canaux unidirectionnels dans chaque direction (Nord, Sud, Est et Ouest), mais elle peut aussi communiquer indirectement avec n'importe quelle unité plus distante ou avec l'ordinateur hôte. Cette non-localité de communication est réalisée par échange de messages. Le message circule d'unité en unité de sa source jusqu'à sa destination, en

choisissant son chemin suivant une stratégie simple : d'abord horizontalement, puis verticalement. Le passage de message introduit un asynchronisme de communication.

Afin d'optimiser les acheminements de messages non-locaux, la tâche de transfert des messages des canaux d'entrée vers les canaux de sortie appropriés a été rendue autonome ; elle est implémentée par un dispositif matériel nommé aiguilleur ou routeur.

L'ensemble constitué par l'unité de traitement, l'aiguilleur et les canaux d'entrée (tampons partagés) est nommé cellule ; c'est lui qui formera notre brique de base *(fig. 5)*. L'aiguilleur et l'unité de traitement fonctionnent indépendamment, s'échangent messages entrants et sortants par l'intermédiaire de tampons de communication, comme le font les aiguilleurs de cellules différentes entre eux.

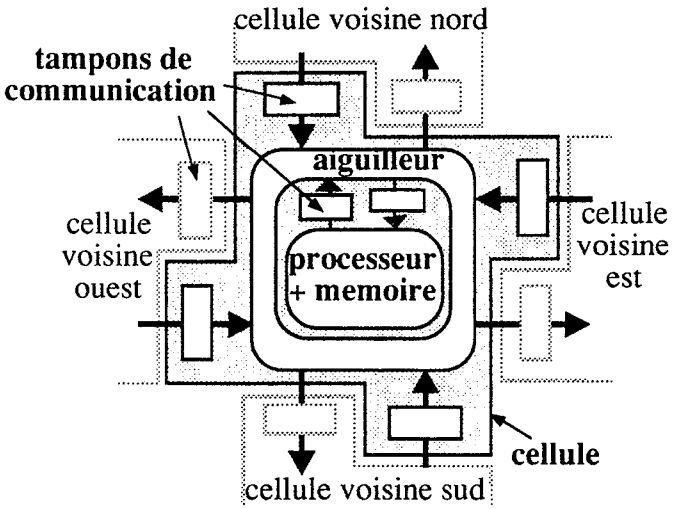


Fig. 5 — *Structure d'une cellule.*

Cette architecture générique permet de construire facilement des accélérateurs matériels pour divers problèmes : le système de communication est inchangé, seule la partie traitement est spécifique. Ainsi, nous avons étudié divers algorithmes :

- simulation logique [BER85,OBJ88] : chaque cellule contient une porte du réseau à simuler, et évalue cycliquement l'état de la sortie en fonction de l'état des entrées ;
- placement [COR88] : chaque cellule contient un élément à placer, ainsi que la position des éléments qui lui sont logiquement connecté ; itérativement, la cellule évalue la distance totale aux voisins et tente de la minimiser par échange d'éléments avec une autre cellule ; la technique de recuit simulé permet d'éviter de tomber dans des minimums locaux ;
- reconstruction d'image en tomographie [LAT89] : le réseau reconstitue une image d'un corps opaque à partir d'un ensemble de projections atténuées ;

- réseaux neuro-mimétiques [FAU90] : un neurone formel est associé à chaque cellule, sa sortie est évaluée en continu en fonction des entrées pondérées ; l'algorithme implémente une méthode d'apprentissage par rétro-propagation du gradient.

Les études sur la simulation logique et la reconstruction d'image ont donné lieu à la réalisation de circuits VLSI. Le coût financier ainsi que l'investissement humain de développement de ces circuits restent assez élevés. De plus, si la partie routage restait fonctionnellement la même, son implémentation reposait sur des paramètres dépendant de l'application (taille et structure des messages), ce qui a rendu le travail effectué à ce niveau impossible à "capitaliser". La première méthode proposée pour remédier à ces inconvénients a été la réalisation d'outils logiciels de génération automatique de réseaux cellulaires qui auraient pris en entrée une description algorithmique de haut niveau et auraient fourni en sortie les masques de fonderie correspondants.

Cette solution ne résolvait que le problème du coût d'étude mais pas celui du coût matériel : il faut que l'application soit particulièrement importante pour justifier la réalisation d'un réseau dédié en vraie grandeur dont le prix se chiffrerait vraissemblablement en millions de francs.

Considérant la taille des parties traitement obtenues par l'approche spécifique qui se situe entre 10000 et 20000 transistors, nous avons jugé plus rentable de changer notre fusil d'épaule et d'étudier une partie traitement programmable non dédiée. En effet, un processeur 8 bits comprend généralement entre 5000 et 10000 transistor et, associé à une petite mémoire RAM (de l'ordre de 256 octets), il peut tout à fait se substituer aux parties traitement spécifiques. Nous ne pouvons certes pas en attendre le même niveau de performance, mais nous apportons du même coup une solution au problème du coût d'étude et du coût matériel. Nous serons ainsi plus libres pour aborder la "vraie question", à savoir celle de la conception d'algorithmes parallèles adaptés aux limitations ne notre architecture :

- processeur peu puissant, notamment au niveau du calcul numérique
- mémoire très limitée
- communication restreinte aux cellules proches
- topologie d'interconnexion peu dense.

Nous rejoignons par la petite porte les préoccupations des projets de machines MIMD à grain fin comme Mosaic et Mega, mais nous nous en démarquons par l'acceptation de contraintes technologiques encore plus rigoureuses qui situent notre projet à une place inédite. Selon la terminologie anglo-saxonne, les machines MIMD appartiennent soit à la classe des *multiprocessors* (machines à mémoire partagée), soit à celle des *multicomputers* (chaque nœud est un ordinateur à part entière avec processeur, mémoire, E/S et système d'exploitation). Notre démarche nous amène à concevoir une

machine qui tient d'un point de vue matériel des *multicomputers* , mais nous refusons de munir chaque PE des ressources nécessaires à lui permettre une autonomie au niveau système. Si l'écriture d'un système minimum est difficile pour une machine à 16K de RAM, que dire d'une machine pourvue de moins d'un 1K !

En fait, nous nous rapprochons des machines SIMD où la programmation ne prend une signification qu'à une échelle *collective* . Le travail d'un PE reste une évaluation d'opérateur complexe, il ne prend jamais la forme d'un réel programme, tel qu'on a l'habitude de les concevoir. Seul le niveau macroscopique, c'est à dire l'ensemble de ces opérateurs et de leurs interactions, est porteur d'un sens algorithmique. Toutefois, le mode SIMD dispose d'un contrôle global d'exécution centralisé, dont nous ne disposons pas, que nous compensons le plein bénéfice d'une hétérogénéité et d'un asynchronisme des opérateurs.

La taille de la mémoire de programme et de donnée, de l'ordre de 256 octets, place notre machine à une échelle de petitesse inégalée : nous avons au moins un facteur de taille 64 avec les autres architectures, qu'elles soient SIMD ou MIMD. La puissance du processeur (8 bits) est inférieure à celle des processeurs des autres machines MIMD et, si elle est supérieure à celle des processeurs 1 bit de la Connection Machine, elle ne dispose pas d'unité spécialisée pour le calcul en virgule flottante. Combinée à une topologie d'interconnexion facilement extensible, cette architecture comporte potentiellement un nombre plus important de PE que les autres architectures.

Mais les échelles de grandeur de chacune des caractéristiques qui résultent de la volonté d'intégrer au mieux les contraintes technologiques peuvent t-elles s'accommoder des autres contraintes qui viennent de utilisateurs ? Nous avons tout lieu de le penser puisque les techniques de programmation introduites et validées lors des études de machines dédiées peuvent être transposées sur une machine à vocation généraliste. Mais si un réseau cellulaire dédié peut être qualifié de *Single Program Multiple Data* (SPMD), le réseau général appartient clairement à la classe des architectures *Multiple Program Multiple Data* (MPMD). Une simple transposition ne peut suffire à elle seule à garantir que le réseau programmable pourra être pleinement exploité.

Des données relatives à la programmation sont néanmoins nécessaires pour trancher les divers choix concernant le processeur (jeu d'instructions, nombre de registres, largeur du chemin de données, etc.), la mémoire (taille), le système de communication (taille des messages, contrainte de débit). Inversement la disponibilité, la non disponibilité ou l'insuffisance d'une fonctionnalité va diriger le programmeur vers tel ou tel algorithme, telle ou telle modélisation. Unités de traitement et communication étant aussi liées par diverses contraintes, nous sommes placés dans la situation de concevoir *concurremment* trois éléments *interdépendants*.

La méthode que nous avons adoptée est un peu de même nature que celle énoncée pour la conception au niveau global : il s'agit d'une sorte de jeu de ping-pong entre les

trois pôles que sont PE, réseau et programmation, jeu qui tente d'établir le meilleur compromis entre coût, performance, et fonctionnalité.

L'architecture globale, en grille bidimensionnelle et avec passage de messages, a été conservée ainsi que, dans un premier temps, la structure de tampons inter-cellulaires.

Afin d'amorcer l'analyse, une solution qui n'est pas pire qu'une autre consiste à choisir arbitrairement une palette d'exemples d'école supposés représentatifs, par leurs caractéristiques et leur comportement, de l'ensemble des programmes susceptibles d'être un jour écrits pour notre réseau cellulaire. Dans la pratique, un certain nombre d'exemples utilisés proviennent d'autres études sur le parallélisme massif MIMD [DAL86, ATH87a], d'autres ont été choisis un peu au hasard dans la littérature. Ce procédé nous imposera une certaine prudence dans nos conclusions globales car on pourra toujours nous reprocher un certain parti-pris ou encore de favoriser les applications "qui se programment bien sur le réseau cellulaire" ; mais nous verrons que l'étude individuelle des programmes peut être aussi très instructive. Cet ensemble d'exemples a été constamment enrichi tout au long de l'étude par reformulation plus adéquate des mêmes problèmes ou introduction de nouvelles applications. Il contient des programmes de tri, des réseaux systoliques, des programmes LUSTRE, des automates cellulaire, un problème de graphes, des réseaux neuro-mimétiques, etc.

L'histoire des microprocesseurs ayant montré qu'il est difficile de descendre en deçà d'un processeur 8 bits sans descendre d'un cran au niveau applicatif, ce qui rendrait difficile l'implémentation de schémas de communication complexes et limiterait le champ d'utilisation de notre architecture. Nous sommes donc partis d'une architecture de PE de type 6800 et nous l'avons retaillée grossièrement pour l'adapter à nos besoins. L'étude des applications programmées pour ce PE nous a permis de raffiner le jeu d'instruction de manière significative.

En matière de 8 bits, la technologie était arrivée à un degré de maturité propre à nous permettre de considérer les performances comme suffisamment stables pour nous fournir les données de dimensionnement des liens de communication. Nous pouvions intégrer des données comme le nombre de PE par chip, la distribution spatiale et temporelle des messages, la sensibilité des applications pour déterminer quels sont les modes de commutation et les techniques de réalisation des liens qui sont praticables.

Pour finir il ne nous restait plus qu'à essayer de tirer les conclusions de notre expérience de programmation, afin de voir dans quelle mesure nous sommes arrivés à prendre en compte les besoins des utilisateurs sans oublier les contraintes technologiques.

Chapitre II : Le réseau de communication

Le réseau de communication est la clef de voûte de toute machine parallèle, son rôle est de permettre aux différentes unités de traitement de se synchroniser, d'échanger des données entre elles, avec des unités de stockage ou avec des unités d'entrées/sorties. Le rôle critique du réseau provient du fait que ce ne sont jamais des tâches totalement indépendantes qui sont assignées aux processeurs, donc que ceux-ci doivent communiquer pour coopérer. Le système qui va permettre cette communication ne doit pas être trop lent car il serait un frein à la génération du parallélisme, il ne doit pas non plus être trop cher car il rendrait impossible la réalisation d'une machine dotée d'un grand nombre de processeurs. En fait, quel que soit l'angle sous lequel nous envisageons la question, nous revenons toujours à l'idée que de l'adéquation rigoureuse du système de communication au système de calcul (et plus spécifiquement encore à l'application) va dépendre le niveau de puissance qu'il sera techniquement possible d'atteindre. Ceci explique que les divers choix architecturaux concernant le réseau de communication revêtent une telle importance et qu'ils soient tant sujets à controverses.

Afin de situer le problème, nous commencerons ce chapitre par une rapide revue des différents types d'architectures de réseau. Nous verrons que, confrontée aux contraintes propres au parallélisme massif, l'apparente profusion de solutions se réduit comme une peau de chagrin, nous confortant dans les orientations prises par notre réseau cellulaire à sa naissance. La partie purement expérimentale de l'étude du système de communication portera donc plutôt sur la manière de l'implémenter, et en particulier sur le *mode de commutation* à adopter. Nous étudierons le jeu de programmes de test sur le plan des besoins de communication, des comportements. Nous cernerons les diverses options qui s'offrent à nous lors de la conception pour en extraire un jeu de solutions cohérentes qui seront comparées au niveau de leurs performances par des simulations d'applications parallèles et au niveau de leur intégration par une estimation de coût. Notre objectif final sera bien sûr de déduire de cette étude les "meilleures" solutions (nous verrons ce que cet adjectif peut recouvrir), mais aussi et surtout de donner la mesure de leur l'influence sur le comportement de l'ensemble de la machine.

A. Généralités sur les réseaux.

Nous aborderons successivement trois aspects fondamentaux des réseaux de communication : la topologie, le routage et le mode de commutation.

1. Topologies.

Les systèmes de communication se divisent grossièrement en deux classes : ceux construits autour de bus, dont nous allons voir qu'ils ne sont pas du tout adaptés au parallélisme massif, et ceux organisés en graphes de liens de communication.

1.1. Structures à bus.

Ce type de structures omniprésent dans toutes les machines monoprocesseurs : bus interne du processeur, bus entre processeur et mémoire, bus de raccordement de périphériques, réseaux locaux, etc. Un bus permet à tous les dispositifs qui lui sont raccordés de communiquer deux à deux (éventuellement de 1 vers plusieurs). C'est tout naturellement que les architectes ont pensé à l'utiliser dès qu'il a été question de connecter plusieurs unités de calcul pour réaliser les premières machines parallèles : un processeur n'est après tout qu'un dispositif comme un autre. De nombreuses organisations sont alors possibles pour réaliser des multiprocesseurs, notamment sur la manière de répartir la mémoire. Celle-ci peut être commune et centralisée, mais on préfère souvent la répartir *(fig. 1a)*, la rendre privée *(fig. 1b)* ou encore commune et distribuée avec accès non uniforme *(fig. 1c)*. Lorsque le nombre de processeurs augmente, le volume d'informations à transférer évolue de façon telle que seules les deux dernières organisations sont réalistes.

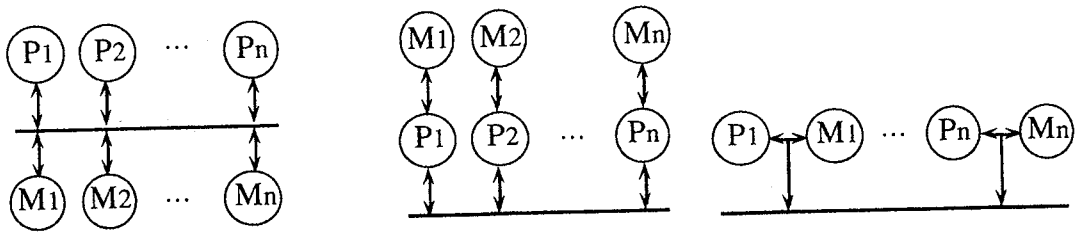


Fig. 1.— *a) mémoire commune* *b) mémoire privée* *c) mémoire locale.*

Ces deux organisations peuvent être mises en œuvre de manière à ce que le bus ne serve plus qu'à la synchronisation et au partage de données, l'accès à la mémoire de programme et aux variables locales étant strictement réparti. Malheureusement, même dans ce cas favorable, un bus ne permet qu'une communication à un instant donné et, de ce fait, se comporte comme une ressource unique, soumise à exclusion mutuelle et arbitrage. Il devient rapidement un goulot d'étranglement dès que le nombre de processeurs augmente : le calcul est parallèle et la communication séquentielle.

Hormis son statut de ressource unique, le bus présente par son allongement d'autres inconvénients majeurs [LIT90] :

- électriques : la géométrie physique du bus devient déterminante afin de ne pas provoquer l'apparition de phénomènes parasites gênants (capacité, inductance)
- architecturaux : l'arbitrage entre plusieurs scripteurs concurrents devient de plus en plus coûteux ; comme pour toute ressource partagée, il faut définir une discipline de service équitable.

Par contre, un bus permet de réaliser facilement les opérations de diffusion (*broadcast*), qui sont très utiles pour bon nombre d'algorithmes [YAN90]. Les performances d'un bus peuvent être améliorées par diverses techniques [LIT90] : l'usage de caches au niveau de chaque processeur, l'élargissement du bus, par la mise en parallèle de plusieurs bus, l'usage de liens haut-débits pour la réalisation physique du bus, etc. Toutefois l'arbitrage et la gestion de la cohérence de tels ensembles deviennent de véritables casse-têtes. Ces techniques ne prétendent pas résoudre le problème de base (le bus reste une ressource critique), mais simplement rendre possible la réalisation de multiprocesseurs à base de composants modernes (Motorola MC88000, Sparc...) en nombre restreint (de 4 à 8)[1].

Il existe quand même une variation sur le thème du bus qui peut présenter un intérêt pour le parallélisme massif : il s'agit des hiérarchies de bus, et d'une façon plus générale des réseaux de bus.

---

1 Rappelons qu'à quelques exceptions près, ces architectures sont à la base des seules machines parallèles qui aient fait l'objet de commercialisation...

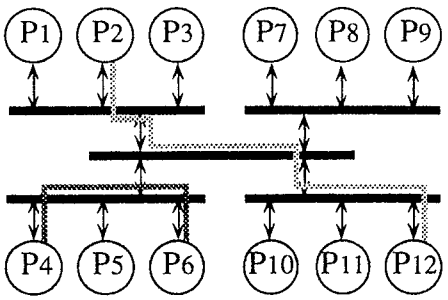


Fig. 2. — *Arbre de bus à deux niveaux.*

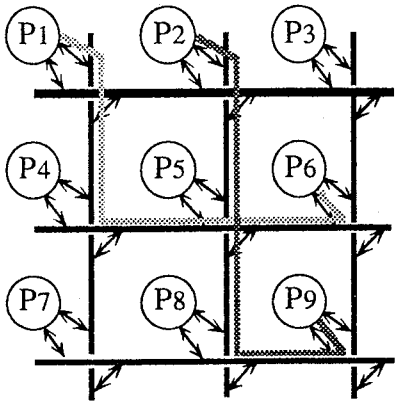


Fig. 3. — *Grille de bus.*

Comme pour les réseaux de liens que nous verrons plus loin, toutes sortes de topologies sont possibles chacune possédant ses avantages et ses inconvénients. Les structures en arbre *(fig. 2)* sont facilement incrémentables avec des bus de longueur fixe, mais le bus racine est une ressource critique, la longueur du chemin entre deux processeurs est très variable et peut devenir importante. Les caractéristiques des structures en grilles *(fig. 3)* sont exactement inverses.

L'argument le plus important contre l'usage des structures de communication à bus pour la réalisation de machines massivement parallèles est la sous-utilisation manifeste de la connectique. Supposons un bus servant de système de communication à n processeurs $P_1 \ldots P_n$. Si P_1 communique avec P_n, toute la longueur du bus est utile, alors que si P_1 communique avec P_2, seule une fraction $1/(n-1)$ du bus est utile. Le bus défavorise donc les communications locales. La solution pour rééquilibrer le système consiste multiplier les bus tout en les spécialisant (tous les processeurs ne sont pas connectés à tous les bus). En poussant cette idée au maximum, elle mène à des bus à un seul processeur, interconnectés entre eux. Une telle structure est fonctionnellement identique aux structures en réseaux de liens bi-points à commutation de circuit virtuel *(cf. 2.1.)*.

1.2. Structures en réseaux de liens

Que l'on conserve une solution de bus ou non, l'obtention d'un haut niveau de parallélisme réel passe toujours par la dé-séquentialisation de la communication. L'interconnexion des processeurs suivant un graphe de liens de communications bi-points, communément appelé réseau, permet de réaliser toute une gamme de niveaux de parallélisme de communication.

Toutes les machines à fort degré de parallélisme ont adopté une architecture en réseau, mais elles se différencient sur le choix de la topologie de ce réseau. Comme nous allons le voir, ce problème est tout à fait crucial et n'est pas simple du tout à trancher. La topologie du réseau va déterminer quelles paires de processeurs vont pouvoir dialoguer directement. Donc deux processeurs non reliés physiquement vont

devoir passer par des processeurs-relais s'ils ont besoin de dialoguer, une communication algorithmique sera décomposée en plusieurs communications physiques. Le temps de communication et la charge du réseau en seront d'autant augmentés. Du point de vue des performances, nous aurons intérêt à avoir une topologie physique qui soit un *sur-graphe* des graphes des applications. Comme ceux-ci sont très peu homogènes, cela nous mène à favoriser des graphes fortement connectés. Par contre, les critères de coût de réalisation feront préférer les topologies comportant des liens bien choisis et en nombre minimum. Outre sur la topologie, les architectures diffèrent quant au mode de commutation (commutation de circuits virtuels, *store-and-forward*, *wormhole*, etc).

1.2.1. Interconnexion complète

En vertu de l'adage "qui peut le plus peut le moins", l'interconnexion par un graphe complet a longtemps représenté un modèle idéal, dont on ne s'écartait que faute de mieux. Les travaux de Dally [DAL86] montrent toutefois que dans les contraintes réelles (densité fixe de connexions, délai de propagation variant avec la longueur des connexions) les graphes à très fortes connectivités ne sont pas les plus performants *(cf. A.1.2.2)*. Sa mise en œuvre pour de gros réseaux est problématique, car le nombre de liens croit en fonction du carré du nombre de processeurs. Au delà de quelques unités, cette solution n'est pas envisageable.

Toutefois, comme les processeurs ne peuvent en général pas effectuer plusieurs communications à la fois, les N(N-1) liens ne pourront être alimentés en permanence ; seuls N liens peuvent être utilisés simultanément (sauf si les communications sont des diffusions). Ces N liens ne sont jamais les mêmes d'une application à l'autre et d'un instant à l'autre dans une application. Une idée intéressante consiste à choisir ces N liens parmi les N(N-1) de façon dynamique. Ceci peut être réalisé par l'intermédiaire d'un commutateur *crossbar*, d'un réseau Ω, etc. (voir [AUG85]). On réalise ainsi physiquement et dynamiquement le sous-graphe de K^N qui nous intéresse.

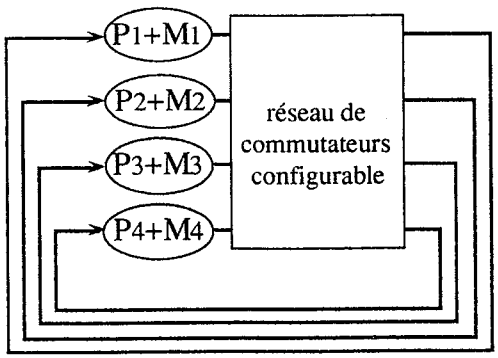


Fig. 4. — *Réseau indirect interprocesseur.*

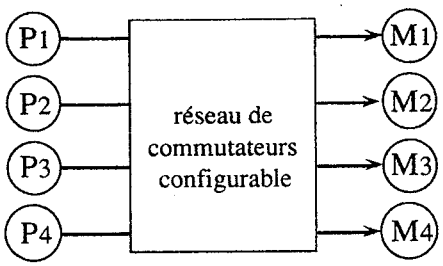


Fig. 5. — *Réseau indirect de distribution de mémoire.*

Ces réseaux sont en général des réseaux indirects c'est à dire avec des processeurs connectés à la périphérie du graphe et non à chaque nœud du graphe. Ils s'insèrent dans l'architecture à la manière d'un bus commun *(fig. 4 et fig. 5)*.

Cette approche a l'avantage de conserver la généralité d'une interconnexion complète à un coût nettement moindre, quoiqu'au moins proportionnel à $log_2 N$, ce qui la rend impraticable pour un véritable parallélisme massif. Les points faibles de cette architecture sont la taille du commutateur ainsi que la complexité des algorithmes à mettre en œuvre pour sa configuration (non triviaux notamment dans le cas des réseaux réarrangeables).

Cette technique de communication intéresse un parallélisme à grain moyen où les paquets transmis sont assez longs et/ou le graphe d'interconnexion est stable ; dans le cas contraire, la reconfiguration devient une opération critique. La machine Supernode [MUN89] *(Chap. I, § 4.2.)* est un exemple de l'implémentation à base de Transputers et de commutateur *crossbar*.

Dans une structure multi-bancs mémoire, ces réseaux indirects permettent de limiter les conflits d'accès aux variables partagées en recombinant les requêtes concernant un même mot mémoire au niveau du réseau, et donc de répondre à ces requêtes simultanément, au lieu de sérialiser ces réponses.

1.2.2. Structures en grilles.

L'autre méthode pour limiter la connectivité sans dégrader les performances consiste à miser sur une certaine localité de référence. La localité de référence est la propriété pour les processus d'un algorithme de ne communiquer qu'avec un petit sous-ensemble stable de processus ; bon nombre d'algorithmes la possèdent. Dans le cas de machines dédiées à une application, il est possible de figer le schéma de communication de l'algorithme dans la topologie physique (réseaux systoliques, automates cellulaires), mais pour une machine généraliste c'est impossible. Par contre, nous pouvons émettre l'hypothèse selon laquelle le graphe de communication des algorithmes peut être *placé* sur une topologie physique de telle façon que les références de processus à processus correspondent à des communications de voisin à voisin ou de faible distance. Ce placement passe parfois par une modification préalable de l'algorithme, pour le rapprocher de la topologie physique. Il est possible de l'automatiser par le biais de l'exécution d'heuristiques (recuit simulé, etc).

Les avis divergent quant à la topologie susceptible de permettre les meilleurs placements au moindre coût. Il est certain que la topologie physique doit être suffisamment régulière pour pouvoir être réalisée et exploitée. Les structures en grilles sont les plus naturelles, surtout lorsqu'on a affaire à des technologies planes (cartes, silicium). Nous verrons plus loin des structures hypercubiques et arborescentes.

Dans une structure en grille, on connecte les processeurs selon un principe de proximité spatiale dans un espace discret de dimension fixe. La fonction de voisinage

utilisée est en général orthogonale (nord, sud, est, ouest pour une grille bidimensionnelle), mais d'autres géométries peuvent présenter des caractéristiques intéressantes, notamment le voisinage hexagonal [CHE90]. Pour ce type de voisinage, des algorithmes de placements d'arbres binaires ont été développés qui se sont montrés très supérieurs à ceux utilisant une géométrie orthogonale [GOR87]. Ces comparaisons supposent des communications limitées aux voisins immédiats, mais si on relâche un peu cette contrainte en introduisant une portée d'au moins deux cellules, il devient facile de simuler une géométrie avec l'autre.

L'espace étant infini et le nombre de processeurs fini, il faut poser des conditions de voisinages au bord. Nous pouvons soit laisser des liens en l'air (pour éventuellement y connecter des périphériques ou des frontaux), soit reboucler chaque coté sur le côté opposé (structure torique). Là aussi, une portée d'au moins 2 permet de simuler une structure torique avec une structure non torique (voir annexe 1, 4.).

En dimension 1, la grille se réduit à une simple ligne de processeur, ou à un anneau si la structure est torique. Aussi triviale que puisse paraître une grille de dimension 1, c'est une structure qui a donné lieu à un certain nombre de réalisations ainsi qu'au développement de algorithmes spécifiques. Toutefois, elle doit être réservée à des architectures où la communication possède une forte localité, car la distance moyenne entre deux processeurs quelconques est de l'ordre du nombre de processeurs. Concrètement, il s'agira en général de machines systoliques ou SIMD[1].

En dimension 2, on a une structure qui est particulièrement bien adaptée au raisonnement humain. En effet, de nombreux problèmes et algorithmes sont exprimés dans un formalisme plan, ne serait-ce que parce qu'ils sont d'abord couchés sur une feuille de papier. De fait, la littérature regorge d'algorithmes systoliques bidimensionnels dont le graphe d'interconnections peut être immergés dans une grille plane en gardant une bonne localité.

L'immersion de structures arborescentes dans une grille a aussi été étudiée, avec entre autre le classique *H-tree*. Bien entendu, cette immersion ne peut pas être parfaite : pour un diamètre identique, le nombre de processeurs croît beaucoup plus vite dans un graphe en arbre (fonction exponentielle du demi-diamètre) que dans une grille (carré du diamètre). Aux liens logiques correspondent donc plusieurs liens physiques, en nombre augmentant avec la profondeur de l'arbre. Nous serons donc limités par la portée d'adressage. Par un système de relais, il est possible d'augmenter arbitrairement le nombre de processeurs potentiellement utilisable. Dans le *H-tree*, avec un adressage NEWS, la moitié des nœuds sont utilisés en relais. Toutefois, il existe des algorithmes

---

1 Yang et Lee ont montré [YAN86] que les tableaux systoliques 2D à un seul front d'activité, sans retour en arrière, peuvent être mappé sur une grille 1D.

d'immersion moins mauvais, qui utilisent les feuilles de l'arbre comme relais (car celles-ci, n'ayant pas de fils, sont censées moins travailler) [GOR87].

Enfin notons que les grilles bidimensionnelles sont très facilement extensibles par simple juxtaposition. Le degré de chaque nœud est constant, ce qui est précieux pour la réalisation : le nœud est une brique de base inchangée quelle que soit la taille du réseau. Cette caractéristique d'extensibilité disparaît dès la dimension 3, car les technologies d'interconnexion VLSI sont bidimensionnelles[1].

Un n-cube k-aire est une grille de dimension n et de côté k, comportant donc k^n processeurs. Dans une telle structure, la distance moyenne entre deux processeurs quelconques est de l'ordre de $\sqrt[n]{k}$, et on pourrait penser que la dimension la plus élevée donne les latences de communication les plus faibles. Or, cela n'est vrai qu'en nombre de nœuds traversés. Dally [DAL86] a montré en introduisant les contraintes de la technologie VLSI que le degré optimal est de dimension faible : il va de 2 pour 256 nœuds à 5 pour 1M nœuds. Aussi surprenant que ce résultat puisse paraître au premier abord, il s'explique tout à fait bien : quelle que soit sa dimension, le n-cube k-aire doit être "étalé" en dimension 2, avec comme conséquence l'allongement des liens physiques et aussi des temps de propagation à travers ceux-ci. La fonction qui donne la dimension optimale doit être utilisée avec précaution car elle est dépendante du modèle électrique des liens et de modèle statistique de communication. Ce dernier a été supposé aléatoire : de façon répétée, chaque nœud sélectionne au hasard un nœud et lui envoie un message. C'est un modèle éminemment défavorable pour les réseaux de dimensions faibles puisqu'il ne tient pas compte d'une éventuelle localité de référence (effective dans de nombreux problèmes). Plus récemment, D.C. Fisher a étudié l'incidence des contraintes du monde réel sur les limites théoriques de performances des algorithmes parallèles [FIS88]. La structure en grille a été utilisée dans les machines CHIP, DAP, Illiac IV, MPP, etc. La structure de n-cube k-aire est utilisée entre autres dans l'Ametek 2010 [SEI88] et Mosaic [SEI90].

1.2.3. Hypercubes.

Les hypercubes sont aussi des n-cubes k-aires, mais c'est le facteur k qui est considéré comme constant (en général k=2), et c'est par l'augmentation de la dimension que se fait l'extension du réseau. Ce n'est plus la capacité à être étendu par juxtaposition qui disparaît, mais la simple capacité à être étendu. En effet, le degré de chaque nœud change lorsque change la taille du réseau, ce qui oblige à modifier au niveau matériel tous les nœuds du réseau. Ce problème peut être résolu à la conception en fixant une dimension maximale. Si l'hypercube effectivement construit est de dimension inférieure, une partie du système de communication sera inutilisée. Il est possible aussi de concevoir le mécanisme de routage de chaque nœud de façon modulaire, c'est à dire avec un module standard par dimension [FLA87], mais cette

---

[1] La machine MEGA [GER89] introduit un système d'interconnexion 3D.

stratégie ne peut être envisagée que comme une aide à la réalisation initiale et non comme un moyen d'étendre des unités installées.

Comme nous l'avons vu plus haut, le critère de performance lié au faible diamètre n'est pas défendable dans le contexte d'une architecture massivement parallèle. L'intérêt des réseaux en hypercubes réside bien plus dans leurs propriétés topologiques [SAA88]. Parmi ces propriétés figure celle d'être un excellent substrat pour immerger les autres structures courantes (anneaux, grilles de dimension quelconque, arbres), propriété qui a comme corollaire une capacité à exécuter de tous les algorithmes conçus pour ces structures en plus des algorithmes spécifiques. Cela fait de l'hypercube une topologie quasi universelle.

Nombreux ont été les projets bâtis autour d'une structure en hypercube. Nous avons vu au chapitre I le Cosmic Cube du Caltech et ses descendants de la série iPSC d'Intel et NCUBE, la Connection Machine. Citons les hypercubes de transputers (série T de Floating Point Systems Corp.[FPS86]), le JPL-Mark III [PET85].

1.2.4. Structures arborescentes.

Les structures arborescentes semblent associer les avantages des grilles (extensibilité infinie, connectivité faible) à ceux de l'hypercube (faible diamètre). Ce sont toutefois des structures moins générales : si on peut immerger un arbre dans un hypercube ou une grille, le contraire est nettement plus problématique.

D'un point de vue algorithmique, elles sont parfaites pour tous les problèmes auxquels on peut donner une solution récursive (tri, recherche, etc, cf. [CAR88]) et d'une manière générale aux problèmes d'intelligence artificielle. Lorsqu'on s'écarte des algorithmes dont la topologie logique correspond à peu près à un arbre, on court le risque de créer un goulot d'étranglement à la racine. Ce défaut peut être atténué par l'usage de liens transversaux à l'arbre, ce qui donne des structures hybrides.

Les structures en arbre sont plus confidentielles que les hypercubes ; citons néanmoins les machines NON-VON, DADO, le Cellular Computer, et la Tree-Machine du Caltech [BRO80].

1.2.5. Autres topologies.

Pour associer les qualités de deux structures différentes A et B, nous pouvons procéder de deux façons :

- greffer des instances de B en lieu et place des nœuds de A, nous obtenons alors une "chimère".
- superposer les deux structures en appariant chaque nœud de A avec un nœud de B, nous obtenons un "hybride".

Dans la Connection Machine CM-1, on a une structure de base en hypercube de degré 12, avec à chaque nœud un module de 4x4 processeurs. De plus, il existe dans la CM-1 une connexion directe en grille bidimensionnelle n'utilisant pas le routage de l'hypercube. Cette connexion en grille est jusqu'à 6 fois plus rapide que le réseau hypercubique, mais l'adressage est implicite, c'est à dire que les processeurs ne peuvent choisir leur correspondant indépendamment les uns des autres. Cette connexion en grille sera supprimée dans la CM-2 au profit d'une liaison en grille de dimension n arbitraire bâtie sur l'hypercube [TUC88].

Les structures pyramidales sont des structures hybrides par excellence : à chaque niveau d'un arbre de degré 4 (QUADTREE), on construit une grille bidimensionnelle de communication transversale. C'est une architecture bien adaptée au traitement d'image : les pixels prennent place à la base de la pyramide, chaque couche de processeur effectue un traitement de combinaison des pixels qui sont sous sa coupe, et interagit avec ses voisins horizontaux, pour ensuite faire remonter une information plus élaborée à son père.

Il est évident que cette architecture, lorsqu'on la considère comme un modèle de traitement, est très spécialisée dans une certaine forme d'algorithme. En effet nous retrouvons le traditionnel goulot au niveau de la racine, qui induit une réduction du volume d'information à chaque niveau de la pyramide.

Plus intéressante est l'approche qui consiste à affecter aux étages supérieurs un rôle de supervision système [MEH88]. Nous obtenons alors une machine à traitement réparti disposant d'un mécanisme de contrôle allant du centralisé pur (tâches affectées à la racine) au réparti (tâches de supervision affectées au niveau coiffant le dernier niveau de traitement). Le système ainsi obtenu laisse entrevoir des solutions dynamiques élégantes aux problèmes traditionnels d'affectation de tâches, d'équilibrage de charge, de détection de terminaison.

De nombreuses autres topologies ont été étudiées. Citons pour mémoire :

- le *Cube Connected Cycles* [PRE81] : cycles interconnectés en hypercubes, où les nœuds sont de connectivité constante égale à 3 ; cette structure remédie à l'un des inconvénients majeurs des hypercubes, elle est utilisée dans la machine BVM [WAG83]
- le *modified mesh* [CAR88] : pyramide tronquée
- les *express-cubes* [DAL91] : n-cube k-aires avec liens supplémentaires non locaux
- les hypercubes augmentés : utilisation astucieuse des liens en réserve destinés à l'augmentation de la dimension du cube pour l'enrichissement de la densité de connexion.

Le lecteur intéressé trouvera une étude comparative des propriétés topologiques des topologies dans [GER89].

L'inventaire des diverses topologies et modes de commutation montre que peu nombreux sont les systèmes de communication compatibles avec le domaine de la programmation parallèle que nous avons choisies pour cible. De façon heureuse (car cela simplifie la décision), le parallélisme à grain fin impose des contraintes spécifiques vraiment draconiennes. Ces contraintes que nous avons mises en regard de la description de chaque topologie, sont toutes des déclinaisons du critère d'*incrémentabilité*.

L'incrémentabilité matérielle impose :

1- une brique de base invariante quelle que soit la taille du réseau car d'une part il faut la réaliser en grande série, d'autre part la taille du réseau n'est pas connue à priori.
2- une faculté à être assemblée et maintenue dans la géométrie qui est la notre (un hypercube de dimension n est simple à construire dans un espace à n dimensions, nettement moins lorsqu'un faut le ramener à un espace à trois dimensions)
3- une indépendance des mécanismes internes vis-à-vis de la taille du réseau

L'incrémentabilité logicielle impose :

4- une géométrie logique suffisamment régulière pour pouvoir être manipulée sans que l'affectation des tâches sur les processeurs soit un problème plus complexe que celui que nous sommes est supposé résoudre
5- les programmes des PE doivent être indépendants de la taille du réseau

Il apparaît que seules les topologies planes "cristallines" (les grilles) répondent à tous ces critères. Les structures en grilles de bus doivent être écartées car elles sont rapidement limitées en taille. De plus le critère 3 rend le synchronisme impraticable, car il est impossible de distribuer une horloge commune sur un réseau de grande taille.

La topologie d'interconnexion choisie est donc une grille bidimensionnelle, afin de s'adapter au mieux à la technologie VLSI. La géométrie est orthogonale, car c'est la géométrie la plus naturelle et la plus simple à manipuler. Une géométrie hexagonale peut très facilement être mappée sur une géométrie orthogonale si la communication n'est pas limitée aux quatre voisins immédiats.

Nous avons opté pour une structure <u>non-torique</u> afin de permettre une connexion du dispositif hôte sans avoir à particulariser certains nœuds (critère de régularité). Les structures en tore unidirectionnel parfois utilisées [DAL86] comportent deux fois moins de liens et des routeurs plus simples, mais elles présentent un défaut rédhibitoire : si, comme c'est le cas pour nous, le contrôle de flux de communication est souvent réalisé par des messages de requête en sens inverse, le couple requête-réponse effectue le tour

complet du tore, interdisant ainsi toute exploitation de localité de référence. Outre la perte d'efficacité induite, il faut noter que la portée d'adressage devient nécessairement dépendante de la taille du réseau. Les structures toroïdales bidirectionnelles, qui pallient à ce défaut, peuvent être aisément simulées par une grille simple, grâce à un placement approprié.

Reste à savoir si ces choix, résultats de contraintes technologiques et architecturales, donneront un système de communication suffisamment performant et général.

2. Stratégies de routage.

Dès lors que le réseau ne peut plus être assimilé à un graphe complet, deux nœuds choisis de façon quelconque ne sont généralement pas connectés directement et doivent faire appel à des nœuds-relais pour communiquer. La suite des relais empruntés pour un message donné constitue son *chemin*. Il existe généralement plusieurs chemins directs de coûts égaux entre deux nœuds donnés, sans parler des chemins indirects (où l'on admet de s'éloigner momentanément de la destination). Par exemple, avec un n-cube k-aire, il faut $k * n$ déplacements élémentaires pour aller d'un coin au coin opposé, suite dont toutes les permutations donnent des chemins corrects, soit (kn)! possibilités.

Le coût d'un chemin est fonction de sa longueur mais, en pratique, il faut lui intégrer le coût lié aux collisions entre messages. L'ensemble des règles mises en œuvre pour choisir un chemin pour un message donné, en essayant d'en minimiser le coût, constitue la *stratégie de routage*. Elle doit certes minimiser le coût *moyen* des chemins, mais aussi prévenir les interblocages et les famines, ce qui introduit des contraintes supplémentaires.

Une stratégie de routage simple est généralement *déterministe*, c'est à dire que le chemin entre deux nœuds est fixé préalablement à l'exécution. Dans les grilles orthogonales, on utilise souvent le *routage X-Y* [1] qui est une stratégie déterministe où chaque message est d'abord acheminé horizontalement jusqu'à la colonne de destination, puis verticalement. Cette stratégie peut être généralisée à un nombre quelconque de dimensions (chaque dimension est routée successivement, selon un ordre préétabli). Ses caractéristiques sont intéressantes : absence d'interblocage (voir § B.2.2.2), une tendance à utiliser uniformément toute la surface de la grille, sans introduire de *hot-spot* (région à forte densité de message) au centre.

Une stratégie aussi rudimentaire ne permet pas de garantir un équilibre de charge entre liens, car elle ne prend pas en compte des particularités de l'application. Il existe des stratégies déterministes qui étudient statiquement la connectique de l'application et

---

[1] En anglais *first-X then-Y* ou *find-row/find-column*.

en déduisent les chemins en minimisant le flux affecté à chaque lien, avec un algorithme de *max-flow* par exemple [BAD89]. Cette approche suppose un comportement de l'application homogène dans le temps, car une application changeant en cours de route ses caractéristiques invaliderait une telle approche statique. Une autre solution consiste à choisir dynamiquement le chemin pour un message donné en fonction de la charge instantanée (ou cumulée) de chaque lien. Les stratégies qui en découlent sont non-déterministes et sont dites *adaptatives*. Elles sont difficiles à mettre en œuvre, car soit l'algorithme d'adaptation est global et donc inadéquat pour un parallélisme massif, soit il est local et il ne peut prendre des décisions sûres par manque d'informations.Certaines choisissent entre les liens rapprochant de la destination, d'autres, comme la stratégie de *routage forcé* [GER90] utilisé dans la machine MEGA, permettent de choisir des chemins indirects lorsque les chemins directs sont indisponibles.

Les stratégies adaptatives sont très souvent présentées comme de bonnes candidates pour les machines à tolérance de panne (systèmes à haut degré de fiabilité, systèmes embarqués). En effet, elles peuvent contourner une cellule, un lien ou un routeur défectueux. Cette tolérance de panne peut être dynamique, il faut alors écrire des algorithmes qui soient eux-même à tolérance de panne (réseaux de neurones par exemple), elle peut être statique, le réseau est alors recomposé sur un réseau physique comportant des éléments hors-service [COD91].

Concernant le parallélisme massif, disons que les stratégies les plus simples semblent avec l'expérience largement suffisantes, car les messages sont de tailles faibles et donc les collisions sans gravité excessive : un message bloqué ne le reste que jusqu'à ce que le message gênant soit transmis, ce qui est généralement rapide. C'est pourquoi nous avons opté depuis l'origine du réseau cellulaire pour un routage X-Y. Aucune technique de tolérance de panne n'a encore été étudiée pour notre projet.

3. Mode de commutation.

La topologie d'interconnexion n'est pas le seul facteur influençant la communication dans un réseau. Dès lors qu'on a une interconnexion non-complète, une communication bi-point peut faire appel à un certain nombre de nœuds pour servir de relais. Plusieurs méthodes existent pour faire coopérer les différents nœuds entrant en jeu dans la communication, chacune présentant des caractéristiques propres qui déterminent leurs domaines d'utilisation.

3.1. La commutation de circuit virtuel.

La commutation de circuit virtuel consiste à établir complètement le chemin entre

la source et la destination. La communication peut alors se dérouler comme s'il existait un lien physique direct entre eux deux. Les inconvénients de ce mode de commutation sont bien connus : l'établissement d'un chemin est une opération coûteuse qui n'est rentable que si le volume d'information à échanger est en rapport suffisamment important. C'est la raison pour laquelle nous ne nous attarderons pas sur cette technique, car le parallélisme massif requiert un grain de calcul fin et donc :

- des volumes élémentaires d'information relativement petits,
- un nombre de processeurs important tendant à augmenter la longueur des chemins et en conséquence leurs délais d'établissement et de libération.

Ce mode de commutation est de fait l'apanage des architectures à grain moyen ou grossier, qui sont en général bâties autour d'un commutateur *cross-bar* ou à étages, ou encore autour d'une structure en hypercube, toutes topologies où le diamètre du réseau varie en fonction du logarithme du nombre de processeurs.

L'existence d'un chemin réservé entre la source et la destination leur permet par contre de communiquer de façon synchrone entre nœuds distants, propriété qui disparaîtra avec le passage de message. Le synchronisme n'est pas toujours un avantage, car il appelle le rendez-vous et peut être, de ce fait, à la source d'une perte de parallélisme.

3.2. Le passage de message.

Ce mode de commutation est basé sur le fait qu'une communication distante peut être décomposée en autant de communications de voisin à voisin qu'il y a de nœuds intermédiaires sur le chemin qu'elle emprunte. L'information transmise devient alors une entité dénommée *message*, qui circule de nœud en nœud jusqu'à sa destination. C'est une technique asynchrone par essence, car aucune assurance n'est donnée quant au temps d'acheminement (sinon qu'il est fini), mais il faut quand même se rappeler qu'elle est implémentée par une série de communications synchrones entre processus logiciels et matériels : un tampon peut être vu comme un processus qui exécute cycliquement deux opérations, réception et émission. De ce point de vue, la commutation de circuit virtuel impose un rendez-vous entre toute la série des acteurs de la communication, alors que le passage de message n'impose qu'une séquence de rendez-vous binaires.

Le passage de message est tout à fait adapté lorsque le grain de communication est assez fin pour que le message puisse être transmis entièrement par un lien parallèle. Si la taille des messages est plus large que le lien physique, plusieurs solutions sont possibles :

- découper les messages "algorithmiques" en plusieurs messages, avec comme conséquence la duplication des informations de routage et la duplication du travail de routage

- sérialiser la communication physique. Cette solution connue sous le nom de *store-and-forward* est l'une des plus mauvaises qui soit, car le délai d'acheminement d'un message varie comme d.m/n, où d est le nombre de nœuds à traverser, m la taille du message et n la taille du flit[1].
- *wormhole routing*
- *virtual-cut-throuh.*

3.3. Le *Wormhole routing*.

Le transfert *wormhole* [SEI84] résulte de l'exploitation de la possibilité de recouvrement entre la réception et la réémission lors d'un transfert sérialisé. Contrairement au transfert *store-and-forward* (fig. 6.), le message acheminé n'est alors jamais localisé dans un seul nœud, mais est distribué sur m/n nœuds successifs (fig. 7.) d'où son qualificatif *wormhole* ("trou de ver") : la tête du message ouvre le chemin et la queue le referme. En ce sens, le *wormhole* est une forme de communication par commutation de circuit, mais avec un recouvrement entre les opérations d'établissement du circuit, de communication et de libération du circuit. Le pourcentage du réseau utilisé pour l'envoi d'un message en est fortement réduit, mais le point de synchronisation présent dans la commutation de circuit disparaît, ce qui classe bien le *wormhole* parmi les communications par messages d'un point de vue sémantique.

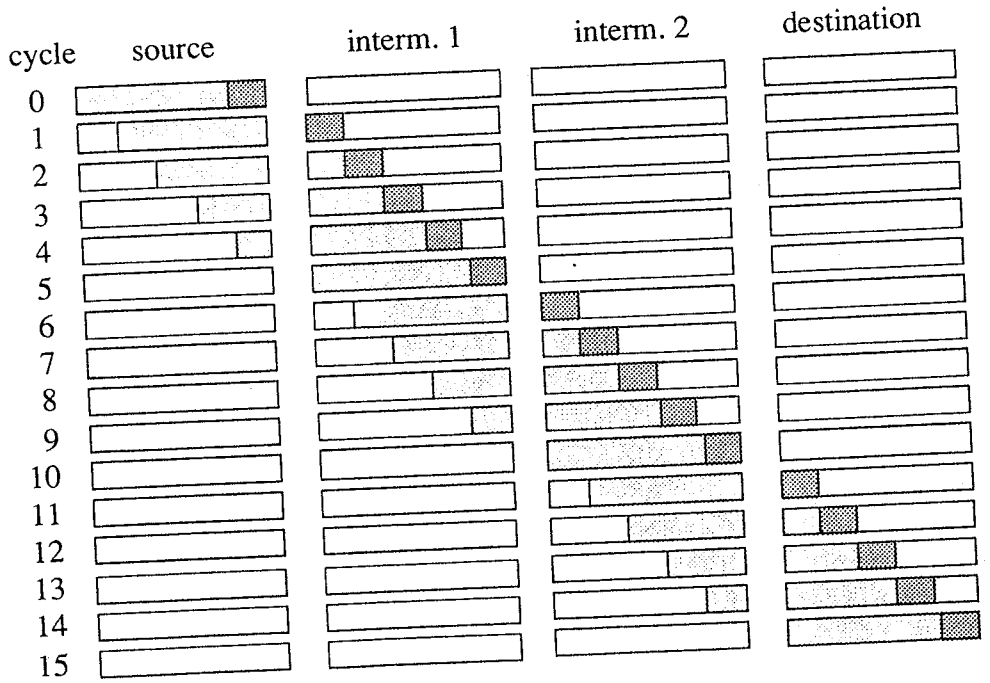


Fig. 6. — *Chronogramme de transfert en store-and-forward.*

---

1 flit : FLow control unIT, c'est le plus petit quantum d'information échangeable entre deux éléments du système de communication.

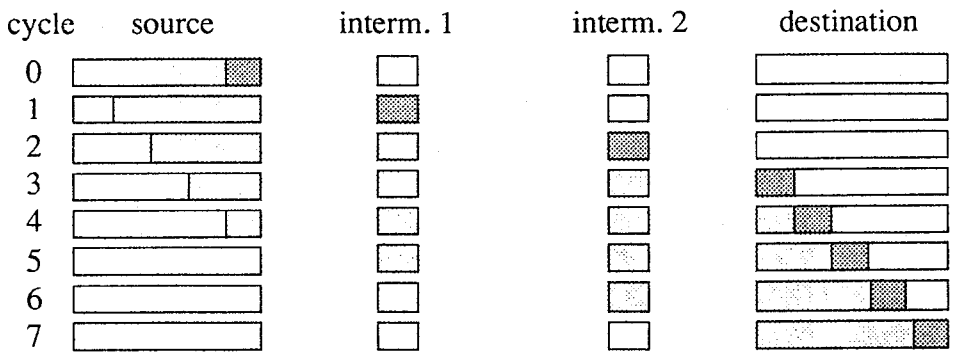


Fig. 7. — *Chronogramme de transfert en wormhole.*

Le délai d'acheminement est bien meilleur que pour le *store-and-forward*, puisqu'il varie comme d+m/n. D'un point de vue capacité de mémorisation, le routage *wormhole* est plus économique mais, corollairement, le pourcentage de tampons utilisés est plus important entraînant un taux de collisions plus élevé.

De bonnes techniques d'implémentation existent, permettant la réalisation d'aiguilleurs modulaires pour les n-cubes k-aires (un module par dimension), et l'usage de messages de longueur variable au niveau de la taille de la donnée comme de celle de l'adresse (encodage par préfixe [FLA87]).

3.4. *Virtual cut-through.*

Assez proche du *wormhole routing*, la technique dite de *virtual cut-through* [KER79] se différencie par le fait qu'un message bloqué est retiré du réseau momentanément et mémorisé dans le nœud, ceci jusqu'à ce que le canal qu'il requiert soit de nouveau libre. Pour comprendre l'intérêt de ce mode de commutation, il faut bien voir qu'un blocage de la tête du message n'empêche pas la queue d'avancer dans une certaine limite, réduisant ainsi la contention dans le réseau. Nous verrons que nous pouvons envisager des intermédiaires entre *virtual cut-through* et *wormhole*, et même bénéficier de certaines particularités techniques pour les améliorer.

B. Le cas du réseau cellulaire.

Nous avons vu dans le début de ce chapitre que les topologies en grilles sont les plus adaptées à nos principaux objectifs (incrémentabilité, simplicité). Concernant le routage, les stratégies complexes de type adaptatives ne sont pas dans un contexte qui leur permettraient de se montrer avantageuses ; nous leur avons donc préféré une stratégie triviale X-Y, jugée plus compatible avec le grain matériel visé. En revanche, choisir un mode de commutation demande une étude plus poussée. En effet, la finesse du grain de parallélisme produit des messages de faibles tailles, qui nous situent juste en deçà de messages que nous pourrions transmettre en parallèle.

Pour mener à bien cette étude, nous adoptons ici une démarche expérimentale. Le rôle du réseau étant en fin de compte de permettre aux processus implantés sur les cellules de communiquer, une étude des besoins *réels* en communication de ces processus est nécessaire pour avoir une idée du niveau de performance que le réseau devra assurer. Un certain nombre d'exemples de programmes ont été codés pour le réseau cellulaire, aussi divers par les domaines dont ils relèvent que par les données qu'ils manipulent ou par leur structure algorithmique. Mis bout à bout, ils forment un *benchmark* que nous supposerons représentatif. Ces programmes sont exposés en détail dans le chapitre IV portant sur la programmation et en annexe 1. Nous ne les analyserons dans ce chapitre que du seul point de vue de la communication.

Nous détaillerons, évaluerons et comparerons ensuite plusieurs structures de routeurs, à la lumière des performances obtenues par simulation sur l'exécution du benchmark.

1. Caractérisation des applications.

L'analyse des programmes est divisée en analyse des caractéristiques statiques (topologie, types des messages, etc) et analyse des aspects dynamiques (sensibilité à la communication, charge, structure temporelle).

1.1. Etude statique.

Tous les programmes dont nous disposons correspondent à des schémas de parallélisme statiques. Il n'y a pas de programme qui utilise la création dynamique de processus, sauf peut être le crible d'Eratosthène et le problème des huit reines, mais dans ces exemples, ce sont toujours les mêmes types de processus qui sont créés, selon une stratégie bien déterminée : en pratique on se ramène à un schéma statique à processus initialement non actifs, la création n'étant plus qu'une activation initiale, ou bien à un multiplexage de processus identiques sur un nombre fixe de cellules. Toutes les caractéristiques statiques sont résumées dans le tableau 1, page 46.

nom du programme	type d'algorithme	topologie logique	connect. moyenne	distance moyenne	longueur des msg.	type des messages
Conway2D	automates	grille 2D	4	1,5	1 octet	bits
Conway1D	dataflow	grille 1D	4	1,5	1	bits
Lattice Gaz Model 1D	dataflow	grille 1D	2,6	2	1	bits
Distance entre mots	tableau systolique	grille 2D	4	1,33	1	octets
XOR neuronal	dataflow	graphe irrégulier	3,66 *	1,52 *	2	virgule fixe
XOR synaptique	dataflow	graphe irrégulier	2,66 *	1,61 *	2	virgule fixe
Multiplication de matrices creuses	dataflow irrégulier	grille 2D de sous-réseaux homogènes	3	1,66	5	flottants 32 bits + octets
Plus court chemin	automates	graphe irrégulier	8 *	6,14 *	1	octets
Plus court chemin (Chandy)	automates	graphe irrégulier	8 *	6,14 *	1	octets
Crible d'Eratosthène	dataflow irrégulier	grille 1D	2	1	2	entiers 16 bits
Tri de chaines par insertions	dataflow	grille 1D	2	1	variable	chaînes C
Tri de chaines par interclassement	dataflow irrégulier	arbre	3	1,9 *	variable	chaînes C
Tri en serpentin	automates	grille 2D	4	1	1	octets
Tri à bulles	automates	grille 1D	2	1	1	octets
Tri en hélice	automates	≈ tore 2D	4	2,35	1	octets
Simulation logique à échéancier	dataflow irrégulier	graphe irrégulier	*	*	2	etat + date
Problème des huit reines	automates	grille 1D	2	1,2	12	processus
FFT Lustre	dataflow	graphe irrégulier	14	2,7*	2	entiers 16 bits

* variable en fonction des données, du placement

Tableau 1. — *Caractéristiques statiques des applications.*

1.1.1. Types d'algorithmes.

Les programmes peuvent être classés en deux catégories selon que des résultats partiels circulent dans le réseau (graphes data-flow) ou que la communication ne serve que de support d'interaction entre automates (réseaux d'automates).

Une distinction plus fondamentale peut être posée sur la régularité des flots de messages. Beaucoup de programmes, les réseaux systoliques par exemple, ont des cellules qui consomment et produisent un nombre déterminé de messages à chaque itération globale, indépendamment de leur régularité topologique. D'autres programmes, éventuellement topologiquement réguliers, ont des flots de communication irréguliers. Les cellules s'échangent des événements : dans la simulation logique par échéancier, un événement en entrée ne provoque un événement en sortie que si la valeur de la sortie change.

Notre jeu d'exemple ne présente pas de programmes où les voies de communications ne sont pas statiques, c'est à dire où les processus choisissent dynamiquement leurs interlocuteurs. Ce type de communication pose un problème général de contrôle de flux pour lequel nous n'avons pas encore de solution validée.

1.1.2. Topologies logiques.

Si l'on classe les topologies logiques en fonction de la façon dont elles sont placées sur un réseau en grille 2D, nous obtenons cinq catégories de base.

a) *Topologies linéaires* : la forme naturelle est une grille K x N où N est la taille du problème et K est une constante ou une fonction de N très inférieure à N. La ligne est alors repliée en colimaçon ou en serpent pour aboutir à un facteur de forme plus proche du carré. Les propriétés de localité sont conservées (un lien logique correspond à un lien physique et un seul). Il faut distinguer le cas où la communication avec l'hôte ne se fait que par les deux extrémités de la ligne (tri par insertion, crible d'Eratosthène), et celui ou la communication peut se faire sur toute la ligne (automates cellulaires et tableaux systoliques repliés) : le placement rend impossible l'accès aux cellules du centre du réseau. Une étude détaillée de ce dernier cas est présentée chap. IV § 3.4.

b) *Topologies en grilles bidimensionnelles* : c'est le cas le plus favorable, puisque l'affectation des processus aux cellules se fait par superposition de la structure logique sur la structure physique, au facteur d'échelle près (un processus sur un carré de cellules, un processus par cellule, un carré de processus par cellule).

c) *Topologies bidimensionnelles toriques ou non orthogonales* : l'association lien logique - lien physique ne peut plus être conservé, mais nous avons vu qu'un placement simple permet de borner supérieurement la longueur de chemins utilisés par les liens logiques (en général 2) et, par conséquent, le temps d'acheminement.

d) *Topologies régulières de dimension supérieure à 2* (hypercubes, *n*-cubes *k*-aires) *ou arborescentes* : un placement non trivial doit être fait, qui introduit éventuellement des cellules relais si la taille du graphe est trop grande pour permettre un placement compatible avec la portée d'adressage. A chaque lien logique correspond une suite, qui peut devenir très longue, de liens physiques. On peut éventuellement utiliser les relais pour réaliser un "pipeline" d'acheminement. Une borne *inférieure* peut alors être posée sur le débit des liens logiques, quelque soit leur longueur physique, mais il faut toutefois que l'algorithme s'y prête. Chacune de ces topologies peut donner lieu à l'écriture de programmes de placement spécifiques.

e) *Topologies aléatoires* : ces topologies ne se prêtent pas plus à un placement direct sur la grille que les précédentes. On doit avoir recourt à des algorithmes de placement généraux comme le recuit simulé. Notons que le résultat du placement n'est pas forcément mauvais comme il l'est pour la catégorie précédente, en particulier la portée de l'adresse n'est pas forcément insuffisante (l'utilisation du programme de placement développé pour le compilateur LUSTRE l'a montré [PAY91]).

Ces catégories peuvent être combinées : si un processus est trop gros pour tenir sur une cellule, on le divise en un ensemble sous-processus possédant leur propre topologie. Par exemple, le programme de multiplication de matrices creuses a une structure générale en grille, mais chaque nœud de la grille est un graphe aléatoire (voir annexe 1).

1.1.3. Types de données.

Du fait de la taille des cellules, les données qu'elles peuvent traiter sont généralement d'un type simple. Certains programmes manipulent des nombres réels en virgule fixe (réseaux de neurone) ou en virgule flottante (multiplication de matrices creuses). Les autres programmes manipulent des entiers codés sur un ou deux octets. La précision observée des entiers est à prendre avec précaution, car des programmes à échelle réelle nécessiteraient parfois des magnitudes plus grandes. Seuls les automates cellulaires ont des données de type bit. Lorsque les données sont complexes, ce sont soit des chaînes (une douzaine de caractères environ), soit des données de type structuré, comprenant plusieurs champs d'information hétérogènes : la simulation à échéancier, la multiplication de matrices creuses "datent" les messages échangés.

Le tableau 1. ne mentionne que les messages de données, mais il ne faut pas oublier la présence de requêtes, où le champ donnée est rarement utilisé.

1.1.4. Niveaux de recouvrement.

Les programmes permettent tous un recouvrement entre réception des données et calcul-émission, sauf les programmes de tri en place (voir étude dynamique, §B.1.2). A ce niveau intra-cellulaire vient souvent s'ajouter un niveau de recouvrement algorithmique extra-cellulaire (niveau "systolique"), et parfois un niveau

supplémentaire lié à la décomposition d'un processus en plusieurs cellules (multiplication de matrice, recherche de plus court chemin).

1.2. Etude dynamique.

Nous commencerons par voir quelle est l'influence des délais de communication sur le temps d'exécution moyen, puis nous chercherons à comprendre plus en détail cette influence au travers de la structure temporelle des programmes, pour terminer avec la charge moyenne du réseau.

1.2.1. Influence de la latence de communication sur le ralentissement.

Comme nous l'avons suggéré en introduction de chapitre, le rôle du système de communication est de permettre une exploitation maximale de parallélisme, qui se traduit par une activité des éléments de calcul la plus haute possible. L'objectif est en fin de compte de minimiser la durée d'exécution des programmes au meilleur coût, et c'est ce critère que nous adopterons pour nos évaluations ; nous n'utiliserons les notions de bande passante et de latence que secondairement. En effet, une bande passante surdimensionnée, si elle minimise effectivement le temps d'exécution, est en partie inutile et nous éloigne donc d'un bon rapport performances/coût.

Nous supposerons que le temps d'exécution minimum est obtenu avec un système de communication présentant une latence nulle[1], c'est à dire que le message est reçu le cycle suivant son émission. Ce temps d'exécution, mesuré en nombre de cycles de base de la partie traitement des cellules, nous servira de référence. La baisse de performance sera mesurée en pourcentage par rapport à cette référence.

Les systèmes de communication que nous serons à même de réaliser introduiront bien entendu des latences non nulles. Afin de maîtriser ces latences, les programmes seront simulés avec un système de routage hypothétique, dont la *latence unitaire* - durée d'un transfert entre cellule voisine - est directement paramétrable. La latence de communication entre deux cellules sera donc égale à leur distance de Manhattan multipliée par la latence unitaire, le tout mesuré en cycles-processeur :

$$L(P_{i,j} - P_{i',j'}) = LU.(|i'-i|+|j'-j|+1)$$

---

[1] C'est une hypothèse qui, dans notre machine, est en toute rigueur fausse, car un message incident vole un cycle au programme pour être stocké. Il existe des cas où une latence non nulle occasionne un vol de cycle dans des sections moins critiques vis-à-vis de la communication qu'une latence nulle, influant ainsi favorablement sur le temps d'exécution. En tout état de cause, ce phénomène de décroissance anormale est rare et toujours d'amplitude très légère.

La constante 1 qui intervient est une correction pour se rapprocher du fonctionnement réel d'un réseau ; elle représente le temps nécessaire pour transférer un message arrivé à destination du tampon du routeur dans les tampons d'interface routeur-processeur. Cette latence unitaire recouvre à la fois le temps de transfert effectif et les phénomènes de contention, ces derniers sont modélisés à un niveau global et non à une échelle microscopique.

Nous commencerons par établir la réaction de notre jeu de test lorsque la latence unitaire augmente *(fig. 8)*. Le ralentissement moyen est assez harmonieux, c'est à dire devenant rapidement linéaire. Il est relativement faible, puisqu'une latence unitaire de 5 cycles donne un ralentissement moyen inférieur à 10%.

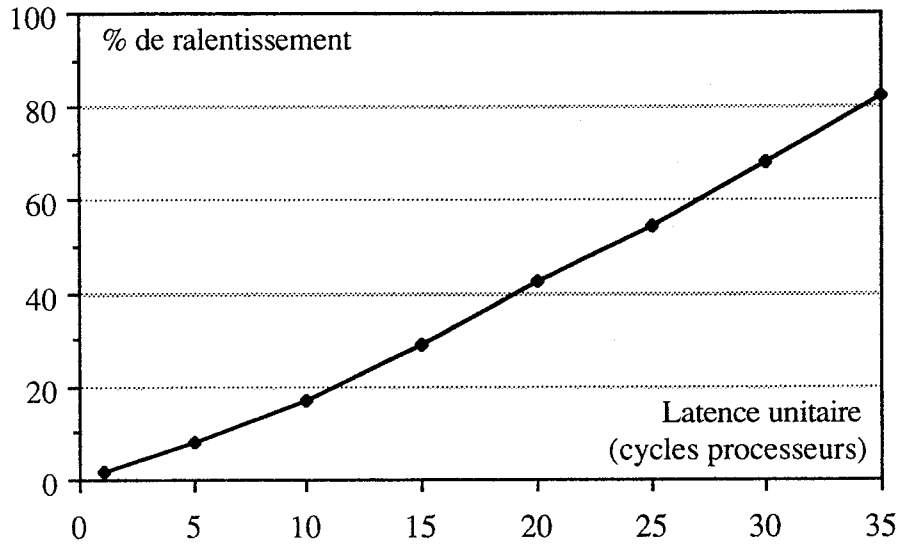


Fig. 8. — *Influence de la communication sur le ralentissement moyen.*

Le ralentissement cumulé cache néanmoins des comportements très disparates. Qualitativement, les courbes de ralentissement peuvent présenter :

- un palier initial, marquant une certaine insensibilité à la lenteur des communications *(fig. 9)*. Ce palier initial peut être très important comme pour le réseau de neurones (voir chap. IV, § 3.3), où il y a une conjonction d'un bon recouvrement entre couches de neurones et d'une période (temps entre deux lectures d'un même canal d'entrée, égale ici à 750 cycles) assez élevée en raison du traitement en virgule fixe et du nombre d'opération à effectuer.

- une non-monotonie de la courbe *(fig. 10)* ou de sa dérivée, plus moins accidentée suivant la régularité du problème.

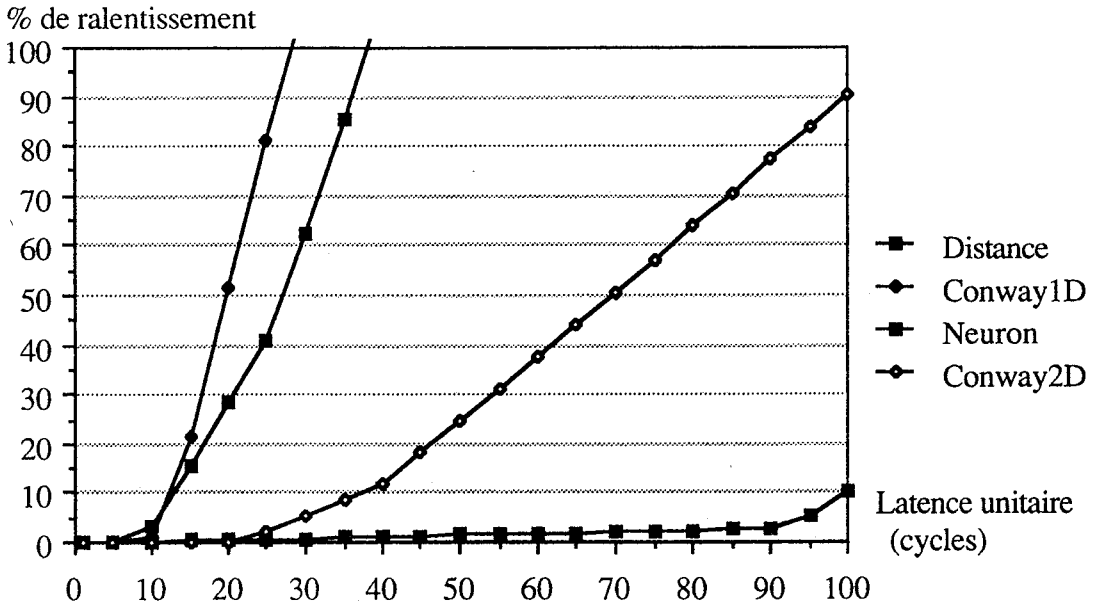


Fig. 9 — *Influence de la communication sur le ralentissement*
(quelques programmes présentant des paliers initiaux)

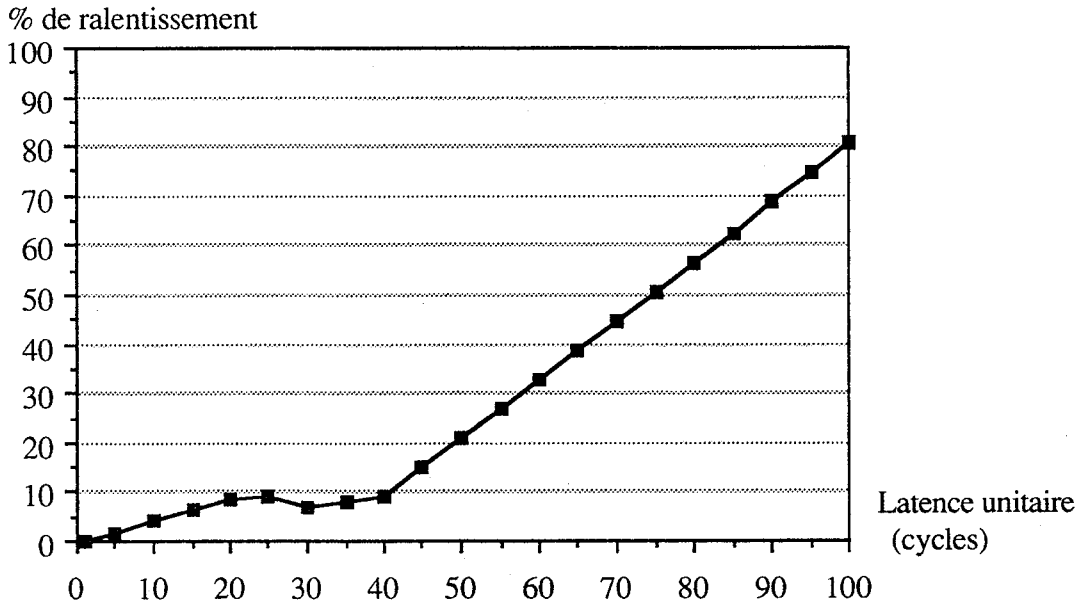


Fig. 10 — *Influence de la communication sur le ralentissement*
(Tri de chaînes par insertion, présentant une décroissance)

Lorsqu'il y a un palier initial, nous désignerons par *latence unitaire critique* (LC) le point à partir duquel la courbe s'infléchit brusquement, le ralentissement devenant plus que marginal. Cette valeur mesure le degré de tolérance de l'application à la lenteur du routage.

nom du programme	forme de la courbe	latence unitaire critique	latence unitaire pour un ralentissement de 10%
Conway2D	palier initial	10	
Conway1D	palier initial	21	
Lattice gaz mod.1D	palier initial	3	
Distance entre mots	palier initial	8	
Réseau neuronal	palier initial	92	
Réseau synaptique	palier initial	15	
Multiplication de matrices creuses	linéaire pour densité=1 ≈ linéaire, dérivée non monotone si d≠1		15 pour densité=1 14 pour densité=1/4
Plus court chemin	courbe très accidentée, avec quelques décroissances		3
Plus court chemin (Chandy)	courbe très accidentée, avec quelques décroissances		3
Crible d'Erathostène			140
Tri de chaines par insertions	présence d'une décroissance		40
Tri de chaines par interclassement	palier initial	6	
Tri en serpentin	linéaire		2
Tri à bulles	linéaire		2
Tri en hélice	linéaire		2
Simulation logique à échéancier	palier initial	6	
Problème des huit reines	palier initial, présence d'une décroissance	5	

Tableau 2. — *Influence de la latence de communication.*

1.2.2. Analyse des comportements.

L'explication de la diversité des comportements observés est à chercher dans la structure temporelle des programmes. Prenons le cas de "Conway2D", qui est relativement facile à analyser grâce à son fonctionnement de type SIMD (même programme pour toutes les cellules, fonctionnant dans un synchronisme lâche). Chaque cycle de cellule peut se décomposer en trois phases et 5 sous-phases *(fig. 11)*.

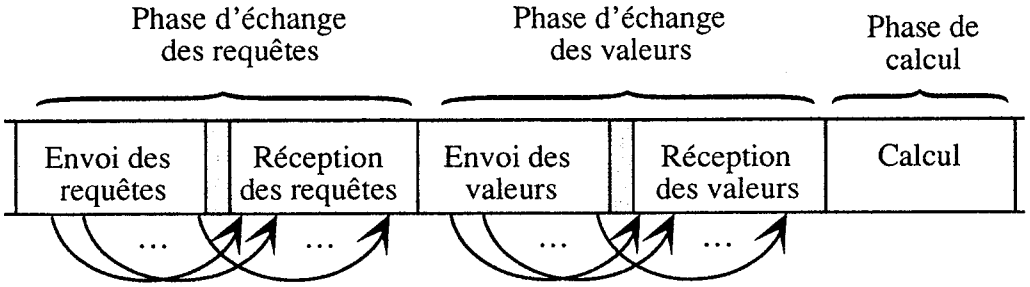


Fig. 11. — *Décomposition d'une période de calcul de Conway2D.*

Des phases d'attentes peuvent s'intercaler entre les phases d'envoi et les phases de réceptions correspondantes si les messages ne sont pas acheminés suffisamment rapidement. A l'intérieur d'une même phase peuvent aussi s'insérer des délais lorsque par exemple deux émissions sont rapprochées et que le routeur n'a pas eu le temps de prendre en charge le premier message au moment où le programme tente la seconde émission (ce type de collision n'est pas modélisé à ce niveau, il le sera dans l'étude des routeurs). De même, lorsqu'on a conflit d'accès à un tampon de réception, l'une des réceptions sera différée pendant un cycle entier de l'algorithme de routage, rallongeant ainsi le délai d'acheminement.

Les diverses phases étant de longueur fixe, ces temps critiques pour l'acheminement seront stables et caractéristiques du programme.

Les temps critiques se trouvent à l'intérieur de chaque série d'échanges : il s'agit de la plus courte durée séparant l'émission d'un message de sa réception. Si la latence de communication est inférieure à cette durée $T\varphi$, la communication se recouvrira entièrement avec l'exécution du programme et n'aura donc aucune influence sur le temps d'exécution global. Dans le cas contraire, elle introduira une attente de $L.D-T\varphi$ cycles. Ces attentes seront cumulées de phase en phase et de période en période, ce qui nous permet d'exprimer le temps d'exécution pour k périodes de la façon suivante :

$$Texec = k\,(\,PP + \sum_{\text{phase } \varphi} \min(0, L.D-T\varphi))$$

L'examen du programme fait apparaître $T\varphi req=64$ et $T\varphi val=112$. $D\varphi val$ et $D\varphi req$ correspondant à une communication diagonale de distance 2 cellules (donc de 3 tampons), ces dépendances causeront des points d'inflexion de la courbe $T=f(L)$ pour $L=64/3=21$ et $L=112/3=37$, points que nous avions figure 9.

Nous pouvons dire que du point de vue des dépendances de communication, ce programme est bien écrit. En plaçant les émissions et les réceptions dans le bon ordre et au bon endroit, il maximise le temps disponible pour acheminer un message. Parmi les programmes testés, d'autres sont nettement bien moins conçus. Les plus caricaturaux

sont certainement les algorithmes de tri par échange de paires (Tri à bulle, Tri en hélice, Tri en serpentin) qui insèrent dans le chemin critique tout le temps nécessaire pour acheminer deux messages *(fig. 12)*. La latence critique est donc nulle, toutes les dépendances introduisent des délais, ce qui se traduit par un comportement purement linéaire.

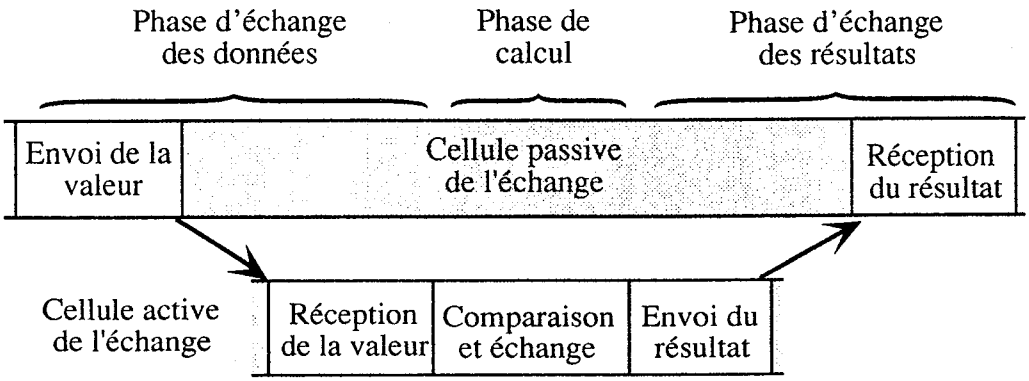


Fig. 12. — *Décomposition d'un cycle d'échange du tri à bulles.*

Il faut toutefois moduler ce comportement catastrophique :

- les exemples programmés trient des octets, un programme réel manipulerait des données plus grosses, ce qui introduirait un recouvrement entre une série d'émissions et les réceptions correspondantes relativisant les temps d'inactivité ;
- l'algorithme peut être mieux programmé, par exemple en faisant effectuer à la cellule passive de l'échange un traitement miroir de celui de la cellule active, ce qui permet de supprimer la phase d'échange des résultats (la cellule passive garde le min, la cellule active garde le max).

Comme pour les réseaux systoliques, la dynamique des programmes doit donc être soigneusement étudiée. Il s'agit ici de permettre un parallélisme maximum en établissant un recouvrement entre calcul et acheminement des messages. Nous pouvons dire qu'à l'instar des processeurs, les programmes ont aussi une architecture temporelle, qui doit s'articuler correctement avec l'architecture spatiale. Bien entendu, lorsque la topologie et les débits sont aléatoires et lorsque les processus hétérogènes, l'analyse de la structure temporelle est très complexe. Les méthodes nécessaires pour la mener à bien sortent très largement du cadre de cette étude. Le cas le plus pénible à prendre en compte est celui où les processeurs sont multiprogrammés. Un retard même très faible sur l'acheminement d'un message peut modifier le résultat de l'évaluation d'exécutabilité du processus auquel il est destiné ; une modification de l'ordre d'exécution des processus dans la cellule peut en résulter qui peut éventuellement réduire ou augmenter le degré de concurrence global, selon les dépendances qui existent dans le programme. Un petit délai peut en engendrer ou en supprimer un grand autrement que par simple accumulation : c'est une sorte d'*effet papillon*. Cet aspect de la multiprogrammation des cellules doit être généralisé à toutes les applications où les

processeurs décident entre plusieurs traitements selon la présence ou la non présence d'une donnée (structure ALT d'Occam). Sur la courbe de ralentissement en fonction de la latence de communication du tri de chaîne par insertion (structure ALT) nous pouvons observer une rupture de monotonie, celles des algorithmes de recherches de plus court chemin (multiprogrammés) deviennent très franchement erratiques pour des valeurs de latence élevées. La figure 13. schématise une situation où une augmentation des performances peut résulter d'un retard sur l'acheminement d'un message.

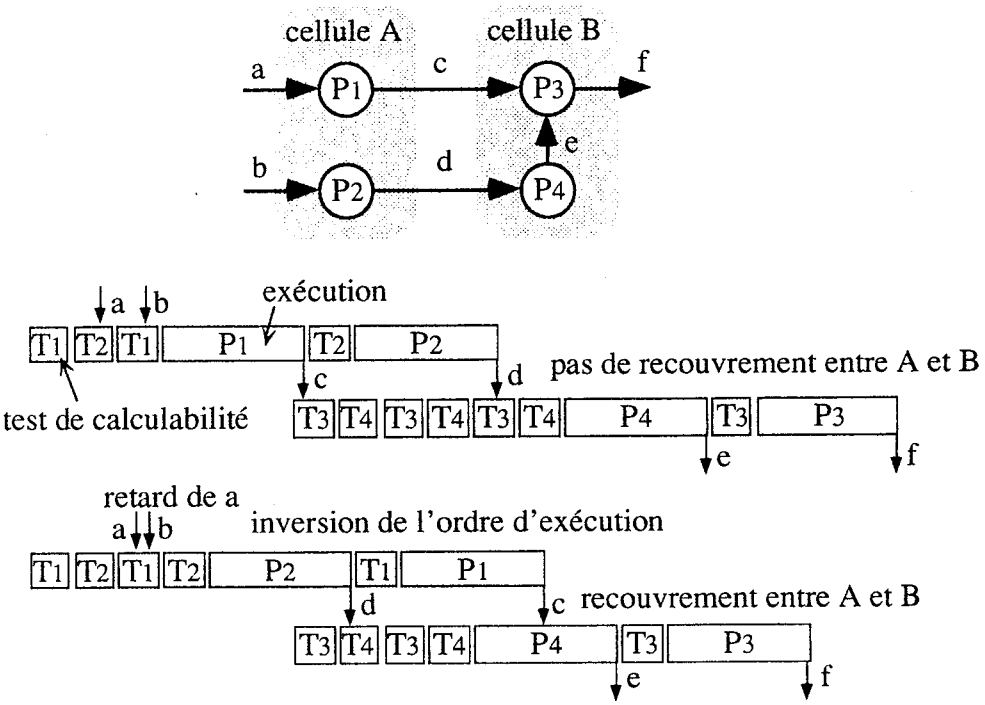


Fig. 13. — *Cas où un retard d'acheminement réduit le temps d'exécution.*

On pourrait penser que les phénomènes de ce genre sont aplanis au niveau d'un réseau de grande taille, pour ne laisser voir qu'un comportement global monotone. Il semble expérimentalement qu'un tel lissage n'a pas lieu, et que nous soyons réellement en présence d'un phénomène plus ou moins chaotique, avec présence de "résonances" (mise en concordances de phase d'anomalies de comportement locales) pour certaines valeurs de latence, qui peuvent induire des anomalies globales de amplitudes notables. Nous n'approfondirons pas ce point car il concerne des valeurs de latence élevées, et n'offre donc pas d'un intérêt fondamental pour le sujet qui nous préoccupe.

1.2.3. Taux d'émission et charge du réseau.

D'un point de vue quantitatif, la densité de communication est assez faible. Le tableau ci-dessous rassemble les taux d'émissions et les charges moyennes observés pour chaque application avec un routage où la latence unitaire est égale à 1 cycle processeur (ramenée à une cellule et classées par charge décroissante).

Programme	Taux d'émission (msg/cell.cycle)	Chargemoyenne des routeurs (msg/cell.cycle)
Distance entre mots	0.0377	0.2515
Tri (helice)	0.0218	0.1614
Tri à bulles	0.0232	0.1582
Tri (serpentin)	0.0228	0.1380
FFT lustre	0.0172	0.1107
Conway1D	0.0170	0.1093
Lattice Gaz Model 1D	0.0150	0.1054
Conway2D	0.0187	0.0860
Tri de chaines par insertion	0.0136	0.0864
Multiplication de matrices creuses d=1	0.0089	0.0607
Réseau synaptique	0.0080	0.0575
Multiplication de matrices creuses d=0.25	0.0070	0.0474
Recherche de plus court chemin (chandy)	0.0036	0.0406
Recherche de plus court chemin	0.0026	0.0297
Problème des huit reines	0.0045	0.0275
Tri de chaines par interclassement	0.0028	0.0218
Réseau neuronal	0.0025	0.0161
Simulation logique à échéancier	0.0021	0.0121
Crible d'Eratosthène	0.0004	0.0025

Tableau 3. — *Taux d'émission et charge moyenne.*

Le taux d'émission le plus élevé correspond à un message par processeurs en moyenne tous les 30 cycles et le plus bas à un message tous les 2500 cycles. La distance moyenne du destinataire étant faible, les charges observées le sont aussi. Même si nous ne pouvons pas en déduire le taux d'activité moyen imposé aux routeurs (celle-ci dépendant de leur structure exacte), nous pouvons supposer que s'ils seront capables de tenir une latence unitaire proche de 1 cycle, ils seront le plus souvent au repos.

Quelles conclusions tirer pour la communication de cette étude des quelques programmes que nous avons écrit pour le réseau cellulaire ?

1 ° Un routage adaptatif ne se justifie pas du point de vue performance.

2 ° Le taux de collisions, lié à la charge du réseau, sera sans doute faible.

3 ° Une marge de manœuvre existe qui permettra d'envisager l'usage de routeurs de plus faible débit, mais plus économiques.

4 ° Un travail sur la structure temporelle des programmes permet de réduire fortement la sensibilité vis-à-vis de la communication.

2. Solutions envisagées et évaluation.

Nous pouvons maintenant décrire plus en détail les architectures de routeurs que nous pourrions envisager d'utiliser. Nous tenterons de les définir aussi précisément que possible, pour pouvoir les évaluer sur des bases réalistes, mais nous serons généralement obligés de nous en tenir à une analyse algorithmique et architecturale. Les données telles que la complexité et la fréquence de fonctionnement ne sont en effet disponibles actuellement que pour un petit nombre de configurations, ce qui nous obligera à travailler avec différentes hypothèses de fréquence.

Nous examinerons successivement un routeur parallèle (solution initiale, peu réaliste), des routeurs sériels *(store-and-forward)*, des routeurs *wormhole* et une structure différente qui consiste à utiliser un bus unique pour faire communiquer les cellules d'un même chip. Ces solutions seront ensuite mesurées, dans diverses configurations de paramètres, en exécutant par simulation (niveau cycle) le benchmark que nous venons de décrire. Nous tenterons enfin, dans la mesure du possible, de comparer ces solutions.

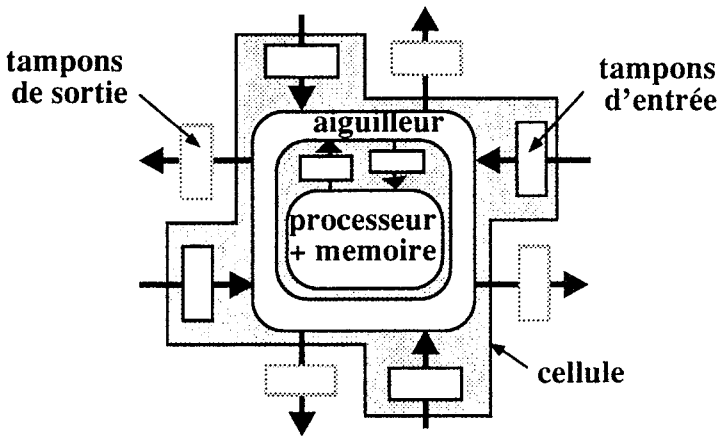


Fig. 14 — *Structure d'une cellule (rappel).*

2.1. Transfert parallèle.

Le transfert parallèle doit son nom au fait que la donnée (le message) est transmise en un seul bloc, tous les bits à la fois *(fig. 16)*. Comme dans la plupart des méthodes sur lesquelles nous nous pencherons par la suite, les cellules communiquent par l'intermédiaire de tampons d'entrées/sortie *(fig. 15)*. Chacun de ces tampons peut être modélisé comme une mémoire partagée par deux cellules, associée à un mécanisme simple d'exclusion mutuelle *(fig. 17)* [COR88]. Les tampons étant unidirectionnels, nous avons deux tampons de directions opposées entre deux cellules voisines.

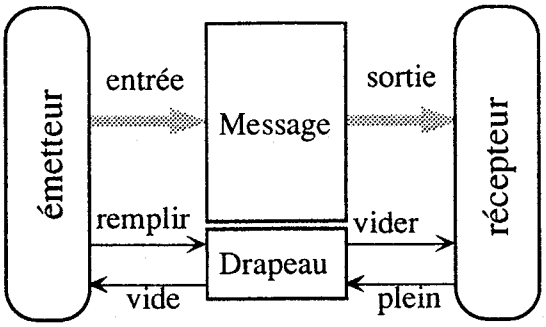


Fig. 15 — *Tampon de communication.*

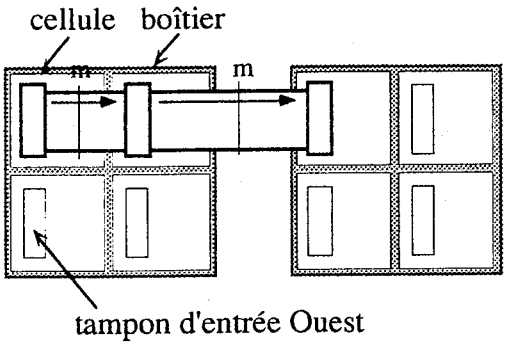


Fig. 16 — *Transfert parallèle.*

Le registre de mémorisation peut être implémenté par une bascule simple, car il ne peut pas y avoir simultanément lecture et écriture. En effet, les décisions de transmissions étant distribuées et il est tout à fait impossible à un routeur de savoir si un tampon de sortie plein sera libre au cycle suivant, donc s'il peut stocker un message dans la partie maître. Un cycle où le tampon est vide est obligatoirement introduit entre deux messages, ce qui supprime tout risque d'écrasement.

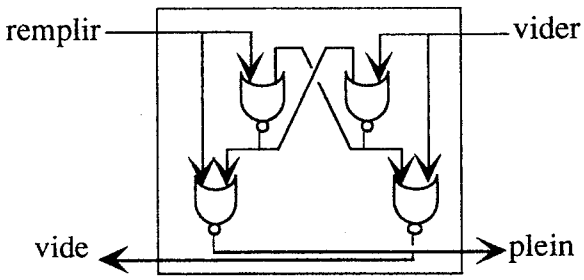


Fig. 17 — *Mécanisme d'exclusion mutuelle.*

Lors de tout transfert, le scripteur se doit de vérifier au préalable que le tampon est vide (signal "vide"), puis après affirmation de la donnée verrouiller et valider celle ci au moyen de la commande "remplir". Symétriquement, le lecteur doit s'assurer qu'une donnée est bien présente au moyen du signal "plein", puis par la commande "vider", l'invalider après l'avoir consommée.

Cette méthode de transfert a été historiquement perçue comme une méthode idéale, que l'on adaptait au pied levé aux exigences de la technologie lors de la conception. Ces exigences - encore actuellement incontournables - se ramènent toutes aux implications d'un chemin de données d'une largeur redoutable (égale à la taille du message, soit pour nous 24 bits) :

- nombre de broches démesuré
- part importante de silicium consacrée à la réalisation des tampons
- perte de surface liée aux pistes de connexion entre tampons à l'intérieur de chaque cellule

Avec des cellules d'environ 20 KT, les capacités d'intégration actuelles permettent de tabler sur des circuits comprenant entre 4x4 (≈ 300 KT) et 8x8 (≈ 1,2 MT) cellules. Il est clair qu'en l'absence de technique de packaging miraculeuse, c'est le nombre de broches qui va devenir crucial *(fig. 18)*, alors que les autres facteurs limitatifs resteront proportionnellement constants.

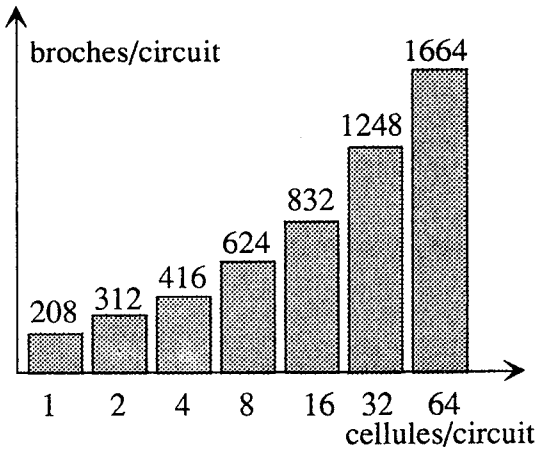


Fig. 18 — *Nombre de broches en fonction de l'intégration.*

Le problème se manifeste pourtant déjà avec une seule cellule par boîtier, ainsi que l'attestent les projets antérieurs de notre équipe. Le réseau de simulation logique de Ph. Objois et R. Cornu-Emieux utilisait des messages de 10 bits et une liaison série entre boîtiers ([COR88] pp. 122-125), le réseau de reconstruction d'image de D. Lattard ([LAT89]) une liaison 16 bits par 16 bits pour des messages de 48 bits et une seule cellule par boîtier.

Ces deux projets utilisaient la sérialisation pour réduire la connectivité à l'interface des boîtiers. Aussi semble-t-il naturel de commencer l'étude de modes de transfert "réalistes" par le transfert sériel. Nous profiterons de la similitude structurale pour intégrer à cette étude le transfert parallèle, lequel n'est qu'un cas particulier où la taille du flit est égale à la taille du message.

2.2. Transfert série

2.2.1. Présentation.

Le transfert série se déduit aisément du transfert parallèle. La largeur des voies de communication est arbitrairement limitée, mais les tampons restent capables de contenir des messages entiers *(fig. 20)*. Ces tampons peuvent être réalisés de diverses façons, mais la plus simple reste sans doute d'en faire des registres à décalage *(fig. 19)*. Le mécanisme d'exclusion mutuelle arbitrant l'accès à ces tampons n'en est pas fondamentalement modifié. Par contre le registre de mémorisation du message ne peut plus être implémenté par des bascules RS, ce qui double virtuellement le coût des tampons par rapport au routage parallèle.

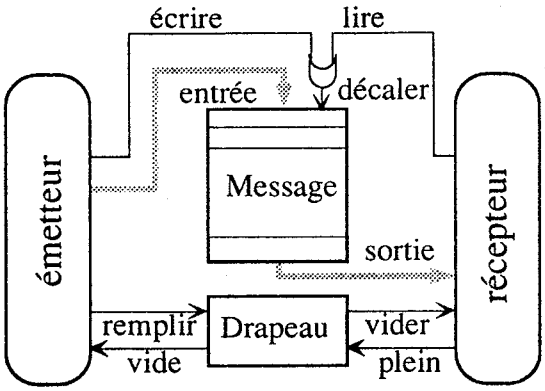


Fig. 19 — *Tampon série.*

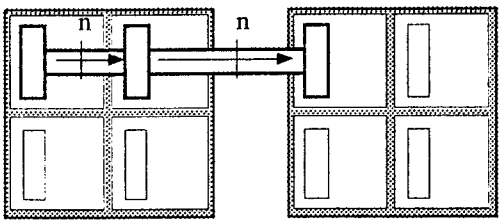


Fig. 20 — *Transfert série.*

Le transfert série peut mener *grosso modo* à trois types d'architectures (baptisées respectivement SERa, SERb, et SERc) qui diffèrent par la façon de prendre en compte les messages incidents.

Structure SERa : Les tampons Nord, Est, Ouest, Sud et le tampon de sortie de la cellule (OUT) sont examinés un par un successivement, grâce un algorithme de scrutation circulaire. Lorsqu'un tampon est plein, le routeur calcule et teste sa destination, puis le transfère.

```
cycle
  si le tampon courant est plein alors
     calculer la destination
     si la destination est vide alors
        transférer
  tampon suivant

fincyle
```

La figure 21 montre une organisation possible du chemin de données nécessaire pour implémenter cette méthode. Un registre à décalage circulaire permet de sélectionner le tampon courant. Les 4 premiers bits (la première composante de l'adresse relative), ou le premier flit si celui-ci est de taille supérieure à 4 bits, sont déposés sur le bus d'entrée E. Ils y sont récupérés par une circuiterie combinatoire chargée de déterminer le tampon de sortie. Si cette première composante de l'adresse relative de destination est nulle, la seconde composante est alors lue pour déterminer enfin la destination. Le transfert peut alors avoir lieu, dans la mesure où le tampon de destination est libre. La logique de décrémentation de composante d'adresse doit décrémenter les valeurs positives et incrémenter les valeurs négatives, ce qui revient à décrémenter la valeur absolue. Un codage signe - valeur absolue semble donc préférable à un codage en complément à deux.

Pour bien comprendre le fonctionnement de ce routeur, il faut remarquer que les messages stockés dans les tampons nord ou sud ont nécessairement, de par la stratégie d'acheminement X-Y, une composante horizontale d'adresse relative égale à zéro, que

nous décidons de ne pas mémoriser car elle gène l'accès à la composante verticale. De même, un message stocké dans un tampon IN ne contient pas du tout d'adresse.

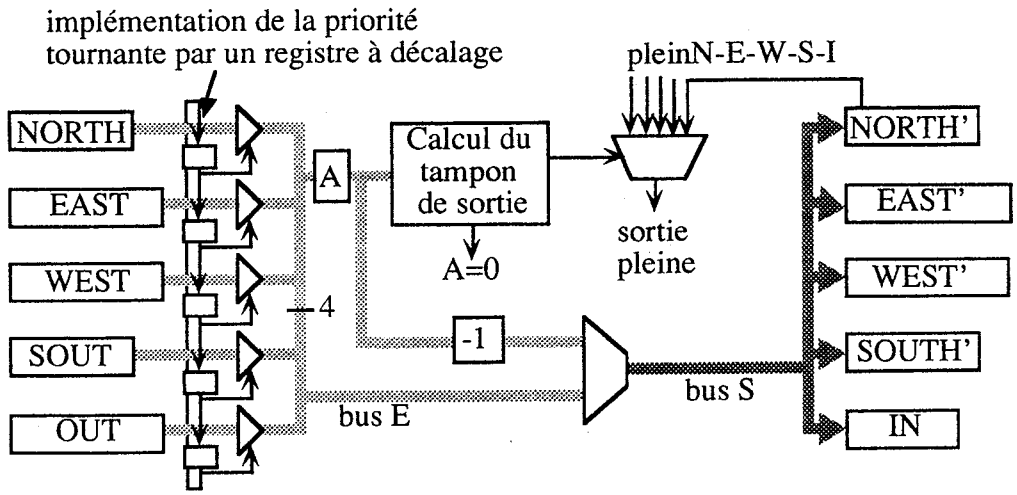


Fig. 21 — *Chemin de données du routage SERa.*

Cette option que nous avons prise permet de diminuer la taille moyenne des tampons, ainsi que le nombre de cycles nécessaires à la transmission d'un message. Par contre, un problème se pose lorsque la composante horizontale est consommée mais que le transfert échoue à cause d'une destination non disponible. Un indicateur est nécessaire pour discriminer les deux cas lors de l'examen d'un tampon. La figure 22 montre l'automate de contrôle résultant des précédentes remarques. L'indicateur en question est simplement nommé `xlu` pour plus de clarté, mais il faut noter qu'il y a un indicateur distinct pour chaque tampon horizontal ou OUT, les indicateurs des tampons verticaux pouvant être considérés comme collés à 1. Un autre indicateur constant lié à chaque tampon indique si le tampon est un tampon vertical : `ybuf`.

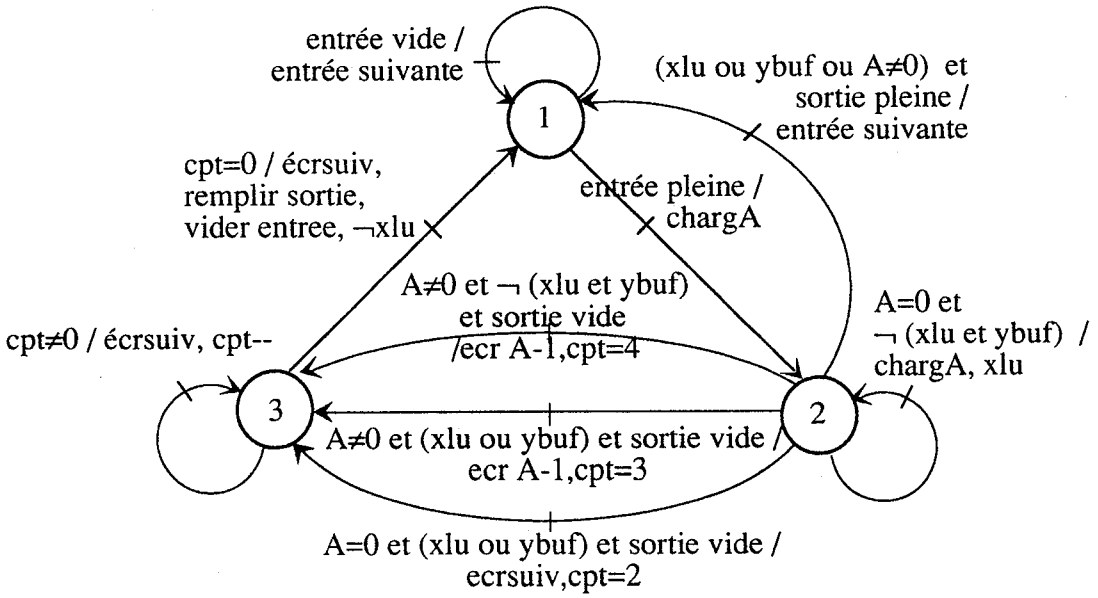


Fig. 22 — *Automate de la partie contrôle du routage SERa, flit de 4 bits.*

Cet automate minimise le nombre de cycles pris par le transfert en fonction de la décision de routage X→X, X→Y, Y→Y, X→IN ou Y→IN. Celui présenté en figure 21 correspond à des flits de 4 bits, taille qui est la plus pratique à manipuler puisqu'elle est égale à celle des composantes d'adresses relatives. Pour des flits de 1 ou 2 bits, le bus E reste de 4 bits, mais doit être multiplexé pour être connecté au bus S qui est, lui, de la largeur du flit. La taille de la partie contrôle s'en ressent fortement. Inversement, si la taille du flit est supérieure ou égale à 8 bits, ce calcul de destination à deux phases se contracte en une seule phase, rendant par là même caduc l'indicateur `xlu`. Dans ce cas, il est sans doute préférable de reporter sur des états supplémentaires l'opération de comptage des flits transmis.

Structure SERb : La seconde forme d'algorithme de routage tient compte du fait que la charge moyenne est relativement basse, rendant la probabilité d'avoir un tampon courant plein assez faible. Dans la forme SERa, un message arrivant dans un routeur totalement inactif va attendre uniformément de 0 à 4 cycles pour être pris en compte. Si la taille du flit est grande, ce délai sera important relativement au temps de transfert une fois le message pris en compte. D'où l'idée de "sauter" automatiquement les tampons vides pour ne traiter un par un que les tampons pleins.

```
cycle
  tampon = suivant plein
  si existe alors
     calculer la destination
     si la destination est vide
        transférer

fincycle
```

L'implémentation de cet algorithme est très proche de celle de l'algorithme SERa, tant au niveau de la partie opérative que de la partie contrôle. La différence la plus importante concerne le mécanisme de sélection du tampon courant, où le jeton circulant dans le registre à décalage ne désigne plus directement ce tampon : il désigne le *point de départ* de la prise en compte, et c'est une logique combinatoire en aval qui va choisir le tampon courant *(fig. 23)*.

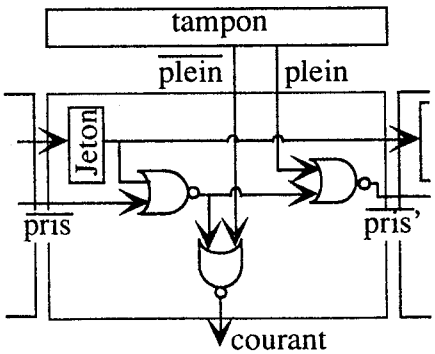


Fig. 23 — *Mécanisme de sélection du tampon courant dans SERb.*

<u>Structure SERc</u> : On peut aussi imaginer de transférer en parallèle les messages contenus dans les tampons d'entrée, lorsqu'il n'y a pas de conflit de destination :

```
en parallèle pour chaque tampon d'entrée :
  cycle
     si tampon plein alors
        calculer la destination
        si la destination est vide
           et un éventuel conflit gagné alors
           transférer
  fincycle
```

Alors que la forme SERb ne variait que de très peu de la forme SERa, cette dernière forme SERc repose sur des bases complètement différentes : nous avons un éclatement de l'automate de contrôle en autant d'automates élémentaires qu'il y a de tampons d'entrée, ainsi qu'une multiplication des voies reliant les tampons d'entrée aux tampons de sorties, de manière à supporter le parallélisme nouvellement créé. Ce parallélisme suppose aussi un arbitrage et un partage équitable des ressources uniques que sont ces tampons de sorties. Un circuit *full-custom* d'un routeur de ce type, a été réalisé par M. Karabernou au sein de notre équipe [KAR89], mais avec un transfert parallèle.

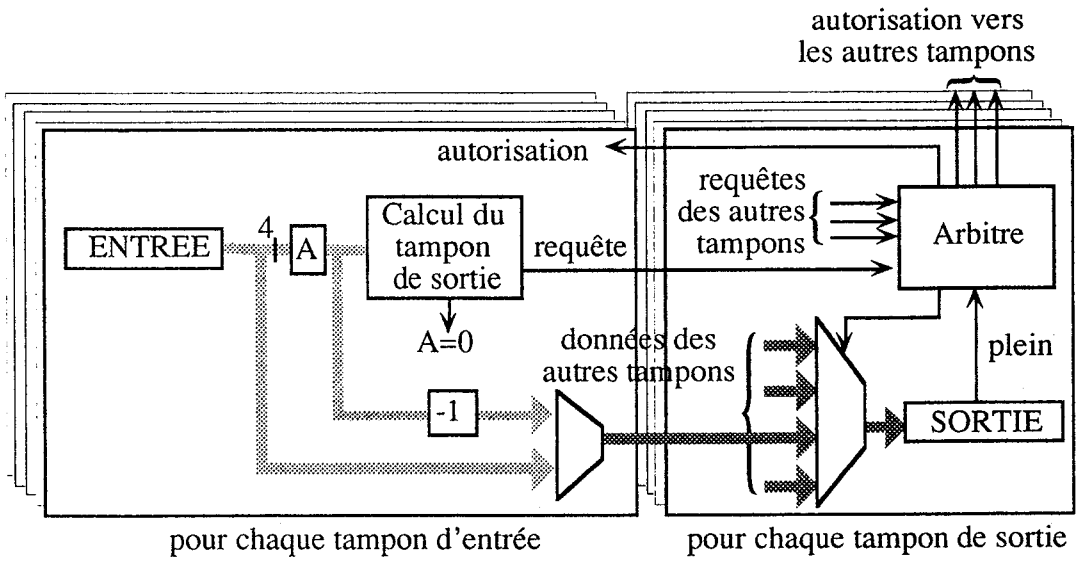


Fig. 24 — *Chemin de donnée pour le transfert SERc.*

La figure 24 montre ce que pourrait être le chemin de données associé à cette méthode de transfert : associée à chaque tampon d'entrée, nous trouvons une logique de calcul de la sortie appropriée et de décrémentation de l'adresse, similaire à celle de SERa et b (le mécanisme de sélection du tampon courant disparaît). Associé à chaque tampon de sortie, nous trouvons un arbitre chargé de recueillir les requêtes des tampons d'entrée. La discipline de service doit au moins assurer une absence de famine ; celle-ci

peut être réalisée par un système de sélection de même structure que le mécanisme mis en œuvre dans la méthode SERb pour choisir le tampon courant. Cet arbitre commande un multiplexeur chargé de sélectionner la voie correspondant au transfert autorisé.

Le nombre d'entrées de l'arbitre et du multiplexeur est au plus égal à 4 (et non à 5 car il n'y a pas de possibilité de rebouclage OUT→IN ni D→D quelque soit la direction D), mais comme le montre la figure 25, ce nombre peut tomber à 2 pour les tampons verticaux de par la stratégie X-Y.

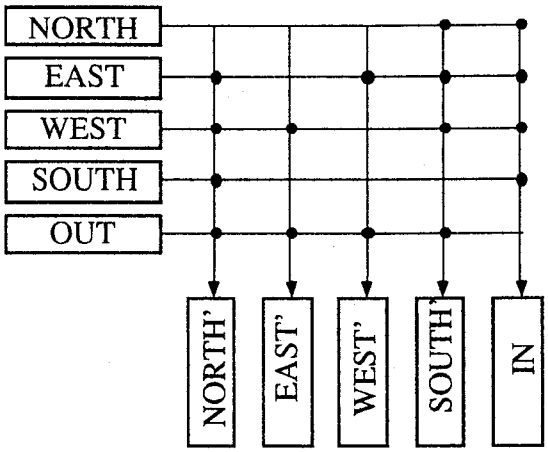


Fig. 25 — *Liaisons entre tampons d'entrée et de sortie.*

Il reviendra au concepteur la décision d'exploiter ou de ne pas exploiter cette caractéristique, selon qu'il cherche la simplicité de réalisation ou une utilisation optimale du silicium.

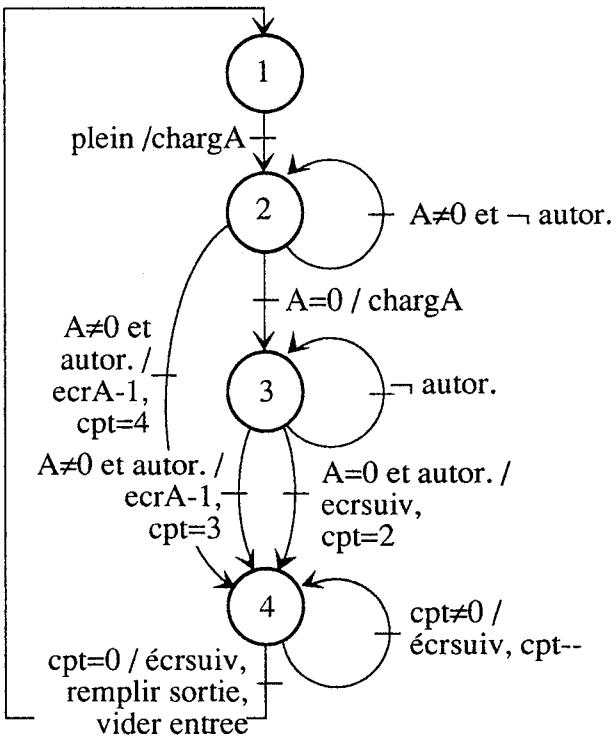


Fig. 26 — *Automate de contrôle pour le transfert SERc (tampon horizontal, flit<8 bits)*

L'automate de contrôle lié à chaque tampon d'entrée est une version simplifiée de l'automate de SERa : disparaissent la boucle de scrutation ainsi que l'indicateur xlu, remplacé par un état supplémentaire (n°2). L'automate présenté figure 26 correspond à un tampon E, W ou OUT ; pour les tampons verticaux, l'état n°2 disparaît de par l'absence de composante horizontale dans l'adresse relative du message.

d) Multiplexage aux frontières de boîtiers.

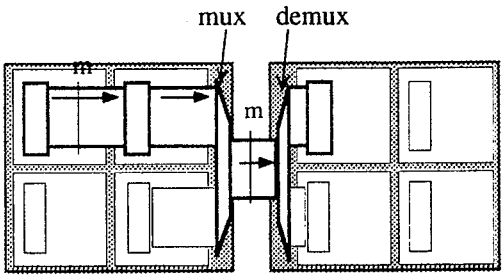


Fig. 27 — *Multiplexage aux frontières.*

Le multiplexage des voies de communication en sortie de boîtier est un autre moyen de diminuer le nombre de broches. Toutes les voies entre un boîtier et un autre sont regroupées sur une seule *(fig. 27)*, ce qui nous donne 4 multiplexeurs c→1 et 4 démultiplexeurs 1→c par boîtier. Le multiplexage peut être limité arbitrairement dans son degré. Pour un multiplexage de degré fixé d, chaque boîtier comprendra 4.c/d multiplexeurs d→1 et 4.c/d démultiplexeurs 1→d . Ce mécanisme de multiplexage peut être adjoint à toutes les variantes de routage série (de même qu'à celles de routage *wormhole*) ; les nouvelles formes obtenues seront désignée SER[ab ou c]-MUX.

Nous utilisons une connexion physique dans chaque sens. Il est en théorie possible d'aller encore plus loin en utilisant des connexions bidirectionnelles, mais le contrôle du multiplexeur serait alors partagé entre deux circuits, alors que des voies unidirectionnelles permettent une décision locale.

En ce qui concerne l'implémentation, ce multiplexeur induit un état de réquisition du multiplexeur avant de pouvoir accéder à un tampon de sortie. Il faut bien admettre que cette technique détruit considérablement la régularité du réseau :

- les cellules en contact avec le bord d'un boîtier sont différentes des autres
- les tampons d'entrée d'une cellule du bord ne sont pas homogènes, selon leur direction et la position exacte de la cellule, ils peuvent ou non être en relation avec un multiplexeur
- il faut examiner l'adresse d'un message pour savoir définitivement si oui ou non il faudra réquisitionner un multiplexeur pour le transférer.
- il y a adjonction dans le boîtier d'éléments autres que des cellules.

L'énorme avantage du multiplexage aux frontières est d'apporter une relative stabilité du nombre de broches des circuits vis-à-vis du degré d'intégration. Les signaux de contrôle des démultiplexeurs sont les seuls qui dépendent du degré d'intégration ; à un circuit de n x n cellules correspondent 8.n ou 8 $\log_2$ n broches, selon que le codage de la voie courante est effectué 1 parmi n ou en base 2.

Notons aussi qu'à nombre de broches égal et pour une faible charge, le multiplexage apparaît comme plus intéressant que la sérialisation, car la probabilité de conflits d'accès aux multiplexeurs reste réduite, le réseau multiplexé se comportant alors comme un réseau non multiplexé (les flits étant de la même taille). Une transmission série ne peut tirer aucun bénéfice d'une charge faible. Mais cette probabilité va augmenter parallèlement au degré d'intégration, quoique que cet effet soit contrebalancé par une réduction de la proportion du nombre de connexions inter-circuits par rapport au nombre de connexions intra-circuit.

e) La capacité à exploiter au mieux les liaisons intra-circuit peut être introduite dans le schéma sériel par un autre biais. Il faut pour cela considérer que les liaisons intra-circuit sont tout de même nettement moins coûteuses que les liaisons inter-circuits (la densité de connexion est de plusieurs ordres de grandeur supérieure). Dès lors il ne devient pas aberrant d'imaginer une taille de flit différente pour chacun des types de connexions *(fig. 28)* : la communication sera parallèle (ou moins fortement sérialisée) à l'intérieur du circuit, et fortement sérialisée à l'extérieur en fonction des exigences du packaging.

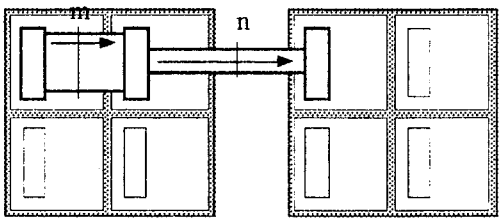


Fig. 28 — *sérialisation aux frontières.*

Cette variante, désignée SER[a,b ou c]-PAR, est déjà intéressante sur des circuits 2 x 2 (62% de connexions parallèles), mais elle ne prend sa véritable dimension que sur des tailles plus grandes (4 x 4 : 81 %, 8 x 8 : 90%). Elle peut en outre être combinée avec le multiplexage aux frontières (SER[a,b ou c]-PAR-MUX), au risque d'induire des différences considérables de performances entre les connexions. Le réseau ne va t-il pas alors fonctionner à la vitesse des connexions les plus lentes, la parallélisation interne ne jouant que comme un réducteur de contention ? Pour répondre à cette question, il faut distinguer deux cas. Si la distance moyenne des messages est longue, le délai d'acheminement (somme des délais élémentaires de transfert) va comprendre des transferts courts comme des transferts longs et nous aurons une sorte d'égalisation. Si les distances sont courtes, la réponse sera liée au schéma de dépendance de

communication de l'application. Certaines applications ont un couplage fort entre cellules, pour d'autres ce couplage est plus lâche, et l'incidence des différences de vitesse jouera moins.

D'un point de vue implémentation, nous retrouvons les désagréments relatifs à la rupture de la régularité que nous avons relevés pour le multiplexage : variabilité dynamique dans le contrôle des tampons, hétérogénéité de la conception. Ici s'ajoute une autre difficulté liée à la différence des tailles de flits : si nous continuons à voir un tampon comme un registre à décalage, les tampons pourront avoir des mots de tailles mixtes. La taille d'entrée est stable quoique non homogène, mais la taille de sortie est variable dynamiquement. En ce qui nous concerne, nous envisageons plutôt une parallélisation totale, ce qui ramène un tel registre à un registre à décalage pourvu d'un port de lecture parallèle.

2.2.2. Prévention d'interblocage.

Il existe deux propriétés structurelles auxquelles on accorde traditionnellement de l'importance en matière de réseau d'interconnexion. La première est absolument fondamentale, elle concerne la possibilité de prouver l'absence d'interblocage.

Cette propriété est assurée dans toute la famille des techniques de transfert sériel de par la méthode routage X-Y [SEI84]. Rappelons simplement qu'elle se prouve en démontrant qu'il n'existe pas de cycles dans le graphe de dépendance du réseau [DAL87a]. Ce graphe a pour sommet les liens d'interconnexion (les tampons) et pour arcs les possibilités légales de transfert entre liens. La figure 29 montre la topologie du réseau et le graphe de dépendance associé sous l'hypothèse d'un routage X-Y.

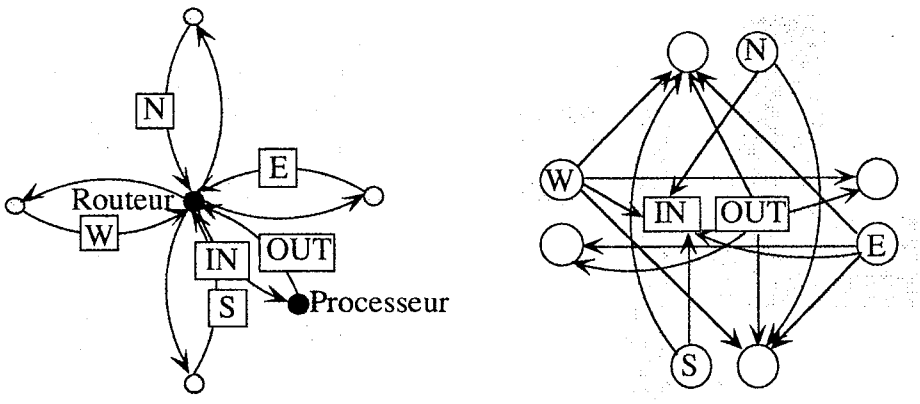


Fig. 29 — *Interconnexion et graphe de dépendance associé.*

Il est facile de voir que ce graphe ne comporte pas de cycles. En effet, si nous notons N, E, W, S, IN et OUT les directions des tampons, la stratégie X-Y induit que toute série de dépendance puisse s'écrire sous la forme :

$$[OUT] \rightarrow (\{W\}+\{E\}) \rightarrow (\{N\}+\{S\}) \rightarrow [IN].$$

Inversement, cette expression recouvre uniquement des séquences de dépendances valides, mise à part OUT→IN que nous interdisons.

La nature orthogonale non torique du réseau[1] impose que pour toute séquence cyclique la propriété nb(W)=nb(E) et nb(N)=nb(S), qui n'est compatible avec l'expression générale précédente que pour nb(W)=nb(E)=nb(N)=nb(S)=0. La seule séquence serait OUT→IN que nous venons d'exclure et qui d'ailleurs ne constitue pas un cycle.

Notons tout de même que nous ne mentionnons pas la possibilité de dépendance IN→OUT qui traduirait le fait que pour retirer le message contenu dans IN, il faille pouvoir envoyer un message.(⇔ OUT libre). Une telle possibilité mettait en défaut notre démonstration en permettant de fermer un cycle. Un exemple de cycle mettant en cause deux cellules voisines 1 et 2 pourrait être :

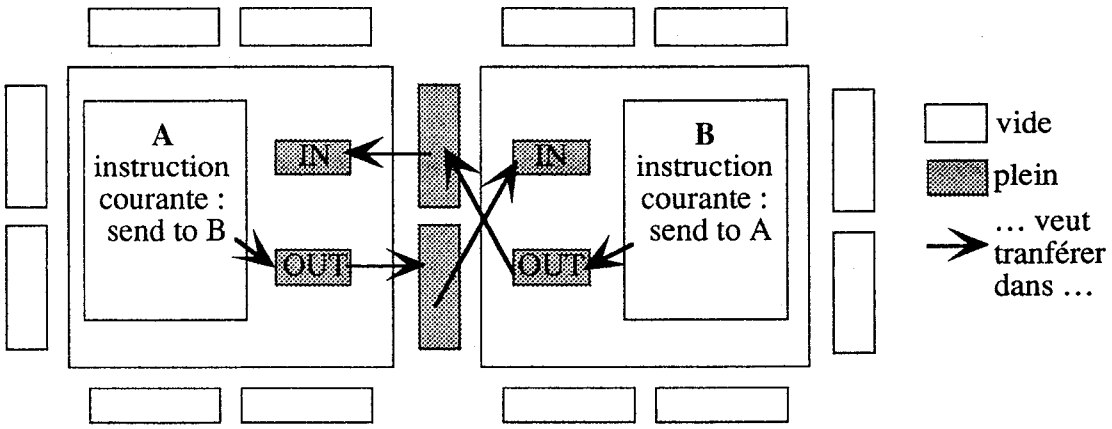


Fig. 30 — *Cycle OUT1→E→IN2→OUT2→W→IN1→OUT1.*

Une dépendance IN→OUT est plus difficile à supprimer qu'il n'y parait ; nous examinons plus en détail les implications de cette contrainte dans la partie consacrée au jeu d'instructions et à la structure interne de la partie traitement.

La sérialisation ne change pas la démonstration car le graphe de dépendance reste inchangé. Le multiplexage aux frontières ne la change pas non plus dans la mesure où tout transfert commencé peut être terminé. Il s'en suit que dans un hypothétique état d'interblocage, les transferts en cours se termineraient, ramenant le réseau dans un état où aucun message ne se trouverait à cheval sur deux tampons et où tous les multiplexeurs seraient libres, c'est à dire un état indiscernable d'un réseau non multiplexé, dont nous avons vu qu'il ne comporte pas d'état d'interblocage.

---

[1] dans une structure torique, la possibilité de "faire le tour du réseau" crée une possibilité d'interblocage nécessitant l'introduction de mécanismes *ad-hoc* d'évitement tels que la notion de *canaux virtuels* [DAL] dont le rôle est d'ouvrir les cycles du graphe physique en mappant un graphe logique exempt de cycle.

2.2.3. Prévention de famine.

L'autre caractéristique importante qu'un mécanisme de communication peut offrir est d'assurer l'absence de famine, et plus généralement l'équité de service. Dans notre cas, il s'agit s'assurer que les requêtes concernant l'accès à un tampon de sortie seront honorées dans l'ordre où elles ont été émises.

Les structures séquentielles de routeur SERa et SERb ne satisfont pas à cette exigence, il est très facile d'exhiber un cas de famine. Le tableau 4 montre un tel cas de conflit incorrectement géré entre deux tampons (Est et Ouest se disputant l'accès à la sortie Nord). L'entrée Est est servie, mais au moment où l'entrée Ouest est examinée par le routeur, la sortie Nord n'est pas encore libérée. Elle ne sera libérée que lorsque le routeur aura fait un tour complet de scrutation et recommencera à tester l'entrée Est, qui contient de nouveau un message à router vers le Nord, monopolisant ainsi cette voie de sortie.

Routeur	Entrée Est	Entrée Ouest	Sortie Nord
test Est	req. Nord	req. Nord	vide
transfert E dans N'	source	req. Nord	dest
test Nord	vide	req. Nord	plein
test Ouest	vide	req. Nord	plein
test Sud	req. Nord	req. Nord	plein
test Est	req. Nord	req. Nord	vide
transfert E dans N'	source	req. Nord	dest
...	...	...	...

Tableau 4 — *Cas de famine dans un routeur SERa.*

Ce motif peut être répété indéfiniment si la fréquence d'alimentation de l'entrée Est est inférieure à celle de consommation de la sortie Nord, de telle sorte que le tampon Ouest ne sera jamais routé malgré l'absence d'interblocage.

Heureusement, l'apparente simplicité de ce schéma cache des conditions très difficiles à réunir :

1) il faut qu'il n'y ait aucun couplage entre des messages en conflit, car la non-arrivée du message bloqué provoquerait progressivement une baisse d'activité entraînant le tarissement du flux responsable du blocage. Cette absence de couplage n'existe pas à l'intérieur d'une application, elle ne peut se rencontrer que si deux applications indépendantes s'exécutent simultanément sur le réseau. Inversement le couplage va avoir tendance à régulariser continuellement les flux par interaction entre application et communication, de sorte que même des famines partielles (forte iniquité de service continue) sont très peu probables et, de fait, n'ont jamais été rencontrées lors des simulations.

2) il faut que le flux bloquant présente une fréquence à la fois stable et très élevée, condition très difficile à obtenir dans un réseau dont la tendance principale est au désordre et à l'irrégularité.

Bien que cela ne puisse être prouvé (car nous ne connaissons pas par avance des applications qui seront écrites pour le réseau), il y a fort à parier que la probabilité de rencontrer une famine est de plusieurs ordres de grandeur inférieure à la probabilité de panne matérielle... Néanmoins, la simple existence d'une possibilité théorique de famine n'est pas satisfaisante. Nous pouvons la supprimer en agissant sur la condition n°1, c'est à dire en introduisant artificiellement un couplage de pure forme entre les applications indépendantes. Cette solution logicielle est pleinement satisfaisante, car pourquoi consacrer des dispositifs matériels coûteux, dupliqués sur toutes les cellules pour une éventualité correspondant à un cas particulier, là où quelques lignes de programmes très localisées suffisent ?

Le routage SERc assure par contre aisément une équité de service grâce à la logique d'arbitrage qui va ordonner la prise en compte des requêtes de telle sorte qu'une requête ne pourra être différée qu'un nombre fini de fois correspondant au nombre de tampons susceptibles d'accéder à l'arbitre (au plus quatre) moins 1. Ce mécanisme est similaire à celui utilisé dans le routeur SERb pour la sélection du tampon courant.

3.2.4. Evaluation.

La table ci-après (tab. 5) donne le *pourcentage de ralentissement* sur l'exécution du jeu de programme de test, en fonctions des paramètres de l'aiguilleur :

- le type de transfert : sérialisé (SER), sérialisé aux frontières (SER-PAR), sérialisé avec multiplexage aux frontières (SER-MUX), sérialisé aux frontières et multiplexé aux frontières (SER-PAR-MUX),
- 4 hypothèses de vitesse de fonctionnement de l'aiguilleur, mesurées en rapports entre fréquence du processeur et fréquence de l'aiguilleur (le rapport de fréquence qui semble le plus correct et égal à 2/1, c'est à dire un routeur cadencé à 20 MHz pour un processeur à 10 MHz),
- le type d'aiguilleur : séquentiel (SERa), séquentiel optimisé (SERb), parallèles (SERc, acheminement simultané de plusieurs messages),
- le degré de sérialisation : 1, 2, 4, 8, 12 ou 24 bits par flit (24 = transfert parallèle).

La taille du circuit est posée égale à 4x4 cellules. Chaque case de la table correspond à une configuration et contient le pourcentage de ralentissement que reflète la teinte de fond utilisée pour la case : les configurations teintées de blanc ou de gris clair peuvent être considérées comme acceptables (moins de 10% de ralentissement en moyenne).

Rapport fq routeur / fq processeur		SERa 24	SERa 12	SERa 8	SERa 4	SERa 2	SERa 1	SERb 24	SERb 12	SERb 8	SERb 4	SERb 2	SERb 1	SERc 24	SERc 12	SERc 8	SERc 4	SERc 2	SERc 1
SER	1/1	8,0	15	19	33	68	138	2,4	8,2	13	24	55	128	2,8	7,9	11	21	44	109
	2/1	2,8	5,5	6,8	12	21	48	1,0	2,8	3,8	8,6	17	43	1,0	2,5	3,3	8,2	14	36
	3/1	1,2	2,6	3,3	6,2	12	24	0,3	1,3	1,7	4,1	9,0	22	0,3	1,2	2,0	3,9	8,2	19
	4/1	1,0	1,7	2,2	3,9	6,9	16	0,05	1,0	1,0	2,7	6,2	14	0,05	1,0	1,0	2,5	4,6	12
SER-PAR	1/1		9,8	12	18	20	35		4,1	5,4	8,4	14	30		3,6	4,6	7,3	13	26
	2/1		3,4	4,7	6,6	7,2	13		1,5	1,6	2,4	4,0	10		1,6	1,3	2,3	3,3	9,1
	3/1		1,7	1,8	2,9	3,0	6,3		0,5	0,5	1,1	2,3	4,5		0,4	0,5	1,0	1,8	3,9
	4/1		0,9	1,5	2,0	2,3	3,7		0,2	0,2	0,6	1,4	2,3		0,2	0,2	0,6	1,2	2,3
SER-MUX	1/1	8,9	17	21	37	72	150	3,5	9,2	14	27	58	136	3,6	8,6	13	24	47	117
	2/1	3,1	6,7	7,1	13	23	53	1,0	3,0	4,6	8,5	19	45	0,9	2,6	5,1	8,3	17	38
	3/1	1,6	2,9	4,0	6,4	12	27	0,3	1,6	2,1	4,7	9,6	24	0,3	1,5	1,9	4,5	8,9	20
	4/1	0,9	1,9	2,4	4,0	7,9	17	0,04	0,9	1,4	2,9	6,4	15	0,05	1,0	1,2	2,7	5,8	13
SER-PAR -MUX	1/1		11	13	17	28	59		5,9	7,1	11,8	21	52		5,7	6,4	10,8	21	51
	2/1		4,1	4,3	5,6	9,7	18		1,5	1,8	2,8	7,0	16		1,4	2,1	3,1	6,6	16
	3/1		1,6	2,2	3,1	4,6	9,9		0,6	0,9	1,4	2,6	7,9		0,6	0,9	1,5	2,6	7,9
	4/1		1,1	1,7	1,7	2,6	5,5		0,2	0,4	0,8	1,7	4,6		0,2	0,3	0,8	1,6	4,3

Pourcentage de ralentissement par rapport à un routage idéal (délai d'acheminement nul)

<5%	<10%	<20%	<50%	≥50%

Tableau. 5 — *Performances des variantes du transfert série.*

Le paramètre de fréquence de fonctionnement est un paramètre technologique que nous nous contentons de subir, nous ne pouvons pas l'influencer autrement que par une conception soignée. La figure 31 montre sous forme de courbes l'évolution du ralentissement lorsqu'on fait varier la taille du flit, pour un rapport de fréquence 2/1 (routeur à 20MHz, processeur à 10 MHz).

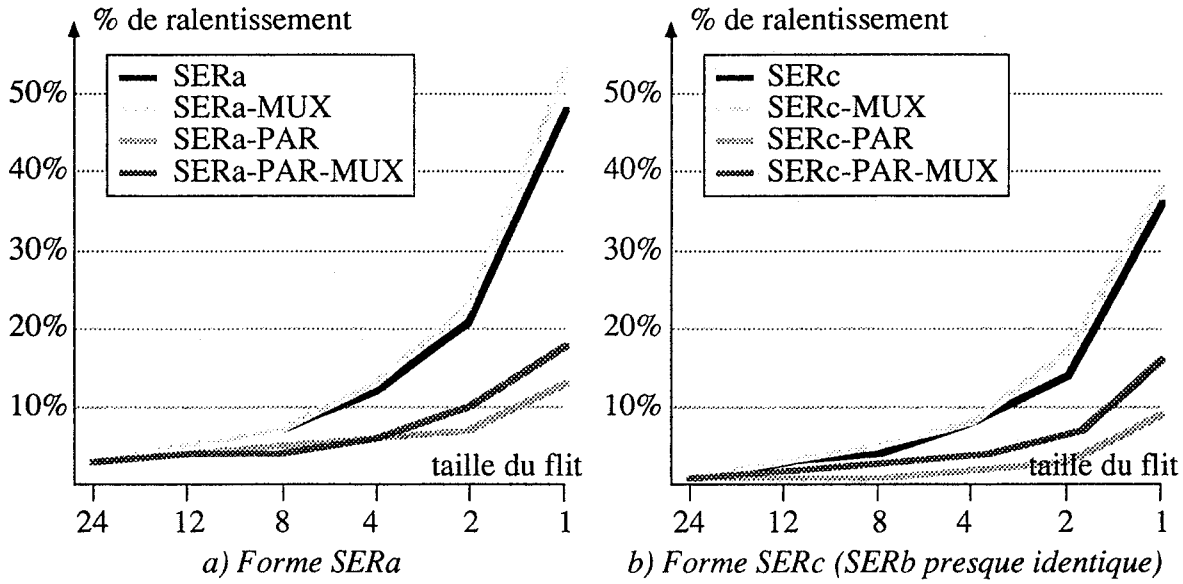


Fig. 31 — *Ralentissement en fonction de la taille du flit pour un rapport de fréquence = 2.*

Nous pouvons observer sur le tableau 5 et la figure 31, en mettant en regard sur le tableau 5 les transferts SER et SER-PAR et leurs homologues multiplexés respectifs, une dégradation de performance est relativement faible. L'influence du multiplexage se fait plus sentir lorsque la sérialisation n'est faite qu'aux frontières, car les conflits d'accès aux multiplexeurs se portent alors sur les liaisons qui sont déjà les plus critiques.

En comparant maintenant SER et SER-MUX et leurs homologues respectifs non sérialisés à l'intérieur du circuit, nous pouvons voir qu'un gain substantiel de performances est obtenu avec ces derniers.

Curieusement, la différence entre les diverses structures d'aiguilleur est peu sensible, nous pouvons rattacher ce fait à une charge trop faible pour que parallélisme des aiguilleurs parallèles (forme SERc) puisse s'exprimer. Mais une charge faible doit défavoriser la forme purement séquentielle (SERa), et c'est ce que nous retrouvons lorsque la taille des flits n'est pas trop petite : le temps de scrutation est alors important par rapport au temps de transfert.

La corrélation entre degré de sérialisation (taille du flit), fréquence de fonctionnement et performance semble assez facile à établir : le débit maximum réalisable sur un lien est fonction du produit de la fréquence de transfert par la taille du flit, ce qui se traduit graphiquement dans chaque sous-tableau du tableau 5 par une symétrie approximative selon une diagonale à 45°. Le temps perdu en scrutation, en

routage, en collisions, la différence éventuelle entre lien intra-chip et lien inter-chip viennent perturber la régularité de cette relation : à débit maximum équivalent, les aiguilleurs à fréquence de transfert élevée donnent les débits réels les plus grands.

Parmi les données obtenues par simulation figure aussi la latence unitaire moyenne observée. Nous pouvons nous attendre à ce que pour une certaine latence observée, le ralentissement soit le même que celui obtenu lors de l'étude des programmes lorsque nous fixions la latence arbitrairement.

Plus formellement, si L_o est la variable statistique qui décrit la latence observée, L_u la variable correspondant à une loi de Dirac centrée sur $\overline{L_o}$, R(L) le ralentissement induit par une certaine distribution de la latence L, est-ce que $R(L_o) = R(L_u)$?

Connaissant la fonction $R(L_u)$, nous pouvons voir comment se situent les points ($\overline{L_o}$, $R(L_o)$) par rapport à la courbe qui la décrit. Les figures 32 a-b montrent les courbes correspondant aux transferts SER et SER-PAR. Quelle que soit la technique de transfert, il y a un ralentissement supplémentaire par rapport à celui induit par une latence unitaire constante, que nous pouvons interpréter comme une conséquence de la dispersion de la latence observée. Certains chemins sont plus rapides que d'autres, et lorsqu'il y a conjugaison d'un chemin physique lent et d'une communication critique pour l'application, le ralentissement sera plus fort que celui explicable par la latence moyenne. Naturellement, cette baisse de performance devient très sensible pour les configurations déjà théoriquement insuffisantes par leur latence moyenne.

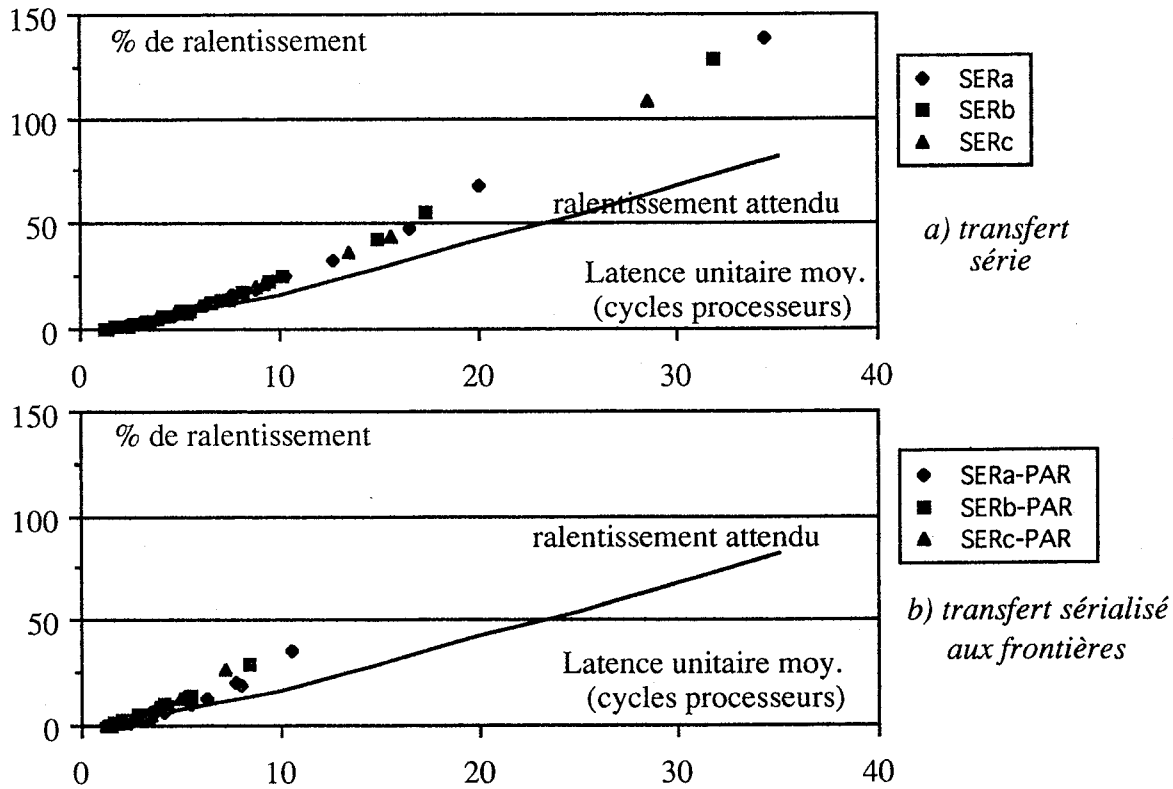


Fig. 32 — *comparaison entre ralentissement induit par une latence unitaire constante et une latence unitaire moyenne.*

Il ne faut toutefois pas conclure de ces considérations que les performances des routeurs à liens hétérogènes sont limitées par le débit de leur lien le plus lent, et donc qu'il n'est pas nécessaire d'utiliser des transferts parallèles pour les liaisons intra-chip. Le tableau 5 atteste bien du contraire : même si un chemin comporte des liaisons lentes, il contient toujours au moins une liaison rapide. Le temps d'acheminement est égal à la somme des temps de transfert sur chaque liaison du chemin, et les liaisons rapides peuvent toujours contribuer à atténuer l'effet des liaisons lentes.

Pour conforter la confiance que nous pouvons avoir dans l'analyse de la sensibilité des programmes, notons que lorsque le ralentissement attendu est "admissible" (<10%), le ralentissement effectivement observé l'est aussi.

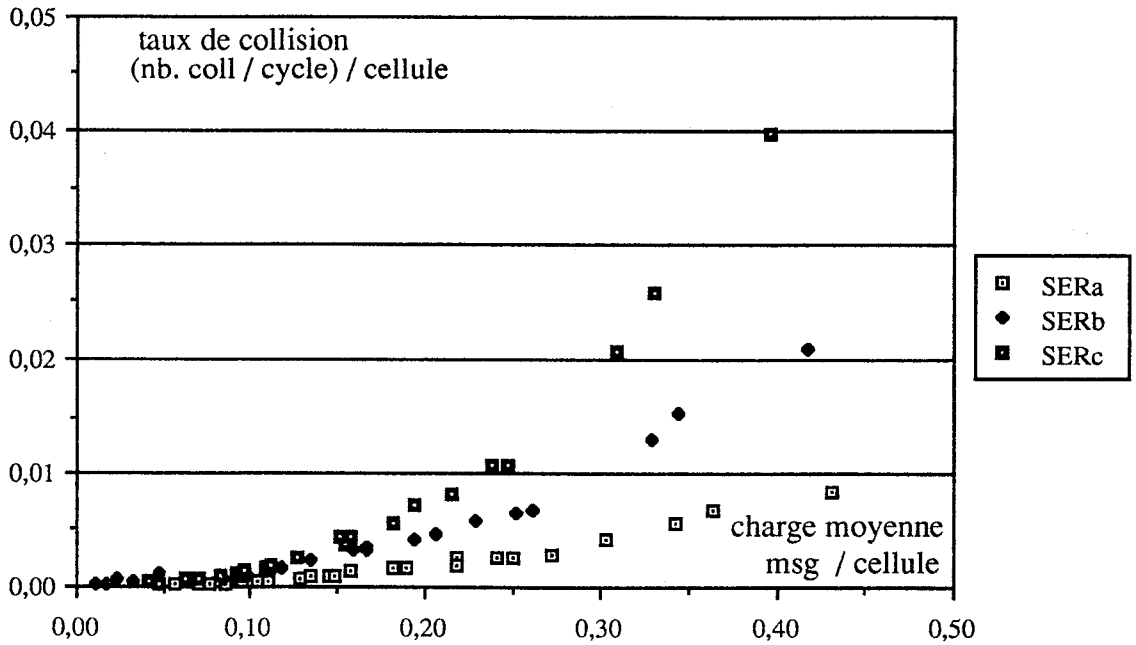


Fig. 33 — *Taux de collision en fonction de la charge.*

La figure 33 présente deux autres variables intéressantes à corréler. Il s'agit d'une part de la charge, prise comme étant le nombre moyen de messages présents dans une cellule, et le taux de collisions, pris comme le nombre moyen de collisions recensées par cycle processeur dans une cellule. Nous voyons que la charge moyenne n'est jamais très élevée par rapport à la charge maximale (6 messages par cellules), bien que les ralentissements correspondants soient eux très sensibles ; nous pouvons l'expliquer par un phénomène d'auto-régulation lié à la structure fortement couplée des processus : lorsque les communications se ralentissent, les processus se ralentissent eux aussi et produisent donc moins de messages. A charge égale, les différentes structures de routeur ont des taux de collisions différents : plus les routeurs examinent de messages en moyenne dans un même cycle, plus ils sont sujets à collisions. Toutefois, le taux de collision n'atteint jamais des valeurs telles qu'il faille considérer les phénomènes de contentions comme prépondérants.

2.3. Transfert *wormhole.*

Lorsque la taille des messages et la distance émetteur-récepteur sont importantes, le routage *wormhole* est indiscutablement supérieur au *store-and-forward* (transfert série). Malheureusement, nous ne sommes pas clairement dans ce cas : l'étude des applications a montré que la distance moyenne est très faible (elle est de toutes façons limitée à 7 par le codage de l'adresse relative des messages), et les messages sont de taille fixe égale à 24 bits. Les messages plus longs sont découpés en plusieurs messages et leur émission successive occasionne un recouvrement naturel entre leur acheminement, qui rejoint les effets d'un transfert *wormhole*, même s'il y a une certaine différence fonctionnelle. Que devient dans ce cas limite cette supériorité ?

2.3.1. Définition de variantes.

Avant de répondre à cette question, nous commencerons par distinguer quelques variantes du routage *wormhole*. Ce routage est intéressant par ses performances, mais aussi par la taille réduite des tampons qu'il nécessite, égale à celle du flit. Nous pouvons donc en envisager l'utilisation pour des raisons économiques ; le *virtual-cut-through* n'apporte rien sur ce plan, mais nous pouvons par contre étudier des capacités de mémorisation au niveau de chaque tampon qui soient intermédiaires entre un flit et un message, afin de trouver un compromis entre un trop grand risque de contention et un risque de sous-utilisation des ressources matérielles.

Ces capacités de mémorisation supérieures à un flit ne prennent tout leur intérêt que conjuguées à une amélioration assez évidente lorsqu'on se penche sur l'implémentation de ces protocoles. Vu par une cellule traversée, l'acheminement d'un message comprend trois types d'opérations :

- l'ouverture du chemin, avec consécutivement
  - l'analyse de l'adresse relative
  - le calcul de la voie de sortie
  - le test de disponibilité de cette voie de sortie
  - la mise à jour de l'adresse relative
  - le transfert de l'adresse relative
- le transfert du corps du message : équivalent à un nombre fixe de transfert de registre à registre
- la fermeture du chemin : c'est une opération simple qui peut se faire en recouvrement avec le transfert du dernier flit, donc en temps apparent nul, si les tampons sont implémentés sous forme de bascules maître-esclave, qui doit être reporté s'ils ne sont pas implémentés sous cette forme.

Il y a donc une énorme disparité de contenu, donc à priori de durée, entre le traitement de la tête et celui des flits suivants, une disparité qu'il est possible de mettre à profit : acheminer plusieurs flits du corps dans le laps de temps nécessaire au routage de

la tête. Mais si la queue du message avance plus vite que la tête, ne va t-elle pas vouloir la dépasser ? Pour bien comprendre la dynamique d'un tel système, examinons ce qui se passe lors d'un acheminement *wormhole* pur, puis avec cette modification.

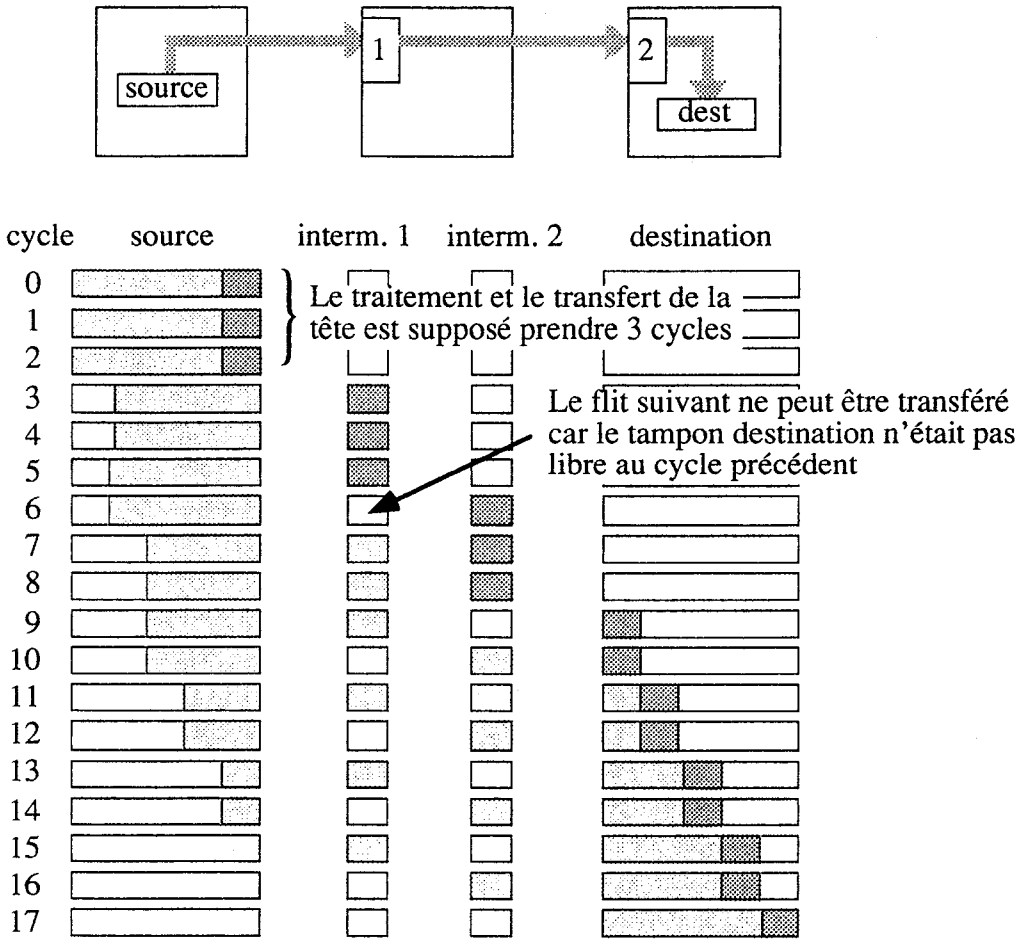


Fig. 34 —*Wormhole pur (dans ce diagramme, l'adresse est supposée entièrement contenue dans le premier flit).*

Contrairement à la figure 6 (§ A.3.2.1), nous prenons en compte l'hypothèse d'asynchronisme des routeurs : comme nous l'avons déjà souligné à propos du transfert parallèle, *un transfert ne peut se faire que dans un tampon préalablement vide, il n'est pas possible pour un routeur de savoir que le tampon destinataire sera vide au cycle suivant, et de faire avancer les messages ou même seulement les flits d'un message en rythme*[1]. Cet asynchronisme a plusieurs conséquences :

---

[1] Un système pour faire avancer un message "en rythme" une fois que le chemin a été ouvert pour la tête peut être envisagé pour des messages pas trop longs : il faut synchroniser temporairement la chaîne de routeurs impliqués dans le transfert, ce qui pose à la fois des problèmes électriques et des problèmes fonctionnels (un routeur peut être impliqué dans plusieurs messages simultanément). Le *wormhole* modifié représente une solution à la fois plus élégante et plus performante.

- Chaque routeur transmet les flits à cadence maximale, ce qui implique qu'un flit peut être transmis à chaque cycle de base et non seulement à chaque macro-cycle de routage de la tête. Il serait en effet inutile et artificiel de faire en sorte que le routeur s'en tienne aux macro-cycles pour le transfert du corps du message. Nous avons donc là déjà une optimisation par rapport au *wormhole* initial présenté § 3.2.1.
- Nous voyons apparaître des *trous* qui voyagent en sens inverse du message. A l'arrivée, lorsque la tête du message ne limite plus la vitesse de transfert, il y a un trou entre chaque flit. La *longueur apparente* du message, nombre de tampons occupés, est alors le double de la longueur logique.
- Bien que la figure 34 ne le montre pas pour rester lisible, des décalages de phase importants peuvent exister entre les routeurs, ce qui peut conduire à des petites variations temporelles.

Le *wormhole* "modifié" remplace les tampons de contenance égale à 1 flit par des structures FIFO, de manière à permettre le transfert, tant que la FIFO n'est pas pleine.

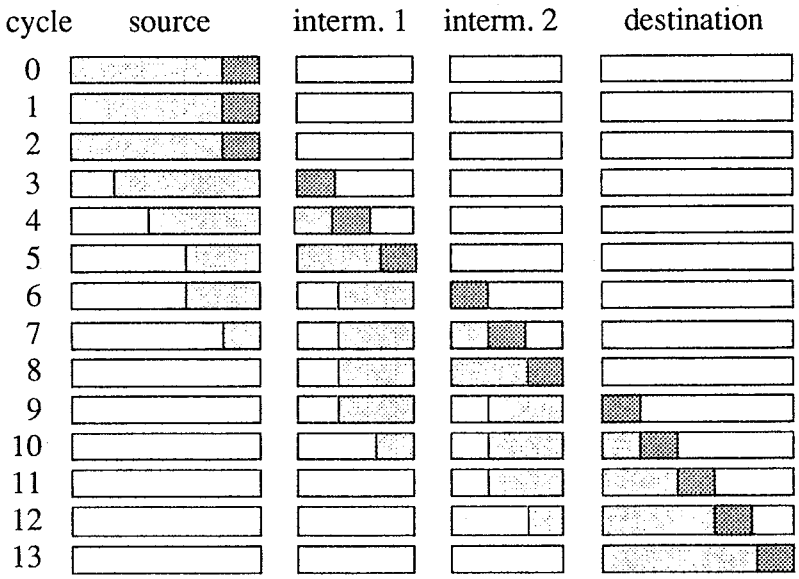


Fig. 35 — *Wormhole modifié.*

Avec le *wormhole* modifié, il y a toujours des trous, mais plus de tampons complètement vides. Soit R la durée d'un macro-cycle (il faudra R cycles pour router la tête), soit L la longueur d'un message en nombre de flits, D la distance en nombre de cellules (correspondant à $D+1$ tampons, destination comprise, source non comprise), le temps d'acheminement *T(wormhole initial)* est égal à :

$$T(\text{wormhole initial}) = R(D+1) + 2R(L-1) \qquad (1)$$

En *wormhole* pur, l'acheminement des flits du corps se fait sur la base du cycle, mais sans faire disparaître l'entrelacement entre trous et flits :

$$T(\text{wormhole pur}) = R(D+1) + 2(L-1) \qquad (2)$$

En *wormhole* modifié, l'acheminement se fait à vitesse optimale, car les tampons contenant le message ne sont jamais vides, et donc peuvent toujours fournir un flit à transférer :

$$T(wormhole\ modifié) = R(D+1) + L-1 \qquad (3)$$

Cette dernière expression nous permet de comprendre pourquoi il n'est pas utile, du point de vue du temps d'acheminement à vide (sans risque de collision), d'avoir des tampons de contenance supérieure à 2 flits : le temps de transfert du corps du message au niveau de la destination ne peut dépasser le débit maximal , c'est à dire n flits en n cycles ; il suffit alors d'avoir un flit du message en permanence par tampon pour alimenter le dernier lien, et donc d'avoir la place pour recevoir au cycle t le flit qui sera transmis au cycle $t+1$. Au delà d'une contenance de 2 flits, la longueur apparente du message va diminuer si R est suffisamment élevé et si la distance D est grande, mais le temps d'acheminement restera inchangé. Cette diminution de la longueur apparente ne joue pas à vide, mais en réduisant le nombre de tampon occupés par un message bloqué, elle peut avoir une influence bénéfique sur la contention. C'est pourquoi nous n'écarterons pas immédiatement des contenances supérieures à 2 flits.

La classe des routages de type *wormhole*, que nous appellerons tous *wormhole* par généralisation, présente donc deux paramètres qui sont la taille du flit, et la taille des tampons en nombre de flits. Nous ne ferons pas une étude au niveau de la taille du flit aussi large que celle faite pour le routage série, car des contraintes algorithmiques rendent certaines valeurs (4 bits et 8 bits) plus intéressante que d'autres :

- la méthode *wormhole* suppose la présence d'informations de routage en quantité suffisante au niveau du flit de tête, ce qui n'est possible qu'avec un flit capable de contenir une composante d'adresse relative (4 bits)[1] ;
- la longueur du message en nombre de flits doit être assez grande pour ne pas tomber dans une implémentation dégénérée où les spécificités du mode de commutation se dissoudraient complètement ;
- certaines valeurs sont maladroites car elles amènent à couper le message à de mauvais endroits (des flits de 6 bits scindent la seconde composante d'adresse).

Comme nous l'avons fait pour le transfert série, nous distinguerons trois formes d'implémentation : séquentielle (WORMa), séquentielle optimisée (WORMb) et parallèle (WORMc). Nous prendrons aussi en compte un éventuel multiplexage aux frontières. Par contre, la notion de parallélisme intra-circuit parait à priori peu cohérente avec la dynamique du *wormhole*, même s'il n'est pas impossible de l'adapter.

---

[1] Des flits de 1 ou deux bits sont utilisables, mais relèvent d'une structure très différente (opérateurs arithmétiques sériels). Nous n'avons pas eu le temps de les évaluer dans le contexte du réseau cellulaire et renvoyons le lecteur à [FLA87] pour une étude détaillée dans le cas général.

2.3.2. Absence d'interblocage.

La prévention d'interblocage pour le routage *wormhole* et ses dérivés est plus délicate que pour le routage série. Le routage suppose le respect de l'intégrité du message, car seule la tête contient les informations de destination. La tête ouvre le chemin, le dernier flit le referme, mais si des trous peuvent s'insérer entre les flits, les flits d'autres messages ne le peuvent pas. Si tel n'était pas le cas, les routeurs seraient tout à fait incapables de déterminer quel flit appartient à quel message. Une solution simple, mais fausse, consiste à faire en sorte qu'un routeur commençant à traiter un message ne puisse examiner un éventuel autre message venant d'une autre direction, que lorsqu'il a fini de transférer le premier. Cette discipline est plutôt funeste, comme le montre le cas trivial d'interblocage présenté ci-dessous.

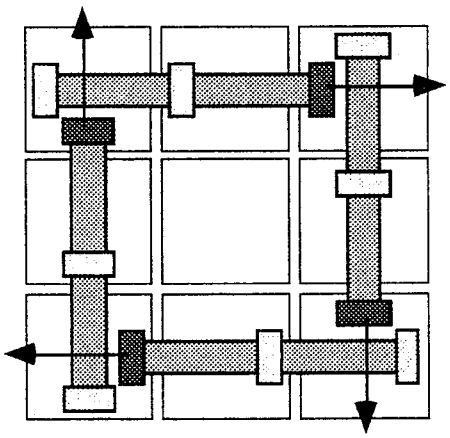


Fig. 36 — *Cas d'interblocage trivial pour le wormhole.*

Il faut donc supprimer les nouvelles dépendances introduites pour revenir à un graphe de dépendances sans cycles. Concrètement cela qui ce traduit par des messages qui peuvent se "couper" (l'intersection des chemins reste au plus *ponctuelle,* les messages ne se mélangent pas).

La combinaison du routage *wormhole* et du multiplexage aux frontières n'introduit pas de nouveau risque d'interblocage. Dans l'hypothèse où les multiplexeurs sont utilisés de manière exclusive par les chemins (un multiplexeur utilisé pour le chemin d'un message bloqué n'est pas ré-alloué), nous avons de nouvelles dépendances dans le graphe. Toute dépendance inter-circuits entre un tampon T et un tampon T' induit l'existence de dépendance de T vers chaque tampon de même direction et de même bord que T' *(fig. 37).*

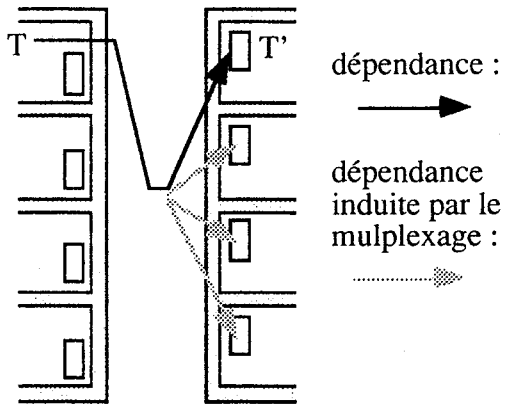


Fig. 37 — *Dépendances induites par le multiplexage.*

Ces nouvelles dépendances ne modifient pas l'expression générale des chaînes de dépendances (§ B.2.2.2) et ne peuvent donc être à l'origine de cycles.

2.3.3. Réalisation.

Le système de tampons présenté pour le routage série est globalement conservé mais, sauf pour le *wormhole* initial (tampons de 1 flit), il doit être adapté à une mémorisation sous forme FIFO *(fig. 38)*.

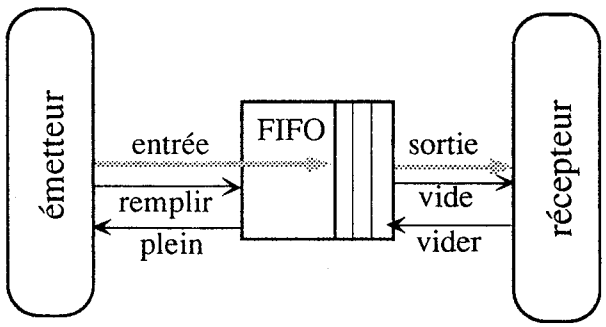


Fig. 38 — *Tampon multi-flits.*

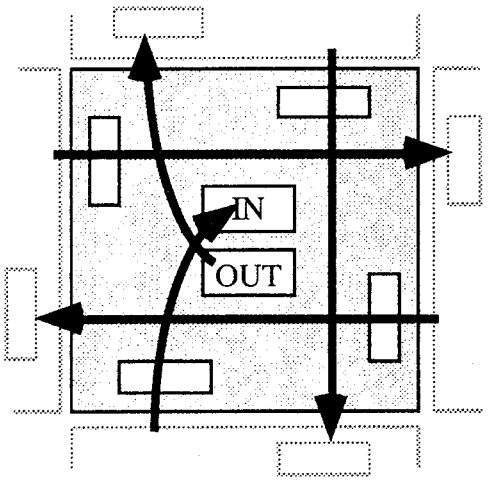


Fig. 39 — *Concurrence de transfert.*

En ce qui concerne la mécanique de contrôle du routage, la situation est relativement différente de celle rencontrée pour le routage série. Comme nous venons de le voir, le transfert d'un message ne peut plus être considéré comme une opération indivisible et, de ce fait même, les routeurs séquentiels (WORMa et b) peuvent être amenés à gérer plusieurs transferts en concurrence : jusqu'à 5 transferts peuvent avoir lieu simultanément *(fig. 39)*.

Nous étudierons donc d'abord la version parallèle WORMc, plus naturelle, puis nous verrons comment en dériver des versions séquentielles.

La structure du chemin de données est identique à celle présentée pour le routeur SERc *(fig. 24, p. 65)* : à chaque tampon d'entrée on associe un système de détermination du tampon de sortie et de décrémentation de l'adresse relative du message, et à chaque tampon de sortie une circuiterie d'arbitrage.

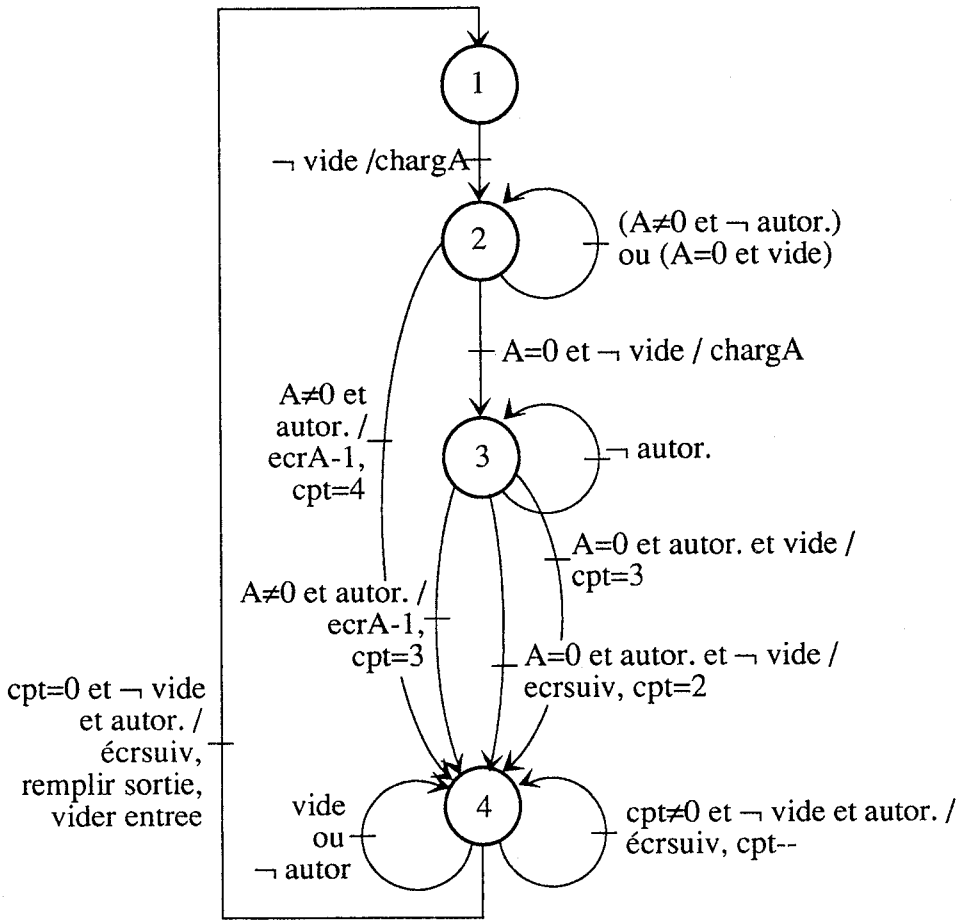


Fig. 40 — *Automate de contrôle de WORMc (tampon horizontal, flit de 4 bits).*

De même, l'automate de contrôle de la partie entrée est pratiquement identique *(fig. 40)*, il diffère sur les deux points suivants :

- la tentative de routage a lieu dès la présence de la tête du message (on teste si la FIFO du tampon est vide ou pas, au lieu de tester si le tampon est plein)
- les flits transférés sont comptés de la même façon qu'en transfert série, mais chaque transfert élémentaire est conditionné par le fait que la FIFO soit non vide (en transfert série, tous les flits sont disponibles par définition).

Globalement le routeur WORMc est donc très proche du routeur SERc. Revenons maintenant aux structures de routeur séquentielles. Avec un transfert série, tout transfert commencé peut être terminé, avec une exception pour le cas où le tranfert routage commande un changement de direction. La composante horizontale de l'adresse

relative, nulle, doit être escamotée, reportant d'un cycle le test de disponibilité du tampon de sortie, test qui peut échouer. Joint au cas où la composante horizontale est omise (tampons verticaux), le "contexte" associé à chaque tampon d'entrée ne contient que les informations nécessaires à déterminer si la composante d'adresse présente en tête du message est horizontale ou verticale. Concrètement, nous avons vu que deux indicateurs permettent de gérer les deux cas au niveau de l'automate :

- `ybuf` qui indique si le tampon est sur un lien vertical ou horizontal,
- `xlu` qui indique si la composante *dx* de l'adresse relative a été consommée (significative uniquement pour les tampons horizontaux).

Dans le routage *wormhole*, un transfert commencé n'est pas forcément terminé, ce qui revient en pratique à simuler une concurrence d'acheminement, un peu comme un système en temps partagé simule la concurrence des processus. Nous devons donc mémoriser un éventuel chemin ouvert, ainsi que le nombre de flits restant à transférer (pour savoir quand il est possible de fermer le chemin). Le contexte doit donc permettre de savoir :

```
un transfert est-il en cours ?
si oui :
  vers quel tampon ?
  combien reste t-il de flits à transférer ?
si non :
  quelle est la situation :
     - pas de message
     - une composante x en tête
     - une composante y en tête (x consommé ou omise)
     - une composante x consommée mais la composante y
       n'est pas encore reçue.
```

En somme, le contexte contient l'état de l'automate de la figure 40 le compteur de transmission et `ybuf`. Avec un contexte aussi gros, nous ne pouvons plus nous permettre de considérer que le contexte est obtenu par une petite logique combinatoire (ce que nous faisions en transfert série). Nous sommes amenés à envisager des méthodes de plus haut niveau. La plus simple consiste à avoir un automate central qui "interprète" les automates locaux. Le changement de tampon courant se fait alors à chaque cycle, ou à chaque échec.

Quels sont les avantages de cette structure par rapport à une structure parallèle ?

- la combinatoire de transition de chaque automate local est regroupé en une combinatoire unique (mais l'automate est plus complexe, car il doit gérer les tampons verticaux et horizontaux, qui se comportent un peu différemment)
- le chemin de transfert est unique (connectique interne réduite)

- la structure d'arbitrage au niveau des tampons de sortie est remplacée par un simple dispositif de verrouillage, mais la prévention de famine apportée par l'arbitre disparaît du même coup
- par contre le chemin critique sera vraisemblablement plus long, et donc la fréquence de fonctionnement plus réduite.

Ces avantages nous semblent bien minces pour que nous étudions les structures *wormhole* séquentielles autrement que par curiosité. Elles sont en quelque sorte "contre nature". Dans la suite, la forme WORMa dénotera une structure séquentielle où le changement de contexte se fait à chaque cycle, la forme WORMb une structure identique, mais optimisée par non-prise en compte des tampons vides. La forme WORMc est la structure parallèle étudiée au départ. La forme WORMc-MUX est obtenue par un multiplexage des liens inter-circuits.

Par rapport au transfert série, le transfert *wormhole* impose des contraintes supplémentaires pour le choix de la taille des flits : au moment du routage, le message n'est plus disponible en entier, seul le flit de tête est accessible. Celui-ci doit au moins pouvoir contenir une composante d'adresse relative, c'est à dire dans notre cas au moins 4 bits. Il s'en suit que seules deux tailles de flits sont pratiques : 4 (une composante par flit) et 8 (l'adresse complète dans le flit de tête). Des techniques d'arithmétique série permettent de s'affranchir de ces limites [FLA87], mais elles ne sont vraiment intéressantes qu'avec des adresses de longueur variable et ne se justifient pas dans notre cas.

2.3.4. Evaluation.

Le tableau 6 présente les performances (pourcentage de ralentissement) des quatre formes WORMa,b et c et WORMc-MUX, pour quatre hypothèses de fréquence de fonctionnement, pour des tailles de flit de 4 et de 8 bits, et pour diverses tailles de FIFO.

La forme purement séquentielle WORMa donne des résultats tout à fait catastrophiques, car avec une charge relativement faible (rarement plus d'un message par aiguilleur), un flit seulement est transmis tous les 5 cycles. L'option consistant à ne rendre la main que lorsque le message est bloqué serait sans doute plus intéressante, au moins dans le cas de tampons multi-flits (avec des tampons mono-flit, nous avons vu que le débit est limité à un flit tout les deux cycles). La forme optimisée WORMb permet un rendement bien supérieur de la structure séquentielle pure, rendant les performances largement suffisantes. Le passage à une structure parallèle n'occasionne pas des gains spectaculaires et ne se justifie que par une plus grande simplicité de conception.

Comme prévu, des tampons de deux flits améliorent beaucoup les performances. A taille de mémorisation égale, un routage avec des flits de 4 bits et des tampons de deux flits donne des performances de même niveau qu'un routeur wormhole pur avec des flits de 8 bits, pour des voies deux fois moins larges.

Rapport fq routeur / fq processeur	Taille des tampon (en nb de bits) 24	20	16	12	8	4
WORMa — WORM flit de 4 bits						
1/1	78	78	77	80	85	122
2/1	29	30	31	30	32	47
3/1	15	15	15	15	16	27
4/1	8,9	9,9	9,8	9,7	10	18
WORMa — WORM flit de 8 bits						
1/1	37		42		57	
2/1	13		15		22	
3/1	7,0		8,0		10	
4/1	4,0		5,3		8	
WORMb — WORM-MUX flit de 4 bits	24	20	16	12	8	4
1/1	15	15	15	16	18	30
2/1	4,4	3,4	4,0	4,4	5,0	8,6
3/1	1,1	1,3	1,4	1,8	1,5	4,5
4/1	0,8	0,6	0,3	0,5	1,4	2,4
WORMb — WORM-MUX flit de 8 bits						
1/1	6,1		7,3		12	
2/1	1,5		1,8		2,9	
3/1	0,5		0,5		0,8	
4/1	0,1		0,1		0,3	
WORMc — WORM-MUX flit de 4 bits	24	20	16	12	8	4
1/1	7,9	8,0	8,9	8,2	8,9	19
2/1	2,1	2,6	2,6	2,5	2,6	6,3
3/1	1,4	1,4	1,4	1,4	0,9	2,6
4/1	0,8	0,8	0,8	0,8	0,9	1,4
WORMc — WORM-MUX flit de 8 bits						
1/1	3,9		4,1		7,8	
2/1	1,2		1,0		1,7	
3/1	0,3		0,3		1,5	
4/1	0,05		0,06		0,3	
WORMc — WORM-MUX flit de 4 bits	11	11	11	12	14	35
1/1	2,8	3,2	3,1	2,9	3,3,	9,6
2/1	1,6	1,4	1,1	0,9	1,4	4,2
3/1	1,0	0,5	0,8	1,0	0,9	2,6
4/1						
WORMc — WORM-MUX flit de 8 bits	4,9		5,6		11	
1/1	1,5		1,2		3,3	
2/1	0,4		0,8		0,8	
3/1	0,2		0,2		0,4	
4/1						

Pourcentage de ralentissement par rapport à un routage idéal (délai d'acheminement nul)

<5%	<10	<20	<50	≥50

Tableau 6. — *Performances des variantes du transfert wormhole.*

Des tampons de taille encore supérieure, en augmentant la capacité de mémorisation du réseau, sont censés réduite la fréquence des collisions et donc d'améliorer les performances. En fait, nous pouvons voir que ce phénomène reste d'ampleur très modeste, décelable uniquement pour le passage de 2 à 3 flits par tampon. Nous pouvons expliquer ce fait comme une conséquence de l'auto-régulation que nous avons signalée dans l'évaluation des routeurs série: lorsque la latence de communication augmente, les programmes se ralentissent et produisent moins de messages, ce qui fait que la charge ne peut devenir très élevée, de même que le taux de collisions. L'incidence d'un mécanisme de limitation du taux de collisions ne peut donc qu'être marginale.

Le multiplexage aux frontières, comme pour le transfert série, a une incidence très faible, la forme WORMc-MUX se positionnant comme un intermédiaire entre WORMb et c, toutes deux acceptables.

D'une manière générale, les routeurs *wormhole* ne se démarquent pas nettement des routeurs série, ni par leur complexité, ni par leurs performances.

2.4. Transfert par bus.

2.4.1. Présentation.

Comme nous l'avons vu, la communication par bus ne semble pas très adaptée à la réalisation d'architectures massivement parallèles. Toutefois, il serait prématuré de l'écarter sans vérifier expérimentalement son inadéquation. L'introduction de bus de communication peut se faire selon deux voies : le bus intra-circuit et la grille de bus.

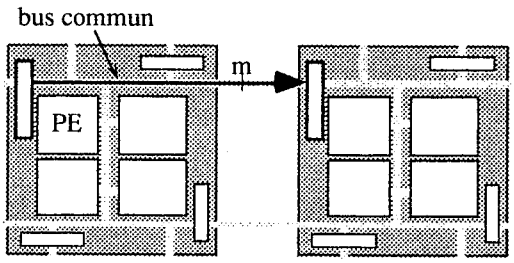


Fig. 41 — *Structure à bus intra-circuit.*

Le bus intra-circuit revient à construire une structure hybride qui se rapproche des machines à grappes de processeurs : chaque circuit contient une grappe de processeurs, mais contrairement au multiprocesseur Cm*, la communication entre grappes ne se fait pas par un bus externe, mais par un système à passage de messages.

Avec un bus intra-circuit *(fig. 41)*, les processeurs d'un même circuit peuvent communiquer directement et d'un circuit à l'autre, les messages peuvent traverser chaque circuit par un seul transfert élémentaire. Nous avons donc l'espoir d'un gain au

niveau de la vitesse de propagation, qui aura tendance à contrebalancer les effets de la contention induite par le partage des bus. Il nous faudra déterminer expérimentalement le nombre de processeurs correspondant au point où la perte de performance liée à la contention devient significative. Soulignons le fait que ce nombre ne constitue pas une limite à l'intégration d'un nombre quelconque de processeurs dans un circuit, car il est toujours possible de regrouper ceux-ci en plusieurs "grappes".

L'usage d'un bus devrait permettre des gains au niveau de l'implémentation, puisqu'il n'y a plus qu'un seul routeur par circuit, et qu'un multiplexage est automatiquement réalisé à la frontière du circuit.

Le routeur devient une ressource partagée, mais il est possible d'ajouter d'autres ressources partagées sur le bus, telles que de la mémoire RAM ou un petit co-processeur arithmétique (on rejoint alors la notion d'objet expert de Dally [DAL86b]). Nous reviendrons dans le chapitre consacré à la programmation sur l'intérêt de ces "aménagements" de la structure cellulaire de départ.

2.4.2. Réalisation.

Le schéma à un seul bus est évidemment un schéma simplifié. La charge du routeur unique de la grappe étant plus lourde que celle d'un routeur ne gérant qu'une seule cellule, nous devons l'optimiser. En pratique, le "transfert" est plus qu'une simple recopie de registre à registre : il faut analyser l'adresse relative contenue dans le message, en déduire le tampon de sortie approprié, vérifier qu'il est vide, mettre à jour l'adresse relative conformément au mouvement effectué. Il est donc impossible de se contenter d'un seul bus. Trois architectures de chemin de données sont envisageables, qui doivent être articulées de façon cohérente avec une architecture temporelle. Un point de mémorisation centralisé du message avant ou après modification peut être introduit *(fig. 42a)*, le bus peut être coupé en deux *(fig. 42b)* comme cela a été fait pour le routage série, ou encore, la modification ne portant que sur l'adresse, la partie adresse du bus peut être la seule à être scindée *(fig. 42c)*.

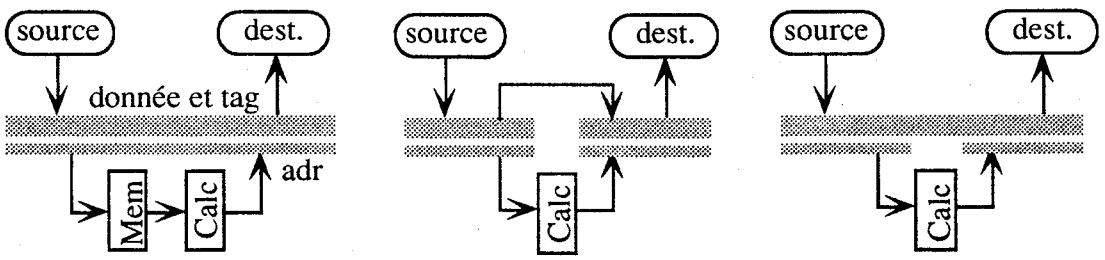


Fig. 42 - *a) architecture à un bus b) architecture à 2 bus c) architecture intermédiaire.*

Dans la première et la dernière de ces structures, un seul transfert peut avoir lieu à un instant donné. Etant donné le caractère de ressource critique du routeur, la solution à deux bus est donc plus avantageuse, car elle permet le découpage du chemin de données en plusieurs étages de *pipeline* (2 ou 3 étage, voir fig. 43).

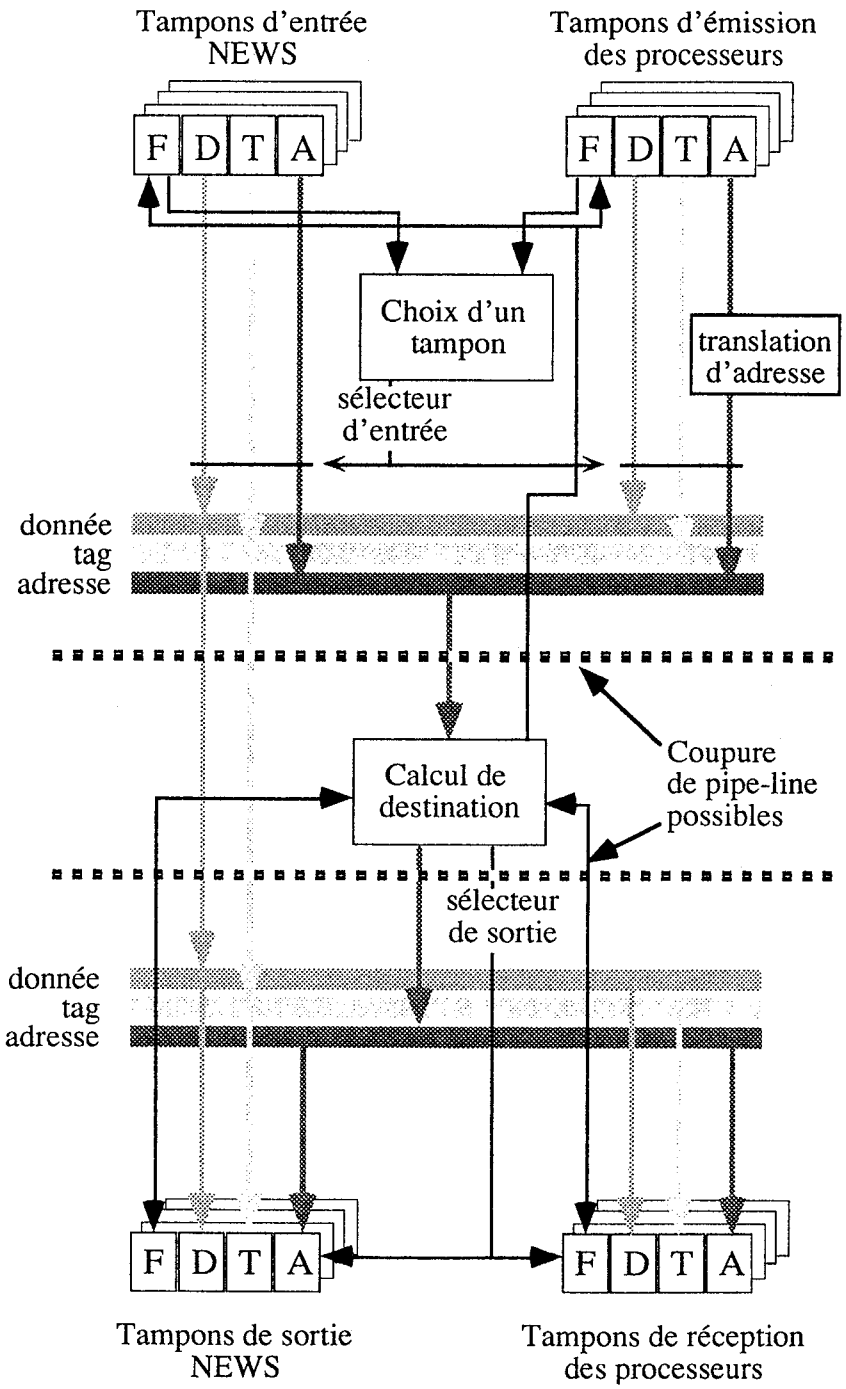


Fig. 43 — *Structure d'un routeur à 2 bus.*

Comme un routeur sert maintenant non plus un processeur unique, mais une grappe de processeurs, la fonction de modification de l'adresse relative introduite pour le transfert série n'est plus suffisante : elle conduirait à traiter indifféremment les messages sans prise en compte de leur origine, ce qui est bien sûr incorrect. La fonction de "décrémentation" doit intégrer la position du processeur d'origine, pour que l'adresse devienne non plus relative à l'émetteur, mais à un point unique de la sous-grille contenue dans le circuit, par exemple le coin Nord-Ouest. Le routeur sera conventionnellement placé à cet endroit. Tout se passe comme si le message était

transféré dans un premier temps de l'émetteur au routeur par translation d'adresse, puis du routeur à la destination. La figure 44 montre le système de référence qui en résulte :

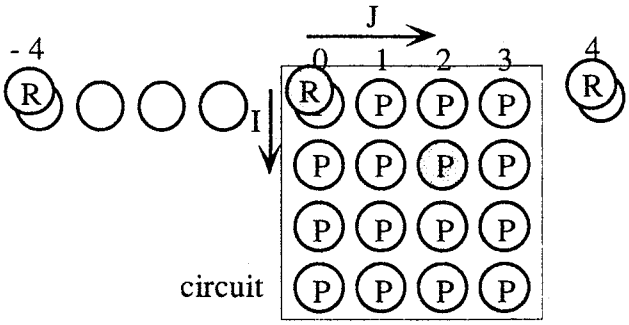


Fig. 44 — *Système de coordonnées.*

Cette opération de translation peut occasionner un dépassement de la magnitude permise par les champs d'adresse relative (deux fois 4 bits) que nous devrons prévenir au niveau du routeur par une augmentation de cette magnitude. Comme le montre le tableau 7, deux fois 5 bits suffisent dans l'état actuel de la technologie.

taille du circuit	magnitude nécessaire	nombre de bits
2 x 2	$[-7,+8]^2$	5 + 5 bits
4 x 4	$[-7,+10]^2$	5 + 5 bits
8 x 8	$[-7,+14]^2$	5 + 5 bits

Tableau 7 — *Taille des adresses translatées.*

Cette translation peut se faire à la sortie des processeurs (chemin d'adresse de 10 bits au lieu de 8, combinatoire de translation dupliquée, mais simplifiée par la présence d'un opérande constant) ou au niveau du routeur (I et J peuvent être générés par la logique de choix du tampon courant).

$\delta j < 0$	® δi	sortie = ouest $\delta j' = \delta j + c$; $\delta i' = \delta i$
$\delta j \geq c$	® δi	sortie = est $\delta j' = \delta j - c$; $\delta i' = \delta i$
$\delta j \in [0,c[$	$\delta i < 0$	sortie = nord $\delta i' = \delta i + c$; $\delta j' = \delta j$
$\delta j \in [0,c[$	$\delta i \geq c$	sortie = sud $\delta i' = \delta i - c$; $\delta j' = \delta j$
$\delta j \in [0,c[$	$\delta i \in [0,c[$	sortie = $P(\delta i,\delta j)$
adresse relative = $(\delta i,\delta j)$ adresse modifiée = $(\delta i',\delta j')$ c est la taille d'un côté du circuit (nombre de PE = c^2)		

Tableau 8 — *Fonction de choix de la sortie et de modification de l'adresse relative.*

Le routeur, à partir de l'adresse translatée, détermine si le message doit être envoyé à l'extérieur (voies NEWS) et l'adresse décrémentée en conséquence, ou s'il doit le ranger dans le tampon d'entrée d'un processeur du même circuit, conformément à la table 8.

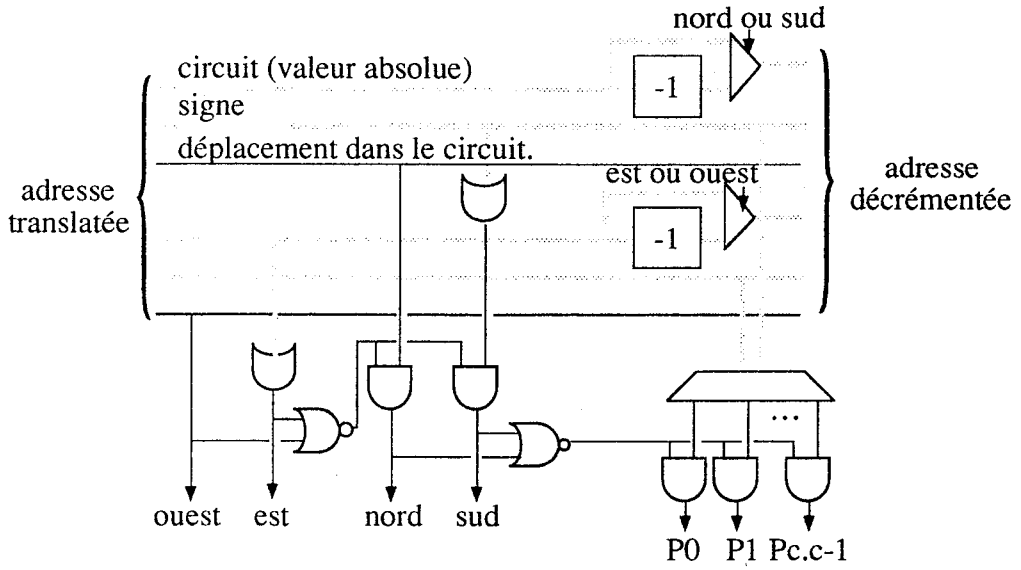


Fig. 46 — *Calcul de destination et décrémentation d'adresse.*

Reste à définir la stratégie de choix du tampon courant. Etant donné le nombre potentiellement grand de processeurs dans un circuit, une stratégie de scrutation naïve sera sans doute trop lente : le temps pour qu'un message en sortance d'un processeur soit pris en compte dans un routeur inactif serait beaucoup trop lent. Une stratégie où seuls des tampons pleins sont soumis à l'automate (comme celle utilisée dans SERb) est préférable, mais si le nombre de processeurs est trop important, on se heurte alors au problème d'un chemin critique dans la logique de sélection qui devient une contrainte. On peut dans ce cas imaginer d'utiliser le même système, mais de façon hiérarchique : chaque ligne de processeur sélectionne un de ses membres, puis on choisit parmi l'une des lignes qui possède du message à traiter. On a alors deux niveaux de jetons pour établir la priorité : des jetons au niveau de chaque ligne, et un jeton de sélection de ligne.

Le problème n'est pas complètement résolu pour autant. Il existe une différence fondamentale entre les *pipelines* mis en œuvre classiquement et celui que nous proposons ici : une opération de transfert peut échouer. Nous ne pouvons donc libérer le tampon courant sitôt que le message qu'il contient est injecté dans le *pipeline*, car un échec laisserait le routeur avec un message sur les bras dont il ne saurait que faire. Il pourrait le remettre à sa place, mais celle-ci a pu être occupée entre-temps pour un nouveau message. Laisser le message dans le tampon jusqu'à ce qu'il soit complètement transmis n'est pas non plus totalement satisfaisant, notamment dans le cas où le routeur n'a que ce message à traiter : il est potentiellement réinjectable au

cycle suivant, à tâche pour le routeur de supprimer les doublons lorsqu'un transfert réussit. Deux autres tactiques plus simples existent :

- verrouiller les tampons contenant des messages déjà injectés pour éviter une éventuelle re-sélection, ce qui dans le cas d'un routeur peu chargé va introduire un délai minimum entre deux tentatives de transfert d'un même message égal à la profondeur du *pipeline*
- verrouiller de la même façon, mais ne pas faire avancer le *pipeline* lorsqu'il n'y a qu'un seul message à router.

Nous avons choisi la première de ces deux solutions pour nos évaluations, considérant comme trop faible la probabilité qu'un routeur soit aussi peu chargé.

2.4.3. Evaluation.

La structure à bus intra-chip et routeur unique que nous venons de présenter a été simulée dans plusieurs configurations différentes :

- pas de pipeline, 2 étages ou 3 étages,
- les quatre hypothèses de fréquence que nous avons utilisées plus haut
- trois hypothèses de tailles de chip (4, 16 et 64 cellules par chip), car ce paramètre est primordial pour ce type de routeur (il influe directement sur la charge imposée au routeur unique.

Rapport fq routeur / fq processeur	Taille du circuit (cellules) 4	16	64
1/1	1,9	4,8	52
2/1	1	0,4	9,1
3/1	0,6	0,5	2,3
4/1	0,5	0,5	0,5

BUS0 : pas de pipeline

	4	16	64
1/1	3,3	6,6	53
2/1	1,6	1,0	11
3/1	1,0	0,8	2,7
4/1	0,9	0,7	1,3

BUS1 : 2 étages de pipeline

	4	16	64
1/1	6,0	8,0	55
2/1	2,3	2,2	11
3/1	1,5	1,1	2,5
4/1	0,1	0,9	1,2

BUS2 : 3 étages de pipeline

Pourcentage de ralentissement par rapport à un routage idéal (délai d'acheminement nul)

<5%	<10	<20	<50	≥50

Tableau 9. — *Performances des structures à bus intra-chip.*

Le paramètre de fréquence de fonctionnement est délicat à manipuler : alors que les routeurs sériels et *wormhole* utilisaientt des chemins de données comparables, que l'on pouvait, en première analyse, supposer fonctionner aux mêmes fréquences, les routeurs à bus vont fonctionner à des fréquences dépendant directement du nombre d'étages de pipeline. Le chemin critique d'un routeur pipeline va être fixé par celui de l'étage le

plus long. L'"étage" d'un routeur sans le pipeline (BUS0) effectuant le travail des deux ou trois étages des routeurs BUS1 et BUS2, son chemin critique sera approximativement égal à la somme des chemins critiques de ces étages ; si les étages sont bien équilibrés (il est possible d'y parvenir), le chemin critique du routeur sans pipeline sera supérieur d'un facteur 2 à celui du routeur à 2 étages, d'un facteur 3 par rapport à un routeur à trois étages. Il faut donc comparer les performances du routeur sans pipeline pour un rapport de fréquence 1/1 avec les performances du routeur à deux étages pour un rapport 2/1 et les performances du routeur à trois étages pour un rapport 3/1. Ceci explique le décalage vertical des tableaux de ralentissement du tableau 9. Nous pouvons faire sur ce tableau les commentaires suivants :

- La technique du pipeline est très efficace sur ce problème : pour des chips de 64 cellules, on passe d'un ralentissement inacceptable (50%) avec le routeur BUS0 (rapport de fréquence 1/1) à un ralentissement quasi nul avec le routeur BUS2 (rapport de fréquence 3/1).
- Une anomalie peut être assez régulièrement notée : un routeur gérant 16 cellules apparaît comme plus efficace qu'un routeur ne gérant que 4 cellules. Pour la comprendre, il convient de se rappeler qu'un message est acheminé en un seul transfert, si l'émetteur et le récepteur sont dans le même chip, indépendamment de la distance qui les sépare. D'une manière générale, augmenter le nombre de cellules dans un chip tend à augmenter la charge du routeur, mais diminue le nombre de transferts pour chaque message, le meilleur routeur se effectuant un compromis entre ces deux paramètres.
- L'architecture à un routeur unique reste valable même pour 64 cellules par chip, ce qui représente le maximum des capacités d'intégration industrielles actuelles (1,5 millions de transistors).

3. Comparaison et conclusions.

L'étude de différents types de routeurs a été au départ motivée par la nécessité de garder un nombre de plots raisonnable en entrées/sorties de chaque boîtier. Nous pouvons maintenant comparer les différentes solutions menant à des nombres de broches par boîtier du même ordre de grandeur. Commençons par *packaging* "riche", comprenant approximativement entre 200 et 250 broches, qui correspond soit à un transfert parallèle multiplexé aux frontières, soit à une sérialisation par flit de 8 bits. La figure 47 résume les ralentissements induits correspondants, pour diverses hypothèses de fréquence de fonctionnement. Le nombre de cellules par boîtier est supposé être de 4 x 4 cellules. Pour fixer les idées, la fréquence de fonctionnement réalisable est généralement de 20 MHz en technologie 1.2µ, soit un rapport de fréquence de 2/1. Dans cette hypothèse, même si l'on fixe la limite de ralentissement acceptée à 5%, pratiquement toutes les solutions peuvent convenir.

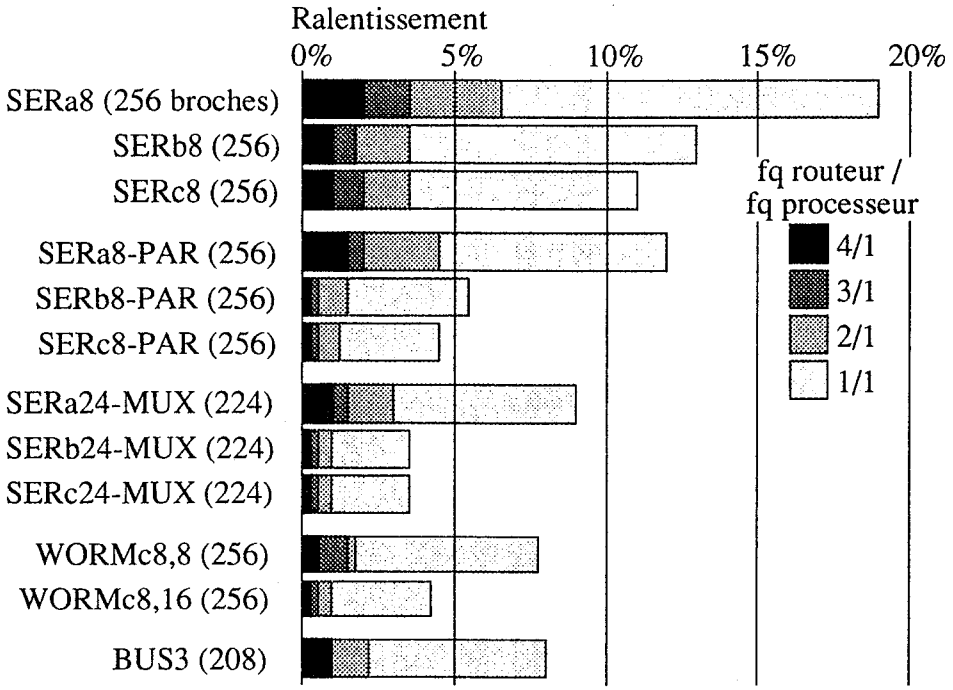


Fig. 47. — *Comparaison des routeurs possibles pour un compte de broches 200 → 250.*

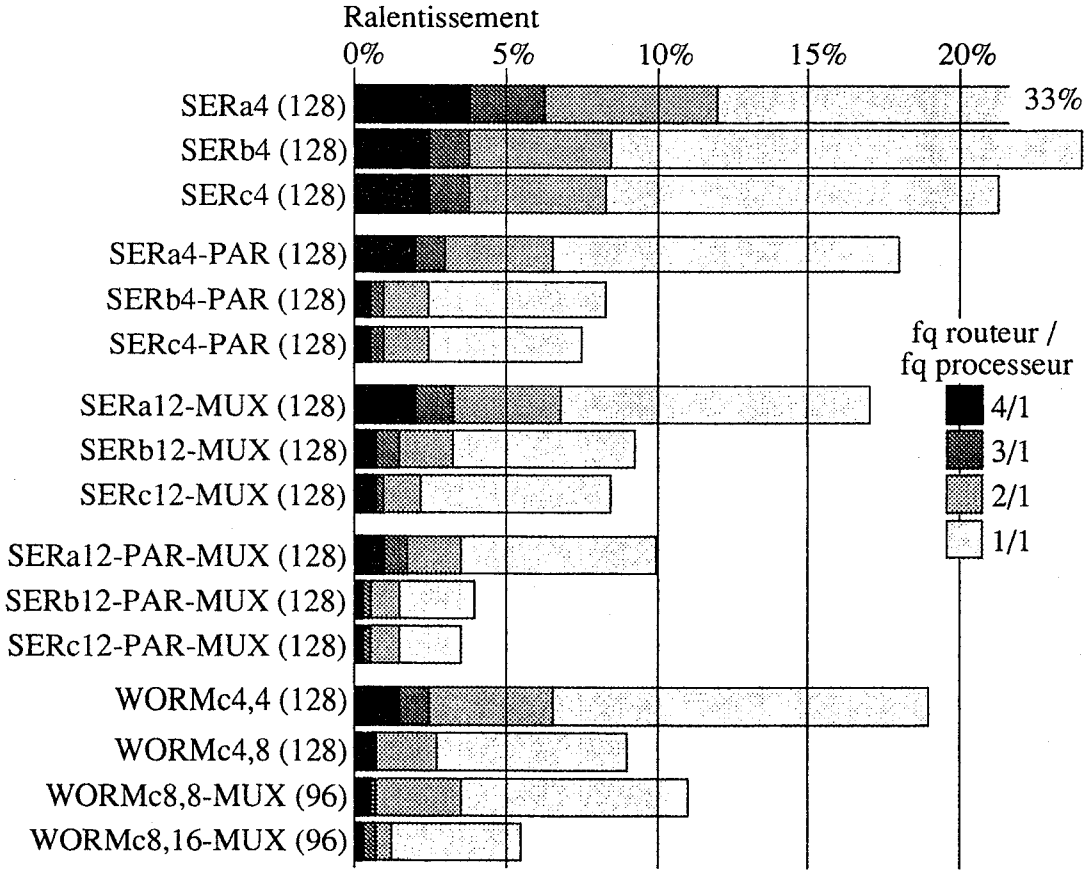


Fig. 48. — *Comparaison des routeurs possibles pour un compte de broches 100→ 150.*

A nombre de broches équivalent se dessine une hiérarchie assez nette de performances effectives :

1- Formes sérialisées aux frontières et multiplexées aux frontières (SER-PAR-MUX) et wormhole avec tampon de deux flits et multiplexage aux frontières (WORMc*n*,2*n*-MUX)

2- Formes sérialisées aux frontières (SER-PAR), formes multiplexées aux frontières (SER-MUX), wormhole avec tampon de deux flits (WORMc*n*,2*n*), et formes à bus, avec un routeur pipeliné par circuit (BUS3).

3- Formes sérialisées simples (SER).

Nous pouvons en conclure que la bande passante permise par un certain degré de connexion inter-circuits est mieux utilisée par un mécanisme de multiplexage que par un mécanisme de sérialisation. Une étude plus approfondie de l'impact du nombre de cellules sur un transfert multiplexé serait souhaitable pour déterminer comment balancer au mieux sérialisation et multiplexage lorsque l'intégration sera supérieure à 8 par 8 cellules, mais jusqu'à cette limite, il semble bien que le routeur SER-PAR-MUX soit un optimum d'un point de vue performances.

Nous n'avons malheureusement pas de mesures de complexité et de vitesse d'horloge pour toutes les solutions présentées ici. Le tableau 10 suivant montre les principales données dont nous disposons actuellement. Les données concernant le routage *wormhole* et la partie traitement proviennent de la conception en cours du circuit, effectuée par M. Karabernou, celles concernant le routage à bus sont le résultat d'une étude d'implémentation en standard cell menées par K. Khoumsi sur SOLO 2030.

	Wormhole	Bus
Surface totale des routeurs	100 mm2	21 mm2 dont routeur : 10 mm2 bus : 11 mm2
Fréquence du routeur	20 Mhz	30 MHz
Surface du circuit	341 mm2	262 mm2

Tableau 10. — *Estimations de surface pour un circuit de 64 cellules en technologie 0.8 μ*

Ces deux structures ne sont pas en toute rigueur comparables : le routeur wormhole utilise des flits de 8 bits, mais n'intègre pas de système de multiplexage aux frontières, il n'est donc pas exploitable pour un circuit à 64 cellules. Ce tableau montre toutefois bien qu'une structure à bus est mieux adaptée pour les circuits à forte intégration.

Chapitre III : Partie traitement

Dans ce chapitre, nous nous intéresserons plus spécialement à la partie traitement de la cellule : le processeur et sa mémoire. Rappelons brièvement les qualités que nous attendons de la partie traitement.

a) Compacité : c'est l'une des clefs de l'accès à une architecture massivement parallèle MIMD. La cellule complète ne doit pas comporter plus que quelques milliers de transistors, de manière à ce que des réseaux de grandes tailles puissent être réalisés à un coût et avec un encombrement raisonnable, par intégration de plusieurs cellules par puce.

b) Généralité : aucun domaine n'est visé en particulier. Tous les algorithmes doivent, autant que faire se peut, pouvoir être implémentés avec des performances correctes.

c) Programmabilité : le critère de compacité tend à réduire la puissance de la partie traitement *(processing element, PE)* au niveau du processeur comme de la mémoire. L'ensemble doit néanmoins rester exploitable, ce qui signifie que chaque cellule doit pouvoir modéliser des comportements algorithmiques assez complexes, justifiant un traitement MIMD.

d) Cohérence interne : comme le réseau doit être dimensionné par rapport au PE, les éléments qui composent le PE (mémoire, jeu d'instructions, registres...) doivent être cohérents entre eux, en dimension relative comme en structure.

Ces aspects ne peuvent pas être traités intégralement au niveau du PE, car certains sont étroitement liés à la programmation du réseau (non plus au niveau du processus, mais à celui de l'application parallèle) que nous aborderons dans le chapitre suivant.

Nous verrons dans un premier temps quels enseignements tirer de l'état de l'art en matière de processeurs. Nous nous attacherons ensuite à définir les fonctionnalités de base et à en déduire une première version, qui sera bien évidemment très imparfaite. Par l'étude de divers programmes, nous améliorerons ce premier jet, en mettant en évidence les fonctionnalités absentes qui font vraiment défaut et celles, présentes, qui sont difficiles à exploiter ou trop peu utiles. Cette démarche se fera sur la base d'une structure classique CISC 8 bits.

1. Les différents processeurs.

Dans le domaine des microprocesseurs, nous bénéficions d'un capital d'expérience considérable du fait de leur utilisation sur les machines séquentielles, ce qui n'était pas le cas avec les réseaux d'interconnexion, dans le chapitre précédent. Apparemment très diversifiées, les structures des microprocesseurs ont tendance à s'homogénéiser au niveau de leurs mécanismes internes et au niveau de l'architecture dans laquelle ils s'insèrent : les principales différences tiennent le plus souvent dans les contraintes de compatibilité ascendante propres à chaque constructeur. Ceci traduit sans doute une certaine maturité dans la conception de ces composants. Pour une machine parallèle MIMD, même si nous ne choisissons pas purement et simplement d'utiliser un circuit du commerce, il est naturel de chercher à s'inspirer du savoir-faire disponible.

1.1. Processeurs actuels.

1.1.1. Processeurs CISC.

Les processeurs dits "CISC" (Complex Instruction Set Computer) sont le résultat de l'évolution naturelle des premiers microprocesseurs. D'un point de vue matériel, l'évolution des capacités d'intégration a permis de mettre sur une même puce des fonctionnalités qui étaient jusqu'alors l'apanage de gros *mainframes* à processeurs multi-circuits. L'homogénéité que nous avons évoquée vient entre autre du fait que le choix d'intégrer un mécanisme donné plutôt qu'un autre ne se pose plus car la place est disponible pour les mettre tous.

Les fonctionnalités demandées à ces processeurs ont été progressivement affinées en relation avec les systèmes d'exploitation, les langages de programmation de haut niveau, le système de mémoire. Certaines tâches anciennement dévolues au logiciel ont

fait l'objet d'un support accru au niveau matériel. Les langages évolués ont contribué à rendre plus complexe le jeu d'instructions (modes d'adressage, instruction d'appel de procédure, etc.). Nous pouvons maintenant dire que tous les produits récents sont conçus pour travailler dans l'environnement suivant :

- Mots de 32 bits, unité de calcul en virgule flottante spécialisée, mémoire physique de plusieurs méga-octets, disques de grande capacité.
- Système d'exploitation multi-tâches multi-utilisateurs fondé sur une multiprogrammation en temps partagé, mémoire virtuelle, mécanismes de protection.
- Programmation à l'aide de langages procéduraux.
- Prise en charge d'interfaces utilisateur de plus en plus évoluées.

A l'évidence, ce cahier des charges ne concorde sur aucun point avec le notre :

- Le critère de surface n'est jamais posé comme impérieux dans le développement de ces circuits. Au contraire, toute la place potentiellement disponible sur un circuit est utilisée pour grossir le processeur, ce qui est en opposition totale avec nos objectifs.
- Les instructions de haut niveau destinées aux langages procéduraux ne nous intéressent pas ; dans notre cas, les appels de procédure sont des appels à distance et les structures de données sont distribuées.
- Le réseau fonctionne comme un coprocesseur exécutant des algorithmes relativement simples mais de coût élevé ; la gestion des entrées/sorties, le système d'exploitation sont reportés sur l'ordinateur hôte.

Cette liste d'incompatibilités est loin d'être complète, son but étant uniquement de montrer l'inadéquation de ces processeurs à notre problème. Malheureusement, leur généralisation, liée à un coût de fabrication de plus en plus bas a eu comme conséquence une absence de travaux sur les configurations plus petites. Pour des environnements bien particuliers, où la taille est un facteur important, les fabricants se sont contentés de bricoler les vieux processeurs 8 bits standard, sans introduire de réelles innovations.

Malgré tous leurs défauts (pour l'utilisation que nous voulons en faire), les processeurs CISC les plus modernes ont été systématiquement utilisés dans les calculateurs parallèles, comme nous l'avons vu dans le premier chapitre.

1.1.2. RISC

Durant les exposés formels ou informels que j'ai pu faire sur le thème du réseau cellulaire, la question de l'utilisation d'un processeur RISC (Reduced Instruction Set Computer) n'a jamais manqué d'être posée. En effet, elle est en apparence très

pertinente : un processeur RISC est par définition de taille très réduite, ce qui est justement notre objectif. Pour cette même raison, l'architecture MIPS de Standford [HEN83] a été retenue par le DARPA pour son projet de processeur AsGa [NAU97] destiné à une machine MIMD de traitement du signal (Advanced Onboard Signal Processor). Les capacités d'intégration de la technologie AsGa étaient à cette époque compatibles avec la surface nécessitée par un processeur RISC de base comme le MIPS.

Il faut d'abord préciser ce qu'on appelle un processeur RISC. S'agit-il simplement d'un processeur où le nombre d'instructions est faible ? Dans ce cas, les premières architectures de processeurs étaient toutes RISC sans le savoir ; nous reviendrons sur ces vieux processeurs un peu plus loin. Il semble que ce critère ne soit pas très bon (on a vu des processeurs "RISC" dédiés à l'exécution de langages de haut niveau ! [SCH84]). Nous devons plutôt voir quels sont les traits caractéristiques de ces processeurs.

Le principe de base des architectures RISC est de simplifier les instructions pour les exécuter plus vite et à moindre coût. Il ne s'agit pas forcément d'en réduire la teneur, mais plutôt d'obtenir une régularité totale (ce qui exclut les instructions complexes des CISC). Cette régularité s'impose sur le plan du codage et sur le plan de l'architecture temporelle. Elle est ensuite mise à profit par l'établissement d'un recouvrement important dans l'exécution des instructions. Par exemple, avec un pipeline à 5 étages, nous avons le découpage suivant : lecture de l'instruction, décodage, lecture des opérandes, calcul, écriture du résultat. On a alors l'impression qu'une instruction est exécutée à chaque cycle, alors que chacune d'entre elles prend 5 cycles.

Ce mécanisme repose sur deux hypothèses. La première suppose une absence de dépendance entre instructions proches : si une instruction a besoin du résultat de la précédente, il y a rupture dans l'alimentation du pipeline. De même, une rupture de séquence (branchement) provoque une rupture d'alimentation, que l'on peut réduire en plaçant une instruction utile, juste après le branchement, qui est exécutée que le branchement soit pris ou non. La gestion des dépendances est reportée sur le compilateur qui va tâcher de réordonnancer les instructions de manière, à permettre un recouvrement maximum.

La seconde hypothèse concerne la bande passante de la mémoire : il faut dans un même cycle charger une instruction, des opérandes et ranger un résultat. Les processeurs s'en accommodent généralement en utilisant des bancs de registres de taille très importante qui permettent de placer la plupart des variables globales, locales et temporaires dans des registres, minimisant ainsi les échanges avec la mémoire. Le banc de registres peut être organisé en *fenêtres* qui, entrelacées, permettre de passer les paramètres d'appel de procédure et les résultats de fonctions implicitement dans des registres, sans déplacement de données : seul le numéro de fenêtre courante change lors de l'appel et du retour (RISC I de Berkeley [PAT80]). L'architecture est généralement de type Harvard (bus d'instructions distinct du bus de données), et suppose l'usage de caches mémoire de taille non négligeable.

Quelles sont les caractéristiques des programmes que nous allons exécuter ? Tout d'abord, la taille de processus visée implique des séquences courtes, entrecoupées de points de communication qui sont autant de points potentiels de rupture de séquence. De plus, la granularité fine favorise un éclatement des évaluations d'expressions sur plusieurs cellules, le code d'une cellule sera essentiellement du code de contrôle, parsemé de tests et de branchements. L'alimentation régulière d'un pipeline d'exécution sera donc difficile à assurer.

Les structures de contrôles inclues dans le programme d'une cellule comportent rarement d'appels de sous-programmes, les mécanismes de fenêtrage sont donc inutiles. Contrairement aux RISC traditionnels, où la simplicité permet de dégager de la surface pour implémenter des registres en grand nombre, nous avons une politique différente de mise à profit de la simplicité qui passe par l'augmentation du nombre de cellules.

La mémoire est pour nous une ressource très limitée, à la fois en nombre de mots et en nombre de ports (il serait souhaitable d'éviter l'utilisation d'une mémoire double accès, plus coûteuse). Or l'orthogonalité des jeux d'instructions RISC induit un codage peu compact des programmes. Toutes ces remarques prêchent clairement contre l'utilisation des RISC dans la situation qui nous préoccupe.

1.2. Processeurs 8 bits.

D'un point de vue dimensionnel, les processeurs 8 bits plus anciens sont sans doute les plus proches de nos besoins : ils ont été conçus avec des contraintes de surface sévères. Notons tout de même que le système de mémoire dans lequel ils s'insèrent est généralement relativement plus gros et plus lent que celui que nous visons. De plus la capacité de gestion d'événement est assez réduite et peu efficace (traitement d'interruptions). S'ils peuvent nous servir de base de travail, ils devront être repensés au moins sur ces deux aspects.

1.3. Processeurs pour machines MIMD.

Bien que la littérature portant sur des processeurs qui ne soit pas exclusivement axés sur la puissance de calcul ne soit pas très abondante, il existe quand même quelques cas intéressants notamment les processeurs conçus pour les machines MIMD.

1.3.1. Transputer.

Le transputer est l'archétype du composant issu d'une démarche descendante. Le modèle de parallélisme mis en œuvre est le modèle CSP (Communicating Concurrent Processes) [HOA78]. Les ingénieurs d'Inmos en ont dérivé un langage de programmation, OCCAM [INM84], et un processeur chargé de le supporter dans un contexte MIMD.

Le transputer est apparemment proche de ce que nous cherchons. En fait, c'est un gros processeur possédant des instructions évoluées qui ne se justifient qu'avec un nombre de processus important sur chaque PE. La sémantique de communication, par rendez-vous, se prête très mal à une structure d'interconnexion de grand diamètre comme la grille, car les processus communicant doivent se trouver soit sur le même processeur, soit sur des processeurs directement connectés. De ce fait, tous les calculateurs récents basés sur ce composant utilisent un réseau d'interconnexion indirect de manière à simuler une interconnexion complète. C'est en fait la seule structure de communication cohérente avec les spécificités du transputer. Dans une structure à passage de messages, où la granularité fine va généralement induire des processus courts et l'affectation de chaque processus à un processeur différent, le transputer apparaît nettement comme insuffisant au niveau communication et surdimensionné au niveau multiprogrammation.

Enfin, le transputer occupe une surface nettement supérieure à celle dont nous disposons : avec une mémoire *on-chip* de 4K, il totalise plus de 60KT. Le successeur du T800, le T9000 est encore beaucoup plus gros.

1.3.2. MDP.

Le Message Driven Processor (MDP) du M.I.T.[DAL87b] prend le contre-pied total du Transputer : alors que ce dernier est conçu pour des voies de communications figées, synchrones, de voisin à voisin, le MDP est basé sur une communication asynchrone (par messages), sur des voies choisies dynamiquement, sans limitation de voisinage. C'est pour cette raison que nous le détaillons ici, bien qu'il n'ait pas l'impact du Transputer auprès de la communauté scientifique et industrielle.

Le modèle de programmation est ici orienté objet, et toute l'architecture interne en découle. Le principal souci des concepteurs a été de limiter la latence globale d'invocation des méthodes. Comme dans les cubes de la seconde génération (iPSC/2), une circuiterie spécialisée au niveau de chaque PE prend en charge toutes les tâches réseau, mais l'originalité de cette architecture est de chercher à limiter aussi la latence postérieure à la réception : détermination de l'objet récepteur, de l'adresse de la procédure à exécuter, création d'un processus et changement de contexte. Avec l'amélioration de la latence d'acheminement proprement dite, cette latence "aval" est devenue prépondérante (de l'ordre de 300µs). L'objectif du MDP est de la ramener à quelques microsecondes, de manière à ce qu'elle soit compatible avec l'acheminement et avec la durée d'activation des méthodes (qui est généralement très faible avec un parallélisme à grain fin).

Pour arriver à ce résultat, l'unité de traitement est dédoublée en un processeur de message MU (Message Unit) et un processeur d'instruction IU (Instruction Unit). La MU, à chaque incidence de message, décide soit de stocker le message en mémoire, soit d'effectuer une préemption de l'IU, selon l'état de cette dernière et le degré de priorité

du message. La détermination de l'objet récepteur ainsi que celle de l'adresse de la méthode sont automatisées grâce à une structure de mémoire duale : le mémoire est à la fois une RAM (accès indexé) et une mémoire associative (accès par clef), ce qui permet d'en transformer certaines parties en tables de conversion d'identificateurs en adresses physiques réseau ou mémoire. D'autre part, la multiprogrammation est optimisée par l'adoption d'un modèle de programmation orienté mémoire, et d'un système de double jeu de registre (comme dans le Z80) qui sert au passage rapide d'un niveau de priorité à l'autre.

D'un point de vue programmation, le MDP fournit tout ce qui est nécessaire pour l'implémentation des langages à objets, mais grâce à un matériel non négligeable. C'est donc une machine langage dont on peut attendre un rapport coût/performance défavorable sur les problèmes statiques.

2. Fonctionnalités requises.

Constatant qu'aucun processeur existant n'est réellement satisfaisant (ce qui n'est pas surprenant compte tenu du créneau un peu original dans lequel nous nous positionnons), il faut définir notre propre processeur. Nous commencerons par étudier les fonctionnalités que nous désirons y inclure.

2.1. La synchronisation entre cellules.

Ainsi que nous allons le voir, les processus de base doivent souvent être décomposés en sous-processus de façon à assurer l'absence d'interblocage au niveau logiciel. Cette contrainte est intimement liée au partage de la voie de communication routeur/processeur par plusieurs canaux. Prenons le cas simpliste d'un opérateur arithmétique à deux entrées, par exemple un additionneur. Le contrôle de flux est de type *demand driven* (envoi de requêtes en sens inverse des données, *fig. 1*). Pour décrire la structure de synchronisation, nous supposerons disposer des instructions suivantes :

- Recevoir (Canal) : réception d'un message
- Envoyer (Canal) : envoi d'un message

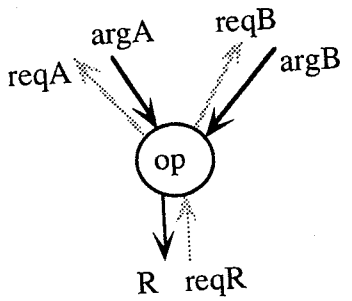


Fig. 1 — *Synchronisation par requêtes.*

L'algorithme de base de la cellule pourrait être :

```
CYCLE
  Envoyer (reqA)
  Recevoir (A)
  Envoyer (reqB)
  Recevoir (B)
  Calculer
  Recevoir (reqR)
  Envoyer (R)
```

Un processus de ce type présente une possibilité d'interblocage direct. Rien n'indique dans quel ordre les messages attendus seront reçus. Si cet ordre n'est pas celui escompté, deux cas peuvent se présenter. Si un contrôle de type est assuré pour vérifier que le message est bien celui attendu, le message présent dans le tampon d'entrée ne sera pas retiré et va gêner l'arrivée du message attendu. Si aucun contrôle n'est effectué, l'algorithme est simplement faux.

En introduisant un typage et une réception obligatoire, cette algorithme devient :

```
CYCLE
  Envoyer (reqA)
  Envoyer (reqB)
  POUR i=1 A 3
     Recevoir(type,valeur)
     SELON type
        A:Ranger(A,valeur)
        B:Ranger(B,valeur)
        ReqR:rien
  Calculer
  Envoyer(R)
```

Mais cette expression n'est pas encore très satisfaisante, car elle empêche tout recouvrement entre les opérateurs : les calculs sont sérialisés. Pour y remédier, il faut introduire une anticipation des demandes.

```
Envoyer (reqA)
Envoyer (reqB)
  CYCLE
  POUR i=1 A 3
     Recevoir(type,valeur)
     SELON type
        A:Ranger(A,valeur)
        B:Ranger(B,valeur)
        ReqR:rien
  Envoyer (reqA)
  Envoyer (reqB)
  Calculer
  Envoyer(R)
```

Mais si cette nouvelle expression est susceptible de générer un bon recouvrement entre les opérateurs, elle introduit le risque d'une autre forme d'interblocage, indirecte, dont nous avons exposé le versant "réseau" § B.2.2.2. du chapitre II. Le versant "processeur" de cet interblocage tient dans la dépendance créée entre émission et réception : pour pouvoir envoyer le résultat, il faut que le tampon de sortie soit vide. Or l'obtention de cet état du tampon de sortie peut être de façon indirecte subordonnée, via une chaîne de dépendances, au retrait du contenu du tampon d'entrée de la cellule (l'une des deux opérandes demandées de façon anticipée). L'émission n'est possible que s'il y a réception, mais dans le programme la réception suit l'émission.

Ces deux cas d'interblocage peuvent être condensés en une loi de prévention : *le retrait d'un message incident doit être fait dans un délai indépendant de toute condition sur le réseau.*

Si tous ces problèmes peuvent être tirés d'un exemple simpliste, que dire de ceux que nous aurions à résoudre sur des algorithmes à dynamique plus complexe. Un modèle multiprogrammé de cellule peut apporter une solution beaucoup plus élégante. La loi de prévention peut être vérifiée dans le cas général en scindant chaque opérateur en un processus de réception et un processus de calcul-émission, technique qui, appliquée à l'exemple précédent, donne :

```
SEQ
  AReçu = BReçu = ReqReçue = faux
  PAR
      SEQ /* processus de réception */
          CYCLE
              Recevoir(type,valeur)
              SELON type
                  A:Ranger(A,valeur), AReçu = vrai
                  B:Ranger(B,valeur), BReçu = vrai
                  ReqR:ReqReçue = vrai
      SEQ /* processus de calcul et d'envoi */
          Envoyer (reqA)
          Envoyer (reqB)
          CYCLE
              Attendre(AReçu)
              Opa = A, AReçu = faux
              Envoyer (reqA)
              Attendre(BReçu)
              Opa = B, BReçu = faux
              Envoyer (reqB)
              Calculer
              Attendre(ReqReçue)
              Envoyer(R)
```

Le rôle du processus de réception peut être vu comme un *éclatement* du canal de réception en canaux d'entrée logiques, qui supprime la possibilité d'interblocage direct. Le contrôle de flux par requête assurera qu'il n'y a pas d'écrasement du contenu d'un tampon logique, et par là même prévient tout interblocage indirect.

2.2. Multiprogrammer ?

La capacité de pouvoir multiprogrammer une cellule est intéressante a priori, pour trois raisons.

a) Comme nous venons de le voir, elle permet une prévention des interblocages liés à la synchronisation des cellules.

b) Elle permet d'affecter plusieurs processus logiques à chaque cellule, ce qui, bien que contraire à la philosophie de base du parallélisme massif, assure un équilibrage de charge rudimentaire. Rien n'indique en effet que la décomposition suivant une granularité fine d'un problème donne des processus de taille homogène. De plus, les dépendances entre processus font que tous les processus ne fonctionnent pas en même temps. Si on choisit de façon correcte les processus regroupés, en mettant sur une même cellule des processus qui ne sont pas activables dans les mêmes plages temporelles, un rapport performance / coût meilleur peut être obtenu. Parfois, le fait de regrouper deux processus liés par une dépendance de séquentialité stricte permet de gagner en performance, car les transmissions de données sont faites par la mémoire et non par le réseau. Un travail d'optimisation globale de ce type a été effectué par E. Payan sur la compilation du langage LUSTRE, qui a donné des résultats très encourageants (voir chap. IV).

c) La dernière raison de multiprogrammer est la possibilité dans un même processus de réordonner dynamiquement les actions en fonction des conditions qui président à leur exécution. Supposons par exemple qu'un opérateur doive envoyer son résultat à deux cellules différentes : si la requête de la première cellule n'est pas arrivée alors que celle de la seconde l'est, il peut choisir d'intervertir l'ordre de service. Nous ne faisons là qu'exploiter un parallélisme intra-opérateur.

Le mécanisme de multiprogrammation doit être en grande partie supporté par le matériel, si nous voulons conserver des délais d'activation des processus compatibles avec leur durée d'activation. C'est ce qu'a bien mis en évidence Dally dans son étude de l'implémentation des langages à objets : l'activation d'une méthode découlant toujours de l'arrivée d'un message, une latence de communication (envoi, transfert, réception, détermination de la méthode, création du processus) trop élevée par rapport à la longueur de la méthode entraînera la concurrence d'exécution faible. La minimisation de la partie processeur de la latence est l'objectif fondamental du MDP ; le coût des tables de méthode y reste très élevé.

Au tout début de notre étude, nous avions retenu la solution d'une multiprogrammation cablée. Mais comme la plupart des programmes vont devoir mettre

en œuvre un processus de réception relativement semblable, et rarement plus qu'un seul processus de calcul, on est en droit de se demander si un système de multi-programmation général est véritablement rentable. Ne serait-il pas plus simple de réaliser directement au niveau matériel ce processus de réception, et de fournir quelques primitives pour implémenter par logiciel la multiprogrammation résiduelle (calcul et émission) ? Cette solution (que nous avons finalement adoptée) présente de nombreux avantages :

- simplicité de réalisation
- latence d'activation d'un processus unique de calcul plus réduite, au détriment de l'activation dans un contexte à plusieurs processus de calcul (cas moins courant)
- gain de place mémoire (le processus de réception n'y figure plus).

Le processus de réception été rendu le plus simple possible pour qu'il soit possible de le microprogrammer.

```
CYCLE
  Attendre un message
  Déterminer le canal d'après le typage du message
  Ranger la valeur dans le tampon du canal
  Affirmer l'indicateur d'arrivée
  Vider le tampon d'entrée
```

Il faut noter l'absence de prévention d'écrasement : si un message arrive, destiné à un canal actuellement plein, la valeur précédente est remplacée. Le contrôle de flux doit assurer que cette situation fâcheuse ne se présentera pas[1].

2.3. Programmation des communications.

Du point de vue des processus de calcul, le processus de réception réalise un éclatement du canal d'entrée en canaux logiques sur lesquelles nous pouvons définir les fonctionnalités suivantes :

- Recevoir (canal) → valeur
  Tant que l'indicateur de présence est nul : attendre
  Remettre à zéro l'indicateur de présence
  Renvoyer le contenu du canal.

[1] Dans l'équipe, l'idée a été émise de supprimer le contrôle de flux explicite, et de laisser le message dans le tampon d'entrée tant que le précédent message du canal n'a pas été consommé par le programme. Le contrôle de flux est alors réalisé par accumulation des messages dans le réseau jusqu'au producteur, qui se trouve bloqué dans ses émissions par la non-libération de son tampon de sortie. Hormis un processus de réception plus compliqué, cette stratégie, comme nous l'avons montré, suppose que le programme consommera le message au bout d'un temps fini. Ajouter des contraintes de programmation et en contention sur le réseau ne nous semble pas une très bonne idée : le contrôle de flux est certes supprimé, le contrôle du programmeur sur son programme risque de l'être lui aussi.

- Envoyer (adresse cellule, canal, valeur)

  Tant que le tampon de sortie est plein : attendre

  Générer un message [adr,canal,valeur]
- Tester (canal) → booléen

  renvoyer l'indicateur de présence du canal.

Cette dernière fonction permet de réaliser une multiprogrammation rudimentaire, par scrutation circulaire de la calculabilité des processus. Un processus est dit calculable si ses données sont présentes et si ses résultats ont été demandés. La structure générale d'un programme de cellule multi-processus peut se décrire de la façon suivante :

```
PROCESSUS P1,P2,P3,...,Pn
CYCLE
  SI P1 calculable ALORS executer P1
  SI P2 calculable ALORS executer P2
  SI P3 calculable ALORS executer P3
  ...
  SI Pn calculable ALORS executer Pn
```

Les canaux que nous avons définis jusqu'à présent sont des canaux unidirectionnels. Le contrôle de flux est assuré par couplage d'un canal de donnée et d'un canal de requête en sens inverse. Nous aurions pu choisir de nous placer à un niveau supérieur, avec des instructions manipulant des canaux bidirectionnels : Ecrire = Recevoir(req) + Envoyer(val), et Lire = Recevoir(val) + Envoyer(req). Mais outre le fait que ces primitives sont plus compliquées et qu'elles nécessitent une primitive annexe d'envoi de requête initiale, elles posent la taille du message égale à celle de la donnée. Des messages physiques de taille variables s'en accommoderaient donc bien, mais avec des messages de taille fixe, un message logique doit être éclaté en plusieurs messages physiques, et donc en plusieurs canaux contrôlés séparément, là ou une seule voie de contrôle de flux aurait suffit.

2.3. Structures de contrôle.

Si la modification du contexte de mise en œuvre impose des fonctionnalités nouvelles, essentiellement de communication, on peut aussi se demander si les fonctionnalités classiques sont bien toutes nécessaires. Le parallélisme à grain fin tend à étaler dans l'espace les structures temporelles usuelles telles que la récursivité et les boucles. D'autre part, les processus sont de petites tailles et ne pourront donc jamais relever de structurations internes bien complexes. En conséquence, toute la mécanique d'appel de sous-programme doit être repensée pour que son coût reflète son importance réelle. Nous ne pouvons affirmer que l'appel de sous-programme est totalement inutile, mais son seul rôle est un gain de compacité de code : routine d'émission d'une suite de messages, routine de multiplication, etc. Déterminer l'importance des structures

itératives dans notre contexte est moins crucial, dans la mesure où leur programmation ne fait pas appel à des instructions spécifiques, tout au moins dans les processeurs simples.

Les processeurs classiques intègrent à peu près tous un système plus ou moins sophistiqué de traitement d'événement. Sous un même mécanisme se cachent en fait des fonctionnalités très diverses. Les interruptions externes sont un moyen de communiquer de façon asynchrone avec l'extérieur ; elles servent souvent à implémenter les entrées/sorties lentes (clavier, capteurs, etc.). Si les interruptions sont générées par un dispositif externe de type *timer*, elles peuvent servir de base temporelle pour implémenter une multiprogrammation en temps partagé ou de faire de la programmation temps réel. Les exceptions (interruptions logicielles) sont utilisées pour extraire du programme le traitement des erreurs de nature matérielle (erreur d'adressage par exemple) et des erreurs de naturel logicielles (division par zéro par exemple). Sur la famille Motorola 68000, les exceptions peuvent servir d'interface entre programmes utilisateur et noyau système.

Le traitement d'interruptions est-il une fonctionnalité utile pour une cellule ? Nous avons vu que le traitement de l'asynchronisme se fait par le biais de messages. L'arrivée d'un message peut être vue comme une interruption, dans la mesure où elle déclenche d'activation d'un processus hardware de réception, mais il serait souhaitable que cette activation prenne moins de temps que la prise en compte d'une interruption sur un processeur classique. La fonction multiprogrammation ne nous intéresse pas dans la mesure où nous nous contentons d'une approximation plus grossière du parallélisme, mais même dans le cas d'une multiprogrammation générale, le changement de processus serait plutôt basé sur la présence/absence des données que sur des tranches temporelles.

Pour utiliser les exceptions en tant qu'interface avec le noyau, encore faudrait-il un noyau. Par contre, le traitement d'erreur serait une fonctionnalité réellement intéressante, mais il ne faudrait certainement pas l'implémenter de cette manière. En effet, une erreur peut rarement faire l'objet d'un recouvrement local à la cellule où elle est survenue. Ouvrons une petite parenthèse pour examiner plus en détail ce point.

Plusieurs approches sont possibles, selon que l'on veut simplement signaler l'erreur ou que l'on veut pouvoir analyser en détail la situation dans laquelle elle s'est produite. Un dispositif matériel du type OU global peut permettre à la cellule qui a détecté une erreur d'informer l'extérieur, et en particulier l'hôte, de l'occurence de cet événement. L'hôte, sachant que le résultat du travail du réseau n'est pas valide, peut décider d'une réinitialisation générale, mais il peut être intéressant pour l'utilisateur d'extraire l'état global du réseau pour l'analyser et tenter de découvrir la source de l'erreur. Une telle extraction est possible au moins d'une technique de *scan-path*[1]. Pour

---

[1] *scan-path* : technique qui au sens strict consiste à chaîner les points de mémorisation d'un circuit en un grand registre à décalage permettant d'extraire ou d'initialiser l'état de ce circuit depuis l'extérieur.

être cohérent, il faudrait ajouter un mode pas-à-pas et une possibilité de poser des points d'arrêt dans le programme.

Une solution de ce type, relativement lourde à mettre en œuvre, demande à être sérieusement évaluée avant d'être adoptée, notamment en conjonction avec le test d'intégrité du réseau. Dans l'attente de données plus précises sur le sujet, nous avons préféré en rester à une autre solution, purement logicielle, qui consiste à faire circuler les erreurs au même titre que les données, ou plutôt à réserver des codages particuliers des données pour marquer l'occurence d'une erreur, sa nature, la position où elle s'est produite. Ces codages seraient analogues aux "Not-a-Number" de la norme IEEE 754 sur les nombres en virgule flottante. Par leur caractère absorbant, ces valeurs d'erreur pourraient être propagées dans le graphe de calcul tout comme le sont les résultats corrects, et être finalement récupérées par l'hôte. Si les flux de données du programme sont réguliers, la mise en correspondance de l'erreur avec les données qui l'ont provoquée peut être assez simple. En fusionnant de la sorte circulation des erreurs et circulation des données, l'intérêt d'un traitement interruptif des erreurs disparaît. Il nous reste finalement peu de raisons de conserver un mécanisme d'interruption.

2.4. Modes d'adressage.

Les processeurs fournissent traditionnellement un grand nombre de modes d'adressage, mais les compilateurs n'en exploitent souvent qu'un petit sous-ensemble. Ils recouvrent quatre grandes fonctions, liées aux principaux types de données structurées :

- l'accès aux *tableaux* (indexation) : mode basé indexé, mode indirect, mode indirect post-incrémenté, mode indirect pré-décrémenté
- l'accès aux *enregistrements* (record PASCAL) : modes basés
- l'accès aux *structures dynamiques* (déréférenciation de pointeurs) : mode indirect
- l'accès aux *contextes d'exécution* (empilement, dépilement, accès aux variables locales) : modes basés, instructions de manipulation de pile, mode indirect post-incrémenté et mode indirect pré-décrémenté (68000).

Tout comme les structures de données, ces modes sont souvent combinés pour donner des modes plus complexes. Dans le réseau cellulaire, les structures de données sont, comme les structures de contrôle, réparties pour permettre un parallélisme maximum. Quoi qu'il en soit, cet éclatement est rendu nécessaire par la faible taille de mémoire prévue. En conséquence, les modes d'adressages destinés à les supporter sont pour nous d'un intérêt très relatif. Si nous ne leur découvrons pas des justifications nouvelles, rien ne nous empêche de n'en retenir que le strict minimum : adressage immédiat et adressage absolu. Nous avons là une opportunité d'un gain de surface considérable, au niveau du séquenceur et du chemin de données.

2.5. Opérations.

Il n'y a pas beaucoup de disparités dans les opérations de base fournies par les processeurs généraux. Des points de désaccord mineurs portent sur la meilleure sémantique pour les opérations arithmétiques et pour les opérations de décalage. Si la taille des données manipulées est inférieure à celle des langages évolués, la question se pose d'avoir des opérations en "multiprécision" (instruction d'addition 32 bits + 32 bits sur un processeur 16 bits, par exemple).

Des différences plus sensibles de performances peuvent apparaître sur les opérations complexes. Ainsi, lorsque la place disponible le permet, la présence d'une multiplication et d'une division microcodée ou cablée s'avère généralement appréciable. Sur les processeurs modernes, des opérations sur les types de données non entiers sont aussi implémentés : nombres en virgule flottante, vecteurs, chaînes de bits. Mais nous avons justement toutes les chances de ne pas disposer de la place nécessaire, et nous serons donc obligés d'en rester à un jeu d'opérations minimum : addition, soustraction, rotations à droite et à gauche, *ou* inclusif et exclusif, *et* logique.

3. Structure retenue.

Nous pouvons maintenant étudier un processeur correspondant à ces fonctionnalités. Afin de justifier les choix techniques que nous avons faits sur ce processeur, nous avons jugé utile calquer la présentation qui suit sur la démarche adoptée : choix d'une taille de processeur (§3.1), définition arbitraire d'une première version (§3.2), puis améliorations successives (§3.3) sur la base des données qualitatives et quantitatives résultant du codage des exemples (ceux du *benchmark* du chapitre précédant), pour finir avec un résumé du processeur final (§3.4). Le lecteur intéressé uniquement par la version finale pourra passer directement de la section §3.2 à la section §3.4.

3.1. Taille du processeur.

Pour supporter les fonctionnalités que nous venons de définir, il ne paraît pas nécessaire de s'écarter des structures architecturales et des jeux d'instructions éprouvés. Nous devons tout d'abord décider des largeurs de mots que nous allons utiliser pour les registres, les différents bus, les adresses, la mémoire, etc., grandeurs que rien n'oblige a priori à poser comme équivalentes, si ce n'est un souci de simplicité conceptuelle dont on peut espérer qu'elle amènera une simplicité de réalisation.

Techniquement, les largeurs de mots peuvent être égale à n'importe quelle valeur à partir de 1 bit. L'usage communément admis des puissances de deux renvoie essentiellement à des normes de représentation des types simples (booléens, caractères,

entiers, réels). Le but de ces "normes" est de permettre un portage logiciel et une communication entre machines. Le premier de ces deux impératifs n'existe pas avec le réseau cellulaire, car il apparaît totalement inaccessible pour d'autres raisons relevant de l'algorithmique. En revanche, le respect des représentations traditionnelles peut nous permettre de faire l'économie de dispositifs de transcodage. De ce point de vue, nous pouvons considérer que le bit, le demi-octet, l'octet, le mot de 16 et celui de 32 bits sont les seules tailles correctes.

Dans un processeur simple, les différents types d'informations qui sont stockés, manipulés et transférés (nombres, adresses, instructions) utilisent tous les mêmes ressources matérielles (mémoire, UAL, bus). Seules des contraintes de coût peuvent susciter le besoin d'une hétérogénéité : par exemple, le 8085 utilisait un bus d'adresse à deux phases, l'une pour les poids forts, l'autre pour les poids faibles, pour réduire la largeur des voies de communication entre processeur et boîtiers mémoire. Avec le réseau cellulaire, nous sommes dans un environnement clos où nous pouvons supposer le processeur à l'abri de ce genre de contraintes. C'est la partie routage qui les supporte, comme nous l'avons vu dans le chapitre II.

Cette volonté d'homogénéité est contrecarrée par la disparité des types d'informations. Pour les données, on peut avoir à manipuler aussi bien des bits (traitement d'image bitmap) que des nombres en virgule flottante. Choisir une taille trop petite impose une décomposition des opérations, décomposition qui se paie au niveau du programme ou du microprogramme selon les cas. Inversement, une taille trop grande se traduit par une sous-utilisation chronique des bits de poids fort, allant à l'encontre de nos exigences d'économie. Le problème se pose avec les adresses : un espace d'adressage trop court peut forcer à utiliser un système de commutation de banc mémoire pour l'augmenter artificiellement, système qui est toujours pénible à gérer. Un espace mémoire trop large, tel qu'un espace de 64Ko pour une cellule, conduit à manipuler des grosses adresses inutilement, d'où une perte de place mémoire. La solution est un système de codage des adresses plus souple, par exemple une forme courte et une forme longue. Au niveau des instructions, il est bien connu que la compacité du code ne va pas de pair avec la simplicité d'un codage orthogonal.

Théoriser la problématique du codage n'étant pas le propos de cette étude, ramenons maintenant ces quelques considérations au cas qui nous préoccupe. La plage de taille mémoire envisagée va de 128 octets (en deçà, la capacité d'expression de comportement d'une complexité propre à justifier un traitement MIMD est compromise) à 1Ko (au delà, étant donnée la taille des processus, un bon taux d'utilisation de la mémoire serait subordonné à une multiprogrammation assez intensive ou a l'augmentation du grain de parallélisme).

Nous aimerions d'autre part contenir la complexité de la cellule à dix ou vingt mille transistors, surface incompatible avec l'intégration d'une véritable architecture 16 bits. A l'autre extrême, une architecture 4 bits semblerait sous-dimensionnée par rapport

à la complexité du jeu d'instruction qui nous est nécessaire pour coder des comportements dont nous souhaitons doter les cellules. Une architecture 8 bits apparait donc comme un compromis bien adapté, sentiment conforté par la longue expérience historique qui nous assure de leur validité pour un usage général.

Le point noir qui subsiste concerne l'espace d'adressage : des adresses 16 bits sont trop grosses, mais des adresses 8 bits nous bloqueraient à un maximum de 256 positions mémoires. Nous avons décidé de limiter les adresses à 8 bits, privilégiant donc l'homogénéité à l'extensibilité, pour la raison suivante : un processus trop gros peut, la plupart du temps, être scindé en plusieurs processus, et créer un niveau de parallélisme supplémentaire. Cette affirmation n'est pas gratuite, elle est basée sur notre expérience de la programmation (cf. multiplication de matrice, recherche de plus court chemin), mais malheureusement celle-ci est encore trop limitée pour avoir valeur de démonstration.

3.2. Premier jet.

Nous avons décidé dans un premier temps de conserver la structure de jeu d'instructions des processeurs 8 bits de type 6800, ne voyant pas de raison immédiate de changer ce qui a fait ses preuves. Nous commencerons par décrire la base de départ avant d'exposer progressivement les modifications et améliorations introduites. Comme nous n'aurons pas la place d'inclure beaucoup de registres, une architecture à accumulateur n'apparaît pas comme une mauvaise solution, nous nous en servirons donc[1].

3.2.1. Communication.

Selon le schéma que précédemment défini, un message est constitué d'une adresse de cellule destination, d'un typage (qui est en fait l'adresse du canal logique), et d'une donnée, autant de champs dont nous devons définir la magnitude.

Pour la donnée, nous avons deux options : soit un message de taille fixe avec un octet de donnée, soit un message de taille variable. Nous avons vu chapitre II que les messages logiques sont de taille modeste et très rarement variable. De plus, des messages de taille variable induisent un sur-coût au niveau des routeurs (séquenceurs et/ou tampons), au niveau des processeurs (l'émission doit spécifier la taille du message et la réception devient une opération complexe) et au niveau de l'interface partie traitement - routeur. En fin de compte des messages à champ donnée variable paraissent peu justifiés algorithmiquement, coûteux à implémenter, et peu cohérents avec les possibilités de manipulation dont nous voulons doter le processeur (niveau octet).

---

[1] Une comparaison avec d'autres architectures, notamment des architectures à registres banalisés, à pile, et à pile hybride, aurait du logiquement compléter cette étude, mais le temps a manqué pour la réaliser.

En ce qui concerne les adresses de destination, nous devrons aussi choisir entre une taille variable ou fixe, et dans ce dernier cas, il faut définir une portée. Le corollaire d'une portée variable est un traitement en arithmétique sérielle au niveau des routeurs, ainsi que l'ensemble des désagréments répertoriés pour un champ donnée variable (en fait, un champ adresse variable va de pair avec un champ donnée variable). Ce coût doit se justifier par une réelle utilité au niveau de la programmation. Est-ce bien le cas ? Pour les programmes à topologie cristalline bidimensionnelle, une portée de 3 ou 4 est généralement suffisante. Pour les programmes à topologie aléatoire, un bon placement permet souvent de se ramener à des portées faibles. Les structures arborescentes ou de forte dimension sont des cas défavorables, pour lesquels toute portée limitée est susceptible d'être dépassée. Heureusement, l'utilisation de relais de communication fourni une solution praticable. Le seul cas réellement problématique est celui où le graphe de communication est dynamique, cas où l'emploi de relais est très délicat.

Une portée variable est donc relativement peu justifiée, et généralement simulable quand elle est nécessaire, au moins pour les programmes statiques. Pour les mêmes raisons, fixer une portée à une valeur arbitraire en se basant uniquement sur les programmes n'est pas non plus très réaliste, car ils ne prendraient pas en compte un certain nombre de critères structuraux importants, tel que l'encodage des adresses de destination. En effet, ces adresses doivent être stockées et manipulées au niveau du processeur, et donc sont liées à la taille du mot de donnée : une adresse sur 2 x 4 bits, 2 x 8 bits, etc. De plus, la taille des adresses détermine le coût de leur traitement et de leur mémorisation au niveau du processeur et des routeurs, et donc une portée importante doit se justifier algorithmiquement, ce qui n'est pas le cas pour de nombreux programmes. Enfin, il convient de garder un rapport entre la partie utile du message (la donnée) et le reste qui ne soit pas trop défavorable, pour que la bande passante soit bien utilisée. Tout cela concoure à faire adopter des adresses relatives de destination codées sur 2 x 4 bits.

Il est évident que la taille du tag va déterminer le nombre de canaux référençables. Une solution qui minimise cette taille consisterait à se servir du tag pour indexer une table donnant l'adresse physique en mémoire du canal. Mais nous perdons alors de la place en mémoire à cause de cette table, et nous rendons plus complexe la réception. Au lieu de cela, nous proposons le poser le tag comme égal à l'adresse physique du canal. Doit-on alors pouvoir référencer n'importe quelle position mémoire (tag de 8 bits) ou bien peut-on limiter l'étendue de la zone de réception (tag de 4 bits par exemple) ? Trois raisons nous font préférer un tag de 8 bits :

- la programmation en a montré que l'utilité
- le chargement du programme de la cellule peut être envoyé sous forme d'une suite de messages normaux (voir annexe 3, §2.)
- le tag peut être facilement manipulé par le programme (voir par exemple le tri de chaines de caractères, dans le chapitre suivant).

La structure de message retenue est donc la suivante :

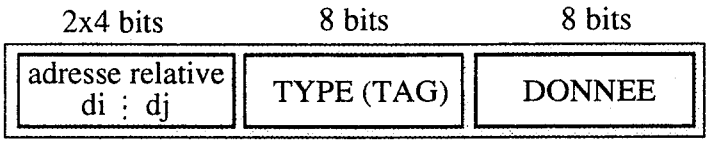


Fig. 2 — *Format des messages.*

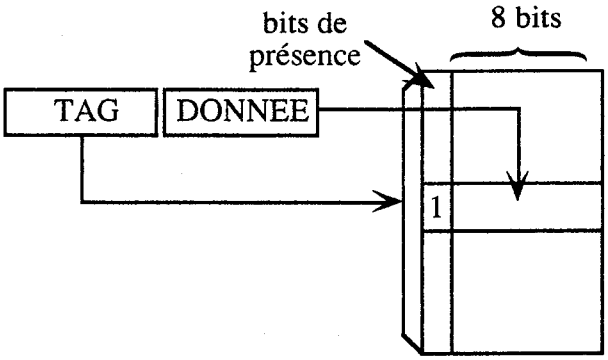


Fig. 3 — *Rangement d'un message en mémoire.*

Les instructions de base de communication sont l'envoi de message (SEND) et la réception blocante (GET).

```
GET (adr) :
  Tantque Presence[adr] = 0 attendre
  Accumulateur := Mem[adr]
  Presence[adr]:= 0
```

Le vecteur `Presence` est le vecteur de "9ème bit" parallèle à la mémoire *(fig. 3)*, qui indique si un message est arrivé ou non dans une position (canal). L'accumulateur contient en sortie la donnée contenue dans le message reçu sur ce canal. Le bit de présence est remis à zéro pour préparer le canal à la réception du message suivant.

```
SEND (adr) :
  Tantque Tampon_sortie.plein = 1 attendre
  Tampon_sortie.donnee := Mem[adr]
  Tampon_sortie.tag := Mem[adr+1]
  Tampon_sortie.adr := Mem[adr+2]
  Tampon_sortie.plein := 1
```

Cette instruction suppose la présence en mémoire d'une structure de donnée contenant le message à envoyer. Cette structure peut être remplie à l'exécution par des mouvements de données, mais il est plus simple de préparer une structure par canal et de l'initialiser à la compilation. Ainsi, dans la plupart des cas, seule la donnée devra être écrite dans la structure et, pour les requêtes, l'instruction SEND se suffit à elle même.

L'ordre de rangement des champs est celui de stabilité croissante (la donnée est très souvent changée d'une émission à l'autre sur un canal, le tag plus rarement, l'adresse pratiquement jamais. Cet ordre, arbitraire initialement, a trouvé depuis des justifications sérieuses (relation avec l'adressage indirect) que nous verrons plus loin. D'autres formes pour la fonctionnalité d'envoi de message peuvent être envisagées :

- Manipulation directe des champs du tampon d'émission : cela permettrait de ne pas recharger dans le tampon de sortie des champs que l'on sait inchangés d'une émission sur l'autre (à condition que le routeur n'en altère par le contenu), mais la fréquence d'occurence de cette opportunité d'optimisation est faible, elle est incompatible avec la multiprogrammation, et plus lourde à gérer au niveau du programme, puisque chaque opération de l'algorithme du SEND est assuré par une opération différente (en plus des instructions d'accès aux champs, il faut une instruction d'attente de libération du tampon de sortie, et une pour dire au routeur que le message est complet).
- Des formes SEND où la provenance des valeurs des champs n'est pas homogène (typiquement la donnée vient de l'accumulateur et le reste de l'instruction ou d'une structure de donnée) :

  a) Donnée := acc, Adresse := littéral, Tag:= littéral

  b) Donnée := acc, Tag := Mem[adr], Adresse := Mem[adr+1]

  La forme *a* suppose que l'on puisse établir une bijection entre points d'émission et canaux, pour ne pas répéter leur adresse destination et tag, ce qui veut dire entre autres, qu'on se coupe de toute possibilité de dynamisme dans le choix des canaux. L'examen des programmes, ainsi que la technique de programmation par tables de messages que nous verrons plus loin permettent de dire que cette forme seule ne peut suffire, même si elle peut présenter un intérêt de temps en temps. La forme *b* est plus intéressante, car la donnée est souvent l'élément variable du message, et représente le résultat d'un calcul. De plus, il n'y a pas de donnée significative dans les messages de requête. Toutefois, entre la fin du calcul et l'émission doit souvent s'intercaler une réception de requête et, dans d'autres cas, le résultat ne tient pas sur un octet, ce qui oblige à le stocker complètement ailleurs. Déterminer si cette forme est préférable à la forme initiale n'est pas facile, nous y reviendrons ultérieurement.

Le prédicat "tester(canal)" est implémentable par un mécanisme d'indicateur : supposons que nous possédions une instruction TST équivalente à celle du 6800, qui positionne les indicateurs Z (zéro) et N (négatif) du registre d'état, nous pouvons ajouter un indicateur F (full) au registre d'état, et le positionner dans l'instruction TST. Cette solution n'a pas été retenue, car elle est d'une manipulation assez lourde lorsque l'on cherche à évaluer la calculabilité d'un processus à plusieurs entrées :

```
processus :
  TST    entrée1
  brancher si non R aller au processus suivant
  TST    entrée2
  brancher si non R aller au processus suivant
  ...
  corps du processus
processus suivant :
  TST ...
```

Nous avons pensé préférable de contracter le test et le branchement au sein d'une même instruction, de manière à optimiser cette utilisation du prédicat "tester". La structure de test de calculabilité que nous voulons mettre en œuvre est la suivante :

- indication du *vecteur d'échec*, qui est l'adresse d'échappement si le processus s'avère non calculable
- pour chaque entrée : test et déroutement si échec

Le vecteur d'échec sera placé dans une position mémoire conventionnellement d'adresse $00. Une instruction EXIT <adr> est fournie pour la positionner. Le test d'entrée et le déroutement sont réalisés par une instruction TRY[1] <adr_canal>.

```
TRY  <adr_canal> :
  si présence[adr_canal] = 0 alors PC := Mem[0]
```

L'exemple qui suit montre comment l'instruction TRY est généralement mise en œuvre. Il consiste à programmer deux opérateurs d'addition indépendants, prenant chacun leur données sur les canaux logiques synchronisés par requêtes, et envoyant leurs résultats dans des canaux similaires *(fig. 4)*.

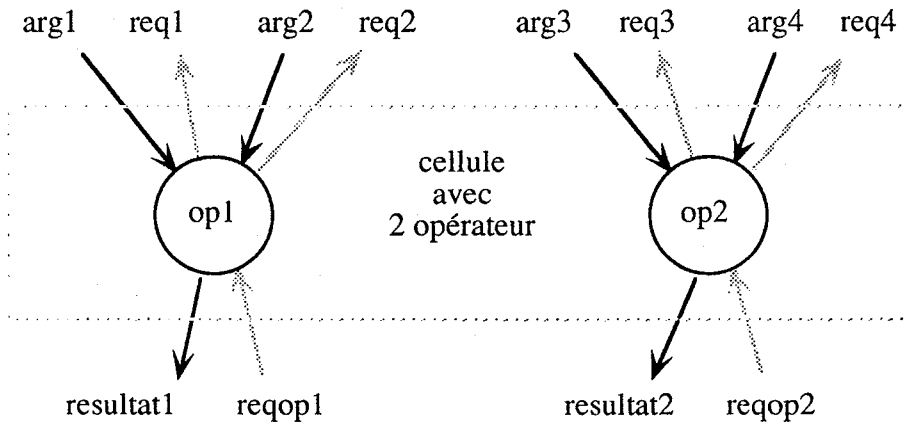


Fig. 4 — *Multiprogrammation de deux opérateurs sur une cellule.*

[1] Le mnémonique "TRY" plutôt que "TST" permet d'éviter la confusion avec le TST habituel du 6800.

```
op1: EXIT  #op2
     TRY   arg1
     TRY   arg2
     TRY   reqop1
     GET   arg1
     SEND  req1
     ADD   arg2
     STA   resultat1
     SEND  resultat1
     SEND  req2
     GET   arg2
     GET   reqop1
op2: EXIT  #op1
     TRY   arg3
     TRY   arg4
     TRY   reqop2
     GET   arg3
     SEND  req3
     ADD   arg4
     STA   resultat2
     SEND  resultat2
     SEND  req4
     GET   arg4
     GET   reqop2
     BRA   op1
```

boucle de scrutation circulaire

Fig. 5 — *Codage de deux opérateurs sur une cellule.*

Cet exemple se code en réalisant une boucle de détermination de la calculabilité des processus (données présentes et résultat demandé), qui revient à faire une sorte de scrutation circulaire pour chercher un processus exécutable *(fig. 5)*. Le code qui l'implémente est mis en évidence par un fond grisé.

Signalons enfin qu'un indicateur a été ajouté au registre d'état, mais il s'agit de l'indicateur S (Send buffer) qui reflète d'état du tampon de sortie. Il est à zéro si le tampon d'émission est vide, et que donc le processeur peut y ranger un message. Il permettra éventuellement d'aiguiller le processeur vers un travail utile dans l'attente de pouvoir émettre.

3.2.2. Structure générale.

Nous sommes partis d'un chemin de données aussi simple que classique, architecturé autour d'un bus d'adresses et d'un bus de données. La figure 6 le schématise pour l'essentiel, bien que certains éléments doivent être encore ajoutés pour que toutes les instructions qui vont être présentées puissent être exécutées.

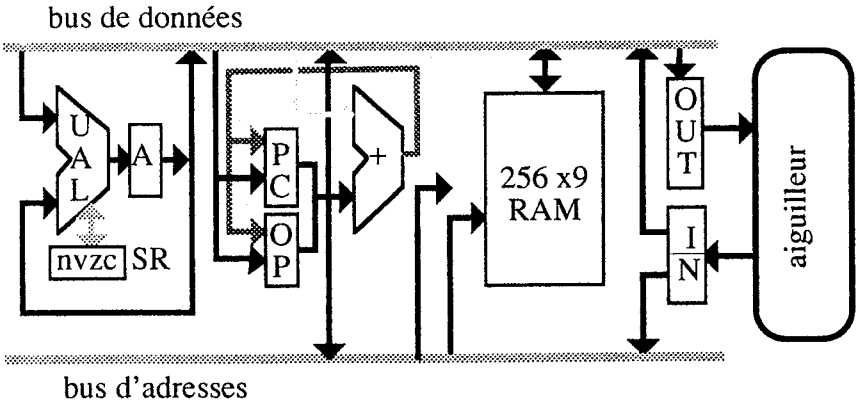


Fig. 6 — *Chemin de données initial.*

La partie calcul est composée d'un accumulateur et d'une UAL de 8 bits de front, et des indicateurs habituels N (résultat négatif), V (débordement), Z (résultat nul) et C (retenue entrante et sortante). Le registre OP sert à mémoriser l'adresse de l'opérande lors de d'exécution d'instructions avec adressage absolu. Au niveau communication, le processeur voit le tampon de sortie comme une FIFO de trois octets, et le tampon d'entrée comme un registre de données et un registre d'adresses (lors du processus de rangement en mémoire, le premier est rangé à l'adresse contenue dans le second). La mémoire est une RAM à 9 bits de front, le neuvième bit n'étant que le vecteur de présence introduit pour l'implémentation des instructions de communication.

3.2.3. Jeu d'instructions initial.

Le jeu d'instructions initial *(tableau 1)* est un pot-pourri de plusieurs jeux d'instructions, principalement ceux du 6502 et du 6800.

Les modes d'adressage ont été ramenés au minimum : adressage "accumulateur" pour les opérations unaires, adressage absolu et immédiat pour les opérations binaires.

Les instructions de calcul binaires sont celles présentes sur le 6800, où addition et soustraction peuvent prendre deux formes selon que l'on prend en compte ou non une retenue initiale. Elles impliquent toutes l'accumulateur comme l'une des opérandes et comme destinataire du résultat. Les instructions unaires comportent les décalages du 6502 et une instruction de complément à 1, qui toutes portent sur l'accumulateur. Toutes ces opérations positionnent des indicateurs du registre d'état, conformément aux usages.

Ces indicateurs sont exploités par l'instruction de branchement conditionnel B*cc* et par l'instruction de positionnement conditionnel S*cc* (utilisé pour les évaluations de conditions complexes). *cc* est un code condition conforme à ceux définis dans le 6800 *(tableau 2)*, avec deux nouvelles conditions,OE et OF, qui permettent de tester l'état du tampon de sortie : si le tampon est plein, une action peut être engagée prioritairement à l'émission.

mnémo	instruction	modes	opération	flags
LDA	Chargement de l'accumulateur	abs,imm	A:=D	nz
STA	Rangement de l'accumulateur en mémoire	absolu	M[adr]:=A	
ADD	Addition	abs,imm	A:=A+D	nvzc
ADC	Addition avec retenue initiale	abs,imm	A:=A+D+c	nvzc
SUB	Soustraction	abs,imm	A:=A-D	nvzc
SBC	Soustraction avec emprunt	abs,imm	A:=A-D-c	nvzc
CMP	Comparaison	abs,imm	A-D	nvzc
AND	Et bit-à-bit	abs,imm	A:=A & D	nz
OR	Ou bit-à-bit	abs,imm	A:=A | D	nz
XOR	Ou exclusif bit-à-bit	abs,imm	A:=A ^ D	nz
NOT	Complément à 1	acc	A:= ~A	nz
ASL	Décalage arithmétique à gauche	acc	C ← A ← 0	nvzc
LSR	Décalage à droite	acc	0 → A → C	nzc
ROL	Rotation à gauche avec la retenue	acc	C ← A ←	nzc
ROR	Rotation à droite avec la retenue	acc	→ A → C	nzc
CLC	Mise à 0 de la retenue	implicite	c:=0	c
SEC	Mise à 1 de la retenue	implicite	c:=1	c
SEND	Envoi de message	abs	tq s attendre OUT:=M[adr…adr+2] s:=1	s
GET	Réception bloquante	abs	tq ¬présent attendre A:=M[adr] présent:=0	
PUT	Envoi d'un message interne	abs	M[adr]:=A, présent =1	
TRY	Test d'un canal	abs	PC:=M[0] si msg absent	
EXIT	Positionnement du vecteur d'échec de TRY	abs	M[0]:=adr	
Bcc	Branchement conditionnel	relatif	PC:=PC+dep si cc=vrai	
Scc	Positionnement conditionnel	acc	A:= si cc alors -1 sinon 0	
JMP	Branchement	abs	PC:=adr	
JSR	Appel de sous-programme	abs	M[1]:=PC+2, PC:=adr	
RTS	Retour de sous-programme	implicite	PC:=M[1]	

Tableau 1 — *Jeu d'instructions initial.*

cc	nom	après CMP #val	décomposition
CC	Carry Clear	A ≥ val (non signé)	C = 0
CS	Carry Set	A < val (non signé)	C = 1
EQ	EQual	A = val	Z= 1
NE	Not Equal	A ≠ val	N = 0
GE	Greater or Equal	A ≥ val (signé)	N ⊕ V = 0
GT	Greater Than	A > val (signé)	Z + (N ⊕ V) = 0
HI	Higher	A > val (non signé)	N ⊕ V = 1
LE	Less or Equal	A ≤ val (signé)	Z + (N ⊕ V) = 1
LS	Less or Same	A ≤ val (non signé)	C + Z = 1
LT	Less Than	A < val (signé)	N ⊕ V = 1
MI	MInus		N = 1
PL	PLus		N = 0
VC	oVerflow Clear		V = 0
VS	oVerflow Set		V = 1
SE	Send buffer Empty		S = 0
SF	Send buffer Full		S = 1

Tableau 2 — *Codes conditions.*

3.3. Améliorations introduites.

Cette section présente tous les enseignements concernant le jeu d'instructions que nous avons tiré de l'expérience de la programmation. Il sont regroupés par thèmes.

3.3.1. Modes d'adressage courts.

L'étude des programmes montre que les positions mémoire référencées dans les adressages absolus sont souvent les mêmes, elles sont principalement :

- les canaux d'entrée
- les structures de message pour SEND
- les variables d'état
- les variables temporaires
- les compteurs de boucle et autres variables de contrôle.

Le programme occupe en général une bonne partie de la mémoire de la cellule, et contient donc peu de variables. D'autre part, le jeu d'instructions initial est loin d'utiliser tous les codes instructions disponibles. On est donc en droit de se demander s'il n' y a pas un moyen d'optimiser l'accès aux variables par le biais d'un adressage absolu "court", dans lequel l'argument serait inclus dans le code instruction. La zone accessible par ce mode étant forcément restreinte, il s'agit de bien la choisir.

Habituellement, les modes d'adressage court sont principalement des modes *base + déplacement*. Les cellules ignorant la notion de procédure, nous mettrons plutôt en œuvre un adressage court tel qu'on le trouvait dans le 6502, qui consiste à affecter des

pages à certains usages : les adresses de $00 à $FF pour les variables, celles de $100 à $1FF pour la pile. Il faut ramener ce principe à l'échelle d'une mémoire totale de 256 octets ; les pages que nous manipulerons ne contiendront que 16 octets. Cette taille de page semble un bon compromis entre utilité et nombre de codes instruction utilisés. Le format des instructions devient donc :

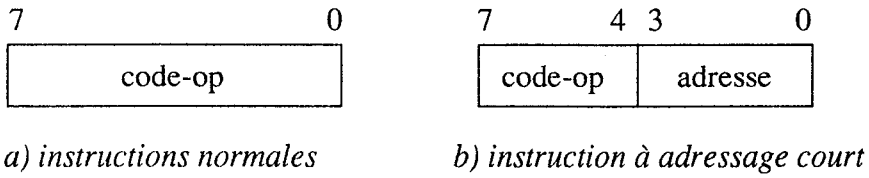


Fig . 7 — *Format des instructions.*

Les exemples contiennent généralement matière à remplir complètement une page de 16 variables, il n'y a donc aucun inconvénient à ce que la page 0 ($00 → $0F) soit réservée à cet usage. Par contre, on peut remarquer qu'il n'y a pas de recouvrement entre les positions référencées par LDA/STA (variables) et celles référencées par GET/PUT/TRY (canaux). On peut donc utiliser une page de variables et une page de canaux distinctes. Cette page de canaux est rarement pleine, mais l'organisation mémoire suivante *(fig. 8)* permet de ne pas rendre inaccessible la portion non utilisée.

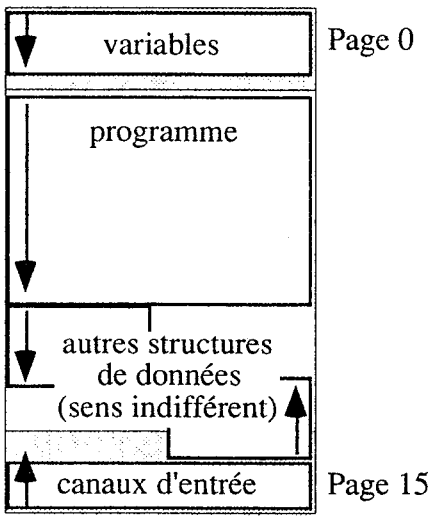


Fig . 8 — *Organisation de la mémoire.*

Les instructions portant sur la page 0 sont LDA et STA, sur la page 15 GET, TRY et PUT. L'adressage court sera noté par adjonction du suffixe "Q" au mnémonique de l'instruction : LDA → LDAQ. Sur l'ensemble de nos exemples, le taux moyen d'utilisation de la page 0 est de 2/3, celui de la page 15 est de 1/5. Globalement, l'adressage absolu court concerne le tiers des instructions, ce qui suffit en soi à le valider. Si on restreint cette estimation aux instructions où l'adressage court est prévu, la grande majorité des occurences du mode absolu sont des formes courtes (tab. 3).

	absolu court	absolu
LDA	97 %	3 %
STA	87 %	13 %
GET	79 %	21 %
PUT	100 % (effectif très faible)	
TRY	97 %	3 %

Tableau 3 — *Pourcentage d'utilisation de la forme courte.*

Ce mode absolu court vient en concurrence au niveau de l'utilisation des codes instructions avec d'autres formes courtes : on peut imaginer des *branchements relatifs courts* ou encore un mode *immédiat court.*

L'idée de branchement relatif court se heurte à plusieurs problèmes. Supposons que nous adoptions une portée courte de [-8,+7], le nombre de codes instructions utilisés est égal à 16 * nb(codes conditions) ; comme nous avons 16 codes conditions, tout l'espace de codage serait utilisé. Nous pourrions privilégier une condition. Choisir le branchement inconditionnel BRA n'est pas intéressant, car cette instruction sert très souvent à réaliser des boucles infinies externes (cycles) dont le corps est assez grand, et ne relève donc pas d'un codage court. Le meilleur choix paraîtrait être BNE, très utilisé pour la réalisation de boucles FOR à indice décrémenté (boucles d'émission, de parcours de table). Dans ce cas, il est évident que les branchements à une distance proche de zéro ne seront jamais utilisés (une boucle doit contenir quelques instructions), en conséquence les codes instructions correspondants sont virtuellement gaspillés. D'autre part, nous verrons qu'il est préférable d'avoir des branchements conditionnels non-relatifs (économie d'un additionneur), ce qui est incompatible avec la notion de branchement court.

Le mode immédiat court, un moment retenu pour LDA, ADD et SUB, n'est pas à l'usage très intéressant. Les valeurs immédiates manipulées sont soit des adresses, soit les constantes 0 et 1. Les autres valeurs de constantes sont marginales et concernent pour l'essentiel des initialisations de compteurs de boucle. Le mode absolu court permet une approximation de LDAQ immédiat : le temps d'exécution de LDAQ *abs* est le même que LDA *#immédiat*, mais il suffit qu'une constante soit utilisée deux fois dans un programme pour qu'il y ait un gain de compacité ; lorsque la constante n'est utilisée qu'une seule fois, le procédé peut être intéressant pour compléter la page 0, et ainsi réduire la taille de la zone programme. ADDQ # et SUBQ # sont très mal adaptées, car les opérations arithmétiques avec un argument immédiat sont presque toujours des décrémentations de compteurs de boucle ou des incrémentations de tag, données qui ne sont jamais présentes dans l'accumulateur naturellement. Des instructions unaires spécifiques sont de beaucoup préférables : INC adr et DEC adr. La dernière utilisation notable de l'adressage immédiat est la comparaison à zéro qui, elle aussi, peut être remplacée à un moindre coût par une opération unaire spécifique TST *adr*.

L'adressage absolu court apparaît donc bien comme la seule forme courte réellement rentable dans notre contexte. Toutefois le choix des instructions sur lequel il porte reste sujet à débattre. Les instructions LDA, STA, GET, PUT et TRY qui ont été retenues pour la première version figée de jeu d'instructions ne semblent en effet pas être le meilleur choix, ainsi que le montre le tableau suivant :

instruction	% d'occurences
STA	17 %
LDA	14 %
SEND	9 %
GET	6 %
DEC	2,8 %
INC	2,3 %
CMP	2,3 %
TRY	1,8 %
autres	< 1,5 %

Tableau 4 — *Occurence du mode d'adressage absolu, par instruction.*

L'instruction SEND est une mauvaise candidate, en dépit de sa fréquence, car une bonne part des messages ne peut être rangée en page 0. L'instruction PUT est d'occurence extrêmement faible ; l'introduction de sa forme courte, essentiellement par souci de cohérence, est une erreur à laquelle il faudra pallier. En revanche, l'instruction TRY, malgré une relativement faible fréquence, est une instruction qui a tendance à se trouver sur le chemin critique des programmes (elle sert à déterminer si un processus peut être exécuté), et nous avons donc tout intérêt à l'optimiser. Des formes courtes de DEC et de INC pourrait avantageusement être introduites, car si statiquement les occurences de DEC sont nombreuses, dynamiquement elles le seront encore plus, l'instruction étant de manière privilégiée placée dans des boucles.

Dans une architecture à accumulateur, il est clair qu'un adressage absolu court tel que nous venons de décrire joue le rôle d'un banc de registres banalisé, à un coût restreint mais avec une généralité et des performances moindres.

3.3.2. Modes d'adressage indirects.

Nous avons vu quelles sont les arguments qui nous poussent à abandonner la plupart des modes d'adressages complexes, notamment les modes indirects et leur dérivés. Cependant cette position n'a pas résisté à l'expérience, qui montre que les cellules manipulent des structures de données purement internes, ainsi que l'atteste le tableau 5.

Programme	structures manipulées
Conway2D	table des voisins, table de transition
Conway1D, Lattice Gaz Model 1D	FIFO de sites, table de transition
Distance entre mots	-
Réseau neuronal, Réseau neuro-synaptique	table des entrées, table des poids, table des sorties
Multiplication de matrices creuses	-
Plus court chemin	table des arcs entrants, table des arcs sortants, table des poids
Crible d'Erathostène	-
Tris de chaines par insertion et par interclassement	tampons de chaîne
Tris par échanges de paires	-
Simulation logique à échéancier	table des entrées et des sorties
Huit reines	échiquiers
Programmes LUSTRE	-

Tableau 5 — *Structures de données internes aux cellules.*

Certes, ces programmes qui nécessitent des structures internes peuvent être transformés de façon à distribuer celles-ci, mais plusieurs facteurs nous en dissuadent :

- Les processus deviennent plus petits et occasionnent une sous-utilisation des cellules ; un regroupement de plusieurs processus peut conduire à regrouper au sein d'une cellule les structures distribuées, non pas au niveau des données mais au niveau du code, avec beaucoup de redondance (c'est une sorte d'expansion en ligne des boucles de parcours des structures).
- L'objet de base manipulé peut s'avérer être un objet complexe, comme c'est le cas pour les tris de chaînes par exemple.

Les structures de données rencontrées sont essentiellement des tables ; un mode d'adressage indirect serait donc bien adapté, avec l'avantage d'être polyvalent et d'un faible coût matériel. Un tel mode peut toutefois revêtir des formes bien différentes que nous pouvons classer en modes *indirect mémoire* (l'adresse de l'opérande se trouve dans une position mémoire) et modes *indirect registre* (l'adresse se trouve dans un registre interne). Le choix d'un mode indirect registre a reposé dans un premier temps sur des considérations de performances (un accès mémoire en moins), allié au fait que dans la plupart des cas, un seul index est suffisant (l'accès simultané à plusieurs structures, qui aurait justifié plusieurs index, n'a pas été rencontré).

Le registre d'indirection I s'insère assez naturellement dans le chemin de données, à côté du compteur ordinal et du champ adresse du registre instruction. La présence d'un additionneur permet de réaliser facilement une instruction d'incrémentation de I,

INCI. Paradoxalement, cette instruction n'a pas été retenue en raison de sa trop forte occurence. En effet, si nous examinons comment l'adressage indirect va prendre place dans les programmes, nous nous apercevons que la notion de parcours séquentiel recouvre pratiquement l'essentiel des accès indirects, l'accès aléatoire n'apparaissant que beaucoup plus rarement (consultation de tables de transition dans les automates, réalisation de FIFO). Lors de ces parcours séquentiels, l'indirection est souvent suivie d'une incrémentation de I, ce qui, compte tenu de la fréquence de ces accès (les accès indirect post-incrémentés représentent 5% des instructions), rend l'instruction INCI encombrante. La solution généralement mise en œuvre dans les processeurs classiques consiste à fusionner indirection et incrémentation du compteur au sein d'un mode d'adressage distinct dit *indirect post-incrémenté*. L'adressage symétrique pré-décrémenté ne sert que pour la réalisation de piles, structures peu courantes dans notre contexte (pas de récursivité ni de structuration procédurale).

Nous voilà donc en train d'ajouter non pas un mais deux modes d'adressages. Par bonheur, l'adressage indirect post-incrémenté ne nécessite aucun nouvel élément dans le chemin de données (l'incrémentation de I se fait par les mêmes dispositifs que celle du compteur ordinal). L'adressage indirect simple sera noté (I) et le post-incrémenté (I++).

L'adressage indirect est nécessaire pour manipuler les structures de données, mais sous la forme que nous venons de définir, il peut se révéler utile dans le cas général. Pour s'en rendre compte, il faut d'abord analyser en détail comment le parcours de table va se programmer. Supposons que nous ayons à programmer une cellule pour effectuer une opération de distribution d'une même valeur vers plusieurs autres cellules, avec une synchronisation par requêtes.

```
ReqAval : tableau [1..N] de canaux d'entrée
AdrAval : tableau [1..N] d'adresses relatives
TagAval : tableau [1..N] d'adresses
Msg,ReqAmont : Message
ValAmont : canal d'entrée

Envoyer(ReqAmont)
CYCLE
  valeur:=Recevoir(ValAmont)
  Envoyer(ReqAmont)
  POUR i:= 1 A N FAIRE
     Msg.donnée:=valeur
     Msg.tag:=TagAval[i]
     Msg.adr:=AdrAval[i]
     Recevoir(ReqAval[i])
     Envoyer(msg)
```

Accéder aux tableaux comme le ferait un compilateur (avec calcul systématique de l'adresse de l'élément référencé) serait tout à fait désastreux car le registre I passerait son temps à sauter d'une table à l'autre. En entrelaçant les différentes tables, nous pouvons faire en sorte que les données soient rangées dans l'ordre où elles seront accédées :

```
Aval : tableau [1..N] de
  struct
     tag : adresse
     adr : adresse relative
     req : canal d'entrée
Msg,ReqAmont : Message
ValAmont : canal d'entrée

Envoyer(ReqAmont)
CYCLE
  valeur:=Recevoir(ValAmont)
  Envoyer(ReqAmont)
  POUR i:= 1 A N FAIRE
     Msg.donnée:=valeur
     Msg.tag:=Aval[i].tag
     Msg.adr:=Aval[i].adr
     Recevoir(Aval[i].req)
     Envoyer(msg)
```

algorithme dont le corps de boucle peut s'écrire :

```
; I pointe sur Aval[i]
LDA  valeur
STA  msg
LDA  (I++)
STA  msg+1
LDA  (I++)
STA  msg+2
GET  (I++)
SEND msg
; I pointe sur Aval[i+1]
```

Cette forme est encore imparfaite : pourquoi composer le message de toutes pièces alors qu'il n'est pas tellement plus coûteux, en terme de place mémoire, de stocker une *table de messages* (3N contre 2N octets). Le champ "donnée" du message dans la table est non significatif jusqu'à ce qu'il soit affecté. Si on met à profit ce fait pour se servir de ce champs comme réceptacle des requêtes de l'aval, nous obtenons des tables de tailles identiques.

```
Msg : tableau [1..N] de
  struct
     union
        req : canal d'entrée
        donnée : octet
     tag : adresse
     adr : adresse relative
ReqAmont : Message
ValAmont : canal d'entrée

Envoyer(ReqAmont)
CYCLE
  valeur:=Recevoir(ValAmont)
  Envoyer(ReqAmont)
  POUR i:= 1 A N FAIRE
     Msg[i].donnée:=valeur
     Recevoir(Msg[i].req)
     Envoyer(Msg[i])
```

ce qui donne en assembleur :

```
; I pointe sur Msg
GET   (I)
LDA   valeur
STA   (I)
SEND  (I++)
; I pointe sur Msg[i+1]
```

Cette notion de parcours des données, liée dans l'exemple qui précède à la structure tabulaire, ne peut-elle être étendue au cas général ? Cela reviendrait à créer deux flux séparés, l'un de données, l'autre d'instructions, au lieu de les mêler comme on le fait avec un adressage absolu. L'exemple peut être optimisé en incluant les références au producteur dans la liste des positions parcourues par le registre I.

```
T : struct
  valAmont : canal d'entrée
  reqAmont : Message
  tableau [1..N] de
     struct
        union
           req : canal d'entrée
           donnée : octet
        tag : adresse
        adr : adresse relative

...
```

L'adressage absolu est alors seulement utilisé pour le code de contrôle, les variables d'état globales et les initialisations (envoi des premières requêtes). Il est souvent possible de définir un ordre de parcours séquentiel sur une bonne part des références d'un programme, même s'il ne présente aucune structure de données interne. Il suffit alors de réordonner les données pour qu'elles "se présentent" naturellement sous le registre I. Pour preuve, nous avons programmé le calcul de distance entre mots avec et sans mode d'adressage indirect. Ce programme systolique est l'archétype du programme purement statique, sans structure de données interne, et représente sans doute l'un des cas les plus défavorable. La version avec adressage indirect permet :

- un gain de compacité de 10%
- un gain de performance de 6%

Ces chiffres se basent sur une version "mode absolu" qui fait un usage intensif du mode d'adressage absolu court, déjà fortement optimisé. La comparaison avec une version utilisant un mode absolu normal (instructions absolues toutes codées sur deux octets) serait encore plus favorable.

Pour exploiter les modes indirects, il faut aussi fournir quelques instructions de contrôle, assurant les fonctionnalités de chargement, de sauvegarde, de mouvement de I et de comparaison avec une valeur. La programmation des exemples a permis de préciser les tâches à accomplir. Pour le parcours de table, il faut :

- positionner I en début de table
- sauver et restaurer I depuis une position mémoire en page zéro (lorsque plusieurs registres d'indirection sont nécessaires)
- tester l'arrivée en fin de table

Pour une gestion de FIFO, il faut en plus :

- comparer I aux bornes du tableau qui sert de support de mémorisation de la structure.

Pour une consultation de table :

- copier une valeur calculée de A dans I

Le mouvement de I est pratiquement toujours assuré par le mode d'adressage indirect post-incrémenté. Une incrémentation simple de I peut être réalisée par LDA (I++) si la destruction du contenu de l'accumulateur n'est pas gênante. Nous avons donc potentiellement besoin de :

LDI	immédiat	STI	absolu court
LDI	absolu	TAI	(transfert de A dans I)
LDI	absolu court (page zéro)	TIA	(transfert de I dans A)
STI	absolu	CPI	immédiat

Les instructions de transfert de et vers l'accumulateur sont indispensables car ce sont des primitives simples à implémenter et qui peuvent permettre de construire à peu près n'importe quelle fonctionnalité de manipulation de I. Les autres instructions doivent être soumises à discussion.

En raison du nombre de codes instruction nécessaires, il ne peut être question de retenir les instructions LDI et STI pour le mode absolu court. LDI et STI correspondent respectivement à LDA + TAI et TIA + STA. Nous pourrions donc nous en passer, à condition que la destruction de l'accumulateur ne soit pas parfois gênante. STI, d'usage rare, n'a pas été retenu, mais LDI l'a été. Rétrospectivement, il semble que cela ait été une erreur ; l'utilité de LDI ne justifie pas le coût de contrôle qu'elle va forcément induire :

- le contenu de l'accumulateur n'est apparu significatif au moment de l'exécution de LDI dans aucun de nos exemples
- la faiblesse d'occurence de LDI (3%) minimise le gain de place et de performance.

L'instruction CPI sert à tester l'arrivée en fin de table dans deux cas :

- la réalisation de FIFO (lorsqu'un pointeur arrive à la fin du tableau, il faut le ramener au début) et la réalisation de boucle d'émission ; dans ce cas CPI *immédiat* est bien adapté, mais l'occurence de cette situation est restreinte
- la réalisation de boucles d'émission de données ; dans ce cas CPI n'est pas adapté car la zone émise ne se trouve pas forcément à une adresse constante en mémoire, l'adresse de fin est relative à celle de début, et la taille de la zone à émettre peut être variable (traitement de chaînes de caractères)
- parcours de table d'entrées et de sortie ; CPI *#immédiat* est mal adapté car la borne varie avec la topologie du graphe logique.

Si nous regardons maintenant l'incidence de CPI au niveau du chemin de données, nous nous apercevons que cette instruction s'y insère mal, le registre I n'étant pas intégré au bloc UAL-accumulateur. Son exécution serait donc inefficiente et coûteuse d'un point de vue contrôle dans la mesure où elle est difficilement regroupable avec d'autres instructions. Comme substitut, nous avons préféré distinguer deux situations : le comptage, où un compteur de boucle est utilisé (instruction DEC, présentée plus loin), et le test de fin de tableau pour les FIFO, où I est transféré dans A pour être ensuite testé.

Les adressages indirects contribuent fortement à augmenter les fonctionnalités des cellules, à un coût assez modeste. Ils sont indispensables pour beaucoup de programmes, et permettent dans tous les cas un gain de performances et de compacité. Statiquement, ils représentent 12% des instructions, contre 34% pour le mode d'adressage absolu court, et 31% pour le mode absolu.

3.3.4. Calcul

Il est apparu nécessaire d'effectuer un certain nombre de modifications et d'aménagements dans les instructions de calcul initialement retenues. Si le choix des opérations binaires n'a pas été remis en cause, celui des opérations unaires n'est pas, semble t-il, le plus judicieux. Voici les changements apportés :

- décalages : l'instruction LSR (décalage à droite avec introduction d'un zéro) a été remplacée par ASR (décalage à droite avec conservation du bit de signe). Cette instruction, qui peut réaliser une division par 2 signée, est plus difficile à simuler avec ROL et ROR que LSR, qui peut se remplacer par CLC suivi de ROR.
- La structure du bloc UAL-accumulateur a été modifiée pour des raisons d'implémentation ; l'entrée de l'accumulateur est maintenant directement connectée au bus de données, alors qu'il était jusqu'à présent chargé au travers de l'UAL. Il était dès lors plus simple d'avoir une instruction LDA qui ne positionne pas le registre d'état. Une instruction TST a donc été ajoutée pour retrouver cette fonctionnalité de test précédemment dévolue à LDA. TST est équivalent à CMP #0.
- La programmation a montré que les instructions unaires portent sur des variables en mémoire plutôt que sur le contenu "naturel" de l'accumulateur (présence constante de la séquence LDA var - Op unaire - STA var). Au mode d'adressage "accumulateur" des instructions unaires, on a donc ajouté d'autres modes : absolu, indirect et indirect post-incrémenté. Cette modification, une fois entérinée au niveau des programmes, a permis d'observer une disparition quasi-totale des opérations unaires sur l'accumulateur (absolu : 84%, accumulateur : 10%, indirects : 6%).
- Les instructions DEC, INC, CLR (liées à l'absence de mode immédiat court), NEG (opposé) et NGC (opposé en multiprécision) ont été introduites.

Plus globalement, les capacités de calcul des cellules sont apparues trop réduites, tant au niveau de la quantité de code nécessaire pour coder les programmes que dans le rapport de puissance avec les fonctionnalités de communication, quoiqu'un tel jugement soit forcément subjectif. A partir du bloc UAL-accumulateur, il est très facile de réaliser une multiplication microcodée, à condition d'étendre l'accumulateur à 16 bits *(fig. 9)*.

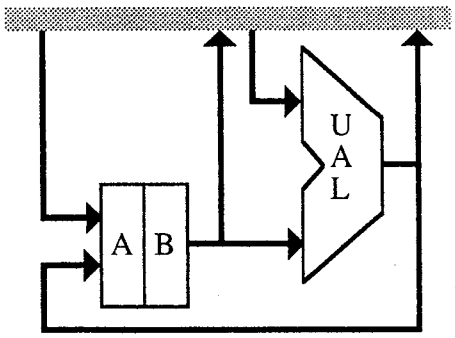


Fig. 9 — *Nouvelle structure du bloc UAL-accumulateur.*

```
A:=M[opd1]
B:=0
pour cpt:= 1 à 8
  si A[0] = 1
     B:=B+M[opd2]
  décaler A,B à droite
```

Le coût minimum est donc d'un registre 8 bit et d'un compteur 1→8. Il faut ensuite déterminer la forme la plus intéressante pour cette opération :

MUL A,adr	A,B:= A * M[adr]
MUL adr	A,B:= M[adr] * M[adr+1]
MUL adr1,adr2	A,B:= M[adr1] * M[adr2]

Le meilleur choix dépend du contexte d'utilisation. Il est rare d'utiliser la multiplication sur des données 8 bits, on doit donc en composer plusieurs pour réaliser des multiplications de plus grande amplitude. Une multiplication 16 bits de A=A1*256 + A0 par B=B1*256 + B0 peut se décomposer en :

A1*B1*65536 + (A1*B0 + A0*B1)*256 + A0*B0

On doit donc multiplier chacun des octets de A par chacun des octets de B. La forme MUL adr1,adr2 est donc la plus appropriée.

Enfin, il faut définir un moyen de manipulation de l'extension d'accumulateur (B). Plutôt que de s'en tenir à un load/store B, nous avons préféré mettre pleinement à profit le couple A,B, afin de doter la cellule de capacités de calcul 16 bits :

ADCW	addition avec retenue de A,B avec une donnée 16 bits
SBCW	soustraction avec emprunt de A,B avec une donnée 16 bits
LDAW	chargement de A,B avec une donnée 16 bits
STAW	rangement de A,B avec une donnée 16 bits

Ces opérations sont décomposées par microcode en opérations 8 bits. Avec MUL adr,adr et ces instructions 16 bits, une multiplication 16 bits par 16 bits avec résultat sur 32 bits peut s'écrire :

```
A:    DC ...,...    ; ordre : poid fort, poid faible
B:    DC ...,...
R:    DS 4

      MUL    A+1,B+1
      STAW   R+1
      MUL    A,B
      STAW   R+3
      MUL    A+1,B
      CLC
      ADCW   R+2 ; pas de retenue
      STAW   R+2
      MUL    A,B+1
      CLC
      ADCW   R+2
      STAW   R+2
      BCC    L1
      INC    R+3
L1:
```

Nous n'avons par contre pas jugé rentable d'introduire une division microcodée : c'est une opération moins utilisée, qui se combine difficilement pour la réalisation de la même opération sur des données de plus grande amplitude.

Ainsi modifiée, la cellule est capable de prendre en charge une opération d'addition (164 octets) ou de multiplication (184 octets) en virgule flottante (simple précision).

3.3.5. Communication.

L'indicateur "S", qui reflète l'état du tampon de sortie, devait initialement permettre de fournir un travail utile au processeur en que celui-ci puisse attendant émettre. Cet indicateur est inexploitable en pratique, car l'attente avant la libération du tampon de sortie n'est jamais longue, et le code de contrôle que nous serions forcés d'introduire nous ferait perdre le faible gain de performance obtenu. Il a donc été très rapidement abandonné.

La forme de la fonctionnalité de test de présence d'une donnée sur un canal (TRY) a été un peu revue deux fois :

1°/ L'instruction EXIT, qui positionne le vecteur d'échec (Mem[0]), est une instruction spécifique qui peut aisément se remplacer par LDA #adr, STAQ 0, séquence qui n'est guère moins longue (3 octets au lieu de 2) et guère moins rapide (4 cycles au lieu de 3). Son maintien au sein du jeu d'instructions ne se justifie donc pas.

2°/ L'accumulateur ne contient jamais de donnée significative au moment d'un test de calculabilité, car ce registre est trop temporaire pour qu'on l'utilise comme variable de communication inter-processus. L'accumulateur est donc disponible pour contenir le vecteur d'échec (TRY ne modifie par l'accumulateur). Cette optimisation est rentable, car le vecteur est plus facile à positionner (LDA #) et l'instruction TRY est plus simple (transfert de A dans PC sur échec).

3.3.6. Instructions de contrôle.

Les branchements conditionnels du jeu d'instructions initial spécifient la cible relativement au compteur ordinal de manière à exploiter la localité de référence, suivant ainsi la tradition. Mais notre contexte n'est pas habituel : l'adresse relative est aussi encombrante qu'une adresse absolue, mais nécessite un additionneur complet au niveau du compteur ordinal. Les branchements relatifs permettre d'écrire du code relogeable, mais cette caractéristique ne nous est d'aucun intérêt. Nous transformerons donc les branchements relatifs en branchements absolus.

La suppression des conditions portant sur l'indicateur S permet de réintégrer le saut inconditionnel (BRA) au sein des branchements conditionnels (qui pour des raisons de codage doivent être limités à 16). Dès lors, l'instruction JMP peut être supprimée.

L'étude préliminaire des fonctionnalités requises au niveau des instructions de contrôle a mis en évidence le faible intérêt que présente la notion de sous-programme dans le contexte du réseau cellulaire. Le principe d'un seul niveau d'appel avait tout de même été avancé, pour ne pas se priver de la possibilité de réaliser des "routines de service" : sous-programmes de multiplication, de division, d'émission.

La fréquence observée de programmation de telles routines est faible : elles n'apparaissent que dans les applications manipulant des chaînes de caractères (tri par interclassement et par insertion). Les instructions JSR et RTS, qui utilisent la position mémoire 0 pour sauvegarder l'adresse de retour, restent des instructions complexes à implémenter et ne se justifient pas par leur rendement. On peut réduire le contenu sémantique des instructions d'appel et de retour en les remplaçant par TPCA (transfert du compteur ordinal dans l'accumulateur) et TAPC (opération inverse). Les séquences d'appel et de retour deviennent, toujours en utilisant la position mémoire 0 :

```
appelant :                    appelé :
TPCA
STAQ  0
BRA   sub
                        sub:  « corps »
                              LDAQ  0
                              ADD   #2
                              TAPC
```

Ces nouvelles instructions, plus simples à implémenter, sont moins efficaces mais plus générales : TAPC peut permettre de programmer des structures de type CASE OF avec branchement tabulé ou calculé :

```
; A = index
ADD   #tab
TAI
LDA   (I)
TAPC
```

Bien que retenue lorsque le jeu d'instructions a été figé pour la version 1, elles n'apparaissent pas en définitive comme indispensables :

a) TPCA n'est pas utile, dans la mesure où le code est statique : le contenu du compteur ordinal obtenu par TPCA est une constante que l'on peut aussi bien donner de façon littérale.

b) TAPC peut être remplacé par une modification dynamique de l'argument d'un branchement inconditionnel, ou mieux par une utilisation détournée de l'instruction TRY: il suffit de chercher l'adresse de retour avec l'accumulateur (LDAQ retour) et exécuter un TRY sur une adresse ne correspondant à aucun canal ; l'échec et le branchement sont alors assurés. La séquence d'appel et de retour reviendrait donc :

```
        appelant :                      appelé :

        LDA   #retour
        STAQ  0
        BRA   sub
retour:                     sub:        « corps »
                                        LDAQ  0
                                        TRYQ  canal_vide
```

L'instruction S*cc* (positionnement conditionnel de l'accumulateur) ne s'est jamais révélée ni indispensable, ni même intéressante. Elle sert habituellement à évaluer des expressions logiques complexes, mais celles-ci sont rarissimes dans notre contexte. Etant donné le nombre de codes instructions qu'elle occupe, nous l'avons très rapidement abandonnée.

INSTITUT IMAG
Informatique, Mathématiques Appliquées de Grenoble
CNRS-INPG-USMG
MÉDIATHÈQUE
38041 GRENOBLE CEDEX
FRANCE
Tél. 76.51.46.36

3.4. Définition de la version 1.

En cours d'étude, nous avons été obligé de figer une version de cellule, afin de permettre la mise en chantier rapide d'un prototype de circuit. C'est sur la base de ce jeu d'instructions que les évaluations ont été faites, notamment les mesures sur le système de communication présentées dans le chapitre II.

3.4.1. Chemin de données.

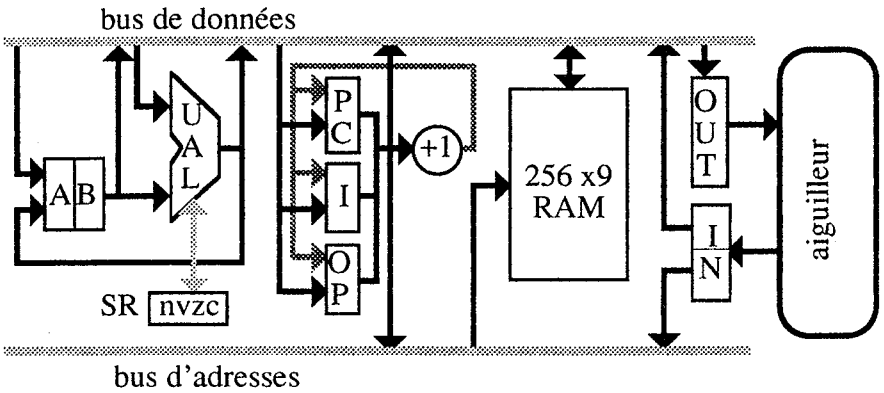


Fig. 10 — *Chemin de données correspondant au nouveau jeu d'instructions.*

Le nouveau chemin de donnée est un peu plus complexe : sont venu s'y insérer le registre d'indirection I et l'extension d'accumulateur B, ainsi qu'un petit compteur de boucle de 1 à 8 pour la multiplication (non figuré). L'additionneur qui existait au niveau du compteur ordinal a été remplacé par un simple incrémenteur.

3.4.2. Jeu d'instructions.

La table présentée page suivante résume le jeu d'instructions figé. Les modifications qu'il faudrait y apporter pour prendre en compte les derniers résultats de l'étude sont : des formes courtes (absolu page 0) pour DEC et INC, la suppression de TPCA, TAPC, et de LDI, ainsi que de la forme courte de PUT.

mnémo	instruction	modes	opération	flags
LDA	Chargement de l'accumulateur	abs,imm,pg 0, (I),(I)+	A:=D	nz
STA	Rangement de l'accumulateur en mémoire	abs,pg 0, (I),(I)+	M[adr]:=A	
ADD	Addition	abs,imm,(I),(I)+	A:=A+D	nvzc
ADC	Addition avec retenue initiale		A:=A+D+c	nvzc
SUB	Soustraction	"	A:=A-D	nvzc
SBC	Soustraction avec emprunt	"	A:=A-D-c	nvzc
CMP	Comparaison	"	SR:=f(A-D)	nvzc
AND	Et bit-à-bit	"	A:=A & D	nz
OR	Ou bit-à-bit	"	A:=A | D	nz
XOR	Ou exclusif bit-à-bit	"	A:=A ^ D	nz
NOT	Complément à 1	abs,acc,(I),(I)+	D:= ~D	nz
NEG	Opposé		D:=-D	nvzc
NGC	Opposé avec emprunt	"	D:=-D-c	nvzc
CLR	Remise à zéro	"	D:=0	
DEC	Décrémentation	"	D:=D-1	nvzc
INC	Incérementation	"	D:=D+1	nvzc
TST	Comparaison à 0	"	SR:=f(0-D)	nvzc
ASL	Décalage arithmétique à gauche	"	C ← A ← 0	nvzc
ASR	Décalage arithmétique à droite	"	A → C	nzc
ROL	Rotation à gauche avec la retenue	"	C ← A	nzc
ROR	Rotation à droite avec la retenue	"	A → C	nzc
CLC	Mise à 0 de la retenue	implicite	c:=0	c
SEC	Mise à 1 de la retenue	implicite	c:=1	c
SEND	Envoi de message	abs, (I), (I)+	tq s attendre OUT:=M[adr…adr+2] s:=1	s
GET	Réception bloquante	abs,pg F,(I),(I)+	tq ¬présent attendre A:=M[adr] présent:=0	
PUT	Envoi d'un message interne	"	M[adr]:=A, présent =1	
TRY	Test d'un canal	"	PC:=A si msg absent	
MUL	Multiplication 8x8 bit → 16 bits	ad1,ad2	A,B:=M[ad1]*M[ad2]	-
LDAW	Chargement 16 bits	abs,imm,(I),(I)+	A,B:=D16	
STAW	Rangement 16 bits	abs,(I),(I)+	D16:=A,B	
ADCW	Addition 16 bits	abs,imm,(I),(I)+	A,B:=A,B+D16+c	nvzc
SBCW	Soustraction 16 bits	abs,imm,(I),(I)+	A,B:=A,B-D16-c	nvzc
TAI	Chargement de I	implicite	I:=A	
TIA	Sauvegarge de I	implicite	A:=I	
LDI	Chargement de I	abs,imm	I:=D	
Bcc	Branchement conditionnel	abs	PC:=adr si cc=vrai	
TAPC	Branchement calculé	implicite	PC:=A	
TPCA	Sauvegarge PC	implicite	A:=PC	

Tableau 6 — *Jeu d'instructions figé.*

4. Conclusions.

Nous venons de voir comment l'expérience de la programmation permet d'améliorer notablement l'efficacité et la compacité du code issu du jeu d'instructions. Des particularités contextuelles (taille mémoire faible) ont été mises à profit pour introduire un mode d'adressage absolu court, qui s'est avéré être prédominant par rapport au mode absolu normal. L'étude parallèle des programmes et du jeu d'instructions a démontré la fécondité des modes d'adressages indirects et indirect post-incrémenté : nécessaires pour coder certains programmes (usage de tables), ils permettent très souvent un gain dans le cas général. Les capacités de calcul ont été augmentées notamment par l'introduction d'une multiplication et d'opérations 16 bits microcodées. En revanche, les instructions de contrôle ont été réduites au strict minimum : les branchements conditionnels.

Rétrospectivement, il apparaît que l'expérience de la programmation est vitale pour permettre d'expurger le jeu d'instructions d'un grand nombre de choix maladroits produits par une définition fonctionnelle intuitive/inductive. La faiblesse d'une telle démarche tient dans le choix des programmes, arbitraire, et dans la méthode de codage, essentiellement manuelle, dont les besoins ne correspondent pas forcément à ceux des méthodes compilées.

Par manque de temps, ce travail n'inclut malheureusement pas de d'études comparatives d'autres architectures possibles de processeur (non-8 bits, chemin de données structuré différemment, machines à piles...), études qui seraient pourtant intéressantes.

Chapitre IV : Programmation

Topologie simple, partie traitement rudimentaire, le parallélisme à la portée de tout un chacun, voilà qui est trop beau pour être vrai. La complexité est comme la poussière que l'on cache sous le tapis en balayant : quelqu'un fini toujours par tomber dessus. Le "tapis" de l'architecture parallèle, c'est la programmation.

Nous avons vu dans le chapitre I que le credo du parallélisme massif est l'exploitation de la concurrence partout où elle se trouve, jusqu'à la granularité la plus fine, avec comme conséquence une forte délocalisation du parallélisme. Lorsque les multiprocesseurs ne comportaient que quelques PE, un contrôle centralisé était possible ; il n'était alors question que de découper quelques boucles DO...FOR... en autant de processus qu'il y a de processeurs, c'est à dire au plus une dizaine. Malheureusement, il est impossible mettre en œuvre de cette façon un grand nombre de processeurs, car le coût de contrôle deviendrait énorme.

Cette granularité fine, si séduisante sur un plan théorique, suppose une structure de processus et de communication complexe, mouvante, de grande ampleur, sur laquelle se reporte toute la complexité présente dans les algorithmes séquentiels. Nous devons en conséquence donner au programmeur les moyens de l'appréhender de la manière la moins fastidieuse possible, lui donner une sorte d'équivalent des structures de contrôle conditionnelles et itératives présentes dans les langages procéduraux.

L'histoire à ce sujet est suffisamment explicite pour avoir valeur d'avertissement. Le circuit GAPP et les unités de traitement de la Connection Machine sont des circuits comparables (SIMD synchrone, UAL 1 bit), le GAPP est tombé dans l'oubli rapidement, la Connection Machine connaît un succès d'estime durable qui pourrait bien se transformer en succès commercial. La différence entre les deux était pour une bonne part une différence d'environnement d'exploitation. Le GAPP était un circuit nu dont personne n'avait la moindre idée quant à la mise en œuvre lorsqu'il s'agissait de lui faire faire plus qu'une extraction de contour sur une image bitmap, alors que la Connection Machine est livrée avec des bibliothèques d'opérateurs, des langages parallèles (*LISP, CM-FORTRAN, C), un simulateur. Cette approche qui consiste à fournir une bibliothèque d'opérateurs, de procédures et à l'envelopper dans divers langages hôtes bien connus de l'informaticien moyen est très habile, car les utilisateurs ont ainsi l'impression de se retrouver en terrain familier et n'ont pas un mouvement de recul devant une tâche qui est objectivement difficile. Les résultats ne seront dans un premier temps que très moyennement probants car on ne peut tirer le maximum d'une architecture que si on en connait les points forts et les points faibles, mais au moins la transition peut-elle se faire en douceur : la maîtrise viendra en son temps, avec l'expérience.

Le parallélisme MIMD étant plus général et plus délicat à manipuler, les difficultés se présenteront avec une acuité accrue, et l'objet de ce chapitre sera de voir quels outils, quelles méthodes nous pouvons offrir à l'utilisateur pour l'aider à appréhender et à exploiter ce type d'architecture. Nous commencerons par envisager une approche de la programmation à base de langages parallèles, et nous approfondirons le cas du langage *data-flow* LUSTRE qui a fait l'objet d'une implémentation pour le réseau cellulaire [PAY91]. Puis nous regarderons plus en détail comment effectuer une programmation de bas niveau par expression dans un langage séquentiel du code de chaque cellule, sur la base de quelques exemples concrets. Nous tâcherons enfin d'en tirer les enseignements pour ouvrir sur une alternative à l'usage de langages parallèles à base d'outils de génération automatique.

1. Langages parallèles.

L'approche "langage" du parallélisme présente à priori tous les avantages. Les langages possèdent la faculté de pouvoir être décrits sur le plan syntaxique comme sur le plan sémantique, voire d'être standardisés. Ils assurent une certaine économie de concepts et une certaine cohérence entre ceux-ci (du moins pour les langages modernes dignes de ce nom).

Un programme décrit dans un langage est un objet manipulable, c'est à dire compilable, mais aussi susceptible de recevoir une preuve formelle de correction dans la mesure où le langage a été bien pensé.

Enfin, et ce n'est sans doute pas le moindre de leur intérêt, les langages créent tous au moins un niveau d'abstraction au dessus de l'architecture, et comme ils peuvent être implémentés sur diverses architectures, nous avons là un moyen d'obtenir une indépendance des programmes vis-à-vis de la machine-cible.

Le bât blesse au niveau de leur mise en œuvre effective. Le soin d'établir un lien entre un langage et une architecture matérielle donnée est laissé aux "implémenteurs", qui, eux, voient de moins en moins comment s'y prendre au fur et à mesure que l'architecture à laquelle ils ont affaire s'éloigne du modèle séquentiel. Tout langage ne peut être implémenté sur toute machine, et réciproquement toute architecture ne peut être pleinement exploitée par tout langage.

Les langages susceptibles de donner lieu à un traitement parallèle peuvent relever de quatre catégories :

1°) *Les langages à sémantique séquentielle où le parallélisme sous-jacent est automatiquement repéré et extrait par le compilateur.* Cette extraction est une opération extrêmement délicate car le compilateur doit analyser les dépendances entre opérations (pour paralléliser une boucle, il ne faut pas à l'itération n avoir besoin des résultats de l'itération $n-1$). Cette approche a été mise en œuvre dans les compilateurs FORTRAN destinés aux calculateurs vectoriels du type Cray, qui sont des machines où le parallélisme est essentiellement de type *pipeline*. Le degré de concurrence est limité par le nombre d'étages des opérateurs arithmétiques, par les problèmes de dépendances, ainsi que par le goulot d'étranglement que constitue l'accès à la mémoire. Bien que le sujet n'ait pas encore été étudié au sein de notre équipe, un traitement analogue par compilateur serait possible pour notre réseau cellulaire.

2°) *Les langages à sémantique séquentielle associés à des librairies parallèles.* Cette approche correspond à une solution de facilité, logique dans le cadre des machines SIMD et économique dans celui des calculateurs vectoriels. Dans une machine SIMD, le contrôle d'exécution de bas niveau est pris en charge par un microcontrôleur ou un

dispositif équivalent, le contrôle de haut niveau est assuré par le calculateur hôte. Seule la manipulation des données est réellement dévolue à la partie parallèle. Un découplage entre la programmation de l'hôte et celle du réseau n'est donc pas aberrante, mais elle laisse l'extraction du parallélisme à la charge du programmeur.Le cas des calculateurs vectoriels est un peu différent. L'important *background* disponible en matière de calcul numérique a permit de dégager des fonctionnalités d'usage fréquent qu'il a paru important de standardiser (exemple : les noyaux "BLAS"). Ces fonctionnalités ont été implémentées de façon très soigneuse, éventuellement directement en assembleur, pour une architecture donnée, encapsulées dans des procédures. Comme elles recouvrent une proportion notable des opérations lourdes, l'extraction automatique du parallélisme devient une phase moins critique, voire relativement marginale du processus de compilation. A la limite, que ce soit en SIMD ou en séquentiel avec accélérateur vectoriel, un bon compilateur classique associé à des librairies parallèles bien pensées peut apparaître comme largement suffisant à bien des utilisateurs n'exploitant qu'un parallélisme géométrique. Ces outils, qui peuvent être très rapidement opérationnels, ont les inconvénients de leurs avantages :

- Ils sont faciles à employer mais permettent aux programmeurs de conserver leurs ancestrales habitudes séquentielles, rendent improbable une réelle compréhension par ceux-ci des aspects importants du parallélisme, et enferment les programmes dans le carcan d'outils qui pour être standardisés n'en sont pas pour autant les plus efficaces. L'éclosion d'une compétence en matière de parallélisme est ainsi étouffée dans l'œuf.

- Par contre, leur rigidité fait que le parallélisme mis en œuvre est perçu comme tel par le programmeur et l'est correctement. Ce n'est pas forcément le cas des compilateurs-paralléliseurs automatiques qui font allégrement croire que l'architecture est exploitée à fond alors qu'elle ne l'est pas, à cause de petits détails qui ont échappés à l'attention du programmeur (dépendances entre instructions, etc.)

3°) *Les langages procéduraux étendus par l'introduction de structures de contrôle à sémantique parallèle*. Ces langages, dont OCCAM est le représentant le plus connu, tentent de remédier aux insuffisances des deux premières approches. L'explicitation du parallélisme reste à la charge du programmeur, mais celui ci peut enfin l'exprimer dans un formalisme relativement lisible, cohérent et élémentaire. OCCAM introduit des structures de contrôle de la concurrence et de l'asynchronisme : la structure PAR qui permet de déclarer un ensemble de tâches à effectuer en parallèle ou en temps partagé (seul le respect de la sémantique est indispensable), la structure ALT qui permet d'engager une action ou une autre selon la présence des données. Il introduit aussi des outils de communication inter-processus par rendez-vous ; les canaux. C'est un langage qui met l'accent sur le parallélisme de contrôle. Le point faible des langages de ce type est qu'ils sont inféodés à un modèle de programmation, à une classe d'architecture. OCCAM a été "dimensionné" pour des machines MIMD à

grain grossier ou moyen, avec des processeurs multiprogrammés possédant une mémoire privée importante, et, éventuellement, une mémoire commune. Concurrent SmallTalk (CST) [DAL86, GOL83], conçu comme langage possible pour les calculateurs MIMD développés au Caltech, suppose un modèle de machine où l'allocation dynamique d'objets et l'appel de procédure à distance sont des opérations très rapides, du même ordre de grandeur que les instructions ordinaires. L'architecture est MIMD, à passage de messages, sans mémoire commune, les processeurs élémentaires sont multiprogrammés, de taille moyenne, en grand nombre (plusieurs milliers). L'association d'un système d'exploitation distribué et de hardware spécialisé gère allocations, *garbage collecting*, migrations, etc.

4°) *Les langages à sémantique intrinsèquement parallèle.* Toute une série de langages parallèles ont été imaginés, souvent avec des objectifs théoriques (preuve formelle de programmes). Les classes de langages les plus connues sont les langages *dataflow* (LUSTRE [CAS87, PLA87], SIGNAL, ESTEREL...) et les langages d'acteurs (ACT[AGH86], CANTOR [ATH86]...). Malgré leurs qualités sémantiques, ces langages sont peu utilisés en raison souvent d'une implémentation difficile sur les architectures classiques. Des langages non-impératifs, comme PROLOG peuvent aussi être vus comme des langages parallèles. Le parallélisme le plus simple à mettre en œuvre est le parallélisme OU (parallélisme entre clauses), mais il se heurte au problème de l'explosion du nombre de processus. Les stratégies actuellement utilisées sont les stratégies multi-séquentielles : les processeurs explorent l'arbre en profondeur comme en séquentielle, mais les processeurs libres peuvent prendre des branches en attente aux processeurs sur-chargés [LUS88, BRI91]. D'autre projets tendent plutôt à modifier le langage (langages gardés : PARLOG, Concurrent Prolog, GHC [CHA89, SHA89]).

Le réseau cellulaire pose des problèmes généraux d'implémentation des langages de haut niveau. La finesse du grain matériel (taille de la mémoire, taille des messages, portée limitée) correspond souvent assez mal au grain logiciel des langages habituels du parallélisme comme OCCAM : le compilateur doit découper les processus de l'utilisateur en plusieurs processus compatibles avec la taille mémoire, ce qui représente une tâche non triviale. Plus qualitativement, les caractéristiques du modèle matériel (communication par message, pas de mémoire commune, pas de création dynamique de processus) ne représente pas toujours une base très intéressante pour supporter la sémantique des langages (pour OCCAM, il faut ramener par logiciel la communication par rendez-vous à une communication par messages).

Deux types de langages ont actuellement retenu notre attention : les langages d'acteurs à cause de leur sémantique à base de messages, étudiés depuis 1991 par Youssef Latrous, et les langages *data-flow* , dont nous allons détailler un représentant.

2. LUSTRE.

Le langage *data-flow* LUSTRE, inspiré de LUCID, a été développé par l'équipe SPECTRE du Laboratoire de Génie Informatique de Grenoble. Sa sémantique parallèle statique et la finesse du grain de traitement le rendent tout particulièrement attractif pour la programmation de notre réseau (il est plus facile de regrouper des processus fins sur un processeur que d'éclater un gros processus sur plusieurs processeurs). Il a donné lieu à un travail d'implémentation au sein de notre équipe, effectué par Eric Payan de 1988 à 1991 [PAY91]. Nous n'en rappellerons ici que les principaux enseignements, après une brève description du langage et de sa mise en œuvre.

2.1. Description.

Un programme LUSTRE consiste en un ensemble *d'équations* définissant des *variables*. Une variable X représente une *suite* de valeurs x0,x1,...,xn,... de même type de base (augmenté d'une valeur particulière notée *nil*). Elle est définie par la donnée d'une équation X=<expr>, où <expr> est soit une constante, soit une variable, soit une composition d'expressions. Les compositions possibles sont :

- les compositions arithmétiques et logiques définies comme suit :

  X op Y = (x0 op y0, x1 op y1, ...)

  op X = (op x0, op x1, ...)

- un opérateur ternaire conditionnel proche du "?:" du langage C est également disponible :

  IF <cond> THEN <expr1> ELSE <expr2,

  expression qui désigne la suite composée de valeurs de <expr1> et de <expr2>, sélectionnées à chaque instant par la valeur courante de l'expression <cond>.

- un opérateur de "retard"

  PRE(x0,x1,...) = (nil,x0,x1...)

- un opérateur d'initialisation "suivi de"

  E -> F = (e0,f1,f2,...)

Les programmes peuvent être structurés au moyen de la construction "node", équivalent des fonctions des langages procéduraux :

```
NODE <nom> ({<arg> : <type> })
    RETURNS (<variable resultat> : <type>)
    VAR <variables locales>
    LET
        <ensemble des équations nécessaire à définir le résultat>
    TEL
```

LUSTRE permet aussi de définir une suite sur une horloge différente de l'horloge de base :

X=Y WHEN H

définit une variable X ne possédant de valeur, égale à celle de Y, que lorsque la suite booléenne H vaut VRAI. L'opérateur CURRENT permet de projeter une variable sur une horloge différente :

instant	0	1	2	3	4	5	6
H	1	0	1	1	0	0	1
Y	y0	y1	y2	y3	y4	y5	y6
X= Y when H	y0		y2	y3			y6
current(X)	y0	y0	y2	y3	y3	y3	y6

Tableau 1 — *Horloges LUSTRE et fonctions associées.*

Enfin, signalons la possibilité de décrire des tableaux de variables.

2.2. Mise en œuvre.

L'implémentation pour le réseau cellulaire se base sur la transformation en un graphe d'opérateurs de l'ensemble d'équations qui constituent le programme.

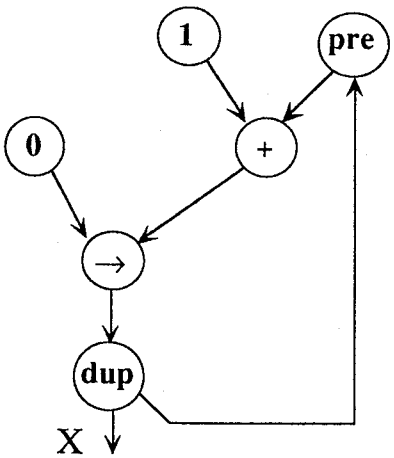


Fig. 1 — *Graphe correspondant à X = 0 -> pre(X)+1.*

La compilation comprend plusieurs phases :

1- Développement des tableaux
2- Extraction du graphe d'opérations
3- Limitation du degré de sortie des opérateurs

4- Ordonnancement des opérateurs
5- Placement
6- Génération de code

La finesse du grain des opérateurs fait que le compilateur, qui associait au départ un opérateur par cellule, en regroupe maintenant plusieurs sur chaque cellule. La synchronisation des opérateurs se fait par un protocole requête-réponse, amélioré par la suppression des requêtes inutiles. Le compilateur tente d'optimiser le placement d'un double point de vue : limitation des longueurs des connexions, respect des opportunités de parallélisme lors des regroupements.

Le regoupement conduit à une multiprogrammation des cellules, pour laquelle deux solutions ont été étudiées et comparées : l'une à base de scrutation circulaire des opérateurs, implémentée par des instructions TRY, l'autre fixant statiquement l'ordre d'évaluation des opérateurs au sein de la cellule. La seconde approche s'est avérée la plus rapide et la plus compacte, bien que la différence dépende de la forme du graphe. Nous ne pouvons pas en tirer des conclusions sur l'utilité de l'instruction TRY, car fixer l'ordre d'exécution de processus est possible en LUSTRE, mais pas dans le cas général.

2.3. Résultats.

Les tests de performances se sont montré fort encourageants, avec une activité moyenne comprise en 30 et 40%. Sur un programme de multiplication de matrices, le rapport de performance est de 2,5 avec le même programme écrit en assembleur (3,6 ramené au même nombre de cellules), le rapport en nombre de processeurs nécessaires de 1,5.

Nous sommes donc dans des ordres de grandeur entre compilation et assemblage relativement traditionnels, ce qui laisse envisager l'utilisation de LUSTRE à des fins générales. Toutefois, si le langage LUSTRE se prête bien par exemple à l'expression des réseaux systoliques, il est fortement limité sur deux points :

- son caractère purement statique le rend impropre à l'expression efficace des algorithmes impliquant la création dynamique de processus (recherche dans un arbre par exemple) ;
- certains algorithmes complexes demandent un effort important de codage qui est difficilement automatisable (une version *data-flow* du programme de recherche de plus court chemin dans un graphe serait à coup sur d'un rapport de performance et de surface beaucoup plus défavorable que ceux annoncés plus haut.

En résumé, LUSTRE est un choix de langage tout à fait opérationnel pour notre réseau cellulaire, mais il ne peut pas être considéré comme un langage polyvalent rendant inutile l'étude d'autres langages et outils de programmation.

3. Approche manuelle.

Programmer explicitement le parallélisme sur une machine telle que le réseau cellulaire nécessite une remise en cause complète des habitudes. Pour illustrer ce fait, il nous semble important d'analyser en détail quelques applications et la manière de les programmer, pour se rendre compte de ce que cela représente et voir quels enseignements on peut en tirer. Les programmes *(tableau 2)* ont été analysés puis codés en assembleur et simulés. Nous présenterons les quatre applications suivant dans ce chapitre (les autres étant reportées en annexe 1, pour ne pas surcharger le corps du document) :

- le tri de chaîne de caractères
- la recherche de plus court chemin dans un graphe
- les réseaux neuro-mimétiques
- les automates cellulaires (Lattice Gaz Model)

Conway2D	annexe 1, §1.
Conway1D	annexe 1, §1.
Lattice Gaz Model 1D	§3.4.
Distance entre mots	annexe 1, §2.
XOR neuronal	§3.3.
XOR synaptique	§3.3.
Multiplication de matrices creuses	annexe 1, §5.
Plus court chemin	§3.2.
Plus court chemin (Chandy)	§3.2.
Crible d'Eratosthène	annexe 1, §3.
Tri de chaines par insertions	§3.1.1.
Tri de chaines par interclassement (codage [BOI91])	§3.1.2.
Tri en serpentin	annexe 1, §4.
Tri à bulles	annexe 1, §4.
Tri en hélice	annexe 1, §4.
Simulation logique à échéancier [JAC91]	annexe 1, §6.
Problème des huit reines	annexe 1, §7.

Tableau 2 — *Programmes codés pour le réseau cellulaire.*

3.1. Tri de chaînes de caractères.

Point n'est besoin de rappeler l'importance du tri. Certains algorithmes trient des structures de données distribuées sur le réseau (tri à bulles et variantes), d'autre trient des flux de données. Nous nous intéresserons ici à des algorithmes du second type, plus spécifiquement au *tri de chaînes de caractères de longueurs variables*, selon deux algorithmes complémentaires : le tri par insertion et le tri par interclassement.

3.1.1. Tri par insertions.

Le tri par insertions se programme en parallèle selon le principe suivant : on constitue une liste de cellules de longueur égale au nombre de valeurs à trier. Par hypothèse, cette liste sera toujours ordonnée. Comme la liste est de taille fixe, on l'initialise avec des valeurs supérieures à toutes les valeurs possibles dans le jeu de données. On injecte ensuite successivement toutes les valeurs à un bout de la liste. Chaque cellule compare la valeur qu'elle reçoit avec celle qu'elle possédait. Si cette valeur est inférieure à l'ancienne, la cellule garde cette nouvelle valeur et donne l'ancienne à la cellule suivante, sinon elle transmet la nouvelle valeur. Les valeurs initiales sont petit à petit expulsées hors du réseau à l'autre bout de la liste. Une fois toutes les valeurs injectées et la propagation terminée, la liste contient le jeu de valeurs trié. Avant même cette fin, une fois toutes les valeurs injectées, on peut injecter des valeurs terminales plus petites que toutes les autres pour expulser le jeu de valeurs. Si on désire trier dans l'autre sens, il faut inverser le rôle des valeurs initiales et terminales.

```
VAR   Cour:chaîne
      Nouv:chaîne
      ReqAval:canal d'entree de requêtes
      Entree:canal d'entree de chaînes
      Sortie:canal de sortie de chaînes
      ReqAmont:canal de sortie de requêtes
ENVOYER(ReqAmont)
CYCLE
  Nouv:=RECEVOIR(Entree)
  ENVOYER(ReqAmont)
  SI Nouv < Cour ALORS
      RECEVOIR(ReqAval)
      ENVOYER(Sortie,Cour)
      Cour:=Nouv
  SINON
      ENVOYER(Sortie,Nouv)
```

Une transposition directe de l'algorithme avec décomposition en boucles des opérations sur les chaînes peut être faite, mais un recouvrement entre traitement et réception suppose un transfert du tampon de réception (Entree) vers une variable (Nouv), et éventuellement un transfert de Nouv dans le tampon de la valeur de la

cellule. Pour supprimer ces transferts de données intra-cellulaires, il est possible de manipuler les tampons par l'intermédiaire de pointeurs, et de permuter leurs rôles au lieu de permuter leur contenu. L'algorithme qui en résulte présente un fort recouvrement : la comparaison peut débuter dès le début de l'arrivée du nouvel élément, et la réception de la donnée suivante peut structurellement commencer dès que la réception de la précédente est terminée. L'important est que l'amont dispose toujours d'un tampon où envoyer ses données.

Transposition directe

```
Cour,Nouv:caractères[]
Entree:canaux d'entrée[]
Sortie:canaux de sortie[]
ReqAmont:canal d'entree
ReqAval:canal de sortie

Envoyer(ReqAmont)
CYCLE
  I:=0
  ITERER /* réception */
    car:=Recevoir(Entree[i])
    Nouv[i]:=car
    SI car=0 SORTIR
    i:=i+1
  Envoyer(ReqAmont)
  i:=0
  ITERER /* comparaison */
    SI Nouv[i]≠Cour[i] SORTIR
    SI Nouv[i]=0 SORTIR
    i:=i+1
  SI Nouv[i]<Cour[i]
    i:=0
    Recevoir(ReqAval)
    ITERER /* émission de Cour, échange */
      Envoyer(Sortie[i],Cour[i])
      Cour[i]=Nouv[i]
      SI Cour[i]=0 SORTIR
      i:=i+1
  SINON
    i:=0
    Recevoir(ReqAval)
    ITERER /* émission de Nouv */
      Envoyer(Sortie[i]:=Nouv[i])
      SI Nouv[i]=0 SORTIR
      i:=i+1
FINCYCLE
```

Utilisation de pointeurs

```
tampons:canaux d'entree[3][]
pCour:pointeur = tampons[0]
pEntree:pointeur = tampons[1]
pNouv:pointeur = tampons[2]
pSortie:"pointeur"
ReqAmont:canal d'entree
ReqAval:canal de sortie

Envoyer(ReqAmont:=pNouv)
CYCLE
  Recevoir(pNouv^[0])
  Envoyer(ReqAmont:=pEntree)
  i:=0
  ITERER
    SI pNouv^[i]<pCour^[i] SORTIR
    SI pNouv^[i]>pCour^[i] SORTIR
    SI pNouv^[i] = 0 SORTIR
    i:=i+1
    Recevoir(pNouv^[i])
  SI pNouv^[i]<pCour^[i]
    Envoi(pCour)
    ech:=pCour, pCour:=pNouv
    pNouv:=pEntree, pEntree:=ech
  SINON
    Envoi(pNouv)
    ech:=pNouv, pNouv:=pEntree,
    pEntree:=ech
FINCYCLE

PROCEDURE Envoi(p:pointeur)
  pSortie:=Recevoir(ReqAval)
  i:=0
  ITERER
    SI Tester(p^[i])
      car:=Recevoir(p^[i])
    SINON
      car:=p^[i] (déjà lue)
    Envoyer(pSortie^[i]:=car)
    SI car=0 SORTIR
    i:=i+1
```

La théorie dit qu'il faut mettre autant de cellules que de valeurs à trier. La pratique impose au contraire de trier autant de valeurs qu'il y a de cellules. Ceci induit que nous allons être obligés de partitionner le problème. On va d'abord trier paquet par paquet, et ensuite interclasser ces paquets triés. En plaçant un numéro de paquet en poids fort de chaque valeur, on peut sans rien modifier à l'algorithme, pipe-liner le tri de chaque paquet. Le numéro de paquet placé en tête de chaque chaîne assurera un cloisonnement entre les valeurs des différents paquets.

Comme les données ne sont pas de même longueur, et qu'indépendamment de la longueur elles prennent des temps de traitement différents (la décision dans la comparaison se prend dès le premier caractère qui diffère), un recouvrement idéal ne sera pas possible, mais en moyenne, un taux d'activité à peu près régulier et d'un bon niveau peut être obtenu (85% une fois le pipe-line rempli, dont à peu près 25% de calcul).

La figure 2 montre l'évolution de la répartition de l'activité dans le temps, pour un tri avec 256 cellules. Dans une première phase, les données non significatives initialement présentes dans le réseau sont éjectées ; la première forte augmentation du taux d'activité correspond à la propagation du front d'éjection, puis le réseau se rempli progressivement de valeurs significatives. Le traitement des valeurs initiales étant plus rapide (ce sont des chaînes de longueur 1), l'activité moyenne est faible au début du remplissage, puis augmente parallèlement au pourcentage de valeurs significatives introduites. Une fois le pipe-line rempli, le réseau entre en régime normal qui reste stable tant qu'on continue à fournir des paquets de données.

On peut remarquer que la synchronisation (requête) représente une part très faible de l'activité du réseau.

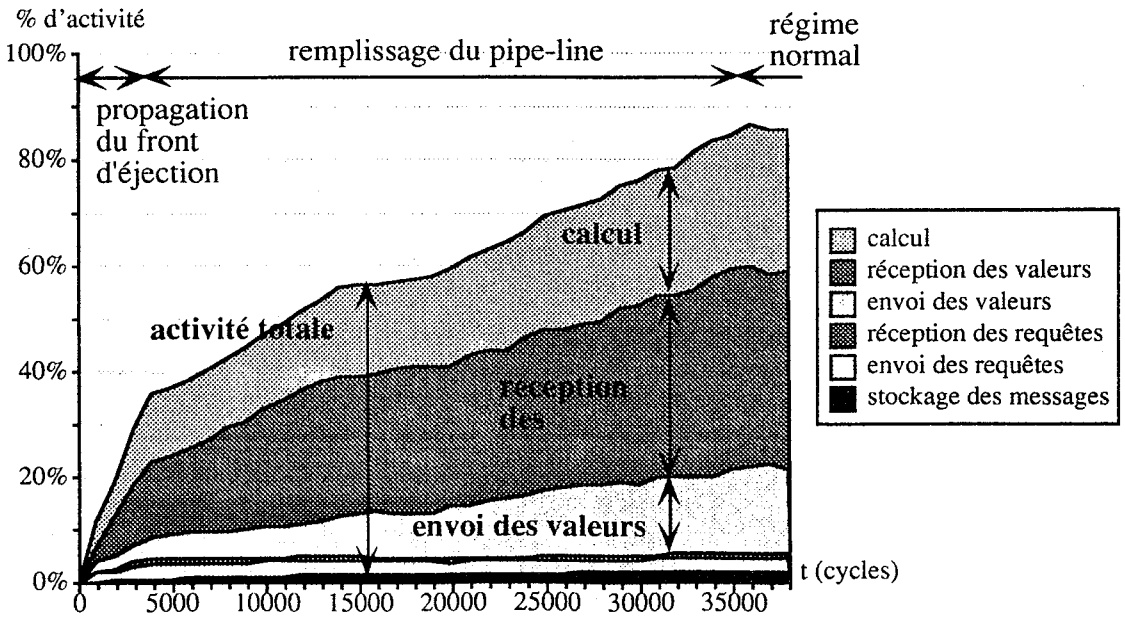


Fig. 2 — *Activité pour le tri de chaînes par insertion.*

3.1.2. Tri par interclassement.

Le tri par interclassement, aussi connu sous le nom de tri par fusion (merge-sort), est un algorithme très efficace dans sa version séquentielle. Son coût est en $O(n \log_2 n)$. Il consiste à diviser récursivement le problème du tri d'un tableau en deux tris de deux sous-tableaux et un interclassement. Cet algorithme se parallélise en "mettant la récursivité à plat". Bien entendu, il n'est plus possible d'avoir un tableau commun comme structure de données. Celui-ci est remplacé par un flux de données. L'opérateur de tri se réduit alors à un arbre binaire d'interclassement dont les feuilles sont les valeurs à trier et dont chaque nœud exécute l'algorithme suivant :

```
VAR   CourA,CourB:chaîne
      ReqAval:canal d'entree de requêtes
      A,B:canal d'entree de chaînes
      Sortie:canal de sortie de chaînes
      ReqA,ReqB:canal de sortie de requêtes
ENVOYER(ReqA) ENVOYER(ReqB)
CourA:=Recevoir(A)
CourB:=Recevoir(B)
ENVOYER(ReqA) ENVOYER(ReqB)
CYCLE
  RECEVOIR(ReqAval)
  SI CourA<CourB
     ENVOYER(Sortie,CourA)
     CourA:=Recevoir(A)
     ENVOYER(ReqA)
  SINON
     ENVOYER(Sortie,CourB)
     CourB:=Recevoir(B)
     ENVOYER(ReqB)
```

Le système de permutation des rôles de plusieurs tampons par échange de pointeurs peut être aussi exploité ici. Le principal inconvénient d'une telle implémentation est que la racine de l'arbre est un goulot d'étranglement par lequel doivent transiter tous les éléments du jeu de données. On a donc un coût de l'algorithme en $O(n)$, ce qui ne représente qu'un gain d'un facteur $\log_2 n$ par rapport à la version séquentielle. En fait, le taux d'activité moyen d'un nœud de profondeur n est borné à $1/2^{n-1}$, ce qui n'est guère satisfaisant.

Il faut noter que cette limite correspond à un fonctionnement continu de l'interclasseur. Le tri par fusion "pur" est un algorithme qui prend ses valeurs soit en place, soit sur un nombre de canaux égal au nombre de valeurs : dans ce cas, par chaque feuille ne transite qu'une seule valeur par tri, alors que la racine est saturée ! Ceci explique le faible taux d'activité observable sur ce programme *(fig. 3)*.

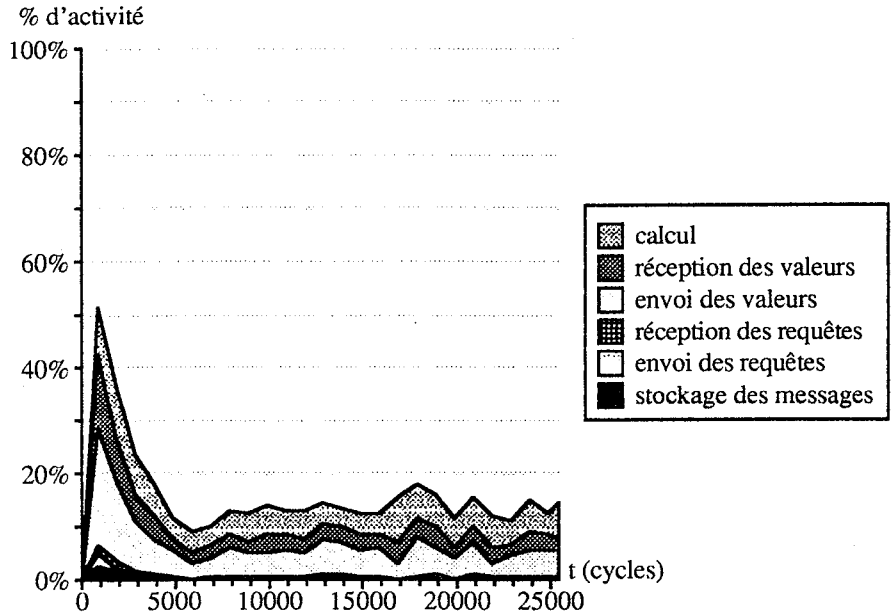


Fig. 3 — *Activité pour le tri de chaînes par interclassement.*

On peut avoir l'idée de transformer cet algorithme en un tri hybride : on limite arbitrairement la profondeur de l'interclasseur et on fait transiter par chaque feuille une suite triée de valeur. Pour automatiser la production de ces suites triées, on peut greffer un tri par insertions, mais c'est une très mauvaise idée. En effet, la fréquence de production sera limitée par la demande au niveau de la feuille. De plus, le gain lié au meilleur taux d'activité du tri par insertion est pour une bonne part illusoire : il rend simplement compte du fait qu'un tri par insertions est moins efficace qu'un tri par fusion ; il y a plus d'opérations effectuées, pour un même résultat.

Une bien meilleure solution consiste à programmer le réseau d'abord en tri par insertions, à trier une suite de paquets et à en mémoriser dans l'hôte le résultat ; ensuite, on reprogramme le réseau en interclasseur de taille maximale par rapport au nombre de cellules disponibles, on effectue des interclassements partiels avec mémorisation dans l'hôte jusqu'à arriver à l'interclassement final de la racine. Reste à savoir s'il est véritablement utile d'utiliser un tri différent pour les paquets initiaux.

En séquentiel, les algorithmes les plus simples, tri par insertions, tri à bulles, sont en $O(n^2)$ et les plus efficaces sont en $O(n \log_2 n)$, mais lorsqu'on passe de la théorie à la pratique, la situation n'est plus aussi claire. Si l'on cherche à prendre en compte tous les aspects d'implémentation, il est évident qu'en dessous d'une certaine taille du jeu de données, le coût du contrôle devient prépondérant, et les algorithmes les plus simples peuvent devenir les plus efficaces. On peut être amené alors à hybrider par exemple un tri par fusion et un tri par insertion, ce dernier prenant le relais lorsque le découpage dichotomique a réduit la taille des sous-problèmes en dessous d'un certain niveau. Ce niveau dépend de l'implémentation ainsi que des données, et peut être fixé empiriquement. Le gain peut atteindre 15 %.

En parallèle, ce sont plutôt des considérations d'entrées/sorties qui vont être à prendre en compte. Les paramètres à déterminer sont la taille des paquets initiaux et les nombres d'interclasseurs et de trieurs initiaux qu'on va faire fonctionner en parallèle.

Ces tris ne sont pas d'une complexité algorithmique spécialement avantageuse (*speedup* avec N processeurs en log2 N seulement), mais leur programmation est intéressante car elle montre comment mettre en œuvre un fort recouvrement entre réception et comparaison/émission, et tirer un profit de la décomposition des messages de chaînes et messages d'octets, décomposition qui était à priori un handicap.

3.2. Recherche de plus courts chemins dans un graphe.

Les calculs sur les graphes constituent une classe d'applications très vaste, tant en nombre d'algorithmes qu'en domaines d'utilisation. Beaucoup de problèmes de graphes sont une combinatoire élevée, et la nature irrégulière des graphes les rend peu pratiques à manipuler sur les supercalculateurs classiques. C'est pourquoi ils ont été pointés très tôt comme de bons exemples de programmes pour les machines massivement parallèles..

Nous nous intéresserons ici exclusivement à la "recherche des plus courts chemins dans un graphe". Ce problème familier consiste à déterminer, pour un graphe dont les arcs sont orientés et pondérés, un chemin entre deux sommets tel que la somme des poids des arcs empruntés soit minimale. En pratique, les algorithmes correspondants fonctionnent par diffusion : c'est l'ensemble des chemins d'un sommet donné à chacun des autres sommets qui est calculé. Une solution séquentielle a été proposée par Dijkstra en 1959 [DIJ59], et une solution parallèle par Chandy et Misra [CHA82]. Une autre version parallèle a été proposée par Dally [DAL86] : cette solution exploite un plus fort degré de synchronisation pour être moins sensible aux graphes pathologiques et à l'explosion combinatoire sur les gros graphes.

Dans l'algorithme de Chandy et Misra, chaque nœud du graphe est associé à un processeur qui exécute le processus suivant :

```
min:=∞
CYCLE
 recevoir(emetteur,distance)
 SI distance<min
    min:=distance
    POUR i:=1 à sortance
       Envoyer(self,min+poids[i]) à successeur[i]
```

A chaque fois qu'un nœud reçoit le résultat de l'exploration d'un chemin jusqu'à lui, il compare la longueur de ce chemin à celle du meilleur chemin connu jusqu'alors, et si ce nouveau chemin est de coût inférieur, il le mémorise et propage ce nouveau chemin à ses successeurs. L'algorithme est initié en envoyant le message (hôte,0) au nœud origine. Notons la variable `prédécesseur`, qui sert globalement à mémoriser l'arbre des plus courts chemins. Sous cette forme simplifiée, l'algorithme :

- ne détecte pas la terminaison
- boucle indéfiniment s'il existe un cycle de poids négatif.
- effectuer aucun contrôle de flux.

Nous ne nous occuperons pas des cycles de poids négatifs, pour lesquels une première passe de détection est normalement prévue. La détection de terminaison peut se faire par l'usage de messages d'acquittement : un compteur est associé à chaque nœud, qui compte le nombre de chemins fournis à charge d'exploration aux successeurs ; ce compteur est incrémenté à chaque message envoyé à un successeur, et décrémenté lorsque le successeur signale la fin de l'exploration. Lorsque tous les successeurs d'un nœud se sont acquittés des tâches qui leur ont été soumises, le nœud peut à son tour envoyer un acquittement à celui de ses prédécesseurs qui est à l'origine du dernier plus court chemin connu. Lorsque le nœud origine envoi un acquittement à l'hôte, l'algorithme est terminé.

Cet algorithme présente malheureusement des cas pathologiques où le nombre de messages générés croît exponentiellement [DAL86]. Ces cas défavorables correspondent à une synchronisation trop faible entre les nœuds : dès qu'un message donnant un nouveau minimum est trouvé, il y a ouverture automatique d'autant de chemins que de successeurs, indépendamment de la disponibilité éventuelle de minimum plus intéressant parvenu depuis. Dally résout ce problème en imposant une synchronisation forte entre un nœud et ses voisins : son algorithme SSP (Synchronous Shortest Path) est un algorithme à deux phases : dans un premier temps, les nœuds fournissent à tous leurs successeurs le meilleur chemin qu'ils connaissent, puis, dans un deuxième temps, chaque nœud recalcule le minimum en fonction de toutes ses entrées. Dans un graphe à n sommets et p arcs, le plus long chemin en nombre d'arcs est de longueur n-1, par conséquent n-1 itérations au plus sont nécessaires pour l'établir. Lors de chaque itération, p messages sont émis : le nombre total de messages nécessaires est donc au pire en O(n.p) [DAL86].

```
predecesseur:=nil
min:=∞
NbChemins:=0
PAR
  SEQ
    CYCLE
       Recevoir(emetteur,distance)
       SI distance<min
          min:=distance
          SI predecesseur≠nil : Acquitter(predecesseur)
          predecesseur:=distance
          POUR i:=1 à sortance
             Envoyer(self,min+poids[i]) à successeur[i]
             NbChemins++
       SINON Acquitter(emetteur)
    FINCYCLE
  SEQ
    CYCLE
       NbChemins--
       SI NbChemin=0
          SI predecesseur≠nil : Acquitter(predecesseur)
          predecesseur:=nil
    FINCYCLE
```

Algorithme de Chandy et Misra (d'après [DAL86]).

Nous allons voir que l'algorithme de Chandy et Misra ne nécessite pas une resynchronisation aussi forte que celle de l'algorithme SSP pour fonctionner correctement dans tous les cas.

Une première programmation de ce problème sur le réseau cellulaire qui n'intégrait pas de détection de terminaison et que nous ne détaillerons pas ici, a mis en évidence deux points importants :

- la mémoire de la cellule est de trop faible taille pour qu'il soit possible de concilier une connectivité "forte" (plus d'une vingtaine d'arcs) et la détection de terminaison.
- le partitionnement du processus en deux cellules permet un gain de performance, bien que l'algorithme soit propagatif et que ce partitionnement double la longueur des chemins en nombre de cellules traversées (environ 20 % sur le temps d'exécution total, avec un graphe construit aléatoirement).

Conceptuellement, le partitionnement d'un nœuds en deux plusieurs cellules revient à ajouter des sommets et des arcs de poids nul *(fig. 4)*. Ainsi transformés, les nœuds du graphe nécessitent un traitement itératif, soit des entrées, soit des sorties, mais pas des deux à la fois, de sorte que la scission ne réduit pas seulement la connectivité, mais aussi la longueur du programme de traitement. La première cellule calcule en

permanence le plus court chemin parmi ceux qui lui sont fournis par ses entrées, et envoie ce chemin à la seconde cellule qui va le propager aux sorties.

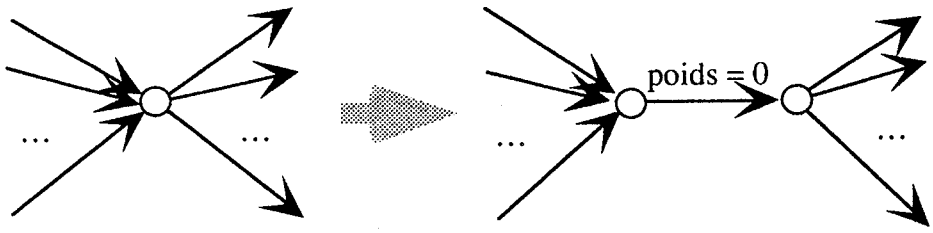


Fig. 4 — *Limitation du degré d'interconnexion par scission des nœuds.*

De nombreuses variantes peuvent être imaginées pour programmer chacune des taches. La première cellule doit déterminer le minimum de ses entrées ; elle peut le faire :

- en testant successivement toutes les entrées, acquittant au fur et à mesure les chemins non favorables, et en ne fournissant qu'à la fin de la scrutation à la seconde cellule le meilleur minimum rencontré (limitation du nombre de messages générés ; c'est une solution proche de cette de Dally)
- en fournissant les nouveaux minimums à la seconde cellule dès qu'ils sont rencontrés, en cours de scrutation (propagation plus rapide).

La première cellule ne peut envoyer un nouveau minimum à la seconde cellule que si celle-ci lui en a demandé un (requête parvenue). Si la requête n'est pas arrivée :

- on peut l'attendre (codage plus simple)
- on peut continuer ou recommencer une scrutation (limitation du nombre de messages générés)

La seconde cellule distribue les minimums reçus vers les sorties. Là aussi, il y a une synchronisation par requêtes avec les sorties. Si, lorsque la cellule est prête pour émettre un message vers une sortie, la requête qui le permettrait n'est pas encore arrivée, on peut :

- passer à la cellule suivante pour revenir plus tard à cette cellule (réduction du temps de service si le degré de sortie est fort)
- attendre la requête (codage plus simple)

Durant la boucle de propagation, un nouveau minimum peut arriver, qui peut être pris en compte :

- entre chaque émission (limitation du nombre de messages)
- lorsque tous les messages ont été envoyés
- en fin de table d'émission (dans le cas où l'absence de requête provoque le passage à la sortie suivante).

Pour palier aux faiblesses de l'algorithme de Chandy et Misra, il faut limiter au moins dans l'une des deux cellules la production de messages. Dans la version qui a été programmée, dans la première cellule, la solution d'une attente blocante a été retenue, dans la mesure où il y a aura anticipation des demandes par la seconde cellule qui fait

Cellule de calcul du minimum

```
min:canal de sortie
minreq : canal d'entrée
courmin: canal d'entrée
distance[entrance] : constantes
val[entrance] : canaux d'entrée
req[entrance] : canaux de sortie
acq[entrance] : canaux de sortie
i,nouv,prec,minprec,src:entrée
acquit:booleen

min:=∞
courmin:=∞
minsrc:=0
POUR i:=0 A entrance-1
  Envoyer(req[i])
acquit:=vrai
i:=0
CYCLE
  SI Tester(val[i])
    prec:=Recevoir(val[i])
    Envoyer(req[cour])
    nouv:=prec+poids[i] (1)
    SI nouv<min
      SI ¬ acquit
        Envoyer(acq[src]:=minprec)
      min:=nouv
      minprec:=prec
      src:=i
      acquit:=faux
      Recevoir(minreq)
      Envoyer(min)
    SINON
      Envoyer(acq[i]:=prec)
  SI courmin=min ET ¬ acquit
    Envoyer(acq[src]:=minprec)
    acquit:=vrai
  i:=(i+1) mod entrance
FINCYCLE
```

Cellule de distribution

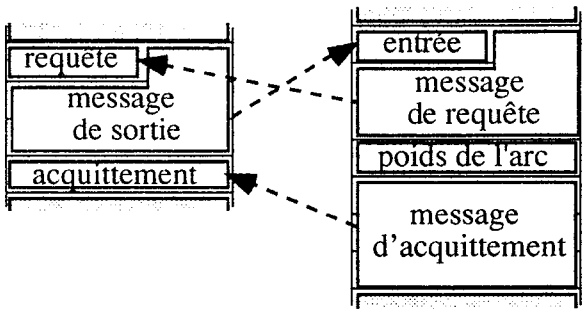
```
min:canal d'entrée
minreq : canal de sortie
courmin : canal de sortie
acq[sortance] : canaux d'entree
req[sortance] : canaux d'entree
dern[sortance] : canaux de sortie
i,nbacqs:entier


Envoyer(minreq)
REPETER
  courmin:=Recevoir(min)
  Envoyer(minreq)
  SI sortance=0
    Envoyer(courmin)
JUSQU'A sortance≠0 (2)
nbacqs:=0
POUR i:=0 A sortance-1
  dern[i]:=∞
i:=0
CYCLE
  SI dern[i]>courmin
    SI Tester(req[i])
      Recevoir(req[i])
      Envoyer(dern[i]:=courmin)
  SINON
    SI acq[i]≤courmin
      nbacqs++
      acq[i]:=courmin+1 (3)
      SI nbacqs:=sortance
        Envoyer(courmin)
    i:=(i+1) mod sortance
  SI Tester(min)
    courmin:=Recevoir(min)
    Envoyer(minreq)
    nbacqs:=0
FINCYCLE
```

(1) la valeur reçue est la distance au nœud précédent, il faut lui ajouter le dernier arc ; on conserve prec et minprec pour acquitter sans avoir à soustraire la longueur de l'arc
(2) les sommets de sortance nulle ne sortent jamais de cette boucle, les autres l'exécutent une seule fois
(3) la valeur courante du canal est incrémentée pour ne pas être prise en compte 2 fois.

que la requête sera envoyée rapidement. La limitation de production de message est faite au niveau de la seconde cellule : lorsque la requête d'une cellule de sortien'est pas arrivée, l'envoi est différé ; entre chaque émission, on teste l'arrivée éventuelle d'un nouveau minimum. La détection n'est pas réalisée exactement de la même manière que dans l'algorithme original. Par hypothèse, le champ donnée du message d'acquittement contient le minimum acquitté. Dans la boucle d'émission, la seconde cellule effectue,

en même temps que ses tentatives de sortie et que son écoute du canal de minimum, un comptage des acquittements reçus égaux au minimum courant. Pour ne pas compter plusieurs fois le même acquittement, on l'incrémente. Lorsque le compteur est égal à la sortance, la terminaison est signalée à la première cellule, qui peut à son tour acquitter l'entrée (si cet acquittement n'a pas déjà été provoqué par un nouveau minimum). Cette méthode permet de supprimer le contrôle de flux sur les voies d'acquittement.



a) structure d'une sortie *b) structure d'une entrée*

Fig. 5 — *Structures de données des cellules d'entrées et de sorties.*

La topologie du graphe est décrite par des tables d'entrées et de sorties, nous allons donc utiliser les modes d'adressages indirects. Comme nous l'avons vu, la meilleure façon de les mettre en œuvre consiste à placer les données dans l'ordre où le programme en aura besoin. Pour la première cellule, l'ordre imposé par l'algorithme est le suivant : pour chaque entrée, canal d'entrée, message de requête, poids de l'arc, et message d'acquittement *(fig. 5)*. L'entrée et la requête correspondante sont contractées, ce qui permet un gain de place et fait en sotre que la donnée du message de requête est égale à la dernière valeur du canal d'entrée (nous aurons besoin de cette propriété plus loin).

Pour l'acquittement, deux cas peuvent se présenter : acquittement immédiat si la valeur reçue n'est pas intéressante, et acquittement différé jusqu'à la fin d'exploration ou l'arrivée d'une nouvelle valeur inférieure. Dans ce second cas, le message d'acquittement est copié dans un tampon unique lors de la lecture de l'entrée, pour utilisation future.

Dans la cellule de sortie, l'ordre idéal est impossible à déterminer, car plusieurs traitements différents peuvent avoir lieu selon l'arrivée des requêtes. L'ordre le plus acceptable est : pour chaque sortie, canal de requête, message de sortie, canal d'acquittement. Lors du parcours, certains éléments sont sautés par incrémentation de I. Comme pour la cellule d'entrée, le canal de réception et le message en sens inverse sont contractés, mais comme la valeur du message doit être conservée après (elle indique si le message a été envoyé ou pas), il importe que la requête qui vient l'écraser ait la même valeur, ce qui est le cas grâce à la contraction dans la cellule d'entrée.

Cet algorithme, moins fortement synchronisé que l'algorithme SSP, est pourtant de même coût en temps d'exécution :

Le temps T de traitement d'un message, c'est à dire le temps entre sa réception et l'envoi de tous les messages correspondant vers les sorties, est borné, quel que soit l'état des cellules du nœud au moment de la réception :

Soient T_1 le temps d'examen et de traitement d'une entrée (constante), T_2 le temps maximum de traversée de la première cellule, T_3 le temps maximum de prise en compte dans la seconde cellule, T_4 le temps d'émission d'une sortie (constante), T_5 le temps d'émission de tous les messages en sortie, $T=T_2+T_3+T_5$. Soient E l'entrance maximale et S la sortance maximale des nœuds (nous supposerons être dans le cas d'un graphe creux, où le nombre d'arcs est du même ordre de grandeur que le nombre de sommets)

Le temps de prise en compte T_3 d'un nouveau minimum par la seconde cellule est unitaire (il est testé après chaque tentative d'émission). Il définit la période à laquelle les nouveaux minimums peuvent être pris en compte dans la seconde cellule et nous permet de préciser T_2.

Dans le plus mauvais cas, E-1 entrées sont examinées avant celle qui nous intéresse, et chacune d'elles est traitée comme un nouveau minimum. En y ajoutant le temps de traitement de la nouvelle entrée, nous avons :

$$T_2 = (E-1).\min(T_1,T_3) + T_1$$

Une fois le nouveau minimum détecté, envoyé à la seconde cellule et pris en compte par celle-ci, le cas le plus défavorable qui se présente est celui où un message doit être envoyé à chaque sortie (pas de nouveau minimum entre temps). A cause du contrôle de flux sur les sorties, il faut attendre qu'une requête soit parvenue pour émettre. Le temps séparant la précédente émission et le retour de la requête peut être borné par T2. Nous avons donc :

$$T_5 = T_2 + S.T_4$$

d'où

$$T = (E-1).\min(T_1,T_3) + T_1 + T_3 + (E-1).\min(T_1,T_3) + T_1 + S.T_4$$

$$T = 2.(E-1).\min(T_1,T_3) + 2.T_1 + T_3 + S.T_4$$

Comme dans l'algorithme SSP, le temps d'exécution est donc proportionnel à la profondeur de l'arbre des plus courts chemins, profondeur que nous pouvons borner par le nombre de sommets.

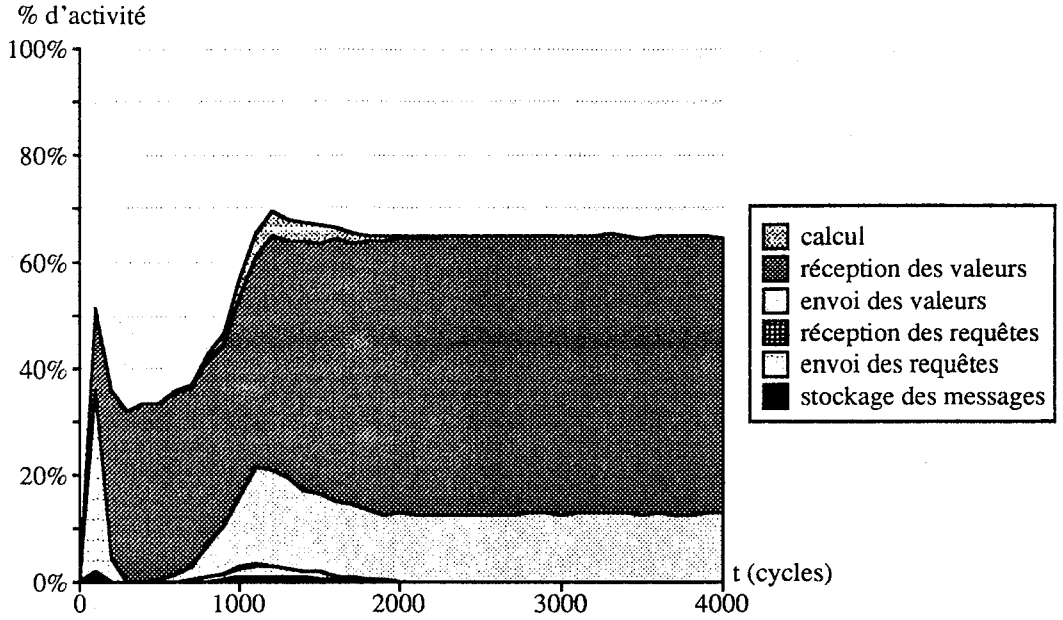


Fig. 6 — *Activité pour l'algorithme de recherche de plus court chemin.*

Cet exemple est intéressant car il met en lumière d'une certaine manière le degré maximum de complexité comportementale dont il est possible de doter une cellule (c'est en tout cas le moins trivial parmi ceux programmés pour cette étude). En prenant un peu de recul, on s'aperçoit rapidement que ce type de solution est tributaire des techniques de placement de graphe que nous saurons mettre en œuvre : il n'est pas nécessaire d'avoir un algorithme performant si le placement des données est d'un coût supérieur à celui de l'algorithme séquentiel original. Comme pour la programmation des automates cellulaires, d'autres voies moins immédiates sont peut-être à envisager.

3.3. Réseaux de neurones

L'implémentation des réseaux de neurones a été étudiée au sein de notre équipe par B. Faure (dans le but de réaliser un réseau cellulaire dédié). Dans [FAU90] est décrite une méthode d'implantation de réseaux multi-couches sur réseau cellulaire, qui associe à chaque cellule un neurone et ses liaisons synaptiques. Les stimuli sont transmis sous forme de messages véhiculés par le système de communication du réseau cellulaire. Nous pouvons appliquer cette méthode au cas du réseau cellulaire général.

Le problème principal des réseaux de neurones est la forte connectivité des neurones, quel que soit le modèle choisi. Etant donné la taille de la cellule de base, nous devons donc distribuer les neurones formels sur plusieurs cellules, ce qui peut se faire de diverses façons *(fig. 7)*. Le placement A est celui décrit par B. Faure : tout neurone trop connecté est remplacé par un graphe fonctionnellement équivalent de neurones plus petits, peuvent fonctionner en parallèle L'association neurone-processeur permet d'exploiter pleinement le parallélisme neuronal, mais l'exploitation du parallélisme synaptique n'est pas réalisée, ou seulement partiellement.

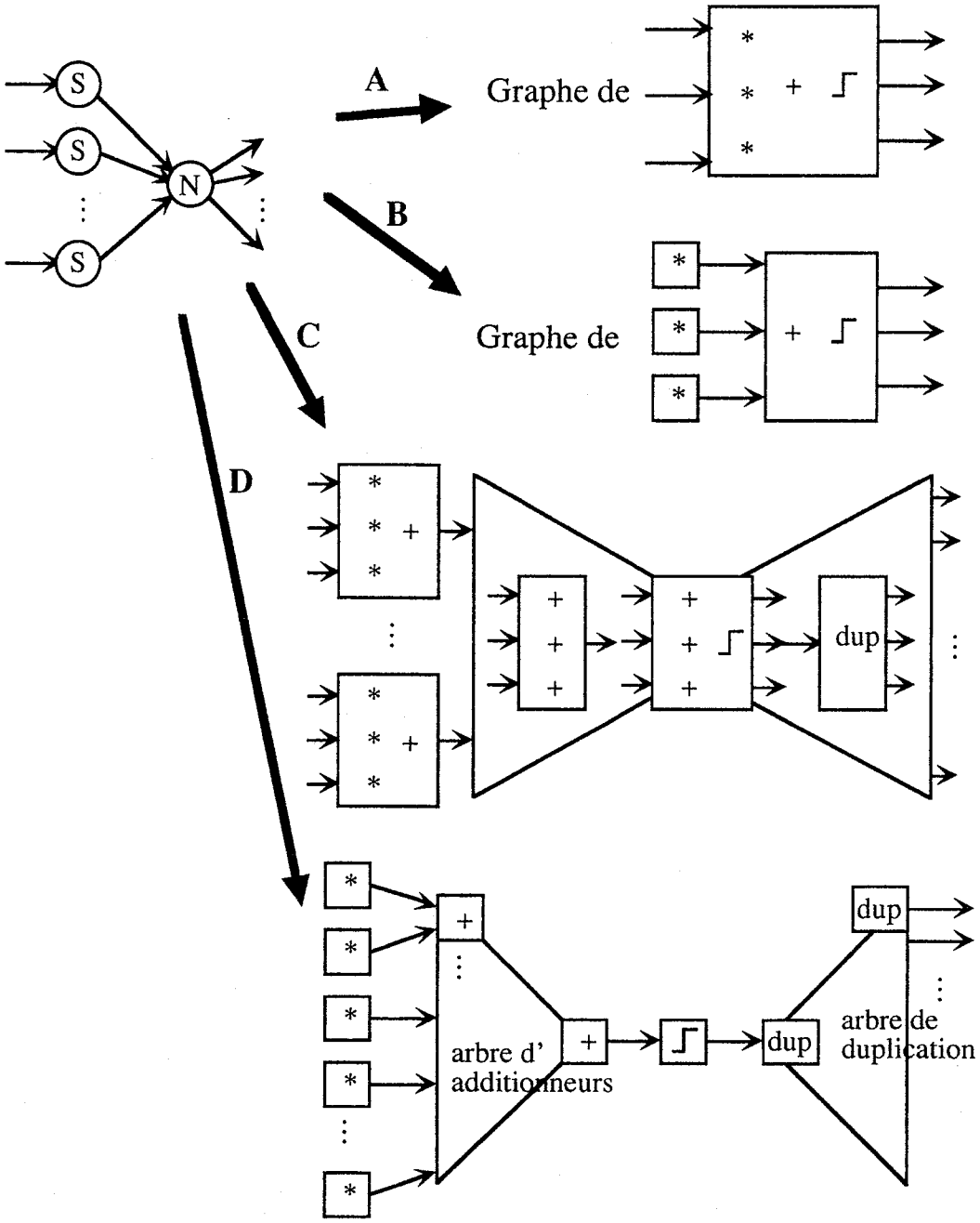


Fig. 7 — *Différents types de parallélisation d'un neurone.*

Le placement B correspond à une exploitation du parallélisme synaptique : les multiplications des entrées par les poids des liaisons synaptiques sont effectuées par des processeurs différents et donc en parallèle. Il ne résout pas le problème de la connectivité des neurones, qui se retrouve au niveau du processeur de sommation-seuillage, ce qui fait que la décomposition d'un neurone en un graphe de neurone reste indispensable. Le placement C se propose de décomposer la fonction de sommation en une arborescence d'additionneurs, et l'axone en un arbre de duplication. La connectivité

de chaque cellule est maintenant naturellement bornée. Le placement D est une combinaison des placements B et C. La longueur du code correspondant à chacune des fonctions est liée à la précision désirée pour les calculs, ainsi qu'à la sophistication de la fonction de seuillage, de sorte que ces paramètres vont entrer en ligne de compte pour l'équilibrage de la charge (en volume de code et en temps d'exécution). Notons que les trois derniers placements sont rendus possibles par la faculté de programmer différemment chaque cellule, possibilité qui n'existait pas sur le réseau dédié. L'exemple suivant montre le placement et l'activité obtenue selon les placements A et B, pour le réseau neuronal implémentant le XOR *(fig. 8)*.

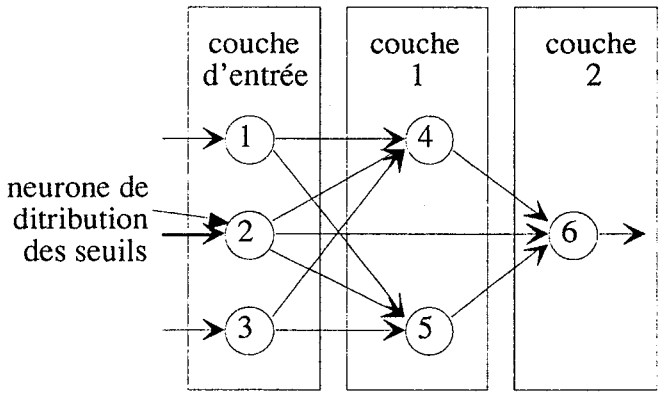


Fig 8 — *XOR neuronal.*

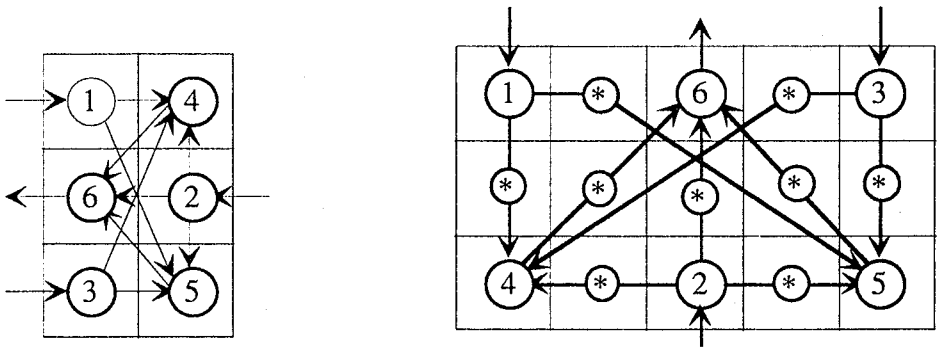


Fig 9 — *a) Placement neuronal du XOR* *b) Placement synaptique du XOR.*

La figure 10 montre l'activité obtenue pour le parallélisme synaptique. Nous pouvons noter une très forte activité en calcul (50% environ), qui est liée à un bon recouvrement entre les opérateurs et à la complexité des opérations effectuées : les calculs sont réalisés dans un format de nombre en virgule fixe que B. Faure a montré comme le meilleur compromis convergence/coût. L'exploitation d'un parallélisme neuronal (placement A) donne une activité moyenne inférieure (65%), mais l'activité en calcul reste la même (50%). Sur le XOR, le placement A fournit un résultat tous les 800 cycles, le placement B tous les 200 cycles.

Nous reviendrons sur le choix de la méthode de placement un peu plus loin.

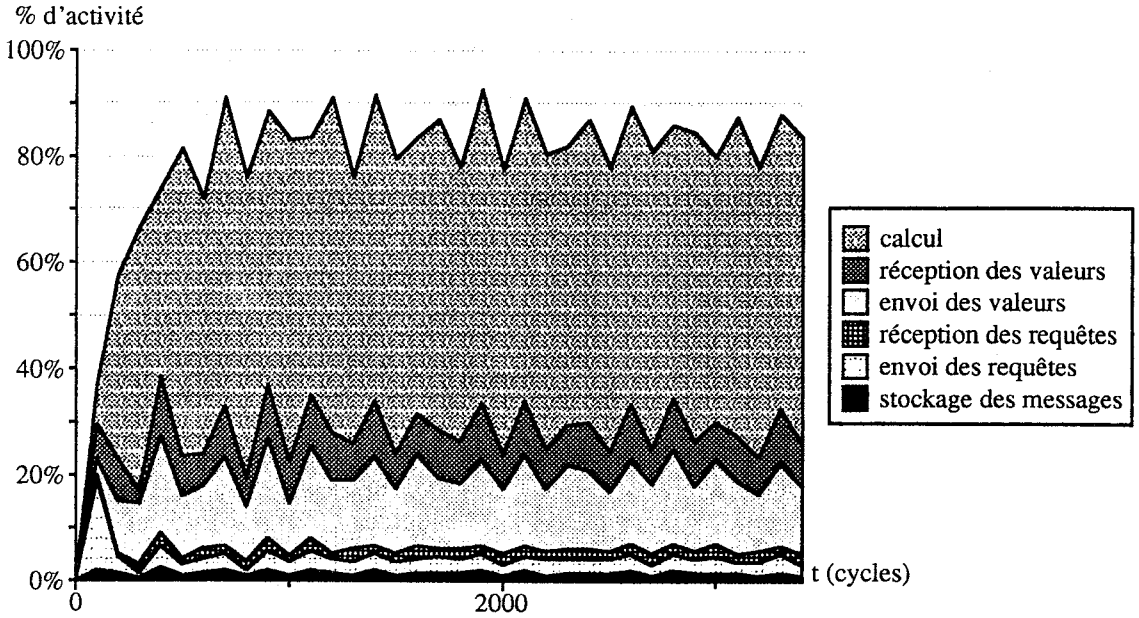


Fig. 10 — *Activité pour un parallélisme synaptique (placement B).*

3.4. Automates cellulaires.

Un automate cellulaire est un ensemble d'automates identiques disposés selon une topologie régulière et interagissant localement. Ces éléments de base, les cellules ou *sites* (pour éviter la confusion avec les cellules du réseau), évoluent de façon synchrone en calculant leur état suivant à partir de leur état courant et de celui de leurs voisins. Le voisinage est l'ensemble des règles présidant au choix des voisins parmi ceux permis par la topologie (distance entre voisins limitée).

Le concept d'automate cellulaire, introduit au début des années 50 par von Neuman et Ulam [NEU66], a beaucoup séduit par le contraste qu'il présente entre sa simplicité de réalisation et la complexité des motifs géométriques et comportementaux qu'il est capable de produire. Certains automates n'ont qu'un intérêt intellectuel et ludique, voire philosophique (J. von Neuman cherchait initialement à exhiber des phénomènes de réplication), mais d'autres ont une réelle utilité calculatoire. Il en va ainsi pour les modèles dits de *gaz sur réseau*, qui cherchent à étudier par une approche microscopique discrète des phénomènes d'écoulement macroscopiques et continus. Parmi ces modèles, le FHP [FRI86], que nous allons étudier plus en détail, est le modèle 2D le plus simple "préservant l'isotropie à un niveau macroscopique" ; le "gaz" artificiel simulé par l'automate se décrit par les mêmes équations que les gaz réels, celles de Navier-Stokes, et peut donc donner lieu aux mêmes phénomènes de turbulence. Cette technique de modélisation peut aussi être appliquée à de nombreux autres problèmes de physique, de thermodynamique, de chimie (diffusion, érosion, croissance de cristaux...), d'optique (interférence d'ondes, réfraction, réflexion) ou encore à la simulation de phénomènes macroscopiques comme les feux de forêts ; l'ouvrage de T. Toffoli et N. Margolus [TOF87] en présente une fort intéressante synthèse.

Les automates cellulaires sont typiquement des objets "SIMD", de programmation très facile, par exemple sur la Connection Machine. La principale difficulté consiste à concilier un fort parallélisme avec la possibilité de suivre l'évolution de l'automate : extraire à chaque instant l'état de chaque site d'un automate qui, en pratique, en comprendra au moins 1000 x 1000 est une tâche largement aussi ardue que le calcul lui même[1]. De plus, si on observe un gaz sur réseau à un niveau microscopique, on ne voit rien d'autre que le mouvement brownien des particules, dont il faut calculer la résultante sur des groupes de sites (*macro-site)* pour voir apparaître des phénomènes macroscopiques. Ce post-traitement est une autre tâche aussi lourde que le calcul.

La solution généralement proposée dans les architectures dédiées consiste à intégrer le dispositif de visualisation dans le chemin de données :

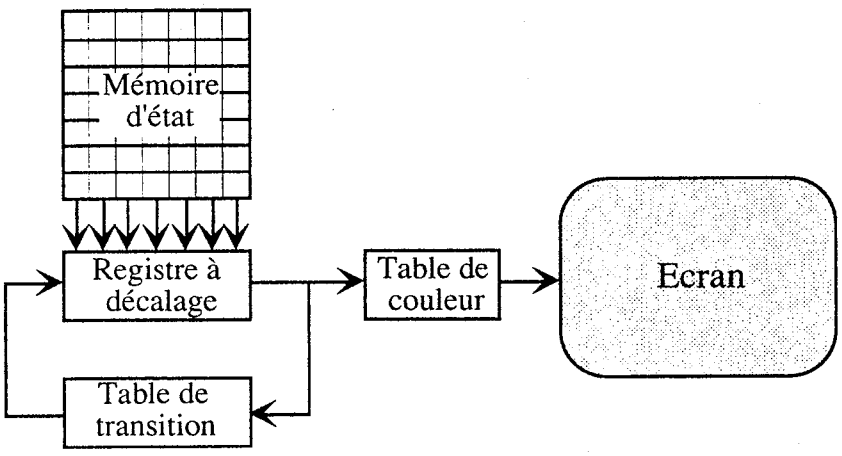


Fig. 11. — *Structure des machines à balayage.*

Le système fonctionne à la même vitesse que le dispositif de raffraichissement de l'écran (50 cycles par seconde), le tout étant synchronisé et cadencé par les signaux vidéo usuels. Cette structure a été adoptée, à quelques améliorations près, par les machines de la série CAM du M.I.T. [TOF87] et par la machine RAP-1 [CLO87]. Si le rapport performance/coût de ces architectures est imbattable, elles présentent un inconvénient primordial qui est leur incapacité à analyser par elles-mêmes les résultats qu'elles fournissent, ne serait-ce qu'en effectuant une moyenne au niveau macro-site. Nous allons analyser en détail les fonctions nécessaires à la mise en œuvre du modèle FHP et leur implémentation sur notre réseau cellulaire. Nous verrons que nous serons conduits à implémenter par logiciel une structure analogue à celle des machines à balayage.

---

1 certains phénomènes accessibles par d'autres méthodes de calcul (simulation spectrale) n'apparaissent pas sur des grilles de moins de 8192 x 8192 sites...

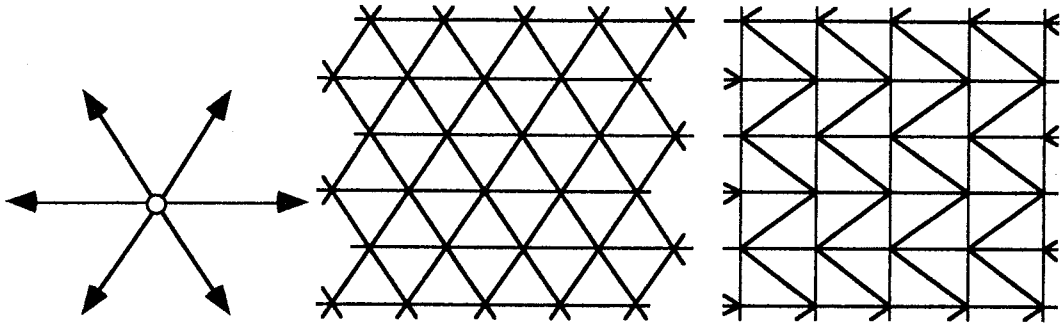


Fig. 12. *a) site élémentaire* *b) réseau hexagonal obtenu par interconnexion de sites* *c) réseau hexagonal placé sur un réseau orthogonal*

Chaque site *(fig. 12a)* est connecté à 6 voisins selon un motif régulier hexagonal *(fig. 12b)*. Ce réseau doit être un peu déformé pour être supporté par un réseau physique de nature orthogonale *(fig. 12c)*. Chaque lien peut véhiculer une particule du gaz, le centre du site étant vide[1]. L'automate évolue en deux phases distinctes :

- Une phase de *mouvement libre* au cours de laquelle les particules sortantes d'un site deviennent des particules entrantes des sites voisins ; si deux sites voisins s'envoient réciproquement une particule, il n'y a pas interaction entre les deux particules échangées.
- Une phase de collision au cours de laquelle les particules entrantes se collisionnent au centre du site, et deviennent des particules sortantes selon une configuration qui doit respecter le principe d'exclusion et un certain nombre d'invariants ; certaines configurations comme la collision frontale peuvent résulter en plusieurs configurations de sortie, dont une est choisie aléatoirement *(fig. 13)*. L'évolution de l'automate est donc en partie stochastique. Toutefois, la qualité d'aléa requise est médiocre, la plupart des implémentations se contentant de générer une valeur fixe pour chaque site lors de l'initialisation. Bien qu'ayant développé une version avec un générateur d'aléa de meilleure qualité, nous présentons ici les données relatives à un générateur fixe, afin de pouvons comparer le réseau cellulaire avec des autres implémentations.

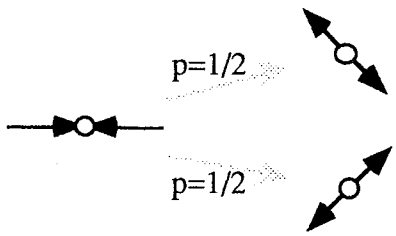


Fig. 13 : *Collision frontale entre deux particules.*

---

[1] le site n'est pas vide dans le modèle FHP2, une particule momentanément immobile peut l'occuper ; ce modèle, identique par tous les autres aspects au modèle FHP1 initial, permet d'atteindre des nombres de Reynolds plus élevé.

Chaque site, cycliquement, envoi les particules sortantes à ses voisins, puis récupère les particules entrantes. La configuration d'entrée est utilisée pour adresser une table de transition, qui donne la configuration de sortie pour le cycle suivant. La table de transition est donc une table à $(6+1+1)^2$ entrées : un bit pour chaque direction, un bit d'aléa et un dernier bit pour coder les sites particuliers (bords, obstacles). L'algorithme exécuté par chaque cycle peut s'écrire de la façon suivante :

```
site :
  entrees E[6]
  sorties S[6]
  entiers i,etat,sort
  /* etat=[ns,s5,s4,s3,s2,s1,s0] où ns=nature du site */
  pour i:=0 à 5 /* envois initiaux */
     Envoyer (S[i],etat mod 2)
     etat:=etat/2
  cycle
     /* etat=[ns] */
     etat:=etat*2+aleatoire([0,1])
     pour i:=0 à 5 /* construction de la configuration d'entrée */
        config:=config*2+Recevoir(E[i])
     /* etat=[ns,alea,e0,e1,e2,e3,e4,e5] */
     etat:=transition[config]
     /* etat=[ns,s5,s4,s3,s2,s1,s0] */
     pour i:=0 à 5 /* envois des particules sortantes */
        S[i]:=etat mod 2
        etat:=etat/2
  fincycle
```

Nous pouvons envisager deux manières de programmer cet algorithme sur le réseau.

a) Affectation statique site - cellule. Il s'agit d'une programmation de type purement SIMD, où le traitement de chaque site se voit alloué à un groupe de processeurs. Le premier problème rencontré réside dans la quantité d'information à mémoriser pour chaque site : le programme de mouvement des particules (6 émissions, 6 réceptions), la table de transition. La taille de la table de transition est réduite à 128 entrées car le bit d'aléa est constant. Le site peut être programmé sur une seule cellule.

Le premier problème réside dans la difficulté d'extraction des résultats. La méthode la plus simple consiste à prendre périodiquement un "cliché" : les automates effectuent un nombre fixe de transitions, puis passent dans un état d'extraction où ils se chaînent en une structure à décalage. Une marque de fin de flux, injectée par l'hôte, permet aux automates de reprendre le cours normal de leur exécution, pour une nouvelle période.

Le second problème est le nombre de cellules nécessaires à la réalisation d'un réseau utile. Un réseau de 1000 x 1000 sites est un strict minimum, les physiciens estiment qu'une bonne taille serait 4000 x4000. Les cellules vont alors se compter par millions, total qui dépasse les objectifs les plus ambitieux. Une version affectant un ensemble de sites à un groupe de traitement devrait alors être étudiée, conduisant à un sur-coût en contrôle qui risque fort d'être considérable.

b) Structure à circulation des sites. Cette version, directement inspirée des machines à balayages et des réseaux systoliques, représente une solution élégante aux problèmes liés à une affectation fixe. Elle consiste à n'avoir qu'un opérateur de transition par ligne de sites, laquelle y sera injectée sériellement *(fig. 14)*.

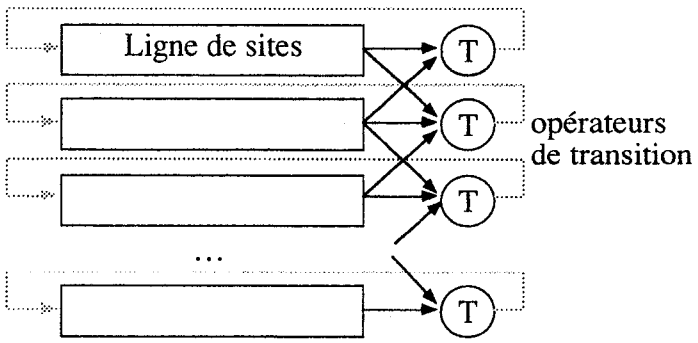


Fig. 14 — *Structure à circulation des sites.*

La fonction de mémorisation peut être assurée par des cellules programmées en registres à décalages (200 sites par cellules), ou par de la mémoire RAM externe (voir annexe 3) . Nous travaillerons sur l'hypothèse d'une mémorisation par cellules. La réalisation d'un opérateur de traitement comprend 4 cellules :

- Une cellule de mouvement : cette cellule découpe les états des sites en trois flux, un pour la ligne supérieure, un autre pour la ligne inférieure, et un dernier pour la ligne courante.

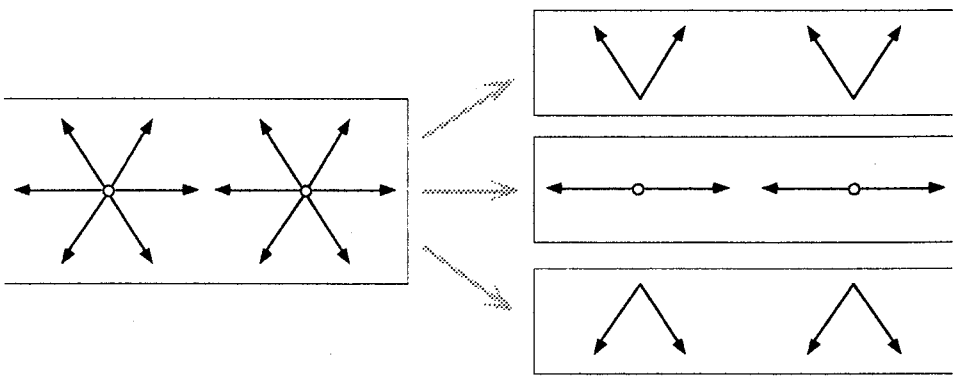


Fig.15 — *Mouvement des particules.*

- Une cellule de calcul de l'entrée dans la table de transition, qui regroupe trois flux pour reconstituer la configuration d'incidence de chaque site.

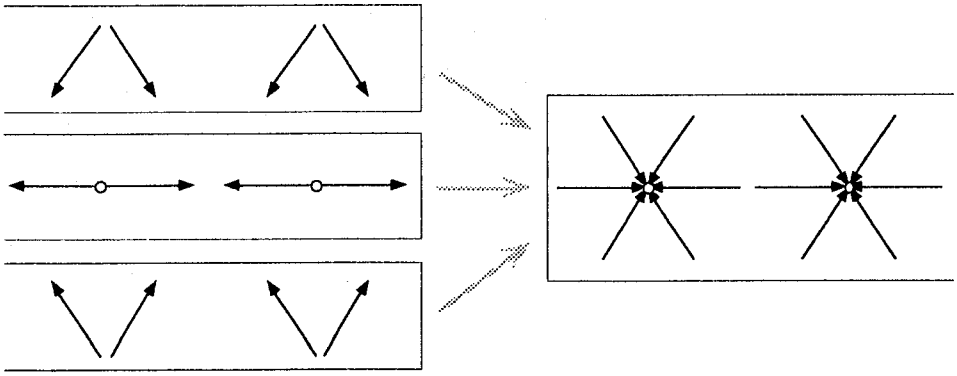


Fig. 16 — *Constitution du flux de collisions.*

- Deux cellules contenant la table de transition ainsi que le code pour les consulter.

Pour un réseau de 1000 par 1000 sites, il faut environ 10000 cellules, et 100 000 cellules pour un réseau de 4000 par 4000 ; ces tailles sont beaucoup plus raisonnables que celles impliquées par stratégie l'allocation précédente. Les tableaux 3 et 4 présentent un comparatif des performances de diverses machines pour l'évaluation d'un réseau 1000 x 1000. Le réseau cellulaire est mesuré pour l'algorithme à circulation (pour un étage de traitement et 4 étages de traitement, avec mise en commun de certains opérateurs sur deux lignes pour un bon équilibrage de charge) et pour l'algorithme à allocation fixe. Il est intéressant de voir que l'algorithme à circulation permet des performances par processeur supérieures pour une bien meilleure utilisation de la capacité de mémorisation (mém./site : taille totale de la mémoire divisé par le nombre de sites). L'allocation la plus évidente n'est donc pas forcément la plus efficace.

pour un réseau 1024 x 1024	Réseau cellulaire 1 étage d'opérateurs	Réseau cellulaire 4 étages d'opérateurs	Réseau cellulaire allocation fixe
nb. proc. trait.	3096	14336	10^6
nb. proc. total	8192	19456	10^6
mém./site	2 octets	4,75	256
sites traités / s	98.10^6	384.10^6	21.10^9
sites traités / s par processeur	12.10^3	22.10^3	21.10^3

Tableau 3 — *Performances du réseau cellulaire.*

Le tableau 3 donne les performances obtenues sur d'autres architectures : sur un IBM 3090 avec accélérateur vectoriel, sur une carte à 4 transputers, sur la machine

RAP1, sur la Connection Machine, et avec le Princeton chip (circuit dédié à l'exécution de ce genre d'automates, basé sur un système de registre à décalage, un peu comme la CAM et le RAP). La seconde valeur de performance pour ce circuit est celle obtenue dans un prototype de machine où le bus l'hôte, un SUN-3, est une limitation .

pour un réseau 1024 x 1024	IBM 3090 / VF	transputers	RAP1 (1)	Princeton chip [WAY88]	CM-1 [WAY88]
nb. proc	1	4			64K
sites traités / s	15.10^6	$0,25.10^6$	$6,5.10^6$		1000.10^6
sites traités / s par processeur	15.10^6	60.10^3	$6,5.10^6$	20.10^6 (666.10^3)	15.10^3
extensibité du réseau	oui (lié à la mémoire)	oui (lié à la mémoire)	non	partielle (sur 1 dimension)	oui
extensibilité des performances	faible (<10 proc)	faible (<10 proc)	non	oui	oui

(1) Limitation à 256 x 512 à cause du couplage avec la vidéo.

Tableau 4 — *Performances de diverses architectures.*

Le réseau cellulaire permet des performances supérieures à toutes les autres machines hormis la CM-1 et le Princeton Chip. Sur ce problème, chaque cellule s'avère aussi performante qu'un transputer. L'activité en calcul sur ce problème est de l'ordre de 20%, ce qui est tout à fait satisfaisant pour un programme manipulant des données aussi petites *(fig. 18)*.

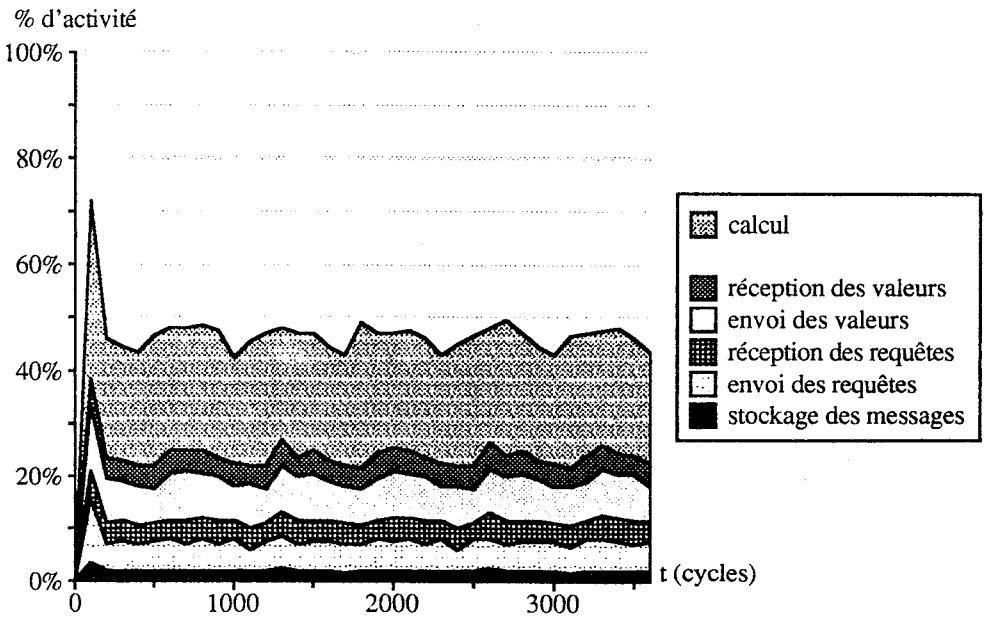


Fig. 17 — *Activité pour le Lattice Gaz Model.*

Mais la solution à circulation présente d'autres avantages qui la rendent plus praticable :

a) Accessibilité des résultats : les états des sites circulant dans le réseau de manière cyclique pour chaque ligne, ils sont tous accessibles séquentiellement. Bien entendu, même sérialisés, le débit d'entrées/sorties correspondant est considérable, mais nous pouvons le réduire en faisant effectuer par le réseau un travail d'élaboration préalable. L'analyse des résultats est un point noir dans nombre d'architectures spécialisées, comme le Princeton chip, où l'intervention de l'hôte est requise et constitue une limitation. Comme nous l'avons souligné, seule la vélocité moyenne des particules présente une signification. Nous pouvons confier cette tâche de calcul de moyenne au réseau : moyenne en X, en Y, et éventuellement temporelle.

La moyenne en X est facile à calculer, en insérant une cellule à la sortie de chaque opérateur de transition. M étant le nombre de valeurs à moyenner, le calcul de moyenne (ou plus exactement la sommation) en X correspond au programme suivant :

```
sumX :
  envoyer(req)
  cycle
     pour d:=1 à 6
        t[d]:=0
     pour i:=1 à M
        e:=recevoir(etat)
        envoyer(req)
        pour d:=1 à 6
           t[d]:=t[d]+ e mod 2
           e:=e/2
     recevoir(reqmoy)
     pour d:=1 à 6
        envoyer(t[d])
```

La sommation en Y requiert de combiner les sommes en X issues des cellules précédentes à un instant donné. Pour obtenir un débit maximum, il faudrait faire une addition arborescente, mais un tel débit n'est pas nécessaire, la fréquence des sommes en X étant N fois inférieures à la fréquence de transition. Une addition séquentielle, ou au pire sur deux étages, est donc suffisante.

La sommation temporelle est plus difficile à réaliser : il faut retenir les sommes partielles des sommes X-Y, ce qui représente, si NxN est la taille du réseau, $6.(N/M)^2$ valeurs. Comme ni la fréquence, ni la taille ne sont identiques, nous les ferons circuler dans un tableau à décalage distinct, plutôt que dans le réseau, parallèlement aux sites.

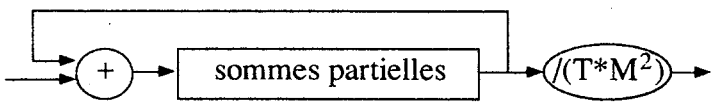


Fig. 18 — *Sommation temporelle.*

Nous obtenons alors le "tunnel d'expérimentation" suivant :

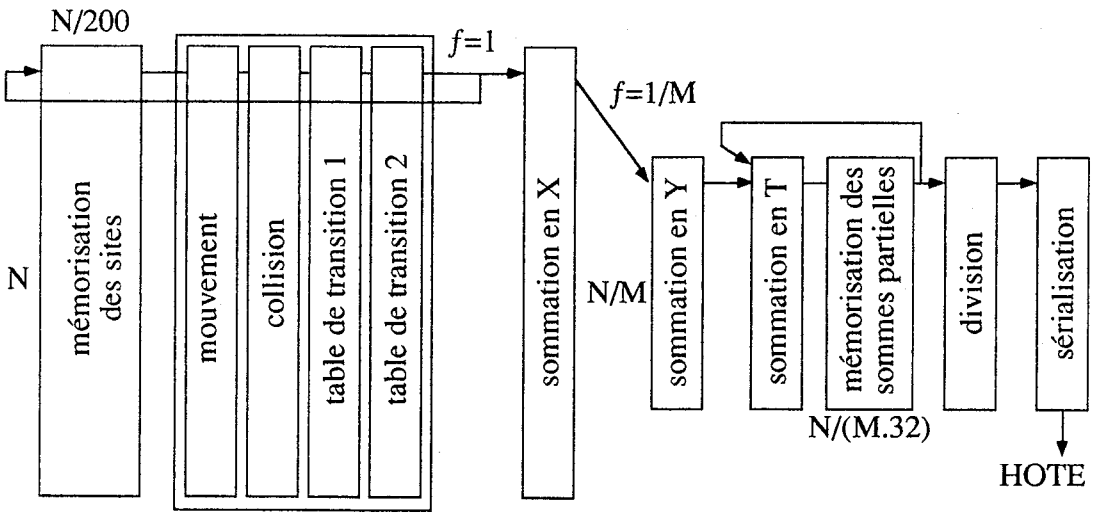


Fig. 19 — *Tunnel d'expérimentation.*

b) Le second avantage de l'algorithme à circulation sur l'algorithme à allocation fixe est une extensibilité des performances, par chaînage de plusieurs étages de transition. Il faut toutefois modifier le système de sommation en X : une série de k étages de transitions ne permet pas, en sortie, d'observer les états que toutes les k transitions ; il faut donc intercaler des opérateurs de sommations entre chaque étage et en sommer ensuite le résultat.

Les fonctions de mémorisation n'étant pas dupliquées lors de l'ajout d'étages de traitement, le gain de performances est en quelque sorte superlinéaire : les performances par processeur convergent vers celle d'une implémentation mettant en œuvre une RAM externe au réseau cellulaire.

Le seul véritable problème de l'algorithme à circulation est le facteur de forme, qui a tendance à être très allongé (beaucoup de lignes et peu de colonnes). Ce défaut peut être corrigé par un repliage en serpentin ou en colimaçon.

Cette étude cas montre l'intérêt que peut revêtir l'utilisation d'un mode de fonctionnement MIMD pour un problème régulier synchrone : une implémentation, qui combine un parallélisme géométrique limité et un parallélisme pipeline, donne une accélération supérieure à celle obtenu avec un parallélisme géométrique normal bidimensionnel, avec en supplément un gain de souplesse dans les performances et l'accès aux résultats, ainsi qu'un coût minimum nettement plus raisonnable.

4. Génération automatique.

La programmation de machines comme le réseau cellulaire n'est pas sans présenter une parenté certaine avec le dessin de circuits intégrés. La programmation manuelle est en quelque sorte l'équivalent du dessin au micron. Les concepteurs en micro-électronique se sont rapidement aperçus de la récurrence de certaines structures et fonctions, sur lesquelles une connaissance manuellement acquise pouvait et devait être capitalisée. Cette capitalisation peut se faire à deux niveaux différents au sein des systèmes de C.A.O. Les éléments fonctionnels peuvent être rangés au sein de bibliothèques de cellules plus ou moins paramétrables. Les méthodes peuvent faire l'objet de générateurs automatiques où l'expertise est encapsulée sous forme d'algorithmes : générateurs de PLA par exemple.

4.1. Générateurs spécifiques.

Le premier type de générateurs auxquels nous pouvons penser s'intéresse à la prise en compte automatique de certains aspects de la parallélisation d'un algorithme donné. L'aspect généralement concerné par une automatisation est, bien sur, le placement des tâches sur le réseau dans le cas où celui-ci est non trivial : graphes plus ou moins aléatoires, topologies de dimension supérieure à 2, arbres. Un tel outil de placement peut prendre en compte la nécessité d'insérer des relais d'acheminement . Le placement étant un problème de forte complexité, le réseau cellulaire peut être mis à profit pour effectuer, par exemple, un recuit simulé. Le générateur, une fois l'affectation des tâches réalisée, va produire le code assembleur ou objet de chaque cellule, en y intégrant les tables de messages qui décrivent la topologie et en initialisant certaines variables locales dont la valeur n'est pas homogène. Le corps du programme des cellules peut être homogène ou bien être particularisé en fonction du type du nœud correspondant dans le graphe (en simulation logique : porte NOR, NAND, etc, dans un réseau synaptique : neurone ou synapse).

Parmi les exemples que nous avons traités, ceux qui sont concernés par ce type de générateurs sont :

- les programmes de recherche de plus court chemin, pour lesquels un générateur simple a été effectivement rédigé (placement des nœuds du graphe sur la grille)
- les réseaux de neurones (placement) : des outils de placement par recuit simulé, et de limitation du degré de connection ont été écrits.
- le tri par interclassement (placement d'un arbre)
- la simulation logique à échéancier (placement et génération des fonctions) : un générateur existe qui n'assure pour l'instant que la génération des fonctions, le placement étant manuel.

4.2. Générateurs généraux.

Certaines phases des générateurs spécifiques sont très ressemblantes. Les phases d'équilibrage de charge, de placement, d'ordonnancement sont de même nature quelles que soient les fonctions implémentées par les processus, et peuvent faire l'objet d'un outil non-spécifique. A priori, l'outil de placement de LUSTRE, qui par recuit simulé effectue d'un seul bloc ces différentes tâches, peut être réutilisé dans cadre plus général. Toutefois, le coût de l'algorithme de recuit simulé [SIA85] (ou de tout autre algorithme remplissant le même rôle) est très élevé, et il faudrait continuer réfléchir à une implémentation sur le réseau cellulaire, en prolongeant l'étude menée par R. Cornu-Emieux [COR88] en vue de la réalisation d'un réseau cellulaire dédié au placement.

4.3. Compilateurs spécifiques.

Dans ce type de générateur, on cherche à intégrer une expertise beaucoup plus complexe, qui lui fait mériter le qualificatif de compilateur. Il s'agit non plus de générer le code pour un problème donné selon un algorithme donné et une implémentation donnée, mais de savoir pour un même problème choisir la meilleure technique, après analyse de certains paramètres jugés caractéristiques.

Nous allons expliciter cette idée sur le problème des réseaux de neurones. Nous avons vu § 3.3 que de nombreuses variantes d'implémentation peuvent être imaginées. Pour décider d'une implémentation, il faut connaître a) la topologie du réseau neuronal, et en particulier sa connectivité (graphe complet ou graphe multi-couches) b) le nombre de processeurs disponibles.

Si le réseau est très fortement connecté, la technique la plus efficace est vraisemblablement du type multiplication de matrices (que nous savons traiter de manière systolique) et, éventuellement, multiplication de matrices creuses (aussi traitée selon une technique analogue). Lorsque la connectivité devient faible, les matrices correspondantes deviennent très creuses, mais le caractère figé du remplissage des matrices utilisées permet un traitement de type réseau d'opérateurs. Toutefois les neurones formels restent généralement trop fortement connectés, et il faut les distribuer sur plusieurs cellules, avec plusieurs placements possibles.

Nous savons aussi que les réseaux multi-couches avec apprentissage par rétropropagation du gradient n'autorisent un parallélisme entre couches qu'en phase de reconnaissance. En phase d'apprentissage, un nouveau jeu de données ne peut être soumis en entrée que lorsque le jeu précédent a traversé le réseau et que l'erreur induite a été rétro-propagée en sens inverse. Les couches sont donc évaluées les une après les autres, entraînant une nette sous-activité du réseau proportionnelle au nombre de couches. Dans ce cas, l'association fixe neurone-groupe de processeurs n'est sans doute plus une association optimale. Contrairement au réseau cellulaire spécifique, le réseau général permet l'utilisation de méthode totalement différente pour les deux phases. Il est possible par exemple de privilégier un parallélisme de couche (placement A) pour la reconnaissance, et un parallélisme d'évaluation avec multiplexage des couches sur le

même ensemble de processeurs pour la phase d'apprentissage (placement plus ou moins de type D), d'autant plus que la phase d'apprentissage induit un code plus complexe, donc éventuellement distribué.

Une différence sensible de performance - et de nombre de cellules - peut être obtenue selon les méthodes de placement. Le rôle d'un compilateur de réseau de neurones serait de choisir des divers paramètres la meilleure méthode, puis d'effectuer un placement eu une génération de code adéquate. Il doit pouvoir inférer la taille du réseau nécessaire en fonction du graphe et des performances escomptées, ou inversement exploiter au mieux un nombre donné de cellules.

Ce genre d'expertise doit pouvoir être dégagé pour d'autre classes de problèmes, en particulier ceux où la même question du bon algorithme est liée à la connectivité d'un graphe. Par exemple, pour la recherche de plus court chemin, le même problème de placement et de scission se pose lorsque le graphe est connecté de façon assez dense ; des techniques de même type doivent pouvoir être mises en œuvre. Les générateurs qui en découlent représente la forme la plus élaborée de capitalisation de l'acquis en matière de programmation.

5. Conclusion.

Nous n'avons pas dégagé ici de méthodologie générale de programmation, et encore moins cherché à définir un langage polyvalent. La recherche d'un tel langage, performant sur tous les problèmes, capable d'extraire automatiquement le parallélisme, sémantiquement propre, (et de surcroît qui soit un sur-ensemble de FORTRAN...) n'est-elle pas une entreprise vouée à un succès très relatif ? Pour notre machine en tout cas, la programmation par langage parallèle de haut niveau est réellement problématique, bien que des solutions partielles comme LUSTRE soient encourageante. La programmation explicite de chaque PE, bien que nécessitant un peu d'habitude, reste néanmoins une méthode tout à fait praticable sur les problèmes qui présentent une certaine régularité. Les principales difficultés relèvent de l'algorithmique parallèle (synchronisation, ordonnancement) et nous nous y heurterions de la même manière avec des langages de type OCCAM.

Une approche "perpendiculaire" à l'approche langage semble plus prometteuse. Nous savons bien traiter certaines applications, ou classes d'applications, et ce savoir peut faire l'objet de générateurs automatiques donnant du code d'aussi bonne qualité qu'avec une programmation manuelle. Certaines phases de la programmation manuelle peuvent aussi être automatisées (placement). Enfin, des analyses de plus haut niveau spécifiques à des applications, mais trop lourdes pour être assurées manuellement, peuvent aussi être automatisées.

Nous avons donc une programmation du réseau cellulaire qui tient à la fois de la CAO de circuits intégrés et de la programmation classique, sans pour autant être réductible à l'une ou à l'autre. Le sujet est très largement ouvert et peut sans doute faire l'objet de solutions imaginatives.

Conclusion

Nous avons dans le chapitre I tenté de définir une méthodologie adaptée à la conception de machines parallèles, méthodologie qui procède par générations successives de machines. Dans ce schéma, les premiers réseaux cellulaires dédiés représentent une première génération. Les principaux enseignements que nous en avons tirés sont que la structure de réseau cellulaire est exploitable d'un point de vue algorithmique et architectural, qu'elle permet des performances intéressantes par rapport aux autres types de machines [LAT89], mais que la conception de machines spécifiques coûte cher en développement (analyse fonctionnelle et tracé du circuit) et en fabrication (il faut pour chaque application fondre plusieurs milliers de circuits). Par ailleurs, il s'est avéré que la complexité des cellules restait importante, de l'ordre de la dizaine de milliers de transistors, complexité proche de celle d'un petit processeur.

Nous avons en conséquence mis de côté l'aspect dédié, tout en conservant la structure cellulaire, pour chercher dans une seconde génération à réaliser une machine à vocation générale. Nous débouchons assez rapidement sur des solutions de partie traitement à base de processeurs 8 bits et de mémoire de quelques centaines d'octets, qui sont en soit tout à fait originales par rapport aux autres machines parallèles, se situant à un niveau intermédiaire entre machines SIMD 1 bits et hypercubes de processeurs 32 bits MIMD. Nous ne pourrons bien entendu pas espérer atteindre le même niveau de performances que celui obtenu par des réseaux spécifiques, mais la programmabilité des cellules pose plusieurs questions nouvelles que nous avons en partiellement traitées dans le présent travail :

- Comment équilibrer au mieux les différentes composantes de la cellule, alors que les applications cibles ne sont pas connues de façon extensive et présentent a priori des besoins peu homogènes ? Des contraintes matérielles doivent être prises en compte (incrémentabilité, packaging, capacité à suivre l'évolution technologique).
- L'algorithme de chaque cellule peut maintenant être totalement différent de celui de sa voisine et, éventuellement, être modifié dynamiquement. Comment mettre cette caractéristique à profit ?
- Quels outils logiciels faut-il mettre en œuvre pour programmer une machine aussi radicalement différente des machines séquentielles ?

Ce mémoire aborde le problème de la programmation d'un réseau cellulaire général, au travers de l'étude de divers algorithmes, toujours à un niveau assez bas (donnée explicite du programme de chaque cellule). Un simulateur niveau cycle et un assembleur "parallèle" ont été écrits pour cette occasion, permettant d'analyser et de mesurer le comportement de ces programmes. E. Payan a précédemment étudié le même problème à un niveau plus haut (extraction automatique du parallélisme), avec comme support le langage data-flow LUSTRE [PAY90], et l'étude de l'utilisation de langages d'acteurs est actuellement en cours (Y. Latrous). La question reste néanmoins largement ouverte.

Les données qualitatives et quantitatives acquises grâce à cette expérience de la programmation ont servi de base de référence pour un travail de dimensionnement relatif des diverses composantes (système de communication, processeur, mémoire). Nous avons vu que le système de routage initial (transfert parallèle) est sur-dimensionné par rapport aux besoins maximum des programmes. Inversement, il est coûteux et surtout incompatible avec les techniques de packaging actuellement disponibles. Des solutions moins coûteuses existent, certaines configurations apparaissant comme des compromis très satisfaisants. La solution la plus intéressante pour les circuits multi-cellules, tant en performances qu'en coût (structure à bus), requièrerait toutefois l'abandon de l'architecture initiale qui dotait chaque cellule d'un routeur : il n'y aurait plus qu'un seul routeur par chip.

Le contenu que nous avons défini pour le processeur diffère assez fortement de celui des processeurs pour machines séquentielles. Une analyse approfondie du contexte dans lequel il s'insère nous a permis de cerner, autant que notre expérience de la programmation nous le permettait, quelles sont les fonctionnalités importantes et quelles sont celles qui sont inutiles ; nous avons ainsi tenter d'optimiser le rapport performance/coût en conservant une cellule simple (et donc un nombre de cellules par chip potentiellement élevé). Les instructions de communication définies permettent une programmation assez souple et efficace, pour un coût matériel très limité.

Une implémentation sur silicium de cette version par M. Karabernou est en cours d'achèvement et nous pourrons bientôt évaluer le réseau cellulaire autrement que par simulation.

Globalement, nous avons montré la possibilité de réaliser une architecture parallèle MIMD à grain matériel fin (circuits multi-cellules, très petite mémoire locale, message court et de taille fixe...) et son adéquation à un parallélisme logiciel à grain fin. Il reste un travail important à effectuer pour préciser l'étendue de cette adéquation, ce qui passe par l'étude de langages et de méthodes de programmation, et de leur impact éventuel sur l'architecture matérielle. Des mécanismes nouveaux s'avéreront éventuellement nécessaires, et viendront compléter ou remplacer ceux présents dans la version définie ici ; c'est peut-être une troisième génération de réseau cellulaire, après les réseaux dédiés et celui que nous avons défini ici, qui reste à élaborer...

Bibliographie

[AKT91] C. Aktouf, S.M. Karabernou, G. Mazaré, P. Rubini, *VLSI design and testing of a massively parallel processor,* Proc. of the Intl. Conf. on Microelectronics, Le Caire, déc 1991.

[AGH86] G. Agha, C. Hewitt, *Concurrent programming using actors: exploiting large-scale parallelism*, Readings in Distributed Artificial Intelligence, Morgan Kaufmann publishers, inc., juin 1986.

[ANC86] F. Anceau, *The Architecture of Microprocessors*, Addison-Wesley 1986.

[ATH86] W. C. Athas, C. L. Seitz, *Cantor User Report version 2.0*, Computer Science Department, California Institute of Technology, 5232:TR:86, 1986.

[ATH87a] W.C. Athas, *Fine grain parallel computations*, Ph.D thesis, California Institute of Technology, 1987.

[ATH87b] W.C. Athas, C.L. Seitz, *Multicomputers*, Department of Computer Science, California Institute of Technology, Technical Report 5244:TR:87, juin 1987.

[ATH88] W.C. Athas, C.L. Seitz, *Multicomputers : Message passing concurrent computers*, IEEE Computer, Vol. 21, n°8, août 1988, pp 9-24.

[AUG85] M. Auguin, F. Boéri, *Réseaux d'interconnextion et leurs commandes asynchrones,* in *Parallélisme, communication et synchronisation,* Editions du CNRS, 1985, pp 489-517.

[BAD89] H.G. Badr, S. Podar, *An optimal shortest-path routing policy for networks with regular mesh connected topologies*, IEEE Transactions on Computers, vol. 38, n°10, octobre 1989, pp 1362-1371.

[BER85] J-P. Bernard, *Etude d'une machine cellulaire pour la simulation logique de circuits intégrés*, thèse de docteur-ingénieur, Grenoble, 1985.

[BOI91] F. Boiteux, M. Robichon, *Réalisation d'une interface matérielle et logicielle entre un ordinateur hôte Macintosh II et un réseau parallèle cellulaire*, rapport de stage de fin d'études, ENSIMAG-ENSERG, section Architecture des systèmes numériques, juin 1991.

[BRI91] J. Briat, M. Favre, C. Geyer, J. Chassin de Kergommeaux, *Scheduling of OR-parallel Prolog or a Scalable, Reconfigurable, Distributed Memory Multiprocessor,* in PARLE 1991.

[BRO80] S.A. Browning, *The tree machine : a highly concurrent computing environment*, Technical report TR-3760, California Institute of Technology, 1980.

[CAR88] D.A Carlson, *Modified mesh-connected parallel computers,* IEEE Transactions on Computers, vol. 37, n°10, octobre 1988, pp 1315-1321.

[CAS87] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, *LUSTRE, a déclarative language for real-time programming*, proc. conf. Principles of Programming Languages, Munich, 1987.

[CHA82] K.M. Chandy, J. Misra, *Distributed Computation on graphs: Shortest path algorithms,* CACM, vol. 25, n° 11, nov. 1982, pp. 833-837.

[CHA89] J. Chassin, P. Codognet, P. Robert, et J.-C. Syre, *Programmation Logique Parallèle*, Technique et Science Informatique (TSI), Vol. 8, n° 3 et 4, 1989.

[CHE90] M-Y. Chen, K.G. Shin, D.D. Kandlur, *Adressing, routing, and broadcasting in hexagonal mesh multiprocessors*, IEEE Transactions on Computers, vol. 39, n°1, janvier 1990, pp 11-18.

[CLO87] A. Clouqueur, D. d'Humières, *RAP1, a cellular automaton machine for fluild dynamics*, Complex Systems 1 (1987) pp. 585-597.

[COD91] B. Codenotti, R. Tamassia, *A network flow approach to the reconfiguration of VLSI arrays*, IEEE Transactions on Computers, vol. 40, n°1, janvier 1991, pp 118-121.

[COR87] R. Cornu-Emieux, G. Mazaré, Ph. Objois, *An integrated highly parallel architecture to accelerate logical simulation*, proceedings de IEEE ISELDECS 87, décembre 1987.

[COR88] R. Cornu-Emieux, *Réseau de cellules intégré : Etude d'architecture pour des applications de CAO de VLSI,* thèse de doctorat en microélectronique, I.N.P. Grenoble, septembre 1988.

[DAL86] W.J. Dally, *A VLSI Architecture for Concurrent Data Structures*, Ph.D thesis, Department of Computer Science, California Institute of Technology, Technical Report 5209:TR:86, mars 1986.

[DAL87a] W.J. Dally, C.L. Seitz, *Deadlock-free message routing in multiprocessor interconnection networks*, IEEE Transactions on Computers, vol. 36, n°5, mai 1987, pp 547-553.

[DAL87b] W. J. Dally, *Architecture of a Message-Driven Processor*, Massachussets Institute of Technology, in proc. of the 14th Annual Symposium on Computer Architecture ACM 1987, pp. 189-196.

[DAL90] W.J. Dally, *Performance analysis of k-ary n-cube interconnection networks*, IEEE Transactions on Computers, vol. 39, n°6, juin 1990, pp 775-785.

[DAL91] W.J. Dally, *Express cubes: improving the performance of the k-ary n-cube interconnection networks*, IEEE Transactions on Computers, vol. 40, n°9, September 1991, pp. 1016-1023.

[DEL90] D. Delesalle, D. Trystram, D. Wensek, *Tous ce que vous voulez savoir sur la Connection Machine (sans oser le demander)*, document interne LMC, avril 90.

[DIJ59] E.W. Dijkstra, *A note on two problems in connexion with graphs*, Numerishe Mathematik, vol. 1, 1959, pp. 269-271.

[FAU90] B. Faure, G. Mazaré, *Neural networks : a cellular architecture*, proceedings of the International Conference "Neural Networks, Biological Computers or Electronic Brains", AFCET - Entretiens de Lyon, Mars 1990.

[FIS88] S. Fiske, W.J. Dally, *The Reconfigurable Arithmetic Processor*, Artificial Intelligence Laboratory and Laboratory for Computer Science, Massachussets Institute of Technology, IEEE 1988.

[FLA87] C.M. Flaig, *VLSI mesh routing systems*, Department of Computer Science, California Institute of Technology, Technical Report 5241:TR:87, mai 1987.

[FLY72] M.J. Flynn, *Some computer organizations and their effectiveness*, IEEE Transaction on Computers, vol. 21, n°9, septembre 1972.

[FPS86] Floating Point Systems, *T-Series User Guide*, Beaverton, 1986.

[FRI86] U. Frish, B. Hasslacher, Y. Pommeau, Phys. Rev. Lett. n° 56, p. 1065 (1986).

[GER89] C. Germain, *Etude des mécanismes de communication pour une machine massivment parallèle : MEGA*, thèse de l'Université Paris XI, décembre 1989.

[GER90] C. Germain, *Une stratégie de routage pour la machine à parallélisme massif MEGA*, Actes du 2ème Symposium PRC Architectures Nouvelles de Machines, Toulouse, septembre 1990, pp 3-16.

[GOL83] A. Goldberg, D. Robson, *Smalltalk-80 : The Language and it's Implementation*, Addison Wesley 1983.

[GOR87] D. Gordon, *Efficient embeddings of binary trees in VLSI arrays*, IEEE Transactions on Computers, vol. 36, n°9, septembre 1987, pp 1009-1018.

[HAR86] J.G. Harp, C.R. Jesshope, T. Muntean, C. Whitby-Strevens, *The development and application of a low cost high performance multiprocessor machine : Supernode Project*, ESPRIT'86, Bruxelles.

[HEN83] J.L. Hennessy et al., *MIPS: a VLSI processor architecture*, Tech. Rep. n° 223, Computer Systems Laboratory, Stantford University, June 1983.

[HIL85] W.D. Hillis, *The Connection Machine*, MIT Press, Cambridge, mai1985.

[HOA87] C.A.R. Hoare, *Communicating Sequential Processes*, Communication of ACM, vol. 21, n°8, pp. 666-677, 1978.

[ING90] P. Ingels, M. Raynal, *Simulation répartie : schémas d'exécution pour un modèle à processus*, Technique et Science Informatiques, vol. 9, n°5, mai 1990, pp 383-397.

[INM84] Inmos Ltd., *Occam Programming Manual*, Prentice Hall 1984.

[JAC91] O. Jacquiot, *Mise en œuvre d'une interface et étude d'algorithmes de simulations distribuées pour un réseau cellulaire,* rapport de DEA d'Informatique, INPG, Grenoble, 1991.

[KAR89] S.M. Karabernou, *Etude et réalisation d'un mécanisme d'acheminement dans un réseau cellulaire,* rapport de DEA de Microélectronique, INPG, Grenoble, 1989.

[KER79] P. Kermani, L. Leonard, *Virtual cut-through: a new computer communication switching technique*, Computer networks, vol. 3, 1979, pp 267-286.

[LAT89] D. Lattard, *Architecture massivement parallèle : Un réseau de cellules intégré pour la reconstitution d'images*, thèse de doctorat en microélectronique, I.N.P. Grenoble, novembre 1989.

[LAV89] D. Lavenier, *Un réseau systolique linéaire programmable pour le traitement des chaîne de caractères*, thèse de doctorat en informatique, Université de Rennes I, juin 1989.

[LIT90] D. Litaize, O. Hammami, P. Sainrat, R. Pulou, *Les liaisons série du multiprocesseur M3S - Justification, problèmes techniques, solutions*, Actes du 2ème Symposium PRC Architectures Nouvelles de Machines, Toulouse, septembre 1990, pp 243-265.

[LUS88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B~Haussman, *The Aurora Or-Parallel Prolog System,* International Conference on Fifth Generation Computer Systems 1988, ICOT, Tokyo, November 1988.

[LUT84] C. Lutz, S. Rabin, C. Seitz, D. Speck, *Design of the MOSAIC Element*, Conf. on Advanced Research in VLSI, M.I.T., 1984.

[MEA80] C.A. Mead, L.A. Conway, "Introduction to VLSI Systems", Addison-Wesley, 1980.

[MEH88] J. Méhat, A. Mérigot, *Control and programmation of the SPHINX pyramid machine,* Proc. of the 2nd Symp. on the Frontiers of Massively Parallel Computation, pp. 423-428, Fairfax, Virginia, 1988.

[MUN89] T. Muntean, *SUPERNODE : Architecture parallèle dynamiquement reconfigurable de Transputer*, 11ème Journées Francophones sur l'Informatique, Nancy, 1989.

[MUN90] T. Muntean, Ph. Waille, *l'architecture des machines Supernode,* La Lettre du Transputer n°7, septembre 1990, pp. 11-40.

[NAU87] B.A. Naused, B.K. Gilbert, *A 32-bit, 200 MHz GaAs RISC for high-throughput signal processing environments*, IEEE Micro, décembre 1987.

[NEU66] J. von Neuman, *Theory of self-reproducing automata,* Univ. of Illinois Press, 1966.

[OBJ88] Ph. Objois, *Réseau de cellules intégré : Mécanisme de communication inter-cellulaire et application à la simulation logique,* thèse de doctorat en microélectronique, I.N.P. Grenoble, septembre 1988.

[PAT80] D.A. Patterson, D.R. Ditzel, *The case of the Reduced Instruction Set Computer,* (ACM) Sigarch, vol. 8, n° 6, Oct 1980, pp. 25-33.

[PAY91] E. Payan, *Etude d'une architecture cellulaire programmable : Définition fonctionnelle et méthodologie de programmation,* thèse de doctorat en microélectronique, I.N.P. Grenoble, juin 1991.

[PET85] J.C. Peterson, J.O. Tuazon, D. Liberman, M. Pniel, *The Mark-III hypercube-ensemble concurrent computer*, Proc. 1985 Int. Conf. on Parallel Processing, Aug. 1985.

[PLA87] J.A. Plaice, N. Halbwachs, *LUSTRE V2: User's Guide and Reference Manual*, Laboratoire de Génie Informatique, Grenoble, Octobre 1987.

[PRE81] F.P. Preparata, J. Vuillemin, *The cube-connected cycles: a versatile network for parallel computation*, Comm. of the ACM, octobre 1981, vol. 24, n° 5, mai 1981, pp 300-309.

[SAA88] Y. Saad, M.H. Schultz, *Topological properties of hypercubes,* IEEE Transactions on Computers, vol. 37, n°7, juillet 1990, pp 867-873.

[SCH84] P.U. Schulthess, *A Reduced High-Level-Language Instruction Set,* IEEE Micro, June 1984, pp. 55-67.

[SEI84] C.L. Seitz, *Concurrent VLSI Architectures,* IEEE Transactions on Computers, Vol C-33, n°12, December 1984, pp. 1247-1265.

[SEI85] C.L. Seitz et al., *The hypercube communication chip*, California Institute of Technology, mars 1985.

[SEI85] C.L. Seitz, *The Cosmic Cube*, Commun. ACM, vol. 28, n°1, janvier 1985, pp 22-23.

[SEI88] C. L. Seitz *et al.*, *The architecture and programming of the Ametek Series 2010 Multicomputer*, proc. Third Conf. Hypercube Concurrent Comput. Appl., ACM, January 1988, pp. 33-37.

[SEI90] C.L. Seitz *et al.*,, *Submicron systems architecture,* California Institute of technology, CS-TR-90-05, march 1990.

[SHA89] E. Shapiro, *The family of concurrent logic programming languages,* ACM computing surveys, 21(3), september 1989.

[SIA85] P. Siarry, L. Bergonzi, G. Deyfus, Optimisation du placement de blocs par la méthode thermodynamique : application à la conception du plan de masse d'un circuit, 1er colloque national sur les circuits à la demande, Grenoble 1985.

[SUC89] S. Succi, D. d'Humières, F. Szelenyi, *Lattice-gaz hydrodynamic on the IBM 3090 vector facility*, IBM J. Res. Develop. vol. 33 n°2, mars 1989, pp. 136-148.

[TOF87] T. Toffoli, N. Margolus, *Cellular automata machines: a new environment for modeling* , MIT Press, Cambridge, 1987.

[TRE91] A. Trew, g. Wilson, *Past, Present, Parallel survey of available parallel computing systems*, Sringer-Verlag 1991.

[TUC88] L.W. Tucker, G.G. Robertson, *Architecture and applications of the Connection Machine*, IEEE Computer, vol. 21, n°8, août 1988, pp 26-38.

[WAG83] R.A. Wagner, *The Boolean Vector Machine*, IEEE Proc. 10th Ann. Intl. Symp. on Computer Architecture, pp. 59-66, 1983.

[WAY88] P. Wayner, *Modeling chaos* , BYTE, may 1988.

[YAN86] C-B. Yang, R.C.T. Lee, *The mapping of 2-D array processors to 1-D array processors*, Parallel Computing 3 (1986), pp. 217-229.

[YAN90] C-B. Yang, R.C.T. Lee, W-T. Chen, *Parallel graph algorithm based upon broadcast communications*, IEEE Transactions on Computers, vol. 39, n°12, décembre 1990, pp 1468-1472.

Annexe 1 : Applications

1. Conway1D et 2D

Le "jeu de la vie" est un autre exemple d'automate cellulaire. L'état suivant d'une cellule est fonction de l'état courant (0 ou 1) ainsi que de la somme des états des huit voisins, selon la table de transition :

somme =	0	1	2	3	4	5	6	7	8
état = 0	0	0	0	**1**	0	0	0	0	0
état = 1	0	0	**1**	**1**	0	0	0	0	0

Tableau 1 — *Règles de transition.*

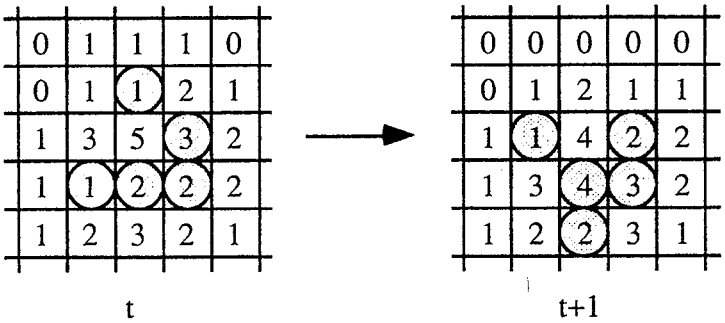


Fig. 1. — *Exemple d'évolution (dans chaque case est inscrit le nombre de voisins).*

Cet automate a été programmé selon la méthode à allocation fixe des sites (CONWAY2D), et selon la méthode à circulation des sites (CONWAY1D) introduite pour le *Lattice Gaz Model* dans le chapitre IV. Cette application ne présente aucun intérêt algorithmique propre, sa présence est due à son utilisation comme programme de test lors du développement des outils de simulation.

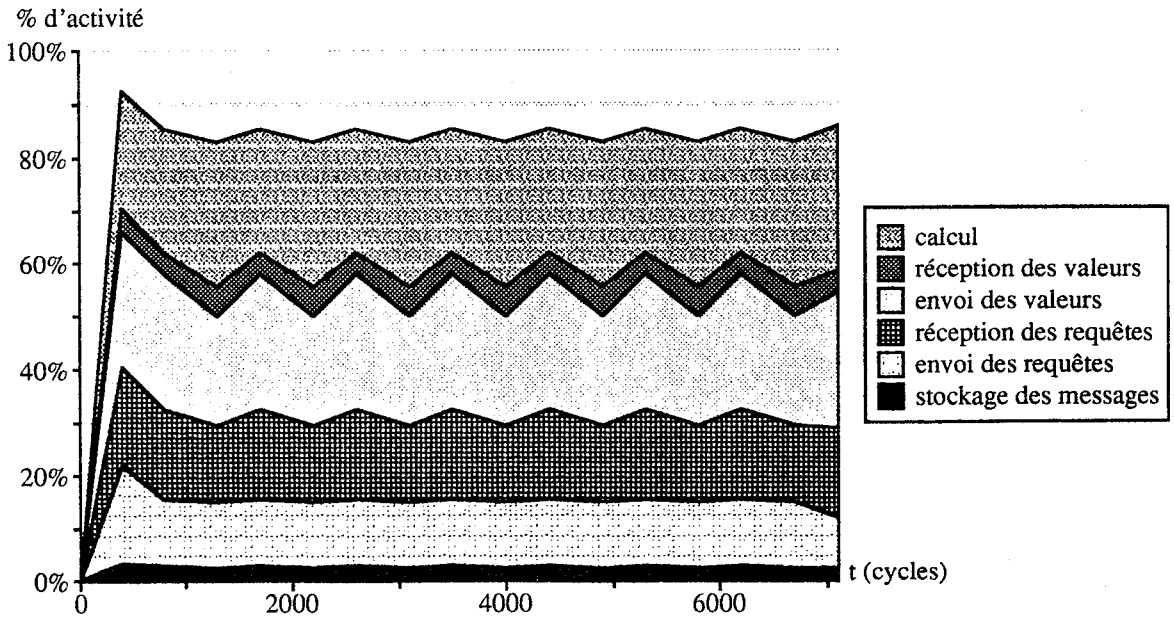


Fig. 2 — *Activité pour Conway2D (allocation fixe).*

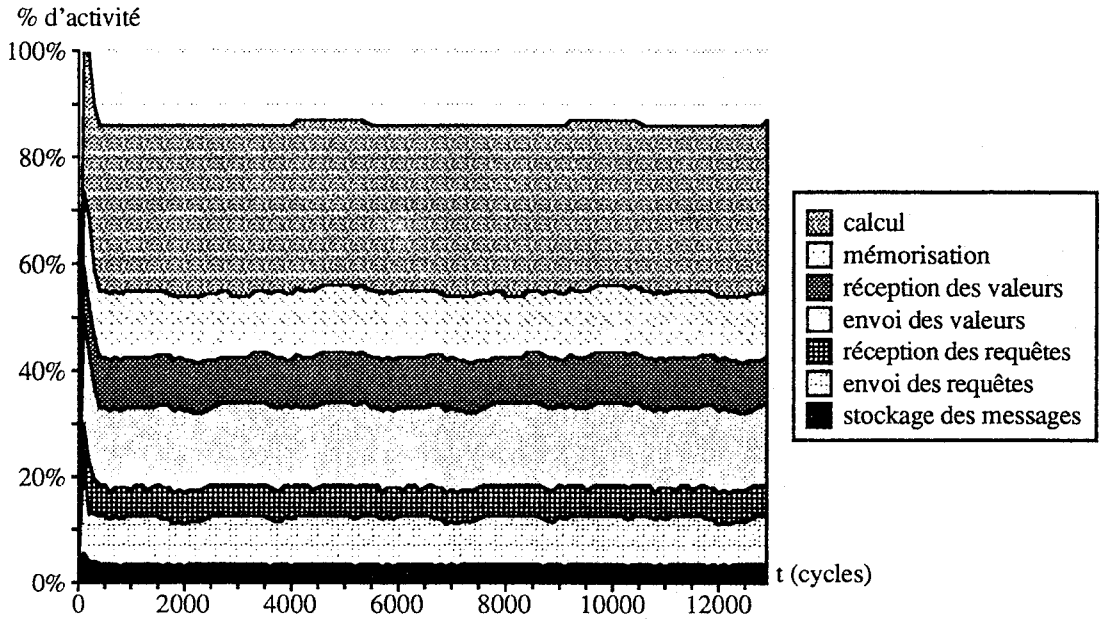


Fig. 3 — *Activité pour Conway1D (circulation des sites).*

2. Distance

Cet exemple d'application est tiré de [LAV89]. Le calcul de distance entre mots consiste à évaluer une *distance d'édition*, somme des coûts des opérations d'éditions nécessaire pour passer d'un mot à l'autre :

- substitution d'une lettre par une autre,
- insertion d'une lettre,
- suppression d'une lettre,
- permutation d'une lettre etc.

Pour passer d'un mot à un autre, il existe généralement plusieurs séquences de coûts cumulés différents. La distance d'édition est le minimum de ces coûts. Cette distance sert, comme son nom l'indique, à mesurer le degré de proximité de deux mots ; elle est particulièrement utile en correction orthographique. L'un des deux mots est le mot à corriger, l'autre appartient à un dictionnaire. Le mot à corriger est successivement comparé à tous les mots du dictionnaire, dont on retiendra les plus proches, qui seront proposés comme correction. On conçoit donc que le calcul de distance est une opération très répétée qui justifie un traitement parallèle.

Si on se limite aux substitutions, insertions et suppressions, la distance D(n,m) entre deux mots x[n] et Y[m] peut se définir par une opération de récurrence :

$$\text{si } i>0 \text{ et } j>0 \qquad D(i,j) = \min \begin{cases} D(i-1,j-1) + d(x[i],y[j]) \\ D(i-1,j) + d(\varepsilon,y[j]) \\ D(i,j-1) + d(x[i],\varepsilon) \end{cases}$$

$$\text{sinon} \qquad D(i,j) = 0$$

d(a,b) est le coût élémentaire d'une opération de substitution de b par a, d'une insertion si a=ε, d'une suppression si b=ε. d(a,b) peut être une fonction complexe, mais nous pouvons simplement adopter

$$d(a,b) = \begin{cases} \text{si } a=b & 0 \\ \text{sinon} & 1 \end{cases}$$

Cette évaluation récursive peut facilement être réalisée grâce à un tableau systolique :

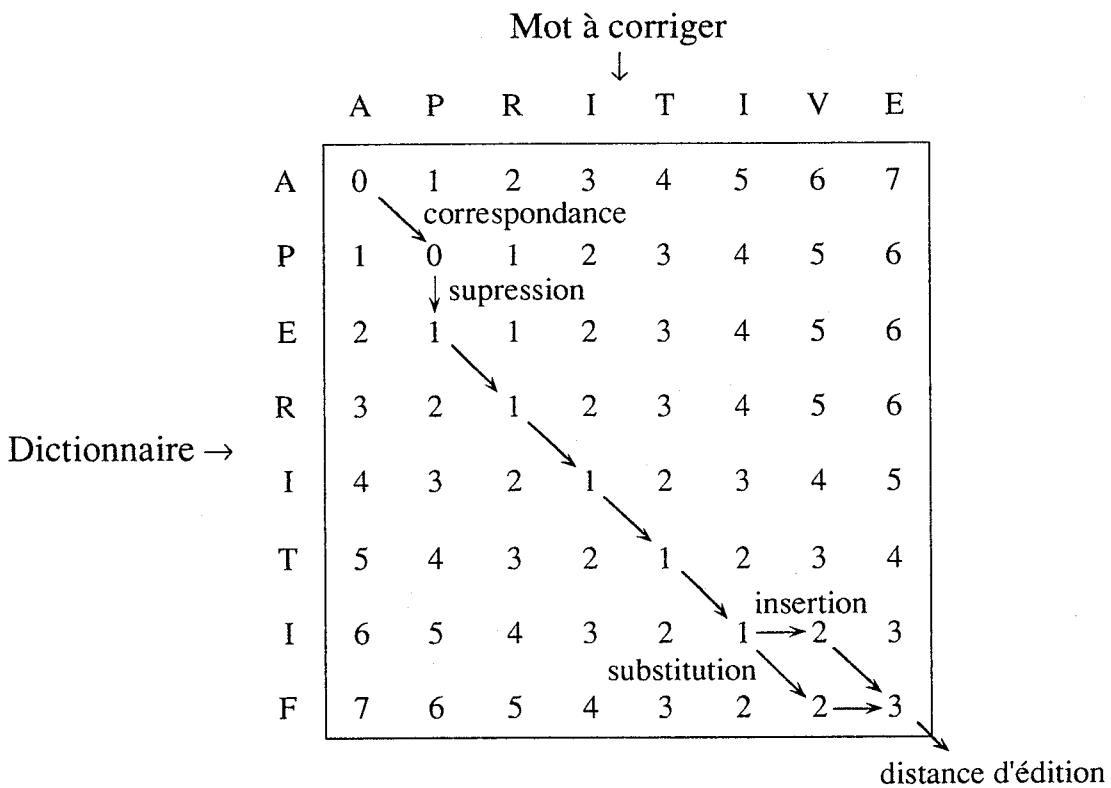


Fig. 4 — *Comparaison de deux mots par un tableau systolique.*

Chaque processeur $P_{i,j}$ est chargé de calculer la distance d'édition D(i,j). $P_{i,j}$ reçoit en entrée une lettre de chacun des mots, ainsi que D(i-1,j), D(i,j-1) et D(i-1,j-1), calcule D(i,j) selon la relation de récurrence, et fournit ce résultat à $P_{i+1,j}$, $P_{i,j+1}$ et $P_{i+1,j+1}$ (fig. 4). En sortie du réseau, au niveau de $P_{n,n}$ nous avons la distance d'édition entre les deux mots injectés n cycles systoliques plus tôt. Les comparaisons peuvent se pipeliner ; typiquement, le mot à corriger sera réinjecté à chaque cycle sur un côté, et le contenu d'un dictionnaire sera injecté sur l'autre coté.

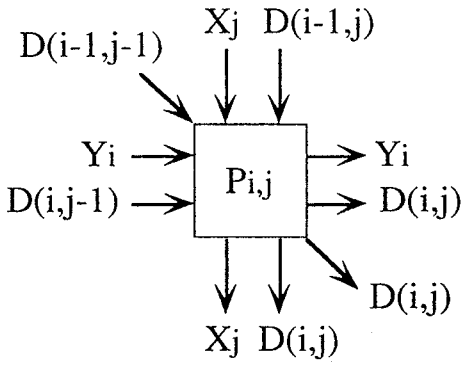


Fig. 5 —*Cellule systolique élémentaire.*

Une implémentation systolique repliée sur une ligne de processeurs (API15C) a été réalisée sur la machine MICMACS [LAV89]. L'implémentation que nous en proposons conserve le tableau bidimensionnel, mais avec des cellules asynchrones et un contrôle de flux de type requête-réponse. Ce dernier point appelle un commentaire. Dans un réseau systolique pur, les contraintes de synchronisation des flux de données et de résultats partiels font que l'introduction d'un registre de retard au niveau de la connexion diagonale est indispensable [LAV89]. Dans notre implémentation, ce point de mémorisation n'est plus nécessaire, car l'asynchronisme induit une souplesse au niveau de la consommation des données. De fait, il apparaît expérimentalement sur la figure 6 que l'asynchronisme permet une relaxation des contraintes, le taux d'activité globale en régime stabilisé étant de l'ordre de 93%. Le taux d'activité en calcul est lui de l'ordre de 30%, ce qui est tout à fait correct compte tenu de la simplicité des opérations à effectuer.

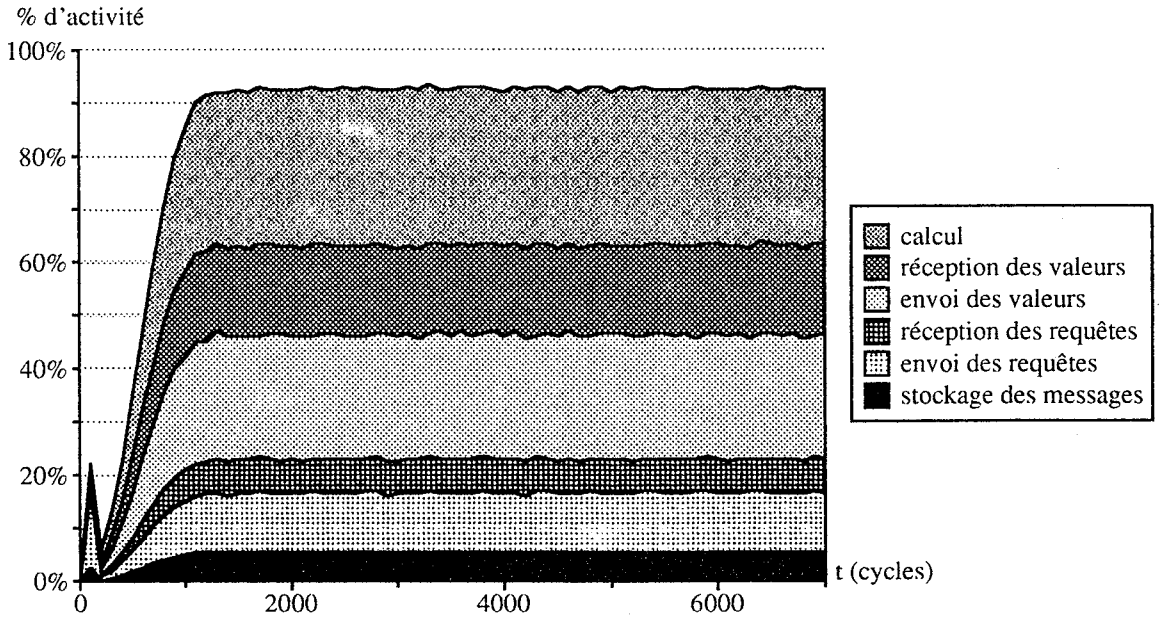


Fig. 6 — *Courbe d'activité pour lle calcul de distance en antre mots.*

Au total, le réseau est capable de fournir en moyenne un résultat tout les 116 cycles, fréquence que nous pouvons comparer avec d'autres architectures :

machine	fréquence	fréquence / nb transistors
MicMacs 18 processeurs	18600 mots/s	0,023 mots / T.s
Réseau cellulaire 18x18	86200 mots/s	0,013 mots / T.s
SUN-3/160	310 mots/s	
IBM-PC/AT	<100 most/s	

Tableau 2. — *Performances de différentes architectures.*

Nous voyons que le réseau cellulaire permet des performances supérieures à la machine MicMacs, ce qui n'est pas surprenant compte tenu de la différence de nombre de transistors. En revanche, les performances ramenées au transistor, sorte de normalisation pour comparer des choses comparables, plaident en défaveur du réseau cellulaire. Ceci est dû au fait que le réseau est dimensionné dans les deux cas en fonction de la taille de mot maximum, entraînant une sous-utilisation d'une partie du réseau, mais avec une influence différente : linéaire pour MicMacs, carré pour le réseau cellulaire.

Le principal inconvénient de cet algorithme de correction orthographique est qu'il ne met en œuvre que la force brute : le dictionnaire entier, ou tout au moins une portion significative, doit être testé, ce qui se traduit par des besoins d'entrées/sorties considérables : pour faire travailler le réseau à pleine vitesse, l'hôte devra fournir 18 caractères et récupérer un résultat tous les 116 cycles, soit plus d'un million et demi de messages par seconde ! Dans ce type de situation, le fonctionnement MIMD peut être mis à profit. L'utilisation de l'algorithme recouvre en réalité trois fonctions :

- mémorisation et accès au dictionnaire
- calcul de la distance d'édition
- tri des distances pour ne garder que les mots de distance moindre

C'est essentiellement la première fonction qui pose problème. Nous sommes dans un cas très proche de celui qui précède (Conway1D), les tenants et les aboutissants sont les mêmes. Soit nous dédions en grand nombre des cellules à la mémorisation (24 mots par cellule, soit 2750 cellules pour 30000 mots de 18 lettres), et nous les faisons fonctionner comme un gros registre à décalage, soit nous attachons des processeurs d'entrées/sortie à des mémoires DRAM ordinaires.

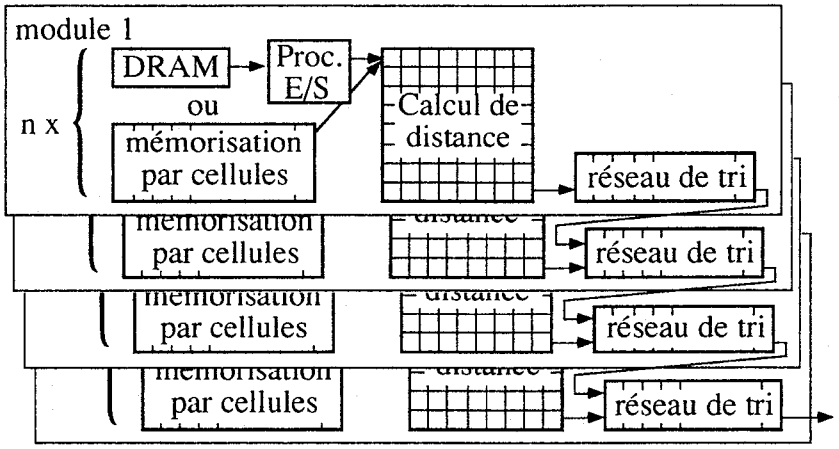


Fig 7 — *Structure multicomparateurs modulaire.*

Une structure modulable en performance peut être construit selon le schéma figure 7. Un système de mémorisation contient le sous-dictionnaire associé au module, dont les mots sont injectés dans le comparateur ; le résultat en sortie est soumis à un sous-réseau de tri linéaire, qui se charge de ne retenir que les k solutions les plus intéressantes. La fin de sous dictionnaire, qui peut être détectée par comptage ou par une marque, déclenche la purge des trieurs de chaque module, chaînées entre eux, pour fournir en sortie une liste triée des résultats dont on pourra ne retenir que la tête. L'opération de fusion des résultats de chaque module est une opération très courte au regard du temps de comparaison, et peut donc être réalisée dans une phase distincte pour le reste de la machine est inactif.

3. Crible d'Eratosthène.

Le crible d'Eratosthène est un algorithme de calcul des nombres premiers. Son principe est très simple : on construit petit à petit une liste des premiers nombres premiers ; chaque entier est successivement déterminé comme étant premier ou non premier en testant sa divisibilité par chacun des éléments de cette liste. Si un entier est déterminé comme premier, il est ajouté en fin de liste, et sert de diviseur potentiel pour les entiers suivants.

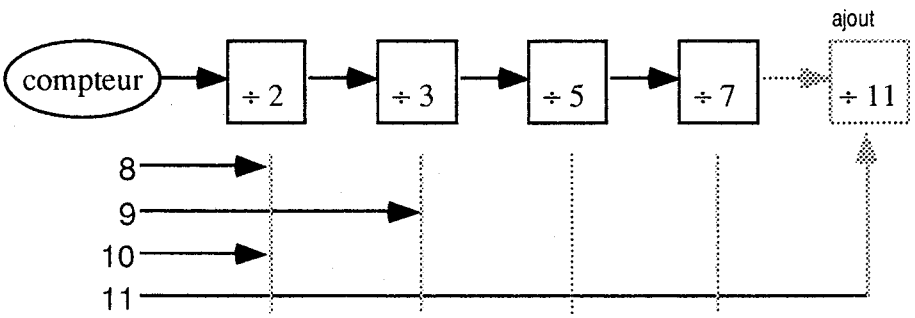


Fig. 8 — *Filtrage des entiers de 8 à 11.*

Cet algorithme s'exprime naturellement sous forme d'un parallélisme de recouvrement. Un processeur est affecté à la génération des entiers, et un processeur est affecté à chaque nombre premier connu. Lorsqu'un processeur reçoit un candidat, il teste sa divisibilité ; si le candidat est divisible, il est oublié, sinon il est passé au processeur suivant. En fin de liste, le candidat est déclaré premier, on lui affecte un processeur, et il est ajouté en fin de liste.

```
VAR  Diviseur,Candidat:valeur
ENTREE Precedent:valeur
     ReqSuivant:requete
SORTIE ReqPrecedent:requete
     Suivant:valeur
ENVOYER(ReqPrecedent)
Diviseur:=RECEVOIR(Precedent)
ENVOYER(ReqPrecedent)
CYCLE
  Candidat:=RECEVOIR(Precedent)
  ENVOYER(ReqPrecedent)
  SI Candidat MOD Diviseur ≠ 0 ALORS
     RECEVOIR(ReqSuivant)
     ENVOYER(Suivant,Candidat)
```

Remarquons que cet algorithme est piètre exemple parallèle : si le compteur et l'étage de division par 2 fonctionnent à plein régime, l'étage de division par 3 est déjà limité à une activité de 1/2 par le crible de la division par 2, et les étages suivants n'ont plus que des activités dérisoires allant sans cesse en décroissant. On peut voir *(fig. 9)* que l'activité a tendance à se stabiliser aux alentours de 20%.

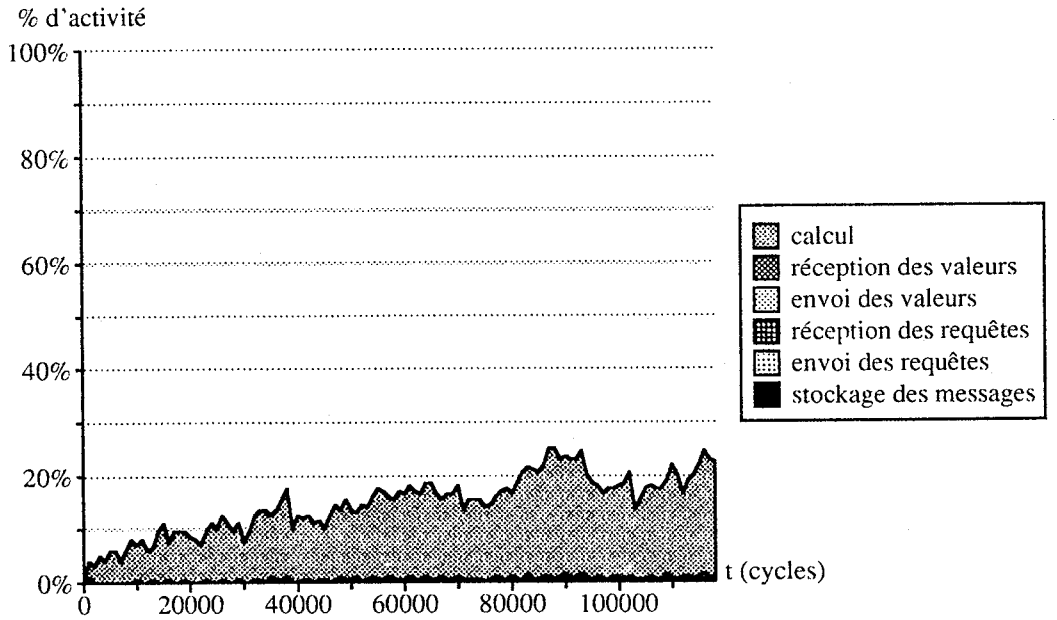


Fig. 9 — *Crible compteur+31 étages (2→27).*

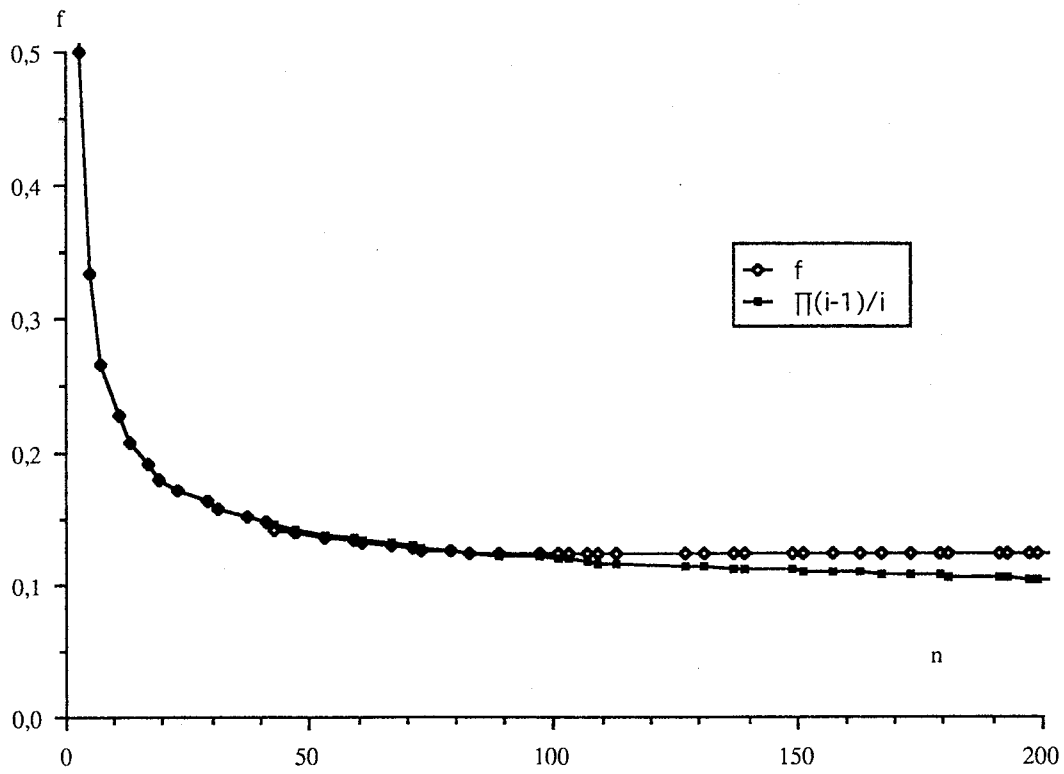


Fig. 10 — *Evolution de la fréquence d'alimentation des étages du crible.*

Les premiers étages du crible font chuter la fréquence d'alimentation de leurs successeurs de façon drastique. La figure 10 rend compte de l'évolution de cette fréquence f, en la comparant à la fréquence de passage "naïve" $\prod_{i \in pred} \frac{i-1}{i}$, produit des proportions que laissent passer les opérateurs précédant un étage. En regard, f décroît plus brusquement, mais de stabiliser plus rapidement aussi à un niveau un peu plus élevé. Pour obtenir une bonne activité, il suffit donc d'avoir un moyen pour alimenter le crible à vitesse maximale au niveau d'un certain étage, par exemple l'étage divisant par 23. On peut faire cela de trois façons :

- substituer au compteur de génération un dispositif plus complexe capable de ne générer que des nombres premiers vis-à-vis des premiers entiers [ATH87],
- dupliquer la première partie du crible autant de fois que nécessaire, et interclasser les flux qui en sortent avant de les injecter au niveau de l'opérateur 23 *(fig 11)*,

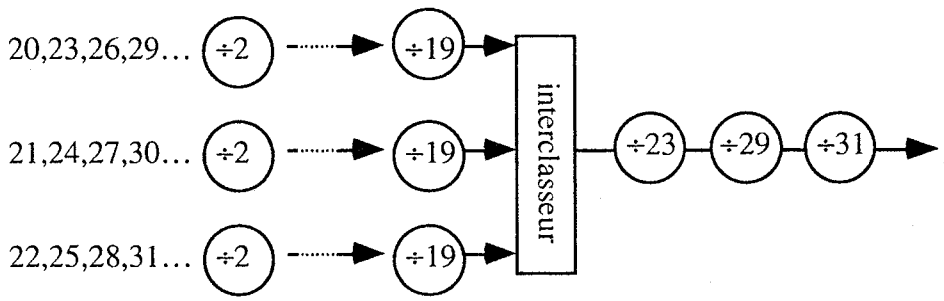


Fig. 11 — *Duplication du début du crible.*

• résoudre le problème en le partitionnant : on programme le réseau en un crible de taille T, qui on lui injecte les entiers de 2 à N ; les K premiers nombres premiers seront reconnus par le crible, et une suite d'entiers premiers par rapport à ces nombres sortira à l'autre bout du crible ; on vide ensuite le crible et on peut recommencer avec la suite résultat autant de fois que nécessaire, c'est à dire (nombre de premiers entre 2 et N) / K fois.

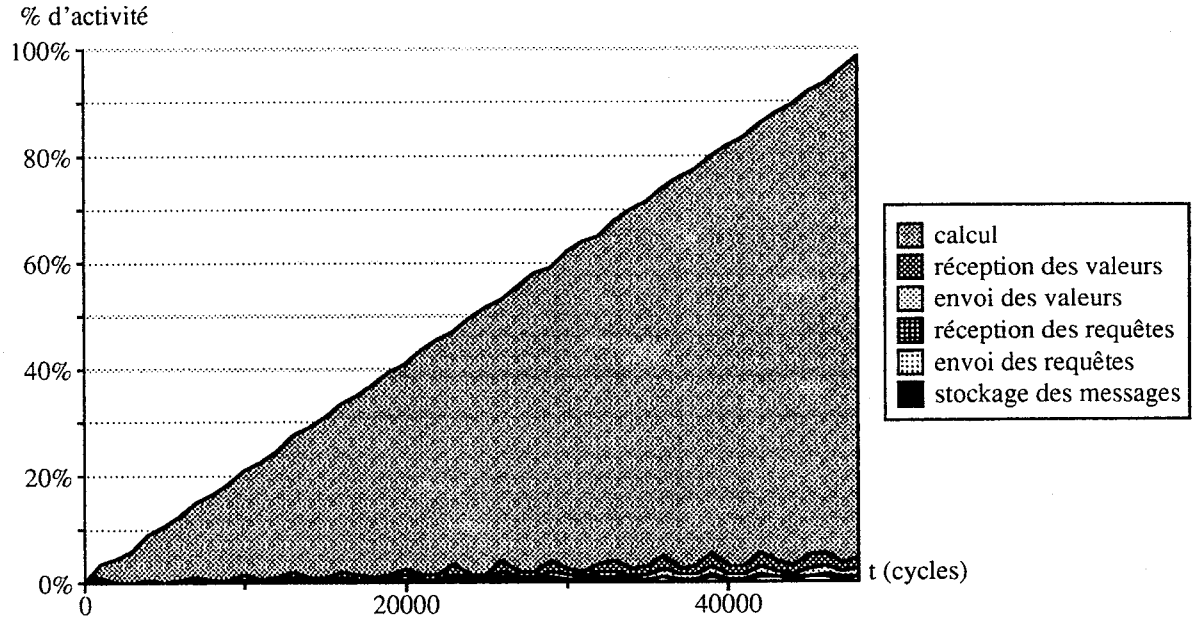


Fig. 12 — *Crible à 32 étages (23 → 173).*

Comme l'atteste la figure 12, le crible devient nettement meilleur même en commençant à l'étage 23. La progression de l'activité est linéaire, traduisant un remplissage à allure régulière, pour arriver à une activité proche de 100%. Dans cet exemple, la première valeur interceptée est 23^2=529, la seconde 23*29=667, ce qui donne une idée de la variation de fréquence d'alimentation dans le crible.

Cette dernière méthode est intéressant car elle permet d'obtenir des nombres premiers sans être limité par la taille du réseau, et avec une activité moyenne de l'ordre de 50% (mise à part pour la première passe où on est de l'ordre de 15%), et ce quelle que soit cette taille. On remarquera aussi que dans tout les cas, l'essentiel de l'activité représente du calcul.

4. Tris avec données en place.

Le tri à bulles de base s'exprime séquentiellement de la façon suivante :

```
PROCEDURE TriABulles(VAR t:tableau[1..nb] DE valeur)
VAR echange:BOOLEEN,i:ENTIER,temp:valeur;
REPETER
   echange:=faux
   POUR i:= 1 A nb-1 FAIRE
      SI t[i]>t[i+1] ALORS
         temp:=t[i]
         t[i]:=t[i+1]
         t[i+1]:=temp
         echange:=vrai
JUSQUA non echange;
```

Le principe de base de parallélisation est simple : on place chaque élément du tableau dans une cellule, puis chaque cellule va essayer d'échanger son élément avec celui d'un de ses deux voisins. Il faut toutefois faire attention au fait que toutes les tentatives se font en parallélisme réel ; pour que le système d'échange de paires soit correct, il faut que chaque cellule ne participe qu'à une seule paire à la fois, ce qui nous conduit à utiliser un algorithme à deux phases. Dans une première phase, on apparie les cellules de rang impair avec leurs suivantes, dans une seconde phase on apparie les cellules de rang pair avec leur suivante.

cellules de rang impair :

```
VAR      ValCour:valeur
         ValSuiv:valeur
ENTREE   DepuisSuiv:valeur
         DepuisPrec:valeur
SORTIE   VersSuiv:valeur
         VersPréc:valeur
CYCLE
    ValSuiv:=RECEVOIR(DepuisSuiv)
    SI ValSuiv < ValCour ALORS
        ENVOYER(VersSuiv,ValCour)
        ValCour:=ValSuiv
    SINON
        ENVOYER(VersSuiv,ValSuiv)
    ENVOYER(VersPrec,ValCour)
    ValCour:=RECEVOIR(DepuisPrec)
```

cellules de rang pair :

```
VAR      ValCour:valeur
         ValSuiv:valeur
ENTREE   DepuisSuiv:valeur
         DepuisPrec:valeur
SORTIE   VersSuiv:valeur
         VersPrec:valeur
CYCLE
    ENVOYER(VersPrec,ValCour)
    ValCour:=RECEVOIR(DepuisPrec)
    ValSuiv:=RECEVOIR(DepuisSuiv)
    SI ValSuiv < ValCour ALORS
        ENVOYER(VersSuiv,ValCour)
        ValCour:=ValSuiv
    SINON
        ENVOYER(VersSuiv,ValSuiv)
```

Le seul point délicat de cet algorithme est la détection de sa terminaison, réalisée en séquentiel en faisant un OU global difficile à réaliser sur un réseau de cellules. On peut appliquer la solution qui consiste à mapper un arbre sur le tableau, dont les feuilles

sont les éléments du tableau. A chaque cycle, chaque nœud du graphe prend pour valeur le OU de ses fils, les feuilles valant 1 s'il y a eu un échange. Dès que la racine vaut 0, le tri est terminé.

Le tri à bulles séquentiel n'est pas un algorithme très intéressant du point de vue de ses performances ; le seul cas où il se montre pas trop mauvais est celui où le tableau est plus ou moins trié, seul les premiers éléments n'étant pas à leur place (ou les derniers si on inverse la boucle interne). En parallèle, même cette situation n'est pas intéressante : plus le tableau sera trié, plus les processeurs vont passer leur temps en tentatives d'échange infructueuses.

Le tri à bulles, de même que sa version parallèle développée plus haut, manipule les données sous la forme d'une liste unidimentionnelle. Comme le réseau cellulaire a été conçus pour être plus proche d'un carré que d'une ligne, nous sommes obligés de replier cette liste suivant une technique quelconque : serpentin, tore hélicoïdal, spirale etc. Ces techniques conservent un lien fort entre voisinage logique et physique (deux voisins logiques se trouvent toujours assignés à des cellules dont la distance de Manhattan est inférieure ou égale à une certaine constante, elle même inférieure ou égale à la portée d'adressage) ; puisque nous sommes en deux dimensions, nous pouvons créer une seconde notion de voisinage logique artificiellement en exploitant le voisinage physique dans la dimension où il n'est pas exploité par le premier voisinage logique. Prenons le cas du tore hélicoïdal :

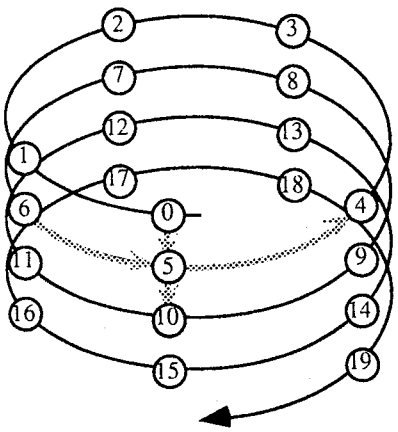


fig. 13

Les échanges normaux du tri à bulles se font selon radialement en suivant le filet; les nouveaux échanges que nous introduisons se font axialement, en sautant un pas sans changer de position angulaire. Sur une hélice de circonférence c, tout élément de rang k E_k est apparié avec E_{k-1}, E_{k+1}, E_{k-c} et E_{k+c}. Le plus long chemin dans une telle structure n'est plus en N, nombre de données à trier, mais en c+N/c, en $2\sqrt{N}$ si on choisi $c=\sqrt{N}$. On peut espérer un gain du même ordre pour le temps d'exécution.

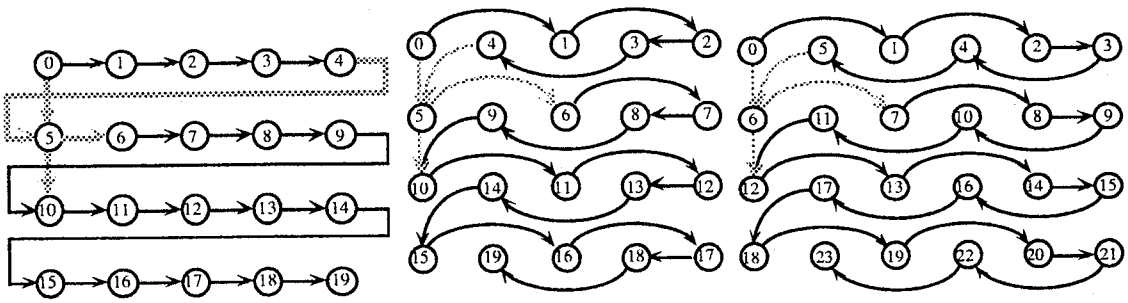


Fig. 14 — a) tore coupé b) nb. impair de colonnes. c) nb. pair de colonnes

Le tore ne doit pas être mappé sur le réseau en reliant simplement les bords droit et gauche du réseau *(fig. 14a)* car si la plupart des distances sont minimum, nous avons

une connection de longueur dépendante de la taille du réseau, et qui se trouvera donc hors de portée d'adressage à partir d'une certaine taille. En mappant le tore non pas en le coupant et en le déroulant, mais en le projetant sur un plan parallèle à son axe, toutes les distances sont de longueur inférieure ou égale à 2 *(fig 14b et c).*

La première surprise concernant cet algorithme de tri est une différence très nette dans la durée d'exécution selon la parité du nombre de colonnes du tore, attestant d'une complexité distincte *(fig. 15, courbes "Tris en hélice non mixés").* Avec un nombre pair de colonnes, l'algorithme se comporte de façon très proche du tri à bulles parallèle ordinaire utilisé comme référence, c'est à dire avec une complexité en N. Par contre, avec un nombre impair de colonnes, la complexité de l'algorithme est visiblement très inférieure, sans toutefois être en $\sqrt{N}$. Les autres courbes se rapportent aux variantes développées dans la suite.

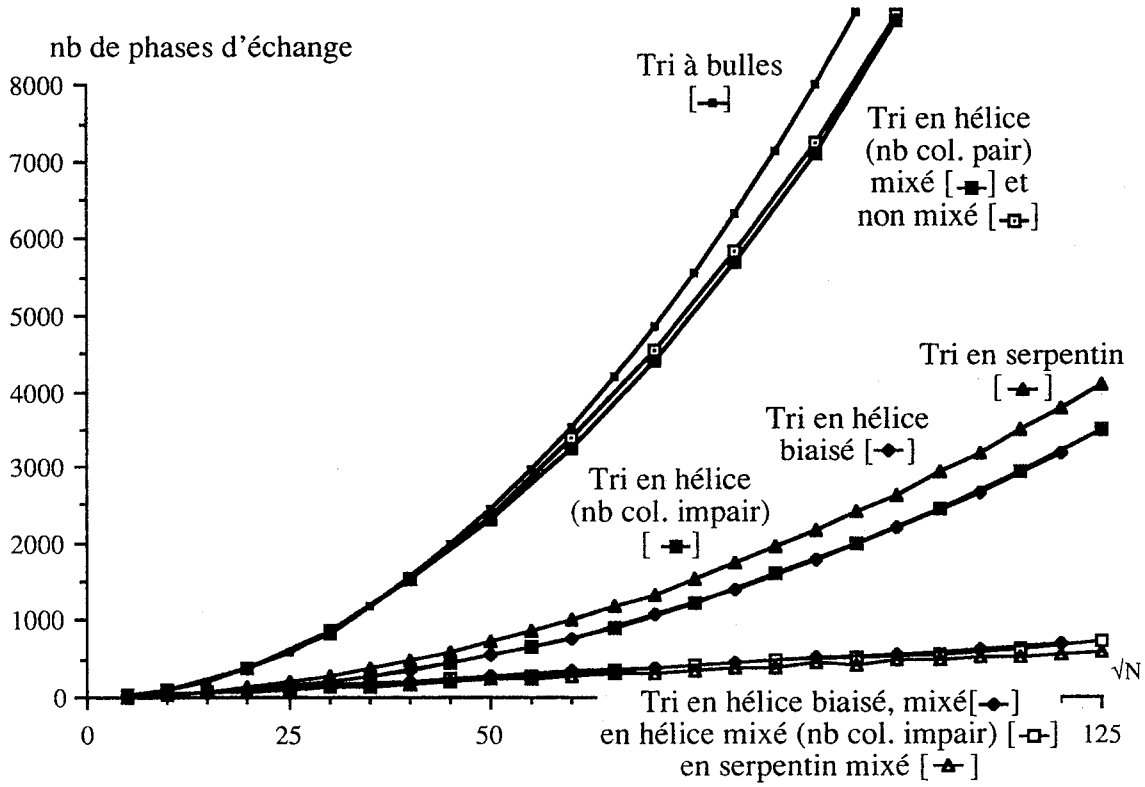


Fig. 15 — *Performances des diverses variantes.*

Comment pouvons-nous nous expliquer cette bizarrerie ? Pour avoir un premier élément de réponse, il faut se référer à l'évolution du ratio Nombre d'échanges/Nombre de comparaisons, verticalement et horizontalement. La figure 16 montre cette évolution pour un tri de 16x16=256 éléments et un tri de 15x17=255 éléments.

Le taux d'échanges verticaux évolue de façon très proche dans les deux cas. Par contre, le taux d'échanges horizontaux est anormalement élevé (allant jusqu'à 100%) et prolongé pour le tri 16x16. Il faut bien avoir à l'esprit qu'un taux d'échanges proche de 100% est aussi peu intéressant qu'un taux nul, car il signifie que l'on effectue des échanges de colonnes, et qu'il n'y a donc pas de mixage entre colonnes susceptibles de donner du travail aux tris verticaux. De fait, l'activité de tri vertical tombe à zéro bien avant la fin de l'exécution.

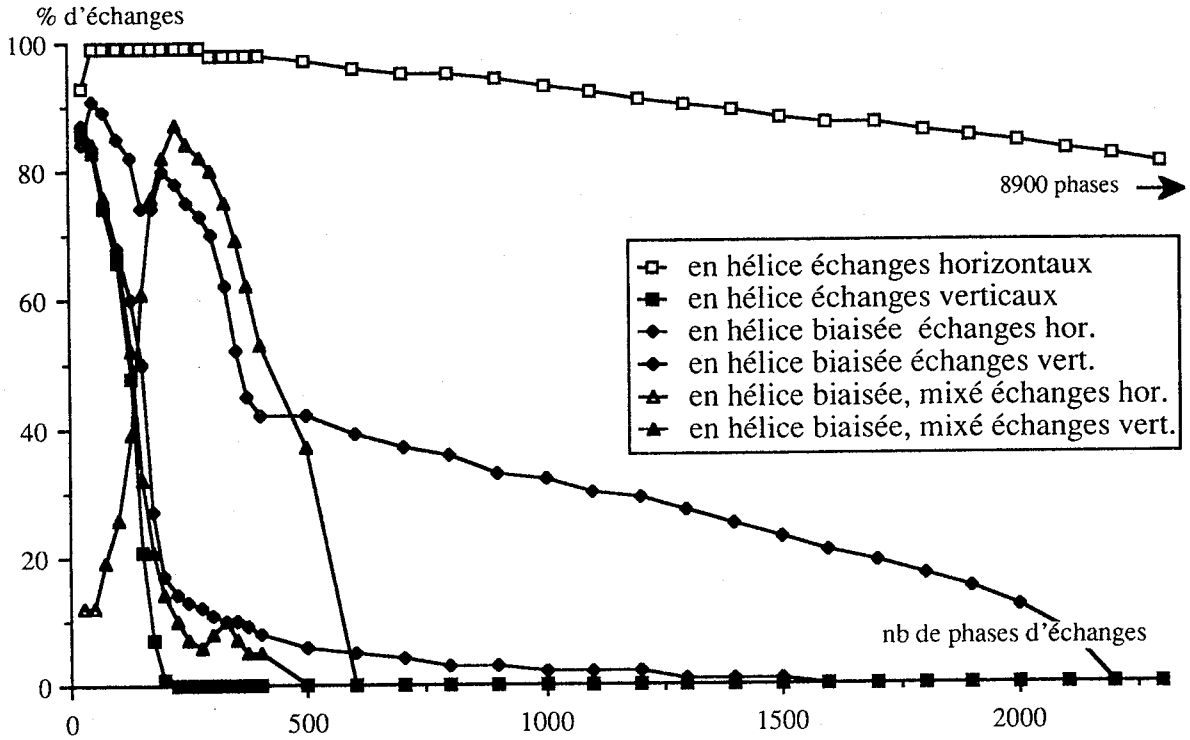


Fig. 16 — *Evolution du taux d'échange lors de l'exécution.*

Mais pourquoi les colonnes sont-elles aussi peu brassées ? Cela s'explique par la façon dont les éléments sont appariés horizontalement d'une ligne à l'autre.

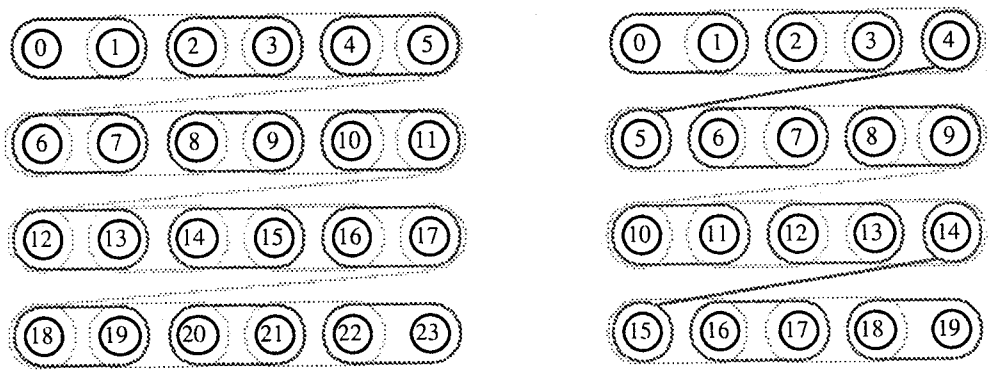


Fig. 17 — *Appariement par un nombre de colonne a) pair b) impair.*

Supposons un tri croissant (0 contiendra l'élément le plus petit). Appelons *élément faible* le plus petit élément dans une paire d'échange, et *élément fort* l'autre. Dans le cas d'un nombre pair de colonnes, les éléments faibles se retrouvent tous sur les colonnes de n° impair, et les éléments forts dans celles de n° pair, et inversement après l'autre passe d'échanges horizontaux. Ainsi, comme les échanges verticaux n'échangent jamais d'éléments de colonnes différentes, il va se produire un partionnement entre colonnes fortes et colonnes faibles. A chaque passe d'échanges horizontaux, chaque colonne forte est comparée et échangée avec la colonne faible qui la suit, et il n'y a rapidement plus de brassage horizontal, seulement un déplacement d'une liste des éléments forts vers le bas, et de celle des éléments faibles vers le haut. Ces listes vont se trier à partir de chaque extrémité de l'hélice, comme un tri à bulle ordinaire. Parallèlement se produit un tri vertical de chaque colonne. Paradoxalement, cette action

verticale va avoir un rôle tout à fait nul : si les éléments les plus forts de colonnes fortes et les plus faibles de colonnes faibles vont se rapprocher plus rapidement de leur place, les éléments les plus petits des colonnes fortes et leur homologues des colonnes faibles vont s'en éloigner, au moins dans un premier temps, car ces éléments, plutôt des valeurs moyennes, vont être repoussés à une extrémité de la colonne.

Ce phénomène de partitionnement ne se produit pas dans le cas d'un nombre impair de colonnes, car l'appariement se fait de telle façon que les éléments forts et les éléments faibles se trouvent répartis uniformément vis-à-vis de la parité du numéro colonne, instituant un brassage continuel qui va permettre au tri vertical de ne pas faire d'échanges éloignant de la solution.

Dans le cas d'un nombre pair de colonnes, on peut l'éviter en effectuant un appariment en biais plutôt que vertical (hélice biaisée) : au lieu d'apparier Ek avec Ek-c et Ek+c, on l'appariera avec Ek-c-1 et Ek+c+1 ou avec Ek-c+1 et Ek+c-1. On peut aussi mapper différemment le tore, comme un serpentin *(fig. 18)*, où l'alternance de sens permet un appariement vertical correct (le sens des paires s'inverse d'une ligne sur l'autre) quel que soit le nombre de colonnes.

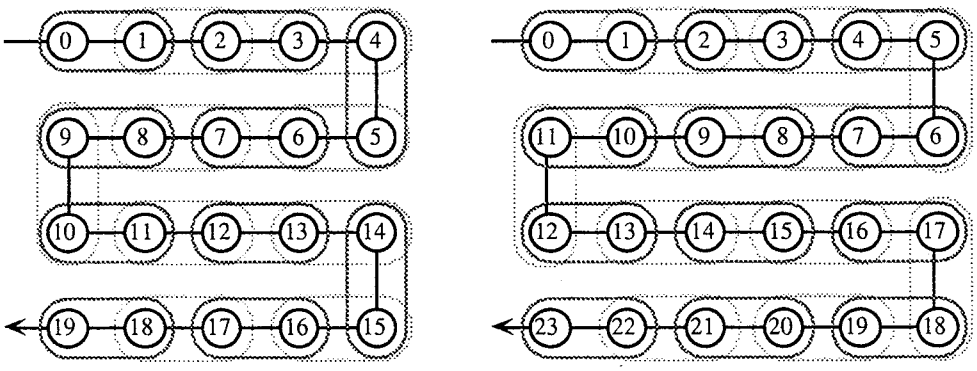


Fig. 18 — *Placement en serpentin avec un nombre de colonnes a) impair b) pair.*

Toutes ces méthodes ramènent le comportement de l'algorithme à au cas vertical, avec nombre impair de colonnes. Malgré ces mesures, on risque quand même d'introduire une composante en N dans le coût d'exécution : les premières versions de tris bidimensionnels testées étaient structurées de la façon suivante :

```
CYCLE
  Appariement avec le précédent horizontal
  Appariement avec le suivant horizontal
  Appariement avec le précédent vertical
  Appariement avec le suivant vertical
```

Malencontreusement, la succession de deux échanges horizontaux nous ramène au cas que nous voulions éviter : le premier appariement horizontal entrelace colonne forte et colonne faible, et le suivant défait cet entrelacement, donnant un entrelacement d'une colonne forte avec une autre colonne forte et d'une colonne faible avec une autre colonne faible. Les algorithmes dits *mixés* alternent strictement appariements horizontaux et verticaux.

Placement	nb de colonnes	mixage	x^2	x	cst
Tri à bulles		non mixé	0,998	-0,63	-5,25
Tri en hélice	pair	non mixé	0,861	3,904	-7,909
	impair	non mixé	0,229	-0,916	21,76
	pair	mixé	0,813	7,076	-74,114
	impair	mixé	0	5,428	-8,467
Tri en hélice biaisée	pair	non mixé	0,227	-0,988	32,045
	pair	mixé	0	5,930	-11,295
Tri en serpent		non mixé	0,243	2,325	-4,361
		mixé	0	4,760	-6,984

Tableau 3 — *Paramètres des courbes de performances.*

Pour conclure, il faut dire que si cet algorithme est intéressant du point de vue de sa complexité, il présente des inconvénients qui peuvent lui faire préférer un autre algorithme, au moins jusqu'à une certaine taille des données :

- il trie les données en place et laisse le résultat en place ; si on n'utilise pas tout un côté du réseau pour les échanges avec l'hôte, les entrées/sorties vont faire perdre le bénéfice de l'algorithme.
- le tri de chaînes de caractères est beaucoup moins naturel qu'avec un tri par fusion ou un tri par insertion.
- il n'y a pas de recouvrement possible entre phases d'échanges, ce qui conduit à une sensibilité élevée à la latence de communication.
- comme pour le tri par insertion, on est limité dans la taille du jeu de données par le nombre de processeurs ; on doit recourir à un tri par fusion pour réaliser l'interclassement des paquets de base.

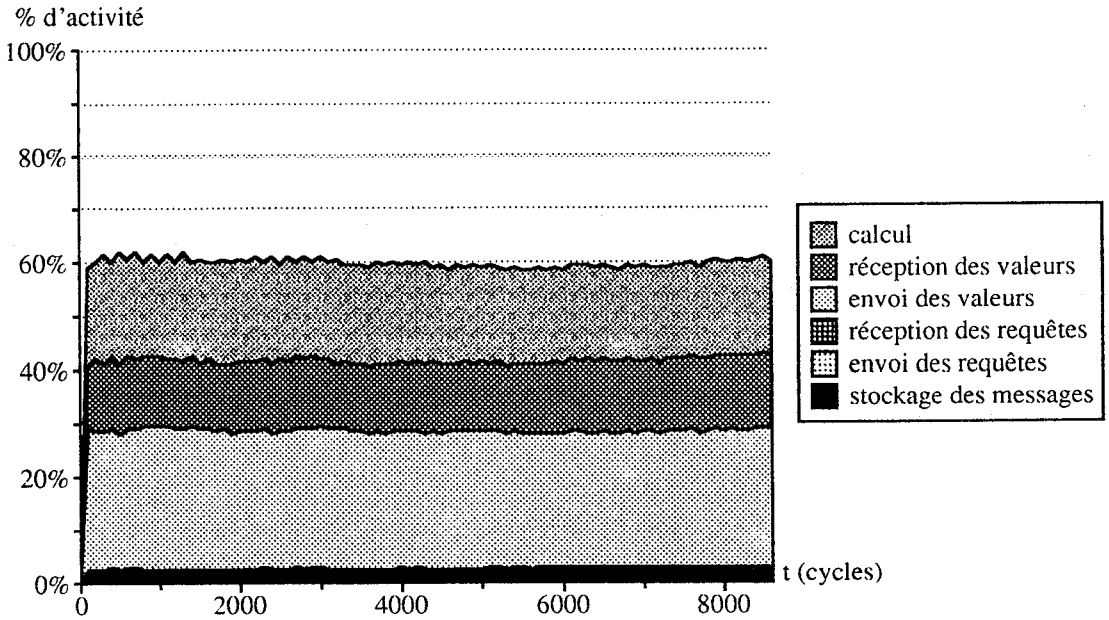


Fig. 19 — *Activité pour le tri à bulle simple*
(pour les tris bidimensionnels, la courbe d'activité est quasi-identique).

5. Multiplication de matrices creuses.

Le produit matrice-matrice est l'une des opérations les plus fréquemment utilisées dans les applications de calcul numérique intensif. A cause de son coût en $O(n^3)$ avec n généralement très grand, de nombreuses techniques matérielles ont été inventées pour l'accélérer : les supercalculateurs de type CRAY utilisent des opérateurs arithmétiques pipeliné et une structure de mémoire adaptée, des solutions très parallèles sont aussi possibles avec des architectures systoliques. Nous pouvons aussi réduire le nombre des opérations effectuées, par des moyens purement logiciels, lorsque les matrices sont assez creuses. Dans cette technique, les éléments nuls des matrices arguments ne sont pas mémorisés, et les opérations effectivement calculées ne mettent en jeu que des valeurs non nulles. Ce gain, potentiellement considérable, a tendance à être annulé par une algorithmique beaucoup plus complexe que celle de l'algorithme naïf.

Malheureusement, ces techniques matérielles et logicielles sont difficilement combinables, car la suppression des opérations inutiles génère des "trous" dans les pipe-lines et dans les flux systoliques. Cette incompatibilité peut s'interpréter de la façon suivante : les accélérateurs matériels sont des machines de type "multiplier et additionner", alors que les algorithmes pour matrices creuses résultent en un grand nombre de comparaisons de rang pour apparier les données.

Notre idée est de paralléliser l'opération d'appariement qui relevait plutôt du code de contrôle. En partant de l'algorithme systolique classique *(fig. 20)* nous construisons une machine "comparer, et éventuellement calculer".

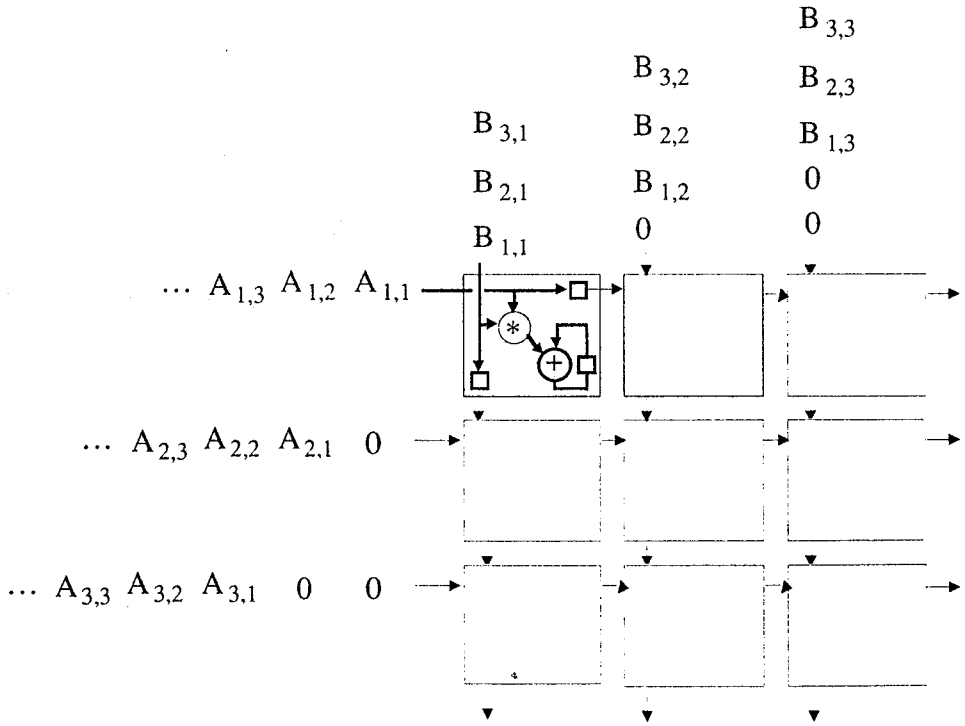


Fig. 20 — *Réseau systolique de multiplication matrice-matrice.*

Naturellement, si nous ne faisons pas circuler les valeurs nulles dans le réseau, le rang de chaque élément circulant n'est plus connu implicitement par sa position dans l'espace et le temps, et nous devons l'identifier explicitement par la donnée de son rang. Nous associons donc à chaque élément de la matrice A son numéro de colonne, à chaque élément de B son numéro de ligne, l'autre coordonnée étant implicitement donnée par la position de l'élément. L'appariement des éléments de A et de B n'est plus apporté par la synchronisation, mais par comparaison de rang. Chaque cellule exécute un algorithme similaire à un interclassement de deux listes triées : une cellule qui a deux éléments de rang différent ne garde que celui de rang le plus élevé et demande la valeur suivante de l'autre flux.

```
cellule_multiplieur (Ain,Bin:flots → Aout,Bout:stream)
a:=get(Ain)
b:=get(Bin)
put(Aout,a)
put(Bout,b)
s:=0
cycle
 if a.rang = b.rang :   s:=s+a.valeur*b.valeur
                        a:=get(Ain) b:=get(Bin)
                        put(Aout,a) put(Bout,b)
    a.rang < b.rang :   a:=get(Ain) put(Aout,a)
    a.rang > b.rang :   b:=get(Bin) put(Bout,b)
```

Cet algorithme montre le gain de parallélisme que permet un schéma de communication asynchrone : comme la multiplication-addition est une opération beaucoup plus longue que l'acheminement, la propagation immédiate des éléments permet à plusieurs cellules de traiter simultanément la même donnée *(fig. 21)*.

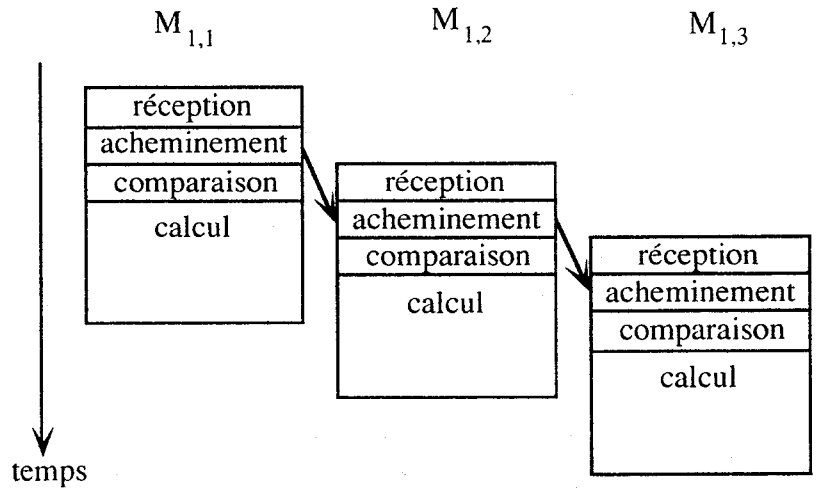


Fig. 21. — *Génération du parallélisme par propagation immédiate.*

Bien entendu, nous ne gagnons rien de cette façon avec des matrices pleines (sauf peut être en temps de remplissage du pipe), car dans ce cas, les cellules en aval sont elle aussi occupées et ne peuvent donc initier un nouveau calcul. Par contre, avec des matrices creuses, la propagation immédiate va en quelque sorte "boucher les trous" dans les flots de données et donc alimenter plus régulièrement les cellules.

Considérant le fait que l'algorithme présenté plus haut ne rentrera pas dans une seule cellule physique, nous sommes obligés de le répartir sur plusieurs, la cellule systolique initiale devenant une "macro-cellule" physique. Plusieurs découpages sont possibles, qui doivent exploiter au mieux les opportunités de parallélisme interne à la macro-cellule tout en respectant les contraintes de taille. La bonne partition dépend donc d'un ajustage fin, qui prennent en compte les caractéristiques du processeur mais aussi de la densité des matrices. Celle qui est implémentée actuellement est la suivante :

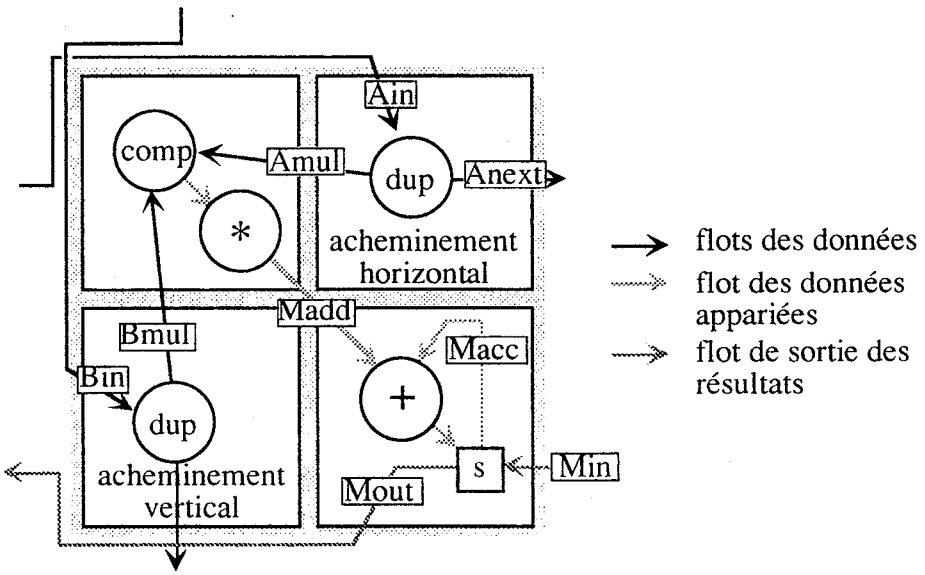


Fig. 22 — *Structure d'une macro-cellule.*

Les opérateurs d'addition et de multiplication au format IEEE 754 32 bits sont très encombrants, à tel point qu'on ne peut pas mettre grand chose à côté. C'est pourquoi les fonctions de propagation ont été extraites et sont traitées par deux cellules dédiées, une dans chaque direction. Ce découpage permet un acheminement très rapide des éléments, d'autant plus que, comme l'algorithme ci-après le spécifie, l'alimentation du canal de propagation est prioritaire sur l'alimentation du canal vers le comparateur-multiplieur. Ce dernier canal n'est alimenté en premier que dans le cas où le canal de propagation est plein. Le comparateur construit un flot vers le multiplieur à partir des couples d'éléments qu'il arrive à apparier. Le multiplieur envoie ses résultats vers l'additionneur qui accumule la somme partielle au fur et mesure, jusqu'à ce qu'une marque de fin de flot apparaisse.

```
acheminement_horizontal(Ain:flot → Anext,Amul:flots)
cycle
  a:=get(Ain)
  if free(Anext)
     put(Anext,a)
     put(Amul,a)
  else
     put(Amul,a)
     put(Anext,a)

acheminement_vertical (Bin:flot → Bnext,Bmul:flots)
cycle
  b:=get(Bin)
  if free(Bnext)
     put(Bnext,b)
     put(Bmul,b)
  else
     put(Bmul,b)
     put(Bnext,b)

comparer_et_multiplier (Amul,Bmul:flots → Madd:flot)
a:=get(Amul)
b:=get(Bmul)
cycle
  if a.rank = b.rang : put(Madd,a*b)
                       a:=get(Amul)
                       b:=get(Bmul)
     a.rank < b.rang : a:=get(Amul)
     a.rank > b.rang : b:=get(Bmul)

additionner (Madd,Min:flots → Mout:flot)
s:=0
cycle
  m:=get(Madd)
  if m.rank ≠ marque_de_fin
       s:=s+m
  else
       repeat
          if s=0 s:=get(Min)
          put(Mout,s)
       until s.rank = marque_de_fin
       s:=0
```

La marque qui déclenche la sortie des résultats et la remise à zéro des sommes partielles est produite de la manière suivante :

- La fin de chaque flux de donnée est codée par un élément fictif de date supérieure à toutes les autres.
- Par convention, les marques des flux horizontaux portent la valeur 1 et les marques des flots verticaux le numéro de colonne.

Arrivées dans le comparateur les marques sont naturellement appariées et multipliées, puis transmises à l'additionneur qui va tester le rang. Ce rang étant une valeur particulière, le numéro de colonne va être extrait pour identifier la position de la somme dans la matrice résultat. Les sommes d'une même ligne sont alors chaînées pour former un flot inverse du flot A, récupéré par l'hôte. Ce flot est lui aussi un flot "creux", les valeurs nulles étant retirées, de sorte que l'hôte ne manipule jamais de matrices pleines.

Concurremment, une nouvelle multiplication peut être amorcée. Pour la multiplication de grosses matrices, la technique classique du partitionnement peut être mise en œuvre : une multiplication de matrices 32 x 32 sera décomposée sur un réseau de 8 x 8 macro-cellules en 16 produits partiels (32 x 8) * (8 x 32). C'est cet exemple que nous avons testé ; la figure 23 montre que l'algorithme est effectivement capable de tirer parti de la densité des matrices : avec une densité de 1/32, le temps de calcul est réduit d'un facteur 12 par rapport à un algorithme naïf.

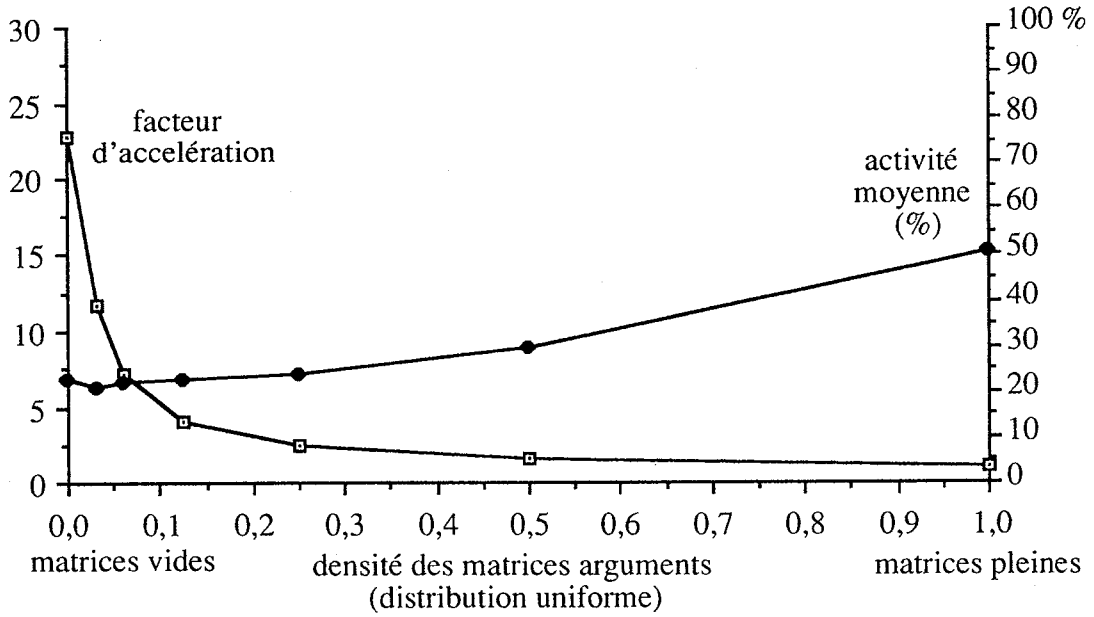


Fig. 23 — *Facteur d'accélération et activité moy. en fonction de la densité des matrices.*

L'analyse du programme de multiplication de matrices creuses est assez délicate, en raison de sa sensibilité aux données et de sa structure temporelle, qui comporte quatre niveaux de recouvrement. Le niveau de recouvrement le plus fin se trouve à l'intérieur de chaque processeur. Il s'agit du parallélisme de recouvrement que nous retrouvons dans (presque) toutes les applications. Le second niveau de pipeline se trouve à l'intérieur du motif de parallélisme : deux cellules sont chargées de faire transiter les données, une autre est chargée de les mettre en correspondance et de les multiplier, et une dernière est chargée de cumuler les produits et de communiquer le résultat à l'hôte. Cet ensemble fonctionne comme un pipeline à 3 étage. Le troisième niveau de pipeline réside dans la structure même de l'algorithme, qui consiste à faire circuler les éléments des matrices de manière à les faire entrer en correspondance. Les éléments sont injectés successivement, et plusieurs éléments circulent en même temps

dans le réseau. Ce pipeline est sans doute le plus important, car c'est lui qui comporte le plus grand nombre d'étage. Le dernier niveau de recouvrement est celui qui existe entre sortie du résultat d'une multiplication et entrée des données de la multiplication suivante. Ce recouvrement est de faible amplitude.

La seconde difficulté qui se pose à nous est la sensibilité de l'application aux données : avec des matrices très creuses, seules les cellules de transfert vont fonctionner à plein régime ; avec des matrices pleines, ce sont les couples multiplieur/additionneur qui vont imposer la cadence. Donc, les chemins critiques vont fluctuer en selon la densité des matrices, ce qui peut avoir une influence sur la latence critique.

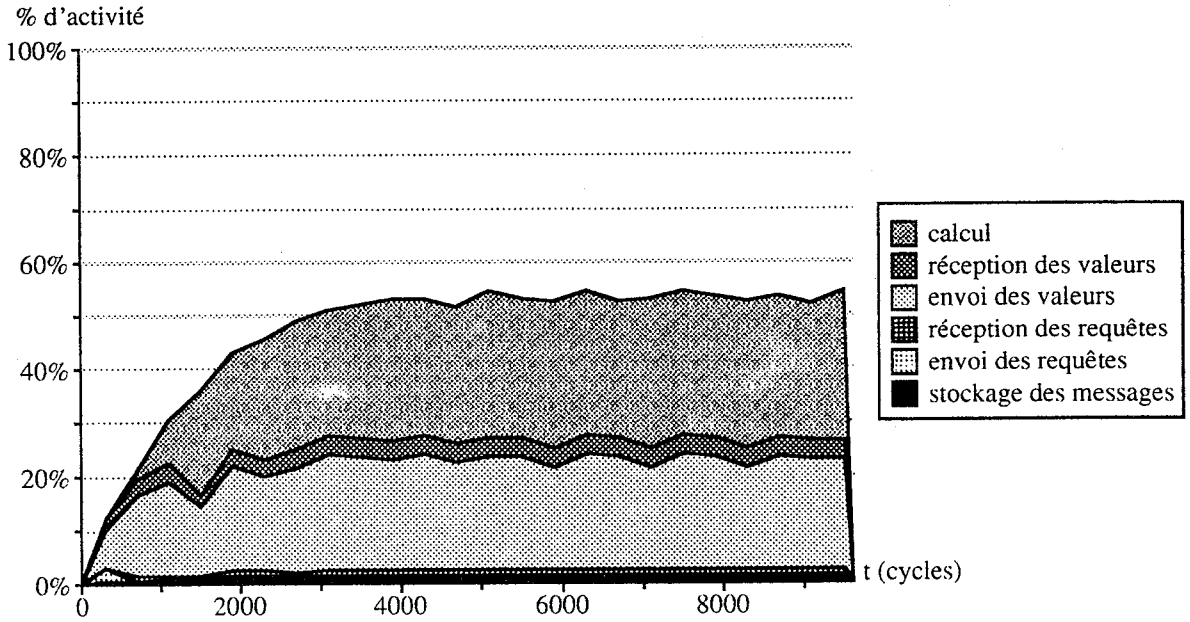


Fig. 24 — *Activité pour la multiplication de matrices (matrices pleines d=1).*

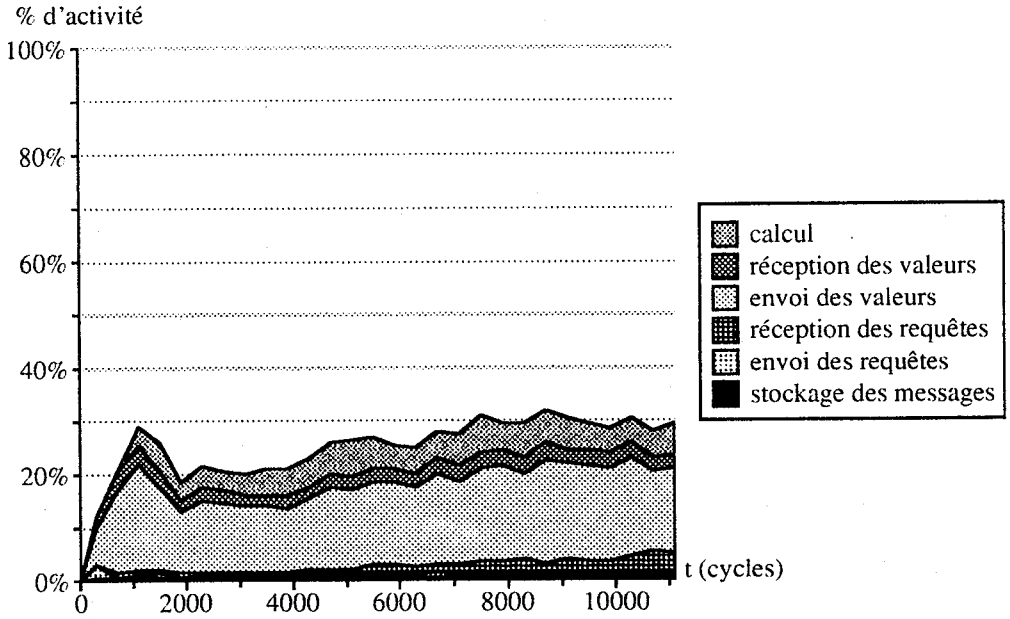


Fig. 25 — *Activité pour la multiplication de matrices (matrices creuses d=0.25).*

6. Simulation logique à échéancier.

La taille des circuits logiques intégrables sur une puce atteint maintenant la centaine de milliers de portes, ce qui rend très coûteuse en temps leur validation par simulation. De nombreux accélérateurs matériels ont été construits pour que la simulation des circuits actuels et futurs reste dans le domaine du possible.

La simulation logique d'une porte doit prendre en compte les aspects temporels liés aux phénomènes électriques sous-jacents (délais de changement d'état), ce qui la rend notablement plus complexe que la simple évaluation de la fonction calculée par la porte. Pour limiter le nombre d'évaluation, on utilise maintenant de façon systématique un algorithme de simulation *à échéancier*. La porte n'est pas évaluée à intervalles réguliers, mais à chaque changement d'état d'une des équipotentielles correspondant aux entrées de la porte. De tels changements sont appelés *événements* ; c'est en nombre d'événements traités par seconde que l'on mesure les performances des algorithmes et des machines. Les modèles de simulation à événements discrets n'intéressent pas seulement la CAO des VLSI, ils peuvent être appliqués à l'étude de toutes sortes de phénomènes évolutifs.

Le processus de modélisation d'une cellule peut se formuler en première approche de la façon suivante :

```
type evenement = (d:date, v:valeur)
E : tableau[1..entrance] de canaux d'evenements
S : tableau[1..sortance] de canaux d'evenements
Dern : tableau[1..sortance] d'evenements
Tsim : date // temps de simulation courant
Etat : tableau[1..entrance] de valeur
Resultat : valeur

Tsim=0
Resultat=indetermine
CYCLE
  POUR i=1 A entrance
     Envoyer(S[i],(Tsim,Resultat))
  PAR POUR i=1 A entrance
     SI (Dern[i].d = TSim)
        Dern[i]:=Recevoir(E[i])
  Tsim = min(Dern.d)
  POUR i = 1 A entrance
     SI Dern[i].d = Tsim
        Etat[i]:=Dern[i].v
  R = f(Etat)
FINCYCLE
```

Une telle décomposition basée sur un parallélisme de porte se prête très bien à une implémentation sur réseau cellulaire : processus de tailles relativement régulières, interconnectés statiquement mais irrégulièrement, des flux de données irréguliers, des données de petite taille, et un calcul simple. Cette application a été la première envisagée pour notre réseau, et elle a donné lieu à la réalisation d'un circuit [OBJ88].

L'algorithme présenté ci-dessus ne marche correctement que pour les graphes de portes acycliques, la présence d'un cycle induit un risque d'interblocage. Plusieurs méthodes existent qui évite les interblocages par prévention ou guérison [ING90]. Une version impliquant l'envoi de message NULL (qui font avancer artificiellement la simulation d'une porte) a été étudiée et programmée pour le réseau programmable [JAC91].

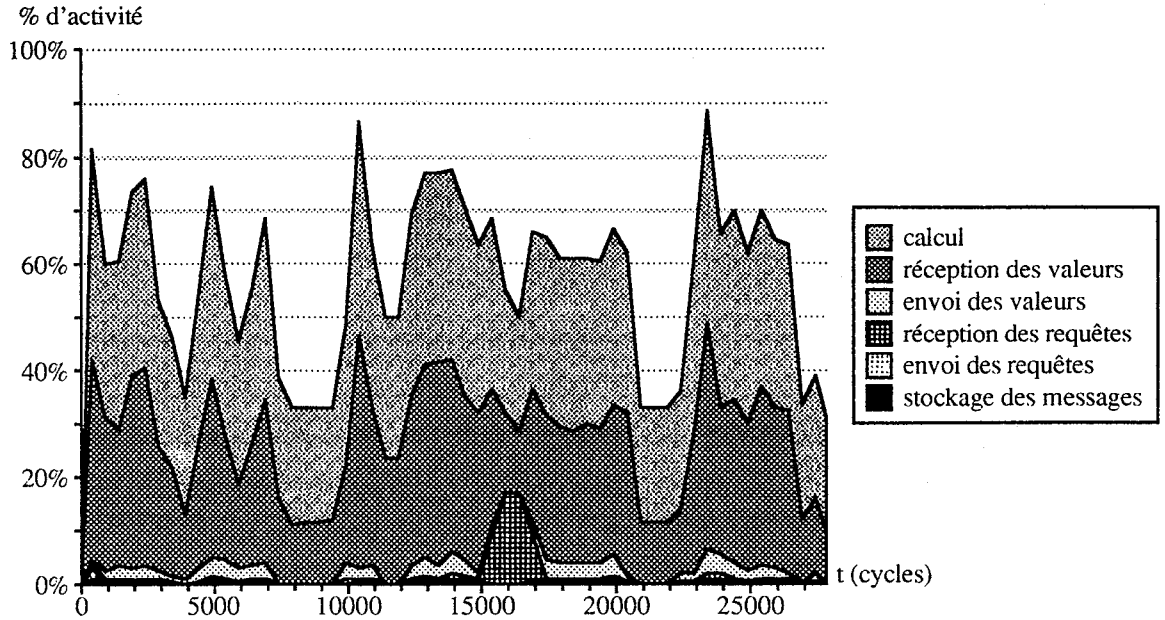


Fig. 26 — *Répartition de l'activité pour la simulation logique à échéancier.*

7. Problème des huit reines.

Le problème des huit reines est un problème d'école classique de recherche de solution. Il consiste à trouver toutes les manières possibles de placer 8 reines sur un échiquier sans qu'elles ne se menacent les unes les autres. L'algorithme habituel pour le résoudre pose une reine, puis une seconde qui ne soit pas menacée par la première, et ainsi de suite. On procède colonne par colonne (il n'y a qu'une seule pièce par colonne et par ligne). La recherche est donc une exploration exhaustive d'arbre.

Paralléliser ce type de parcours est simple en théorie, mais bien plus difficile en pratique. Lorsqu'on arrive à un nœud de l'arbre, on crée autant de processus qu'il y a de branches attenantes à ce nœuds. Concrètement, il s'agit d'affecter les processus créés à des processeurs et de les exécuter dans un certain ordre, de manière à :

- équilibrer la charge de travail entre les processeurs
- contrôler le nombre de processus en attente (qu'il faut mémoriser), en jouant sur la proportion entre parcours en largeur (fortement générateur) et en profondeur (faiblement générateur), afin à la fois d'assurer du travail pour tous les processeurs et d'éviter une explosion combinatoire.

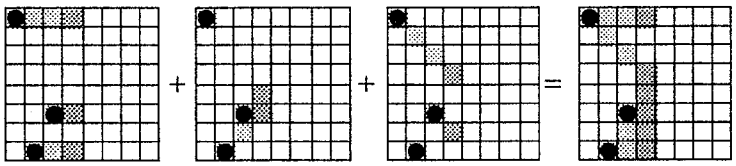
Dans le cas général d'exploration d'arbre, c'est une tâche très difficile que nous n'ambitionnons pas de traiter ici. Néanmoins, sur un problème simple comme celui des

8 reines, l'adaptation au réseau cellulaire est assez facile, comme nous allons le voir.

Commençons par examiner comment décrire une solution incomplète, c'est à dire un processus en attente. Il faut connaître le nombre NR de reines déjà posées, leur position Position[NR]. Pour poser une nouvelle pièce, il faut comparer sa position prévue avec celles des précédentes, pour déterminer si elle est menacée ou non, horizontalement et selon les deux diagonales.

```
menacée(lig,col) :
  pour j = 1 à col-1
     si lig = Position[j] alors repondre vrai
     si lig-Position[j] = j-col alors répondre vrai
     si lig-Position[j] = col-j alors répondre vrai
  repondre faux
```

Il est toutefois plus simple de construire au fur et à mesure les projections des reines déjà posées, horizontalement et diagonalement, sur la nouvelle colonne, ce qui nous donnera les positions non menacées directement :



Horizontale Diagonale 45° Diagonale -45° Positions menacées

Fig. 27 — *Calcul des positions menacées par somme de projections.*

La somme logique des trois projections donne le vecteur des positions menacées. Ces projections sont faciles à manipuler, puisqu'elles peuvent être codées sous la forme d'un octet. A chaque colonne supplémentaire, les projections diagonales sont réactualisées par décalage dans un sens ou dans l'autre.

```
type vecteur = tableau[1..8] de booléen
processus recherche  (  NR:entier,
  Pos:tableau[1..8] de vecteurs
  hor,diag1,diag2:vecteurs ) :
  var menace,reine:vecteurs
  diag1:=dec_gauche(diag1)
  diag2:=dec_droite(diag2)
  NR:=NR+1
  menace:=hor ou diag1 ou diag2
  reine:=[00000001]
  repeter
     si menace et reine alors
        Pos[NR]:=reine
        recherche(NR,Pos,hor ou reine,
                  diag1 ou reine,diag2 ou reine)
     reine:=dec_gauche(reine)
  jusquà reine=0
```

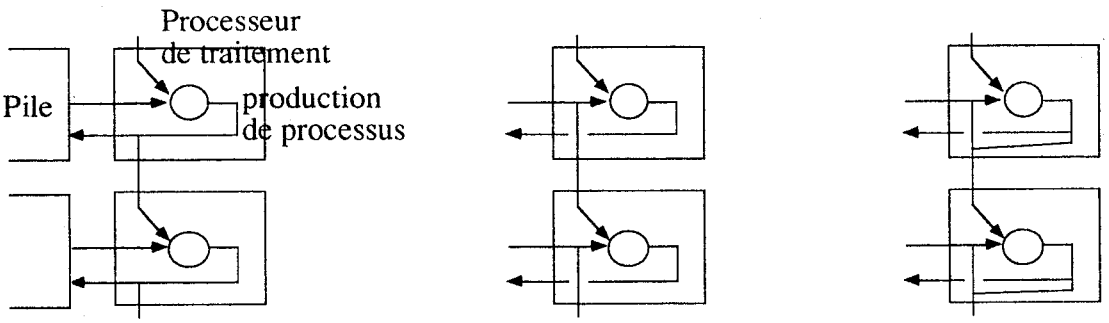
a) par structure globale *b) par emprunt local*

Fig. 28 — *Méthodes de répartition de la charge.*

Le travail d'équilibrage de la charge peut être envisagé de deux façons *(fig. 28)* : a) au niveau de la structure de mémorisation des processus (solutions partielles) en attente, avec un tri selon le degré d'avancement, de manière à favoriser un parcours en profondeur, b) par un mécanisme d'emprunt local, où un processeur devenu inactif demande un processus à traiter à un voisin. La première solution est difficile à implémenter notamment en raison du nombre de voies que doivent gérer de façon asynchrone les cellules d'interface de la structure de mémorisation (liens d'équilibrage, liens d'échange avec le processeur de traitement). La seconde solution est plus simple à mettre en œuvre, et s'avère suffisante.

La stratégie de service du lien d'équilibrage n'est pas indifférente : si on adopte le choix de ne fournir le voisin à cours de travail qu'avec des solutions partielles *produites* par le traitement *(emprunt en aval, fig. 29a)*, le processeur 1 qui initie la recherche va traiter des processus de profondeur 0 et plus, le processeur 2 va traiter des solutions de profondeur 1 et plus, et ainsi de suite jusqu'au processeur 7, les suivants n'étant jamais alimentés, et le l'ensemble étant sujet à une très mauvaise répartition.

Il est préférable d'effectuer l'emprunt en *amont* du traitement, de façon à diffuser le plus possible des solutions de profondeur faible *(fig. 29b)*. Toutefois, dans les cas où tous les processeurs sont inactifs (au début par exemple), il faut se garder de faire tourner indéfiniment le processus initiateur le long des liens d'équilibrage : une cellule de traitement doit être désignée qui ne permettent pas un emprunt amont. Une solution mixte, permettant à la fois des emprunts en amont et en aval permet de servir le lien d'équilibrage le plus rapidement possible, c'est celle que nous avons implémentée.



a) emprunt en aval *b) emprunt en amont* *c) mixte*

Fig. 29 — *Topologie des voies d'équilibrage par emprunt.*

L'algorithme déroulé par les cellules de traitement contient une optimisation qui permet de garder la dernière solution partielle générée comme donnée pour le traitement suivant, ainsi qu'une garde qui impose un emprunt si la pile de mémorisation est pleine.

```
taille_pile:=0
proc_gardé:=0
cycle
  si non proc_gardé
     si taille_pile = 0 alors
        demander(voisin_précedent)
     sinon
        demander(pile)
     recevoir(processus)
  sinon proc_gardé:=faux
  si nbreine(processus)=8 alors
     envoyer(processus) a l'hote
  sinon si demande_voisin_suivant
     envoyer(processus) au voisin_suivant
  sinon
     pour chaque position correcte faire
     (* voir plus haut la recherche de position *)
        si proc_gardé
          si taille_pile=max_prof
            attendre(demande_voisin_suivant)
            envoyer(processus) au voisin_suivant
          sinon
            si demande_voisin_suivant
              envoyer(processus) au voisin_suivant
            sinon
              envoyer(processus) a la pile
              taille_pile:=taillle_pile+1
        sinon
           proc_gardé:=vrai
fincycle
```

Le chronogramme d'activité *(fig. 30)* montre que le réseau atteint rapidement un degré de concurrence soutenu, voisinant 50%. Les 50% d'inactivité correspondent pour l'essentiel à des attentes de données lors des dépilements, et ne peuvent donc être interprété comme une mauvaise répartition de charge (avec 8 cellules de traitement, le taux d'absence de processus à traiter est de l'ordre de 10% en régime continu). Malgré la taille des données manipulées (12 octets par processus), le taux de calcul reste d'environ 20%.

Le temps d'exploration de l'arbre complet est de 134000 cycles (0,134s) pour 8 cellules de traitement (les piles ne nécessitent chacune qu'une seule cellule de mémorisation). Avec 16 cellules de traitement, ce temps tombe à 108000 cycles. Ces performances peuvent être comparées à celle obtenue avec un 68030 à 25MHz, programmé en Pascal, qui est de 0,25 s.

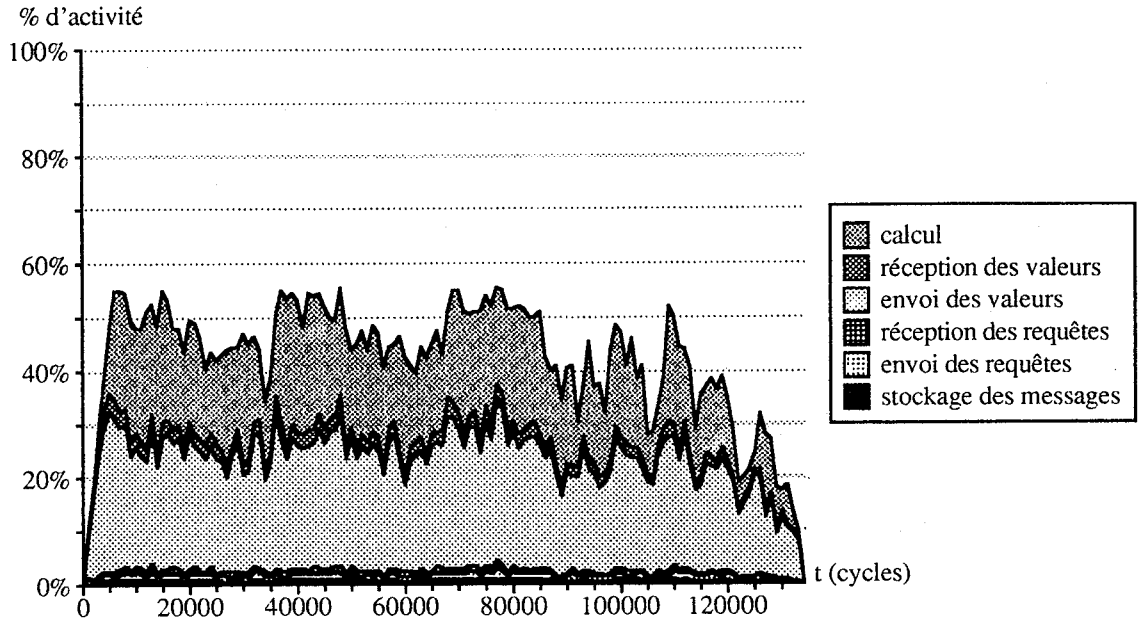


Fig. 30 — *Répartition de l'activité pour 8 cellules de traitement et 8 de mémorisation.*

Malheureusement, la combinatoire du problème des huit reines n'est pas suffisante pour permettre une étude significative du mécanisme d'équilibrage lorsqu'on fait varier le nombre de cellules.

Cette application aura été instructive sur deux points :

- elle montre comment il est possible de transformer un programme à création dynamique de processus en un programme statique à flux irréguliers
- comment programmer une pile distribuée (nous ne les avons pas détaillé ici, mais il y a quelques problèmes de codage liés au fait que le sommet de pile est connecté à deux canaux, l'un d'empilement, l'autre de dépilement, qu'il faut gérer de façon asynchrone).

Annexe 2 : Environnement de programmation et de simulation

1. Un outil modulaire d'aide à la conception et à l'expérimentation.

Plusieurs raisons ont présidé à l'adoption d'une méthode d'investigation basée sur des simulations et ont justifié le temps non négligeable qui a été consacré à l'écriture des différentes pièces logicielles.

- le caractère inhabituel de la programmation parallèle asynchrone fait que l'expérience pratique manque : faute de machine pour se "faire la main", sentir où se trouvent les problèmes et en tirer les conclusions au niveau de la conception, nous devons recourir à un simulateur. Un tel outil a à l'évidence un rôle didactique primordial, ainsi que peuvent en témoigner tous ceux qui l'ont utilisé au sein de ce projet.
- le parallélisme qui a tendance à devenir très désordonné rend difficile voire impossible l'évaluation sur le papier, ainsi que la mise en évidence de phénomènes relativement fins.
- pour la même raison, l'évaluation des différentes options d'implémentation est tout aussi délicate, que se soit au niveau du jeu d'instruction ou du système de communication.

En conséquence, un simulateur a été écrit, avec les caractéristiques suivantes : précision temporelle rigoureuse par un travail au niveau du cycle d'horloge, interface graphique interactive, contrôle de cohérence des opérations effectuées, vitesse d'exécution et tailles de réseau les plus élevés possible, modularité.

Un soin tout particulier a été porté à ce dernier point, le simulateur étant capable de supporter et de combiner un nombre quelconque de parties traitement, de jeu d'instructions, de routeurs, de protocoles de communication avec l'hôte, de d'outils d'analyse.Pour préserver les performances, une approche semi-ouverte a été mise en œuvre : interfaces standardisées pour chaque type de module, mais nécessité de recompilation à chaque ajout.

L'environnement de simulation comporte en outre un assembleur symbolique parallèle, et peut être étendu par des générateurs de plus haut niveau (LUSTRE [PAY91], simulation logique à échéancier [JAC91], réseau de neurones [FAU90]. Il représente actuellement :

Assembleur	3300 lignes de C
Simulateur	7700 lignes de C 190 lignes d'assembleur
Autres outils	500 lignes de C
total	11700 lignes

Tableau 1 — *Taille de l'environnement de simulation.*

2. L'assembleur parallèle

La spécification d'un programme parallèle peut se faire par la donnée en extension de l'ensemble des programmes élémentaires de chaque unité de traitement, mais dans le contexte du parallélisme massif, il est évident que ce n'est pas la meilleure manière de procéder. L'assembleur "parallèle" permet de mettre à profit l'existence de similitudes entre les programmes élémentaires, sans pour autant perdre en capacité d'expression des singularités. En cohérence avec le simulateur, il supporte par sa structure modulaire un nombre quelconque de jeux d'instructions.

Globalement, un programme parallèle se compose d'un programme assembleur normal, dont chaque ligne s'applique potentiellement à chaque processeur. Des directives d'assemblage conditionnel permettent de *spécifier le groupe des processeurs cibles*. Chaque processeur dispose d'un *contexte symbolique propre* , et peut accéder aux contextes des autres processeurs. Ainsi le programmeur est affranchi de la nécessité de maintenir manuellement toutes les dépendances entre cellules lors des opérations de communication.

Voici un fragment de programme source où un processeur envoi un message dans un canal de son voisin de droite :

```
 emetteurs:   EQU    ...            ; specification des groupes
 recepteurs:  EQU    ...            ; de cellules

              IF     self < emetteurs
        dest: EQU    self+0:1       ; =di:dj : voisin de droite
"WS"/   msg:  DS     1              ; revervation d'un mot
""/           DC     dest.canal     ; val de canal dans dest
              DC     self-dest      ; adresse relative
              ...
"X",send/     LDA    valeur
              STA    msg
              SEND   msg
              ...
              ENDIF

              IF     self < recepteurs
"G"/  canal:  DS     1
              ...
"X",rec/      GET    canal
              STA    valeur
              ...
              ENDIF
```

Notons les adresses de cellules, relatives ou absolues, qui sont manipulées sous la forme vectorielle di:dj ou i:j, ainsi que la forme dest.canal qui permet à l'émetteur de référer l'emplacement `canal` dans la cellule `dest` sans avoir besoin de connaître son adresse exacte.

En tête de chaque ligne, nous pouvons trouver des informations destinées au simulateur :

- "WS", "X" ou "G" indiquent quels sont les opérations permises sur les mots que génèrent postérieurement l'assembleur ; "WS" pour le champ donnée du message autorise le programme à écrire une valeur (`STA msg`) et à effectuer une émission (`SEND msg`) sur cette adresse ; cette notion de permission inspirée des machines à capacités n'existe qu'au niveau du simulateur, et permet à celui-ci de vérifier - autant que possible - la cohérence des informations qu'il exécute, de signaler les opérations suspectes et ainsi d'aider à la mise au point des programmes.
- `send`, `rec` sont des informations de zone : les instructions des programmes élémentaires sont classées par zones en fonction du rôle qu'elles jouent ; pratiquement, nous distinguons une zone calcul, une zone émission de requêtes, une zone réception de requêtes, et ainsi de suite ; le simulateur peut alors comptabiliser l'évolution du temps consacré à chaque type d'opération et aider à l'analyse du comportement du programme.

Un simulateur peut par ce type d'aptitudes se montrer utile même quand un réseau existe réellement.

La syntaxe qui soit est extraite de la documentation utilisateur de l'assembleur.

2.1. Syntaxe d'assemblage.

Pour bien comprendre les points importants de la syntaxe, il convient de garder à l'esprit que l'assemblage génère du code un ensemble des cellules. Dans le cas d'un programme très régulier, le même code sera généré pour chaque cellule du réseau, mais dans le cas inverse, les cellules peuvent être programmées différemment. Cette différenciation s'exprime au travers de la technique d'assemblage conditionnel. La principale conséquence en est que le compteur de génération (adresse à laquelle l'instruction suivante sera assemblée) peut être différent suivant les cellules, et d'une manière plus générale que chaque cellule possède une table de symboles propre.

La syntaxe d'assemblage s'inspire très fortement de celles des autres assembleurs. L'unité de base de traitement est la ligne. Une ligne peut contenir une instruction, une directive d'assemblage, un simple label, ou même rien du tout. Sa forme générale est la suivante :

<ligne> ::= [<infos>] [<label>] [<instruction>] [<commentaire>]

<label> ::= <symbole>:

Le champs <label> permet de définir un symbole comme étant égale à la valeur courante du compteur de génération. Devant une directive EQU, le label est obligatoire, et indique le symbole auquel on associe le résultat de l'expression suivant le mnémonique EQU. La présence d'un label est toutefois interdite dans certain cas (devant une directive IF, ELSE ou ENDIF, et devant une directive ORG), car sa signification ne serait pas très claire.

<instruction> ::= <mnémonique>[.<ext>] [<expression>]

Le champ <instruction> sert à spécifier une directive ou une instruction à assembler. Le mnémonique et l'extension sont des symboles pris dans des ensembles spécifiques à chaque jeu d'instruction. L'expression suit les règles grammaticales décrites plus loin ; elle n'est pas forcément entièrement évaluable,car certaines constructions syntaxiques servent à spécifier les modes d'adressages. C'est le générateur de code choisi qui va décider quels sont les portions de l'expression qui devront être calculées. Se reporter à la description du jeu d'instruction pour la forme exacte des expressions valides.

<commentaire> ::= ; <texte>

Le champ <commentaire> commence par un point virgule et se termine par le premier retour chariot rencontré. Son contenu est totalement libre. Il est à noter que dans le listing, tous les débuts de commentaires sont alignés verticalement sur la première colonne pour les commentaires débutant en première colonne dans le source, sur une même colonne calculée pour tous les autres.

<infos> ::= <zone> /
<zone>,<perm> /
<perm> /
<zone> ::= <symbole> | <entier>
<perm> ::= " <caractères> "

Le champ <info> est une nouveauté par rapport aux assembleurs traditionnels. Il sert exclusivement en conjonction avec le simulateur. Sa présence permet de spécifier certaines propriétés liées aux instructions qui vont le suivre. La zone est une information sur le rôle de l'instruction. Le simulateur est capable d'utiliser cette information pour visualiser et comptabiliser dynamiquement ce que font les cellules (calcul, émission de résultat, réception de données etc). On peut ensuite analyser (grâce à Chrono par exemple) le profil d'exécution parallèle de l'application. Pour plus de

lisibilité, on déclarera par des EQU au début du programme, des symboles pour désigner les zones[1].

Le champ <perm> sert à indiquer ce que le processeur pourra faire sur les données générées. Ces permissions sont de natures différentes est peuvent être cumulables. Voici leur signification pour la cellule standard :

R	Read : L'adresse peut être lue par le processeur
W	Write : L'adresse peut être écrite
X	eXecute : L'adresse est une instruction
S	Send : L'adresse est le début d'un message
G	Get : L'adresse est un canal d'entrée, qui peut donc recevoir un message et être lu par un GET ou un TRY
O	Overwrite : L'adresse est un canal d'entrée non utilisé pour se synchroniser. Le flag de présence n'est donc pas toujours repositionné, bien que la donnée soit lue. Cette permission permet au processeur d'écraser le message lorsqu'un nouveau message arrive (sinon une erreur est détectée).

Lorsque le simulateur détecte une violation de permission dans une cellule, il en informe l'utilisateur et s'arrête.

Une déclaration d'information reste active jusqu'à la déclaration suivante, sur la base de la cellule : elle peut être différente suivant les cellules (cf assemblage conditionnel).

Pour les lignes trop longues, le caractère anti-slash \ permet de couper la ligne logique en plusieurs lignes physiques.

Les directives sont en nombre limité : IF ELSE ENFIF, ORG, DC, DS,END et EQU.

ORG <expr entière>

Cette directive permet de spécifier une nouvelle adresse pour le compteur de génération. La valeur par défaut au début de l'assemblage pour les compteurs de génération est 0.

<symbol> **EQU** <expr>

Cette directive permet de définir des constantes symboliques. Ces constantes peuvent être de n'importe quel type calculable : entier, vecteur ou ensemble.

---

[1] Par convention, la zone 0 ne doit pas être utilisée. Elle est réservée à la comptabilisation des cycles passés à transférer les messages des tampons d'entrée dans la mémoire. Les zones à utiliser sont 1, 3, 5, etc. Lorsqu'une instruction échoue(SEND ou GET bloqué), elle est comptabilisée dans le même numéro de zone que si elle avait réussi +1.

IF <expr entière>
... (éventuellement rien)
[**ELSE**
... (éventuellement rien)]
ENDIF

Ces directives sont à la base de la différenciation des cellules. L'expression condition sera souvent un test sur la position de la cellule : IF self = 0:0 ou IF self < BordGauche ou encore IF self.j=0... Cette structure d'assemblage conditionnel est imbricable indéfiniment.

... **DC** <expr entiere ou vectorielle>{,<expr entiere ou vectorielle>,}

Cette directive permet de générer des tables de constantes ou de variables initialisées dans la mémoire des cellules. Le format exact des constantes générées dépend du générateur de code utilisé. Les expressions entières seront généralement sur un mot, les expressions vectorielles (adresses relatives dans les messages) sur deux demi-mots ou deux mots. Une directive DC peut être précédée d'un label et d'informations (en général de permissions).

... **DS** <expr entiere>

Cette directive réserver un nombre calculé de mots dans la mémoire des cellules. Ces mots sont initialisés à 0, et héritent des informations de zone et de permission courantes. Une directive DS peut être précédée d'un label et d'informations (en général de permissions).

END

Directive facultative en fin de fichier source.

L'assembleur connaît trois types de données : les entiers signés, les vecteurs et les ensembles. Les "entiers" sont ceux codés en base 2 sur 32 bits. Les entiers servent aussi à coder les booléens à la manière de C : un entier non nul est vrai, un entier nul est faux.

Un entier peut être donné sous forme littérale binaire, décimale ou hexadécimale :

binaire	%bbbb... où b∈ {0..1}
décimal	ddddd... où d∈ {0..9}
hexadécimal	$hhhh... où h∈ {0..9,A...B}

Les "vecteurs" sont des éléments de (entier x entier). Ils servent à désigner soit des adresses relatives, soit des adresses absolues. Le premier entier désigne la ligne et le second la colonne (ligne 0 = 1ère ligne) ; si v est un vecteur v.i désigne le n° de ligne, v.j désigne le n° de colonne.

Les "ensembles" sont des sous-ensembles de (0..m-1 x 0..n-1), où m est le nombre de lignes du réseau et n le nombre de colonnes. Ils servent à désigner des groupes de cellules.

Un certain nombre de variables d'assemblage sont pré-déclarées :

nom	type	contenu
I	entier	n° de ligne de la cellule
J	entier	n° de colonne de la cellule
SELF	vecteur	adresse absolue de la cellule
PC	entier	compteur de génération
SIZE	vecteur	taille du réseau

conventions :

- e,e1,e2 sont des ensembles,
- n,n1,n2 sont des entiers
- v,v1,v2 sont des vecteurs.

Opérateurs arithmétiques.

Les opérateurs arithmétiques se comportent de façon traditionnelle vis à vis des entiers. Vis à vis des vecteurs, seuls les opérateurs additifs sont disponibles, ils opérèrent comme des sommes vectorielles. Pour les ensembles, seule la différence ensembliste à un sens.

forme syntaxique	forme sémantique	signification
expr + expr	n1 + n2 v1 + v2	somme de n1 et n2 v tel que v.i = v1.i +v2.i et v.j=v1.j+v2.j
expr - expr	n1 - n2 v1 - v2 v - e e - v , v ^ e e1 - e2	différence de n1 et n2 v tel que v.i = v1.i -v2.i et v.j=v1.j-v2.j {v} - e différence ensembliste e - {v} e1 - e2
expr * expr	n1 * n2	produit de n1 et n2
expr / expr expr ÷ expr expr DIV expr	n1 / n2	quotient de n1 et n2
expr MOD expr	n1 MOD n2	modulo de n1 et n2
- expr	- n - v	complément à 2 de n v2 tel que v2.i = -v.i et v2.j = -v.j

Opérateurs booléens.

Comme en C, les opérateurs de composition logique sont distinct des opérateurs booléens bit à bit, mais seulement pour les entiers.

En ce qui concerne les ensembles, le OU est une union, le ET une intersection, le OU exclusif une disjonction et le NON un complément par rapport au plus grand ensemble possible. La différence ensembliste est décrite plus loin avec les opérateurs arithmétiques. Les vecteurs peuvent être utilisés dans les opérations ensemblistes qui les voient comme des singletons.

forme syntaxique	forme sémantique	signification
expr OR expr	n1 OR n2	ou logique de n1 et n2
expr || expr	v1 OR v2	{v1} ∪ {v2}
	e OR v , v OR e	e ∪ {v}
	e1 OR e2	e1 ∪ e2
expr AND expr	n1 AND n2	et logique de n1 et n2
expr && expr	v1 AND v2	{v1} ∩ {v2}
	e AND v , v AND e	e ∩ {v}
	e1 AND e2	e1 ∩ e2
NOT expr	NOT n	négation logique de n
! expr	NOT v	[0..m-1] x [0..n-1] - {v}
¬ expr	NOT e	[0..m-1] x [0..n-1] - e
expr | expr	n1 | n2	ou bit à bit de n1 et n2
	v1 | v2	{v1} ∪ {v2}
	e | v , v | e	e ∪ {v}
	e1 | e2	e1 ∪ e2
expr & expr	n1 & n2	et bit à bit de n1 et n2
	v1 & v2	{v1} ∩ {v2}
	e & v , v & e	e ∩ {v}
	e1 & e2	e1 ∩ e2
BNOT expr	BNOT n	complément à 1 de n
~expr	BNOT v	[0..m-1] x [0..n-1] - {v}
	BNOT e	[0..m-1] x [0..n-1] - e
expr ^ expr	n1 ^ n2	ou exclusif bit à bit de n1 et n2
	v1 ^ v2	({v1} ∪ {v2}) - ({v1} ∩ {v2})
	e ^ v , v ^ e	(e ∪ {v}) - (e ∩ {v})
	e1 ^ e2	(e1 ∪ e2) - (e1 ∩ e2)

Comparaisons.

Les comparaisons d'entiers sont celles traditionnelles

Les comparaisons d'ensembles sont le test d'égalité et de non égalité ensembliste. Les opérateurs > < ≥ et ≤ sont compris comme des tests d'inclusion au sens large (⊆).

forme syntaxique	forme sémantique	signification
expr = expr	n1 = n2 v1 = v2 e = v , v = e e1 = e2	n1=n2 (v1.i=v2.i) et (v1.j=v2.j) e = {v} e1 = e2
expr ≠ expr expr != expr	n1 ≠ n2 v1 ≠ v2 e ≠ v , v ≠ e e1 ≠ e2	n1≠n2 (v1.i=v2.i) ou (v1.j=v2.j) e ∩ {v} ≠ ø e1 ∩ e2 ≠ ø
expr < expr	n1 < n2 v1 < v2 e < v v < e e1 < e2	n1<n2 {v1} ⊆ {v2} e ⊆ {v} {v} ⊆ e e1 ⊆ e2
expr ≤ expr expr <= expr	n1 ≤ n2 v1 ≤ v2 e ≤ v v ≤ e e1 ≤ e2	n1≤n2 {v1} ⊆ {v2} e ⊆ {v} {v} ⊆ e e1 ⊆ e2
expr > expr	n1 > n2 v1 > v2 e > v v > e e1 > e2	n1>n2 {v2} ⊆ {v1} {v} ⊆ e e ⊆ {v} e2 ⊆ e1
expr ≥ expr expr >= expr	n1 ≥ n2 v1 ≥v2 e ≥ v v ≥ e e1 ≥ e2	n1≥n2 {v2} ⊆ {v1} {v} ⊆ e e ⊆ {v} e2 ⊆ e1

Autres opérateurs.

forme syntaxique	f. sémantique	signification
expr . expr	v.expr	expr évalué dans le contexte de la cellule v cas particuliers : dans v.i et v.j, v, v ne désigne pas forcément une cellule, mais on peut quand même les évaluer.
(expr)	rôle usuel des parenthèses sauf cas où elles servent pour désigner un mode d'adressage	
if expr then expr else expr endif	if n then expr1 else expr2 endif	si n est vrai, alors l'expression vaut expr1 sinon elle vaut expr2.
expr:expr	n1:n2 v1:v2	v tel que v.i=n1 et v.j=n2 e tel que $\forall v \in e, v1.i \leq v.i \leq v2.i$, $v1.j \leq v.j \leq v2.j$ (rectangle minimum de cellules englobant à la fois v1 et v2)

Opérateurs non évaluables.

Un certain nombre d'opérateurs ne servent que comme constructeurs pour exprimer les modes d'adressages. On peut ainsi inventer toutes sortes de syntaxes, suivant les besoins des jeux d'instructions.

forme syntaxique	signification
[expr]	non évaluable, pour exprimer les modes d'adressage
{expr}	non évaluable, pour exprimer les modes d'adressage
expr(expr)	non évaluable, pour exprimer les modes d'adressage
expr[expr]	non évaluable, pour exprimer les modes d'adressage
expr{expr}	non évaluable, pour exprimer les modes d'adressage
expr++	non évaluable, pour exprimer les modes d'adressage
expr--	non évaluable, pour exprimer les modes d'adressage
++expr	non évaluable, pour exprimer les modes d'adressage
--expr	non évaluable, pour exprimer les modes d'adressage
#expr	non évaluable, pour exprimer les modes d'adressage
@expr	non évaluable, pour exprimer les modes d'adressage
expr , expr	non évaluable, pour exprimer les modes d'adressage

Règles de priorités et d'associativité

opérateurs (par priorité decroissante)	associativité
() [] { } if then else endif	-
x++ x-- .	gauche à droite
++x --x ! ~ -	-
:	gauche à droite
* /	gauche à droite
+ -	gauche à droite
x() x{} x[]	gauche à droite
= ≠ < > ≤ ≥	gauche à droite
&	gauche à droite
^	gauche à droite
|	gauche à droite
AND	gauche à droite
OR	gauche à droite
# @	-
,	droite à gauche

2.2. Fonctionnement.

A la différence d'un méta-assembleur, l'ajout d'un nouveau jeu d'instructions se fait par celui d'un module C de génération de code. En fait, L'assembleur se contente de réaliser une mise en commun des aspects analyse grammaticale, manipulation de symboles, évaluation d'expressions, assemblage conditionnel etc.

Il fonctionne en plusieurs étapes :

1. Analyse lexicale

2. Analyse syntaxique : un arbre d'analyse est construit selon une grammaire bien précise commune à tous les jeux d'instructions. Le générateur de code devra partir cette structure pour repérer les sous-arbres correspondant à l'expression de modes d'adressages. Par exemple, la construction (<expr>++) est possible au niveau de la grammaire, mais non évaluable. Le générateur de trouver les sous-arbre du type (<reg>++) pour les interpréter comme des adressages post-incrémentés.

3. Analyse sémantique et calcul d'adresses : les adresses correspondant aux labels sont calculées, ainsi que les diverses expressions. En raison de la possibilité de références en avant et de cellule à cellule, il s'agit d'un processus itératif qui ne s'arrête que lorsque toutes les références sont résolu ou qu'une passe ne permet pas de réduire le problème (symboles indéfinis). Le nombre d'erreur affiché à ce niveau est peut signification, car il totalise toutes les erreurs pour toutes les cellules. Les erreurs sont signalées de façon plus précise dans le fichier listing.

4. Génération de code : tous les codes sont placés dans un même fichier objet.

3. Le simulateur

Le simulateur prend en entrée le code généré par l'assembleur, ainsi que la spécification des échanges avec l'hôte et simule à la fois celui-ci et le réseau. Il peut maintenir à volonté un certain nombre du *vues* sur le réseau par l'intermédiaire du système de fenêtrage, et générer des fichiers d'analyse de comportement : statistiques d'activité, de charge, trace des cellules, des entrées/sorties etc.

Il est paramétré par le ratio entre cycle processeur et cycle routeur, par la partie traitement et le routeur utilisé, la taille des files d'émission et de réception au niveau de chaque processeur, etc.

L'hôte

Un seul modèle d'hôte est actuellement implémenté. Il est modélisé sous la forme de flots d'entrée et de flots de sortie. Chaque flot est attaché à une cellule du bord du réseau, à une cellule partenaire avec laquelle il va communiquer, et à un fichier (fig. 1).

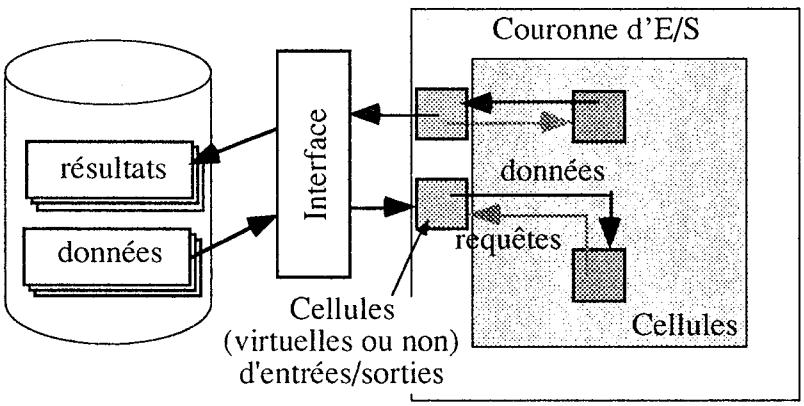


Fig. 1. — *Hôte modélisé par le simulateur.*

La stratégie de routage X-Y introduit une difficulté au niveau des entrées/sorties. Si toute cellule du bord du réseau est connectée à l'hôte, il est alors possible de choisir la cellule de bord alignée en X ou en Y comme point de raccordement et le chemin des messages est soit complètement horizontal, soit complètement vertical. Si au contraire seules quelques cellules soit reliées à l'hôte, les choix des points de raccordement est alors moins libre et les chemins des messages peuvent être amenés à faire des coudes. Or comme les messages sont d'abord acheminés en X, les messages injectés sur les côtés Nord et Sud vont vouloir partir horizontalement, ce qui est interdit pour tout tampon vertical. Le problème symétrique se présente lorsqu'une cellule veut émettre un message à destination d'un point de raccordement sur un bord Est ou Ouest.

Nous avons donc introduit la notion de *cellule d'E/S*, qui possède le même routeur que les cellules de calcul, mais auxquelles on substitue en guise de partie traitement une circuiterie spécialisée dans la gestion du protocole hôte-réseau. Le message introduit dans le réseau l'est directement dans la partie "traitement" et non dans le tampon périphérique, qui peut à la limite être supprimé. Il s'agit là d'une

solution "riche", puisqu'elle met en jeu deux versions de cellules, et donc deux circuits différents, qu'il n'est pas nécessaire d'implémenter dans un premier temps : les cellules de calcul raccordées à l'hôte peuvent être programmées pour jouer le rôle de cellule d'E/S (relais, multiplexage des voies) et former une couronne d'E/S partielle. Une étude détaillée de l'implémentation des fonctions d'entrées/sorties, ainsi qu'une évaluation des débits en fonction des solutions matérielles, pourra être trouvée dans [BOI91].

Le protocole de communication est à la base de type requête-réponse, mais avec possibilité de moduler la taille de la réponse : nombre fixe de messages successifs, ou chaîne C (terminée par un NULL).

Fragment de fichier des spécifications d'E/S:

```
fi dir=w num=0 di=0 dj=1 in=253 out=255 size=2 d1=0 d2=0 d3=0 d4=1 :prime2.dat
fo dir=e num=0 di=0 dj=-1 in=255 out=253 size=2 d1=0 d2=0 d3=0 d4=1 :prime2.res
ci dir=n num=0 di=1 dj=0 in=160 out=255 d1=0 d2=0 d3=0 d4=1 :prenoms128.bin
co dir=s num=0 di=-1 dj=0 in=255 out=0 d1=0 d2=0 d3=0 d4=1 :prenoms.res
```

Le premier symbole spécifie la taille des données (f=taille fixe, c=chaîne c, i=in, o=out), *dir* le côté de raccordement, *num* le numéro sur le côté (i pour ouest ou est, j pour nord ou sud). La cellule avec laquelle l'hôte communique est donnée par son adresse relative *di dj*.

Comme tout message comporte un tag, il faut le spécifier : *in* est le tag injecté dans le réseau, et dépend de sa destination dans la cellule réceptrice, *out* est le tag émis par le réseau, qui sera utilisé pour discriminer entre plusieurs voies logiques affectées à un même point de raccordement.

size est nombre de messages correspondant à chaque valeur logique. Lorsque *size* est différent de 1, les différents messages n'ont pas le même tag, afin de ne pas s'écraser mutuellement. Par convention, l'hôte envoie le premier message d'une valeur avec le tag *in* , et l'incrémente ensuite à chaque message jusqu'à la fin de la valeur logique. Il suppose que le réseau fait symétriquement de même. Ainsi, si l'application manipule des mots de 16 bits, ceux-ci seront rangés à des adresses successives. L'homogénéité dans la communication entre cellule (une requête pour une valeur logique) peut être respectée entre l'hôte et le réseau, ce qui supprime des cas particuliers et augmente diminue la sensibilité à la communication de certaines applications (voir dans l'annexe A, l'application TriChaîne).

d1,d2,d3,d4 sont des délais en cycles routeur sensé modéliser ceux de fonctionnement de l'hôte. Pour un canal d'entrée, ils sont interprétés de la manière suivante :

req ***d1*** RECREQ ***d2*** {***d3*** ENVVAL} ***d4***... où

req : événement d'émission d'une requête par le réseau
d1 : temps matériel de lecture de la requête
RECREQ : événement de retrait de la requête du tampon par l'hôte
d2 : temps logiciel pour déterminer la réponse
d3 : temps construire un message de réponse
ENVVAL : événement de dépôt du message par l'hôte dans un tampon
d4 : délai de ré-initialisation

Pour un canal de sortie :

d1 ENVREQ ***d2*** {...val***d3*** RECVAL} ***d4*** où

d1 : temps pour composer la requête
ENVREQ : événement de dépôt de la requête par l'hôte dans un tampon
d2 : temps pour être prêt à recevoir une réponse
val : événement d'arrivée d'une réponse
d3 : temps matériel de lecture d'une réponse
RECVAL : événement du retrait par l'hôte de la valeur du tampon
d4 : temps de traitement de la réponse

Vues

Les vues que l'utilisateur peut créer peuvent impliquer une cellule, un groupe de cellules, ou le réseau entier ; elles peuvent être générales ou spécifiques à une application.

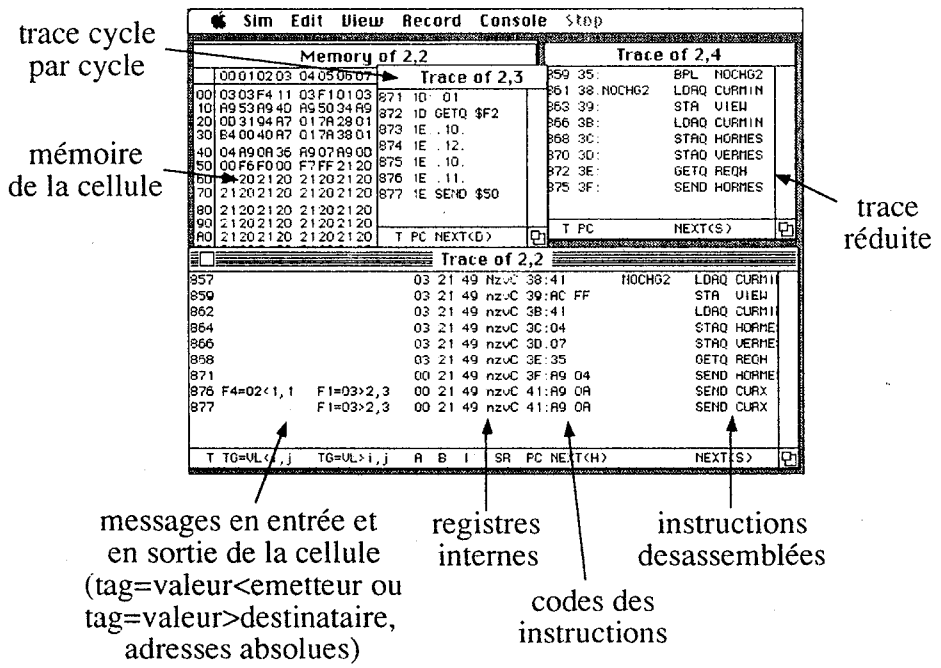


Fig. 2. —*Vues sur des cellules individuelles.*

Les vues individuelles disponibles actuellement (fig. 2.) sont les vues sur les mémoires des processeurs où chaque octet est constamment mis à jour, en rouge si le flag de réception positionné, ainsi que des vues de trace des instructions exécutées. Le contenu de chaque vue de trace est paramétrable au moment de leur création ; le format peut inclure :

- la date d'exécution
- les messages entrant ou sortant de la cellule
- les registres internes (accumulateur, registre d'index, flags)
- le compteur ordinal
- l'instruction sous forme hexadécimale, assembleur symbolique ou non symbolique.

La trace peut être basée sur les instructions (une ligne par instruction) ou sur les cycles (une ligne par cycle). Chaque trace est stockée, dans un fichier séparé, afin de pouvoir être analysée ultérieurement.

La possibilité de maintenir un nombre quelconque de vues de trace (plus d'une dizaine) est très précieuse pour analyser les interactions d'une cellule avec son voisinage.

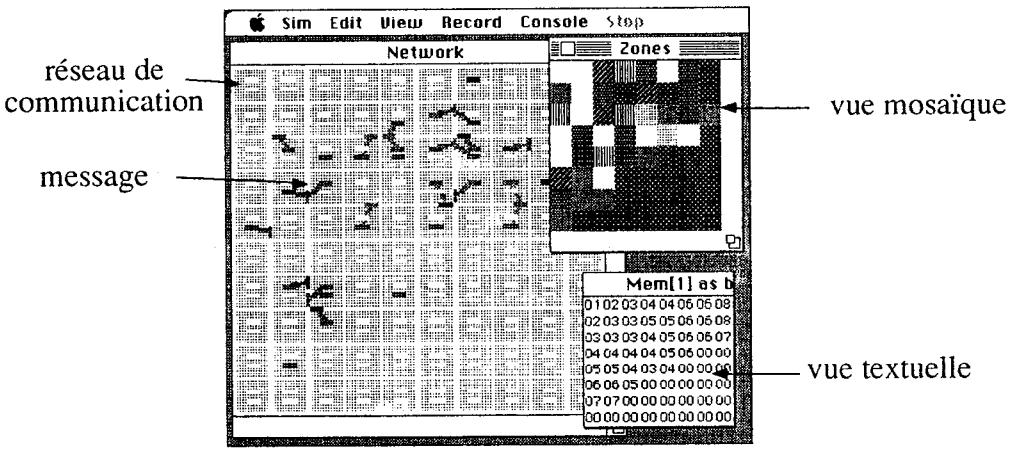


Fig. 3. — *Vues sur le réseau entier.*

Les vues globales sont actuellement de trois types. Les vues textuelles permettent de voir le contenu d'un même mot mémoire pour toutes les cellules. La figure 3 montre des vues sur l'exécution de l'application "Distance" (tableau systolique). La vue textuelle contient la variable dans laquelle est stocké la distance minimale courante. La vue réseau permet de suivre le cheminement de chaque message dans le réseau. Les vues mosaïques sont des vues chargées de présenter sous forme de couleur divers types d'information. Sur la figure B.3, l'information est le numéro de zone courant, c'est à dire le type d'activité qu'effectue chaque cellule, mais on peut assigner n'importe quel type d'information :

- binaire : 0 blanc, ≠0 noir pour une adresse donnée (pour le traitement de pixels par exemple)
- graduée : la teinte est d'autant plus vive que la valeur est élevée.

La correspondance cellule-carré est pratique, mais pas obligatoire : la vue mosaïque spécifique de l'application "Conway1D" prend pour chaque colonne de la vue une suite de mots dans la mémoire d'une cellule.

Le rôle des vues graphiques ne doit pas être négligé en ce sens que certains phénomènes peuvent être facilement appréhendé de façon intuitive grâce à ces vues là où une vue textuelle est incompréhensible. C'est le cas par exemple du problème du choix des paires d'échange dans le tri en hélice (voir Annexe A).

Rapports

Les rapports sont des fichiers textuels créés au fur et à mesure de l'exécution de l'application :

- traces d'exécution des processeurs
- trace d'activité moyenne par zone
- trace d'activité moyenne du réseau (charge, taux de collision, etc)
- trace des échanges avec l'hôte

etc.

Jobs

Les jobs permettent d'enchaîner automatiquement l'exécution d'un nombre quelconque de simulation, et de cumuler divers moyennes, afin de réaliser des benchmarks.

Simulation de pannes

L'utilisateur peut déclarer un routeur, un processeur comme défectueux. Cette fonctionnalité a été ajoutée au simulateur pour permettre le développement de programme de test pour le réseau.

Annexe 3 : Communication entre le réseau cellulaire et l'extérieur

Par principe, tout calculateur a besoin de communiquer avec son environnement, ne serait-ce que pour acquérir ses données et fournir le résultat de leur traitement. L'environnement peut être constitué en entrée par des mémoires de masse, des capteurs physiques, des consoles, en sortie par des dispositifs de visualisation et d'impression, des mémoires de masses. Pour ce qui est des supercalculateurs, on a coutume de considérer l'environnement extérieur sous la forme d'un autre ordinateur, l'hôte, qui prend en charge d'interfaçage avec le monde. Le calculateur ainsi déchargé du maximum de tâches annexes peut de consacrer pleinement sur le traitement proprement dit, les données lui étant fournies prêtes à l'emploi par l'hôte. Cela est tout particulièrement vrai pour le réseau cellulaire, qui est dépourvu de tout système d'exploitation propre.

La communication avec l'hôte peut se faire par des voies spécialisées, mais nous avons volontairement laissé de côté ce type de solutions (au demeurant intéressantes) pour une solution plus naturelle qui est de connecter l'hôte aux liens libres des cellules du bord du réseau. La bande passante potentiellement disponible est considérable, mais son exploitation effective se heurte à divers problèmes, que nous allons maintenant évoquer, sans toutefois prétendre les traiter complètement. Une maquette d'interface NuBus pour le réseau cellulaire a été réalisée par Marc Robichon et Frédéric Boiteux[BOI91], ainsi que le logiciel de gestion de cette interface. Ils ont clairement mis en évidence le goulot d'étranglement constitué par les entrées/sorties, même pour un petit nombre de canaux, et ont montré l'intérêt de dispositifs dédiés de type DMA.

1. Interface hôte / réseau.

Les entrées/sorties sont un point noir dans toutes les machines parallèles. La puissance de traitement ne peut en effet être exploité que dans la mesure où il est possible de fournir les données des calculs aux processeurs suffisamment vite. L'hôte étant généralement une machine séquentielle classique, il y a une disproportion radicale entre d'une part alimentation en données et consommation des résultats, et d'autre part consommation des données et production des résultats. La Connection Machine utilise pour ses entrées/sorties jusqu'à huit *Data Vault* ("fermes" d'une quarantaine de disques chacune) ainsi qu'un dispositif de visualisation de type écran couleur.

Dans les machines générales MIMD, nous devons aussi considérer le chargement des programmes comme une forme d'entrées/sorties, mais comme le problème est légèrement différent en terme de débit et de contrôle, nous le traiterons séparément dans la section suivante.

En ce qui concerne les entrées/sorties liées à l'exécution d'un programme, la situation peut paraître au premier abord totalement désespérée : comment faire, à partir d'un hôte séquentiel, pour alimenter plusieurs milliers de processeurs et en récupérer les

résultats ? Lorsqu'on rentre dans les détails, on trouve des cas très hétérogènes. Si nous ne pouvons en faire un inventaire complet, nous pouvons les caractériser par quelques remarques générales :

- Les E/S ne concernent souvent qu'un nombre restreint de processeurs. Le cas le plus simple est celui des tableaux systoliques où seul les processeurs périphériques communiquent avec l'hôte. Cette différence traduit le fait que les algorithmes parallèles correspondent à des problèmes de complexité supérieure à N, et donc qu'une même donnée est soit utilise plusieurs fois, soit initiatrice d'une longue série de calcul.
- Les processeurs possédant leurs propres variables locales, les échanges avec l'hôte sont limités aux données véritables. Dans notre cas, l'activation moyenne par message d'un processeur descend difficilement en dessous de 50 cycles, et est en général bien supérieure. Le débit demandé à un canal d'E/S se situe très en deçà du débit d'un bus ordinaire.
- Un gros volume de résultats implique généralement un post-traitement pour que celui-ci soit exploitable. Comme l'a mis clairement en évidence l'étude des automates de gaz sur réseau (chap. IV), ce post-traitement peut être prit en charge par le réseau (grâce au mode de fonctionnement MIMD) pour amoindrir le volume des résultats et les rendre directement exploitable.
- Souvent, des échanges importants avec l'hôte traduisent un traitement en plusieurs phases, avec des résultats partiels à mémoriser hors du réseau à entre chaque phase. Il est tout à fait possible de laisser les résultats partiels dans le réseau (cellules particulières ou positions mémoire particulières), et de reprogrammer ce dernier en prenant soin de ne pas les toucher.

Toutes ces remarques montrent bien que le problème n'est pas identique à celui de l'accès à la mémoire dans un multiprocesseur à mémoire partagé. Par contre, la nature irrégulière des flux de données peut compliquer considérablement la tâche de gestion des E/S au niveau de l'hôte. Lorsqu'une requête sort du réseau, il faut déterminer à quel canal logique elle correspond, chercher la réponse correspondante, composer un message avec un *tag* et une adresse appropriée et l'émettre. C'est la gestion de ces protocoles qui nous posera le plus de problèmes. Dans le cadre d'une machine grandeur réelle, il n'est pas envisageable de laisser cette tâche à l'hôte, lequel serait directement relié à la périphérie du réseau par un simple bus. L'introduction d'un élément actif entre hôte et réseau au niveau de chaque point d'E/S paraît nécessaire pour gérer le protocole de communication avec la cellule, pour absorber les irrégularités de flux (utilisation de FIFO), et pour minimiser la latence de réponse de l'interface. L'hôte se trouve alors en situation de manipuler les flux de données par segment plus gros, plus compatible avec des transferts DMA par exemple.

D'autre part, il n'est pas souhaitable de connecter toutes les cellules périphériques à l'hôte, car les points d'interface seraient alors non seulement sous-utilisés, mais aussi

très difficile à gérer simultanément. Une connexion à intervalles réguliers (toutes les 15 cellules) permet un accès aussi général à moindre coût. Cette option présente néanmoins une incompatibilité avec le routage X-Y, comme l'illustre la figure 1.

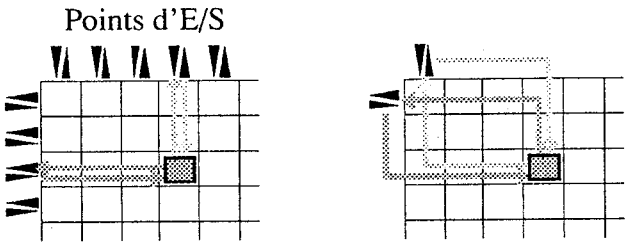


Fig. 1. — *Problème de routage aux fontières du réseau.*

Lorsqu'une cellule se trouve à portée d'adressage depuis le bord du réseau, une connexion complète de la périphérie permet d'avoir toujours au moins un point d'E/S sur la même ligne ou la même colonne. Lorsque les points d'E/S se raréfient, cette propriété disparaît et les messages sont obligés de cheminer en X puis en Y. Si le point d'E/S est sur un bord est ou ouest, les messages cellule→hôte ne pose pas de problème, mais les messages hôte→cellule arrivent au bord du réseau avant d'avoir pu cheminer verticalement. Pour les points nord et sud, la situation est inversée. Plusieurs techniques peuvent être utilisées pour y porter remède :

- utiliser la cellule attenante au point d'E/S comme relais : le cheminement vers le bord s'arrête avant la sortie pour permettre le cheminement dans l'autre direction, mais on introduit une étape supplémentaire dans la communication, et on rompt éventuellement la régularité dans l'affectation du réseau par une cellule "hors algorithme" ; cette cellule peut devenir un goulot d'étranglement si plusieurs cellules se partagent le même point d'E/S comme c'est souvent le cas.
- utiliser un signal "bord" qui indique à une cellule si elle est au bord du réseau et empêcher la sortie d'un message avant qu'il n'ait effectué le cheminement dans la direction problématique ; cette technique hardware, bien que séduisante à priori, est beaucoup moins simple à mettre en œuvre qu'il n'y paraît, car elle s'articule très mal avec la situation standard : un message sortant sur un bord est ou ouest est susceptible de cheminer horizontalement puis verticalement, puis encore une fois horizontalement ; or nous avons vu que pour de nombreuses structures d'aiguilleur, il est pratique de ne pas stocker les composantes horizontales (normalement nulle) des messages verticaux ; comment dans ce cas savoir si un message a été détourné par le signal de bord et qu'il doit subir un nouveau virage ou bien si le message a effectué normalement un changement de direction ?

Pour résoudre l'ensemble de ces problèmes, nous proposons une toute autre méthode. Il s'agit, plutôt que de les reporter sur l'hôte, de les intégrer au réseau en faisant de celui-ci l'élément actif de l'interface. Ainsi, le débit de l'interface pourra varier en même temps que la taille du réseau. Le principe de base est de spécialiser les

cellules périphériques dans la gestion des entrées/sorties : le routeur reste identique à celui des cellules normales, mais le processeur est un peu modifié pour permettre à la cellule d'accéder à une mémoire de plus grande taille, partagée avec l'hôte, par laquelle se feront les échanges. Ainsi, les cellules d'entrées/sorties peuvent prendre en charge le protocole avec les cellules de calcul, avec au niveau logiciel toute la souplesse de la programmabilité et la symétrie que permet la similitude des acteurs du protocole, et au niveau matériel la non-nécessité pour l'hôte d'implémenter un protocole physique symétrique de celui des cellules (sérialisation, multiplexage aux frontières, etc.). La tâche d'accès aux données est de cette façon distribuée comme l'est le calcul. De plus, comme les échanges ne se font plus par les tampons de bord, le problème évoqué plus haut disparaît. Il faut par contre éviter de mixer dans un même circuit cellules normales et cellules d'E/S, pour ne pas multiplier le nombre de circuits différents à fabriquer.

Nous avons souvent rencontré dans le chapitre IV portant sur la programmation la nécessité de mémoriser des données de façon massive (gaz sur réseau, problème des 8 reines). La présence d'une telle mémoire sur les bords du réseau permet de ne pas gaspiller des cellules (qui présentent un ratio *mémorisation / coût* très médiocre) pour implémenter de simple FIFO.

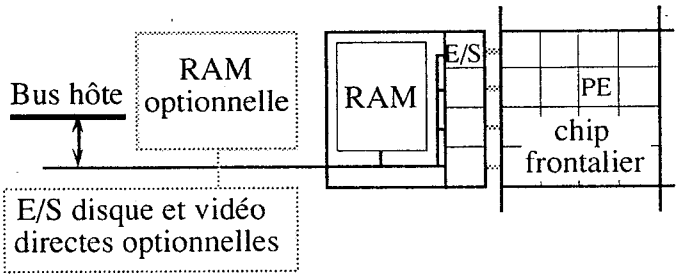


Fig. 2. — *Système d'entrées/sortie avec cellules spécialisées.*

Les fonctionnalités supplémentaires à inclure dans les processeurs d'E/S ne sont pas très compliquées : il s'agit simplement de pouvoir adresser une position mémoire externe (adressage 32 bits). Pour le reste, la cellule d'E/S garde toutes les capacités d'une cellule normale, dont un programme en mémoire privée. La conception de la cellule d'E/S sera donc très simple, par vulgaire adaptation de la cellule normale. Etant donné les relativement faibles débits imposés aux points d'interface élémentaires, une connexion de toutes les cellules d'E/S d'un même chip à un bus commun ne pose pas de problème. La place libre du chip est utilisable pour de la RAM, qui peut être étendue de manière externe avec des circuits du commerce. Sur ce bus externe, nous pouvons connecter toute sorte de dispositifs comme des E/S disque et vidéo directe (à la manière de l'I/O system de la Connection Machine), mais aussi et surtout, nous pouvons nous raccorder au bus de l'hôte. Ce dernier pourra alimenter directement, ou à l'aide d'un contrôleur DMA la RAM locale de chaque point d'E/S, qui se chargera de la mise en forme finale des données et de leur "instillation au compte-goutte" vers le réseau.

Cette méthode, qui laisse une grande autonomie de gestion des entrées/sortie au réseau, devrait nous permettre sans problème d'alimenter un réseau de taille quelconque à un coût très raisonnable en conception et en matériel.

2. Chargement.

Le mécanisme d'acheminement de messages tel qu'il est implémenté fixe une limite à la portée d'adressage, c'est à dire la distance maximum à laquelle on peut envoyer un message. Théoriquement, il faudrait que chaque cellule puisse communiquer avec n'importe quelle autre, quel que soit son éloignement. Le réseau n'étant pas de taille bornée, la portée nécessaire est donc infinie ou au moins très grande. Malheureusement, un tel adressage infini n'est pas gratuit d'un point de vue matériel, et d'un point de vue performance. L'expérience nous enseignant que les programmes présentent une localité de référence importante en phase calcul, il convient d'examiner ce problème plus en détail.

Lorsqu'on cherche à implémenter un adressage infini, des problèmes apparaissent à plusieurs niveaux :

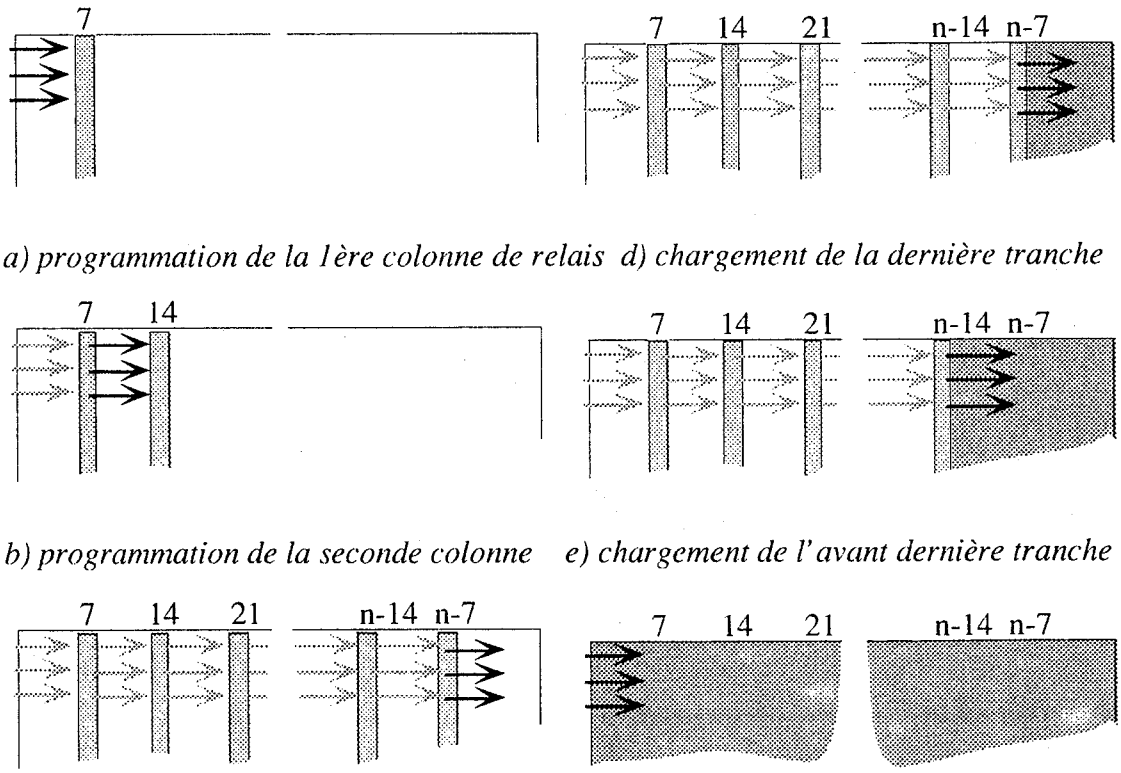
- pour porter loin, il faut générer ou stocker des adresses longues dans la partie traitement. On a donc un coût en encombrement et en temps de manipulation.
- pour véhiculer des adresses longues, il faut avoir des canaux de communications larges, donc coûteux et sous utilisés dans la plupart des cas. On peut aussi se résigner à sérialiser, ce qui implique baisse de performances et augmentation de la charge du réseau (cf étude de l'implémentation des liens de communication).
- pour aller vite en sérialisant, il faut réduire la longueur des messages au minimum, et donc représenter les adresses dans des codages de longueurs proportionnées à la portée correspondante (adresses de tailles variables). Il n'est en effet pas utile de représenter sur 32 bits les adresses dans des programmes où la communication se fait de voisin à voisin ! Le wormhole permet des adresses de taille variable, mais il occasionne un sur-coût matériel au niveau de l'aiguilleur, et au niveau de la taille des messages. Cette méthode est sans doute très intéressante si les données sont elles mêmes de longueurs variables.

C'est pourquoi nous avons opté pour un adressage de longueur fixe et courte, qui permet un traitement optimal dans la grande majorité des cas, et qui de coût tout à fait modeste. La détermination de la taille exacte des adresses a été ensuite plus une question de facilité de manipulation au niveau du processeur et de respect des proportions à dans la composition u message que le résultat d'une étude statistique des programmes. Sur une architecture 8 bits, le message ayant 16 bits de données (dont 8 de typage), une adresse de 8 bits semble tout à fait suffisante. Cette adresse 8 bits est divisée en deux composantes 4 bits codées en complément à 2.

Ce choix comporte toutefois un inconvénient : comment fait-on dans les 1% des cas où une telle portée est insuffisante, et surtout comment fait-on pour charger le programme dans les cellules ? Rappelons que le passage de message étant le mode de communication exclusif des cellules, et donc que le chargement du programme élémentaire de chacune d'elle doit se faire par une série d'envois de messages contenant le code exécutable. Rappelons aussi que notre architecture est sensée pouvoir contenir un nombre quelconque de cellules, ce qui veut dire au moins quelques milliers. Avec une topologie plane, seule la couronne externe est adressable par l'ordinateur hôte.

2.1. Utilisation de relais.

Une méthode purement logicielle consiste à programmer des cellules en "relais d'acheminement", c'est à dire à utiliser certaines cellules pour réémettre les messages qu'elles reçoivent vers d'autres cellules. En mettant bout à bout plusieurs relais, nous pouvons adresser n'importe quelle cellule. Si ces relais peuvent être envisagés en phase calcul (quelques cellules sont programmées statiquement en relais, chacune prenant en charge plusieurs liens logiques de communication), pour le chargement du réseau ils sont d'un maniement relativement lourd. A titre d'exemple, nous pouvons étudier l'une des multiples stratégies de remplissage, qui consiste à programmer le réseau tranche par tranche.



a) programmation de la 1ère colonne de relais *d) chargement de la dernière tranche*

b) programmation de la seconde colonne *e) chargement de l'avant dernière tranche*

c) programmation de la dernière col. de relais. *f) chargement de la première tranche.*

Fig. 3. — *Programmation par relais.*

2.2. Utilisation de signaux globaux.

N'existe t-il pas de solution hardware peu coûteuse pour résoudre le problème du chargement ? Il est possible de s'affranchir des limites d'adressage avec un seul signal externe par cellule. Le principe est assez simple : puisque les cellules périphériques nous gênent pour accéder au cœur du réseau, faisons en sorte de les cacher, de les rendre "transparentes" vis à vis du routage.

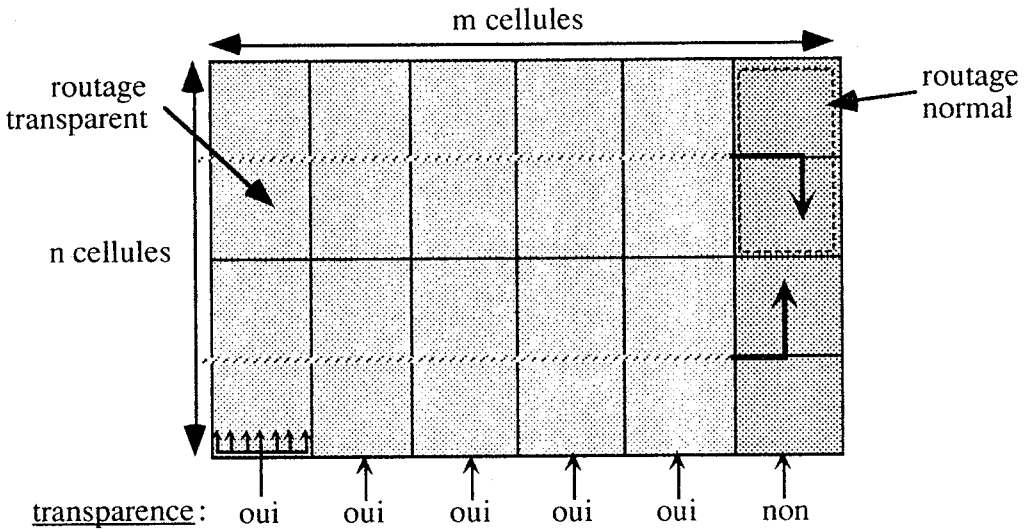


Fig. 4. — *Mécanisme de transparence horizontale.*

Le réseau est découpé en tranches de 8 colonnes de large. Les messages sortent inchangées des cellules des colonnes transparentes qu'ils traversent, la décrémentation des adresses relatives ne s'effectuant que dans les cellules des colonnes non transparentes.

Pour chaque combinaison de transparence, nous avons donc 8 x n cellules directement accessibles. Les regroupements en tranches permettent de diviser par 8 le nombre de signaux différents à générer par la logique externe. De plus, pour un circuit multi-cellules, si nous restons dans la limite de 8x8 cellules par circuit, ce qui est considérable, il n'y a besoin que d'un seul plot puisque le signal de transparence est le même pour tous les éléments de la tranche.

Ce principe peut être amélioré en ajoutant à cette transparence "horizontale" une transparence "verticale". En effet, il faut connecter à l'extérieur une cellule sur 16 du bord ouest pour charger tout le réseau. On peut se ramener à une seule cellule connectée pour le chargement complet du réseau grâce à la combinaison de ces deux transparences.

L'ordinateur hôte ne peut de tout façon alimenter un nombre quelconque de points d'entrée dont le débit est d'environ 15 Mo/s à 10 MHz (un message, soit 24 bits tous les deux cycles).

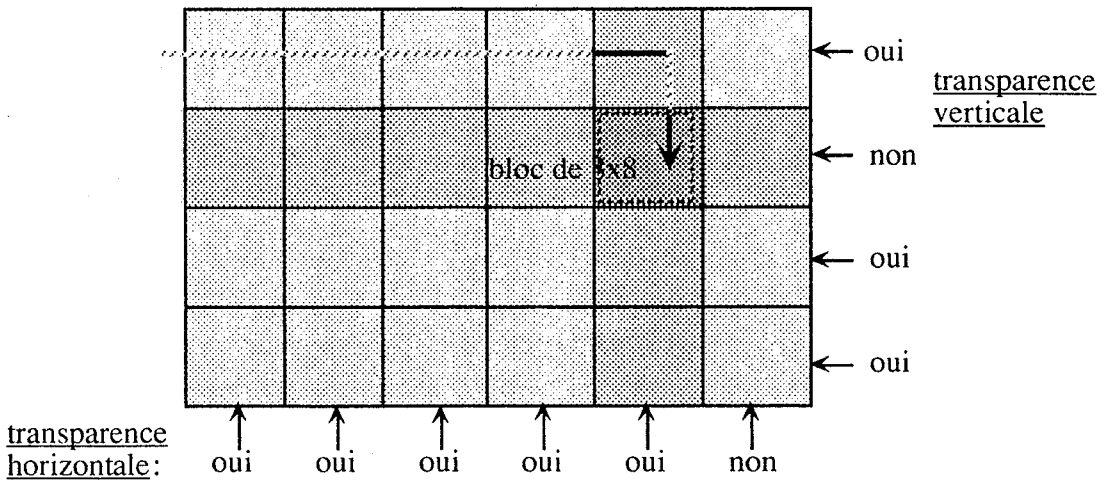


Fig. 5. — *Mécanisme de transparence bidirectionnelle.*

Le bloc directement accessible n'est plus alors que de 8 x 8 = 64 cellules, mais il reste possible d'avoir plusieurs points d'entrées.

L'algorithme de chargement est le suivant :

```
pour Ih = 1 à n/8
  pour Ih2 = 1 à n/8
    si Ih = Ih2 alors
      transparence_verticale(Ih2) = 1
    sinon
      transparence_verticale(Ih2) = 0
  pour Jh = 1 à m/8
    pour Jh2 = 1 à m/8
      si Jh = Jh2 alors
        transparence_horizontale(Jh2) = 1
      sinon
        transparence_horizontale(Jh2) = 0
    pour Il = 1 à 8
    pour Jl = 1 à 8
      pour k = 1 à taille_programme
          envoyer prog(k) à (Il,Jl)
```

Quel que soit le mécanisme de chargement adopté, il est nécessaire d'assurer que le démarrage asynchrone des cellules n'induit pas un comportement incorrect. Avec un chargement par relais, les premières cellules complètement programmées n'ont pas de moyen de savoir quand les dernières seront complètement chargées, ni quand l'ordinateur hôte sera prêt à supporter les entrées sorties relatives au calcul.

On pourrait imaginer d'avoir un signal "GO" sous forme de messages, comme première "entrée" de l'exécution, que toutes les cellules attendraient pour démarrer, mais on a le même problème pour atteindre les cellules non périphériques.

Le mécanisme de transparence offre à peu de frais un autre moyen de synchroniser le démarrage. C'est d'autant plus important qu'il ne faut absolument pas que des messages d'exécution commencent à circuler tant le réseau n'est pas en totale non transparence, faute de quoi le routage de ceux-ci serait complètement faussé.

Si on se réfère à l'algorithme de chargement cité plus haut, les cellules sont chargées ligne par ligne. Au cours du chargement, toutes les cellules passeront une seule fois en non-transparence horizontale. A condition d'avoir un état complètement transparent horizontalement au début et à la fin du chargement, et de remettre tout le réseau en non transparence d'un seul coup ensuite, on a un signal global de départ : la seconde transition transparence/non-transparence horizontale. Il est très facile de concevoir l'automate de contrôle du processeur pour qu'il détecte ce signal.

2.3. Performances.

La solution matérielle permet un chargement beaucoup plus rapide des programmes que la solution logicielle à base de relais, sauf dans un cas que nous avons pas encore mentionné : il s'agit de celui où les programmes sont relativement semblables et où leur chargement s'apparente à une diffusion. Une cellule chargée peut alors programmer par clonage d'elle-même plusieurs autres cellules. Le temps de chargement total peut être alors en $O(\sqrt{N})$ où N est le nombre de cellules du programme[1].

Le tableau 1 montre les temps prévus pour diverses tailles de réseaux. Les méthodes de chargement par donnée du programme de chaque cellule (relais et transparence), sont évaluée pour un seul point d'E/S et pour autant de points d'E/S que possible ($4\sqrt{N}$ cellules frontalières, ou $28\sqrt{N}$ cellules adressables depuis le bord), ce qui constitue les bornes extrêmes. Pour plusieurs points, le temps de chargement est divisé d'autant par rapport à un seul point, ce qui nous permet d'affirmer que le temps de chargement sera essentiellement limité par la vitesse à laquelle l'hôte manipulera le programme et alimentera les points d'E/S.

---

[1] Il serait en $O(\log_2 N)$ si nous pouvions faire une diffusion arborescente binaire avec un temps de communication sur chaque arc qui soit homogène, mais cela est impossible sur une grille.

taille du réseau	mémoire totale	relais (500 K/s) 1 point	relais 28√N points	transparence (5 Mo/s) 1 point	transparence 4√N points	diffusion
1 K cellules	256 Ko	0,5 s	1,4 ms	0,05 s	< 1 ms	1,3 ms
16 K cellules	4 Mo	8 s	5,7 ms	0,8 s	1,5 ms	1,5 ms
64 K cellules	16 Mo	32 s	11 ms	3,2 s	3 ms	1,7 ms
1 M Cellules	256 Mo	524 s	46 ms	52 s	13 ms	3,1 ms

Tableau 1 — *Temps de chargement estimés.*

La structure de réseau permet donc le chargement de très gros réseaux dans des temps proche de la milliseconde, les performances ne dépendant que de l'hôte et du système d'accès au programme à charger. D'autre part, le chargement par transparence, si le nombre de points d'E/S est suffisant, peut se montrer compétitif avec le chargement par diffusion même sur un million de cellules.

Pour conclure sur le chargement, il convient de noter d'une part que le coût de la solution matérielle est faible (1 ou 2 signaux globaux distribués à chaque chip et une petite logique d'inhibition de la modification d'adresse au niveau de chaque routeur), d'autre part que les deux méthodes ne sont pas exclusives l'une de l'autre, le chargement par diffusion logicielle pouvant être mis en œuvre dans les programmes réguliers, et le chargement à l'aide de signaux globaux dans le cas des programmes irréguliers.

Si nous reprenons la structure d'interface avec cellules spécialisées, une solution intermédiaire entre diffusion et chargement naïf peut être mise en œuvre : l'hôte fourni à chaque point d'entrée sortie un prototype de programme, qui sera ensuite distribué par le point à chaque cellule dont il est responsable, avec éventuellement des altérations mineures liées aux conditions de bord. Le volume manipulé par l'hôte est ainsi fortement réduit. Une solution encore plus bâtarde consiste à fournir un prototype de programme global associé à la donnée individuelle des tables de paramétrage de ce prototype. Elle est intéressante pour les programmes à opérateur homogène et topologie aléatoire (dans nos exemples : recherche de plus court chemin, réseaux de neurones, simulation logique).

3. Conclusion.

La conclusion principale que nous pouvons tirer de ce qui précède est qu'il est important de laisser au réseau le contrôle des opérations d'entrées/sorties et ainsi de permettre au sein de celle-ci un parallélisme réel. Toute autre option revient fatalement

à placer l'hôte en position de goulot d'étranglement, et à limiter la taille de réseau exploitable.

Une solution haute permet un accès concurrent par les cellules au système de stockage de masse, une solution basse laisse son contrôle à la charge de l'hôte, mais permet à celui-ci de fournir les données au réseau sous une forme brute, qui sera ensuite élaborée en parallèle par les cellules d'entrées/sorties. Dans l'hypothèse d'une solution basse, le chargement du programme de très gros réseaux cellulaires (millions de cellules) reste raisonnable en volume et en temps.

La structure de point d'entrées/sorties proposée (cellule spécialisée associée à une mémoire externe locale), permet d'allier une souplesse de gestion susceptible de réduire les volumes d'informations avec un parallélisme d'entrées/sorties garant d'un débit évoluant avec la taille du réseau.

Bien entendu, nous n'avons encore qu'une esquisse de solution au problème de l'interface avec l'hôte, qui nécessite encore beaucoup de travail pour être implémentée ; l'objet de cet aperçu n'aura été que de montrer que le problème d'interface hôte/réseau n'est pas sans espoir.

60984 81800