

À cause de french.sty :  
Orsay  
N° d'ordre :

UNIVERSITÉ DE PARIS-SUD  
CENTRE D'ORSAY  
THÈSE  
présentée  
pour obtenir  
le GRADE de DOCTEUR EN SCIENCES  
DE L'UNIVERSITÉ PARIS XI ORSAY  
par  
Ronan KERYELL

Sujet :

POMP :  
d'un Petit Ordinateur Massivement Parallèle SIMD à base de  
processeurs RISC  
—  
Concepts, Étude et Réalisation

Soutenue le 1<sup>er</sup> octobre 1992 devant la Commission d'examen

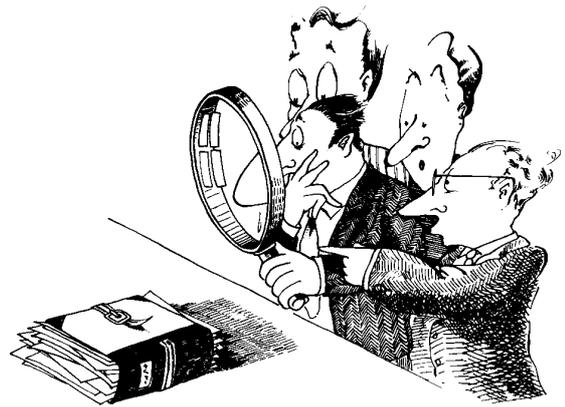
MM.	Patrice QUINTON	Président
	Michel AUGUIN	Rapporteur
	Daniel LITAIZE	Rapporteur
	Luc BOUGÉ	
	Francis DEVOS	
	Nicolas PARIS	
	Philippe MATHERAT	Directeur de Thèse



## Quelques remerciements

**J**E tiens à remercier tout particulièrement Philippe Matherat de m'avoir proposé ce sujet de thèse intéressant et éclectique qui m'a fait apprendre beaucoup de choses dans de nombreux domaines et fait prendre conscience que l'architecture des ordinateurs comprend bien sûr la description matériel mais aussi les problèmes de langage, la programmation, la compilation, les applications, etc. Philippe s'est aussi occupé de toute l'équipe alors qu'elle était sur le point de disparaître à la fin du LIE et m'a permis ainsi de préparer et soutenir cette thèse.

Merci globalement aux membres du jury qui me font l'honneur de s'intéresser à mes travaux de recherche en acceptant de perdre un peu de leur temps précieux à me juger, et tout particulièrement les membres rapporteurs qui n'ont pas demandé à lire cette thèse avant d'accepter de la rapporter<sup>1</sup>. J'espère qu'ils me pardonneront que ce temps coïncide bizarrement avec leur période de vacances et que ma thèse tienne mal à l'eau de mer et au sable. Je les en remercie en souhaitant que leur lecture n'a pas été trop insupportable à travers les tronçons successifs et incomplets que je leur ai envoyé. Qu'ils se rassurent en se disant que peut-être cela m'arrivera un jour d'être rapporteur, ou tout au moins membre d'un jury de thèse<sup>2</sup>.



Merci à Daniel Litaize, rapporteur, des discussions intéressantes qu'on a eu ensemble, particulièrement lors de PARLE '89. J'espère qu'il me pardonnera de ne pas l'avoir écouté lorsqu'il me disaient du mal du wrapping. Pardon, j'étais jeune, je ne recommencerais pas !

En particulier merci à Michel Auguin, aussi mon rapporteur, qui m'a envoyé toutes les thèses sur la très intéressante machine Opsila alors qu'elles étaient devenues des choses rares.

Merci à Patrice Quinton de vouloir perdre son temps sur ma thèse en acceptant d'en présider le jury.

---

1. Heureusement car cela aurait repoussé encore plus la soutenance !

2. Mais avant, j'irais peut-être les retraumatiser avec une habilitation...

On a eu des discussions intéressantes avec Luc Bougé et son équipe sur les problèmes de sémantique et je lui dois des idées de présentation, en particulier pour les modèles de programmation. En outre il a eu la bonne idée de créer la section Parallélisme de Données (ParDon ?) du PRC C<sup>3</sup>. Qu'un merci lui soit rendu aussi pour sa participation au jury.

Merci à Francis Devos d'avoir accepté de faire partie de mon jury et qu'il pardonna le fait que mon unique moyen de communication à distance soit le courrier électronique.

Nicolas Paris a été et est toujours la locomotive de l'équipe et j'ai appris beaucoup de chose à son contact (parfois explosif lorsque je l'exaspérais avec mes questions idiotes alors qu'il rédigeait sa thèse!). Il m'a beaucoup aidé à entretenir une ambiance « démente » dans le bureau B1 du passage bleu. Cela a été un réel plaisir de travailler avec lui, que ce soit sur de l'informatique (on s'est vraiment bien amusé avec nos projets) ou à tester différentes recettes de gâteaux au chocolat, sa spécialité.

Philippe Hoogvorst a encore accentué cette ambiance à son retour et nous a bien aidé dans le développement des applications en POMPC et la mise au point de ce langage.

Les travaux exposés sont aussi les fruits d'une équipe et je tiens à en remercier les membres : Philippe Matherat, Nicolas Paris et Philippe Hoogvorst bien sûr mais aussi ceux qui ont fait partie intégrante de l'équipe d'architecture des ordinateurs du LIENS<sup>3</sup> au moment où je suis arrivé : César Douady qui supervisait le projet, Patrice Ossona Demendez qui a étudié l'ALU flottante et qui appréciait particulièrement la division<sup>4</sup>, Théodore Papadopoulo qui avait fait les premières études sur le langage de la machine et les problèmes de virtualisation, Pierre Chicourrat qui a travaillé sur MARE (un Macro-Assembleur REconfigurable), tout récemment Boukhalfa Mustapha Hadim qui a travaillé sur les opérations préfixes parallèles en POMPC qui m'a permis au passage de m'exercer au co-encadrement de DEA<sup>5</sup>, Thierry Porcher qui nous a rejoint après et qui a travaillé sur le compilateur POMPC pour iPSC/860 et a répandu le langage à l'ETCA tout en assumant son support local.

François Irigoïn m'a ouvert la bibliothèque, un compte sur les machines du Centre de Recherche en Informatique de l'École des Mines de Paris ainsi que ses pensées en ce qui concerne les dessus et les dessous de la vraie recherche en informatique. Cela m'a fait prendre conscience de beaucoup de chose qui m'auraient échappé sans le style d'humour local un peu hexafluorosulphonique. J'espère qu'il ne culpabilise pas trop de m'avoir accordé autant de temps. Merci aussi à la bibliothécaire, Annie Pech-Ripaud, à qui j'ai dû fournir du travail supplémentaire alors que je n'avais pas à être là. Je leur dois la majeure partie de ma bibliographie (période de 1968 à nos jours) et j'espère que leur photocopieuse et leur budget s'en remettront. J'ai aussi appris au CRI beaucoup de choses sur la parallélisation et la vectorisation en général

---

3. ALIENS en abrégé.

4. Et de manière générale tout ce qui était complexe. D'ailleurs il me semble qu'il fait ou a fait une thèse pluridisciplinaire sur la complexité, du VLSI à la théorie des graphes en passant par la biologie. J'espère qu'il ne m'en veut pas de l'avoir rencontré en haut du Nemrut Dağ en Turquie et que cela ne lui a pas gâché ses vacances.

5. J'espère qu'il s'en remettra et qu'il me pardonnera le fait de ne pas lui avoir consacré autant de temps que j'aurais dû, à cause de ma fin de thèse. J'espère aussi qu'il me pardonnera le fait de ne pas avoir trouvé de fonte T<sub>E</sub>X berbère et donc de devoir citer son nom platement approximé avec l'alphabet français.

et de Fortran en particulier.

J'ai passé de bon moments avec les chercheurs de Brest et Rennes, en particulier Jean-Marie Filloque et Bernard Pottier, à échanger des idées sur nos projets respectifs et nos philosophies de la vie qui ont beaucoup de points communs. La thèse de Jean-Marie m'a bien été utile en particulier pour mettre au propre la partie sur les synchronisations (§ 12.6.1). J'espère que les projets communs que nous avons initié autour de l'apprentissage de POMPC à ArMen ne lui fera pas oublier CArMen. Robert Rannou m'a exhumé de la documentation sur la machine PROPAL II, ainsi que [Rap83].

Merci à Benoît de Dinechin de m'avoir fait découvrir les opérations préfixes parallèles en me passant les articles qu'il possédait sur le sujet. Ces opérations sont décidément bien utiles dans de nombreux domaines et ont certainement un sens philosophique fondamental (que je cherche encore).

Je remercie le personnel travaillant dans notre bibliothèque, en particulier Marie-José Carrez pour son travail sur nos publications mais surtout pour la coopération que nous avons eu sur l'informatisation de la bibliothèque, la joie de voir Texto faire une gestion correcte des caractères accentués, etc. J'ai appris plein de chose sur les bases de données, les systèmes documentaires, les codes à barres, les bugs de Texto sans service après vente et surtout sur la complexité des normes de cataloguage des bibliothèques et des indexations.

Merci à toutes les personnes qui ont contribué de près ou de loin à ma bibliographie, que ce soit le personnel du centre de documentation de l'INRIA à Rocquencourt (principalement la période 1960-1970) ou M<sup>me</sup> Renzetti de la Médiathèque de l'IMAG à Grenoble pour quelques articles d'avant 1960, Christian Carrez de m'avoir fourni [Ung58]<sup>6</sup>.

Dominique d'Humières et Stéphane Zaleski ont bien voulu répondre à mes questions en ce qui concerne les machines RAP et plus généralement sur les gaz sur réseau, ce qui m'a permis de faire un petit exemple d'application intéressant.

Merci à Philippe Clermont de nous avoir fait confiance et nous avoir poussé à une époque où on voulait tout laisser tomber. Son optimisme a permis de nous motiver avec Nicolas Paris et Georges Quenot lors des réunions au sujet de la succession industrielle de nos projets, et ce malgré toutes nos tentatives de « retour à la raison ».

Les personnes avec qui j'ai eu l'occasion d'échanger des idées (parfois opposées) lors de congrès ou de séminaires, en particulier le PRC C<sup>3</sup>-SIMD, les rencontres ArMen et le PRC ANM, ont été très intéressantes et productives.

Merci à tous ceux qui nous ont donné des accès à des machines parallèles, à l'ETCA, au LIP, à Computer General.

Je tiens bien sûr à remercier tous les gens plus ou moins anonymes qui nous ont donné des moyens de subsistance sur ce projet, à savoir le MRT 88.E.0459 avec TDI, MRT 91.S.0891, ma bourse BDI, le CNRS, le LIENS, l'ENS<sup>7</sup>, l'intendance avec le logement et la bonne nourriture (surtout le « rad », indispensable à l'alimentation du projet POMP grâce à la complicité de toute l'équipe du « Pôt » — la cantine — et son Grand Chef<sup>8</sup>), le PRC ANM, etc.

6. Sur lequel il avait fait un exposé de Master à l'époque (!).

7. ULM : Université à Loyer Modéré...

8. Qui sait par exemple faire la crème renversée sans trop la cuire. Merci à lui de m'avoir révélé entre autres ce secret.

Jacques Beigbeder a eu la patience, comme avec beaucoup de monde, de répondre à mes questions de débutant en ce qui concerne la manière de bien jouer à UNIX et danser dans les bals folks. En plus il a permis à toute l'École de travailler dans un environnement système exceptionnel<sup>9</sup>. J'espère que j'arriverai à faire suivre cet environnement avec mes déplacements. Evidemment, je remercie aussi le SPI en général qui s'occupe de toutes nos petites machines et aussi Catherine Le Bihan qui m'a aidé à faire du test de stations de travail en milieu musical hostile<sup>10</sup>.

Un grand merci aussi aux secrétaires du labo, en particulier Annie Lapteff qui s'occupait des comptes de l'équipe et des bons de commande ainsi que des diverses fournitures indispensables, et Brigitte Van Eisen qui connaît vraiment tout sur le CNRS et m'a donné les renseignements permettant de m'inscrire au chômage après une bourse BDI.

Merci à Michel Baudin de m'avoir parlé du Centre de Documentation de Bull et de m'avoir fait rencontrer Mathieu Barrois qui m'a fourni un certain nombre de documentations [Bul57, Bul60].

Merci à Stéphanie Even de m'avoir, entre autres choses au moins toutes aussi importantes, su[pl\_]porté dans mon travail et pour les discussions que nous avons eu en matière de simulation numérique, de modélisation et de programmation en environnement de « vrai programmeur » avec la découverte des **common**, **entry** et autre **equivalence**, la non portabilité des programmes Fortran, etc. J'espère que le Centre d'Informatique Géologique de l'École des Mines de Paris ne m'en voudra pas de lui avoir conseillé le C pour écrire son modèle. Mais c'était plus que justifiable. Il y a un début à tout... Enfin, elle m'a convaincu que « faire de la biblio » pouvait servir à quelque chose.

Merci à Patrick Cassam-Chenaï d'avoir trouvé la citation du chapitre 13 que j'ai repris de sa thèse [CC92].

Monsieur Jean Bru, un dieu de l'autocommutateur CP 400, qui s'est occupé du Club Microtel de Limoges, m'a fait confiance et m'a permis de faire mes premières armes en informatique, rare et chère à l'époque. Merci aussi à Philippe Vigier avec qui j'ai bien « bidouillé » à Microtel puis dans la société SICTEL.

Mes pauvres parents, ma pauvre sœur et ma famille en général ont eu bien du mérite de me supporter pendant toutes ces années avec des fils électriques disparates dans toute la maison qui débordaient de ma chambre. Mais après tout, mon père n'avait pas qu'à m'offrir Electricité 2000 et Le petit radio puis à me prêter son HP-97...

Merci à Louis et Emilienne Mahé (respectivement oncle et tante) pour m'avoir changé les idées pendant les fins de semaines du temps où j'étais à Rennes. Je n'oublierai pas les bonnes parties de rigolade (ni mon permis de conduire!). Merci aussi Louis pour tes solutions subtiles à certains exercices de maths.

Stéphanie Even, Jacques Beigbeder, Christophe Herbaut, Gwenola Lestarquit, Catherine Lebihan et les autres ont su me concocter ou aider à la confection et au bon déroulement d'un pot de thèse dépassant même mon appétit habituel, ce qui fut une gageure. Je ne remercie pas par contre les gens qui n'ont pas mangé suffisamment...

Laurent Daverio m'a aussi aidé à sonoriser la « salle machines » et autres bals avec son accordéon diatonique, quand je ne l'exploitais pas à corriger (traduire?...)

9. C'est à dire qu'il convient au rôle exigeant que je suis.

10. Elles ont résisté à l'accordéon diatonique et à la bombarde.

nos articles et nos transparents en anglais<sup>11</sup>. Merci aussi à Jacques Beigbeder et Catherine Le Bihan, pour ne citer que les « locaux », pour leur coopération à d'autres bals.

Les gens du Laboratoire d'Informatique de l'École Normale Supérieure ont souvent été très sympathiques. J'espère que je le leur ai rendu au moins autant de sympathie et ne les ai pas trop embêtés. Après tout, on faisait partie de ces attractions reliquaires qui trempaient encore dans le cambouis lors que tout le reste du DMI ou presque faisaient de la théorie. Peut-être ne savent-ils pas : l'arôme de l'informatique, le goût de la soudure en fusion avec parfois le silicium qui décolle et le système, c'est quelque chose !

Merci à Isabelle Gaudron d'être venu papoter de temps en temps dans notre bureau.

Merci en particulier aux membres inconnus du défunt Laboratoire d'Informatique Expérimentale qui ont laissé traîner M0 les fils à l'air devant une fenêtre du labo lorsque j'étais en première année : sans cela je n'aurais jamais su qu'il y avait autre chose que le «  $\lambda$ -calcul » en informatique à Ulm.

Malheureusement, pour le Laboratoire de Chimie de l'École Normale Supérieure, cela s'est traduit par une réorientation. Je suis donc très reconnaissant à Gilberte Chambaud surtout mais aussi à d'autres de m'avoir permis de revenir « en douceur » à mes amours de jeunesse après avoir découvert les plaisirs de la chimie. Il m'était difficile de mener de front correctement plusieurs choses...

Donald Knuth et Leslie Lamport ont eu l'idée géniale de proposer des programmes de composition de texte (respectivement  $\text{T}_{\text{E}}\text{X}$  et  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  capable de faire croire à un chercheur qu'il fait de la recherche en informatique algorithmique en tapant ses bons de commande alors qu'il fait du « secrétariat ». En outre, comme il doit bien falloir 40 ans pour maîtriser  $\text{T}_{\text{E}}\text{X}$ , cela à l'avantage de pouvoir occuper toute une vie de chercheur. Le problème est de savoir si on a le temps d'apprendre  $\text{T}_{\text{E}}\text{X}$  avant qu'il soit désuet... Enfin, cela m'a permis de faire du  $\lambda$ -calcul appliqué, un peu comme Monsieur Jourdain faisait de la prose. C'est possible : on peut programmer en  $\lambda$ -calcul !

Je remercie Bernard Gaulle pour avoir développé un très bon style (`french.sty` [`Gau92`]) pour utiliser  $\text{T}_{\text{E}}\text{X}$  avec notre belle langue. Par son intermédiaire, ainsi que celui des membres du GUT (Groupe des Utilisateurs de  $\text{T}_{\text{E}}\text{X}$ ) concernés par la francisation, j'ai appris beaucoup de chose en ce qui concerne la typographie française.

Je ne remercie pas la personne qui a volé le numéro 161 de « Pour la Science » [`Cor91`]. Merci par contre à Christophe Vedel et surtout son père pour m'en avoir prêté un exemplaire.

Merci à tous ceux qui ont fait les dessins que j'ai repris dans ma thèse, à savoir Bernhard Geiger (repris de la thèse à Monique Teillaud [?]), les personnes qui ont faits les dessins des Data Books de la défunte société MMI, des publicités du RS/6000, du Courier du CNRS, du Monde Diplomatique et des Cahiers du Monde Diplomatique, mais avant tout à Nicolas Paris et Stéphanie Even qui m'ont fait des dessins tout spécialement.

Je remercie aussi tous ceux qui ont développés de manière générale tout le matériel et les programmes que j'ai utilisé dans mon travail (ainsi qu'en dehors de mon

---

11. Mais qu'est-il donc allé faire à Sciences-Po' plutôt que de faire une thèse ?

travail...). En particulier un grand merci à Dieu qui a créé non seulement « tous ceux qui ont développés de manière générale tout le matériel et les programmes que j'ai utilisé dans mon travail (ainsi qu'en dehors de mon travail...) » mais aussi tous les autres et toutes les autres choses ! Dans l'ensemble la thèse de Dieu n'est pas trop mauvaise.

Je ne remercie pas certains « commerciaux » de logiciel, de matériel informatique ou électronique qui, sous prétexte qu'ils avaient à faire à un universitaire sans le sou ou trop surs de leurs « compétences », ont négligé leur travail, pour ne rien dire d'autre.

Par contre d'autres ont été très sympathiques, comme M. Terranova.

Enfin je remercie aussi tous ceux que j'ai oublié de remercier, en espérant qu'ils accepteront de me pardonner de les avoir oubliés. Mais tout ceci est bien subjectif, immanent...

Qu'on me pardonne dans la suite les pieds de pages, les dessins peu sérieux, la présentation, les « ! » et les « ... », voire même de l'humour mal contenu qui m'aurait échappé. J'ai essayé de tout concentrer dans les remerciements<sup>12</sup>, mais on ne sait jamais...

Ronan KERYELL

Paris, le 8 octobre 1993.



---

12. Que certains aurent donc intérêt à avoir sauté. Evidemment, annoncer cela à leur fin tient du sadisme.

# Sommaire

De quelques remerciements

III

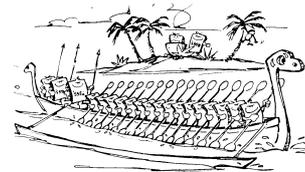
1 Introduction

2



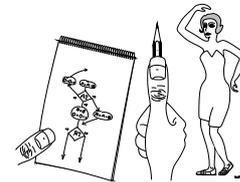
2 Des architectures graphiques au parallélisme

12



3 Le modèle de programmation

42



4 Le langage : POMPC

62



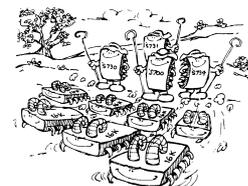
5 Les processeurs élémentaires

92

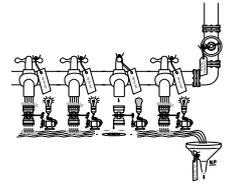


6 Le contrôle de la machine

118



7 Le contrôle de flot SIMD



132

8 La génération du code



160

9 Le réseau d'interconnexion

184

10 Réalisation



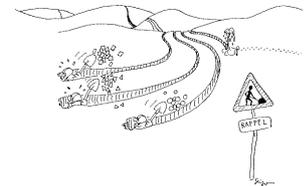
220

11 Divertissement sur la superlinéarité



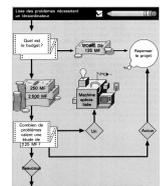
252

12 Vers une machine SPMD ?



264

13 Conclusion



304

Bibliographie



XIV

Index

XLII



Chapitre **1**

**Introduction**



*“Supercomputers, which derive their title from the relative power they possess in any given period of computer life cycle, possess certain qualities which make their creation unique in the general milieu of computational engines. The interaction of technology, architecture, manufacturing, and user demands gives rise to compromises and design of some of these challenges and their resolution [...] is discussed in this paper in an attempt to elevate supercomputer development from the mystique of being an art to the level of a science of synergistic combination of programming, technology, structure, and packaging.”*

[Lin82, page 349]

## 1.1 Le développement des ordinateurs dits « parallèles »

Pour reprendre le vieux proverbe chinois

« *une image vaut dix mille mots* »

on peut dire aujourd’hui

« *une image de simulation numérique en 3 dimensions EST facilement plusieurs milliards de mots (mémoire...)* »

et une station de travail classique ne suffit plus pour afficher rapidement et aider l’utilisateur à exploiter la quantité d’information contenue dans une seule image. Sans parler de la puissance encore plus importante qui est souvent nécessaire pour effectuer la simulation numérique d’un problème, rien que sa visualisation nécessite une puissance de calcul et une capacité mémoire considérable. Ce problème devient critique à un moment où la majorité des gros ordinateurs actuels ne sont plus utilisés à faire des tâches de contrôle ou d’analyse de données comme dans les années 1970 mais à faire de la simulation dans tous les domaines, que ce soient des tâches de conception, de visualisation médicale, de modélisation moléculaire, de mécanique des fluides, etc. [CK91].

L’utilisation d’ordinateurs de plus en plus gros permet de modéliser des problèmes identiques de manière de plus en plus fine en un temps équivalent, ce qui a de quoi séduire même les utilisateurs satisfaits qui estiment que leur ordinateur est suffisamment puissant pour leurs besoins. Ainsi un modèle avec un maillage plus précis permettra de mieux prévoir par exemple les effets de la pollution d’une nappe phréatique, de prévoir le climat à plus long terme et de manière plus précise géographiquement et temporellement ou encore d’augmenter la compréhension de l’effet de serre par exemple<sup>1</sup>.

Ce besoin d’avoir toujours plus de puissance de calcul n’est certes pas nouveau puisque c’est lui qui a fait apparaître les premières machines de calcul et les à fait évoluer vers les ordinateurs de plus en plus performants que nous connaissons<sup>2</sup>.

---

1. Néanmoins, il faut bien remarquer que ce n’est pas la puissance de calcul qui améliorera un modèle qui est faux et qui ne représente pas correctement la réalité. On se gardera donc de présenter l’ordinateur puissant comme le modèle miracle court-circuitant tout travail de modélisation.

2. L’évolution peut même s’entretenir elle-même: « *un ordinateur, quelle que soit sa puissance, trouvera toujours une utilité récurrente dans le fait qu’il servira de toute manière à simuler la génération*

La première approche pour accélérer un ordinateur a été d'améliorer la vitesse de ces composants élémentaires tout en conservant le modèle de fonctionnement associé au nom de VON NEUMANN et inchangé depuis les années 1940 [Bac78]. Dans ce modèle, un ordinateur obéit à des instructions stockées dans une mémoire et agit en conséquence sur des données qui sont elles aussi stockées dans une mémoire. À chaque « cycle », l'ordinateur va chercher une instruction et agit sur une donnée à la fois qu'il peut aller chercher ou stocker en mémoire. La liste des instructions exécutées forme le programme de l'ordinateur qui est intrinsèquement séquentiel. Pour effectuer un calcul, un tel ordinateur est obligé de faire des transferts incessants entre sa mémoire et son processeur. On voit donc apparaître le goulet d'étranglement de la machine entre le processeur et sa mémoire car on ne peut augmenter arbitrairement ce débit à cause des contraintes technologiques. Néanmoins, ce débit suit l'évolution de la technologie et augmente régulièrement, ce qui a permis de concevoir des ordinateurs séquentiels de plus en plus performants.

L'autre approche, basée sur des architectures nouvelles — dans le sens relatif à chaque instant présent puisqu'elles existent depuis longtemps [PHE77] — ou tout au moins différentes, part du principe que de toute manière on veut faire des ordinateurs dont les performances évoluent plus vite que la rapidité des composants élémentaires qui de toute manière se heurteront à des problèmes physiques comme la vitesse de la lumière qui fixe une borne supérieure à la vitesse de propagation d'information. Un moyen efficace de contourner cette limitation est de faire des machines machines parallèles, ainsi appelées car constituées de plusieurs processeurs coopérants au lieu d'un seul gros processeur. Bien que les processeurs soient toujours sensibles à la vitesse de la lumière, on peut essayer de compenser cette limitation par la mise en œuvre de plus de processeurs en conséquence. En première approximation, on peut estimer que plus une machine possède de processeurs et plus elle est potentiellement puissante et ce de manière proportionnelle au nombre de processeurs. Le goulet d'étranglement entre le processeur et la mémoire peut être diminué en parallélisant aussi la mémoire au niveau de chaque processeur.

Pour faire face aux avalanches de données produites par des ordinateurs puissants, il faut en plus de quoi stocker ces données. Et là encore la solution est au parallélisme : on utilise des tableaux de petits disques permettant des capacités et surtout des débits beaucoup plus importants que quelques gros disques, à coût moindre de surcroît.

La fabrication d'ordinateurs parallèles plutôt que d'ordinateurs vectoriels rapides n'est pas gratuite. Outre le problème de vitesse de transmission déjà évoqué, le parallélisme essaye d'amener une solution aux problèmes rencontrés lors de l'élaboration des supercalculateurs et de leurs composants, à savoir :

### **fiabilité :**

- un monoprocesseur très rapide nécessitera des composants à la pointe de la technologie de taille très petite qui le rendront vulnérables à des problèmes tels que les rayonnements ionisants<sup>3</sup> au bruit thermique ( $kT$ ) qui sera, à température ambiante, de l'ordre de l'énergie commutée par les composants vers

---

*suivante.* » Bon mot que je tiens de l'exposé de David MAY à PARLE '92 où il argumentait avec humour le développement de son T9000.

3. Plus ou moins naturels...

l'an 2020 si on extrapole l'évolution de la technologie actuelle, sans parler du comportement très quantique et donc non déterministe d'un composant reposant sur l'état de quelques électrons<sup>4</sup> [Mar92a] ;

- on peut espérer construire un ordinateur parallèle qui pourrait utiliser la redondance de ses éléments de calculs couplée éventuellement à une redondance temporelle [PDGO87] pour continuer à fonctionner globalement même si quelques éléments tombent en panne ou adopter des modèles parallèles intrinsèquement redondants, comme les systèmes neuronaux. Mais la redondance au niveau des composants eux-mêmes est assez inintéressante car elle est équivalente à avoir des composants plus gros ;

**vitesse :** elle intervient sur plusieurs paramètres :

- la vitesse d'un processeur est, entre autres, limitée par la vitesse de propagation des signaux électriques dans les fils<sup>5</sup>. La vitesse est obtenue en rassemblant tous les composants dans un espace minimal pour diminuer les temps de propagation ;
- mais alors survient LE problème capital : la dissipation thermique. Celle-ci empêche de faire des circuits intégrés à la fois très denses et très rapides (comme en AsGa ou en ECL). Plus la vitesse d'horloge du système est rapide et plus les composants dissipent de l'énergie calorifique. Cette concentration en chaleur pose des problèmes techniques devenant rapidement insolubles. Le parallélisme est très intéressant car, même dans le cas où il ne permettrait pas une diminution significative (ce qui est peut probable car en général on peut se contenter de technologies « froides » comme le CMOS consommant beaucoup moins que les technologies bipolaires, même lorsqu'on renormalise les résultats à fréquence égale), il permet une dilution de la puissance dissipée dans la machine ;

**coût :** il est dû à l'utilisation de composants très chers, car ils doivent être les plus rapides du marché, mais aussi aux problèmes de conditionnement : dans les machines vectorielles, une grande partie du coût provient de la technologie nécessaire pour refroidir les composants très rapides. A partir d'un certain nombre de MW/m<sup>3</sup> le problème se rapproche du refroidissement d'un cœur de centrale nucléaire<sup>6</sup>.

Bien entendu, le développement et l'utilisation de machines parallèles ne peut se faire qu'avec un développement des architectures parallèles, terme englobant pour nous aussi bien le matériel que le modèle de programmation, le langage et le système, en harmonie avec le développement des algorithmes et des méthodes de programmation

---

4. Un sujet passionnant et futur serait de créer une branche homéopathique de l'informatique. Si on considère comme certains que nous ne nommerons pas que « le coût d'une machine n'est pas un critère scientifique », on peut alors assister au développement des ordinateurs molaires, puis aux ordinateurs universels, ainsi dénommés car ils sont composés respectivement de  $\mathcal{N} \simeq 6,023 \cdot 10^{23}$  processeurs et de  $10^{80}$  particules, nombre de particules estimées dans l'univers.

5. Vitesse de l'ordre d'un tiers de la vitesse de la lumière. On voit donc qu'une technologie « tout optique » ne permettrait pas d'obtenir plus d'un gain de 3 sur ce point. Mais l'optique possède d'autres avantages.

6. On laissera le soin au lecteur de faire l'analyse du rapprochement entre le CEA et un fabricant de matériel électronique bien connu... Mais le sodium fondu n'a pas encore remplacé les CFC.

parallèle, ou tout au moins des méthodes de parallélisation automatique, sinon les machines parallèles ne seront pas réellement utilisées et resteront un domaine de curiosité.

Il existe un certain nombre de problèmes ouverts quant à l'exécution parallèle d'algorithmes, en particulier en ce qui concerne le placement des calculs sur les processeurs de la machine afin de minimiser les temps de communication, au choix des réseaux de communications selon plusieurs critères, aux problèmes d'ordonnement des calculs et de la gestion de la concurrence, aux problèmes de synchronisation. Autant de problèmes qui posent des questions intéressantes aux chercheurs en informatique mais qui sont souvent sans intérêt pour les autres.

Mais mis à part l'intérêt qu'on peut porter au parallélisme en tant que tel, il existe beaucoup de problèmes qui sont intrinsèquement parallèles, tout particulièrement en physique à travers une discrétisation de l'espace, voire du temps pour la simulation de problèmes ergodiques<sup>7</sup>, la simulation d'un ensemble de particules, etc.

Afin que ces problèmes puissent bénéficier de nouvelles machines parallèles, il faut développer à la fois les algorithmes parallèles bien entendu, mais aussi des modèles de calcul et de programmation parallèle pour penser les problèmes et les penser de manière rationnelle, ainsi que des langages parallèles afin d'exprimer le plus naturellement possible le parallélisme intrinsèque de l'algorithme initial, sans avoir l'esprit bridé par la description selon un modèle de VON NEUMANN du problème [Bac78].

Ainsi certains calculs sont intrinsèquement parallèles (comme par exemple les multiplications de matrices) et lorsqu'on écrit le programme pour une machine séquentielle, on introduit artificiellement de la séquentialité. Il y a une perte d'information au niveau du parallélisme et une surinformation séquentielle qui pourra se traduire par une perte d'efficacité si on veut porter le programme sur différentes machines parallèles. Cela implique un nécessaire développement de l'algorithmique parallèle *à la base* dans des langages parallèles suffisamment puissants pour conserver l'algorithme d'origine.

À cette approche théorique on peut néanmoins opposer facilement deux arguments :

- un langage parallèle trop abstrait, même s'il peut avoir des propriétés mathématiques très intéressantes, peut être très difficile à implanter pour programmer une machine parallèle efficacement car le compilateur, même s'il n'a pas à extraire le parallélisme d'un programme décrit de manière purement séquentielle, doit organiser le parallélisme de manière à utiliser au mieux la machine cible ;
- il est parfois plus simple de décrire des algorithmes de manière modérément parallèle au sein d'un langage impératif séquentiel par exemple. Cela permet d'adapter le langage à un utilisateur qui, s'il arrive à manipuler mentalement des données parallèles, a plus de mal à manipuler des concepts en parallèle<sup>8</sup>.

Alors, pourquoi jusqu'à présent les machines parallèles ne sont-elles utilisées que de manière marginales, cantonnées presque exclusivement aux chercheurs intéressés par le parallélisme ? Parce que tant que les machines parallèles n'auront pas dépassé de plusieurs ordres de grandeur les machines vectorielles classiques, les utilisateurs industriels ne voudront pas investir dans la réécriture de leurs programmes de manière parallèle

---

7. Où on peut faire des simulations identiques sur des données différentes en parallèles.

8. L'utilisateur, même s'il a un cerveau à architecture parallèle, semble raisonner en général séquentiellement sur des concepts...

et préféreront pousser à bout le développement de la technologie exubérante des supercalculateurs vectoriels. Mais il faut bien remarquer que maintenant les ordinateurs vectoriels sont multiprocesseurs et sont de plus en plus parallèles [Mye91] : le parallélisme est un fait, mais seulement de manière récente, même si cela fait déjà 40 ans que l'on dit que les machines séquentielles vectorielles n'ont plus d'avenir face aux besoins croissants des numériciens [Amd67]...

Il faut probablement rajouter à cet état de fait le manque d'information du « milieu vectoriel », manque d'information peut-être entretenu pour défendre les parts de marché protégé. Ainsi dans un article décrivant une machine vectorielle [Che83, page 56] on trouve des exemples qui sont exécutés scalairement sur cette machine vectorielle alors qu'ils peuvent l'être parallèlement grâce à des opérations parallèles préfixes [Kog74b, Cal91].

Aussi, cela commence à changer avec l'apparition d'outils de plus en plus performants, capables de paralléliser automatiquement de plus en plus de programmes écrits de manière séquentielle et une sensibilisation des utilisateurs au parallélisme grâce au « faible » prix du MFLOPS parallèle par rapport au MFLOPS vectoriel [DKMS90] et à la standardisation de langages parallèles, comme FORTRAN 90 par exemple.

Si on considère par exemple le fait qu'une campagne de forage en mer coûte environ  $10^8$  FF, on comprend que les compagnies pétrolières s'intéressent de près aux nouvelles possibilités de simulation permises par l'utilisation de calculateurs parallèles plus puissants.

## 1.2 Les origines du projet POMP

Le thème architecture des ordinateurs aux LIENS (UA 1327) est la continuité du thème architectures graphiques spécialisées de l'ex-LIE (UA 813) qui a évolué vers l'étude d'une machine massivement parallèle plutôt générale mais permettant néanmoins l'animation d'images complexes sur un écran vidéo.

Les motivations du projet POMP de construction d'une station de travail à architecture parallèle sont basées sur les caractéristiques suivantes qui nous semblent nécessaires à la vulgarisation des machines puissantes :

- la machine doit être de petites dimensions physiques, de taille comparable à la taille d'une station de travail habituelle et sa consommation électrique raisonnable afin que la machine puisse se placer sous un bureau de chercheur par exemple ;
- l'augmentation de la puissance est faite par le parallélisme massif plutôt que par le développement de composants rapides mais coûteux ;
- la programmation doit l'emporter sur la spécialisation : ce doit être un ordinateur avec un langage de programmation, un système d'exploitation, capable de s'adapter à de nombreux algorithmes ;
- bien que prévue pour traiter des objets graphiques la spécialisation se limite à l'inclusion d'un mécanisme de transmission de données sur un écran et la machine est plutôt un ordinateur parallèle général.

L'architecture actuelle et les idées que nous présentons ne se sont bien évidemment pas imposées telles quelles au début du projet mais est le résultat de l'interaction entre

l'évolution de la technologie et de nos connaissances du domaine. L'évolution du sujet de thèse montre clairement que le domaine est en pleine effervescence, plein de vitalité, et soumis aux idées nouvelles du milieu et aux nombreux changements technologiques.

A l'origine, au printemps 1987, voire même avant, POMP consistait en la réalisation d'une carte comportant 4096 processeurs 1 bit, puis 1024 processeurs 8 bits, offrant ainsi une puissance de calcul de l'ordre de 500 MIPS et 125 MFLOPS sur une carte. Il s'agissait de « faire mieux qu'une PIXEL-PLANE 4 ». Ces chiffres, quoique ambitieux pour l'époque peuvent paraître bien faible par rapport à l'offre actuelle en matière de machines classiques. C'est surtout le chiffre de la puissance en calculs flottants qui méritait d'être notablement augmenté et qui a justifié le passage à des processeurs 32 bits. L'apparition sur le marché de la machine de chez WAVETRACER qui correspond assez à POMP du printemps 1987 développe bien 1000 MIPS environ mais seulement 10 MFLOPS, à cause de la largeur des processeurs qui est bien trop faible.

Pour avoir un débit mémoire important tout en logeant sur une seule carte, il fallait intégrer processeur et mémoire sur un même circuit intégré. Cette conclusion rendait le projet assez complexe mais cette voie, suivie par le thème de recherche [Dou89], semblait la seule possible.

Au printemps 1989, les schémas de la machine étaient bien définis mais l'intégration sur un même circuit intégré d'une UAL 32 bits et flottante avec 2 Mbits de RAM se heurtait à des problèmes technologiques liés principalement à l'absence de support d'un industriel intéressé.

Notre étude a beaucoup été influencée par le fait qu'on avait accès à une Connection Machine 2, situé à l'ETCA<sup>9</sup> à une époque où les machines parallèles n'étaient pas très répandues. L'étude de cette machine nous a permis de voir ses qualités et ses défauts, de profiter de cette étude tout en essayant d'éviter les écueils de la CM 2.

Réaliser un processeur puissant dans un petit environnement universitaire était une gageure et on s'est aperçu qu'on pouvait peut-être récupérer un processeur classique du commerce afin de l'utiliser en mode SIMD. Le problème de la compilation semblait résolu avec l'expérience acquise sur la CM 2 et le développement d'un compilateur du langage parallèle POMPC pour la CM 2. Un processeur perverti pour fonctionner en SIMD avec la mémoire statique rapide pouvait être utilisé efficacement comme brique de base d'une machine [Ker89, HKMP91], le problème de débit étant résolu par une machine plus grosse, de la mémoire plus rapide ainsi que par la présence de registres dans le processeur dont l'utilisation était rendue possible par une refusion du compilateur.

Le travail nécessaire à la réalisation d'une machine était considérablement amoindri : on pouvait se contenter d'implanter le réseau de communication dans des circuits programmables et réutiliser les compilateurs standards livrés avec le processeur dans notre chaîne de compilation globale.

### 1.3 Plan du mémoire de thèse

Nous présenterons d'abord le sujet qui nous a amené au parallélisme, à savoir les machines graphiques pour la synthèse d'image et l'augmentation de leur puissance. Cela nous permettra d'exposer quelques méthode d'augmentation de puissance à travers le parallélisme dans le chapitre 2. La bonne compacité des machines SIMD en terme de

---

9. Établissement Technique Central de l'Armement.

performance sur des problèmes à parallélisme de donnée issus du concept de *smart memory* associée à une programmation simple nous feront choisir cette voie.

Dans le chapitre 3 nous parlerons d'un modèle de programmation qui nous convient bien pour de nombreux calculs numériques classiques : le modèle de type parallélisme de données, un modèle simple incluant des calculs sur des tableaux typés à l'intérieur d'un modèle impératif classique s'occupant du contrôle de flot et des calculs scalaires. Ce modèle a de bonnes propriétés sémantiques et permet d'exprimer de manière simple le parallélisme habituel que l'on rencontre dans la plupart des applications.

Le chapitre 4 décrit un certain nombre de langages concernés par le modèle de programmation précédent avec en particulier le langage POMPC développé par Nicolas PARIS à partir du langage C pour programmer les machines parallèles, et en particulier POMP, dans une optique parallélisme de données ainsi que SPMD.

Comme il n'y a pas d'ordinateur sans processeur, le chapitre 5 parlera du choix délicat qui consiste à prendre un processeur du commerce pour construire un ordinateur ou bien à concevoir aussi un processeur avec tous les paramètres que l'on peut raisonnablement ajuster pour répondre au mieux à nos besoins tout en conservant une vision économique et technique réaliste.

Le chapitre 6 parle du contrôle scalaire de la machine puisque le modèle de programmation choisi impose que les instructions parallèles soient plongées dans une trame scalaire séquentielle<sup>10</sup>. Nous développons une méthode très simple permettant de factoriser le processeur scalaire et le séquenceur d'instructions parallèles sous la forme d'un couplage VLIW de la machine au niveau processeur scalaire et processeurs parallèles.

En ce qui concerne le contrôle de flot parallèle au sens SIMD du terme, le chapitre 7 expose les méthodes classiquement utilisées ainsi que de nouvelles méthodes de factorisation de l'activité, qui peuvent s'appliquer aussi bien aux machines SIMD que MIMD, et une technique plus spécifique basée sur le contrôle interne du pipeline des processeurs élémentaires pour le SIMD. On y décrira aussi des optimisations possibles dans le cas de blocs parallèles alternatifs terminaux. On présentera une application de ces méthodes à la compilation du contrôle de flot parallèle de POMPC bien entendu.

Le fait d'avoir une machine parallèle pose des problèmes bien sûr de programmation en amont mais aussi des difficultés de génération de code à l'aval. Pour cela, le chapitre 8 aborde différents problèmes qui vont de la génération du code, à savoir comment programmer POMP pour qu'elle reste synchrone, au niveau des PES mais aussi entre les PES et le processeur scalaire, tout en utilisant l'environnement logiciel existant autour du processeur choisi pour la machine grâce à une heuristique travaillant au niveau du graphe de dépendance des instructions de la machine, la prise en compte de la virtualisation, jusqu'au développement pragmatique d'un environnement de débogage et son intégration dans le monde UNIX.

Comme une machine parallèle n'est que bien peu de chose sans un réseau adapté, le chapitre 9 parle des problèmes liés aux réseaux et propose un réseau hybride à vision soit dynamique soit statique adapté aux besoins de notre machine tout en restant le plus simple possible à réaliser, au point de loger dans un circuit reconfigurable du commerce.

Le chapitre 10 décrit plus précisément le prototype que nous avons construit pour

---

10. Séquentielle au moins d'un point de vue sémantique du modèle de programmation qui peut bien entendu avoir un modèle d'exécution différent.

mettre en pratique les concepts exposés précédemment de pragmatisme ainsi que de coût et d'effort minimaux. On y trouve la description depuis l'interface VME pour notre SUN jusqu'au module processeur élémentaire de POMP. Même si une réalisation industrielle de la machine peut nécessiter une révision de certains schémas, cette description permet de se donner une idée de ce que doit contenir la machine pour fonctionner et donner des indications quant au coût et aux performances de la machine.

Dans le chapitre 11 je me suis fait plaisir en parlant de superlinéarité dans le cas précis du SIMD et du VLIW qui contredit un certain nombre de « théorèmes folkloriques » du parallélisme de la même manière que [ACF92]. On trouvera après une présentation succincte des problèmes d'estimation de performances liés au parallélisme un exemple de superlinéarité basé sur le couplage fort de plusieurs unités de calcul permettant une factorisations de certaines parties de programme. Cette exemple va plus loin que les problèmes d'échelle liés au nombre de registres ou à la taille des caches.

Le chapitre 12 présente une évolution future possible du projet tenant compte des tendances actuelles technologiques et des progrès dans le domaine de la compilation pour machines parallèles asynchrones. Il semble raisonnable de s'orienter maintenant vers une machine de type MIMD ou SPMD qui offriront bientôt des densités<sup>11</sup> de MFLOPS/dm<sup>3</sup> atteignant et dépassant même celles des machines SIMD si on a des applications pouvant exploiter efficacement les mémoires cache.

Enfin, après la conclusion sur le projet POMP, on trouvera en annexe la présentation de deux applications développées en POMPC : un programme de simulation de gaz sur réseau hexagonal 2D et un programme de résolution d'équations différentielles elliptiques par une méthode explicite de différences finies multirésolution dans l'annexe ???. On trouvera aussi en annexe ??? une présentation de l'assembleur de la machine pour donner une idée de ce à quoi cela peut ressembler.

## 1.4 But de cet ouvrage

Cet ouvrage essaye de répondre aux objectifs principaux suivants :

- bien entendu être un mémoire de thèse pour ne pas trahir ce qui est écrit sur la couverture ;
- faire le point sur le projet POMP dans son état actuel ;
- être le rapport final de ce projet, même si on manque sûrement encore d'un peu de recul ;
- faire une présentation synthétique du projet, sur la conception de la machine, du modèle de programmation, du langage, des exemples d'application, etc. alors que ceux-ci se sont faits souvent simultanément, voire dans l'esprit d'autres personnes ayant participé au projet comme on aura l'occasion de le mentionner.

Le partage de toutes ces interactions en chapitres pourra paraître arbitraire au niveau de l'historique du projet et présentera un graphe de dépendance des idées possédant des cycles. Les chapitre sont amenés de la manière où, maintenant avec un tout petit peu plus de recul, on voit la méthodologie de conception d'une machine...

---

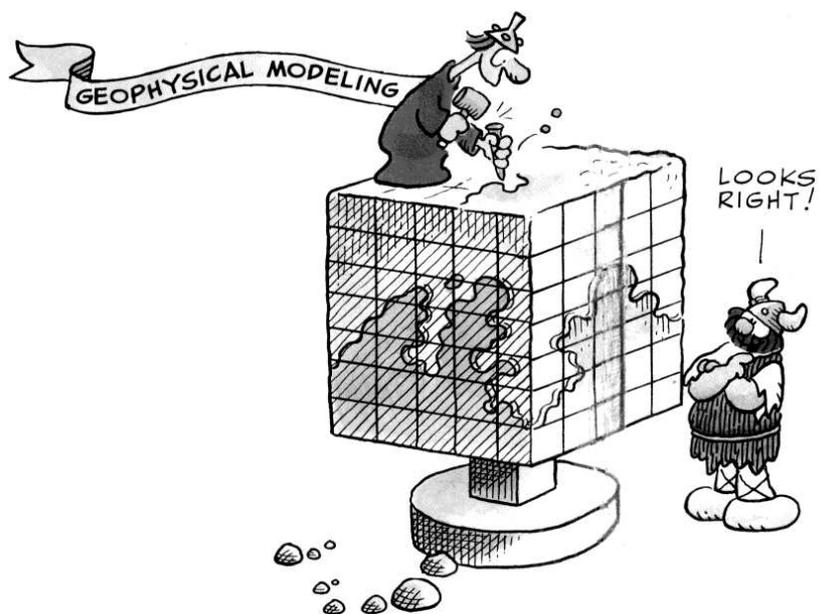
11. On trouve mention de la notion de MIPS/ft<sup>3</sup> ainsi que par watt et par livres dans [ABT82].

- servir de support à toute personne intéressée par un certain aspect des architectures — terme pris encore au sens général que nous avons déjà vu — parallèles et des langages à parallélisme de données associés ;
- ne pas trop ennuyer ni le lecteur ni le jury.

J'espère néanmoins qu'en voulant ainsi donner plusieurs objectifs à ce rapport de thèse je n'aurai pas échoué sur tous les points ci-dessus.

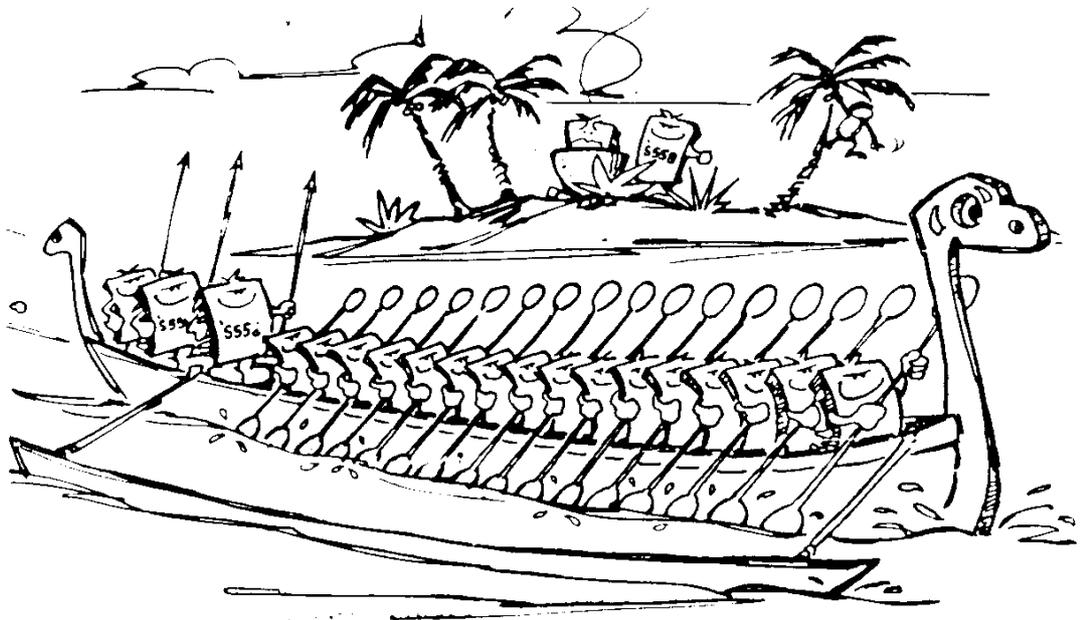
La bibliographie à la fin de la thèse n'est pas annotée mais est indexée, ce qui à mon goût est mieux lorsqu'on veut en extraire de l'information sur un sujet en particulier. Les références croisées ont été rajoutées à la bibliographie pour aller voir dans le texte à quels endroits les citations sont faites et retrouver ainsi leur contexte. Evidemment, elle aurait pu être aussi annotée mais cela n'a pas été fait pour économiser quelques arbres. L'index commun en fin d'ouvrage porte donc d'une part sur la thèse et d'autre part sur la bibliographie.

## 1.5 Typographie du mémoire de thèse



Chapitre **2**

# Des architectures graphiques au parallélisme



**D**ANS ce chapitre nous présentons les éléments qui nous ont amenés à développer une machine parallèle plutôt générale après s'être intéressé aux *ordinateurs graphiques*. Comme nous allons le voir, mis à part la présence d'une sortie graphique haut débit pour afficher les images, les ordinateurs graphiques ont besoin d'une puissance de calcul importante, d'un débit mémoire en conséquence pour alimenter la mémoire d'image et, somme toute, ont des caractéristiques assez proches des supercalculateurs.

Ce chapitre présentera donc en première partie un survol de quelques architectures graphiques significatives en montrant que beaucoup font appel au parallélisme pour obtenir des performances élevées. En deuxième partie, on exposera les principaux types d'architectures parallèles, avec enfin une discussion sur le grain du parallélisme, à savoir si on doit utiliser des processeurs de puissance faible ou importante, et le couplage entre processeurs, qui peuvent soit communiquer par messages, soit partager une mémoire commune.

## 2.1 Les machines graphiques

L'équipe architecture du LIENS avait depuis longtemps des projets de stations de travail dotées de bonnes capacités d'affichage graphique. Après les stations de travail THÉMIS<sup>1</sup> qui étaient basées sur le processeur 6800 puis 6809 et les circuits 9364 et 9365 développés localement et qui ont été commercialisés par EFCIS, il est clairement apparu que ces architectures avaient un goulet d'étranglement qui empêchait le développement d'applications plus puissantes : le débit entre l'opérateur d'affichage et la mémoire d'affichage.

### 2.1.1 La problématique

Mais étudions d'abord en quoi consiste les applications graphiques classiques. Dans notre développement, nous sommes partis de la synthèse d'image. Schématiquement, cette dernière peut être découpée en étapes successives qui vont des scènes décrites par l'utilisateur jusqu'à l'affichage sur un écran de télévision des images produites (figure 2.1).

L'utilisateur crée une base de données graphique où il décrit par exemple la scène qu'il veut visualiser. Celle-ci est traitée par un processeur de liste qui la traduit en entités géométriques que la machine saura visualiser. On peut considérer qu'à ce niveau un processeur classique tel que ceux qui équipent les stations de travail convient tout à fait à cette tâche.

Un processeur géométrique effectue les translations, les rotations, les affinités, etc. nécessaires pour les mettre à leur positions dans l'espace. Si on effectue les calculs dans un espace à 3 dimensions en coordonnées homogènes, on peut se ramener à faire ces transformations par des multiplications de matrices  $4 \times 4$  (dont certains coefficients sont néanmoins toujours nuls) qui se font très bien avec une architecture pipelinée ou systolique par exemple.

---

1. Bien entendu, cela fait sourire maintenant lorsqu'on constate qu'on pouvait considérer des ordinateurs à base de 6800 comme des stations de travail. Mais, de même, dans quelques temps, la machine décrite dans cet ouvrage paraîtra bien faible en performances...

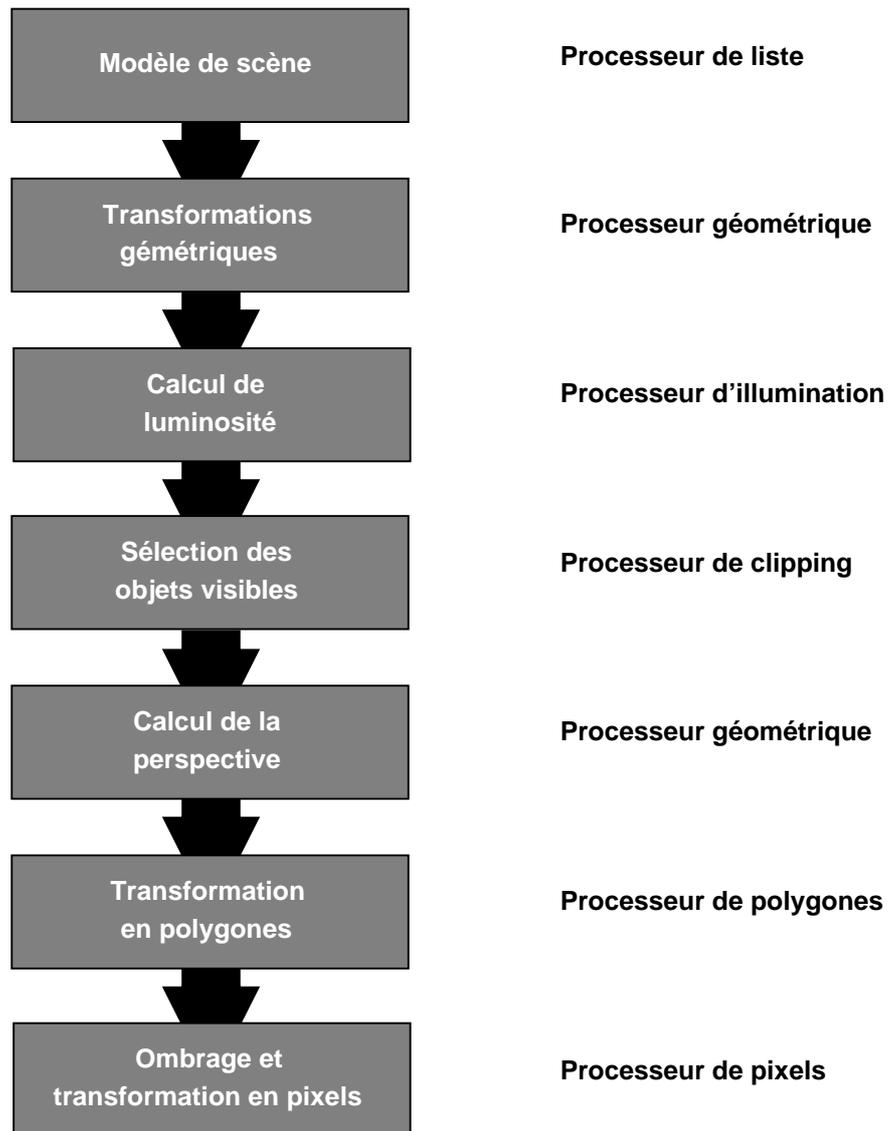


FIG. 2.1 - Schématisation des étapes entrant dans la synthèse d'une image.

Ensuite un modèle d'illumination tel que la radiosité, puisque c'est une méthode bien maîtrisée localement [SP89], est appliqué à tous les éléments de la scène afin de calculer leur couleur et leur luminosité.

Les objets contenus dans la zone de visualisation sont sélectionnés par un mécanisme de boîtes englobantes par exemple. Cela peut se faire grâce à une batterie de comparateurs pour chaque face de la boîte entourant l'espace de visualisation qui ne conservent que les objets possédant au moins un point dans l'espace affiché.

Ces objets géométriques sont placés à leur position dans l'espace virtuel de visualisation grâce au calcul de perspective qu'effectue le processeur géométrique. En fait il s'agit d'opérations comparables à celles effectuées par le processeur géométrique déjà utilisé ci-dessus.

On peut alors découper les objets en facettes ou polygones, plus simples à manipuler

et afficher avec du matériel spécialisé que les objets initiaux plus compliqués comme des sphères par exemple.

Enfin, il reste à transformer ces polygones en *pixels* [GGSS89], points lumineux élémentaires affichés à l'écran. Cela peut se faire avec un ombrage, interpolation sur la couleur des sommets (modèle de GOURAUD) et avec l'intervention de la normale à la facette qui est pareillement interpolée (modèle de PHONG). Cette opération peut être effectuée par un processeur de pixels.

On ne considère que les écrans à affichage ligne par ligne classique (de type télévision) car il sont à la fois économiques à cause de leur diffusion massive et parce que la vitesse de rafraîchissement de l'affichage est indépendante de la vitesse d'écriture dans la mémoire d'image, ce qui permet de diminuer considérablement les difficultés de conception des systèmes graphiques. Le fait que la mémoire d'affichage soit de taille plus importante qu'avec le balayage cavalier et donc plus lente à remplir est contourné en ayant 2 mémoires d'images (*double buffering*) permettant ainsi de séparer encore plus les 2 opérations : on continue d'afficher l'image précédente pendant qu'on construit la suivante que l'on pourra afficher par un échange des 2 mémoires de manière instantanée. Enfin, seule une méthode à écran à balayage ligne permet d'avoir à coût raisonnable des objets pleins, alors que le balayage cavalier est plus réservé aux objets en fil de fer.

Toute cette partition est très schématique et souvent il y a un mélange des étapes exposées. Par exemple le calcul des facteurs de forme nécessitera pour avoir une solution plus simple<sup>2</sup> d'avoir préalablement découpé la scène en polygones impliquant une inversion de certaines des étapes écrites.

Pour donner un ordre d'idée, faire de la synthèse d'images réalistes en temps réel nécessite l'affichage de plusieurs millions de polygones par seconde, chaque polygone étant composé en moyenne de l'ordre d'une centaine de pixels. La transformation géométrique d'un petit vecteur 3D et son affichage à l'écran nécessite 59 opérations flottantes et entières [KV90]. L'affichage d'un quadrilatère 3D nécessite de l'ordre de 400 opérations flottantes [AJ88]. De bonnes performances nécessitent donc de disposer de plus d'1 GFLOPS, c'est-à-dire 1 milliard d'opérations flottantes par seconde, ce qui est (encore) loin d'être négligeable.

Mais avoir encore plus de performances permet de faire des calculs redondants pour améliorer le réalisme de l'image, comme pour rajouter l'impression de mouvement et la notion de profondeur de champ [HA90] propres à la photographie et au cinéma, ou faire du suréchantillonnage de l'image pour en améliorer le rendu final (*anti-aliasing* qui se traduit dans ce cas particulier par anticrénelage) [Bar90]. Il n'y a donc pas de problème de ce côté là : quelle que soit la puissance de la machine, elle sera toujours utilisée pour améliorer la qualité du rendu final des images.

### 2.1.2 Quelques solutions

La présentation précédente laisse à penser que les meilleures performances seront obtenues avec du matériel spécialisé, c'est-à-dire une fonction spécifique par tâche à effectuer. Evidemment, il faut aussi considérer le coût de la solution par rapport aux performances désirées, ce qui explique que l'on aille des solutions où tout est fait avec des composants spécialisés, très chères en général mais performantes (typiquement les

---

2. Évidemment moins exacte aussi, mais l'œil y est assez peu sensible.

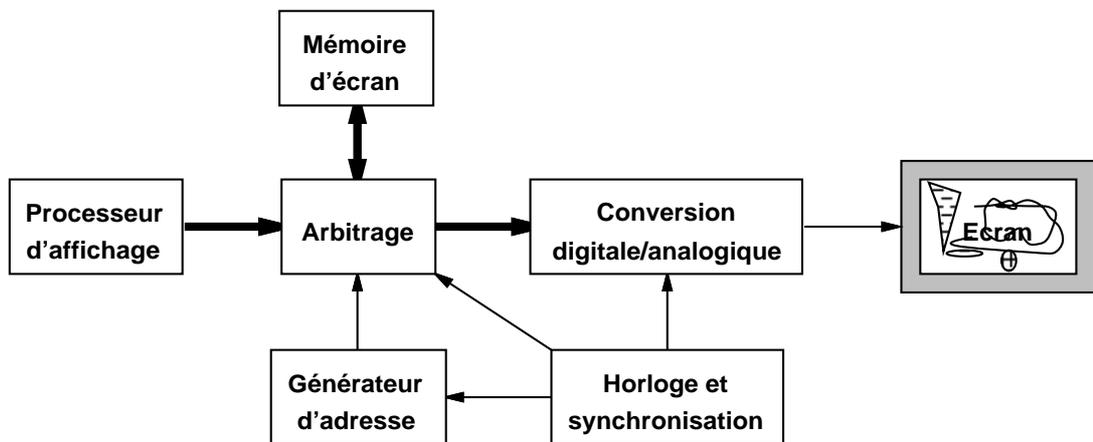


FIG. 2.2 - *Partie visualisation d'un ordinateur sans matériel spécialisé pour faire du graphique.*

simulateurs de vol), à une solution où tout est fait sur un processeur bon marché par logiciel, peu chère mais aussi peu performante.

Dans la suite, un certain nombre de notions sont utilisées sans avoir été expliquées, en particulier en ce qui concerne les formes de parallélisme. Elles le seront dans les sections suivantes.

### 2.1.2.1 Un matériel très réduit

La première approche, plutôt bas de gamme par rapport aux suivantes, est de dire qu'on peut tout laisser faire au processeur d'un ordinateur général: c'est ce qui est fait souvent dans les « ordinateurs personnels » ou les stations de travail d'entrée de gamme. Néanmoins, comme la puissance de ces machines augmente constamment, les performances graphiques font de même, même si elle ne répondent pas à toutes les attentes.

Dans ce cas, la visualisation est simplement un système qui lit la valeur des pixels dans la mémoire centrale de l'ordinateur et les envoie sur un écran. Le matériel nécessaire est minimal : génération des signaux vidéo à partir des données fournies par un système d'accès directe à la mémoire (DMA).

Comme ce mode d'utilisation est courant en informatique, des mémoires spécialisées, les Vidéo-RAMs, ont été développées en rajoutant une (ou plusieurs) sortie série permettant un affichage presque indépendant de l'utilisation « normale » de la mémoire par le processeur. La seule contrainte étant d'initialiser régulièrement un cycle d'envoi d'une ligne de mémoire vers la partie vidéo, cette dernière se chargeant ensuite d'envoyer de manière régulière et rapide les pixels à l'écran.

Si on constate que le problème du débit mémoire entre la mémoire d'écran et la partie vidéo est résolu par le mécanisme des Vidéo-RAM par exemple, il n'en est pas de même entre l'unité qui trace les pixels et la mémoire d'écran où ils sont écrits pour être visualisés. En effet, les écritures dans la mémoire d'écran sont réparties de manière aléatoire et il n'est pas possible d'exploiter efficacement une cohérence des accès. À supposer qu'on ne puisse écrire en mémoire qu'un pixel à chaque cycle, que l'on veut

afficher  $10^7$  de polygones par seconde, composés en moyenne de 100 pixels, il faut avoir une mémoire d'1 ns de temps de cycle, soit un facteur 100 en dessous des valeurs courantes.

Il y a donc clairement un goulet d'étranglement entre le processeur et la mémoire et on retrouve là une caractéristique classique des machines de VON NEUMANN : puisque les données sont en mémoire ainsi que les instructions, un processeur ne peut pas aller plus vite que sa mémoire [Bac78].

Il n'empêche que cette approche de visualisation est très répandue et on peut trouver deux approches intéressantes car minimalistes, où il n'y a même pas de DMA supplémentaire : on utilise le processeur pour faire le travail. Il y a :

- l'ALTO, développé à XEROX PARC<sup>3</sup> [?]. Tout y était organisé autour du processeur de la machine. Pour éliminer les temps d'arbitrage de bus multiples c'est le processeur qui était partagé et gérait chaque bus d'entrée-sortie et celui de la mémoire. Ainsi la micro-tâche d'affichage était réveillée régulièrement pour remplir une file d'attente de pixels à afficher depuis la mémoire.

En fait on peut dire que cette approche est minimaliste autour du processeur. Elle ne l'est pas du tout dans le processeur : processeur complexe micro-interruptible et rapide pour gérer toutes les fonctions d'entrées-sorties, principalement des transferts de données. Clairement le processeur était un haut de gamme pour l'époque. Plutôt que de rajouter du matériel, on rajoutait des micro-tâches, dans la mesure des performances du processeur.

- à l'extrême inverse dans le dénuement, le Zx81, qui utilisait le registre de rafraîchissement de mémoire de son processeur, un Z80, pour faire du transfert de pixels vers la partie vidéo. Le processeur ne pouvait faire des calculs que lorsqu'il n'affichait rien, c'est à dire que pendant les retours de lignes et de trame. Evidemment, cela ralentissait énormément les programmes mais n'était pas trop gênant dans la mesure où le marché visé était le grand public.

Néanmoins ces approches existent encore et leurs performances augmentent avec le développement de processeurs standard comme le i860 [INT89a] ou le MC88100 [?] incluant des opérations permettant de faire de la génération de pixels à partir de polygones telle que *z-buffer* et de l'ombrage de GOURAUD.

### 2.1.2.2 Les processeurs spécialisés de type BitBLT

Une autre utilisation, différente de la synthèse d'image mais néanmoins à prendre en considération dans un ordinateur moderne, est la gestion de fenêtres, chaque fenêtre pouvant contenir d'ailleurs une image. En effet, outre la transformation de polygones en pixels, un écran graphique est souvent divisé en fenêtre. La manipulation de ces fenêtres, telle que leur mouvement à l'aide d'une souris revient à la déplacer en mémoire d'écran et donc à faire du transfert de blocs de mémoire.

Pour saturer la mémoire d'affichage graphique par des écritures de pixels ou déplacer des blocs de pixels, il faut avoir en général des processeurs performants pour ce genre d'opération qui sont de type *Bit Bloc Transfer* (BITBLT) [Kor85, Par89, ?].

---

3. Palo Alto Research Center

De même qu'afficher une image dans une fenêtre revient à recopier l'image ou un morceau de l'image visible, à l'endroit de l'écran où elle doit apparaître, afficher du texte consiste à copier l'image de chaque caractère de la fonte correspondante dans la mémoire d'écran là où il doit être affiché. Il s'agit là encore de BITBLT.

Une fois qu'on a ce type de processeur, on est capable de saturer la mémoire d'écran et il faut considérer de nouvelles méthodes pour augmenter sa bande passante. Une méthode classique en informatique dans ce cas est de faire appel à de la mémoire banquée : plutôt que d'utiliser 1 mémoire, on va utiliser  $N$  « bancs » de capacité divisée par  $N$ . Si on s'arrange par exemple pour allouer la mémoire d'écran par colonne et qu'on trace une ligne horizontale de  $N$  pixels, comme chaque pixel ira dans un banc différent cela pourra se faire en parallèle et donc la bande passante sera multipliée par  $N$ .

Afin d'exploiter encore plus le parallélisme, on peut dédier un processeur de type BITBLT par banc [Dou89, Par89]. Ce type de systèmes de visualisation est en fait une utilisation particulière du système de mémoire parallèle capable de faire des accès et des transferts par blocs décrit dans [Sto70].

### 2.1.2.3 Iris

On peut trouver un bon exemple de l'intégration à l'extrême de la chaîne d'affichage graphique de la figure 2.1 et des algorithmes associés dans les stations de travail IRIS [AJ88] dont les concepts se retrouvent dans l'utilisation d'un pipeline géométrique [Cla82] et de *smart memory* [HH80], deux concepts parallèles différents.

Dans cette machine on retrouve tous les styles de parallélisme (figure 2.3) :

- la puissance amont permettant de faire du calcul scientifique, des simulations, des modèles d'illumination évolués, etc. est fournie par une machine MIMD à mémoire partagée basée sur des processeurs de type RISC ;
- les ordres graphiques sont récupérés sur le bus rapide global de la machine par le système de calcul géométrique, *Geometry Engine* (GE) [Cla82], composé d'un pipeline de 5 processeurs flottants microprogrammés spécialisés ;
- les polygones sont ensuite décomposés en trapèzes à bases verticales par le *Polygon Processor* (PP) suivi d'un processeur qui calcule aussi les pentes des côtés ;
- les valeurs intrinsèques le long de ces côtés sont interpolées par les *Edge Processors* qui découpent les trapèzes en *spans*<sup>4</sup> petits segments verticaux, de manière SIMD ;
- ceux-ci sont découpés en pixels par les *Span Processor* (SP) de manière SIMD ;
- enfin les *Image Engines* (IE) sont chargés d'écrire de manière SIMD dans les bancs de mémoire d'affichage avec gestion du *z-buffer*. C'est la *smart memory* : il y a un couplage fort entre les processeurs qui écrivent les pixels et les mémoires d'écran mais faible entre les processeurs. On peut voir chaque ensemble processeur mémoire comme une mémoire intelligente et en tout cas l'ensemble des processeurs pixels comme une machine SIMD à mémoire distribuée [HH80].

---

4. Contrairement à ce que l'on pourrait penser, il s'agit là du nom anglais et non de l'homonyme breton désignant l'ustensile de cuisine ayant la même forme mais servant à retourner les crêpes...

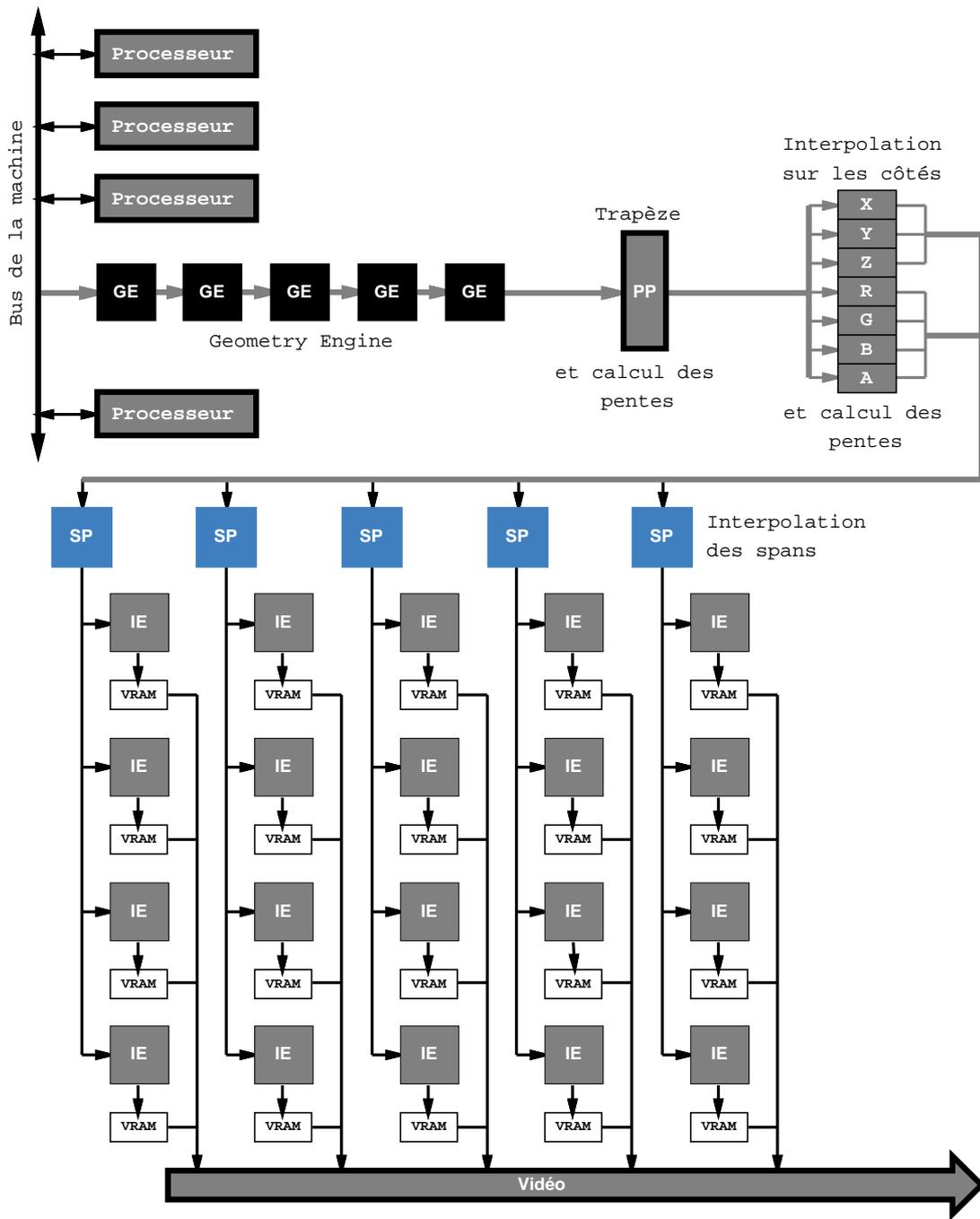


FIG. 2.3 - *Synoptique de la station de travail IRIS.*

On constate qu'il y a autant de processeurs dédiés que de tâches à effectuer. Cela rend la conception de la machine complexe mais permet d'avoir de très bonnes performances à coût moyen. C'est payé par une spécialisation poussée qui empêche toute modification importante des algorithmes de rendu graphique.

### 2.1.2.4 La machine Pixel-Plane

L'étape ultime de cette répartition par banc est d'avoir un banc et un processeur par pixel! C'est ce concept original<sup>5</sup> développant la notion de *smart memory* qui est le fondement des machines PIXEL-PLANE [FP81]. Ainsi en un cycle d'horloge de la machine on peut faire une opération sur tous les points de l'écran.

L'idée de base est simple : on envoie à chaque processeur la description d'un polygone fournie par un pipeline graphique classique en amont qui a déjà fait toutes les transformations géométriques. Chaque processeur se pose la question de savoir si le pixel qu'il contrôle fait partie ou non du polygone envoyé. Si oui, il allume son pixel à la couleur correspondante. On a donc affaire à du parallélisme de type SIMD (voir la section 2.2.1).

Chaque polygone 2D est décrit par une intersection de demi-plans dont on envoie la description des droites frontières sous forme de fonctions affines canoniques :

$$F(x, y) = Ax + By + C$$

Il suffit d'allumer les pixels où toutes les fonctions affines sont positives et le tour est joué.

On peut réaliser aussi le *z-buffer* et l'ombrage selon le même principe en mettant d'une part la « profondeur » et la couleur du polygone sous une forme affine de  $(x, y)$  au niveau du processeur géométrique amont.

Le matériel permettant de faire ces opérations est décrit sur la figure 2.4. Le calcul de la fonction affine est découpé en deux parties,  $Ax + C'$  et  $By + C''$  qui peuvent être calculées séparément. Chaque processeur à la position  $(x, y)$  calcule la fonction  $F$  en additionnant les 2 fonctions partielles fournies orthogonalement comme indiqué sur la figure. Les multiplieurs en  $x$  et  $y$  sont des multiplieurs bit-série qui fournissent toutes les valeurs des fonctions affines pour les entiers variant de 0 à  $\ell - 1$ , où  $\ell$  est le nombre de processeurs sur une ligne ou une colonne du circuit. Le principe est basé sur la réalisation d'un analyseur différentiel numérique<sup>6</sup> sous forme d'opérations préfixes parallèles par doublage récursif [KS73] (voir aussi § 3.6.5) pour calculer simultanément les  $\ell$  valeurs successives.

A ma connaissance, la PIXEL-PLANE 4 est la machine parallèle construite possédant le plus de processeurs : 256K (262144) qui contrôlent donc un écran de  $512 \times 512$  pixels.

Le problème est que comme les additions sont faites en bit-série par des additionneurs 1 bit dans les processeurs, le temps de calcul des fonctions affines est proportionnel au nombre de bits décrivant celles-ci et donc à la précision de calculs. L'idéal serait par conséquent d'avoir du parallélisme à « gros grain » tout en conservant la même intégration... Malheureusement cela semble difficile à cause des contraintes technologiques.

L'intérêt de système de *smart memory* est que quelle que soit la taille du polygone, le débit en polygones ne varie pas, alors qu'en général les machines graphiques sont limitées par le débit en pixels. C'est normal puisqu'on a choisi ici une bande passante mémoire maximale, à savoir que tous les pixels peuvent être écrits en 1 pas de calcul (qui correspond à plusieurs cycle de processeurs puisque les opérations sont faites sur des processeurs 1 bit). On a optimisé le cas pire : être capable d'afficher à la même vitesse des polygones recouvrant totalement l'écran.

5. Ce qui explique que cela va être plus particulièrement décrit...

6. Le même principe en particulier que dans la machine analytique de BABBAGE un siècle plus tôt.

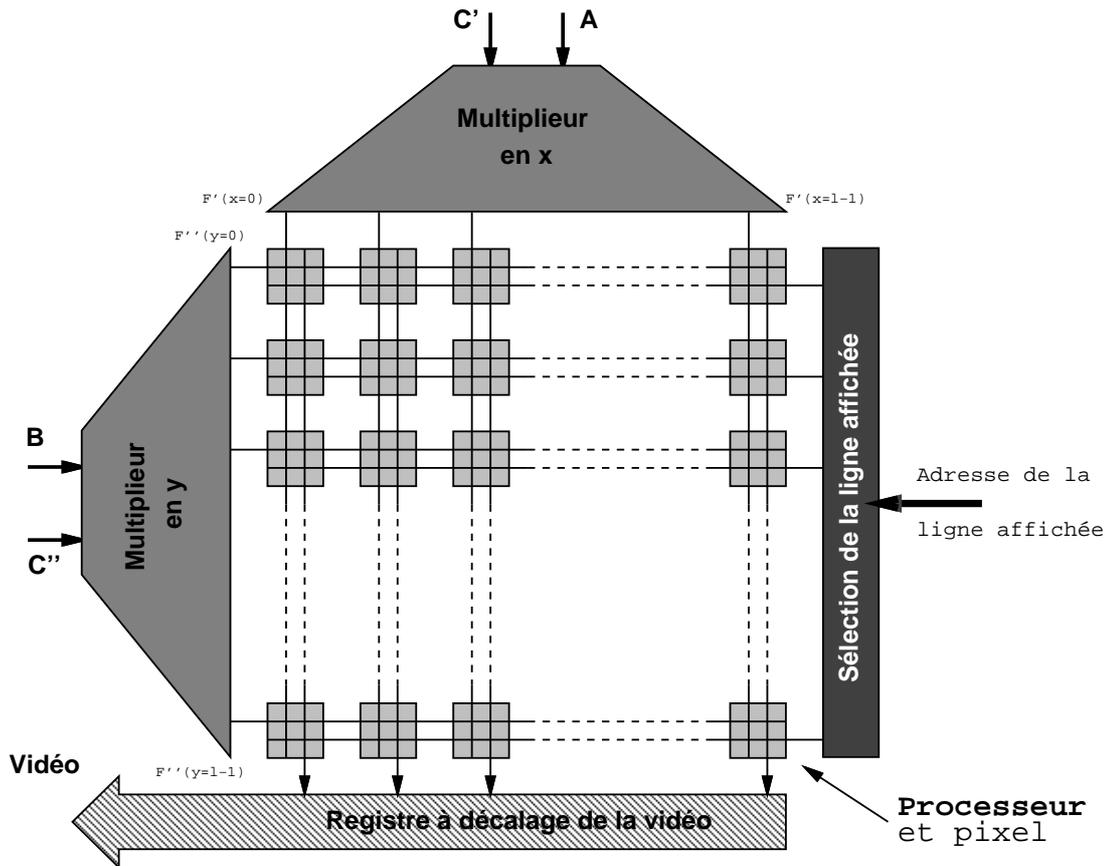


FIG. 2.4 - *Synoptique de la gestion des pixels dans les machines PIXEL-PLANE.*

On voit donc apparaître un autre goulet d'étranglement : à un moment donné il n'y a qu'un polygone traité pour tout l'écran, même si celui-là ne représente qu'une toute petite partie de l'image et ne concerne donc qu'une faible proportion des processeurs.

Mais alors, cela veut dire que cette bande passante est gâchée en moyenne : il est peut-être dommage d'avoir autant de processeurs 1 bit étant donné qu'en moyenne il n'y en a pas beaucoup qui travaillent, *ie* qui allument un pixel. Un parallélisme à gros grain où chaque processeur tracerait seulement les points appartenant à un polygone serait probablement plus dense.

Ce problème a été résolu en grande partie sur la PIXEL-PLANE 5 [FPE<sup>+</sup>89] : la machine a été divisée en blocs de processeurs indépendants. Chaque bloc n'est plus associé à une partie d'écran mais est alloué dynamiquement à une partie de l'écran ( $128 \times 128$  pixels) où un polygone doit être affiché. On peut alors tracer simultanément jusqu'à<sup>7</sup> autant de polygones qu'on a de blocs de processeurs, étant donné que certains polygones peuvent être à la frontière de plusieurs blocs. Les blocs d'images sont chargés depuis la mémoire d'écran composée de Vidéo-RAM de manière pipelinée par rapport aux calculs : pendant qu'on calcule un polygone on écrit le bloc précédent et on lit le bloc où doit être tracé le polygone suivant.

7. Étant donné que certains polygones peuvent être traités par plusieurs blocs lorsqu'ils sont larges que la taille d'un bloc ou à la limite de plusieurs blocs, le « jusqu'à » s'impose.

Dans cette version, les fonctionnalités se rapprochent de la machine IRIS si ce n'est que la distribution et le grain des processeurs d'image sont différents.

Afin d'alimenter la machine en polygones, une machine parallèle composée de 32 i860 en anneau est chargée de faire les calculs de polygones à partir d'une base de donnée en PHIGS+. Sur ce réseau sont placés aussi les blocs de calcul des pixels.

Une amélioration notable est d'avoir remplacé l'arbre d'évaluation d'expressions linéaires par un évaluateur d'expressions quadratique de type :

$$F(x, y) = Ax^2 + Bxy + Cy^2 + Dx + Ex$$

qui permet de faire de l'ombrage de type PHONG rapide [BW86], de faire de l'ombrage directement sur des sphères et des textures procédurales, etc.

Pour ce faire, l'architecture est modifiée par rapport à celle présentée sur la figure 2.4 : il n'y a plus 2 arbres orthogonaux calculant les termes en  $x$  et en  $y$  mais 2 arbres calculant les termes en  $x$  et  $x^2$  alimentant  $\ell$  arbres pour les termes en  $y$  et  $xy$  et  $\ell$  arbres pour les termes en  $y^2$ . Un arbre supplémentaire calcule le produit partiel  $Bx$  [GF86]. Le principe reste donc basé sur une réalisation en arbre d'un analyseur différentiel numérique.

Un point intéressant est qu'on peut utiliser un bloc de processeurs pour calculer les facteurs de formes plus précis d'une scène en projetant tous les polygones sur des approximations quadratiques de sphères en vue de l'afficher avec le modèle de radiosité.

On trouve une approche similaire dans la machine IMOGENE où les processeurs sont en plus attachés aux objets graphiques plutôt qu'aux pixels et ceux-ci sont capables de manipuler et tracer des coniques en 3D [Ata89, Lep89, CKM91].

Mais sans aller jusqu'à une solution extrême comme la PIXEL-PLANE, il existe des architectures intermédiaires basées sur des processeurs de 16 pixels [ABM88] ou bien qui tendent à rajouter de plus en plus d'« intelligence » aux mémoires d'écran de type Vidéo-RAM qui, outre le fait d'avoir plusieurs ports série permettant de faire un *z-buffer* rapide en plus de l'affichage, possèdent des masques d'écriture et de couleur pour pouvoir écrire par blocs de couleur dans la mémoire [Mic91, pages 3-1-3-259]. Il est prévisible que ce type de mémoire aille en se développant de plus en plus jusqu'à intégrer un processeur (graphique) complet afin d'augmenter la bande passante avec la mémoire à plus faible coût.

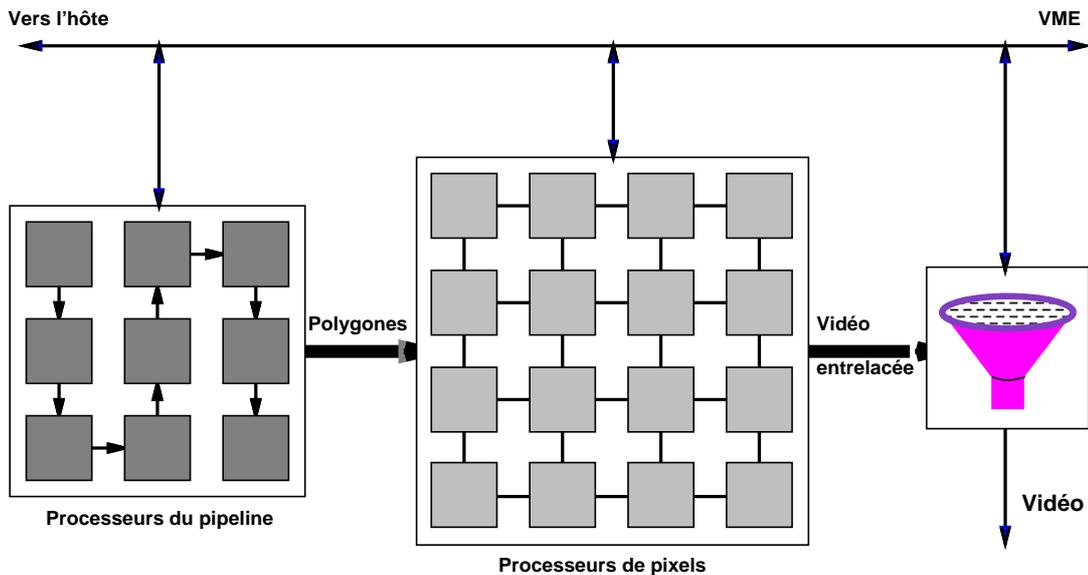
### 2.1.2.5 La Pixel Machine

Un certain nombre de machines parallèles plus classiques ont été proposées. En particulier la PIXEL MACHINE a l'intérêt de ne pas être spécialisée et permet donc de s'adapter à beaucoup d'algorithmes graphiques, qu'ils soient de bas niveau (polygones) ou de haut niveau (lancer de rayon) [PH89].

Les auteurs ont bien séparé la partie « pipeline » graphique, plutôt chargées de toutes les transformations géométriques, de la partie transformations de polygones en pixels (figure 2.5).

Chaque partie est basée sur des processeurs de traitement du signal (DSP) de type DSP32 qui exécutent chacun leur programmes. On a donc un ordinateur parallèle MIMD (voir la section 2.2.1).

Chaque processeur de la partie pixel possède une partie de la mémoire d'écran qui est répartie de manière très éclatée : un bloc de pixels de l'écran de la taille du bloc des

FIG. 2.5 - *Synoptique de la PIXEL MACHINE.*

processeurs possède un pixel sur chaque processeur, ce qui permet de faire simplement un « entonnoir » capable d'envoyer une image à l'écran sans passer par une mémoire d'écran intermédiaire externe.

Il semble y avoir quelques inconvénients au système :

- l'entrelacement des pixels de l'écran sur les processeurs se prête assez mal aux calculs par bloc qu'on a l'habitude de faire ;
- les processeurs composant le pipeline pourraient très bien fonctionner en parallèles plutôt que de fonctionner en mode pipeliné, c'est à dire que chaque processeur fait une partie du travail à la chaîne avant de le passer à son voisin. Un fonctionnement parallèle éviterait ce temps de transfert entre chaque processeur ;
- et par conséquent pourquoi avoir subdivisé la machine en 2 parties architecturales, d'autant plus qu'elles sont composées des mêmes processeurs ? Il suffirait de faire du pipeline logiciel et donc de faire du multiplexage temporel sur les 2 phases de l'algorithme, ce qui aurait en plus comme effet bénéfique *de facto* la répartition de charge entre les 2 parties.

Pourquoi alors ne pas avoir une machine parallèle plus générale, en utilisant des processeurs plus puissants plutôt que de concevoir une machine faite de plusieurs parties distinctes à étudier ?

## 2.2 Les machines parallèles

En fait, il faut surtout éviter d'avoir à construire une machine aussi souvent qu'un nouvel algorithme graphique est créé, ce qui peut arriver souvent dans un domaine aussi créatif que la synthèse d'image<sup>8</sup>. Les machines spécialisées permettent effectivement

8. Notons que cela n'est pas le cas pour des applications comme la CAO mécanique, par exemple.

des rapport performance sur prix considérable mais l'adaptabilité est nulle : il faut reconcevoir le système, les circuits intégrés, le microcode, les bibliothèques graphiques, etc. Il faut nuancer le rapport performance sur prix en considérant qu'il est intéressant seulement à partir d'une certaine quantité de machines produites qui n'est pas toujours atteinte pour des machines haut de gamme.

En cela, le fait d'avoir une machine générale *programmable* permet de s'adapter efficacement à une grande classe d'algorithme<sup>9</sup>. Un argument qui va aussi dans ce sens est que dans une application graphique on constate souvent le paradoxe suivant : la partie affichage est finalement peu importante par rapport à la partie calcul amont [Mat88] et il est utopique de proposer une machine graphique sans suffisamment de puissance de calcul. Cela va dans le sens de l'approche RISC : autant accélérer les parties qui prennent le plus de temps.

Il faut compenser ce manque de spécialisation par un accroissement de la puissance de calcul et du débit mémoire de la machine. Nous pensons que cela peut être fait grâce à l'utilisation du parallélisme. L'utilisation de machines multiprocesseurs permet d'augmenter les performances tout en séparant bien les aspects algorithmiques du choix des processeurs, de l'augmentation de leur performance et de leur architecture, et ne pas avoir à microcoder une application seulement après avoir du matériel qui fonctionne.

Il faut à la fois une puissance de calcul en nombres flottants très importante pour faire tous les calculs d'illumination et les transformations géométriques, et une puissance de calcul très importante en nombres entiers (y compris les opérations booléennes) pour faire la génération finale des pixels.

Pourquoi ne pas faire un pipeline logiciel? Comme on l'a déjà vu, cela aura comme avantage de faire de l'équilibrage de charge implicite entre les différentes phases. Il suffit de décomposer le problème en phases de calculs qu'on exécute l'une après l'autre, telles que :

- une partie application : simulation physique, gros calculs ou synthèse d'image ;
- une partie visualisation, contenue par exemple dans une bibliothèque graphique s'occupant :
  - de la transformation des repères ;
  - de la sélection de polygones visibles ;
  - de la transformation en pixels.

Par exemple, un processeur comme le 88110 [Cel91] est capable de faire les transformations géométriques à 50 MFLOPS et de faire du GOURAUD à 16 Mpixels par seconde en crête, ce qui le situe en tête avec les architectures spécialisées, donc l'idée n'est pas ridicule !

On a vu que lorsque c'était réalisé de manière matérielle, chaque phase utilisait souvent un parallélisme de type différent. Mais si on doit construire une machine plutôt générale il va falloir choisir un mode de parallélisme en particulier permettant toutefois d'exécuter efficacement l'ensemble de l'application.

À partir de maintenant, on ne va plus considérer que les machines parallèles en général dans la suite de la thèse, l'inspiration graphique du projet n'intervenant qu'aux

---

9. En cela, j'avoue honteusement trouver cela plus motivant...

niveaux de certains choix comme celui du mode de parallélisme en fonction des applications ou le rajout d'une sortie vidéo par exemple.

### 2.2.1 La nature du parallélisme

Une bonne introduction à la problématique liée aux choix architecturaux dans un ordinateur parallèle peut-être trouvée dans les articles amusants mais sérieux [Cor91, CK91], dans [PHE77] on trouvera une généalogie du parallélisme et dans [GP85] un aperçu du parallélisme.

Depuis le début de l'informatique, toutes les architectures de machines construites (et aussi certaines autres) ont été classifiées selon différents critères afin de voir un peu plus clair dans le foisonnement des idées.

Une des taxinomies marquantes est celle de FLYNN [Fly66, Fly72] qui a introduit les concepts classiques basés sur la notion de flots d'information ou « *stream* » : les flots de données ( $D$ ) et d'instructions ( $I$ ) qui interagissent. Chaque flot pouvant être simple ( $S$ ) ou multiple ( $M$ ), on étudie le produit cartésien  $\{S, M\} \times \{I\} \times \{S, M\} \times \{D\}$  :

**SISD — Single-Instruction Stream Single-Data Stream :** il s'agit là du processeur dans le sens commun : les instructions sont exécutées l'une après l'autre et elles agissent séquentiellement sur les données. Le flot d'instructions et le flot de données proviennent d'une seule ou de deux mémoires séparées.

**SIMD — Single-Instruction Stream Multiple-Data Stream :** par rapport au processeur « classique » précédent, on permet l'exécution d'instructions qui agissent identiquement sur plusieurs données à la fois, ce qui introduit le parallélisme. Le flot d'instructions provient d'une mémoire qui est commune à tous les processeurs.

**MIMD — Multiple-Instruction Stream Multiple-Data Stream :** on introduit encore plus de parallélisme puisque la machine peut exécuter en même temps des instructions différentes sur des données différentes. Cela donne le plus de liberté au programmeur. Chaque flot d'instructions provient de mémoires de programme différentes ou pas.

**MISD — Multiple-Instruction Stream Single-Data Stream :** l'exécution de plusieurs instructions sur un même flot de données fait de cette catégorie celle des machines pipelinées, telles que les machines graphiques. voire les machines vectorielles ou celles à flot de données. Cette classe est souvent discutée car elle dépend de la distance focale de l'observateur : de loin on peut voir une machine qui exécute des instructions de manière bien séparées alors que si on regarde de près, au niveau du microcode, toutes les machines modernes sont MISD : les données passent séquentiellement par plusieurs entités internes à la machine qui sont les étages de pipeline.

Nous allons détailler chaque classe un peu plus tard dans la suite, avec un cas un peu intermédiaire, le SPMD.

Notons que bien souvent ces termes ont été repris sans les mots « *stream* », ce qui en change le sens. Ainsi, un processeur VLIW capable d'exécuter plusieurs instructions en même temps et donc que certains classent dans *Multiple-Instruction* est en fait

bel et bien un *Single-Instruction Stream* puisqu'il n'est alimenté que par un seul flot d'instruction depuis sa mémoire d'instruction.

Cette taxinomie, comme toutes les autres, ne peut englober toutes les caractéristiques de toutes les machines<sup>10</sup> et n'est pas exhaustive, d'autant plus qu'elle commence à dater. En particulier on a du mal à voir où mettre les machines à flot de données : MISD ou MIMD ? Néanmoins cette taxinomie suffit dans bien des cas à donner une idée de la machine et a une importance culturelle indéniable sur le mode de pensée informatique actuel.

En plus la notion de parallélisme est assez floue et toute relative : n'importe quel ordinateur des années 1960 est parallèle (il manipule des mots) devant une vraie machine de Turing (elle manipule 1 bit à la fois) [Fly66] et tout processeur moderne est hautement parallèle par rapport à cet ordinateur des années 1960. Tout dépend de ce que l'on considère comme taille de donnée naturelle traitée.

On peut faire des analogies entre les modes de parallélisme et certains faits de la vie courante qui sont parallèles. Ainsi lorsqu'on doit faire des crêpes pour la soirée informelle d'un congrès d'informatique théorique, on peut prendre par exemple 2 « billig<sup>11</sup> » plutôt qu'une, afin d'augmenter le débit en crêpes.

Mais alors il y a plusieurs manières de les utiliser :

- soit on fait la crêpe sur une billig et pendant que la première face cuit on fait cuire la 2<sup>ème</sup> face de la crêpe précédente que l'on garnit aussi : c'est le mode pipeliné, avec 2 étages SISD mais qu'on peut aussi considérer comme MISD vue qu'au niveau de chaque billig on fait une opération assez compliquée ;
- soit on fait sur chaque billig une crêpe en parallèle qui peuvent être différentes : c'est le mode SIMD ;
- soit il y a un(e) crêpier(ère) (processeur d'instruction) par billig qui font chacun(e) une crêpe indépendamment : c'est le mode MIMD.

Chaque application a souvent un mode de parallélisme mieux adapté. La dernière solution est la mieux adaptée pour le problème précédent à condition qu'il n'y ait pas un encombrement stérique au niveau des 2 crêpiers(ères) ou de l'approvisionnement en pâte, mais nécessite 2 processeurs d'instruction et il y a donc un compromis économique à trouver<sup>12</sup>.

D'autres taxinomies plus ciblées révèlent plus les caractéristiques de certaines machines avec des niveaux de détails différents. On pourra se reporter par exemple à [Sny88] pour les machines synchrones et [Tuc90] pour les machines SIMD en particulier.

---

10. Il n'y a pas d'ordre total : certains critères valables pour certaines machines n'ont pas de sens pour d'autres.

11. Appareil à faire les crêpes, composé principalement d'une plaque métallique circulaire chauffante sur laquelle on étale la pâte.

12. Nicolas PARIS me propose une analogie avec l'organisation d'une grosse lessive lorsqu'on a à sa disposition plusieurs machines à laver le linge et plusieurs sèche-linges comme à « MONTROUGE ». On remarque qu'on est constamment confronté au parallélisme dans la vie de tous les jours, autant de problèmes qu'il faut savoir résoudre efficacement...

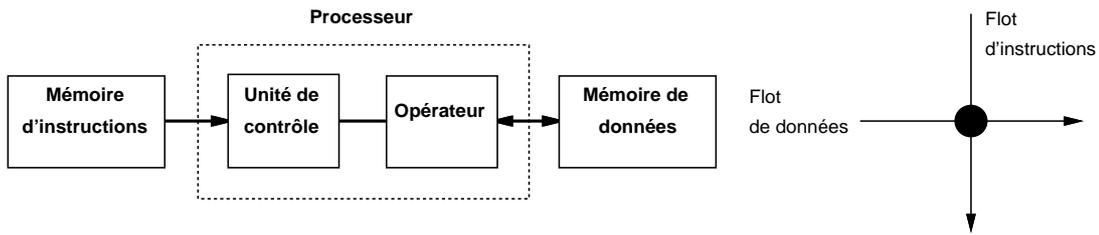


FIG. 2.6 - *Synoptique d'une machine SISD et sa représentation sous forme de flots.*

### 2.2.2 SISD

Mais avant de décrire les types de machines parallèles, décrivons d'abord ce qu'on entend par machine non parallèle.

Une machine typique de cette classe est représentée sur la figure 2.6 avec son équivalent symbolique en terme de flot. Une instruction est lue dans la mémoire d'instruction et a un effet soit interne au processeur (sur des registres) soit externe, sur la mémoire. On peut étendre la notion de flot de données aux registres, en considérant les registres comme étant un cas particulier de la hiérarchie mémoire de la machine : un sous-ensemble de mémoire petite mais très rapide.

On constate qu'exécuter une instruction revient en fait à faire une suite d'opérations telles que : lire une instruction, la décoder, lire les données dont elle a besoin, exécuter l'opération sur ces données, réécrire les résultats. Si on réalise l'opérateur du processeur sous forme purement combinatoire, on s'aperçoit que grossièrement, à cause des temps de propagation dans les portes logiques, il y a propagation d'un front d'information utile précédé par l'information concernant l'instruction précédente. Il y a finalement peu de portes logiques utilisées réellement à un instant donné. En coupant régulièrement la partie combinatoire par des barrières identiquement espacées dans le temps, on peut contrôler de manière fine la propagation de l'information utile. En particulier, on peut mettre de l'information utile entre chaque barrière sans risque d'interférence si on fait passer de manière contrôlée une information à travers chaque barrière simultanément. À un instant donné, on peut donc avoir autant d'informations utiles que de barrières comme l'indique la figure 2.7.

L'efficacité est accrue par un effet de travail à la chaîne : plutôt que d'avoir un gros processeur qui fait le travail, on a plusieurs petites entités qui effectuent chaque partie du travail avant de laisser l'étage suivant continuer le travail. Un des côtés magiques de la méthode est que la vitesse est augmentée sans modification importante de la partie combinatoire, simplement au coût du rajout des barrières.

Si la latence d'une instruction — *ie* le temps écoulé entre le début et la fin réelle d'une instruction — est inchangée au temps de gestion des barrières près, le nombre d'instructions exécutées pendant ce temps est égal au nombre d'étages. Le nombre d'instructions exécutées à un instant donné par le processeur ainsi que cette propriétés sont appelés *confluence*.

Il y a plusieurs limitations principales dans cette machine :

- 1° la vitesse des processeurs augmente plus rapidement que celle des mémoires et par conséquent la mémoire a tendance à devenir de plus en plus limitante, même si on hiérarchise de plus en plus la mémoire en supposant que les accès

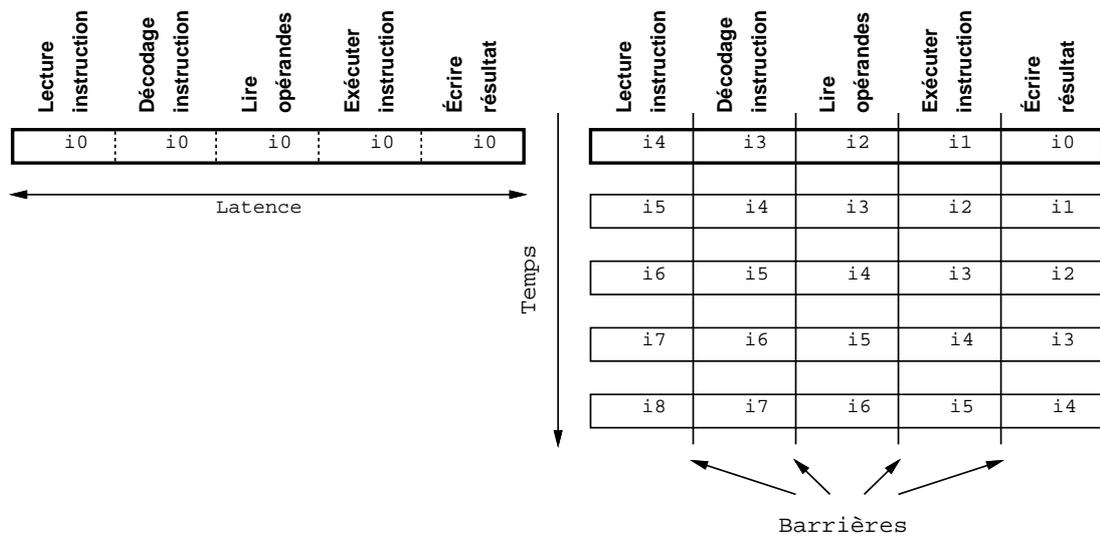


FIG. 2.7 - Amélioration de la bande passante d'instructions grâce au pipeline.

les plus nombreux auront lieu dans les mémoires les plus rapides : registres, mémoire cache primaire, mémoire cache secondaire, mémoire centrale, disques magnétiques et disques optiques, pour aller du plus rapide au plus lent, de la plus faible capacité à la plus grande ;

- 2° si on rajoute trop de barrières, le temps utile au calcul entre chaque barrière devient négligeable devant le temps de gestion d'une barrière et cela ne sert plus à rien d'augmenter le nombre d'étages de pipeline ;
- 3° si on considère qu'un processeur exécute des instructions manipulant des données on se heurte aux problèmes des dépendances au niveau des données (voir la section 8.3.2.1). Une instruction aura probablement besoin d'un résultat calculé par une instruction précédente et ne pourra donc commencer avant que la première ait fini. Si les 2 instructions sont consécutives dans le programme, la seconde instruction est obligée d'attendre pendant toute la latence de la première instruction : on perd le bénéfice de la confluence ;
- 4° on retrouve ce problème de dépendance au niveau du contrôle de flot : une instruction de débranchement conditionnelle peut avoir besoin d'une condition calculée par l'instruction précédente. Il faut dans ce cas aussi attendre pendant toute la latence de l'instruction calculant la condition. Cela se traduit par une certaine « inertie » : on ne peut changer rapidement le flot d'instruction [Fly72]. Plus le programme contient de débranchements de la sorte, plus il est turbulent et moins on bénéficie de la confluence [RF72]. Or en augmentant la confluence de l'ordinateur, on rapproche les débranchements les uns des autres et on est limité par leur latence : c'est un aspect de la loi d'AMDAHL que ce dernier n'avait d'ailleurs pas vraiment considéré lorsqu'il vantait les avantages du SISD dans [Amd67] par rapport aux machines parallèles.

Les supercalculateurs vectoriels exploitent au maximum les techniques de confluence associée à un pipeline au niveau de chaque opérateur flottant en plus : les opérateurs

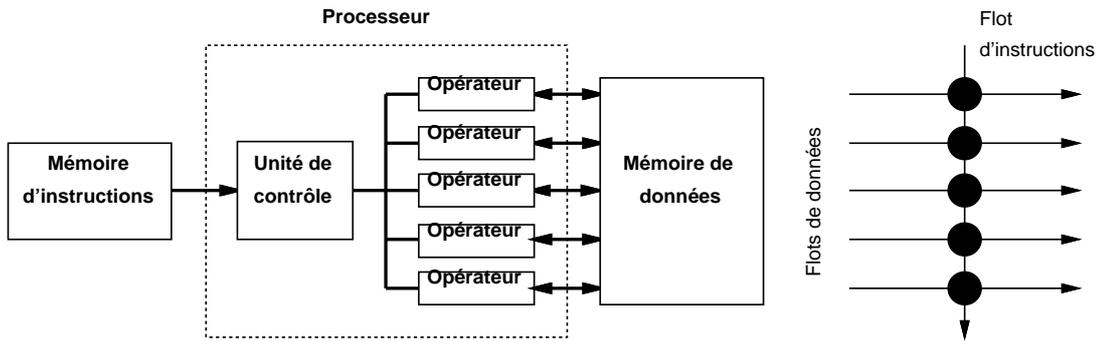


FIG. 2.8 - Architecture typique de machine SIMD avec ses flots.

« vectoriels », capable de manipuler des vecteurs de nombres comme entité naturelle. Le pipeline des opérateurs flottants se fait de manière assez naturelle puisqu'on peut décomposer chaque opération en un calcul de la mantisse, un calcul de l'exposant, un alignement de la mantisse, etc. La limite asymptotique d'une telle machine qui n'utilise que le pipeline comme facteur d'accélération est de toute manière bornée par la vitesse du registre à décalage le plus rapide que l'on sait faire, qui est en fait une opération « nulle » pipelinée. Il faut donc trouver autre chose.

### 2.2.3 SIMD

L'amélioration la plus simple que l'on puisse faire est d'augmenter le nombre de flots de données pour augmenter la puissance de calcul sur des problèmes parallèles ou vectoriels. Ainsi on peut dépasser les limitations de vitesse inhérente à chaque processeur. Comme dans la plupart des calculs de ce type la même instruction est répétée sur plusieurs données, il est raisonnable de factoriser le flot d'instructions et ainsi de garder le flot d'instructions unique vu précédemment. La figure 2.8 illustre le propos. Les problèmes d'interconnexion avec la mémoire, de couplages et de réseaux seront discutés ultérieurement (§ 2.4).

Dans [Fly66] une machine SIMD est constituée d'opérateurs simples et non pipelinés, caractéristiques qui n'ont bizarrement rien à faire dans cette taxinomie puisqu'elles lui sont totalement orthogonales. L'argument évolue dans [Fly72] au point que l'ILLIAC IV [BBK<sup>+</sup>68, Hor82] par exemple perd quasiment son statut de machine SIMD ! L'idée bien pensante associée au SIMD est que la puissance de la machine se fait par un accroissement du nombre de processeurs élémentaires (PE) au détriment de la puissance intrinsèque de ceux-ci.

Est-ce cette remarque qui serait la cause d'une absence de machines SIMD à gros processeurs sur le marché, mis à part les machines vectorielles, à architecture différente, et l'utilisation de la CM-2 en mode *slice wise*? Probablement pas, comme cela sera discuté en 5.1.1.

L'intérêt d'une machine SIMD est qu'elle est simple à concevoir *a priori* et à utiliser dans le cas d'applications où le parallélisme de données (chapitre 3) est important, comme de nombreuses applications scientifiques ou graphiques. Les développements logiciels sont importants en ce moment pour ce type de machines alors que pendant très longtemps elles ont été programmées au niveau du langage machine.

On peut dégager quelques caractéristiques propres :

- les opérations vectorielles directes (élément à élément) sont très rapides car le couplage à la mémoire est souvent optimisé en ce sens 2.4.2 ;
- le fait que certains algorithmes nécessitent une coopération étroite entre processeurs puisse ralentir la machine semble être un faux problème puisque dans le cas d'une machine scalaire cela se traduira aussi par des accès à la mémoire centrale, ce qui prend du temps. En particulier, l'architecture se prête bien à un modèle d'exécution synchrone puisque elle en est la transposition matérielle directe. Il n'y a donc pas de problèmes de synchronisation interprocesseur par définition ;
- en ce qui concerne le code non parallélisable, on est limité bien entendu par le processeur qui exécute ce code [Amd67] de la même manière que les machines vectorielles ;
- enfin le problème le plus gênant est le fait d'avoir un seul contrôleur d'instructions qui impose une certaine rigidité quant à la liberté de prise de décisions au niveau des processeurs. Plus les processeurs voudront prendre leur indépendance et plus l'efficacité de la machine baissera et on sera obligé de faire appel à des artifices (voir [Ker92] et le chapitre 7 pour plus de détails). Néanmoins ce n'est pas trop un problème pour beaucoup d'applications bien parallèles ;
- le problème majeur est l'impossibilité d'utiliser les caches pour accroître la localité des transferts au niveau des données. En effet cela rajouterait des asynchronismes non prévisibles dans l'architecture et empêcherait le beau fonctionnement synchrone. C'est dommage car cela semble être le seul moyen de dépasser le goulet d'étranglement que constitue la liaison processeur-mémoire ;
- un SIMD avec recouvrement du temps de séquençement [KNS91] est ce qu'on peut faire de plus simple au niveau du séquenceur par rapport à un processeur vectoriel, lui aussi SIMD [Fly72], ou un SISD très confluent qui demandera du matériel très compliqué pour garantir que les dépendances entre données sont bien vérifiées. On factorise la partie contrôle des instructions dans le processeur scalaire ce qui simplifie d'autant la construction des processeurs parallèles.

Etant donné que les machines vectorielles sont souvent constituées d'un processeur scalaire relié à des processeurs vectoriels, il est courant aussi de les considérer comme des machines SIMD où chaque tranche de pipeline d'un opérateur vectoriel serait considéré comme un processeur élémentaire dans le cas d'une machine vectorielle sans registre vectoriel [Lin82] ou bien chaque élément de registre vectoriel associé à une portion d'opérateur vectoriel dans le cas d'une machine vectorielle à registres vectoriels [Rus78] serait équivalent à un processeur élémentaire.

#### 2.2.4 MIMD

Afin de palier à la limitation du SIMD en ce qui concerne le manque d'individualité des processeurs, on peut étendre le parallélisme aussi au flot d'instruction : chaque PE est alors capable d'exécuter un programme différent sur des données différentes [Cur63], ce qui amène à une architecture telle que celle de la figure 2.9.

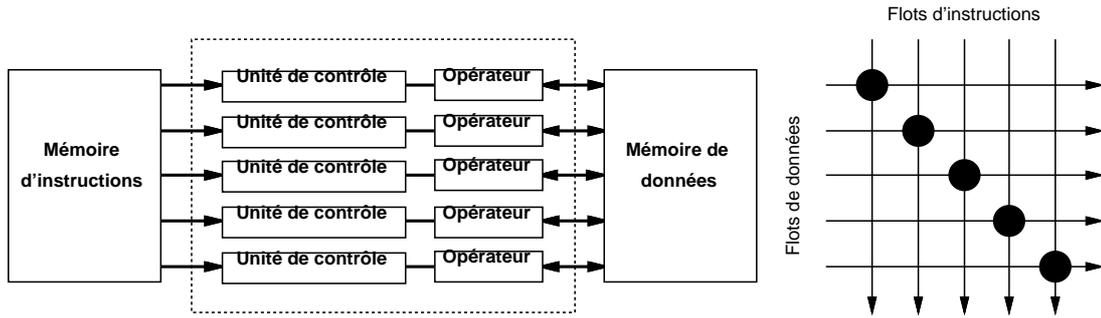


FIG. 2.9 - Architecture typique de machine MIMD avec les flots correspondants.

Les caractéristiques principales des machines MIMD sont :

- plus de liberté dans la programmation du fait de l'indépendance des processeurs. Une application idéale se trouve dans les stations de travail sous UNIX : on peut faire fonctionner plusieurs processus complètement indépendants en même temps sur plusieurs processeurs ;
- le flot d'instruction est multiplié par le nombre de processeurs par rapport au SISD ou au SIMD. Il faut donc alimenter la machine en conséquence ce qui peut être un gros problème dans le cas d'une machine à mémoire partagée mais peut s'arranger par l'adjonction de mémoires caches importantes et d'un système d'exploitation adéquat ;
- la complexité des processeurs est accrue du fait de la présence de la partie contrôle des instructions. Mais comme rien ne différencie un processeur de machine SISD d'un processeur de machine MIMD, on peut utiliser un processeur standard et par conséquent économiser d'office la complexité de conception ;
- qui dit processeur standard dit suivi de l'évolution technologique et utilisation d'un maximum de transistor. Comme ce n'est plus le nombre de transistors qui est actuellement un facteur limitant par rapport au débit mémoire et au nombre de pattes d'un circuit, on peut mettre en œuvre beaucoup d'améliorations grâce à ces transistors supplémentaires pour pallier au mieux à ces facteurs limitants principalement par l'intégration de mémoires caches de plus en plus importantes ;
- lorsqu'on veut faire coopérer plusieurs processeurs dans un modèle de programmation SPMD par exemple, on est amené à synchroniser les processeurs avant de continuer sur une autre phase de calcul, ce qui peut diminuer fortement l'efficacité de la machine<sup>13</sup> si on n'a pas prévu de matériel de « rendez-vous » efficace.

Le BULL GAMMA 60, merveille de la technologie pour l'époque puisqu'il introduit la notion de bus asynchrone, le multitâche, le parallélisme, etc. [Bul57, Bul60], est aussi classé dans cette catégorie [Fly72] même s'il s'agit d'une révolution particulière : une machine MIMD où plusieurs flots de contrôles, donc programmes, se partagent dans

13. Selon Thierry PRIOL lors des Journée d'Etude de Programmation pour les Machines Parallèles du PRC c<sup>3</sup>, le 24 juin 1991, la synchronisation d'un ipsc/2 prend environ 3 ms, donc il ne faut pas trop avoir besoin des synchronisations sinon la machine ne fait plus que ça...

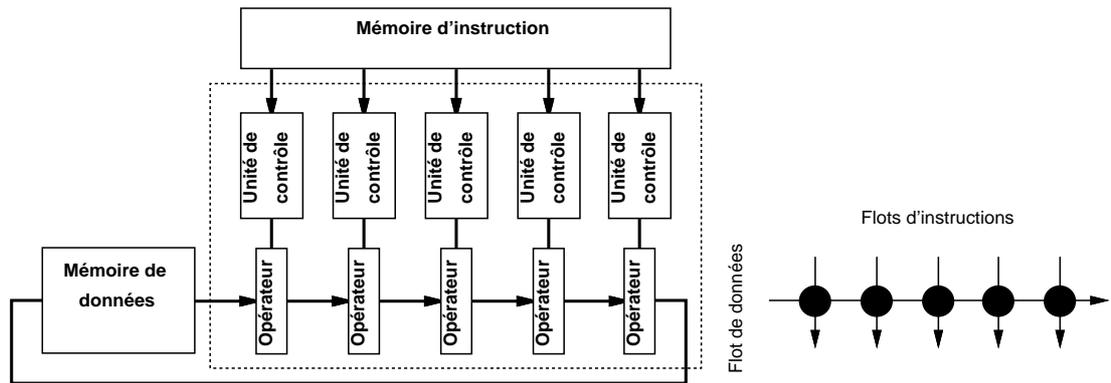


FIG. 2.10 - Synoptique d'une machine MISD avec les flots correspondants.

l'espace les éléments disponibles, à savoir les éléments de calculs (entier, flottant et logique) et les éléments d'entrées-sorties, ainsi que des sémaphores pour contrôler le tout.

Dans cette catégorie on peut aussi à la rigueur classer les machines à flot de données *data flow* qui fonctionnent sous contrôle des données : une opération est déclenchée par un concours de circonstances sur la présence de données au niveau des opérateurs. Comme les opérations sont *a priori* distinctes, on peut dire qu'il s'agit de MIMD. L'intérêt est que les problèmes de synchronisation entre unités fonctionnelles sont résolus implicitement par l'architecture. Ce type de machine prend une importance considérable à l'heure actuelle avec le développement des architectures superscalaires modernes à gestion de confluence dynamique. Un processeur capable d'initier plusieurs instructions à un instant donné avec gestion dynamique des dépendances peut être vu comme un cas particulier d'une architecture à flot de données où les dépendances sont exprimées par des noms de registre [Tom67, TF70, OMMN90]. Même si la complexité d'un processeur superscalaire est exponentielle [Fly66, TF70], c'est un moyen d'utiliser les transistors disponibles.

Du fait de l'évolution technologique de la densité des transistors par circuits intégrés, il est probable que les machines MIMD vont supplanter les machines SIMD et vectorielles puisque la notion d'efficacité par transistor n'est plus une bonne métrique dans la conception des ordinateurs.

### 2.2.5 MISD

Il s'agit d'une classe un peu méconnue qui regroupe les machines où une série d'opérations est effectuée sur un même flot de données. Par rapport au SISD confluent, il faut voir la différence au niveau du grain de la tâche : alors que dans une machine SISD chaque opération est découpée en une suite très simple d'opérations, ici il s'agit de plusieurs processeurs effectuant des calculs plus compliqués l'un derrière l'autre, comme le montre la figure 2.10.

Les caractéristiques clés du MISD sont :

- le débit mémoire est très réduit puisqu'un seul flot traverse tous les opérateurs de la machine ;

- le débit d'instructions est important mais est à relativiser compte tenu que les tâches exécutées sur chaque opérateurs sont généralement très simples. On peut donc stocker les programmes dans des mémoires de petite capacité, donc très rapide, et on peut même s'affranchir du débit d'instruction en intégrant cette mémoire au processeur ;
- le problème de l'architecture est son style de programmation très particulier qui impose qu'on soit capable de découper un problème en un ou à la rigueur quelques flots de données, ce qui n'est possible que sur des applications assez spécifiques. La restriction à ce niveau est donc encore plus forte que pour une machine SIMD : même si on a la liberté d'exécution, on n'a pas de choix possible au niveau des données.

La version moderne du MISD apparaît dans les architectures systoliques telles que les machines PCS [BCW89, Rou90] ou iWARP [Int88] qui sont bien adaptées aux calculs très réguliers comme les multiplications de matrices pleines, les corrélations, le traitement du signal, etc. Ce genre d'applications ne semblaient pas encore effleurer l'auteur de la taxinomie puisque de [Fly66] à [Fly72] le MISD avait disparu...

Se pose la question de l'adéquation du MISD à des applications qui sont moins régulières, comme les applications scientifiques plus générales où on a plusieurs flots d'instructions et des besoins d'indirections globales de tableaux par exemple. Pour cette raison, les machines de type flot de données se prêtent mieux pour faire des accélérateurs spécialisés plutôt que des machines générales.

Mais si on regarde d'un peu plus près un processeur comme le iWARP, on s'aperçoit qu'il s'agit en fait d'un processeur normal et la machine globale est en fait MIMD : seul le modèle de programmation est systolique ou MISD. Pour des raisons similaires à celles évoquées dans la partie décrivant le SIMD, il est fort probable que les machines systoliques évolueront vers des machines MIMD, ce qui souligne encore une fois l'importance du modèle de programmation et son découplage relatif de l'architecture.

### 2.2.6 SPMD

On peut rajouter un autre type d'architecture à la taxinomie de FLYNN qui serait quelque part entre du SIMD et du MIMD : *Single-Programm flow, Multiple-Data flow*. Cette architecture pourrait permettre une exécution optimale du modèle à parallélisme de donnée habituel en calcul numérique : globalement on exécute un seul programme mais il peut y avoir des variations locales qu'il faut être capable d'exécuter rapidement.

A défaut de définir une architecture très spécifique, une machine globalement MIMD mais avec des mécanismes de synchronisation permettrait de répondre au problème (voir le chapitre 12).

C'est l'approche prise par exemple dans la CM-5 [Thi91], malgré une architecture assez baroque : des machines vectorielles SIMD sont reliées entre elles par un réseau de synchronisation réalisant une barrière floue globale [GE90].

Une question qu'on peut se poser, puisqu'on est dans une situation intermédiaire entre le SIMD et le MIMD, est de savoir si oui ou non on gardera un processeur scalaire sur la machine ou on répliquera le code scalaire sur tous les processeurs [HLJ<sup>+</sup>91] ou bien encore si un des processeurs sera à la fois le processeur scalaire et un processeur parallèle. Il semble que cette question ait de moins en moins d'importance dans

la mesure ou la puissance des processeurs parallèles est souvent identique à celle du processeur scalaire. Le fait de garder un processeur scalaire peut faciliter l'intégration de la machine dans un univers scalaire déjà existant (station de travail) et accélérer certaines tâches séquentielles pendant que les processeurs scalaires travaillent (voir à ce sujet le chapitre 11).

### 2.2.7 Les ordinateurs vectoriels

On peut se demander si cette classe de machines capables d'opérer sur des vecteurs fait partie ou non des classes précédentes. En fait cela dépend de la distance focale de l'observateur : de près, une machine vectorielle exécutera plusieurs instructions à suivre sur un même flot de données<sup>14</sup>, donc appartiendra à une classe MISD, mais de loin peut apparaître comme une machine SIMD opérant en une instruction vectorielle sur plusieurs données.

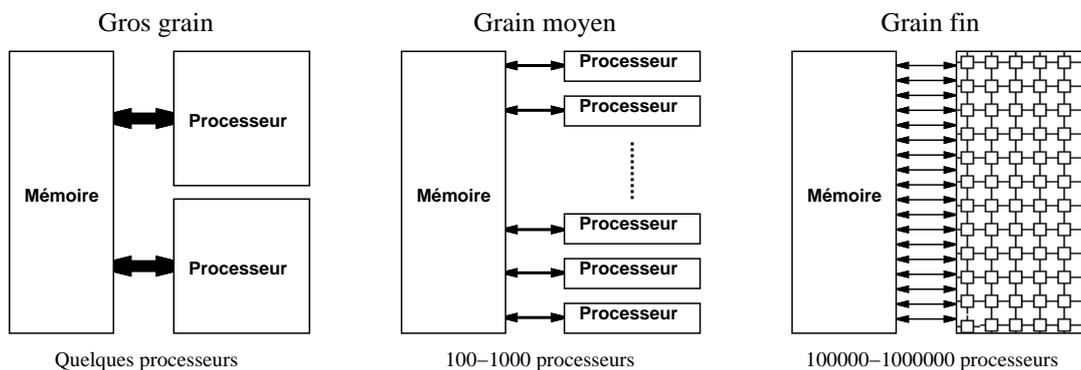
Les hautes performances sur les vecteurs sont atteintes grâce à une grande confluence des opérations et un pipeline de ces dernières poussé à l'extrême. L'idée est que si on travaille sur des vecteurs, on sait qu'on aura à faire des calculs sur des flots de données *assez longs* (composés des éléments de chaque vecteur) et donc que l'inertie du pipeline sera *négligeable*, du moins pour des vecteurs *assez grands*. Bien entendu tout dépend des constantes « *assez* ». Pour cette raison, en plus des performances de la machine on précise souvent un nombre  $N_{\frac{1}{2}}$  qui correspond à la taille minimale d'un vecteur pour atteindre la moitié de la puissance crête de la machine et qui donne une idée de l'inertie de la machine.

La classification d'une machine vectorielle dépend de l'échelle d'observation : de près il s'agit d'une machine SISD très confluente et pipelinée au niveau des opérateurs arithmétiques. A distance moyenne on peut voir chaque tranche d'opérateur arithmétique comme étant un des opérateurs d'une machine MISD très simplifiée où les opérateurs ne sont capables de faire qu'un seul type d'opération chacun. Si la machine permet le chaînage des pipelines [Rus78, HSN81] on peut alors parler clairement de MISD puisque les opérateurs effectuant des opérations différentes les uns des autres sont, bien sûr, programmables et peuvent être reliés à la suite les uns des autres pour éviter à avoir à repasser par la mémoire centrale, sans même attendre le remplissage complet du registre vectoriel intermédiaire. Enfin de loin, comme la machine opère simultanément sur des grands ensembles de données, on peut l'assimiler à une machine SIMD. En fait, c'est dans cette classe qu'il faut globalement considérer une machine vectorielle puisqu'au niveau de l'utilisateur, l'équivalence sémantique prévaut. Disons qu'une architecture vectorielle est au SIMD ce que le BULL GAMMA 60 est au MIMD : c'est une machine SIMD à multiplexage temporel du processeur pipeliné.

Les caractéristiques sont exactement les mêmes qu'une machine SIMD si ce n'est que la notion de taille de machine est différente. En effet, comme sur une machine SIMD, une seule opération vectorielle déclenche l'exécution d'une opération sur plusieurs éléments d'une variable parallèle. Au nombre de processeurs d'une machine SIMD on rapprochera la taille des registres vectoriels qui précisent la taille des données traitées par cycle vectoriel. Une petite différence apparaît si la machine vectorielle autorise des tailles de vecteurs non multiples de la taille des registres vectoriels car dans ce cas elle pourra

---

14. Cela est encore plus vrai dans le cas des ordinateurs permettant le chaînage de plusieurs pipelines effectuant des opérations différentes.

FIG. 2.11 - *Principale classification du grain du parallélisme.*

être plus efficace qu'une machine SIMD de même taille puisque cette dernière exécutera des calculs inutiles liés à une inadéquation de la taille des données avec la taille de la machine [Fly72, HSN81].

Enfin les machines vectorielles modernes ont tendance à mélanger tous les styles d'architectures et sont constituées de multiprocesseurs vectoriels que l'on peut voir comme une hiérarchie d'architecture : MIMD macroscopique multitâche ou SPMD au dessus d'un SIMD microscopique, etc. ce qui rajoute encore plus de flou à la taxinomie précédente.

Le choix de construire une machine SIMD ou vectorielle est un problème de technologie : une machine vectorielle possède moins de transistors car il n'y a qu'un opérateur très pipeliné mais très rapide donc qui dissipe beaucoup et la technologie est chère et peu intégrée alors qu'une machine SIMD peut être construite avec une technologie moins rapide mais très intégrée où le nombre de transistors utilisés importe assez peu.

Un élément déterminant est le degré de parallélisme : alors qu'on ne peut augmenter à volonté le nombre d'étages de pipeline, comme on l'a vu en § 2.2.2, on peut augmenter arbitrairement le nombre de processeurs d'une machine parallèle, dans la mesure où le parallélisme du problème à résoudre suit en conséquence [LV82].

L'évolution semble donc orientée vers le massivement parallèle avec l'annonce de futures machines MIMD/SPMD comme la CM-5 ou le CRAY MPP plus simples à construire car basées sur des technologies plus « froides » que des machines comme le CRAY 3, véritable luxure technologique.

## 2.3 Le grain du parallélisme

C'est une notion difficile à introduire dans les taxinomies car il n'y a pas souvent de relations simples entre la taille des processeurs élémentaires (discutée plus précisément dans la section 5.1.1), leur nombre et l'efficacité d'une machine sur un algorithme donné. En plus, avec l'évolution technologique, on assiste au développement de machines avec de nombreux processeurs de puissance unitaire importante, ce qui a tendance à décaler les machines à grain fin vers toujours plus de processeurs.

On peut néanmoins classer les machines en 3 grandes catégories représentées de manière symbolique sur la figure 2.11.

### 2.3.1 Gros grain

On entend par là les architectures où la puissance de calcul est obtenue en prenant les processeurs les plus gros et les plus complexes possibles, représentant le SISD technologiquement ultime. Comme cela, il suffit de quelques processeurs pour obtenir des performances intéressantes.

C'est l'approche prise par toutes les machines vectorielles actuelles ou plus anciennes [HT72, Rus78, HSN81, Lin82, Che83, MU84, NEC91, Cra91a] avec pour les plus récentes utilisation du parallélisme MIMD pour augmenter encore plus les performances [Che83, NEC91, Cra91a].

Le problème principal de ces machines est dans leur définition même: elles demandent la conception de processeurs spéciaux très (trop?) complexes si on en croit le temps consacré à la mise au point du CRAY 3. Plutôt que d'utiliser des briques de bases à haute intégration tels que des processeurs, ces machines sont basées sur des circuits intégrés à plus faible intégration (10000 portes par circuit pour le CRAY Y-MP C90 [Cra91a], 480 circuits différents en AsGa) mais plus « chauds » ce qui pose tous les problèmes de dissipation thermique classiques.

Par conséquent la complexité de l'organisation et la technologie liées à ces machines les éloignent clairement du milieu universitaire standard, pour des raisons de coût aussi bien que de personnel (ce qui est d'ailleurs lié).

### 2.3.2 Grain fin

A l'inverse, on peut construire une machine qui a beaucoup de processeurs si le parallélisme de l'application est important. Même si les processeurs sont peu puissants, comme il y en a beaucoup, on peut espérer obtenir des performances considérables. C'est cette approche qui est prise dans des machines comme celle décrite dans [Ung58], SOLOMON [SBM62], la CM-2 (64K processeurs 1 bit [Thi87a]), la MP-1 (16K processeurs 4 bits [Bla90b]), la WAVE TRACER DTC (16K processeurs 1 bit [Jac90]), les PIXEL-PLANE 4 et 5 (256K processeurs 1 bit [FPE<sup>+</sup>89]).

Toutes les machines décrites précédemment sont SIMD et les processeurs sont très simples, ce qui permet d'avoir des processeurs suffisamment petits pour être intégrés à plusieurs par circuit intégré (parfois par centaine par CI) et de faire une machine compacte malgré le nombre important de processeurs. Le côté SIMD est intéressant car on rejette tout le contrôle du flot d'instructions sur le processeur scalaire qui peut être tout à fait standard, simplifiant ainsi la conception de la machine.

Paradoxalement, de même que dans les machines à gros grain on est obligé de construire des processeurs spécifiques tellement ils sont gros, dans les machines à grain fin les processeurs sont tellement petits qu'on est aussi obligé d'en construire spécialement puisqu'ils doivent être intégrés à plusieurs par circuit<sup>15</sup>! Néanmoins, vue la simplicité de chaque processeur, le travail reste à la portée d'une équipe de recherche universitaire, voire d'une seule personne. La preuve en est que les machines précé-

---

15. On trouve aussi sur le marché des processeurs très simples que l'on nomme *microcontrôleur* car ils sont réservés à des tâches de type gestion d'une machine à laver ou d'un téléphone, etc. Malheureusement, vues les applications visées ils ne sont qu'en un exemplaire par circuit intégré. Peut-être cela changera-t-il lorsque les machines à laver parleront et estimeront le programme à utiliser en fonction de certains paramètres du linge...

dentes ont toutes à la base un projet universitaire et les PIXEL-PLANE ont été réalisées en milieu universitaire.

Si on veut réellement construire une machine, c'est clairement une bonne solution, plus accessible que la solution à gros grain.

### 2.3.3 Grain moyen

Enfin, entre les deux approches on trouve le reste, c'est-à-dire les machines qui sont basées sur des processeurs de taille conventionnelle, tels ceux qu'on trouve dans les stations de travail. On trouve dans cette catégories toutes les machines à base de TRANSPUTER [?, ?, INM91, INM89], les machines multiprocesseurs à mémoire partagée comme les BULL DPS 7000 [Bul92], CAMPUS/800 [All91], ENCORE 90 [Enc91], BBN MONARCH [RCCT90] ou bien MIMD à mémoire distribuée comme la PARAGON [Int91c] ou la CM-5 [Thi91].

Bien entendu, il est plus facile de réaliser une machine avec des processeurs du commerce que lorsqu'il faut construire aussi le processeur. En plus l'évolution technologique suivant son cours, on peut choisir un des processeurs le plus rapide du marché pour avoir une des machines les plus performantes.

Du fait que les processeurs sont complets en soi avec un contrôleur d'instructions, les machines de ce types sont de type MIMD, à l'exception de la machine PASM qui était SIMD et MIMD bien que basée sur des 68010 [SSKD87]. Il faut tenir compte de ce fait lors du choix du grain de la machine.

## 2.4 Le couplage entre processeurs et mémoires

On entend par là la manière dont les processeurs sont reliés entre eux et la place occupée par la mémoire vis-à-vis des processeurs.

Le rêve de l'utilisateur non perverti par l'étude du parallélisme en soi est de voir toute machine comme un ordinateur SISD : un processeur avec sa mémoire, même s'il y a plusieurs processeurs et plusieurs mémoires. Comme cela il peut continuer sa programmation d'antan sans avoir à retoucher son programme, que cela soit fait de manière automatique ou manuelle.

Une solution simple en théorie, est d'avoir une unique mémoire pour tous les processeurs qui servira bien sûr en tant que mémoire pour chaque processeur mais aussi comme moyen de coopération entre tous les processeurs, sa synchronisation grâce à des sémaphores en mémoire, etc.

Malheureusement, plus la mémoire d'un ordinateur est grosse et plus elle est lente, ce qui peut enlever de l'intérêt au parallélisme de la machine. Cela vient du fait que chaque circuit ne peut commander qu'un nombre limité d'autres circuits et qu'il faut faire des distributions de signaux avec des amplificateurs intermédiaires, ce qui prend du temps, problèmes auxquels se rajoutent les temps de propagations accrus par une augmentation de l'encombrement physique de la mémoire : les fils sont plus longs et la vitesse de propagation des signaux est finie, de l'ordre de  $\frac{2}{3}c$ , où  $c$  est la célérité de la lumière dans le vide.

Outre le problème de la taille de la mémoire, survient le problème de son partage entre tous les processeurs qui veulent y accéder simultanément et cela crée un important goulet d'étranglement.

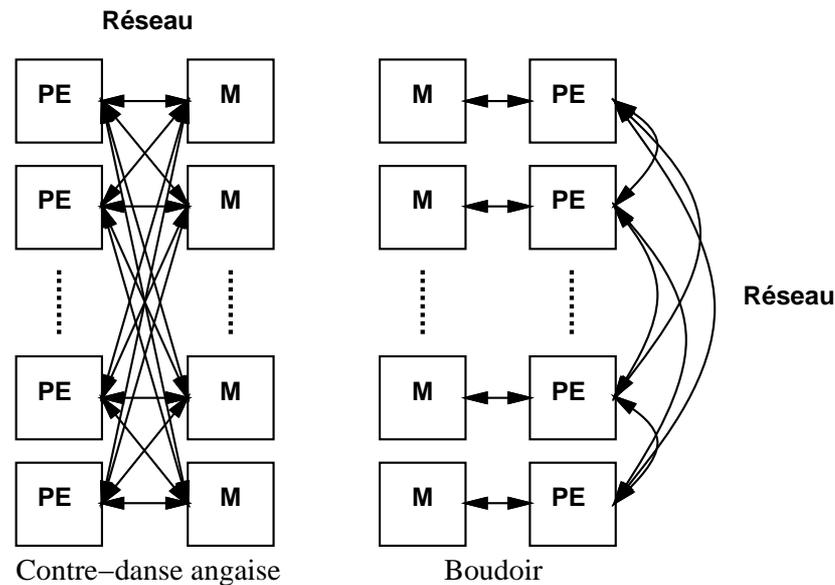


FIG. 2.12 - Comparaison entre le couplage fort et le couplage faible.

Deux méthodes sont utilisées pour essayer de résoudre les problèmes précédents.

#### 2.4.1 Contre-danse anglaise : couplage fort

Une manière de contourner le problème est de remplacer l'énorme mémoire par plusieurs mémoires plus petites, les bancs mémoires, mais plus rapides que l'on essaiera de relier ensemble sans que cela prenne « trop » de temps ni trop de matériel [Sto70].

On a toujours le problème de la distribution des signaux qui croît grossièrement en  $\mathcal{O}(\log n)$  de la capacité mémoire, mais on peut pipeliner les accès et donc augmenter le débit même si on ne peut guère diminuer la latence.

Il s'agit là de la méthode utilisée dans les supercalculateurs et les stations de travail haut de gamme. La première solution est donc de relier chaque processeur à toutes les mémoires, évitant ainsi de particulariser une plus qu'une autre. Tout se passe comme dans une contre-danse anglaise : chaque cavalier peut danser avec une cavalière différente en fonction de la figure et de la danse (respectivement phase et motif de communication). On dit qu'on a affaire à un couplage fort, comme indiqué sur le dessin de gauche de la figure 2.12.

Un intérêt du système est qu'on peut dissocier le nombre de processeurs du nombre de bancs de mémoire afin de diminuer les schémas de communication conflictuels pouvant survenir [Law75] lors de calculs sur des tableaux (vecteurs, matrices). Ainsi, prendre un nombre de bancs mémoires premier et supérieur au nombre de processeurs est un moyen de résoudre le problème et cette méthode a été étudiée dans la machine BSP où 17 bancs mémoires étaient prévus pour 16 processeurs [LV82]. En effet il faut que la plupart des accès mémoire typiques des applications puissent avoir lieu sans conflits si on veut que la machine soit performante. En particulier on essaye de favoriser les accès par colonne, par ligne, par diagonale directe ou inverse, en échiquier, etc.

La réalisation de ce couplage fort est en général faite de manière structurée : un processeur accède à plusieurs canaux mémoire (ce qui revient à avoir un commutateur à croisillons), chaque canal est relié à une section parfois divisée en sous-sections contenant elles-mêmes les bancs physiques de mémoires [dD91]. Le problème est que la répartition des accès à la mémoire peut créer des conflits au niveau de chaque hiérarchie et annuler l'effet pipeline.

Pour ce faire, de nombreuses fonctions de biais ou de hachage ont été développées indiquant dans quel banc mémoire une donnée correspondant à tel ou tel élément du tableau doit se trouver tout en essayant de limiter les conflits d'accès, que ce soit au niveau des canaux mémoires ou des éléments mémorisants eux-mêmes [Law75, LV82, FJL85, dD91]. Une approche astucieuse de ce problème est de le ramener à la résolution d'un « carré magique » [BJR88]. Malheureusement, faire un adressage rapide de la mémoire suivant une méthode peu facile à calculer semble irréalisable car elle nécessite des tables de valeurs importantes.

Le problème de cette méthode est qu'on optimise le cas pire, celui où on ne sait rien sur les relations entre les processeurs et le placement des données impliquées par l'algorithme. En effet, même si un réseau est optimisé pour permettre un échange quelconque entre les processeurs et les mémoires — pour peu que cela soit possible — la présence de l'échange préférentiel où chaque processeur veut communiquer avec sa<sup>16</sup> mémoire ne sera pas optimisé [AP91a]. Par conséquent l'interface mémoire est plus compliquée, surtout lorsque le nombre de processeurs augmente.

Néanmoins, malgré sa complexité, cette méthode est celle employée sur les ordinateurs vectoriels qui justement possèdent peu de processeurs mais beaucoup de bancs mémoire, comme par exemple le CRAY Y-MP C90 qui possède un débit mémoire technologiquement impressionnant de 250 Go/s avec 64 processeurs [Cra91a] ainsi que les machines multiprocesseurs à faible nombre de processeurs. À noter tout de même le cas de la machine BSP [KS82] qui devait être à couplage fort bien que typiquement SIMD et qui fait ainsi l'exception.

### 2.4.2 Boudoir

L'autre approche est de tabler sur la localité des calculs en remarquant que bien souvent on peut décomposer un problème en sous-domaines de façon à regrouper dans une même mémoire les calculs qui interviennent. Comme le matériel est du coup assez simple à réaliser, on va optimiser le cas le plus courant, ce qui correspond encore à la philosophie RISC. Le cas du modèle vectoriel se prête bien à ces regroupements puisque la majorité des calculs font appel à des interactions de vecteurs en correspondance élément à élément.

Dans ce cas, afin d'obtenir le débit le plus élevé possible, chaque processeur est connecté à sa propre mémoire et les calculs où les processeurs ont besoin de données possédées par les mémoires d'autres processeurs sont réalisés après réception de celles-ci à travers un réseau d'interconnexion entre les processeurs, comme indiqué sur le dessin de droite de la figure 2.12.

L'interface mémoire est réduite à sa plus simple expression : celle d'un processeur

---

16. Dans la mesure où ce possessif a un sens : dans le cas d'une machine à couplage fort, l'attribution se fera arbitrairement à la compilation ou à l'exécution.

scalaire. Par conséquent son débit est maximal à faible coût<sup>17</sup>, et la complexité du réseau est moindre que celle du réseau nécessaire à la contre-danse anglaise puisqu'il ne relie que les processeurs entre eux et non plus tous les processeurs à toutes les mémoires. En plus, les processeurs peuvent faire du travail supplémentaire et alléger le réseau étant donné que le compilateur peut souvent prévoir qu'il y a des communications à faire.

On trouve en gros dans la catégorie des machines à couplage faible toutes les machines SIMD [TR88, Bla90b, Jac90] ou plus anciennes comme l'ILLIAC IV [BBK<sup>+</sup>68, Sto70] ou OPSILA [Aug85], ainsi que les machines MIMD possédant beaucoup de processeurs comme la machine DELTA [Int91c] ou la CM-5 [Thi91].

C'est dans cette voie dite du couplage faible ou en boudoir<sup>18</sup> que se développent la plupart des machines actuelles possédant beaucoup de processeurs car il devient impossible raisonnablement d'assurer une mémoire globale vraie. Néanmoins certains projets ou machines vont dans le sens d'une mémoire globale partagée virtuelle : on émule logiquement une mémoire globale [?, PM92, ?, NL91].

## 2.5 Conclusion

Le produit cartésien de ces choix conduit à des ensembles de machines qui peuvent être vides dans la pratique ou au contraire très remplies.

Notre choix pour la version actuelle de POMP découle des considérations précédentes ainsi que d'autres exposées plus tard :

- parallélisme de type SIMD car il simplifie les problèmes de synchronisation et de contrôle tout en permettant de réaliser un nœud plus compact (discuté en §§ 5.3, 10.1.2.1 et ?? ;
- parallélisme à grain moyen car il permet de suivre le développement technologique des grands constructeurs de microprocesseurs ;
- couplage faible entre le processeur car c'est le seul moyen réaliste de construire une machine dans notre petite équipe.

L'architecture devant être précisée aussi bien au niveau des processeurs élémentaires au niveau du contrôle et du processeur scalaire, le synoptique plus détaillé sera donné au chapitre ?? (figure ??).

Mais avant, détaillons de plus près le modèle de programmation et le langage qui nous influencera dans les choix plus fins de l'architecture finale.

---

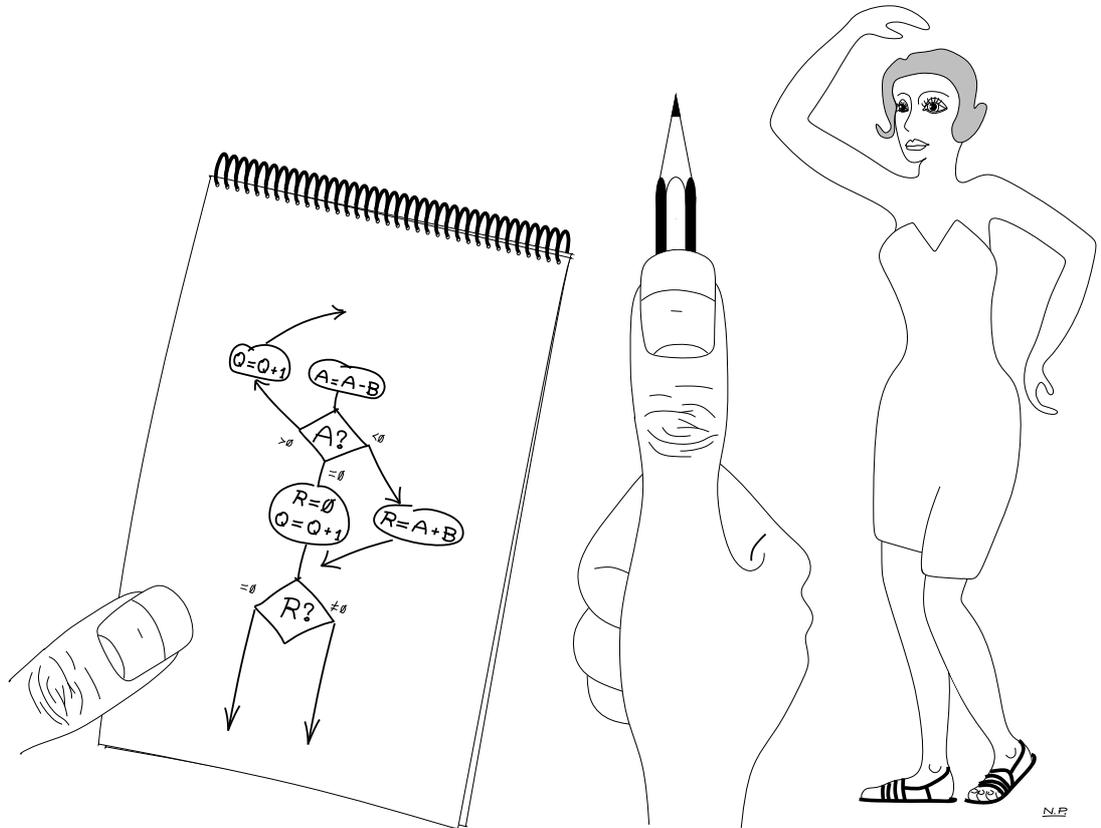
17. Cette vision des chose est un peu idyllique car avec l'apparition de processeurs de plus en plus performants comme le 21064-AA [Dig92a], on est obligé de diviser la mémoire en plusieurs bancs même sur les ordinateurs scalaires.

18. Ce terme possède une connotation aux danses américaines très marquée. Une meilleure traduction française serait probablement « dañs Léon » ou « Piller Lann » [?], qui est une danse bretonne exhibant ce type de couplage.



# Chapitre 3

## Le modèle de programmation



TAB. 3.1 - *Généralité contre contrôlabilité.*

	<i>Généralité</i>	<i>Contrôlabilité</i>
1970	<b>L'ère du GO TO</b> (FORTRAN, assembleur)	<b>Programmation structurée</b> (PASCAL, C)
1990	<b>Parallélisme</b> (ADA, OCCAM)	<b>Parallélisme synchronisé</b> (C*, MULTIC, MPL, ?)

Causer de ICPP93/data\_parallel

 Le problème principal que se pose un programmeur devant un ordinateur parallèle est de savoir comment il va pouvoir utiliser cet ordinateur pour exécuter son algorithme le plus efficacement possible, distribuer le travail à l'ensemble des processeurs pour les faire coopérer vers le résultat. Dans ce but, il doit utiliser un langage de programmation permettant de traduire son algorithme avec des notions de parallélisme en un langage compréhensible par la machine et de préférence aussi par le programmeur.

Il semble donc nécessaire d'avoir un *modèle de programmation* décrivant la philosophie de programmation et garantissant une sémantique de l'algorithme indépendante de la machine cible, pour des raisons d'efficacité de programmation et de portabilité. Dans la suite de ce chapitre, comme dans [Kar87], on se place dans le cadre des besoins de la programmation scientifique qui comporte beaucoup de calculs sur des nombres flottants et la discussion ne concerne donc pas d'autres domaines telle que la programmation logique par exemple.

Puisqu'une machine utilisable est composée à la fois d'une architecture et d'un langage, il est normal que le développement d'architectures parallèles s'accompagne d'une apparition de langages qui prennent en compte le parallélisme. En effet, lorsqu'un programmeur utilise un langage séquentiel, il introduit souvent trop de séquentialité car il est obligé de tout penser en séquentiel, ce qui se traduit par une perte d'information pour le compilateur. Or c'est justement un problème que l'on retrouve sur beaucoup de supercalculateurs actuels : même si le programmeur pense à des données parallèles, il doit séquentialiser le programme en FORTRAN, programme duquel le compilateur vectoriseur doit essayer d'extraire le plus de parallélisme possible. C'est d'autant plus dommage et inefficace que beaucoup de problèmes ont une solution parallèle plus simple et plus élégante que la solution séquentielle. On a donc intérêt à garder le plus d'information possible sur l'algorithme de départ formulé grâce à un modèle de programmation parallèle si on veut l'exécuter le plus efficacement possible.

Il est intéressant de constater que le parallélisme touchant aux applications calculatoires se développe selon deux axes : de même que les langages des années 70 (et même encore maintenant) pouvaient être classés en langages structurés, avec la notion de blocs sémantiques, ou non structurés (on précise tout le flot de contrôle), le parallélisme peut être soit arbitraire (on précise tout le parallélisme, processeur par processeur), soit structuré de manière plus synchrone, comme indiqué sur le tableau 3.1 [Bou91].

Dans les machines et les langages parallèles actuels on retrouve la notion de synchronisme et d'asynchronisme (tableau 3.2) [Ste90] que l'on rapproche de manière assez naturelle :

- à une machine de type MIMD, où chaque processeur est indépendant des autres, on

TAB. 3.2 - *La notion de synchronisme dans les langages et les architectures parallèles.*

	<i>Asynchrone</i>	<i>Synchrone</i>
<i>Architecture</i>	<b>MIMD</b> + général	<b>SIMD</b> + contrôlable
<i>Langage</i>	<b>OCCAM</b> + général	<b>C*</b> + de contrôle

associera un langage asynchrone de type OCCAM mettant l'asynchronisme comme philosophie de base permettant au programmeur la libre expression la plus totale ;

- à une machine de type SIMD, où chaque processeur est synchronisé avec les autres, on rapprochera un langage à parallélisme synchrone tels que C\*, MULTIC ou MPL qui prennent ce synchronisme des opérations comme principe fondamental de coordination entre les processeurs.

De même que la programmation structurée a apporté à la programmation non structurée beaucoup d'avantages tels que :

- une conception plus facile des programmes ;
- une approche descendante du problème, « de haut en bas » ;
- une maintenabilité accrue liée à la modularité d'écriture et de décomposition du programme ;
- une lisibilité améliorée pour soi et surtout pour les autres ;
- un temps de développement réduit et donc de meilleurs rendements ;
- une réutilisation du code plus facile pour construire d'autres programmes.

Ces avantages ont été acquis parfois au prix d'une efficacité d'exécution un peu réduite, il est vrai, mais négligeable par rapport aux avantages<sup>1</sup>, le suivi, d'un modèle de programmation parallèle synchrone pour l'écriture d'un programme parallèle apporte, lorsque c'est possible [Ste90, HQ91] :

- une exécution garantie déterministe et bornée liée à la sémantique ;

---

1. Néanmoins il faut nuancer ceci par deux remarques :

- il est des cas où il faut programmer de manière « sale », lorsqu'on a des impératifs de temps réel ou des primitives systèmes de très bas niveau à faire dans un système d'exploitation ;
- la plupart des lignes de code utilisées de nos jours sont probablement écrites en Cobol en Fortran ou en assembleur. Malheureusement, il faut probablement mettre ce triste état du monde sur le compte de la pollution de l'environnement en général, liée au manque d'éducation de base en écologie, associée à une très forte inertie des environnements sociaux héréditaires (particulièrement ceux de la gestion et de la physique). Mais cela nécessitera, comme toujours, une « internalisation des externalités » [LG92, Pig20] : le coût d'un programme ne se limite pas au nombre d'heures d'écriture, comme on aurait tendance à le penser, mais inclut aussi le coût du recyclage ou de la réutilisation du code, voir des usines de retraitement capables d'incinérer les milliards de lignes de Cobol qui seront mises au rebut, la récupération du papier des listings faits par manque d'utilisation de débogueur symbolique par les sus-dits programmeurs, etc.

TAB. 3.3 - *Petit dictionnaire vectoriel-parallèle.*

Langue française	Vectoriel	Parallèle
<i>Emission parallèle</i>	<b>Scatter</b>	<b>Send</b>
<i>Réception parallèle</i>	<b>Gather</b>	<b>Get</b>
<i>Virtualisation</i>	<b>Strip-mining</b>	<b>Processeurs virtuels</b>

- une diminution de la complexité puisqu'elle ne croît plus avec le nombre de processeurs : le programme est unique quelque soit le nombre de processeurs<sup>2</sup> ;
- une validation plus simple ;
- la notion d'état global existe, donc la mise au point des programmes est facilitée : on peut afficher une variable parallèle, calculer une condition globale, etc. contrairement à une programmation orientée tâche qui complique énormément la mise au point [AP87] ;
- le parallélisme est structuré et facilement embrassable.

On retrouve donc le dilemme de la structuration contre l'efficacité dans le cas du parallélisme synchrone (structuré) contre le parallélisme asynchrone. Il faut voir cela comme un investissement logiciel à long terme que l'on pourra réutiliser pour d'autres programmes tout en conservant des performances plus que raisonnables [Ste90, HQ91].

Bien que très intéressant, le parallélisme synchrone n'est pas toujours possible, par exemple en ce qui concerne les systèmes d'exploitation multiprocesseurs, où chaque processus est indépendant, ou dans le cas de certaines applications purement fonctionnelles ou de bases de données, par exemple.

### 3.1 Au sujet de quelques problèmes de langage

Mais avant de décrire quelques langages existants, il convient de préciser quelques points au sujet des notations et des langages tenus par deux communautés scientifiques différentes et apparemment antagonistes. Il semble qu'il y ait deux religions bien séparées qui invoquent les mêmes dieux mais avec des noms différents. De peur de blesser les partisans des deux bords qui pourraient mettre les yeux sur cette thèse, de déclencher des querelles de clocher et de provoquer des incompréhensions, je préfère expliquer les termes litigieux qui risquent d'apparaître par la suite comme expliqué sur le tableau 3.3.

- les communications représentent des envois de messages entre processeurs virtuels lorsqu'on veut faire autre chose que des interactions éléments par éléments dans une machine parallèle et correspondent aux calculs faisant intervenir des éléments de vecteurs qui ne se correspondent pas élément à élément, comme par exemple par l'intermédiaire des indirections vectorielles *scatter-gather* ;

---

<sup>2</sup>. « *Il n'est plus nécessaire d'avoir un stagiaire de DEA par processus à écrire...* » Petit mot que je dois à Jean-Paul SANSONNET, Luminy, octobre 1990.

- le *strip-mining* de la vectorisation est simplement un découpage des vecteurs du programme en sous-vecteurs de taille adaptés aux capacités du matériel, à savoir la taille maximale des vecteurs ou la taille des registres vectoriels et correspond à la notion de virtualisation sur une machine parallèle qui consiste à découper une variable vectorielle en *vp-ratio* sous-variables de taille compatible avec le nombre de processeurs de la machine physique.

## 3.2 Quelques modèles de programmation

On trouvera dans [Kar87] un aperçu des modèles de programmation couramment utilisés en programmation scientifique parallèle.

### 3.2.1 Le parallélisme de tâches

Il s'agit probablement du parallélisme le plus populaire et le plus répandu car à la base de systèmes d'exploitation multi-tâches comme UNIX ou introduit dans certains langages comme ADA [Tho86].

L'unité de parallélisme de base est la tâche, connue aussi sous le nom de processus. Plusieurs processus pouvant appartenir à différents utilisateurs, ils sont intrinsèquement indépendants et peuvent être exécutés en parallèle sur plusieurs processeurs ou sur plusieurs stations de travail par exemple.

Un autre type de parallélisme se développe grâce à l'apparition de machines multiprocesseur à mémoire partagée qui est l'exploitation du parallélisme à grain plus fin que les processus : chaque processus peut être composé de *processus légers* (*Light Weight Process*: LWP), pouvant s'exécuter sur différents processeurs tout en partageant la majorité des données, chaque LWP pouvant encore être divisé en *flot d'exécution* parallèles (*threads*) [PKB<sup>+</sup>91]. Cette hiérarchisation du parallélisme est dû au compromis entre le niveau du parallélisme et le poids de mise en œuvre du parallélisme : un processus est très indépendant alors qu'un *flot d'exécution* est très lié aux autres et son surcoût d'exécution est très faible, dans la mesure où on a une machine à mémoire commune.

On peut voir ici un moyen efficace d'extraire du parallélisme dans des applications à taux de parallélisme moyen comme on peut en trouver sur les stations de travail : par exemple un écran est composé de fenêtres indépendantes (processus) contenant chacune des boutons pouvant générer des actions en parallèles (threads), que l'on peut parfois redécouper en flots de calcul parallèles.

Un autre modèle, celui des processus séquentiels communicants (CSP) [Hoa78] met l'accent sur des tâches ne partageant que les entrées-sorties, un peu à la manière des « | » du *shell* d'UNIX. Néanmoins, ce modèle est assez sommaire (pas de garde de type `else` et surtout allocation statique des tâches, problèmes de syntaxe) et semble difficile à mettre en œuvre de manière efficace mais contient de bonnes idées (notion de communications explicites plutôt que d'avoir de communications implicites cachées dans des affectations).

Ce type de modèle semble bien adapté à des machines MIMD, à mémoire partagée (UNIX) ou à mémoire distribuée. Les machines parallèles à base de TRANSPUTERS ont consacré le modèle CSP à travers le langage parallèle OCCAM qui en est l'expression directe.

### 3.2.2 Le modèle flot de données

Il s'agit d'une adaptation du modèle précédent, plutôt prévu pour des petits processus, typiquement quelques opérations arithmétiques. On peut voir ce modèle comme un sous-ensemble du modèle de type CSP où on a des flots de données qui passent à travers plusieurs processus de traitement. Puisque le contrôle est de type producteur-consommateur, la synchronisation du système est faite par les données, d'où le nom du modèle.

Ce type de modèle s'adapte bien à des calculs répétitifs sur des jeux de données importants comme le traitement du signal, du calcul vectoriel comme les multiplications de matrices, la modélisation numérique, etc. comme on l'a vu en 2.2.5, donc plutôt à grain fin au niveau des opérations traitées.

Ce modèle semble bien adapté aux architectures systoliques, SIMD et MIMD, en fait à pratiquement toutes les machines, dans la mesure où on se restreint dans l'expression du parallélisme.

### 3.2.3 Le parallélisme de données

Connu aussi sous le nom de *data parallel* en anglais, *dataparallèle* en français<sup>3</sup>, il permet de préciser que certaines opérations vont s'exécuter sur plusieurs données à la fois, d'où le nom du modèle.

Ce modèle se prête particulièrement bien aux gros modèles numériques qui ont à faire des calculs sur plusieurs éléments d'un même problème, comme des modèles à éléments finis par exemple, car il conserve souvent le parallélisme intrinsèque de l'algorithme. Le parallélisme du modèle est explicite.

Malgré le côté parallèle, la vision du calcul est très locale, comme si chaque élément du problème était calculé sur un processeur différent. Cela facilite grandement les problèmes de compilation car la projection du modèle sur une machine réelle à couplage faible est triviale et seules restent les difficultés liées par exemple aux localités des communications dans le cas où elles ne sont pas explicites et que la machine est à couplage faible par exemple. Mais celles-ci sont souvent résolues suivant la loi de possession, l'« *owner computes rule* », qui conduit à faire les calculs là où le résultat doit être mémorisé, mais cela peut éventuellement ne pas être vérifié si on veut faire des optimisations plus poussées.

Selon les cas, l'espace des noms peut être global ou pas, c'est-à-dire que des éléments différents d'un vecteur peuvent interagir directement ou pas sans une opération de *communication* explicite, typiquement l'envoi de messages dans une machine MIMD à passage de messages. On peut aussi avoir un espace global de nommage mais encore conserver la notion de communication afin de bien mettre en valeur les endroits qui risquent de prendre du temps lors de l'exécution du programme.

Ce modèle est bien adapté à tous les types de machines, avec une préférence toutefois pour les machines SIMD et SPMD, mais aussi vectorielles puisque le modèle vectoriel est un modèle à parallélisme de données dont l'entité de calcul est le vecteur. Le choix de la machine cible est un choix d'ingénierie.

---

3. Cette traduction est on ne peut plus officieuse mais n'est pas plus choquante que l'étymologie de *télévision* qui vient respectivement du mélange du grec et du latin puisque *dataparallèle* vient du mélange du latin et du grec... Néanmoins, *ParDon*, un des termes issus des discussions du groupe  $C^3$  en rapport avec le sujet, semble une traduction beaucoup plus agréable.

### 3.2.4 Le modèle SPMD

Il s'agit d'un mélange des modèles précédents dont on espère bénéficier des avantages :

- on a une exécution de plusieurs tâches comme dans le modèle à parallélisme de tâches ;
- toutes les tâches sont identiques mais traitent des données différentes en parallèle, comme dans le modèle à parallélisme de données.

Ce modèle n'est en fait pas fondamentalement différent du précédent si ce n'est dans sa mise en œuvre, plus efficace si on dispose d'une machine optimisée pour ce modèle, et peut être vu comme une implantation particulière du modèle à parallélisme de données.

Le synchronisme n'est pas une propriété intrinsèque d'une architecture mais plutôt un choix d'ingénierie [Sny88], conséquence d'autres décisions. Le modèle de programmation peut être synchrone même si la machine est asynchrone. C'est d'un point de vue sémantique que l'on se place.

## 3.3 Un modèle de programmation pour POMP

L'étude des modèles existants nous a amenés à faire une série de choix pour le modèle de notre machine.

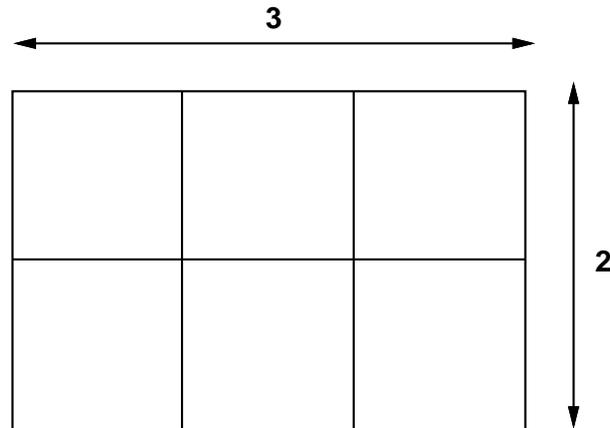
### 3.3.1 Parallélisme de données

Vues les applications visées par POMP, nous avons choisi de partir d'un modèle de programmation à parallélisme de données pour les raisons suivantes :

- beaucoup d'applications numériques et le graphisme en particulier nécessite le traitement de grandes quantités de données semblables en parallèle et ce parallélisme de données représente la majorité du parallélisme de ces applications, principalement parce que la plupart des applications sont basées sur une discrétisation de l'espace ;
- simplicité de mise en œuvre : le programme reste simple car sa structure de contrôle est bien définie et on manipule des grandeurs concrètes (vecteurs, morceaux d'espace, etc) ;
- langage impératif parce que le style de programmation est familier à la plupart des programmeurs contrairement aux langages fonctionnels<sup>4</sup> ;
- portabilité sur diverses architectures dans la mesure où ce qu'on demande correspond à un minimum que l'on retrouve sur la plupart des ordinateurs tout en offrant de bonnes performances [HQ91] ;
- typage fort du parallélisme et niveau d'abstraction suffisant qui facilite la compréhension d'un programme ainsi que la mise au point et l'évolution des programmes.

---

4. Ce qui est notre cas.

FIG. 3.1 - Une machine  $\equiv$  un tableau de processeurs.

### 3.3.2 Virtualisation

Il semble important de cacher certains aspects de la machine réelle sur laquelle est exécuté un programme, ne serait-ce que sa taille. Ce procédé est appelé *virtualisation* et est à considérer de la même manière que la mémoire virtuelle permet de cacher la quantité de mémoire physique réellement disponible sur une machine.

En effet un utilisateur n'a pas à réécrire un programme différent suivant le nombre de processeurs physiques de la machines ou la topologie de la machine. Même si la machine a par exemple  $2 \times 3$  processeurs (figure 3.1), c'est à dire qu'elle est composée d'un tableau bidimensionnel de processeurs, ce qui l'intéresse est la résolution de son problème qui est de taille tridimensionnelle  $a \times b \times c$  par exemple (figure 3.2) : il veut voir la machine comme un processeur de tableaux et pas du tout comme un tableau de processeurs. C'est la dualité de vision entre l'architecte et l'utilisateur qu'il faut contourner.

On doit donc être capable d'accepter des tableaux d'un nombre quelconque de dimensions, chacune pouvant avoir une taille quelconque. On ne garantit pas que toute la machine sera utilisée<sup>5</sup> mais par contre que le programme fonctionnera comme prévu par le modèle de programmation.

L'unité de parallélisme est donc le tableau et les calculs s'effectuent sur un tableau de même taille de *processeurs virtuels*, par analogie avec la mémoire virtuelle.

Un autre niveau de virtualisation est parfois nécessaire, même si on a tendance à l'oublier : c'est celui de la taille des processeurs. Il ne suffit pas d'en avoir suffisamment, il faut qu'ils soient suffisamment larges pour être adaptés à la taille des données à traiter. C'est d'autant plus important que bon nombre de machines SIMD à grain fin sont utilisées comme des supercalculateurs travaillant sur des entiers ou flottants 32, voire 64 bits. Néanmoins, ce problème sera supposé résolu à un autre niveau par la suite : on génèrera du code pour processeurs virtuels à gros grain, laissant la virtualisation de la largeur des PES à un niveau logiciel inférieur.

---

5. En particulier si le nombre d'éléments du tableau est inférieur au nombre de processeurs physiques de la machines ou s'il y a des problèmes d'alignement de dimensions virtuelles sur des dimensions physiques.

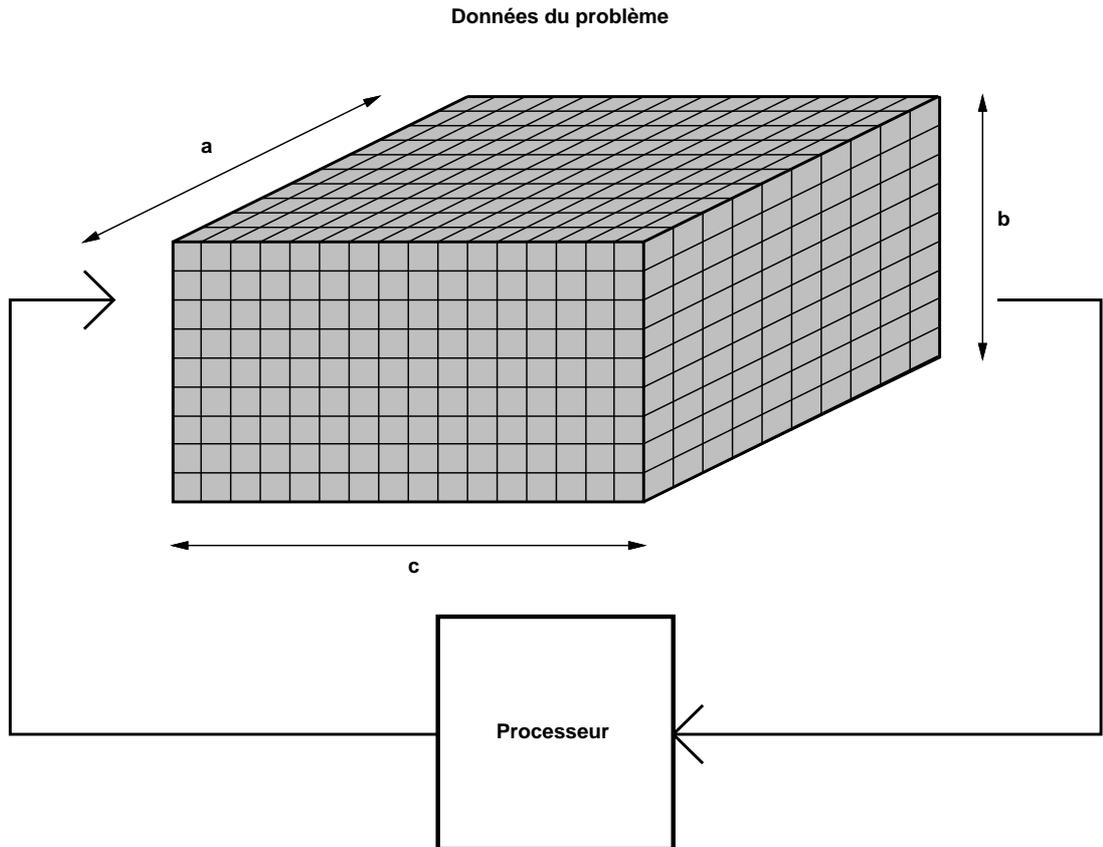


FIG. 3.2 - Résoudre un problème  $\equiv$  avoir un processeur de tableaux.

En plus il existe des cas où on veut travailler sur des processeurs réels pour des raisons d'efficacité algorithmiques, telles les routines d'algèbre linéaire comme les multiplications de matrices [HQ91, page 149] ou des opérations de type parallèle préfixe [KRS85] (voir la section 5.1.1.2).

### 3.4 Parallélisme orienté objet

Le parallélisme de donnée est vu dans notre modèle comme un typage fort du parallélisme [Kil91] qui assure une programmation robuste des algorithmes : alors qu'en FORTRAN on manipule des tableaux dont on ignore tout, qu'on peut mélanger n'importe comment, laissant la liberté au programmeur de rendre obscur à loisir son programme<sup>6</sup>, on choisit d'empêcher la possibilité d'ajouter des carottes à des lapins, du moins facilement, à travers le typage. Chaque classe de données parallèles est une *collection* [Ble89a, BS90]. La programmation se fait donc en pensant à des objets qui interagissent élément par élément plutôt qu'en tableaux qu'il faut gérer à la main. Notons que le

6. On possède un exemple de programme où le programmeur a choisi de n'avoir qu'un seul tableau qu'il a subdivisé en plusieurs parties ayant un sens totalement différent. On comprend alors que la parallélisation puisse être une tâche difficile... Malheureusement pour certains, ce programme fait partie d'un test de paralléliseurs automatiques !

parallélisme orienté objet est indépendant de la nature objet du langage séquentiel servant de base. Ainsi on peut très bien avoir du parallélisme orienté objet basé sur LISP [BS90] ou C.

Un des autres avantages du typage est que les modules de code écrits peuvent être réutilisés directement pour d'autres types de données parallèles permettant un certain polymorphisme du parallélisme.

Contrairement à [Kil91], on n'a pas choisi de proposer une imbrication de types de parallélisme car la projection sur la machine physique n'est pas triviale, surtout si elle est SIMD. Par contre on peut faire la même chose en créant une collection multiple de la précédente possédant les mêmes dimensions plus une nouvelle. Le programmeur pourra jouer sur les préférences géométriques afin de préciser les rapports entre les 2 collections dans l'algorithme (notion d'*alignement*) et avoir un meilleur placement des données sur la machine cible.

## 3.5 Contrôle de flot

### 3.5.1 Contrôle de flot scalaire

Il ne semble pas nécessaire d'avoir un contrôle de flot scalaire différent de ceux des modèles d'exécution classique : le code scalaire inclut des manipulations d'objets parallèles mais le contrôle reste le même. Le modèle d'exécution est très simple et on conserve le contrôle de flot du langage scalaire support.

Le contrôle de flot sur une condition globale calculée à partir de données parallèles ne nécessite pas d'instruction supplémentaire : le test séquentiel est fait normalement sur la valeur scalaire calculée par une fonction de bibliothèque, par exemple.

### 3.5.2 Le contrôle de flot parallèle : une dessynchronisation bornée

Par contre il est nécessaire d'introduire de nouveaux principes pour permettre des changements dans l'exécution du code parallèle. En effet il est souvent nécessaire de ne pas effectuer le même traitement sur tous les éléments d'un objet parallèle, suivant la valeur de chaque élément, par exemple.

Néanmoins, dans la philosophie du langage à parallélisme de données, on restreint les modifications possibles localement afin de conserver un état global déterministe de la machine. On n'autorise pas en particulier une scission du programme en plusieurs parties très indépendantes qui communiquent chacune de leur côté.

Dans le cas d'une machine physique MIMD, on assistera à une dessynchronisation bornée, seulement, car les processeurs se resynchroniseront aux endroits nécessaires pour respecter les dépendances entre les données du problème et ainsi conserver la sémantique du programme.

Ce contrôle de flot est assuré par de nouveaux opérateurs qui contrôlent dans un bloc d'instructions l'exécution des opérations sur les données de la collection à laquelle appartient la condition.

### 3.6 Interactions complexes = communications

Dans notre modèle, on choisit de différencier les interactions entre tableaux d'une même collection élément par élément des autres interactions, à savoir les interactions non- $\alpha$  ou les interactions entre tableaux de collections différentes par exemple. Dans ce cas, on a affaire à ce que l'on nomme des *communications*, ainsi nommées car ce type d'interaction nécessite souvent des communications de données entre processeurs différents.

En cela, on n'offre donc pas au programmeur un espace des noms de variable totalement général comme en FORTRAN 90 par exemple. Néanmoins, alors que ces différences pourraient être cachées par le modèle de programmation en offrant un espace des noms global, il nous semble important de l'exprimer différemment pour plusieurs raisons :

- cela fait partie du typage : une interaction d'objets de natures différentes a bien évidemment une sémantique différente d'une interaction d'élément à élément et donc doit être gardée afin d'éviter les erreurs de programmation, surtout dans le cas d'une émission avec collision ;
- une communication coûte souvent beaucoup plus cher qu'une interaction directe, même sur les machines à mémoire commune car celle-ci n'est pas accédée de manière optimale (à savoir par blocs consécutifs) ;
- le programmeur est sensibilisé au problème ci-dessus et reste en contact avec la dure réalité de la vie, évitant ainsi d'écrire des programmes totalement inefficaces ;
- on ne veut pas que le compilateur génère des communications sans en avertir l'utilisateur, provoquant insidieusement une mauvaise utilisation de la machine.

On pourrait différencier beaucoup de types de communications comme ne pas en différencier du tout. Nous avons choisi d'en différencier plusieurs *a priori*, sachant que certaines communications non proposées dans la suite peuvent être disponibles dans des bibliothèques, écrites à partir des communications suivantes (comme par exemples les opérations préfixe, comme les *scans* qui ne sont généralement pas réalisées grâce à un matériel particulier).

#### 3.6.1 Communications générales

D'abord le type de communications dont on ignore tout ou dont le programmeur sait qu'elles sont irrégulières. Dans ce cas, il n'y a guère d'optimisations à faire, si ce n'est, peut-être, précompiler certaines communications répétitives lorsqu'on le peut et que la communication est connue avant exécution.

Ces opérations correspondent aux fonctions d'indirection que l'on trouve sur les machines vectorielles sous le nom de *scatter* et *gather* et sur les machines parallèles sous le nom de *send* et *get*<sup>7</sup> qui correspondent respectivement aux émissions et aux réceptions représentées sur la figure 3.3.

---

<sup>7</sup> On notera au passage l'aspect mnémotechnique de ces 2 paires de mots que l'on peut associer suivant leur première lettre.

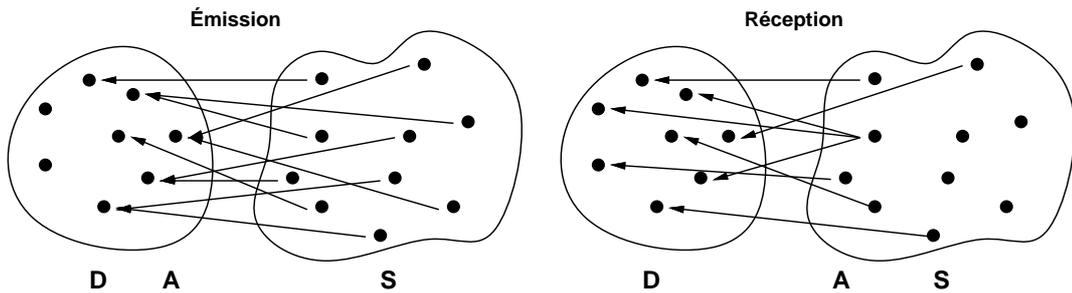


FIG. 3.3 - Les deux types de communications générales.

### 3.6.1.1 Émissions

Cette opération, correspondant au *scatter* vectoriel et au *send* parallèle, consiste à envoyer les éléments d'un vecteur source  $S$  dans un vecteur de destination  $D$  aux places indiquées par un vecteur d'adresse  $A$  selon le schéma suivant

$$\forall i, D_{A_i} = S_i$$

et peut être vue comme une généralisation de l'écriture indirecte en mémoire, au niveau de la machine complète.

Cette opération étant clairement surjective (il suffit qu'il y ait  $i \neq j : A_i = A_j$ ), comme un élément ne peut avoir qu'une valeur, en cas de conflit le choix de la valeur est non spécifié *a priori*. En outre, plusieurs écritures dans un même élément de vecteur ne peuvent se faire que de manière séquentielle (modèle EW des PRAMS) en général, ce qui est à éviter algorithmiquement, sauf cas particulier de machines à réseau recombinaut par exemple [GGK<sup>+</sup>80, Got84].

Une manière de résoudre cela peut être d'utiliser un opérateur associatif pour combiner plusieurs valeurs, si cela a une signification algorithmique intéressante comme on va le voir en 3.6.3.

### 3.6.1.2 Réceptions

À l'opposé, cette opération, qui correspond au *gather* vectoriel et au *get* parallèle que l'on peut voir comme une lecture dans une hypothétique mémoire globale, consiste à récupérer les éléments d'un vecteur source  $S$  aux places indiquées par un vecteur d'adresse  $A$  pour les mettre dans un vecteur de destination  $D$  tel que

$$\forall i, D_i = S_{A_i}$$

Physiquement, cela ne peut se faire qu'en 2 opérations d'émissions la première où la destination envoie son adresse à la source qui peut ensuite par une émission retour lui renvoyer la valeur demandée<sup>8</sup> et par conséquent une réception coûte le double d'une émission, ce dont on doit tenir compte lors de la conception d'un algorithme.

De même que l'émission, plusieurs requêtes à un même élément sont traitées séquentiellement, avec la même remarque sur les réseaux recombinauts, et est donc plutôt à éviter dans la mesure du possible (modèle ER des PRAMS).

8. Si on disposait d'une architecture exécutant les `come from` de manière aussi efficace que les `go to`, on pourrait faire une machine où la réception serait aussi rapide que l'émission...

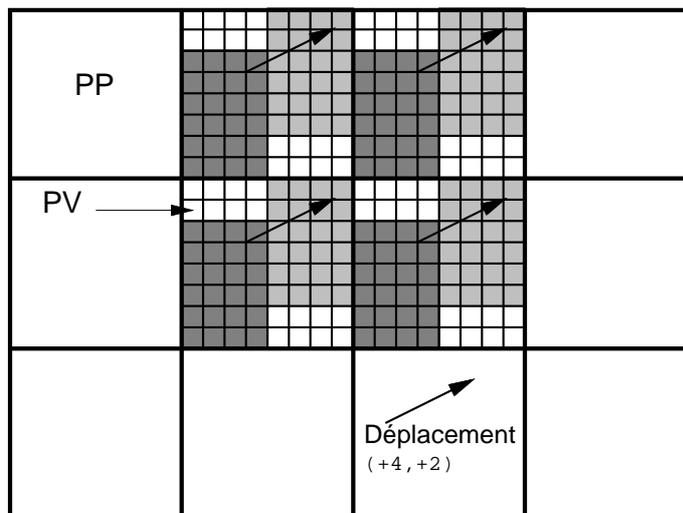


FIG. 3.4 - Exemple de communication où des données dans des processeurs virtuels (PV) en gris foncé restent dans les processeurs physiques (PP).

Mais contrairement à l'émission, il n'y a toujours au plus un élément écrit dans la destination puisque chaque élément de la destination n'envoie au plus qu'une requête. Par conséquent, cela n'est pas la peine de prévoir un opérateur associatif associé aux réceptions.

### 3.6.2 Communications sur grille

Par contre, il nous semble indispensable de différencier les communications sur grille, émissions aussi bien que réceptions, et d'avoir un moyen de les exprimer très simplement, même si les communications générales font bien entendu l'affaire, pour les raisons suivantes :

- comme c'est un motif de communication très courant lié au fait que nos objets de base sont des tableaux multidimensionnels, il faut proposer à l'utilisateur une expression simple de celui-ci plutôt que de lui laisser le calcul d'adresse de la communication ;
- parce que de nombreuses machines proposent un réseau à base de grille multidimensionnelles (dont l'hypercube est un exemple) et par conséquent ce réseau est utilisé au mieux pour ce type de communications. Puisque ce genre de communications apparaît souvent dans les programmes et est utile, autant l'exprimer directement plutôt que d'écrire un compilateur capable de deviner que l'on va faire une communication sur grille ;
- ensuite parce que même si le réseau sous-jacent de la machine ne permet pas des communications sur grille rapides, on donne à la machine une information intéressante en cas de virtualisation. En effet, si on a besoin d'un déplacement suivant un vecteur compris dans la maille de virtualisation, la routine de communication sur grille est capable de ne communiquer que pour les processeurs virtuels « sur

les bords », alors qu'elle se contente de ne faire que du transfert mémoire local très rapide pour ceux de « l'intérieur<sup>9</sup> », comme indiqué en grisé sur la figure 3.4 ;

- enfin, cela clarifie le programme en augmentant sa lisibilité : le fait que le programmeur a écrit par exemple un gradient, une divergence ou un laplacien apparaîtra immédiatement par son motif de communication typique. Ainsi la discrétisation du laplacien à l'ordre 1 en 2 dimensions sur une grille régulière s'exprime comme :

$$\Delta_{i,j}U = \left(\frac{\partial^2 U}{\partial x^2}\right)_{i,j} + \left(\frac{\partial^2 U}{\partial y^2}\right)_{i,j} \approx \frac{U_{i+1,j} + U_{i-1,j} - 2U_{i,j}}{\delta_x^2} + \frac{U_{i,j+1} + U_{i,j-1} - 2U_{i,j}}{\delta_y^2}$$

motif très régulier dont on a tout intérêt à conserver le plus possible d'information.

Pour cette raison il semble important d'être capable d'exprimer cette sous-classe des communications générales de manière spécifique dans notre modèle.

Il se peut qu'on soit amené à utiliser un motif de communication très courant qui est réalisé de manière performante sur une machine donnée. Dans ce cas il peut être intéressant de pouvoir conserver de l'information sur la régularité de ce motif pour que le compilateur soit capable de reconnaître le motif et le compiler de la meilleure manière si la machine cible dispose de matériel spécialisé pour ce motif. En particulier on peut être amené à différencier pour des algorithmes de FFT par exemple des communications sur hypercube, des *shuffle*, etc.

### 3.6.3 Concentrations ou réductions associatives

Par exemple on peut être amené à calculer un vecteur où chaque élément est calculé à partir de la somme des éléments de chaque ligne d'une matrice. Plutôt que de mettre une boucle pour recevoir chaque élément et additionner les éléments reçus, il est beaucoup plus puissant d'introduire un opérateur de réception associative, ici le « + » et d'envoyer tous les éléments d'une ligne vers un élément du vecteur résultat.

De même il est souvent nécessaire de calculer une valeur globale à tous les éléments d'une variable parallèle, ou au moins à plusieurs éléments de celle-ci, comme le montre la figure 3.5.

Le modèle, en plus de la réception standard, doit donc inclure la réception associative, aussi nommée concentration ou réduction associative, permettant de retrouver un peu de la puissance et de l'expressivité savoureuse du langage APL [Ive62, Ber91].

Sur une machine séquentielle, on calcule ces réductions avec une boucle qui impose un ordre d'évaluation, par exemple selon l'ordre de précedence classique de gauche à droite sur le dessin de gauche de la figure 3.6, ce qui demande une complexité<sup>10</sup> en temps de  $\mathcal{O}(n)$  pour une variable parallèle de taille  $n$ .

9. Il est intéressant de noter que cette notion reste valable même lorsqu'on n'a que des approximations de grilles, comme dans le cas de modèles à volumes finis avec des maillages adaptatifs formant localement une sorte de grille dont on peut exploiter la localité.

10. Les personnes non familières avec ce genre de notations pourront se reporter à [Tar83, pages 1–7]. Pour résumer, on peut dire que si  $f$  et  $g$  sont des fonctions de variables positives  $(v_1, \dots, v_n)$  de l'ensemble de définition  $\mathcal{D}$  et «  $f = \mathcal{O}(g)$  » ou «  $f$  est  $\mathcal{O}(g)$  » alors

$$\forall (v_1, \dots, v_n) \in \mathcal{D}, \exists (c_1, c_2) \in \mathbb{R}^2 : f(v_1, \dots, v_n) \leq c_1 g(v_1, \dots, v_n) + c_2$$

De même la notation réciproque existe :

$$f = \Omega(g) \iff g = \mathcal{O}(f)$$

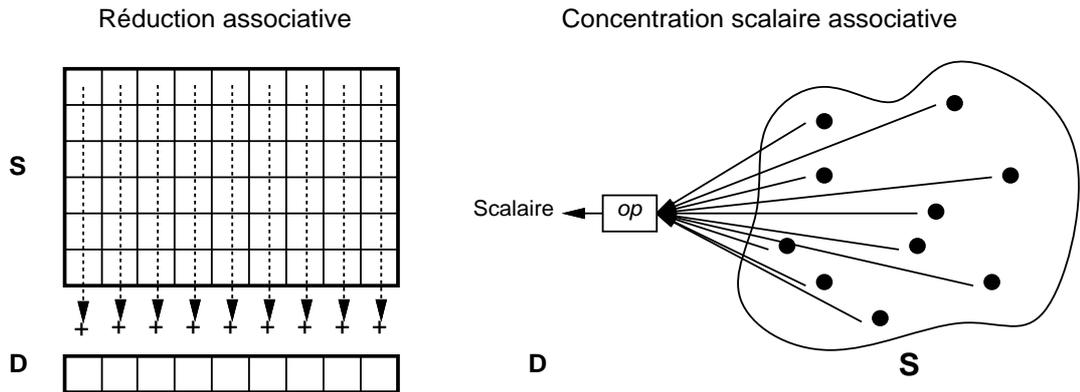


FIG. 3.5 - Concentrations ou réductions associatives.

Sur une machine parallèle, on a tout intérêt à faire le calcul suivant un arbre afin de gagner du temps comme le montre le dessin de droite de la figure 3.6 : si le nombre d'opérations reste inchangé, le temps d'exécution n'est plus que  $\mathcal{O}(\log n)$  [Kuc68]. Remarquons que si la machine a  $n$  processeurs, seules  $n - 1$  opérations seront exécutées sur les  $\mathcal{O}(n \log n)$  possible pendant le temps  $\mathcal{O}(\log n)$ , ce qui montre bien qu'on n'utilise que  $\mathcal{O}(1/\log n)$  de la machine. C'est une des caractéristique du parallélisme : il faut souvent être amené à gaspiller des processeurs pour gagner du temps. Bien entendu il faut trouver à ce niveau un compromis entre le nombre de processeurs et le nombre de données (voir la section 5.1.1.2 et [KRS85]), ce qui se traduit en rapport coût sur performance [KS73, Kog74b, Kog74a, LF80, Kog81].

On constate en comparant les 2 dessins de la figure 3.6 que pour des raisons d'efficacité, on ne garantit plus aucun ordre d'exécution des opérateurs de réduction, ce qui peut être gênant si les opérateurs ne sont pas tout à fait associatifs<sup>11</sup>. La méthode séquentielle et parallèle peuvent donc ne pas donner le même résultat.

Outre la différence entre méthode séquentielle et méthode parallèle, il faut voir que selon la machine, principalement en fonction de son réseau de communication, le calcul ne sera probablement pas non plus effectué dans le même ordre, donc il faut s'attendre aussi à avoir des résultats différents suivant les machines, en plus des différences liées

---

et l'équivalence :

$$f = \Theta(g) \iff (f = \mathcal{O}(g) \text{ et } f = \Omega(g))$$

qui est tout de même moins forte que la notation d'équivalent  $\sim$  qui précise la constante  $c_1$ .

On remarquera bien sûr que ces considérations sont asymptotiques et que les théoriciens ne s'embarrassent pas des constantes, ce qui peut être fâcheux dans la vie réelle : on peut avoir un algorithme asymptotiquement complexe mais dont les constantes font qu'en pratique il est plus efficace qu'un algorithme *a priori* plus simple. Des considérations statistiques sur les cas moyens et les cas pires peuvent aussi amener à de tels inversions. Il faut donc considérer ces estimations parfois avec prudence dans la « vie réelle » non asymptotique.

11. En effet, si les opérateurs sont associatifs mathématiquement, ce n'est pas toujours le cas lorsqu'on les réalise concrètement, tout particulièrement en ce qui concerne les nombres flottants, pour des raisons de précision. Par exemple, si on veut réduire additivement le vecteur  $(10^{20}, -10^{20}, 1)$  et qu'on le fait de gauche à droite, on trouvera bien 1. Par contre si on fait le calcul en commençant par le premier et le dernier élément, la somme partielle sera  $10^{20}$  car la dynamique de la mantisse d'un nombre flottant IEEE 64 bits étant d'environ  $4,5 \cdot 10^{15}$  (52 bits), le deuxième terme sera négligé et par conséquent la réduction additive du vecteur complet sera 0 au lieu de 1...

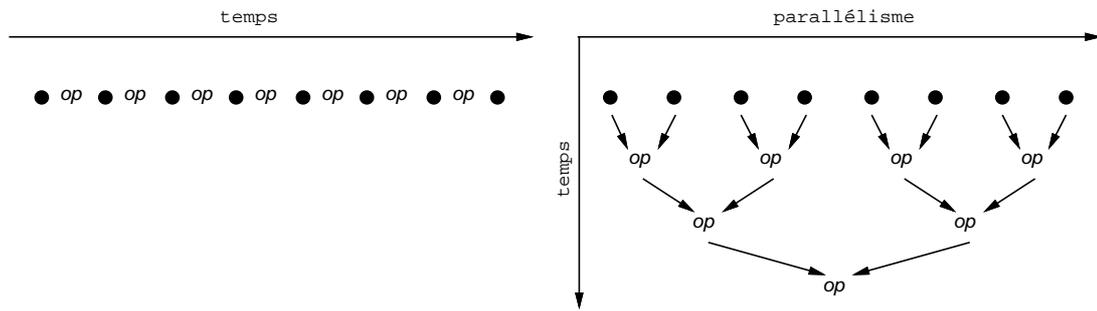


FIG. 3.6 - Comparaison entre une réduction exécutée séquentiellement ou parallèlement suivant un arbre binaire.

aux différents opérateurs flottants.

Le problème devient carrément pathologique lorsqu'on veut récupérer dans un tableau la copie de la réduction scalaire par exemple d'un autre tableau. Si on le fait de manière parallèle et arborescente entre tous les processeurs afin d'éviter d'avoir à passer par un processeur centralisant la valeur de la réduction, chaque calcul aura été fait dans un ordre différent lié à la topologie physique de la machine et pourra donner sur chaque processeur une valeur différente alors que la sémantique impose une même valeur dans tous les éléments de la destination, sensés contenir la copie d'un scalaire unique ! Une solution simple est de calculer une seule fois la réduction de manière centralisée puis d'envoyer le résultat afin de garantir une valeur identique pour tous les éléments de la destination. Si on a un réseau en hypercube, on peut néanmoins trouver des évaluations parallèles faites dans le même ordre qui peuvent éviter une centralisation du résultat. Reste à voir si pratiquement sa mise en œuvre est plus rapide que la méthode centralisée.

Même si ce dernier problème est résolu, il reste celui de la différence entre l'évaluation séquentielle et l'évaluation parallèle. Mais il faut bien voir qu'un programme qui a besoin d'un ordre d'évaluation de la réduction est soit instable numériquement, soit très spécifique et dans ce cas on peut bien dérouler explicitement le code de manière scalaire, seule manière de garantir l'ordre d'évaluation. Néanmoins, il s'agit là d'une entorse au but avoué au début, à savoir une sémantique indépendante de la machine cible. Mais il semble que ce soit la seule entorse à la sémantique. De toute manière, tout programme qui se base sur des effets aussi peu portables que l'imprécision numérique est probablement un programme mal conçu à la base ou nécessite tout au moins un minimum de prudence lors de sa mise en pratique.

Une solution raisonnable semble être par exemple les réductions effectuées par valeurs positives croissantes pour limiter le bruit numérique. Une méthode pour ce faire peut être de faire un tri sur les valeurs puis de lancer une réduction suivant un arbre binaire pour limiter la perte de précision.

### 3.6.4 Communications scalaires

Un cas très intéressant à inclure est la possibilité d'envoyer une valeur scalaire à certains ou à tous les éléments d'un tableau, ne serait-ce que pour l'initialisation des données. Ce dernier mécanisme est appelé diffusion scalaire ou *broadcast*.

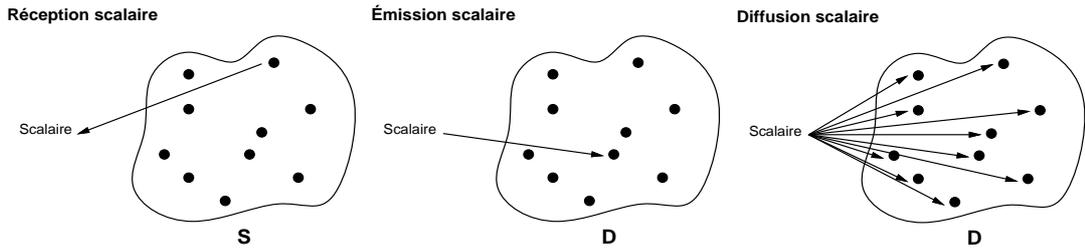


FIG. 3.7 - Les types de communications scalaires.

Il faut bien entendu aussi considérer le cas inverse : la récupération de la valeur d'un seul élément parallèle dans un scalaire, si on veut écrire le contenu d'une variable parallèle dans un fichier séquentiel, élément par élément.

Ce type de communications est résumé sur la figure 3.7.

Enfin il faut prévoir la réduction ou concentration associative d'une variable parallèle en un scalaire, permettant par exemple de calculer un produit scalaire (grâce à la somme) ou une condition globale (à travers un *ou global*) de convergence sur un vecteur par exemple. Ce type particulier de réduction associative est indiqué sur le dessin de droite de la figure 3.5.

### 3.6.5 Opérations parallèles préfixes

Ce dernier type d'opérations, qui correspond en gros à un mélange de communications sur grille et de concentration associative partielle, nous semble très utile car il constitue un des piliers de la puissance algorithmique importante d'APL et peut être utilisé pour résoudre de nombreux problèmes n'ayant pas *a priori* de solution parallèle évidente.

L'application directe en calcul numérique concerne tous les domaines ayant à résoudre des récurrences linéaires portant sur des opérateurs associatifs, du style :

$$\begin{aligned} x_1 &= A_1 x_0 + B_1 \\ &\vdots \\ x_i &= A_i x_{i-1} + B_i, \quad \forall i > 1 \end{aligned}$$

La méthode a été développée initialement pour permettre de dépasser les relations de dépendances empêchant la vectorisation du calcul de ces récurrences [Kog74a, Kog81], au prix d'une modification de l'ordre d'évaluation.

La méthode peut être utilisée pour résoudre des systèmes linéaires tridiagonaux car on peut mettre la résolution de ces derniers sous forme de récurrence linéaire [Sto73, Hel76]. Comme l'ordre d'évaluation est modifié, il vaut mieux qu'elle soit à diagonale dominante pour garantir une stabilité raisonnable à la méthode.

Les opérateurs préfixes parallèles sont généralisables à des récurrences possédant des opérateurs non associatifs dans la mesure où elles peuvent se mettre sous la forme

$$x_1 = b_1 x_i = f_i(x_{i-1}) = f(a_i, g(a_i, x_{i-1})), \quad \forall i > 1$$

avec  $f$  qui est associative,  $g$  qui est distributive sur  $f$  et  $g$  est semi-associative, c'est-à-dire qu'il existe une fonction  $h$  telle que  $g(x, g(y, z)) = g(h(x, y), z)$  [KS73, Kog74b].

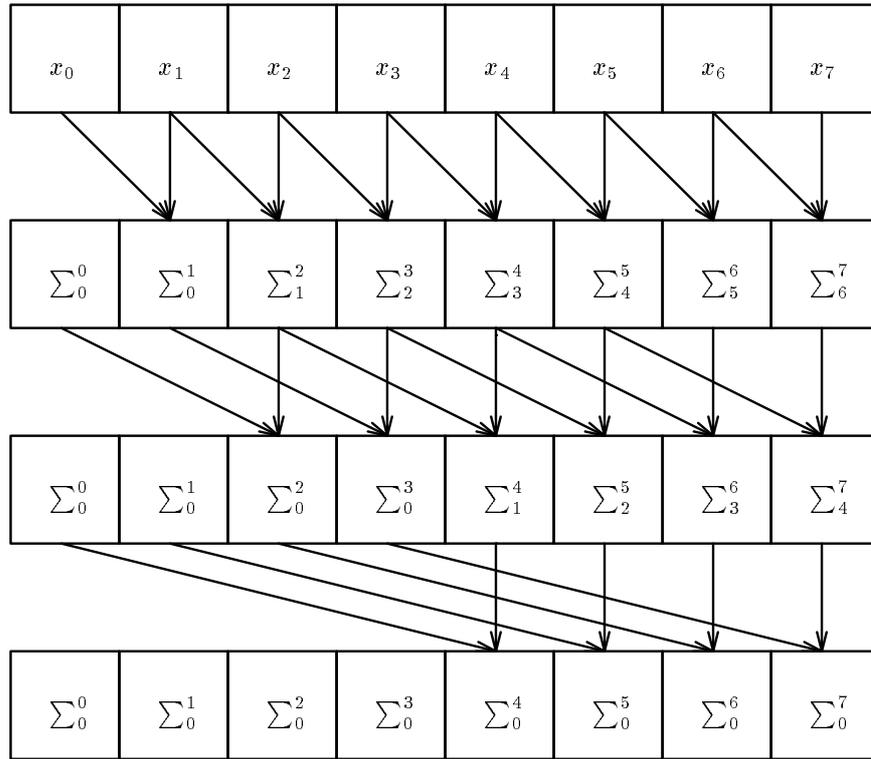


FIG. 3.8 - Addition préfixe parallèle.

Même si l'opérateur n'est pas associatif mais que l'ensemble  $\mathcal{E}$  des valeurs sur lesquelles agit l'opérateur est fini<sup>12</sup>, la composée des opérateurs  $f_j$  dans

$$f_i(f_{i-1}(\dots f_2(f_1(x_0)) \dots))$$

est en soit associative et devient une opération exécutable en parallèle

$$f_i \circ f_{i-1} \circ \dots \circ f_2 \circ f_1(x_0)$$

qui est seulement ensuite appliquée à  $x_0$  puisque la composition d'applications est associative et est facilement calculable si  $\mathcal{E}$  est petit [RJH92].

Le schéma d'une méthode logarithmique dans le cas très simple où on veut calculer

soit encore

$$\forall i \in [0, n - 1], \quad S_i = \sum_{j=0}^i x_j$$

est indiqué sur l'illustration 3.8 suivant l'algorithme du *doublage récursif* [Kuc68, Kog74b, Sto73, KRS85] avec comme convention d'abréviation  $\sum_a^b \equiv \sum_{j=a}^b x_j$ . On remarque que certains calculs sont faits plusieurs fois ce qui n'est pas gênant sur une

<sup>12</sup>. Ce qui signifie en terme d'informatique pratique et réaliste que son cardinal est faible...

machine parallèle où les processeurs peuvent faire quelque chose plutôt que de rester inactifs. Mais il existe aussi d'autres algorithmes optimisant le nombre de calcul ou qui sont mieux adaptés que le précédent, inspiré des additionneurs de type *carry look ahead*, lorsque le réseau de la machine ne permet pas des décalages sur grilles efficaces par des puissances de 2.

Une variation utile des opérations parallèles préfixes est la segmentation : en plus de la variable source on fournit une variable booléenne parallèle qui indique pour chaque élément de la variable source s'il sera à nouveau un début d'opération. On peut donc considérer qu'il y a autant d'opérations parallèles préfixes indépendantes qui ont lieu que de bit à 1 dans la variable booléenne de segmentation [Ble89a, Ble89b].

Mais les opérations préfixes parallèles, en particulier segmentées, peuvent servir à des choses beaucoup moins numériques et parfois « typiquement sérielles » telles que :

- des tris tels que le *radix sort* [HS86], le *quick sort* [Ble89b] ;
- des algorithmes numériques comme la factorisation *LU* ou la méthode du simplexe [Sto73, Ble89a] ;
- de l'analyse syntaxique [HS86] ;
- du tracer de segments sur un écran [Ble89b] ;
- calcul sur des listes chaînées, problème qui semble pourtant séquentiel *a priori* [KRS85, HS86] ;
- des comparaisons de listes chaînées [HS86] ;
- de la recherche de composantes connexes en traitement d'image [HS86] ;
- des calculs sur des graphes comme la détermination d'arbre recouvrant minimal [Ble89b].

En ce qui concerne tous ces calculs, il faut faire attention que dans la pratique un ordinateur physique se rapproche plus d'une PRAM EREW que CREW, *ie* on ne peut lire qu'une valeur à la fois sur chaque processeur, ce que ne suppose pas tous les algorithmes précédents, particulièrement ceux qui concernent les listes, et cela peut faire apparaître des sérialisations parasites.

Néanmoins, devant les possibilités offertes par ces opérateurs, la construction de machines incorporant directement ces opérateurs a été étudiée [Ble89b] et réalisée dans le cas de la CM-5 pour les opérandes entiers [Thi91]. Mais des opérations de récurrence avaient aussi été incluses dans la machine BSP bien avant [KS82] et logiquement sur l'ILLIAC IV [Kuc68].

Il nous semble utile de proposer, outre les primitives de base agissant sur des types habituels de données vectorielles :

- bien sûr des primitives agissant sur certaines dimensions seulement de données parallèles organisées en tableaux, très utile par exemple pour faire de l'étiquetage de composantes connexes dans des images ;

- mais aussi des opérateurs plus généraux acceptant des types de données définis par l'utilisateur avec une fonction associative arbitraire. Une application en est par exemple la généralisation de la récurrence du premier ordre à l'ordre  $n$  en se ramenant tout simplement, selon une méthode classique concernant l'étude des suites ou des équations différentielles, à une récurrence linéaire du premier ordre dans un espace vectoriel d'ordre  $n$  et en raisonnant sur un vecteur  $(y_i, \dots, y_{i+n-1})$ . Ce type d'opérateur préfixe parallèle généralisé correspondant au `\` en APL doit par exemple accepter de faire des opérations sur des variables parallèles dont les composants sont des matrices de dimension  $n$  et les opérations associatives des additions et multiplications matricielles. En fait, comme dans le cas particulier précédent les matrices sont très creuses, cela rajoute encore de l'intérêt à ce que ce soit l'utilisateur qui fournisse ses propres opérateurs.

### 3.7 Conclusion

Nous nous sommes intéressés surtout au modèle impératif à parallélisme de données car il nous semble actuellement le plus à même de conceptualiser les algorithmes numériques habituels, comparés aux modèles à parallélisme de contrôle, fonctionnels ou à flot de données<sup>13</sup> [Kar87].

Même si les langages fonctionnels ne souffrent pas des problèmes de synchronisation, la lourdeur de leur implémentation ne les rend souvent pas assez compétitifs au niveau des performances par rapport à des langages synchrones sur des problèmes parallèles. c'est pourquoi, certains y ajoutent un modèle à parallélisme de données impératif [OPP91] pour décrire les parties numériques parallèles de l'algorithme, ce qui nous semble être une approche très intéressante. Mais nous avons écarté une telle approche du fait de notre culture plutôt « impérative ».

Un modèle à parallélisme de donnée ou de type SPMD nous semble donc convenir à la manière de penser une grande classe d'algorithmes :

- trame de contrôle séquentielle décrivant le déroulement temporelle d'un algorithme de manière simple ;
- parallélisme de données permettant de manipuler directement des données dont le parallélisme est adapté à chaque problème considéré ;
- simplicité pratique et formelle du modèle combinant à la fois une sémantique claire et une écriture efficace ;
- modèle synchrone facilitant la mise au point des programmes.

Bien évidemment, ce modèle ne résout pas en lui-même tous les problèmes de répartition des données qui sont une des difficultés majeures du parallélisme [Kar87]. Ce problème sera résolu, faute de mieux, par un placement explicite au niveau du langage, comme nous allons le voir.

Maintenant que nous avons vu quels sont les constituants de notre modèle de programmation, on va s'intéresser à l'intégration de ceux-ci dans un langage utilisable sur notre machine.

---

13. J'espère que tous ceux qui travaillent dans ces domaines ne se sentiront pas offensés... Il s'agit juste d'une question de domaine d'application qui est considéré ici.

# Chapitre 4

## Le langage : POMPC



**S**E chapitre aborde brièvement les différentes approches prises dans le développement d'environnements de programmation à parallélisme de données pour machines parallèles et présente rapidement quelques langages de programmation avant de donner un aperçu du langage POMPC, nécessaire pour aborder les choix architecturaux et logiciels ultérieurs du projet POMP.

## 4.1 Les langages de programmation parallèle

On constate que les langages de programmation qui ont été implantés sur des machines parallèles dans le domaine d'application scientifique suivent principalement deux stratégies :

- soit le langage est purement séquentiel et le programmeur laisse au compilateur la totale extraction du parallélisme à partir du programme, écrit typiquement en FORTRAN 77 ;
- soit la syntaxe du langage reflète le parallélisme de l'architecture de la machine cible et reste plutôt proche d'un langage d'assemblage : typiquement CPARIS sur la Connection Machine ou le microcode de GF11 enrobé dans du PASCAL [BDW85].

Heureusement, un troisième choix existe où le programme et les structures de données du langage permettent d'exprimer le parallélisme intrinsèque de l'algorithme sans avoir à se perdre dans des considérations architecturales précises.

L'introduction d'un langage qui contient des directives traitant du parallélisme est intéressant pour plusieurs points que l'on a déjà vu dans le chapitre 3 :

- il évite au programmeur de décrire explicitement les calculs vectoriels au niveau de chaque élément de chaque vecteur qui n'apportent absolument rien à la description ou la compréhension de l'algorithme ;
- cela clarifie le programme et limite les risques d'erreurs et d'usure prématurée de claviers par manque de concision ;
- il fournit une indication explicite du parallélisme qui est directement exploitable par le compilateur ;
- cela simplifie la conception de l'environnement de programmation.

Dans de nombreux projets de machines parallèles, la nécessité d'avoir une transcription automatique avec extraction de parallélisme de programmes écrits dans des langages séquentiels pour machines séquentielles a souvent sonné le glas des projets à cause de la complexité et l'ampleur de la tâche, ce que nous voulons éviter. Cela milite vers le parallélisme explicite.

En plus, si l'on peut avoir des programmes qui sont écrits dans un langage intermédiaire qui soit portable sur la plupart des machines parallèles, l'exécution sur une nouvelle machine possédant un compilateur de ce langage ne posera pas de problèmes majeurs [Per79].

On peut dégager 2 types de langages de haut niveau pour le parallélisme de données en fonction de l'étendue du typage des objets qu'il manipule [Kil91] :

- les langages de gestion de tableaux de type FORTRAN 90 ou APL, qui traitent des vecteurs qui n'ont qu'à être conformants, voire même des vecteurs de tailles différentes [Mar92b] ;
- les langages à parallélisme de données un peu orienté objet qui traitent des données parallèles de même type, comme MPL ou C\*, donc à parallélisme typé mais qui offrent tout de même souvent<sup>1</sup> des coercitions au niveau du parallélisme pour faire interagir différentes classes de variables parallèles selon l'algorithme.

On retrouve dans la seconde catégorie les avantages des langages typés séquentiels à coloration « objet » :

- vérification et compréhension plus faciles des programmes ;
- meilleure structuration ;
- réutilisation simplifiée de programmes déjà existants ;
- robustesse du modèle de programmation.

Il est à remarquer que le vernis « parallélisme orienté objet » est indépendant du caractère orienté objet séquentiel du langage. Ainsi on peut très bien écrire un langage parallèle orienté objet basé sur C et non sur C++, comme le sont MPL ou C\* par exemple.

#### 4.1.1 Les langages de type Fortran

Le langage FORTRAN est tellement répandu dans le milieu scientifique qu'il est difficile de considérer sérieusement d'autres langages s'ils n'ont pas des avantages indéniables sur celui-là.

C'est pourquoi de nombreuses versions parallèles et vectorielles ont été et sont développées qui vont du FORTRAN standard autovectorisé<sup>2</sup> [KKLW80] aux versions plus spécifiques à certaines machines, parfois limitées à de simples directives de compilation rajoutées au FORTRAN standard.

Malheureusement, FORTRAN étant un des premiers langages scientifique d'assez haut niveau créé, il possède un certain nombre de défauts que sa large diffusion a bien répandu dans la communauté scientifique<sup>3</sup> comme étant des manières normales de programmer, voire même souhaitable. Même si certaines méthodes sont reconnues comme obsolètes et déconseillées [MR90] les compilateurs sont assez bridés par les **equivalence** osées ou les **common** farfelus pour ne considérer que les problèmes d'allocations sur machines parallèles.

---

1. Certains, du fait du manque de virtualisation, ne peuvent faire interagir des objets parallèles appartenant qu'à une seule classe : celle sous-jacente à l'architecture physique de la machine, ce qui nous semble trop limité.

2. On entend par là un compilateur qui accepte en entrée un langage *a priori* séquentiel et qui extrait automatiquement, dans la mesure de ses possibilités et suivant le programme d'entrée, du parallélisme et génère des instructions parallèles ou vectorielles pour une machine cible.

3. Selon le vieil adage : « on n'apprend pas FORTRAN, on imite ».

#### 4.1.1.1 Fortran 77

Bien que ce ne soit pas un langage de programmation parallèle en soit, le fait que des paralléliseurs automatiques de programmes écrits avec ce langage aient été introduits pour de nombreuses machines et architectures en font un langage de programmation pour machines parallèles.

L'intérêt principal est de ne pas avoir à retoucher (en théorie...) des programmes déjà écrits pour des ordinateurs « classiques » lorsqu'on veut les faire exécuter sur un ordinateur parallèle. Malheureusement, on est souvent obligé de changer le programme pour qu'il s'exécute plus efficacement (comme par exemple échanger des boucles), ce qui limite la portabilité de l'approche.

En fait, plutôt qu'à du FORTRAN 77 pur, on a souvent affaire à des extensions, empruntées aux langages parallèles que nous allons décrire après, pour des raisons d'efficacité. On peut avoir des directives d'aide au compilateur, par exemple pour savoir s'il est intéressant de paralléliser ou pas telle procédure, s'il faut stocker d'une certaine manière telle matrice, etc. et dans ce cas on retrouve l'approche assembleur, de haut niveau, dans la mesure où il faut retoucher les programmes en fonction de la machine cible, ce qui ne va pas vers une amélioration de la clarté algorithmique déjà assombrie par l'absence de parallélisme dans la syntaxe qui augmente aussi le nombre d'erreurs de programmation possibles.

On en arrive ainsi à des problèmes de mauvaise portabilité infirmant le but initial : le modèle climatique du MET OFFICE composé de 200000 lignes de FORTRAN 77 pour CYBER 205 a nécessité tout de même 50 hommes-années pour le réécrire en FORTRAN 77 sur CRAY Y-MP [Dic92], ce qui peut dissiper certaines croyances...

Néanmoins, le langage FORTRAN est la raison principale de la relative facilité d'utilisation des ordinateurs vectoriels actuels car il est relativement indépendant de la machine cible et cache au programmeur le foisonnement d'idées autour des machines parallèles : langages à passage de message dans les machines MIMD à mémoire distribuée, langage vectoriel pour les machines SIMD, langage avec des primitives de synchronisation dans le cas du MIMD à mémoire partagée, etc. Mais FORTRAN n'est pas le seul langage pouvant offrir cette facilité d'utilisation indépendante de la machine.

Un utilisateur choisira un nouvel ordinateur en fonction de ses performances bien entendu mais aussi en fonction de la garantie à moyen terme de la pérennité logicielle. Pour cette raison FORTRAN a encore un avenir commercial.

#### 4.1.1.2 HEP Fortran

HEP FORTRAN [Jor85] langage multitâche basé sur des appels à des procédures FORK et JOIN pour passer à des flots d'exécution multiples, qui agissent comme les `fork` et `wait` d'UNIX, possède aussi des boucles de type DOALL, des barrières de synchronisation et des variables « asynchrones » fonctionnant sur le principe de producteur/consommateur.

Le parallélisme est donc explicite, pouvant être à gros grain.

#### 4.1.1.3 EPEX/Fortran

Il s'agit d'un compilateur capable de générer un programme SPMD multiprocessus à partir d'un programme FORTRAN 77 contenant des macroinstructions précisant les

parties du programme qui peuvent être exécutées en parallèle [DGNP88], quelles sont les variables privées ou partagées, où doivent être mis les point de synchronisation, etc.

Le compilateur a été conçu à l'origine pour pouvoir générer du code pour la machine multiprocesseur à mémoire partagée RP3 mais peut aussi servir à programmer un IBM sous VM/SP à des fins de simulation : chaque processeur est simulé par une machine virtuelle. Malheureusement, le débogage du programme ne peut se faire qu'en mode uni-processeur, ce qui est fastidieux lorsque le temps d'exécution du programme est important.

Dans ce modèle, c'est au programmeur de protéger la sémantique de son programme car il n'y a aucun test de dépendances entre variables qui est effectué. Si par exemple on se contente d'écrire

où `@D0` est la version parallèle du `D0`, la dépendance séquentielle entre les éléments de `A` passera inaperçue et le résultat sera imprévisible.

Néanmoins, un point intéressant est l'allocation dynamique des morceaux de boucles parallèles aux différents processeurs, rendue possible par la présence du réseau de communication entre les processeurs et la mémoire dans la machine RP3 ainsi que du mécanisme de *Fetch & Add*, qui permet de lire une variable tout en la post-incrémentant d'une certaine valeur de manière atomique. Ce dispositif permet à un processeur d'accéder à l'indice courant de la boucle tout en l'incrémentant, pour le processeur suivant, du nombre de boucles que le premier va exécuter.

De plus un mécanisme astucieux d'*horloges* locales et globales associées à chaque section du programme permet une distribution dynamique du travail efficace même quand on a des boucles parallèles imbriquées, modulo le fait que la sémantique soit respectée :

- une horloge, locale à un processeur, est incrémentée lorsqu'un processeur entre dans la section correspondante, séquentielle ou parallèle ;
- une horloge globale contrôle l'avancement global du programme dans une section, *ie* le nombre de fois qu'à été exécutée cette section.

Si un processeur arrive dans une section alors que son horloge retarde, c'est que le travail est déjà fait. Il peut passer directement à la section suivante après avoir mis à l'heure son horloge.

Bien que tout soit sous la responsabilité du programmeur, les principaux concepts sont présents dans ce langage pour constituer un modèle de programmation SPMD à distribution dynamique de la charge. Mais comme il est fondé sur la présence d'une mémoire partagée et d'un *Fetch & Add* efficace et bien entendu atomique, le langage est difficilement adaptable pour d'autres architectures de machines.

#### 4.1.1.4 Fortran 90

Un des aspects intéressants de ce langage est que c'est un standard et que c'est pratiquement le seul qui a été implanté sur plusieurs machines, démontrant ainsi sa portabilité mais pas forcément son efficacité, car bien souvent certaines fonctionnalités du langage étaient restreintes ou seul un sous-ensemble du langage était considéré, selon la spécificité de chaque machine, comme nous le verrons à la fin de la section.

Par rapport à FORTRAN 77, les différences notables sont l'introduction du contrôle de flot structuré (**while** par exemple), la récursion, la surcharge des opérateurs (très pratique si on veut étendre la signification d'opérateurs existants à de nouveaux types de données) et surtout des opérateurs parallèles qui nous intéressent tout particulièrement ici.

En ce qui concerne ces opérateurs parallèles, il est dommage qu'on ne puisse imbriquer des opérateurs de masquage (**where**) et que la sémantique des opérateurs de réduction associative au sein d'un **where** ne dépende aucunement du **where** mais du paramètre optionnel **MASK=** [MR90, pages 114–116 et 170–171]. Par exemple si on veut calculer

$$\forall i \in I : A_i > 0, \quad B_i = \frac{A_i}{\sum_{A_i > 0} \log A_i}$$

on ne peut écrire

mais

car dans le premier cas la somme est faite sur tous les éléments de  $A$ , donc le logarithme est calculé même sur les éléments négatifs ! L'introduction du paramètre optionnel **MASK=** au lieu d'une généralisation du **where** semble un mauvais choix sémantique profond et fait preuve des inconsistances du langage.

En outre, le **where** ne peut pas être imbriqué. Il faut probablement voir là une réminiscence des machines vectorielles qui gèrent assez mal le contrôle de flot parallèle, comme on aura l'occasion de le voir dans le chapitre 7.

Le problème est que la définition de FORTRAN 90 laisse trop de liberté au programmeur pour que tout soit parallélisé dans l'état actuel de l'art, ce qui fait que soit le standard n'est pas implanté complètement, soit ce qui n'est pas parallélisé est exécuté séquentiellement.

En particulier l'introduction des pointeurs trouble le compilateur qui a du mal à reconnaître certaines parties parallélisables, surtout entre différentes procédures. C'est pour cela que les effets de bord dans les procédures sont interdits.

Enfin, bien qu'étant un langage « moderne » et raisonnable si on le compare à FORTRAN 77 — puisqu'il permet l'allocation dynamique, les pointeurs, les structures, etc. — il souffre de la compatibilité nécessaire avec FORTRAN 77 et d'une syntaxe qui laisse beaucoup à désirer<sup>4</sup>. Beaucoup de ces problèmes découlent de l'absence de mots-clés réservés en FORTRAN, ce qui complique énormément l'analyse syntaxique.

FORTRAN 90 semble suivre un funeste destin : il a mis si longtemps à être figé qu'il fait son apparition au moment où les machines vectorielles classiques sont en voie de récession, machines pour lesquelles il était destiné *a priori*. À peine figé il est très discuté par les défenseurs du nouveau FORTRAN hautement parallèle HPF à cause de ses difficultés d'application complète sur les machines existantes. La preuve en est, qu'à

---

4. En particulier, on se reportera avec intérêt à [MR90, page 22]: « *This is necessary in order to enable compilers to parse statements simply* ». Il est presque scandaleux que de telles considérations peuvent influencer la définition d'une norme qui marquera probablement les programmeurs à venir, à une époque où les analyseurs de types **lex** et **yacc** sont monnaie courante (en fait, gratuits sous forme GNU!).

notre connaissance, le seul compilateur FORTRAN 90 complet existant est celui vendu par NAG qui est en fait un traducteur vers le langage C, vu comme un assembleur portable. On ne peut pas générer du FORTRAN 77 car ce dernier est incapable de gérer l'allocation dynamique, les structures, etc. Libre ensuite au compilateur C de n'importe quelle machine de reparalléliser le code C...

#### 4.1.1.5 CM-Fortran

Il s'agit d'un sous ensemble du langage précédent qui a été conçu pour la Connection Machine [KLG90] où les problèmes d'allocation de tableaux sur les processeurs sont abordés de manière explicite afin de réduire les communications entre processeurs, notion qui n'existe pas en FORTRAN 90.

Il s'agit principalement d'essayer de placer les éléments des tableaux qui interviendront ensembles dans les calculs sur des processeurs identiques. La topologie globale est choisie afin de réduire le nombre des communications globales au profit de communications sur grille, plus économiques. Evidemment, certaines configurations de dépendances impliquent une perte de parallélisme et donc une exécution séquentielle.

Il n'est pas toujours facile de savoir quelle partie du programme sera effectivement exécutée en parallèle ou sur l'ordinateur hôte et l'utilisateur peut avoir des surprises.

Enfin ce langage est clairement le cheval de bataille commercial de TMC pour pénétrer le marché protégé des supercalculateurs [DKMS90], aux dépens des anciens langages disponibles sur la Connection Machine.

#### 4.1.1.6 Fortran D

Il s'agit d'un FORTRAN mettant en valeur le parallélisme à travers 3 principes en « D » : données, décomposition et distribution, qui résument bien le langage [HKK<sup>+</sup>91, FHK<sup>+</sup>92].

Les tableaux à traiter comme des entités parallèles doivent être alignés sur des *décompositions* permettant d'exprimer la localité des relations de calcul entre différents éléments de variables parallèles. Les alignements sont dynamiques pour permettre une adaptation de la répartition des données en fonction des différentes phases de calcul.

Les calculs sont exécutés *a priori* là où le résultat doit être stocké (selon le principe « *owner computes rule* ») mais ce comportement peut être modifié par la directive `on` permettant au programmeur d'exprimer l'endroit où doit être exécuté le calcul.

Une fois établis tous les alignements, le programmeur peut distribuer ses décompositions sur les processeurs de la machine cible afin de limiter les temps de communication entre processeurs. Il s'agit là d'optimiser les communications entre processeurs alors que la phase précédente optimisait le placement des données entre éléments interagissant directement de manière intensive.

Cette distribution peut être faite par blocs ou de manière cyclique si on veut favoriser la répartition de la charge, par exemple. Elle peut aussi être faite suivant une carte fournie dynamiquement par le programmeur afin de permettre des répartitions irrégulières sur les processeurs.

En ce qui concerne les domaines d'itération parallèles, ils sont exprimés par le constructeur `FORALL` qui possède une sémantique SIMD et le contrôle de flot parallèle

est exprimé par un `if` standard, ce qui n'est pas gênant conceptuellement dans la mesure où il agit que sur un élément vu comme un scalaire à l'intérieur du `FORALL`.

Enfin on retrouve un moyen d'exprimer les opérations de réduction, leur domaine d'application étant défini par un `FORALL` englobant.

On constate que ce langage contient beaucoup de choses raisonnables dans la mesure où il laisse beaucoup de liberté d'expression au programmeur dans la manière de décrire le parallélisme, les relations entre les données, le placement des données sur les processeurs.

#### 4.1.1.7 Vienna Fortran

Ce langage est basé sur des principes philosophiques très proches de `FORTRAN D` : la distribution des données et un modèle `SPMD` [CMZ92, ZBC<sup>+</sup>92].

Les différences principales résident dans l'absence d'alignement intermédiaire sur une décomposition à projeter ensuite sur des processeurs (ce qui empêche de dire que `FORTRAN D` est inclus dans `Vienna FORTRAN`) et dans un développement très poussé de toute la partie langage gérant les alignements.

Outre ce qui est disponible en `FORTRAN D`, le programmeur peut décider avec précision ce qui arrive aux alignements des arguments d'appel de fonctions ou de procédures, avant, pendant et après l'appel, facilitant ainsi le travail de compilation.

Un certain nombre de fonctions permettent d'extraire les caractéristiques d'alignement et de distribution pour écrire des programmes dépendant de la distribution, ce qui peut être utile pour optimiser des bibliothèques par exemple. À l'inverse, il est possible de déclarer de nouvelles fonctions de distribution et d'alignement.

En ce qui concerne le `FORALL`, ce dernier peut contenir des déclarations de variables locales, ce qui clarifie la lecture des programmes ainsi que la compilation. De plus les opérations de réductions comprises dans les `FORALL` peuvent être faites selon 2 directions ou suivant un arbre binaire<sup>5</sup>.

Les primitives d'allocation dynamique ont été reprises de `FORTRAN 90`, même si le langage est plutôt basé sur `FORTRAN 77`, pour permettre l'utilisation de tableaux dont la taille n'est pas connue à la compilation par exemple.

Les primitives d'entrées-sorties parallèles ne sont pas non plus oubliées et les primitives de `FORTRAN 77` ont été généralisées pour prendre en compte les problèmes de distribution des données sur les processeurs ainsi que leur entrelacement correct dans les fichiers physiques.

Enfin une primitive d'assertion permet au programmeur de garantir au compilateur que certaines conditions seront toujours vraies, ce qui peut faciliter des optimisations.

Pour résumer, ce langage est très complet au niveau des primitives de description des alignements et des distributions ainsi que d'aide à la compilation, mais malheureusement cela a pour conséquence de mener à un langage assez complexe.

#### 4.1.1.8 HPF

Le cas du langage `HPF` est intéressant car il est le fruit d'une collaboration internationale ouverte discutant par courrier électronique et ayant des discussions ciblées

---

5. Pour peu que cela ait un sens, dans le cas d'un `FORALL` à plusieurs indices. Mais bizarrement cela ne semble pas précisé.

régulièrement. Le but est de proposer très rapidement (en 1 an) un langage standard dérivé de FORTRAN 77 et FORTRAN 90 grâce auquel le programmeur puisse indiquer explicitement suffisamment d'information sur le parallélisme de son problème et le placement de ses données pour qu'il soit possible d'écrire des compilateurs, dans un premier temps pas forcément très optimisé, pour la plupart des ordinateurs, qu'ils soient parallèles, vectoriels, ou même scalaires.

## 4.1.2 Les langages de type Pascal

### 4.1.2.1 Concurrent Pascal

Ce langage a été créé à l'origine pour écrire de manière structurée des systèmes d'exploitation et est basé sur la notion de processus et de moniteurs afin de gérer les conflits d'accès sur des ressources partagées [Han75].

Il y a peu de choses de changées par rapport au PASCAL d'origine et le parallélisme reste à gros grain : celui du multi-tâche, ce qui est trop gros pour faire de la programmation orientée vers le parallélisme des données et la notion de multiprocesseur n'est pas envisagée *a priori*.

### 4.1.2.2 Actus

Le langage ACTUS [Per79] fait partie des premiers langages orientés vers les machines vectorielles et les processeurs de tableaux.

Le parallélisme est indiqué par l'intermédiaire du typage mais on est toujours obligé de mentionner l'étendue d'un vecteur ou d'un tableau parallèle même lorsqu'on le considère dans son ensemble, ce qui est lourd. Par exemple, l'addition de deux vecteurs s'écrira :

$$a[1:50] := b[1:50] + c[1:50];$$

et non pas de manière plus sympathique :

$$a := b + c;$$

Remarquons au passage qu'on est obligé d'avoir la même « extension du parallélisme » de part et d'autre du signe d'affectation, ce qui est certes restrictif mais simplifie de beaucoup la compilation du langage.

Les mouvements de données entre des éléments de variables parallèles identiques ou différentes se font grâce aux opérateurs d'alignement **shift** et **rotate**. Seuls sont donc disponibles les décalages sur grille, héritage des premières machines processeurs de tableaux de type ILLIAC IV [BBK<sup>+</sup>68], ce qui est insuffisant dans le cas général.

La fonction **if** parallèle de sélection d'éléments dans une variable parallèle correspondant au **where** de FORTRAN 90 existe et, contrairement à celui-ci, permet plusieurs imbrications. Par contre la sémantique du conditionnement parallèle est peu claire, en particulier lorsque cela s'applique à des parties scalaires.

Il est dommage que la taille des variables parallèles ait à être connue à la compilation, mais cela vient de la compatibilité avec le PASCAL d'origine.

Enfin, un point intéressant est de pouvoir préciser lors du typage d'une variable de type tableau certains attributs facilitant la gestion de la mémoire virtuelle : on peut préciser d'aller lire en avance par exemple les 5 colonnes suivantes de tel type

de tableaux, caractéristique qui reste unique à notre connaissance dans les langages parallèles.

Un compilateur a été écrit pour le CRAY-1 [PCMP85]. Le compilateur utilise le mécanisme de contrôle de l'écriture à l'affectation d'un vecteur du CRAY pour faire du contrôle de flot parallèle, ce qui oblige à exécuter le membre droit d'une affectation, ce qui est très fâcheux pour l'exemple suivant :

```
if a[1:50] > 0 then b[\#] := 1/a[\#]
```

où # représente l'extension du parallélisme sélectionnée par le test parallèle. Si un élément de **a** est nul, cela provoquera une division par 0 que l'on veut justement éviter, ce qui n'est pas raisonnable. Mais il s'agit là plutôt d'un héritage de l'architecture du CRAY plutôt que d'une caractéristique du langage.

#### 4.1.2.3 BLAZE

Décrit comme un langage de programmation pour calculs scientifiques [MR87], ce langage contient ce qu'il faut pour exploiter le parallélisme à grain fin : arithmétique sur les tableaux, opérateurs de type « APL ».

L'approche faite est de se rapprocher des langages fonctionnels ou de type flot de données pour éviter les problèmes d'alias entre variables. En particulier il n'y a pas de pointeurs ni d'accès possible à des variables non locales aux procédures. Evidemment, la liberté du programmeur s'en ressent.

Le parallélisme est introduit par l'opérateur **FORALL** et pour des raisons sémantiques les variables utilisées dans cette boucle sont copiées pour chaque indice si besoin est.

Le choix de l'allocation mémoire est de répartir les tableaux et variables vectorielles sur les processeurs sauf un, celui qui sert de contrôleur et de processeur scalaire. Il s'agit donc bien d'un modèle de programmation SPMD.

La décomposition semi-automatique de domaines en blocs pour l'allocation de variables parallèles sur un ensemble de processeurs à mémoire distribuée afin de minimiser les communications a été abordé [KMR87] mais la partie synchronisation entre les différents processeurs semble être passée sous silence. Outre le fait que la validité sémantique n'est pas vérifiée sans ces synchronisations, lorsqu'elles sont présentes elles peuvent représenter un temps important par rapport au calcul global. En plus, le réordonnement des communications n'est pas abordé alors qu'elles peuvent se faire en même temps que les calculs, si la machine cible le permet.

#### 4.1.2.4 Hellena

Le langage vectoriel Hellena [Jeg87] a été développé à l'origine pour la machine SIMD/SPMD OPSILA [AB86], à partir d'un PASCAL francisé<sup>6</sup>.

Il n'y a pas de différenciation entre les tableaux et les variables parallèles, ce qui est raisonnable puisqu'on peut considérer que la machine cible a une mémoire partagée.

---

6. On voit là l'influence néfaste du pays GALLO dans lequel est situé RENNES qui semble avoir eu raison du caractère bretonnant de son auteur... Néanmoins, il est dommage que les lettres accentuées aient été supprimées dans l'implémentation du langage par rapport à sa définition dans [Jeg87]. En plus les fonctions typiques d'OPSILA n'ont pas été traduites : le lecteur ne saura donc pas par exemple comment traduire *gather* ou *scatter* en français...

Le fonctionnement de la machine en mode SPMD se fait en déclarant des blocs de type `spmd...fin_spmd`, dans lesquels les variables déclarées sont locales aux processeurs, tout en restant accessibles en mode SIMD. L'approche est donc la même que pour BLAZE.

L'exécution SPMD est faite sur un modèle d'exécution de type *fork/join* et le nombre de processeurs physiques de l'ordinateur intervient explicitement dans le programme. Il est dommage que le mode SPMD manque de virtualité : on est obligé d'en tenir compte en programmant. Mais ce qui est finalement le plus gênant est cette différence aussi manichéenne entre SIMD et MIMD alors que le compilateur pourrait faire lui-même un peu plus la part des choses. Néanmoins, cette information permet de toute manière de faire des optimisations supplémentaires impossible à faire autrement.

Les interactions entre des éléments de variables parallèles se font grâce aux *patrons*, ce qui a pour effet de configurer de manière adéquate le réseau d'adressage à la mémoire.

### 4.1.3 APL

APL est un langage déjà ancien qui avait été conçu particulièrement pour résoudre des problèmes mathématiques et est capable de gérer des données parallèles telles que les vecteurs ou les matrices de n'importe quelle dimension, un peu à la manière d'une super-calculatrice<sup>7</sup> [Ive62].

Les constructions parallèles sont orthogonales contrairement à FORTRAN 90, qui ne peut faire des calculs directement que sur des matrices à 2 dimensions, et possède en particulier toute sorte de produits scalaires, de réductions, ainsi que des opérateurs préfixes parallèles généraux que l'on peut appliquer à d'autres opérateurs.

A l'origine interprété, des compilateurs ont été réalisés, avec en particulier une version réduite pour la machine parallèle RP3 suivant un modèle SPMD [JC91] : le langage est compilé en autant de tâches identiques qu'il y a de processeurs, chacune traitant une partie du problème.

La difficulté avec ce langage n'est pas tant liée à son utilisation dans un modèle parallèle qu'à ses caractéristiques intrinsèques. A la concision et l'expressivité du langage on peut rétorquer une gestion de la mémoire trop coûteuse, l'utilisation d'une fonte de caractère (et souvent d'un clavier) exotique et le manque de structures de contrôle itératif [Ber91]. Néanmoins, la généralisation de XWINDOW SYSTEM qui possède les caractères APL entre autres, ainsi que l'apparition de dialecte à base de caractères standards va peut-être permettre une vulgarisation d'APL<sup>8</sup>.

Enfin, tout problème écrit en APL se résume souvent à quelques lignes cabalistiques pour deux raisons principales :

- 1° le problème, une fois exprimé en APL, ce qui peut demander une réflexion longue et intense, est souvent très court ;
- 2° un humain normalement constitué a de toute manière du mal à écrire un long programme du fait de l'absence de structures itératives.

---

7. En fait on pourrait aussi étudier tous les langages de calcul mathématiques qui sont capables de gérer des opérations sur des vecteurs ou de matrices, tel que MAPPLE par exemple, mais il ne s'agit pas à proprement parler de langages généraux et il ne semble pas que des essais de parallélisation aient été menés jusqu'à présent sur ceux-ci.

8. Une solution intermédiaire serait peut-être de faire un convertisseur de  $\text{\TeX}$  vers APL car on trouve des fontes APL sous forme de METAFONT disponibles par ftp anonyme ainsi que le langage sur `wuvieai.wu-wien.ac.at` dans `/pub/lang/apl` [GSMN90].

#### 4.1.4 Les langages de type C

De même que l'importance du langage FORTRAN dans les milieux ayant besoin de calcul numérique a suscité de nombreux développements de langages à partir de celui-ci, la vulgarisation du langage C dans les milieux informaticiens a provoqué de nombreuses extensions au langage C dans le domaine du parallélisme, qu'elles soient vectorielles, à passage de messages ou à parallélisme de données, qui nous intéressent plus particulièrement ici.

##### 4.1.4.1 Le vieux C\*

Ce langage [Thi87b], un des premiers langages parallèles commerciaux à parallélisme de données<sup>9</sup>, a été introduit pour programmer la Connection Machine dans un langage de plus haut niveau que CPARIS [?]. Il consiste en une extension du langage C pour permettre une gestion du parallélisme de données.

D'un point de vue conceptuel, des espaces de processeurs virtuels dénommés **domain** sont introduits pour préciser l'extension du parallélisme pour chaque classe de données du problème à résoudre. La partie séquentielle du programme s'exécute sur un ordinateur hôte séquentiel tandis que les parties parallèles de C\* s'exécutent sur autant de machines parallèles virtuelles qu'il y a de **domain**.

D'un point de vue pratique, il n'y a qu'une machine parallèle qui simule toutes les machines parallèles virtuelles lors des différentes phases du calcul : pour chaque **domain**, chaque processeur physique simule *vp\_ratio* processeurs virtuels. S'il s'agit d'une simulation de machine parallèle sur une machine séquentielle, tous les processeurs parallèles virtuels seront simulés par l'ordinateur séquentiel.

Le langage est synchrone au niveau non seulement des instructions mais aussi des opérations, ce qui garantit la sémantique d'une évaluation parallèle de type  $\mathbf{a} = \mathbf{f}(\mathbf{a})$  : tout ce passe comme si tous les éléments de **a** sont lus avant que le premier élément de **a** soit écrit. Notons que comme cela **a** possède un sens clair, sans problème d'ordre d'évaluation.

Il n'y a aucune précision quant à la topologie des variables parallèles dans la machine et c'est là que le bât blesse : la plupart des programmes écrits en C\* sur la Connection Machine n'utilisaient pas toute la puissance de calcul de la machine à cause du temps perdu dans les communications générales, puisque toutes les topologies pouvaient être vues comme générales. En particulier la notion de pointeurs parallèles était tout à fait générale, chaque pointeur pouvant référencer un objet sur un autre processeur et la déréréférence pouvait générer des communications. Le nombre de types de pointeur dans ce langage est impressionnant.

Le compilateur C\* n'étant pas capable de s'apercevoir des topologies sous-jacentes aux programmes, de nombreux utilisateurs ont préféré se tourner vers le langage de plus bas niveau CPARIS, où on devait tout expliciter.

Même si la déclaration du parallélisme est simple, toutes les variables d'un **domain** sont déclarées comme si elles étaient des éléments d'une structure C possédant le nom du **domain** correspondant, l'utilisation des variables parallèles est très lourde car on est obligé de préciser à chaque fois le nom du **domain** en préfixe comme le montre l'exemple de la figure 4.1. Même si on voit qu'on peut factoriser le nom du **domain** sur des blocs

---

9. C'est la raison pour laquelle nous l'avons particulièrement étudié, ainsi que la Connection Machine.

---

```

domain espace {int a,b,c; ...}; /* On cr'ee ici le nouveau domaine espace.
    qui contient entre autre les variables entières parallèles a,b et c. */

espace.a = 0; /* On met tous les 'el'ements de a à 0. */
...
[domain espace].{
    a = b + c; /* a, b et c sont toutes du domaine espace. */
    b++;
}

```

---

FIG. 4.1 - *Petit exemple de C\**.

d'instructions à la manière du `with` de PASCAL, la syntaxe n'est pas très claire. En plus, il peut y avoir des conflits si une variable parallèle et une variable séquentielle ont le même nom, ce qui devrait être possible en se reportant à la notion de structure de C, ou alors la notion de `domain` n'est vraiment pas dans la philosophie des structures de C.

Entre autres, il n'est pas possible de créer des fonctions génériques pouvant accepter et retourner des variables appartenant à différents `domains`, ce qui empêche la création de bibliothèques de fonctions génériques.

Enfin, la notion de communication n'apparaît pas explicitement dans le langage. C'est le compilateur qui génère les communications nécessaires lorsque la vérification des types l'impose. Si cela peut simplifier la vie du programmeur, cela lui laisse peut-être trop de liberté puisqu'il peut générer beaucoup de communications sans s'en apercevoir... avant de mesurer les performances déplorables à l'exécution !

#### 4.1.4.2 DPC

Une adaptation du langage C\* pour des machines MIMD de type hypercube a été réalisée [HLJ<sup>+</sup>91] avec certaines améliorations en partant des considérations précédentes et ont mené au langage *Data Parallel C*: DPC.

La syntaxe reste la même mais la notion de topologie virtuelle apparaît. L'intérêt est de pouvoir préciser une topologie où des processeurs virtuels *voisins* seront voisins sur des processeurs physiques, limitant ainsi les coûts de communication si dans le programme chaque processeur doit utiliser beaucoup de données présentes chez ses processeurs voisins.

Un certain nombre de macro-instructions sont ainsi rajoutées pour faire des accès suivant une topologie de voisinage dans chaque `domain` ainsi que pour déclarer la méthode d'allocation des processeurs virtuels sur les processeurs physiques correspondant au mieux à la géométrie du problème.

Au niveau de la réalisation, puisque les seules interactions entre les processeurs virtuels sont dues aux communications, les primitives de synchronisation ont été introduites dans les primitives de communication.

Enfin, un point intéressant est la réalisation du compilateur sous la forme d'un traducteur de C\* vers C, permettant d'utiliser le compilateur C de la machine cible et d'augmenter ainsi la portabilité entre différentes machines. Bizarrement, cela c'est

aussi accompagné du portage d'un compilateur C lorsque le langage a été adapté sur une machine SYMMETRY [HQ91] ce qui a enlevé beaucoup d'intérêt à l'approche.

En ce qui concerne le processeur scalaire, si celui des machines cibles n'est pas assez puissant ou inexistant et de toute manière n'a généralement pas un système d'émission de valeurs scalaires simultanément à tous les processeurs physiques, toutes les variables scalaires sont dupliquées dans chaque processeur physique. Mais au niveau de l'utilisateur, tout se passe comme s'il y avait toujours un processeur scalaire. L'intérêt est que les émissions scalaires sont très rapides, car souvent faites localement. L'inconvénient principal est que si ces variables scalaires sont par exemple une grosse base de données graphiques, elles occupent une place non négligeable dans chaque processeur.

Actuellement, les machines sur lesquelles on peut programmer en DPC incluent les machines à mémoire distribuées (IPSC, NCUBE) et à mémoire partagée (SEQUENT, SYMMETRY) mais le gros problème reste le langage qui n'a pratiquement pas évolué depuis la version de la CM et qu'il faudrait probablement moderniser.

#### 4.1.4.3 Le nouveau C\*

C'est justement ce qui a été fait par la firme créatrice de l'ancien C\* afin de programmer plus efficacement leur machines [Thi90].

Totalement incompatible avec la précédente version, on peut se demander à juste titre pourquoi TMC a gardé le même nom au risque de troubler un certain nombre d'utilisateurs...

Par rapport à l'ancienne version, la notion de topologie est rajoutée, ce qui simplifie beaucoup le compilateur et surtout garantit à l'utilisateur que, si les données de son algorithme ont une certaine répartition spatiale compatible avec l'architecture de la machine exécutant son programme, il sera exécuté avec une bonne efficacité, ce qui n'était pas le cas avec la version précédente.

Les pointeurs globaux ont disparu et on ne peut plus faire que des indirections locales aux processeurs sur des tableaux, ce qui est du coup un peu trop restrictif puisque la CM possède l'indirection locale qui est tant utile.

Les notations de réduction scalaire sont concises puisqu'elles ressemblent un peu à celles d'APL : une réduction par une fonction  $f$  s'écrit respectivement dans chaque langage  $f=$  et  $f/$ . Mais du coup la sémantique C de cette notation n'est plus toujours valable : si  $a$  est scalaire et  $b$  est parallèle,  $a += b$  n'est pas la même chose que  $a = a + b$ , qui n'a d'ailleurs pas de sens.

Par contre il faut toujours préciser sur quelle « *shape* », notion plus géométrique qui a remplacé le `domain`, on doit effectuer les calculs alors que le typage permet de savoir quel *shape* est concernée.

La notion de *shape physical* a été rajoutée pour permettre de travailler au niveau des processeurs physiques. Malheureusement, cela est très inefficace sur la CM car cette machine étant à grain fin et possédant une station de travail comme générateur d'instructions, le rendement n'est élevé que pour des *vp\_ratio* suffisants, le contraire d'une collection physique qui a par définition un *vp\_ratio* = 1. En plus, il n'y a pas de possibilité de coercion entre une collection et la collection physique, ce qui complique la factorisation de certaines variables parallèles dans certains processeurs physiques par exemple.

Les *shapes* peuvent avoir des tailles dynamiques mais des variables dont le parallélisme est dynamique ne peuvent pas être globales, ce qui est plutôt gênant.

Enfin, il n'y a pas d'opérateur de communication sur grille torique : on est obligé d'écrire un déplacement par rapport à un axe avec un modulo de la taille correspondante que le compilateur repère et retraduit en une opération de communication sur grille torique ! En plus, les communications sur une *shape* multidimensionnelle sont indiquées par autant de `[]` à gauche qu'il y a de dimensions. Cela semble un mauvais choix dans la mesure où, contrairement aux `[]` du C, une absence d'un des `[]` à gauche n'a aucune signification et est une erreur.

Clairement, cela laisse l'impression d'un langage qui aurait eu besoin de plus de temps de développement.

#### 4.1.4.4 MultiC

Ce langage a été créé pour permettre de programmer la machine WAVETracer DTC et est basé sur du C ANSI. Ses principales extensions sont [Wav91] :

- l'introduction du mot-clé `multi` qui déclare des variables comme étant parallèles ;
- la possibilité de travailler sur un nombre de bits quelconque<sup>10</sup> ;
- les communications sur grille 2D ou 3D ;
- la virtualisation ;
- rajout du type complexe.

Les principales restrictions des langages viennent souvent directement de celles de la machine et le langage MULTIC spécialement conçu pour la machine DTC n'échappe pas à cette règle sur la plupart des points :

- pas de communication générale ;
- pas de pointeurs parallèles (pas d'indirection locale sur les PEs) ;
- pas plus d'un type `multi` actif à un instant donné : seules des variables de même taille parallèle peuvent coexister à un instant donné ;
- pas de différence syntaxique entre un `if` séquentiel et un `if` parallèle.

Néanmoins, un des points-clés du langage est que l'intégration avec la machine hôte est parfaite au niveau de la programmation et de l'édition des liens. Le langage permet aussi de programmer de manière parallèle sur des machines séquentielles pour faire des simulations.

---

<sup>10</sup>. Il faut rappeler que la machine cible est monobit, donc les calculs sur des entiers de 32 bits sont très coûteux. Par conséquent le langage doit permettre de proposer des variables de petite largeur.

#### 4.1.4.5 MPL

Basé sur le C K & R et maintenant ANSI, ce langage a été conçu pour la machine MASPAR MP1. Les extensions principales sont :

- déclaration de variables parallèles par le type `plural` ;
- pointeurs parallèles vers variable scalaire<sup>11</sup> ;
- pointeurs parallèles vers données parallèles (appartenant aux mêmes processeurs).

Ces limitations principales sont :

- pas de virtualisation : le langage ne connaît qu'un type de variables parallèles, celles qui ont la taille de la machine. Cela pose bien entendu des problèmes de portabilité des programmes d'une machine à l'autre ;
- la syntaxe est assez maladroite :
  - pour choisir un seul processeur (utilisation du mot-clé `proc.`) ;
  - pour exprimer les communications (`router[adr].var`) ;
  - les concentrations scalaires associatives ne sont pas élégantes et ressemblent plus à du CPARIS qu'à un langage de haut niveau ;
- la sémantique du `if` parallèle est vraiment étrange puisque les instructions scalaires contenues dans le bloc conditionné parallèle sont exécutées seulement si au moins un des processeurs parallèles est actif ;
- l'intégration entre l'hôte et la machine parallèle est bien moins faite que dans le cas de MULTIC dans la mesure où elle est loin d'être transparente.

Il s'agit donc d'un langage qui n'est pas vraiment fini et qui reste de bas niveau. Il reflète en fait beaucoup l'architecture physique de la MP-1. Néanmoins, le constructeur pense proposer un autre langage qui devrait inclure la virtualisation.

#### 4.1.4.6 EVA

Contrairement aux langages précédents basés sur C qui sont parallèles au sens où ils sont plutôt liés à des processeurs de tableaux, celui-ci, aussi inspiré de C mais moins directement, est aussi orienté machines vectorielles et contient de nombreux mécanismes de description de vecteurs [Mar92b].

En ce sens, le langage est moins typé que les précédents — il permet même bizarrement des opérations sur des vecteurs de tailles différentes — et ressemble plutôt aux langages de type FORTRAN. Comme il est orienté vers le calcul vectoriel classique, il n'y a pas de pointeur ou de structures de données compliquées, ni de contrôle de flot parallèle compliqué.

Dans la mesure où la garantie des dépendances est assurée par le programmeur, la sémantique dépend de ce dernier. Mais c'est aussi très intéressant car cela permet d'économiser des vecteurs temporaires simplement et augmenter les performances sans avoir à faire d'étude de dépendance, parfois impossible lorsqu'on fait appel à des descripteurs dynamiques de vecteurs.

---

<sup>11</sup>. C'est utile lorsqu'on a une base de donnée typiquement scalaire qu'on ne veut pas répéter sur chaque processeur. Malheureusement, les accès sont *forcément séquentialisés*...

### 4.1.5 Les langages de type C++

L'intérêt de développer un langage parallèle à partir de C++ vient des possibilités de « surcharge » des opérateurs, *ie* on peut par exemple étendre sa signification à d'autres objets parallèles comme des vecteurs ou des matrices.

On peut donc développer un simulateur du langage parallèle facilement et un prototype de compilateur si on a un compilateur C pour la machine cible, les opérations parallèles étant effectuées par des appels à des fonctions de bibliothèque. Cela explique le nombre important de langages parallèles développés autour de C++, même si beaucoup sont plus orientés parallélisme de contrôle ou langage à passage de messages.

#### 4.1.5.1 Porta-SIMD

C'est cette approche qui a été choisie dans [Tuc90] pour développer le langage PORTA-SIMD, décrit comme un langage *optimalement portable* pour machines de type SIMD, c'est à dire que tout programme s'exécutant sur une machine donnée peut s'exécuter sur une autre machine en un temps comparable à un facteur constant près.

L'abord de cet aspect de la portabilité est intéressant mais est malheureusement vu au niveau de l'exploitation maximale du matériel par un jeu d'instructions spécifiques, plutôt que par un langage qui ne reposerait sur aucune assertion préalable sur l'architecture de la machine cible.

Il s'agit donc d'inclure dans le langage une instruction ou un appel à une procédure pour chaque spécificité de chaque machine dont le choix est basé sur une taxinomie des architectures existantes. Le langage est portable sur plusieurs machines dans la mesure où on peut faire facilement un compilateur pour n'importe laquelle de ces machines mais les programmes écrits ne le sont que s'ils n'utilisent que des spécificités communes à toutes les machines. Dans le cas contraire, le compilateur est obligé d'émuler certaines spécificités matérielles sur certaines machines et il n'y a plus vraiment de « portabilité optimale » du point de vue du programmeur.

Il est donc dommage que la portabilité ne soit pas considérée au niveau algorithmique mais simplement au niveau des instructions capables d'exprimer toutes les architectures de machines SIMD.

Comme ce langage a été développé dans le cadre du projet PIXEL-PLANE 5, le langage est plutôt orienté parallélisme à grain fin et explique la philosophie de l'approche.

#### 4.1.5.2 CM++

Il s'agit plus d'un emballage en C++ du macroassembleur PARIS de la Connection Machine que d'un langage de programmation vraiment nouveau [Col90] et reste très lié à la machine cible, ce qui limite forcément la portabilité.

En plus, étant fortement basé sur PARIS, le langage n'est pas d'un niveau d'abstraction suffisant pour permettre une utilisation générale.

### 4.1.6 Parallaxis : un langage basé sur Modula 2

Il s'agit d'un langage fortement typé basé sur un sous-ensemble de MODULA 2, ce qui peut être un inconvénient si on considère le faible taux d'utilisation de ce langage.

L'expression du parallélisme est basée sur la définition préalable d'une machine virtuelle avec des processeurs reliés par un réseau de communication de voisinage. Le parallélisme est statique et il n'y a qu'une seule machine virtuelle en action à un moment donné [Brä89, BBES92]. On peut voir ce langage un peu comme l'intersection des langages MULTIC et MPL, donc héritant de l'union de leurs inconvénients, en ce qui nous concerne.

Clairement, ce langage semble plus adapté à des machines de traitement d'images qu'au calcul scientifique de par le manque de souplesse dans le parallélisme et l'absence de communication générale. En plus, la notion de « machine virtuelle » à définir, même dans le cas où elle a un sens, peut effrayer outre mesure un utilisateur  $\lambda$ .

Du fait du côté statiquement reconfigurable du réseau de la machine virtuelle cible, le langage semble avoir été fait pour programmer une machine à TRANSPUTERS avec réseau statique configurable globalement de manière SIMD, mais pourtant cela ne semble pas avoir été envisagé.

Si la machine virtuelle ne correspond pas à la machine physique, il doit apparaître les problèmes de placement classiques des processeurs virtuels et des liens virtuels de communication sur leurs homologues physiques.

## 4.2 POMPC

Après avoir exposé grossièrement un panorama des langages à parallélisme de données, on peut présenter le langage de la machine POMP, dénommé POMPC car basé sur C. Bien que développé par Nicolas PARIS [Par88, Par92], il nous semble particulièrement utile de décrire ce langage dans cette thèse car de nombreux choix ont été faits de concert avec les choix architecturaux. Bien entendu, cet exposé n'est en rien un texte de référence comme [Par92] mais souligne les points intéressants du langage.

On peut se poser deux questions : pourquoi encore un autre langage parallèle et pourquoi un langage basé sur C ?

En ce qui concerne le langage, comme on l'a vu dans le chapitre 3 sur le modèle de programmation, il s'agit de raisons purement culturelles : on était habitué à C, langage de haut niveau intéressant car permettant d'être en même temps d'assez bas niveau pour être près de la machine lorsqu'on le désire. Tout ce qu'on peut faire rapidement en FORTRAN, on peut le faire en C<sup>12</sup>. Enfin, nous disposions de compilateurs C sur de nombreuses machines afin de tester des prototypes du langage et cette profusion de compilateurs C explique que de nombreuses personnes connaissent le langage C, laissant espérer de nombreux utilisateurs potentiels.

Pour ce qui est de la création d'un nouveau langage, les langages qui existaient ne répondaient pas forcément à notre attente, comme on a pu le voir : soit ils sont trop éloignés de toute machine concrète et il est alors très difficile de les compiler efficacement, soit au contraire ils sont trop près de la machine et sont à considérer plus comme des langages d'assemblage que comme des langages réellement portables. Pour d'autres, ce sont leurs faiblesses syntaxiques ou sémantiques qui ne nous semblaient pas permettre un développement plus large du langage. Le fait d'avoir un langage propre

---

12. En FORTRAN, on peut faire par exemple des calculs flottants sur des nombres de taille arbitraire. Malheureusement, les machines cibles ne sont pas virtuelles... Donc tout calcul flottant s'écartant de la simple ou double précision sera exécutée très lentement car émulé logiquement.

permet de le faire évoluer vers ce que l'on estime de mieux adapter à nos besoins sans être enfermé dans un carcan standardisé, d'autant plus qu'on maîtrise alors le savoir faire.

Cet argument est appuyé par le fait que de toute manière on n'aurait jamais eu accès aux sources des compilateurs des langages que nous avons survolés et il aurait fallu tout réécrire. Partant de là, comme les langages existants ne répondaient pas vraiment à notre approche, on pouvait bien autant redéfinir quelque chose nous convenant mieux.

Un langage portable, même s'il est destiné *a priori* pour une machine particulière, nous semble très important car, bien sûr, cela permet de se détacher un peu de la réussite d'un projet de machine, mais cela peut intéresser d'autres personnes utilisant d'autres machines et permettre de valider le langage sur des machines parallèles en grandeur nature. Aussi il est toujours plus économique de pouvoir simuler un programme avec un jeu de données réduit sur une station de travail par exemple que d'immobiliser une grosse machine parallèle dans un premier temps.

Mais si on a dit qu'on ne voulait pas de langage autovectorisant ou parallèle de trop haut niveau, car l'investissement nécessaire dépassait le cadre de notre petit équipe, le fait d'avoir un langage portable sur plusieurs machines permet à d'autres équipes de prototyper efficacement leurs projets de langages de plus haut niveau en faisant abstraction d'un certain nombre de problèmes résolus par le langage portable.

#### 4.2.1 L'expression du parallélisme : les collections

Nous avons choisi d'avoir un typage fort du parallélisme et chaque variable parallèle appartient à une classe de parallélisme, une *collection*<sup>13</sup>, qui a la particularité de regrouper toutes les variables possédant le même parallélisme, à la manière d'un collectionneur de parallélisme qui passerait en revue toutes les variables de son programme.

Cette notion n'est pas abstraite et se retrouve d'ailleurs dans de nombreux langages à parallélisme de données. Lorsqu'on écrit un programme parallèle devant par exemple modéliser un phénomène physique, on est amené à modéliser la réalité par l'intermédiaire d'un certain nombre de grandeurs parallèles telles que des potentiels, des intensités, des champs de force, etc.

Dans les équations de modélisation, la séparation entre ces notions est claire et on ne voit pas pourquoi ces caractéristiques, telle que la notion d'intensivité, d'extensivité, de variables d'espace, perdraient leurs caractéristiques pour être traduits en tableaux FORTRAN linéarisés insipides et impénétrables à toute personne (non avertie). Le fait de pouvoir ajouter des tensions avec des températures nous semble moralement dangereux et c'est ce qui motive le développement du typage. Évidemment, partir d'un langage assez peu typé comme C implique l'héritage de ces règles de typage. Par contre, on peut rajouter au niveau du parallélisme le typage que l'on veut.

De même, si on aborde une simulation plutôt orientée objet, on peut discrétiser l'espace : cela nous donne une première notion de parallélisme, donc une collection. Après peuvent intervenir des interactions entre certains points considérés de l'espace qui nous donnent une autre collection de parallélisme, celle des interactions. Une telle approche permet de structurer efficacement de nombreux problèmes massivement parallèles et le chapitre ?? en donnera un petit aperçu.

---

13. On pourrait à ce sujet proposer pour POMP le concept de *Collection Machine*...

---

```

collection [100,100]matrice; /* Une collection de matrices carrées
                               de taille 100. */
collection vecteur; /* Définit une classe de vecteurs de
                       taille et de dimensions non spécifiées statiquement. */
collection [900,1152]ecran; /* Déclare une collection pour l'écran
                               du Sun. */
collection [50,50,50]espace; /* La discrétisation spatiale du problème. */
...
ecran char image_noir_et_blanc; /* Une variable parallèle de 900 × 1152 c.*/
ecran char image_couleur[3]; /* Un tableau scalaire de 3 variables
                               parallèles de 900 × 1152 pour le rouge, le vert et le bleu. */
espace double temperature, vitesse[3]; /* La température et le vecteur
                                           vitesse en tout point de l'espace 50 × 50 × 50. */
vecteur double un_vecteur; /* Une variable parallèle de type vecteur. */

```

---

FIG. 4.2 - Exemple de collections et de déclarations de variables en POMPC.

Pour exprimer cette notion de collection dans le langage C de départ, il suffit de déclarer une collection grâce au mot-clé `collection` qui a un effet similaire à un `typedef`. L'identificateur de la collection ainsi définie devient un constructeur de type parallèle orthogonal aux autres types du C comme indiqué sur la figure 4.2, contrairement à la construction étrange de C\*.

Remarquons sur cette figure l'indépendance qu'il y a entre le parallélisme et les tableaux : on peut très bien avoir une variable parallèle dont chaque élément est un tableau. Le parallélisme est repéré par un `[ ]` à gauche tandis que les tableaux sont décrits par des `[ ]` à droite, ce qui correspond respectivement aux `[ : ]` et `[ . ]` du langage ACTUS [Per79]. La différence est qu'une indirection sur un élément du tableau ne nécessitera pas de communications, tandis qu'une indirection à gauche sera une communication, éventuellement locale.

On peut aussi, si on le désire, indiquer la taille de la collection lors de sa déclaration si on la connaît avant la phase de compilation. Le fait de permettre de toute manière des variables parallèles de taille dynamique nous semble primordial puisqu'il s'agit là de la limitation principale de FORTRAN en particulier. On peut aussi redéfinir lors de l'exécution la taille d'une collection définie de manière statique<sup>14</sup>. Les bibliothèques `pc_start_collection` et `pc_kill_collection` sont utilisées à cet effet, ainsi que d'autres permettant de récupérer des informations sur la géométrie d'une collection.

Notons néanmoins qu'on ne peut bien entendu pas allouer une variable parallèle avant d'avoir défini la taille de la collection. Par contre on autorise les variables parallèles globales de collections non initialisées, ce qui est indispensable pour dépasser les limitations de FORTRAN ou de C\*.

On autorise des pointeurs parallèles mais ceux-ci sont locaux aux processeurs virtuels, pour des raisons d'efficacité et pour empêcher de déclencher des communications

---

14. Cela signifie donc qu'actuellement le compilateur ne peut pas faire d'optimisations en supposant que la taille ne changera pas. Mais il suffirait de rajouter au langage un attribut comme `collection static` pour assurer au compilateur que la taille de cette collection ne peut pas changer.

lorsqu'on les utilise, comme dans le cas du vieux C\*. Les pointeurs scalaires vers des variables parallèles ne posent pas de problème.

#### 4.2.2 Le contrôle de flot parallèle

Le constructeur principal de contrôle de flot parallèle est le **where** que l'on peut considérer comme un **if** parallèle et qu'on retrouve dans de nombreux langages vectoriels, parfois sous d'autres noms.

Dans un modèle à parallélisme de données, la trame principale du programme est régie par le contrôle de flot scalaire, à savoir ici tous les constructeurs du C tels que **if**, **for**, **while**, etc. et on se contente de réaliser ou non des calculs sur des variables parallèles grâce aux instructions de contrôle de flot parallèle. Comme on aura l'occasion d'y revenir au chapitre 7, il ne s'agit donc pas à proprement parler d'un vrai contrôle de flot d'instructions parallèles puisqu'il n'y a souvent pas vraiment de débranchements parallèles, du moins sur une machine SIMD. Par analogie avec une machine SIMD, on conserve la notion d'*activité* pour chaque processeur de la machine même si le langage est utilisée sur une machine MIMD: chaque processeur peut être actif ou pas, selon le test correspondant au niveau du **where** le concernant et donc exécuter ou pas les instructions parallèles contenues dans un bloc parallèle conditionné.

Un **where** n'a de portée et de signification que sur les variables de la collection de la condition parallèle et non pas sur les variables d'autres collections ou sur les variables scalaires, contrairement au **if** qui a un effet sur toutes les variables et instructions du bloc contrôlé. C'est cette différence sémantique profonde qui nous a fait choisir d'appeler l'indentificateur de contrôle de flot parallèle différemment du **if**, comme en FORTRAN 90 ou C\* par opposition à d'autres langages tels que MPL ou MULTIC qui conservent le **if** pour des variables parallèles, ce qui peut troubler les utilisateurs. Sur la figure 4.4 on trouvera un exemple détaillant la sémantique du **where** qui veut que le bloc **where** soit exécuté avant le bloc **elsewhere**, comme indiqué dans le commentaire de la figure page 86. Cette sémantique peut sembler abusive car sur une machine MIMD par exemple on peut exécuter les 2 branches du **where** en parallèle. Mais il s'agit ici de sémantique: tant mieux si le compilateur est capable d'optimiser tout en garantissant une équivalence du programme. Cette sémantique est aussi celle de FORTRAN 90 et de HPF par exemple, mais est loin de la « non sémantique » de MPL par exemple.

La raison de permettre des mélanges de collections différentes ou de variables scalaires est donnée par l'exemple de la figure 4.3 qui calcule une valeur absolue d'une variable parallèle ainsi que le nombre d'éléments négatifs d'un\_vecteur. (vecteur int) 1 est la constante parallèle 1 de la collection vecteur et est donc affectée par le **where**. Comme on le verra, **+<-** est ici l'opérateur de concentration scalaire additive. Le mélange de collections apparaîtra avec l'introduction des communications.

Il nous semble aussi indispensable d'autoriser l'imbrication d'autant de niveaux de conditionnements parallèles que l'on veut, qu'ils concernent des conditions identiques ou pas, contrairement à FORTRAN 90 où il faut faire à la main des compositions de conditions parallèles pour simuler un empilement de **wheres**. L'imbrication de **wheres** en POMPC correspond à celle de l'imbrication des extensions de parallélisme du compilateur pour le langage ACTUS [PCMP85]. Les techniques employées dans le compilateur POMPC sont en fait très similaires à celles-là.

Mais il ne faut bien sûr pas perdre de vue que le fait d'avoir un nombre élevé

---

```

int n_negatif;
where (un_vecteur < 0)
{
    un_vecteur = -un_vecteur;          /* Change le signe des 'el'ements
                                       n'egatifs seulement. */
    n_negatif = +<- (vecteur int) 1; /* Met dans n_negatif le nombre
                                       d''el'ements r'epondants à la condition. */
}

```

---

FIG. 4.3 - *Petit exemple à base de valeur absolue parallèle.*TAB. 4.1 - *Opérateurs de contrôle de flot en POMPC.*

<i>Séquentiel</i>	<i>Parallèle</i>
if	where
else	elsewhere
for	forwhere
do	dowhere
while	whilesomewhere
break	break
continue	continue
switch	switchwhere
default	default
return	return
&&	&&
...? ...: ...	...? ...: ...
—	everywhere
goto	—

d'imbrications de conditions parallèles dans un modèle d'exécution purement SIMD peut faire baisser de manière conséquente l'efficacité de la machine [Fly72], donc on s'arrangera pour éviter le plus possible un trop grand empilement de conditions.

De même que le `if` a comme pendant parallèle le `where`, la plupart des opérateurs de contrôle de flot séquentiel du C sont étendus en POMPC comme indiqué sur le tableau 4.1.

La différence entre le `where` et le `whilesomewhere`, outre la même différence qu'il y a entre un `if` et un `while`, est qu'on n'exécute le corps de boucle que s'il y a au moins un élément qui vérifie la condition dans le cas du `whilesomewhere`.

La sémantique des `&&` et `||` a été étendue aux variables parallèles et leur activité est modifiée en conséquence.

Il est clair que le `goto` n'a pas d'équivalent parallèle car il n'a pas sa place dans notre modèle de programmation structuré à parallélisme de données.

Enfin, il est souvent nécessaire de faire certaines manipulations système de bas

niveau ou algorithmiques particulières<sup>15</sup>. Pour cela, un `everywhere` est proposé pour faire exécuter du code parallèle contenu dans un bloc sur tous les éléments des variables parallèles, quels que soient les blocs parallèles conditionnés qui contiennent ce bloc. Un contrôle encore plus fin de l'activité peut être réalisé par des fonctions de bibliothèque mais reste à utiliser avec parcimonie, du fait des effets de bords non portables qui peuvent en résulter.

### 4.2.3 Les communications

Les communications que nous avons présentées dans la section 3.6 sont incluses dans le langage POMPC.

Les communications possèdent une sémantique différente d'une affectation car certains éléments d'une destination peuvent être le siège de collisions (cas d'une émission non associative) et ont une autre représentation en POMPC : `<-` qui évoque le déplacement de données, mais le `=` est utilisé pour les réceptions dans la mesure où la sémantique est la même que celle d'une affectation. En outre, l'écriture de l'opérateur `<-` permet au programmeur de savoir qu'il y a effectivement une communication rien qu'en regardant l'instruction, ce qui n'est pas évident lorsque cela est fait seulement par le typage — la définition des variables peut être ailleurs et très loin dans le programme et donc être visuellement inaccessible simultanément.

Que ce soit pour l'émission ou la réception, l'activité qui est prise en compte est celle du membre gauche de la communication, qui est aussi celle de l'adresse. Ce choix est guidé par les raisons suivantes :

- lors d'une émission, on peut restreindre facilement l'envoi de valeurs valides par un `where` sur la collection de l'émetteur ;
- dans le cas d'une réception ou d'une émission, on peut facilement garantir que seulement les adresses valides provoqueront une communication.

L'opérateur `[]` à gauche permet d'accéder aux éléments d'une collection de manière linéaire dans le cas d'une collection multidimensionnelle, un peu comme cela est possible en FORTRAN avec les tableaux à plusieurs dimensions. Néanmoins, il est bien sûr possible d'accéder à un élément en fonction de ses coordonnées dans l'espace de définition, non linéarisé. Cela peut se faire par l'intermédiaire de la fonction `pc_build_address` qui construit une adresse linéarisée en fonction des coordonnées. Mais pour que cela soit plus pratique et plus lisible, une écriture plus simple est fournie au niveau des communications : il suffit de remplacer par exemple dans le cas d'une variable parallèle de collection à 3 dimensions `[adr_1in]` par `[[x,y,z]]` où ces variables parallèles sont les coordonnées dans chaque dimension.

Mais pour communiquer dans un espace à plusieurs dimensions, il faut pouvoir s'orienter et savoir énumérer les éléments de variables parallèle. À cet effet est fournie la fonction `pc_coord` qui permet d'obtenir la position d'un élément dans une dimension. Construire par exemple un vecteur qui contient les valeurs de 0 à  $n - 1$  s'écrit donc en POMPC :

---

<sup>15</sup>. Même si cela peut faire crier certains sémanticiens...

TAB. 4.2 - Différentes sortes de communications en POMPC.

Type de Communication	Transcription en POMPC
<i>Émission</i>	<code>[adresse]destination &lt;- source</code>
<i>Réception</i>	<code>destination = [adresse]source</code>
<i>Émission associative</i>	<code>[adresse]destination op&lt;- source</code>
<i>Réception unaire</i>	<code>[adresse]source</code>
<i>Réception scalaire</i>	<code>scalaire = [scalaire2]source</code>
<i>Réception scalaire associative</i>	<code>scalaire op&lt;- source</code>
<i>Diffusion scalaire</i>	<code>destination = scalaire</code>
<i>Émission scalaire</i>	<code>[scalaire2]destination = scalaire</code>

---

```
collection [n]vecteur ; /* Pour simplifier, la taille est statique. */
vecteur int un_vecteur;
...
un_vecteur = pc_coord(0);
```

---

qui correspond à « `unvecteur ← in` » en APL.

La transposition d'une matrice peut s'écrire par exemple :

---

```
matrice double une_matrice, sa_transposee;
matrice int x,y;
```

```
x = pc_coord(0); /* L'intérêt de mettre les coordonnées dans des */
y = pc_coord(1); /* variables est qu'on pourra les réutiliser. */
[[y,x]]sa_transposee <- une_matrice; /* On transpose. */
```

---

On s'aperçoit qu'avec l'opérateur `[[[]]]` et la fonction `pc_coord` on peut réaliser les communications sur grille. Néanmoins, l'utilisation d'un `pc_coord` dans chaque dimension est lourde à écrire aussi bien qu'à relire. En plus c'est difficile à reconnaître pour le compilateur si on veut qu'il utilise des routines optimisées pour de telles communications. C'est pourquoi on autorise une autre écriture basée sur l'opérateur `[.]` signifiant « ici dans la dimension courante » (qui a aussi un équivalent en APL `[?]`): un décalage sur grille « d'un cran vers la droite<sup>16</sup> » d'une variable à 3 dimensions s'écrit :

```
temperature2 = [[.+1 , . , .]]temperature;
```

---

16. Interprétation très subjective et anthropomorphe...

---

```

plan double v; /* D'efinit une variable bidimensionnelle. */

where((pc_coord(plan,0) + (pc_coord(plan,1)) & 1)
      /* Calcule d'abord sur les cases noires. */
      v = [[.-1,.]]v + [[.+1,.]]v + [[.,-1]]v + [[.,.+1]]v - 4*v;
elsewhere
      /* Et SEULEMENT ENSUITE sur les cases blanches. */
      v = [[.-1,.]]v + [[.+1,.]]v + [[.,-1]]v + [[.,.+1]]v - 4*v;

```

---

FIG. 4.4 - Sémantique du *where* sur fond de laplacien dans une méthode de l'échiquier.

Les communications sur grilles peuvent être effectuées de manière torique ou pas. Cela est paramétrable au niveau de chaque dimension de chaque collection par la fonction de bibliothèque `pc_set_torus`.

Un cas particulier de communications est celui des émissions scalaires utilisées par exemple pour faire des initialisations qui sont notées comme sur la figure 4.2. Deux autres cas courants sont aussi la diffusion scalaire et la réception scalaire.

Orthogonalement à la notion de communication générale ou sur grille on peut rajouter la combinaison avec un opérateur associatif comme indiqué sur la figure 4.2 dans le cas de l'émission parallèle associative et de la réception scalaire associative, très utile pour récupérer une condition globale. En plus des opérateurs associatifs du C on retrouve l'opérateur `>?` et `<?` de C\* permettant de calculer un maximum et un minimum.

Enfin, pour permettre d'autres utilisations, un opérateur unaire de communication est fourni et permet par exemple la définition d'une réception additive ou l'écriture d'un laplacien dans une méthode itérative suivant la méthode de l'échiquier `[?]` comme indiqué sur la figure 4.4. Même si la méthode indiquée n'est pas forcément optimale en calcul, elle montre clairement le mode de fonctionnement du `where` dont la sémantique est d'exécuter d'abord le bloc `where` et ensuite le bloc `elsewhere` s'il existe. En particulier le bloc `elsewhere` peut très bien avoir besoin de variables calculées dans le bloc `where`, par conséquent il faut que le compilateur soit capable de garantir ce fait. On voit donc bien qu'il y a une différence sémantique très nette avec le `if` scalaire.

Les opérations préfixes parallèles sont appelées par l'intermédiaire de fonctions de bibliothèque mais on pourrait éventuellement les noter avec un `\` comme en APL étant donné que ce symbole n'est pas réellement utilisé en C<sup>17</sup>, sous forme d'opérateur unaire pour une opération préfixe parallèle et sous forme d'opérateur binaire dans le cas de la version segmentée.

#### 4.2.4 Placement des données

Qui parle de communications pense aussi à la diminution de celles-ci pour des raisons d'efficacité : mieux vaut pouvoir faire des accès à la mémoire locale plutôt que de faire une communication beaucoup plus lente.

Un moyen de diminuer ces communications est de s'arranger pour projeter des collections de manière à ce que les communications les plus courantes puissent se faire

---

17. Mais pour des raisons de préprocesseur et de fins de ligne on a peut-être intérêt à choisir `\\`.

---

```

collection une_collection double factorielle(une_collection double n)
{
    where(n <= 1)
        return 1;                                /* 1! = 1 */

    if ( <- ( n > 1))                            /* S'il y a encore des (n > 1) */
        return n*factorielle(n - 1);          /* on continue la r'ecursion. */
}

```

---

FIG. 4.5 - Exemple de fonction parallèle générique.

comme des accès locaux lorsque cela est possible. Il s'agit là bien évidemment d'un compromis à trouver: si on met tout le problème sur un seul processeur par exemple, certes il n'y aura plus de communications coûteuses, mais il n'y aura plus non plus de parallélisme...

La spécification de l'allocation physique d'une collection est faite par un tableau optionnel fourni à la fonction `pc_start_collection`. Un élément du tableau permet de spécifier pour chaque dimension de la collection si on préfère la répartir sur plusieurs processeurs ou au contraire plutôt la compacter sur le minimum de processeurs.

Ce mécanisme permet de préciser la répartition de manière moins manichéenne que l'attribut `BLOCK[n]` de `HPF` et d'indiquer des compromis. Dans l'état actuel du compilateur, il n'est pas possible de préciser une répartition cyclique sur certaines dimensions (`CYCLIC` en `FORTRAN D` par exemple), pourtant utile si par exemple on a souvent à faire des calculs sous des sous-vecteurs ou des vecteurs de matrices afin d'améliorer l'équilibrage de la charge.

#### 4.2.5 Les fonctions parallèles

On peut passer des arguments parallèles à des fonctions, arguments pouvant appartenir à plusieurs collections différentes puisque l'appel de la fonction en lui-même est scalaire. On peut aussi bien sûr renvoyer un résultat parallèle.

Il nous semblait nécessaire de pouvoir écrire des fonctions génériques acceptant plusieurs sortes de parallélisme, ne serait-ce que pour écrire une bibliothèque de fonctions mathématiques par exemple qui puisse marcher avec toutes les collections. Pour se faire, en plus de la valeur, un paramètre formel peut hériter aussi de la collection du paramètre d'appel. Il suffit de mentionner le mot `collection` devant le premier nom de collection de l'entête de déclaration d'une fonction, comme le montre la figure 4.5.

Ce petit exemple calcule de manière récurrente la factorielle des éléments d'un vecteur. Puisque les appels de fonctions font partie du contrôle de flot scalaire, on est obligé de rajouter une garde scalaire pour terminer les appels lorsque tous les éléments ont été calculés. Si on remplaçait le `if` par un `where` on aurait une récurrence infinie.

Bien entendu, on peut aussi écrire des fonctions n'acceptant que des collections précises, pour des raisons de lisibilité suite au typage accru, d'optimisations à la compilation et d'efficacité à l'exécution, ou tout simplement car elles font intervenir des variables parallèles globales.

Mais il existe aussi le cas intermédiaire où une petite fonction ne sera exécutée

que sur un nombre restreint de parallélismes différents et on voudrait bénéficier des avantages des fonctions à collections fixées. Cela est possible par le mécanisme de surcharge emprunté à C++, lui-même repris dans C\* et dans FORTRAN 90, à travers le mot-clé `overload`. Un cas particulier mais important est la définition de la bibliothèque mathématique : on préfère appeler les fonctions avec le même nom qu'elles s'appliquent à des scalaires ou à des variables parallèles. Elles sont donc écrites en 2 versions : une qui accepte des scalaires et une autre qui accepte des collections quelconques. La surcharge parallèle complète donc le mécanisme permettant d'offrir une interface uniforme à des fonctions travaillant sur des variables, types et parallélismes différents.

Les opérateurs de base du langage ne sont pas surchargeables comme en C++ puisqu'ils sont déjà polymorphes au point de vue du parallélisme. On ne peut donc pas redéfinir le `*` pour que, par exemple, appliqué à 2 matrices, qui est par défaut une multiplication élément par élément, il effectue une opération de multiplication de matrice au sens commun du terme.

#### 4.2.6 Notions de virtualisation

On est souvent amené pour des raisons d'efficacité à faire des calculs d'abord dans les processeurs physiques puis entre processeurs physiques, plutôt que de tout faire au niveau des processeurs virtuels. Cela est important par exemple pour écrire les routines de communications ou les opérateurs de réduction [KRS85] (§ 5.1.1.2).

Pour ce faire, on introduit la notion de collection physique associée à une collection normale comme possédant tous les caractères de la collection normale sauf qu'elle ne possède qu'un élément par processeur de la machine physique. Le nombre d'éléments de la collection physique est donc en  $1/vp\_ratio$  de la taille de la collection normale. Une variable est déclarée comme appartenant à une collection physique simplement en rajoutant devant le nom de la collection le mot-clé `physical`.

Afin de préserver une sémantique raisonnable aux manipulations faisant intervenir une collection et sa collection physique associée, ces manipulations doivent être confinées dans des blocs `with` qui définit une boucle de virtualisation explicite. Puisque celle-ci est explicite, on peut obtenir l'index de cette boucle par l'opérateur `□`. Celui-ci n'entre pas en conflit avec son homonyme de communication puisqu'une boucle explicite de virtualisation ne peut contenir de communications ou d'effets de bords affectant d'autres collections.

Un exemple d'utilisation de `with` est indiqué sur la figure 4.6 où on doit compacter tous les éléments du vecteur `un_vecteur` dans `vpack` selon le vecteur booléen `condition`. L'opérateur `□` sert ici pour conserver en mémoire la place originale dans `un_vecteur` des éléments comprimés en attente d'une décompression future du vecteur. Ce genre de routines est utile si on a de gros calculs à faire sur des variables parallèles très « creuses » : pour des raisons d'efficacité d'utilisation de la machine on a tout intérêt à compresser les données dans des variables pleines permettant une utilisation maximale des processeurs parallèles de la machine.

Il est un autre cas où on veut modifier le comportement standard de la virtualisation en POMPC : puisqu'on ne fait pas d'analyse interprocédurale, le compilateur ne peut pas voir par exemple qu'une fonction ne possède aucun effet de bord susceptible d'entraver le fonctionnement normal de la virtualisation à son voisinage. C'est pourquoi tout appel de fonction se traduit par une rupture le cas échéant de toute boucle de virtualisation.

---

```

vecteur double vpack; /* Vecteur qui va contenir le vecteur comprimé. */
vecteur int adr_vpack; /* Permettra une décompression future. */
physical vecteur int index; /* Un index unique sur chaque processeur. */

index = 0;
where (condition) /* On comprime suivant le vecteur booléen */
with (vecteur) { /* condition de collection vecteur. */
    adr_vpack[index] = . ; /* On se rapellera de la place du
                            indexieme 'el'ement dans le vecteur d'origine. */
    vpack[index++] = un_vecteur; /* Range l'el'ement de */
} /* un_vecteur dans la première place libre de vpack. */

```

---

FIG. 4.6 - Virtualisation explicite dans une routine de compaction de vecteur.

Cela est gênant car souvent on sait très bien qu'une fonction ne possède pas de tels effets de bord. Pour en assurer le compilateur, il suffit de préciser le mot-clé `spmd` devant la déclaration de la fonction en question pour que le compilateur ne casse pas une boucle de virtualisation comprenant un appel à cette fonction.

Cela est particulièrement utile pour l'écriture des fonctions de la bibliothèque standard mathématique et permet de s'affranchir des passages de variables parallèles après recopie : au niveau de chaque itération de boucle, une variable parallèle est passée élément par élément et peut donc se contenter d'un registre du processeur au lieu d'une variable temporaire parallèle sur la pile beaucoup plus coûteuse en mémoire.

#### 4.2.7 Les bibliothèques

Un certain nombre de bibliothèques sont fournies au même titre qu'il en a en C standard :

- une bibliothèque permettant un démarrage des programmes avec une initialisation correcte (équivalent parallèle du `crt0.o`);
- la bibliothèque mathématique parallèle qui est la traduction de `libm.a` pour les variables parallèles ;
- une bibliothèque de spécification et de contrôle de collections ;
- des fonctions de communications plus complexes ou plus spécifiques à une architecture donnée ;
- des fonctions d'entrée-sortie parallèles sur des fichiers ;
- une bibliothèque de fonctions graphiques permettant principalement « d'ouvrir des fenêtres » et de visualiser des variables parallèles à l'intérieur, non seulement indispensables pour la visualisation des résultats mais aussi très pratiques pour mettre au point les programmes ;
- et donc une bibliothèque liée au débogueur que nous aurons l'occasion d'apercevoir en § 8.4.1.

Comme ces fonctions de bibliothèques sont généralement beaucoup moins utilisées que le reste du langage, il nous a semblé préférable de suivre la philosophie du langage C et de ne pas surcharger à outrance le langage.

En outre, comme elles constituent souvent les parties qu'il faut retoucher lors du passage du langage sur une autre machine, il est intéressant de pouvoir isoler les parties non portables du langage pour faciliter la création d'une plate-forme portable minimale à laquelle on accroche différents modules en fonction de la machine cible.

### 4.3 Conclusion

Comme on l'a déjà mentionné en § 3.7, nous n'avons considéré que les langages impératifs, mieux adaptés à notre avis au calcul scientifique habituel contenant du parallélisme à grain fin. Dans la pratique, on est souvent amené à modifier des programmes, voire des algorithmes, pour en augmenter les performances et il est, par exemple, souvent problématique de savoir où se trouve la partie qui consomme le plus de calcul dans un programme fonctionnel, ce qui rend difficile une amélioration de l'efficacité du programme.

Dans le développement de POMPC, la facilité de portabilité l'a souvent emporté sur l'obtention des performances maximales. En effet, cela aurait nécessité des développements dépassant de loin les capacités de notre petite équipe en terme d'hommes-années :

- utilisation de graphes de dépendance pour limiter les points de synchronisation sur machines MIMD ou SPMD [HQ91] ;
- analyse interprocédurale [TIF86, IJT90, ?] au niveau des collections passées en paramètre évitant parfois l'utilisation du mot-clé `spmd` et permettant de faire de nombreuses optimisations statiques ;
- parallélisation plus poussée du code pour en extraire tout le parallélisme qui a échappé au programmeur ou au moins diminuer comme ci-dessus les synchronisations ;
- optimisations précédentes en présence de pointeurs ou d'indirections [Bod90, LC91].

On constate que cela aurait demandé rien de moins que la maîtrise de l'état de l'art dans tous les domaines de la compilation, ce qui nous conforte dans notre approche conservatrice. Le fait que de nombreuses choses soient gérées dynamiquement en POMPC est la conséquence de cette volonté pragmatique d'avoir un langage et un compilateur qui fonctionnent à un coût raisonnable.

Bien entendu, cela ne signifie nullement que le langage et le compilateur ne puissent pas être étendus et améliorés. On peut proposer par exemple :

- un mécanisme de gestion de collections creuses et de sélection plus souple du style `FORALL` [ALS90] ou tout au moins une fonction `triplet` qui est souvent la plus utilisée afin de séparer plus l'espace d'itération de l'espace de définition des variables. Cela revient de manière plus générale à séparer l'espace d'itération de l'espace de définition des collections ;

- un mot-clé ou une fonction permettant de modifier de manière incrémentale la taille d'une collection ;
- rajouter la possibilité de répartition cyclique de certaines dimensions d'une collection sur les processeurs ;
- peu de choses sont gérées statiquement en POMPC car on veut qu'un programme puisse fonctionner sans recompilation sur une même machine mais avec un nombre de processeurs différents. Néanmoins, on pourrait rajouter une option de compilation pour des optimisations prenant en compte une machine avec un nombre de processeurs fixe. Mais cela nécessitera de faire les optimisations statiques beaucoup plus poussées de parallélisation tenant compte aussi des temps de communication, etc. [McK92].

La portabilité du langage est démontrée avec l'existence de compilateurs pour de nombreuses machines — pour machines séquentielles UNIX, de machines parallèles SIMD (CM-2, MP-1) et MIMD (iPSC/860 avec le travail de Thierry PORCHER) — et l'intérêt pour ce langage dans la communauté scientifique dépasse largement le projet POMP, ce qui semble indiquer que la conception d'un nouveau langage répond à un besoin *a posteriori*.

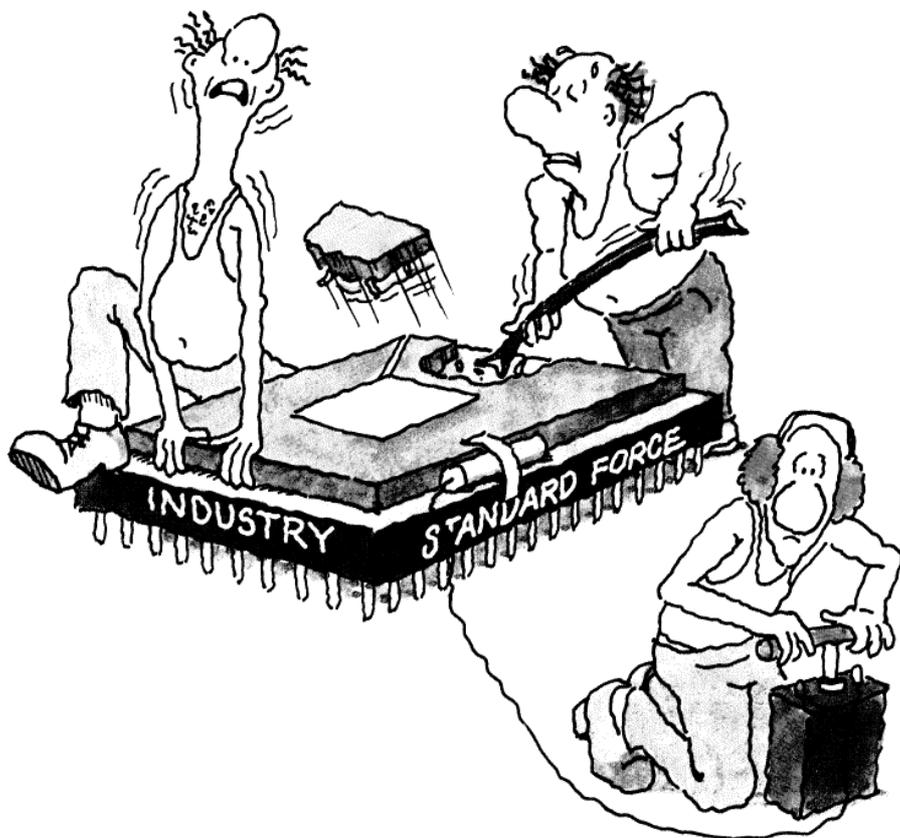
POMPC est une bonne base portable pour le développement de langages de plus haut niveau de parallélisme ou plus abstrait au niveau du parallélisme et peut servir comme langage intermédiaire à des compilateurs paralléliseurs, notamment de FORTRAN. Mais on peut tout aussi bien travailler en POMPC à un niveau proche de la machine en supprimant la virtualisation qu'à un niveau plus élevé grâce à la virtualisation, très utile sur des machines ne le proposant pas. Et en ce sens on pense donc avoir réussi à traduire les caractéristiques intéressantes du C dans le domaine du parallélisme.

Un travail futur est donc d'essayer d'unir les deux philosophies différentes qui semblent exister dans le domaine de la compilation pour machines parallèles : d'une part les paralléliseurs qui font appel à des techniques subtiles travaillant sur les graphes de dépendances et d'autre part les langages à parallélisme explicite qui permettent un contrôle plus fin car exprimé dans la syntaxe afin de simplifier le compilateur. Les deux philosophies ont chacune à apprendre quelque chose de l'autre : la première peut récupérer la syntaxe parallèle de la deuxième pour avoir des conseils de parallélisation et la seconde peut récupérer de la première tout le système de test de dépendances pour faire beaucoup plus d'optimisations.

Maintenant que nous avons décrit le langage de base de notre machine, nous allons pouvoir détailler l'architecture permettant de faire fonctionner des programmes écrits en POMPC.

Chapitre **5**

# Les processeurs élémentaires



*“In general, we believe that it’s possible to make two major mistakes at the outset of a project like C.mmp. One is to design one’s own processor; doing so is guaranteed to add two years to the length of the project and, quite possibly, sap the energy of the project staff to the point that nothing beyond the processor ever gets done. The second mistake is to use someone else’s processor. Doing so forecloses a number of critical decisions, and thus sufficiently muddies the water that crisp evaluations of the results are difficult. We can offer no advice. We have now made the second mistake<sup>a</sup> — for variety, next time we’d like to make the first!”*

<sup>a</sup>Twice, in fact. The second multiprocessor project at C-MU, Cm\*, also uses the PDP-11.

[Wlh81, page 276]

**S**E chapitre va traiter d’un problème existentiel inhérent à l’architecture des ordinateurs, à savoir s’il faut concevoir un processeur « maison » ou plutôt utiliser un processeur commercial. Il s’agit là d’un problème d’autant moins trivial qu’on s’attaque à des machines dont la structure sort un peu des sentiers battus, telles les machines SIMD.

Mais avant tout, regardons ce que nous attendons des processeurs élémentaires de la machine SIMD ciblée et précisons les caractéristiques comme leur taille, ce qui est nécessaire pour faire du SIMD, la technologie, l’intégration, etc.

## 5.1 Caractéristiques générales

Dans la suite de la discussion nous resterons assez conservateurs car nous nous restreindrons aux ordinateurs déterministes travaillant en base 2 [Rei60] pour des raisons de rapidité et de compacité raisonnable [PDG087].

Les technologies évoquées seront celles que l’on peut utiliser couramment<sup>1</sup> et sont bien maîtrisées afin de conserver un critère de choix clair et actuel.

### 5.1.1 Largeur des processeurs

Une des questions fondamentales semble être la manière d’obtenir une puissance de calcul donnée : doit-on mettre beaucoup de processeurs peu puissants ou moins de processeurs mais individuellement plus puissants ?

Grossièrement on peut estimer que si on a une application bien parallèle au niveau des données on puisse écrire l’approximation<sup>2</sup> [Bre74]

$$P = Np$$

où la puissance  $P$  de la machine serait en fait le produit de la puissance  $p$  d’un processeur élémentaire par le nombre de processeurs  $N$ .

1. Cela semble écarter les logiques faites à base de tores ferrites [BC63] ou utilisant la mécanique des fluides [Gla63], pourtant si astucieuses...

2. Cette équation n’a rien avoir avec une équation mythique homonyme.

En fait, cette décision peut même être court-circuitée si on choisit comme but l'étude du parallélisme hypermassif : finalement peu importe si la machine est incapable de faire quelque chose ou pas, le but est qu'elle soit très parallèle avec le maximum de processeurs possibles pour en étudier les conséquences. Il suffit d'écrire  $N = P/p$  et  $p \rightarrow 0$  pour construire la machine avec beaucoup de processeurs. Mais il ne s'agit pas ici de notre centre d'intérêt.

### 5.1.1.1 Efficacité calculatoire

Il semble crucial d'avoir des performances en calcul plus que raisonnables si on en croit l'optique commerciale des calculateurs parallèles : devancer les ordinateurs vectoriels pour les applications scientifiques. Construire un ordinateur sans offrir de fortes performances en calcul flottant double précision semble donc commercialement suicidaire à moins de viser les créneaux très spécialisés (traitement d'image, traitement du signal,...) comme le font WAVE TRACER et MASPAR ou encore [Ung58]. On peut penser qu'un bon départ pour faire une machine parallèle puissante est de partir de processeurs puissants.

Mais à ce niveau il faut mentionner que la plupart des machines SIMD sont spécialisées pour des applications de type traitement d'images [Cas85], qui traitent des images de 1 bit par pixel. Malheureusement ce type d'applications est limité aux fonctions booléennes ce qui perd beaucoup de son intérêt. Dès qu'on veut traiter une image en niveaux de gris ou en couleur, il faut faire des calculs sur 8 bits et plus, pour des opérations aussi simples (mais coûteuse) que des convolutions par exemple. Or la plupart du temps, ce sont tout de même ce type d'applications qui sont visées [KPDL<sup>+</sup>79, ZD91] et on n'exploite pas le parallélisme trivial sur le codage des données [Dou89] : on travaille sur des données sur 32 ou 64 bits et avant d'essayer d'extraire du parallélisme « difficile » autant profiter déjà du parallélisme « facile » bien maîtrisé dans les processeurs modernes.

On peut lire dans [RB89, page 744] au sujet du bas niveau des instructions d'une machine SIMD :

*« A massively parallel SIMD **must** operate at this level **since** its basic elements are only simple, one-bit functional elements. »<sup>3</sup>*

On trouve un argument de ce type aussi dans [Fly72] si bien qu'il semble y avoir une relation directe entre le choix du mode SIMD et la faible largeur du processeur élémentaire, souvent 1 bit, comme l'ont été la machine décrite dans [Ung58] et la machine SOLOMON [SBM62], parmi les toutes premières machines SIMD conçues. Effectivement on peut penser qu'une malédiction empêche la création de machines SIMD commerciales à gros grain si on considère les machines de recherche ILLIAC IV [BBK<sup>+</sup>68, Hor82], BSP [Sto77, KS82, LV82], OPSILA [Aug85, ABD90] et PASM [SSDK84, SSKD87], alors qu'il existe de nombreuses machines SIMD à grain fin qui sont commercialisées : Connection Machine 2 [Thi87a, TR88], MASPAR 1 [Bla90b, Bla90a], WAVE TRACER DTC [Jac90], AMT-DAP [AMT89], GAPP [NCR84] et encore plus de machines de recherche, comme PROPAL 2 [?, Rap83], BLITZEN [BDR87], etc. Dans le cas de la CM-2, il faut nuancer l'affirmation car celle-ci possède un coprocesseur flottant 32 ou 64 bits partagé par 32 processeurs 1 bit, ce qui ne va pas sans lui rajouter quelques caractères baroques.

3. Les mots mis en valeur ne l'étaient pas dans le texte original bien entendu.

Mais la largeur des processeurs dépend beaucoup des performances en calcul que l'on veut obtenir et sur quel type de données. S'il est clair que pour faire des opérations 1 bit un processeur 1 bit est optimal, pour faire des opérations sur 32 bits, il vaut mieux avoir un processeur 32 bits. En effet, si on veut faire une addition de deux nombres entiers de 32 bits sur un processeur 1 bit, cela prendra de l'ordre de 32 cycles, ce qui est raisonnable puisqu'on peut argumenter qu'on pourra toujours mettre plus de processeurs 1 bit sur la même surface de circuit intégré qu'un processeur 32 bits. Par contre si on veut effectuer maintenant une multiplication, cela prendra de l'ordre de 1024 cycles sur un processeur 1 bit : l'équivalence en surface et en efficacité n'est plus du tout la même !

La comparaison entre les processeurs 1 bit et les processeurs flottants 64 bits est encore pire. Cela s'explique par le fait qu'un processeur à gros grain possède des structures arborescentes câblées pour effectuer les calculs en  $\mathcal{O}(\log n)$  là où on a des  $\mathcal{O}(n)$  pour les processeurs 1 bit [?, ?].

Certaines machines ont des petits processeurs mais avec quelques caractéristiques permettant d'accélérer les calculs en flottants : registre flottant, instructions permettant de gérer des morceaux de mantisse ou d'exposant. C'est ce que l'on retrouve dans la machine MASPAR MP-1 qui est construite à base de processeurs 4 bits [Bla90b, Bla90a].

Les bons avocats auront remarqué qu'on ne compare que les processeurs sur des données 32 ou 64 bits et non 1 bit. Certes, mais un bon processeur à gros grain sait faire des opérations binaires bit à bit sur des entiers de 32 ou 64 bits et donc fait 32 ou 64 opérations binaires en 1 cycle, lire ou écrire 32 ou 64 mots de 1 bit par cycle en mémoire, et communiquer dans une direction par un décalage du mot de 32 ou 64 bits plus la gestion du bit qui entre et qui sort.

Cet aspect a d'ailleurs été utilisé pour émuler la machine BLITZEN (1 bit) sur un CONVEX (64 bits) en assembleur (!) [RB89]. Mais pourquoi, alors, vouloir fabriquer des machines avec des processeurs 1 bit alors que les gros processeurs sont mieux *a priori* ?

Tout simplement, SIMD est souvent associé à « conception d'un processeur spécialisé » [Fly72] comme on l'a vu en § 2.2.3 et il est beaucoup plus facile de faire un processeur 1 bit dont le motif peut être répété plutôt qu'un gros processeur 32 bits avec toute la partie calcul flottant. Les équipes de recherche universitaires n'ont souvent pas la masse critique suffisante pour mener à terme la conception d'un gros processeur comme peuvent le faire des équipes industrielles de l'ordre de 100 personnes.

On repousse donc la difficulté au niveau de la programmation intensive de micro-code, au prix d'une baisse des performances.

Il semble intéressant d'étudier l'histoire de la Connection Machine pour argumenter sur la taille des processeurs élémentaires. La première version, basée sur des processeurs 1 bit, était censée résoudre des problèmes de réseaux sémantiques et les langages étaient de type fonctionnel (\*LISP) [Hil85], ce qui était logique pour une machine développée dans un laboratoire d'intelligence artificielle.

Ensuite est venue la CM-2 qui était dotée en plus de coprocesseurs flottants 32 et 64 bits qui ont permis de vendre la machine comme supercalculateur. Des langages comme C\* et CM-FORTRAN ont été développés afin de programmer plus « classiquement » des applications scientifiques. Dernièrement, une nouvelle version plus efficace de ces compilateurs se contente d'utiliser les processeurs flottants et plus du tout les processeurs 1 bit [DKMS90, Sab92] ! Ces compilateurs sont dénommés « *slicewise* » par opposition à la version « *fieldwise* » car ils adressent directement la mémoire en paral-

lèle sans passer par une lecture séquentielle des bits dans les processeurs 1 bit<sup>4</sup> et une conversion série-parallèle bidirectionnelle niveau de l'interface avec les co-processeurs flottants<sup>5</sup> (voir plus loin les problèmes d'adressage dans la section 5.1.1.5). Même si la CM-2 possède des coprocesseurs flottants, l'utilisation simultanée de ceux-ci avec les processeurs 1 bit est gênée par la nécessité de faire constamment une adaptation de format série au niveau des processeurs 1 bit et parallèle au niveau de la mémoire et des coprocesseurs. Cette conversion prend du temps et pénalise l'utilisation du coprocesseur. En outre, il faut au moins autant de données que de processeurs 1-bit tandis que si on n'utilise que les coprocesseurs, aucune conversion n'est nécessaire et il suffit d'avoir une donnée par coprocesseur, donc la machine se contente d'un parallélisme moins élevée.

Sur les 145000 portes logiques que comportent un nœud de CM-2 sans la mémoire (32 PES 1 bit et un coprocesseur flottant), le flottant en occupe 40% contre 5% pour les processeurs 1 bit. Ces derniers ne sont plus du tout utilisés mais le gain en retour est considérable : un facteur 10 [Sab92] car les 40% formant le coprocesseur sont bien mieux utilisés.

Enfin la dernière version de la machine, la CM-5, a vu la disparition des processeurs 1 bit au profit de processeurs SPARC 32 bits associés à quatre processeurs flottants. La machine n'est plus SIMD qu'au niveau de chaque nœud et on peut la programmer suivant un modèle aussi bien SIMD que MIMD. On est loin du parallélisme à grain fin de [Hil85].

Pour résumer déjà à ce niveau, on peut dire que pour faire une machine très puissante non spécialisée, il faut partir de processeurs très puissants et non spécialisés.

### 5.1.1.2 Efficacité algorithmique

Mis à part les algorithmes purement séquentiels qu'on ne sait pas du tout exécuter sur plus d'un processeur et les algorithmes totalement parallèles, il reste une classe d'algorithmes qui sont assez parallélisables pour être accélérés sur une machine parallèle mais qui ne le sont pas suffisamment pour l'être de manière efficace sur une machine ayant autant de processeurs que le problème a de données.

Dans ce cas, il est beaucoup plus intéressant d'avoir moins de processeurs plus puissants plutôt que beaucoup de processeurs peu puissants car cela limite le nombre de communications nécessaires entre les processeurs, souvent *le* facteur limitant des machines parallèles.

Considérons des calculs sur graphe tels que calculer la somme d'un vecteur ou d'une liste chaînée en parallèle et leurs opérations préfixes associées, algorithmes couramment utilisés en calcul parallèle<sup>6</sup>.

---

4. Chaque bit d'un mot accédé en mémoire par un processeur est stocké à une adresse consécutive du bit précédent. C'est en ce sens que l'accès est fait en série. Cela pose des problèmes d'indirection complexes.

5. Il est amusant de constater que cette fonction est très courante en informatique : elle sert donc dans la CM-2 pour l'interface mémoire, dans le GAMMA 60 sous forme d'une matrice de tores en ferrite pour transposer les codes lus ou à écrire sur les cartes perforées (l'appareil les lit sous forme de lignes alors que les codes ont un sens seulement par colonnes) [Bul57, page 38], dans POMP sous le nom de PAV au niveau des entrées-sorties et de la vidéo (§ 10.1.1.6 et [Ker88]). Voir aussi les applications logicielles dans [RB89] sous le non de « *corner turning* » et dans la section § ??.

6. Ou plutôt qui devrait être utilisés plus souvent à cause de leur efficacité [HS86, Ble89b, Ble89a]. Un

TAB. 5.1 - Complexité des opérations préfixes parallèles

Nombre processeurs	Nombre de pas	Puissance de la machine	Accélération	Efficacité
$n$	$\mathcal{O}(\log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}\left(\frac{n}{\log n}\right)$	$\mathcal{O}\left(\frac{1}{\log n}\right)$
$p < n$	$\mathcal{O}\left(\frac{n}{p} + \log p\right)$	$\mathcal{O}(n + p \log p)$	$\mathcal{O}\left(\frac{n}{n/p + \log p}\right)$	$\mathcal{O}\left(\frac{1}{1 + p \log p/n}\right)$
$\frac{n}{\log n}$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{\log n}\right)$	$\mathcal{O}(1)$
1	$\mathcal{O}(n)$	$\mathcal{O}(n)$	1	1
$p$	$T_p$	$p \times T_p$	$\frac{n}{T_p}$	$\frac{n}{pT_p}$

Ce genre de problèmes sur une PRAM de taille  $n$  prend un temps  $\mathcal{O}(\log n)$  à s'exécuter et un temps  $\mathcal{O}(n/p + \log p)$  sur une machine de taille  $p$  avec  $p < n$  [KRS85, Ble89b], car chaque processeur contient  $n/p$  données sur lesquelles il travaille séquentiellement en  $\mathcal{O}(n/p)$  et les  $p$  processeurs terminent en  $\mathcal{O}(\log p)$ .

Si on regarde le nombre d'opérations exécutées sur toute la machine et qu'on le compare au produit du nombre de processeurs  $p$  par le nombre de pas de temps nécessaires à l'exécution de l'algorithme, en fonction du nombre de processeur (tableau 5.1), on constate que l'efficacité de la machine décroît avec  $p$ , avec l'efficacité étant définie comme le rapport de l'accélération de l'algorithme parallèle vis-à-vis du séquentiel ramené au nombre de processeurs, l'accélération étant elle-même le rapport du temps d'exécution séquentiel par le temps d'exécution parallèle [KMC72].

Ainsi, si on construit une machine avec  $n$  processeurs, ceux-ci ne seront pas plus utilisés que si on n'en utilise que  $n/\log n$  par exemple. Or si on a moins de processeurs, on peut se permettre qu'ils soient plus puissants et la machine sera alors plus rapide. On a raisonné sur des ordres de grandeurs alors que l'effet est accentué si on considère que les  $\mathcal{O}(n/p)$  opérations se font sans communications, donc sont généralement rapides, alors que les  $\mathcal{O}(\log p)$  sont des communications et sont des opérations plus lentes.

La constatation remarquable est que si on se limite par exemple à  $\mathcal{O}\left(\frac{n}{\log n}\right)$  processeurs, on a le même ordre d'efficacité que dans le cas scalaire pour des  $n$  grands : on parle de linéarité (voir le chapitre 11) [KRS85]. Si on essaye d'avoir plus de processeurs pour exploiter pourtant le parallélisme  $n$  du problème, le rendement tombe à  $\mathcal{O}(1/\log n)$ .

---

des sujets de recherche en parallélisation automatique est d'être capable de remplacer automatiquement certaines récurrences par des opérations parallèles préfixes [KMC72, Cal91].

### 5.1.1.3 Utilisation de la mémoire

Une des questions fondamentales est de savoir quelle quantité de mémoire il va falloir mettre dans la machine. La capacité mémoire intervient à deux niveaux :

- d’abord la capacité mémoire ramenée à chaque processeur est importante car certains algorithmes, si on ne veut pas les exécuter avec du parallélisme ultrafin du style flot de données qui risquerait de toute manière d’effondrer le réseau, nécessitent un minimum de mémoire pour fonctionner sans trop de modifications<sup>7</sup>.

L’évolution de la capacité mémoire par processeur en fonction de la puissance de celui-ci est non triviale car elle dépend aussi bien des programmes que de la psychologie de l’utilisateur, suivant la règle de « l’impatience constante » (voir [GREC91] et la section 11.1.2). Cela donne néanmoins un ordre de grandeur, disons que la capacité mémoire doit généralement évoluer quelque part entre  $\mathcal{O}(p)$  et  $\mathcal{O}(p^2)$  de la puissance  $p$  des processeurs élémentaires de la machine. On peut néanmoins retenir un autre d’ordre de grandeur classique de la littérature de 1 Mo/MIPS [?];

- l’autre point est la capacité mémoire totale de la machine. Certains utilisateurs font appels à des supercalculateurs non pas pour leurs capacités numériques mais simplement parce qu’ils ont une application ayant besoin de plusieurs gigaoctets de mémoire rapide et quelques téraoctets de disques. Il faut être capable de construire une machine possédant suffisamment de mémoire centrale pour répondre aux besoins courants, et ne pas oublier les interfaces pour des disques, bien entendu.

Un autre besoin de mémoire apparaît pour certaines applications qui ont besoin d’une base de donnée scalaire ou de tables de valeurs (*look up table*). Sur une machine massivement parallèle, on ne peut se permettre de répliquer sur chaque processeur une copie de ces bases de données car même dans le cas où cela ne dépasse pas la capacité mémoire de chaque PE, cela risque d’en occuper une bonne partie.

Les solutions possibles sont soit de stocker la base de donnée sur le processeur scalaire et de subir le goulet d’étranglement au niveau de la communication entre le processeur scalaire et les PE, soit de distribuer la base de donnée scalaire, solution qui donne le meilleur résultat en débit mais nécessite encore une réception globale, voire un mécanisme d’adresse globale, des mécanismes de répartition dynamique de la base de données, de mémoire virtuellement partagée [NL91, ?], etc.

Dans le cas où on a des processeurs de taille plus importante, on en a moins mais la mémoire qui leur est associée est en général plus grande. Si on prend l’exemple de la CM-2, les PE 1 bit ont chacun 128 ko de mémoire mais si on la considère au niveau des processeurs flottants, chaque nœud possède 4 Mo de mémoire [Sab92].

Dans ce cas on peut plus facilement avoir une copie locale de la base de donnée ce qui évite toute communication : un accès à la donnée n’est qu’une indirection locale et est donc beaucoup plus rapide, d’un facteur 100 dans le cas de la CM-2 [Sab92].

---

7. J’entends par là le fait qu’on exécute un programme à parallélisme de donnée avec comme fil directeur l’« *owner computes rule* », donc qu’un processeur exécute une tranche de l’algorithme correspondant à un sous-domaine des données du problème plutôt que de réorganiser totalement le problème.

La prise en compte de ce fait apparaît en POMPC à travers le mot-clé `physical` qui sert entre autre à dupliquer localement au niveau des processeurs physiques des tables de valeurs (voir le § ??). Cela est utilisé dans le programme de gaz sur réseau (section ??).

Il faut donc avoir une capacité mémoire par nœud importante qui va à l’opposé d’une machine à grain fin. Qui dit capacité mémoire importante sous-entend qu’il sera difficile d’intégrer processeur et mémoire, seule manière de dépasser de manière définitive le goulet d’étranglement entre processeur et mémoire puisqu’à l’intérieur d’un circuit intégré on peut se permettre d’avoir des bus très larges pour compenser le temps d’accès et en pas être limités par le nombre de pattes. Remarquons néanmoins qu’en cas d’usage intensif d’indirections locales, la mémoire redevient un facteur limitant car on n’utilisera à chaque accès mémoire qu’une petite partie du mot accédé. Dans ce cas on se rapproche du débit qu’on aurait si on accédait à la mémoire par un bus externe. Mais on estime qu’en moyenne sur les applications, 80% des accès sont directs et par conséquent on est gagnant.

La contrainte d’intégration du processeur et de la mémoire sur un même circuit intégré favorise l’approche à grain fin d’une part parce que les contraintes d’intégration actuelles ne permettent pas de mettre beaucoup de gros processeurs et d’autre par parce que la capacité mémoire par processeur étant faible ne permet pas toujours d’avoir des algorithmes à gros grain.

On doit faire face au paradoxe suivant : soit on met une grosse mémoire mais on ne peut pas l’intégrer avec le(s) processeur(s) et donc le débit est limité par le bus mémoire externe, soit on intègre une mémoire avec le(s) processeur(s) et le débit est très rapide mais la capacité est faible.

Mais il ne faut pas se faire d’illusion : les contraintes de réalisation l’emporteront sur la plupart des considérations d’ordre théorique et il est fort probable qu’un coût réparti équitablement entre processeur et mémoire sera un compromis satisfaisant [Dou89], que le coût soit en terme de surface de circuit intégré, dans le cas d’une intégration processeur et mémoire sur un unique circuit, ou en prix de circuits intégrés séparés.

#### 5.1.1.4 Importance du débit mémoire et utilisation des registres

Dans toute machine où un calcul se traduit par des échanges d’informations entre des opérateurs et des mémoires, le facteur limitant est le débit mémoire qui restreint l’utilisation des opérateurs à de plus faibles performances qu’il n’est possible de le faire.

Un moyen de supprimer le facteur limitant du débit mémoire est d’intégrer processeur et mémoire dans le même circuit mais ce n’est pas réaliste car la capacité mémoire nécessaire pour exécuter la plupart des applications scientifiques empêche une intégration d’un processeur et d’une mémoire sur un seul circuit actuellement. Même en restreignant ses prétentions au niveau de la mémoire, il est difficile de réunir des intérêts industriels pour aller dans cette voie (voir § 5.3.1).

Si on considère dans un premier temps qu’une opération vectorielle de type  $a = b + c$  demande 2 lectures et 1 écriture en mémoire par opération et que la mémoire n’est pas sur le même circuit intégré que le processeur, ce qui va compter est le nombre de pattes réservées à la mémoire sur chaque circuit processeur. Une fois atteinte la saturation de ce bus mémoire, rien ne sert d’augmenter le nombre de processeurs par circuit dans le but de calculer plus vite : ils seront inutilisés.

Un moyen pour augmenter le débit mémoire est d'augmenter la largeur du bus mémoire mais on se heurte au nombre de pattes maximum possibles sur un processeur.

Mis à part le fait que c'est inutilisable en SIMD, l'intégration de plusieurs processeurs avec leur cache sur un même circuit est un moyen de dépasser en partie ce problème si l'application le permet. Mais l'évolution des processeurs actuels va plutôt vers l'intégration d'un processeur plus complexe avec plus d'opérateurs, mieux à même pour l'instant d'exploiter le parallélisme au niveau instruction d'une application pas spécialement parallèle.

En général, le cas précédent où  $a = b + c$  nécessite effectivement 3 accès mémoire est assez peu courant car souvent cette opération se trouve entourée par d'autres qui fournissent  $b$  et  $c$  ou utilise directement  $a$ . On peut donc souvent mettre certaines variables en registre si l'architecture le permet. La supériorité des machines où les opérations se font entre registres plutôt que directement entre des mots mémoires n'est plus à démontrer, que ce soit au niveau des machines vectorielle<sup>8</sup> ou des machines scalaires (RISC contre CISC). Cela est dû au fait que les registres permettent de mieux exploiter la localité des données : ils ont une taille bien plus petite que la mémoire principale de la machine mais sont en revanche bien plus rapides. Ils évitent de devoir ranger inutilement des données en mémoire centrale.

Si on considère que l'algorithme exécuté et la compilation ne changent pas en fonction de la taille et du nombre de processeurs, les processeurs devront avoir le même nombre de registres et leur taille seront identiques car la largeur des données traitées par l'algorithme aussi. Ce qui limite la puissance d'un ordinateur est le débit utilisable entre les unités de traitement et les mémoires, à travers leur hiérarchie, où sont rangées les données.

Si on diminue la taille des processeurs tout en gardant leur registres, arrive un moment où les circuits intégrés ne contiennent presque plus que des registres et que les processeurs élémentaires ne sont plus assez puissants de toute manière pour faire le travail.

Soit  $r$  la proportion du processeur réservée aux registres. On ne pourra dépasser  $\lfloor 1/r \rfloor$  bancs de registres et donc le même nombre de processeurs sans augmenter le débit mémoire ou diminuer les performances de la machine. Il suffira que ces processeurs assurent chacun au moins  $\frac{1}{\lfloor 1/r \rfloor}$  de la puissance du processeur initial pour qu'il n'y ait pas de baisse de performance. Notons toutefois qu'on ne gagne pas sur la performance globale, limitée par le débit mémoire. Bien entendu, le fait d'avoir  $\lfloor \frac{1}{r} \rfloor$  processeurs est utopique car cela signifierait de faire globalement le même travail que le processeur de départ sur une proportion  $\frac{1-r\lfloor 1/r \rfloor}{1-r}$  de la surface de silicium utilisée par le processeur initial, puisque une proportion  $r\lfloor 1/r \rfloor$  serait composé de registres !

Ainsi, dans un MC88110 les registres occupent environ 15% de la partie opérative (donc sans la MMU ou les caches) et on ne pourra pas mettre plus de 6 processeurs, chacun pouvant se permettre d'avoir une performance d'1/6 sur l'application. Mais dans ce cas, il faut être capable de faire cela sur 1/8,5 de la surface de l'opérateur initial, ce qui est non réaliste.

Une valeur raisonnable pour le nombre de processeurs par circuit est quelque part

---

8. Si on compare par exemple 2 machines de la même époque, le CRAY-1 [Rus78] (avec registres vectoriels) et le CYBER-205 [Lin82] (sans registre vectoriel) qui on finalement une puissance assez proche, ils obtiennent leur résultats avec des systèmes mémoires possédant des débits crêtes respectifs de 640 Mo/s et 4,8 Go/s.

entre 1 et  $\lfloor 1/r \rfloor$ , sachant qu'à cause du calcul en nombres flottants ce sera probablement 1, c'est-à-dire qu'on gardera le gros processeur.

On peut aussi approcher cette problématique d'un point de vu différent. Si on veut résoudre des problèmes numériques sur une machine parallèle, il faut des registres adaptés sur les processeurs, en taille et en nombre. Or, si on considère un processeur à grain fin, il faudra lui associer un énorme bloc de registre comparé à sa taille pour le faire bénéficier de « l'effet registre ». Sinon, la mémoire sera de nouveau le facteur limitant et les performances seront faibles, comme c'est le cas sur la CM-2 (sans considérer les coprocesseurs flottants). Dans la MP-1, chaque PE est associé à une réserve de registres importante qui permet d'améliorer quelque peu les performances en nombres flottants par rapport à une WAVETRACER ou une CM-1 mais qui restent faibles par rapport à la CM-2 utilisée en *slice-wise* et même en mode *field-wise*.

La comparaison des 2 modes d'utilisations de la CM-2 est intéressante à ce niveau : alors que l'utilisation des processeurs 1 bit associés aux coprocesseurs flottants empêche l'utilisation de registres et les performances sont de l'ordre de 1,5–2,5 GFLOPS, l'utilisation en mode *slice-wise* des seuls coprocesseurs à gros grain WTL6164 possédant chacun 32 registres de 64 bits permet d'atteindre 14 GFLOPS en FORTRAN [Sab92] grâce à l'effet registre et à l'économie des conversions entre les formats série et parallèle.

Il semble par conséquent qu'on ait intérêt à avoir un processeur dont la taille est adaptée aux données traitées, aussi bien au niveau des bancs de registres que de la largeur des chemins de données des unités de calcul.

Remarquons néanmoins que ce problème est inexistant sur les instructions dans le cas du SIMD puisque le bus d'instruction est factorisé pour tous les processeurs, donc aussi au niveau d'un circuit intégré : une architecture SIMD est donc implicitement de type HARVARD [BBJ<sup>+</sup>62]. Ce problème n'existe pratiquement pas non plus si on intègre plusieurs processeurs d'une machines MIMD et qu'on l'utilise suivant un modèle de programmation SPMD car alors il est fort probable que les instructions demandées par les processeurs se trouveront toutes dans le même domaine et qu'un seul cache d'instructions suffira, ou tout au moins un seul remplissage de caches multiples servira à tous les processeurs d'un circuit. C'est un argument supplémentaire en faveur du modèle SPMD. Mais bizarrement ce ne sont pas des processeurs MIMD qu'on a tendance à concentrer sur des circuits intégrés : les applications séquentielles actuelles motivent l'intégration d'un gros processeur avec le maximum de mémoire cache.

#### 5.1.1.5 Adressage local et global de la mémoire locale

L'accès à la mémoire locale nécessite, outre un flot de données, un flot d'adresses, dual du précédent. Alors que le premier donne la quantité d'informations traitées, le second donne la qualité et la souplesse d'utilisation de ces données [Dou89].

Les problèmes de programmation des machines où chaque processeur ne pouvait pas contrôler localement sa mémoire a montré que beaucoup d'algorithmes nécessitent un indexage local pour être exécutés efficacement, même si certains comme [Hil85] ont clamé pouvoir l'émuler en un temps constant<sup>9</sup>.

---

9. Qui n'est autre que la taille de la mémoire locale ou le nombre de processeurs, si on récupère sur le processeur scalaire toutes les adresses désirées par les processeurs et qu'on fait générer par le processeur scalaire tous les accès aux adresses voulues ! Comme quoi il faut toujours se méfier des constantes, surtout celles qui grandissent vite avec le temps, comme la capacité mémoire ou le nombre

C'est un point difficile à émuler efficacement s'il n'est pas prévu au départ dans la machine et c'est pourquoi la machine MPP [Bat80a] a évolué vers la machine BLITZEN [BDR87].

Un point qui est souvent oublié est qu'il ne suffit pas d'avoir une mémoire adressable, encore faut-il l'adresser rapidement ! Si on veut un adressage local efficace, il faut un système pour calculer rapidement les adresses, donc avoir une UAL d'une taille comparable à la capacité d'adressage sinon on sera obligé de sérialiser les calculs d'adresse, comme sur toutes machines SIMD à grain fin. Cela implique des choix dans le style de programmation (voir en particulier les problèmes de conditionnement SIMD qui peuvent reposer sur le fait qu'on a un adressage rapide de la mémoire, section 7.1.5).

On assiste alors à la création d'architectures baroques où on a des processeurs 1 bit dotés de registres d'adressage « intelligents », c'est à dire contenant un incrémenteur de la largeur du registre, beaucoup plus puissant donc que le processeur 1 bit utilisé pour faire les calculs [Ni90]. Dans ce cas pourquoi ne pas transférer cette logique des registres d'adresse dans le processeur, pour que toute la machine en profite ?

De toute manière un autre problème survient, dès qu'on intègre plusieurs processeurs dans un même boîtier : celui du multiplexage des bus mémoire. Si on considère que le produit largeur  $l$  des processeurs par leur nombre  $n$  par circuit est constant, il n'y aura pas encore de problème pour les données : le bus de données aura la même taille<sup>10</sup>. Par contre, dès qu'on regarde le bus d'adresse, celui-ci ne peut pas être subdivisé : soit la mémoire de taille  $M$  est globale à tout le circuit et il faut un bus d'adresse unique de  $\lceil \log_2 M \rceil$  bits d'adresse toujours mais qu'il faudra multiplexer temporellement, soit on a  $n$  mémoires indépendantes de taille  $M/n$  qui nécessiteront  $n$  bus d'adresse de  $\lceil \log_2(M/n) \rceil$  bits de large, ce qui est considérable ! Seule la première méthode est réalisable et par conséquent l'adressage indirect a des performances divisées par le nombre de processeurs par circuit. C'est la limitation principale de la CM-2 qui pousse à l'abandon de l'usage des processeurs 1 bit au profit de l'unique usage des coprocesseurs flottants en mode *slice-wise* [Sab92].

Mais pour des raisons d'économie de transistors, outre le fait qu'on puisse espérer que les indirections locales soient peu courantes dans les programmes qui auront à être exécutés sur la machine cible, on peut considérer que le ralentissement des indirections locales au niveau des PE est compensé par le nombre plus élevé de processeurs par circuit intégré, au niveau des accès en registre s'il y a, puisque c'est le débit mémoire qui va être limitant de toute manière.

Enfin, si on décide d'intégrer la mémoire aux processeurs, on peut se permettre d'avoir un adressage local quel que soit le grain puisque de toute manière les bus mémoire ne coûtent plus au niveau des pattes du circuit.

En ce qui concerne l'adressage global, c'est à dire d'avoir un mécanisme permettant une indirection non seulement sur la mémoire mais en plus sur le numéro de processeur, cela revient à pouvoir adresser toute la mémoire de la machine, ce qui nécessite donc des calculs d'adresse encore plus gros.

---

de processeurs...

10. Évidemment, ce n'est plus le cas si on considère que c'est par exemple  $l\sqrt{n}$  qui est constant. C'est donc un problème à considérer pour évaluer certaines machines.

### 5.1.1.6 Coût du réseau

Une machine qui possède des PEs plus petits interconnectés entre eux, ce nécessite un réseau plus important pour relier les PEs qui sont plus nombreux. Comme le coût d'un ordinateur parallèle dépend fortement de celui de son réseau, on a intérêt à avoir le réseau le plus simple possible, tout en conservant des performances honorables.

Alors que le coût du réseau dépend en général linéairement de la largeur des chemins de donnée et plus que linéairement du nombre de PEs<sup>11</sup>, il est fort probable qu'une machine avec des PEs moindres en nombre mais plus large sera plus facile à réaliser et plus intéressante d'un point de vue pécuniaire au niveau du réseau.

### 5.1.1.7 Coût de la machine

Il s'agit d'un des arguments de poids avancé par les adeptes du parallélisme depuis la fin des années 40 déjà : on va se heurter aux limites technologiques, aussi bien physiques que financières vu le prix des technologies extrêmes, et le seul moyen de les dépasser est le parallélisme. Même si en 45 ans on a surtout assisté au développement de machines parallèles non commercialisées, le tournant semble être pris même par les supercalculateurs « classiques » comme le CRAY Y-MP C90 qui peut avoir jusqu'à 64 processeurs.

L'argument de prix semble favoriser une machine faite de beaucoup de composants choisis parmi quelques modèles seulement plutôt qu'une machine construite à partir de moins de composants de types plus nombreux car on peut bénéficier du facteur d'échelle industriel qui diminue les coûts de fabrication, tout particulièrement si les circuits sont développés spécialement pour la machine étudiée. Remarquons tout de même que ce même argument a été défendu pour développer des machines non parallèles de plus en plus puissantes en exploitant le phénomène d'échelle à technologie identique, argument techno-économique connu sous le nom de loi de GROSCH.

L'exception qui confirme la règle est le CRAY 1 qui était fabriqué à partir de 4 types de circuits intégrés seulement [Rus78] ! Mais il s'agissait aussi de circuits du commerce, l'intégration de la machine étant assez faible. Le prix se retrouvait sur technologie mise en œuvre pour faire fonctionner ces circuits permettant les vitesses d'horloges les plus rapides. Par contre, le CRAY 3, constitué de 480 types de circuits intégrés de fabrication maison en AsGa confirme la règle si on observe tous les problèmes qu'a cette machine à être construite, pour peu qu'elle le soit un jour.

En plus du paramètre d'échelle permettant d'utiliser beaucoup de composants identiques, il faut donc se restreindre au niveau de la vitesse d'horloge plutôt que d'essayer d'utiliser à tout prix les composants les plus rapides du moment.

En fait, on constate qu'on ne peut pas appliquer la loi de GROSCH, qui dit que le coût d'un ordinateur de puissance  $p$  est en  $\Theta(p^g)$  où  $g$  est le coefficient de GROSCH, proche de 0,5, sur des ordinateurs de différentes classes [ED85], par exemple vectorielle par rapport à une autre massivement parallèle ou à un micro-ordinateur qui sont apparus depuis. On constate un effet inverse entre différentes technologies : par exemple l'efficacité d'un micro-ordinateur est de 80 MIPS/M\$ comparée à 2,5 MIPS/M\$ pour les

---

11. Comme exception on peut citer les réseaux basés sur les grilles et les tores, le CCC, etc. Comme cela n'est pas magique non plus, les performances de ces réseaux évoluent moins vite que les besoins des processeurs. On aura l'occasion de voir ces aspects plus en détail dans le chapitre 9.

supercalculateurs selon [ED85], soit un rapport 32 en faveur des micro-ordinateurs ! Evidemment, si après on veut faire coopérer plusieurs de ses micro-ordinateurs pour avoir des performances équivalentes à celle des supercalculateurs lorsqu'on dispose d'applications s'y prêtant bien, il faut rajouter du matériel dont le coût fait baisser l'efficacité par unité monétaire. Mais le rapport 32 exposé nous laisse une marge confortable pour développer des machines parallèles puissantes, même avec des réseaux qui représentent une bonne partie du prix des machines.

Cela peut peut-être s'expliquer par le fait que si on a un besoin absolu de puissance, on peut développer une nouvelle classe de machine avec une autre technologie qui arrivera à se vendre très cher à cause de la demande impérative et spécifique.

Si on considère une classe de machines parallèles, on a intérêt à avoir à puissance égale la machine composée des processeurs les plus gros à technologie comparable, à savoir par exemple une machine où un processeur est forcément compris entièrement dans un seul circuit intégré, ce qui va du grain fin au grain assez gros.

### 5.1.2 Coût technologique et balance processeurs-mémoire

On peut aussi considérer une autre fonction de coût, plus technologique, pour nous permettre d'estimer l'équilibre à trouver entre la quantité de processeurs et de mémoires à mettre dans notre machine. Comme exemples de fonctions de coût on peut considérer entre autres le nombre de pattes par carte, le nombre de transistors, la dissipation thermique, la surface de circuits intégrés, etc. Cette grandeur est constante sur toute la machine et est considérée comme le facteur limitant principal.

Si on choisit d'avoir un coût de la mémoire égal au coût des processeurs on peut mener le raisonnement astucieux suivant [Dou89] pour chaque application :

- soit on s'est trompé au niveau des processeurs et on n'en n'a pas assez. Une certaine quantité de mémoire est inutilisée mais de toute manière on ne pouvait pas mettre plus du double de processeur ;
- soit la machine pêche par son manque de mémoire et a trop de processeurs. Mais le coût empêche de mettre là aussi plus du double de mémoire.

On constate que dans les deux cas on n'est au pire qu'à un facteur 2 en performance<sup>12</sup> de l'équilibre à trouver entre le nombre de processeurs et la mémoire de la machine. Comme cet équilibre dépend des applications, le choix du partage 50:50 est raisonnable, sachant que le facteur 2 est pessimiste car rares sont les applications qui peuvent se passer totalement de mémoire ou de processeur !

La taille mémoire à mettre par processeur dépend de la complexité de celui-ci à travers la fonction de coût et la taille globale de la mémoire est déterminée par la technologie mémoire qui permet de retrouver la capacité possible à partir de la fonction de coût.

En pratique, plusieurs facteurs fortement limitants peuvent intervenir, ce qui peut amener à s'éloigner du facteur 50:50 pour chaque fonction de coût mais néanmoins rester en moyenne autour de ce facteur donne déjà un bon compromis.

---

<sup>12</sup>. En supposant que la puissance de la machine dépend en gros linéairement du nombre de processeur ou de la quantité de mémoire.

On peut appliquer un raisonnement similaire en ce qui concerne le réseau en considérant par exemple que le nombre de pattes des processeurs dédiées à la mémoire et le nombre de pattes dédiées au réseau est un facteur limitant majeur. Dans ce cas les nombres de pattes auront intérêt à être équilibrés.

### 5.1.3 Caractéristiques SIMD

Un processeur SIMD a tout de même quelques spécificités qu'on ne retrouve pas toutes dans les processeurs commerciaux :

- chaque instruction lue par une unité de séquençement est envoyée à tous les processeurs de la machine. Une architecture de type HARVARD où il y a bien séparation des bus d'instructions et de données, vus ici comme bus d'instructions global et bus de données local ;
- comportement synchrone au niveau de l'ensemble de la machine. Chaque instruction doit être exécutée en un temps égal sur tous les processeurs, quel que soit le contexte local dans lequel elles viennent s'exécuter<sup>13</sup>. Sinon, on sera obligé de synchroniser la machine sur le temps d'exécution le plus long, ce qui ralentit et complique la machine, comme c'est le cas de la machine PASM [SSDK84, SSKD87] ;
- un système contrôlant l'activité des processeurs pour le contrôle de flot local, correspondant à la description du chapitre 7 ;
- outre les indirections en mémoire locale, il peut être utile d'avoir une indirection sur les registres des processeurs si on possède des bancs de registres importants [Dou89] ;
- des instructions adaptées au parallélisme de données telle que les accès post-incrémentés pour gérer la virtualisation ;
- système de gestion des communications ;
- système de gestion des entrées-sorties ;
- système capable de traiter les exceptions locales.

Pour ces raisons, il semble nécessaire de créer un nouveau processeur qui intégrerait toutes ces caractéristiques en plus afin de disposer d'une brique de base de construction d'ordinateurs SIMD. Le seul processeur SIMD commercialisé est déjà ancien et n'est pas très puissant selon les critères actuels car même si chaque circuit contient 72 PES, ce ne sont que des processeurs 1 bit [NCR84].

### 5.1.4 Caractéristiques supplémentaires

De plus, on peut vouloir certaines caractéristiques qu'on retrouve dans les processeurs habituels pour atteindre leurs performances :

- pipeline et confluence pour augmenter le débit d'instructions ;

---

13. Par exemple, on ne veut pas que le temps d'exécution d'une instruction de décalage, dont le rang du décalage est calculé localement, dépende de ce dernier. C'est souvent le cas dans les processeurs microcodés.

- VLIW ou superscalaire qui augmente le nombre d'opérateurs et d'instructions exécutées simultanément ;
- architecture de type RISC pour simplifier le compilateur et le processeur.

### 5.1.5 Conclusion sur les caractéristiques

Comme la plupart des applications visées manipulent des données sur 32 ou 64 bits, il n'y a aucune raison de ne pas exploiter ce parallélisme facile, c'est-à-dire d'avoir des processeurs à gros grain.

Même dans le cas d'applications à grain fin, le fait de devoir manipuler des adresses locales impose une contrainte en défaveur du grain fin.

Comme les applications visées par POMP sont assez générales dans le domaine massivement parallèle, des processeurs de taille adaptée aux données traitées, à savoir 32 ou 64 bits avec flottant intégré s'imposent.

Par contre le développement d'un processeur, surtout si on veut en plus intégrer sa mémoire avec, va dans le sens du parallélisme à grain fin.

## 5.2 Le développement d'un processeur : un mal nécessaire ?

Puisqu'il n'existe pas de processeur du commerce qui serait à la fois puissant et intégrerait toutes les fonctionnalités du SIMD, un peu comme existe le TRANSPUTER en tant que brique de base de machines MIMD [INM89, INM91] où est intégrée la gestion des communications et de la mémoire dynamique, on peut à l'inverse essayer d'adapter un processeur standard à la tâche requise tout en essayant de minimiser le coût architectural de l'opération.

Cette approche est justifiable pour plusieurs raisons :

- utilisation de processeurs très performants issus d'importantes équipes de développement ;
- processeur disponible facilement (« sur l'étagère ») et immédiatement ;
- frais de développement limités à son intégration dans la machine,
- environnement logiciel existant avec de bons compilateurs, un assembleur, des bibliothèques de fonctions standard, etc ;
- on peut choisir le processeur le mieux adapté à notre problème à un instant et en changer pour un autre encore plus approprié lorsque de nouveaux processeurs apparaissent sur le marché : on n'est pas prisonnier de mauvais choix effectués *a priori* ;
- on bénéficie des progrès constants de la technologie à travers l'apparition des nouvelles versions ou familles de processeurs<sup>14</sup> et on évite la dérive technologique du projet ;

---

<sup>14</sup> C'est intéressant si on considère que les processeurs voient leur puissance augmenter environ de 70% chaque année.

- les aspect d'un point de vue purement processeur (l'UAL, les registres et le contrôle de la mémoire) sont les mêmes que pour un processeur SIMD ;
- décodage des instructions intégré et donc taille de code compacte évitant d'envoyer du microcode à tous les processeurs de la machine par un bus large. A titre d'exemple, cela était fait par un bus de 200 bits dans l'ILLIAC IV [BBK<sup>+</sup>68],
- on peut envisager la présence d'une instruction `execute` permettant d'interpréter une donnée comme étant une instruction. Cela permet à peu de frais de rajouter un mode MIMD lorsqu'on en a besoin comme sur la machine OPSILA [AB86] sous le nom de `CASE`. Malheureusement cette fonction a disparue des processeurs actuels<sup>15</sup>,
- la présence de signaux d'attente sur le bus (d'instructions) peut permettre d'empêcher l'exécution de certaines instructions et ainsi faire un contrôle de flot local depuis l'extérieur du processeur,
- le pipeline et les branchements non retardés peuvent trouver une utilisation inédite dans le contrôle de flot SIMD (voir section 7.2.6.1).

Donc *a priori*, on peut penser pouvoir utiliser des processeurs « tout faits ». Mais cette méthodologie a aussi ses inconvénients :

- absence de matériel pour gérer directement les aspects SIMD ;
- intégration telle qu'on n'a plus un contrôle déterministe fin du processeur depuis l'extérieur. On sait que le processeur fournira le bon résultat mais on ne saura pas exactement dans quel ordre les instructions auront été exécutées ;
- il ne faut pas de mémoire cache qui vient amplifier le point précédent. Si une donnée est dans le cache pour certains processeurs et pas pour d'autres, les premiers prendront de l'avance et il faudra propager un signal de ralentissement à tous les processeurs en avance, compliquant et ralentissant à outrance la machine. La latence rajoutée par cette synchronisation est de toute manière bien supérieure au temps gagné par l'ajout des mémoires caches, rendant celles-ci nuisibles ;
- cela implique d'avoir une mémoire extérieure et par conséquent d'être limité par le débit mémoire ;
- son architecture pas toujours bien adaptée au parallélisme en général et au SIMD en particulier ;
- nécessite l'utilisation de beaucoup de matériel pour l'adaptation dans la machine.

Le problème le plus gênant semble être celui de l'asynchronisme des processeurs. Ainsi la machine PASM était basée sur des processeurs du commerce 68010 [SSKD87]. Mais puisque le temps d'exécution d'une même instruction sur chaque processeur n'est pas égale pour tous les processeurs, le séquenceur ne fournit pas l'instruction suivante tant que la précédente a été exécutée par tous les processeurs. L'optique était d'utiliser

---

<sup>15</sup>. Outre sur la machine OPSILA, elle était présente sur l'ILLIAC IV [BBK<sup>+</sup>68], mais seulement au niveau du processeur scalaire, ce qui perdait beaucoup de son intérêt.

d'abord ces processeurs dans un petit prototype avant de construire une machine plus importante avec des processeurs « maison » qui auraient pu être synchrones. Mais le provisoire c'est éternisé. Notons que cette approche leur a tout de même permis de construire une machine car s'ils avaient dû concevoir leur processeur dès le départ, c'est la construction de la machine qui aurait probablement jamais été menée à terme.

Mais la méthode utilisée sur la machine PASM est difficilement applicable avec les processeurs récents rapides car si ce mécanisme ralentit une machine avec des processeurs lents, il ralentira encore plus une machine avec des processeurs très rapides, car le temps de synchronisation globale, qui est borné inférieurement par la vitesse de la lumière, n'a pas beaucoup diminué depuis. En outre, rajouter une synchronisation systématique après chaque instruction est un comble pour une machine SIMD !

### 5.2.1 Processeurs en tranche

Il existe une classe de processeurs « en kit » qui pourraient s'adapter à notre problème. En particulier on peut essayer d'utiliser une UAL du commerce et rajouter autour tout ce qu'il manque pour faire un processeur SIMD.

C'est cette approche qui a été utilisée dans la machine OPSILA mais maintenant les processeurs à plus forte intégration sont plus puissants que les tranches de processeur couplées avec la logique supplémentaire [Adv88a, IDT89, Cyp90a] car il y a moins de transferts d'information à faire entre boîtiers, toujours la même limitation au niveau des pattes des circuits.

L'utilisation de technologies extrêmes (UAL de 200 MFLOPS par exemple [BIT89]) est possible mais nous éloigne de notre but en se rapprochant du gros processeur vectoriel et de ses problèmes : dissipation de la chaleur, etc.

En outre le support logiciel des processeurs en tranche se limite à la mise à disposition de microassembleurs, ce qui est loin du compilateur parallèle final. C'est pourquoi cette solution semble être un dernier recours.

### 5.2.2 Processeurs de type RISC

Actuellement, les processeurs RISC<sup>16</sup> [PP82] sont les processeurs sous forme de un à quelques circuits intégrés les plus rapides du marché et sont, par conséquent, tout désignés pour notre machine.

Les points particulièrement intéressants [CCYHJ+85, PP82, Hen84, HJG+82] en ce qui nous concernent sont :

- l'exécution d'une instruction par cycle. Le comportement du processeur est plus déterministe que les processeurs CISC<sup>17</sup> et le contrôle SIMD global de la machine est plus facile à faire, même en cas d'exception ;
- la puissance unitaire des processeurs laisse présager un gain en puissance comparable pour toute la machine, par rapport à des processeurs 1 bit bien évidemment, mais aussi par rapport à des processeurs CISC, pour des opérations sur des opérandes de 32 ou 64 bits ;

---

16. *Reduced Instruction Set Processor*: processeur à jeu d'instructions réduit.

17. *Complex Instruction Set Processor*: processeur à jeu d'instructions complexe.

TAB. 5.2 - Comparatif entre quelques processeurs du marché.

Processeur	Absence de cache intégré	Flottant intégré	Architecture HARVARD	Canaux de communication	Bibliographie
AMD29000	•		1/2 <sup>a</sup>		[Adv88b]
AMD29050◇	•	•	1/2		[Adv90a]
C400◇	•				[SM91, Int91d]
DEC21064◇		•			[Dig92a]
DSP96002	•	•	• <sup>b</sup>		[MOT89]
hyperstone					[Hyp90]
i860XR		•			[INT89a, KM89]
i860XP◇		•			[Int91b]
MC88100	•	•	•		[MOT88a, Mel89]
MC88110◇	•				
SPARC	○	• <sup>c</sup>			[Cyp89]
R3000	○	○			[Int90, NEC88]
R4000◇	○	•			[Int91a, Sie91]
T9000◇	○	•		(4 + 2) × 12 Mo/s	[INM91]
TMS320C40◇		○ <sup>d</sup>	• <sup>b</sup>	6 × 20 Mo/s	[?]

<sup>a</sup> Architecture demi-HARVARD : le bus d'adresse est partagé entre les données et le programme.

<sup>b</sup> Possède 2 bus indifférenciés.

<sup>c</sup> Rajouté sur les versions récentes.

<sup>d</sup> Non IEEE.

- comme les instructions sont toutes au même format, les champs à surcharger dans certaines instructions sont tous au même endroit, ce qui permet de réaliser des émissions scalaires simplement avec un multiplexeur.

On peut trouver plusieurs articles qui comparent les différents processeurs existants ([PW89]<sup>18</sup>, [HFYT91, Sla92]) mais dans une approche plus classique, à savoir ce pour quoi ils ont été dessinés à l'origine : les stations de travail à usage plutôt général. Mais en ce qui nous concerne, certains des avantages dans le cas des stations de travail peuvent devenir des tares pour une machine SIMD. C'est pourquoi le tableau comparatif 5.2 est présenté avec des critères plus SIMD, marqués par un « • » lorsqu'elles son présentes. Lorsque les processeurs existent en plusieurs versions ou que la caractéristique est optionnelle, la version la plus intéressante pour le SIMD est retenue et notée par un « ○ ».

Les DSP96002 et TMS320C40 ne sont pas des processeurs RISC mais des processeurs de signal (DSP) qui ont des particularités intéressantes qui les ont fait mettre ici. Malheureusement, le fait qu'ils aient plusieurs bus génériques rend l'écriture de compi-

18. Ce dernier contient quelques erreurs.

lateurs de langages standards difficile et les compilateurs qui existent ne sont pas très performants par rapport aux autres processeurs.

Un processeur intéressant est aussi le processeur RISC ARM qui inclut un bit d'activité : chaque instruction peut être exécutée ou non suivant un bit du registre de condition du processeur pouvant ainsi faciliter son intégration dans un nœud SIMD. Malheureusement ses faibles performances et son bus non HARVARD ne permettent pas de le retenir. On retrouve une particularité semblable dans le processeur ALPHA DECChip 21064 sous forme de transferts de registres conditionnels.

Les critères de choix interviennent à plusieurs niveaux cruciaux :

- le synchronisme global de la machine ;
- la performance de la machine qui nécessite un processeur flottant ;
- la compacité du nœud de la machine qui détermine la taille globale de la machine.

Chacun de ces points implique un compromis à trouver entre taille de la machine et puissance de la machine. Le fait d'avoir par exemple l'unité de calcul flottant intégrée dans le processeur apporte un gain de place appréciable.

Le fait que la machine soit synchrone interdit l'utilisation des caches mais oblige à avoir une mémoire locale très rapide sous forme de mémoire statique, plus chère que la mémoire dynamique et de capacité moindre d'un facteur 2 à 4 environ à place équivalente en comptant les boîtiers nécessaires au rafraîchissement de la mémoire dynamique. L'utilisation de mémoire dynamique banquée, ou même tout simplement de mémoire dynamique avec un mode page, n'est pas satisfaisante car là encore le temps d'accès à la mémoire dépend des accès précédents ce qui mène à une dessynchronisation. Mais l'utilisation de mémoire statique rapide n'est pas un gros problème car cela ne représente que la moitié du coût de la machine. Ce type de mémoire se retrouve dans de nombreuses machines performante, en particulier le CRAY Y-MP C90 qui contient 2 Go de mémoire à 15 ns [Cra91a].

Le choix du SIMD où on envoie le code aux PES pendant que ceux-ci accèdent à leur mémoire oblige à avoir des processeurs à architecture HARVARD avec séparation nette entre le bus d'instructions, relié au séquenceur global de la machine, et le bus de données relié à la mémoire locale. Dans le cas contraire on est obligé de multiplexer ces 2 bus à l'extérieur du processeur et cela entraîne le rajout de l'ordre de 20 circuits intégrés par PE et un ralentissement important de la machine : pendant qu'un processeur lit une instruction il ne peut pas accéder à sa mémoire de données locales.

Les architectures demi-HARVARD telles celles des AMD29000 et AMD29050 sont aussi utilisables que les architecture HARVARD puisque s'il y a partage du bus d'adresse celui-ci n'est jamais utilisé pour les instructions puisque le PE n'effectue jamais de branchement, sauf lorsque survient une exception. Il n'y a donc pas à multiplexer le bus d'adresse. Malheureusement, il faut rajouter une dizaine de circuits intégrés pour faire comprendre le mode « par à-coup » (*burst*) de ce bus à la mémoire.

Enfin certains processeurs possèdent des canaux de communications très intéressants pour l'interconnexion entre processeurs voisins. Cela évite la conception d'un circuit spécialisé gérant cette partie. Malheureusement il n'y a pas encore de circuit de routage commercial permettant de décharger les processeurs de l'acheminement des paquets de proche en proche.

Le milieu des processeurs étant en perpétuel changement, il est difficile de faire un choix de processeur à un instant donné alors qu'un processeur plus adapté peut apparaître quelques temps après. Le choix du processeur pour POMP a été fait il y a environ 3 ans, à un moment où n'existaient pas les processeurs les plus modernes (avec «  $\diamond$  » sur la figure 5.2). Néanmoins on peut constater qu'un bon choix est encore actuellement celui du MC88100 qui possède tout ce qu'on lui demande, sauf les mécanismes spécifiques de contrôle SIMD et la partie réseau.

## 5.3 Le module processeur élémentaire de POMP

### 5.3.1 Le système mémoire

La première étude de notre machine consistait à prendre la mémoire dynamique la plus dense du moment et d'en remplacer la moitié par un ou plusieurs processeurs, se partageant donc 2 ou 8 Mbits de mémoire dans une technologie actuelle [Dou89]. Pour dépasser le goulet d'étranglement de VON NEUMANN [Bac78] l'accès se faisait à travers un bus de données de 256 bits qui remplissaient une ligne de 8 registres 32 bits à chaque cycle mémoire, un peu à la manière d'un CRAY-1 [Rus78] ou d'un système à base d'ALPHA [Dig92a].

Les contraintes technologiques étaient très fortes :

- intégrer un processeur (logique rapide) avec de la mémoire dynamique (technologie particulière), donc 2 technologies différentes, sur un même circuit intégré. Il faut faire des compromis entre les 2 technologies pour en trouver une commune ;
- s'associer avec un industriel pour développer cette technologie d'intégration qui nécessite la maîtrise de la technologie mémoire dynamique de haute capacité et celle des processeurs rapides. On ne peut en effet utiliser des circuits prédiffusés du commerce intégrant de la mémoire dynamique avec de la logique car la mémoire est accédée à travers un bus étroit (8 bits typiquement, dû aux applications visées par les produits), ce qui est trop faible.

Il faut donc s'associer avec un industriel capable de réaliser le circuit complètement. Le seul industriel en Europe<sup>19</sup> est SIEMENS. Nous les avons contactés mais malheureusement pour nous ils ne se sont pas montrés intéressés outre mesure pour initier un travail sérieux de coopération.

Il a fallu reconsidérer totalement le problème étant donné l'impossibilité d'intégrer processeur et mémoire. La contrainte sur le grain fin et la faible capacité mémoire disparaissait, mais le goulet d'étranglement réapparaissait au niveau du débit mémoire. À cause de cette dernière contrainte qui nous imposait de l'ordre de 300 MIPS/carte [Dou89], il a fallu d'une part renoncer à réaliser une machine sur une seule carte mais sur plusieurs et d'autre part à utiliser de la mémoire statique pour augmenter le débit par patte de circuit intégré par un facteur 4 environ.

Du coup, le système mémoire devient très classique mais aussi très simple : chaque processeur adresse sa mémoire externe et comme la mémoire est statique, il n'y a même plus de système de rafraîchissement de la mémoire à prévoir.

---

<sup>19</sup>. Suite au transfuge des concepteurs de mémoire d'INMOS aux USA qui travaillent dans une société de conseil en mémoire...

L'inconvénient en retour est que la densité en GFLOPS de la machine baisse et que des problèmes de connectique apparaissent du fait de la répartition de la machine sur plusieurs cartes qu'il faut maintenant relier.

### 5.3.2 Le MC88100

Plusieurs versions de processeurs ont été étudiées dans l'équipe au début du projet, des versions aussi bien avec UAL flottante [dM88] qu'entière pour avoir une version plus simple plus rapidement [Dou89]. Les performances devaient être atteintes avec la technologie 2 ou 1,2  $\mu\text{m}$  du CMP alors que les processeurs commerciaux sont réalisés avec des technologies dont la largeur des transistors est de 0,8, voire 0,6  $\mu\text{m}$  et donc beaucoup plus rapide. Cette différence technologique devaient être compensée par un travail architectural conséquent. Mais il arrive un moment où une technologie meilleure moyennement utilisée est plus efficace qu'une technologie moyenne mieux utilisée, même dans un contexte hors production industrielle, car le temps de conception est fortement réduit.

C'est aussi une des raisons pour laquelle le développement d'un processeur puissant a été abandonné, en plus de l'argument de la section 5.3.1, non pas au profit du développement d'un processeur peu puissant, mais à l'adaptation d'un processeur du commerce, le MC88100 en l'occurrence comme on vient de le voir pour ces bonnes caractéristiques. Mais cela a entraîné une forte remise en cause des idées architecturales, particulièrement au niveau du système mémoire.

Le MC88100 est un processeur RISC pipeliné avec UAL flottante intégrée, délivrant une moyenne de 10 MFLOPS et 21 MIPS pour la version à 25 MHz utilisée dans notre prototype, et à architecture HARVARD qui nous permet de l'utiliser avec un minimum de circuits externes et de conserver simplement la synchronisation globale de la machine de par son absence de mémoire cache intégrée. Une machine composée de 256 processeurs nous permet donc d'obtenir environ 2 GFLOPS pour des calculs favorables.

Les bus du processeur étant aussi pipelinés, il faut rajouter deux circuits de verrous, en plus du PAL rapide de décodage sur le bus d'adresse de la mémoire locale pour pouvoir l'interfacer avec de la mémoire statique standard. En fait, maintenant ce n'est plus la peine d'utiliser de verrous car les bus pipelinés se généralisant avec les processeurs rapides, des mémoires pipelinées deviennent disponibles.

Puisque le processeur est utilisé comme un processeur SIMD, il n'y a pas de flot d'adresses d'instructions qui sort du PES et par conséquent le bus d'adresses d'instructions du MC88100 est inutilisé : on nourrit de force chaque processeur à partir d'une mémoire de code parallèle unique contrôlée par le processeur scalaire ou le séquenceur.

### 5.3.3 Les diffusions scalaires

Pour les réaliser, on utilise le mécanisme très simple qui permet au processeur de prendre des valeurs immédiates dans ses instructions. En effet, puisque certaines instructions font intervenir la valeur contenue dans les 10, 16 ou 26 bits de poids faible de celles-ci, selon les instructions.

Si le processeur scalaire est capable de surchargé avec ses propres valeurs les champs des instructions correspondantes, il pourra envoyer des « constantes » à tous les processeurs qui pourront en faire l'usage désiré. Il suffit donc d'avoir un multiplexeur au

niveau du séquenceur sur le chemin des instructions pour les PEs qui permettra de choisir si on envoie une instruction lue telle qu'elle ou avec la constante remplacée par un registre écrit par le séquenceur. Cette surcharge peut être soit déclenchée par l'écriture dans le registre de surcharge, soit par un bit de l'instruction de la machine, VLIW comme on aura l'occasion de le voir par la suite.

### 5.3.4 Le circuit de contrôle et de communication

Afin d'adapter le processeur au mode SIMD, il est nécessaire de rajouter un circuit « maison » qui s'occupe du contrôle du processeur en SIMD, des communications avec les processeurs voisins formant le réseau (voir le chapitre 9) et les entrées-sorties. Ce circuit sera dénommé dans la suite par le terme « HyperCom ».

#### 5.3.4.1 La gestion de l'activité

La partie constituant le contrôle SIMD sera discutée en détail dans le chapitre 7 avec le système de gestion de l'activité de processeur, chaque processeur pouvant exécuter ou pas une instruction suivant une condition locale.

#### 5.3.4.2 Les entrées-sorties parallèles

Pour les entrées-sorties il est nécessaire d'assurer un minimum de débit puisqu'on veut pouvoir se connecter directement à un écran vidéo, ce qui peut représenter jusqu'à 800 Mo/s (32 bits à 200 MHz) envoyés de manière assez synchrone.

Pour les autres types d'entrées-sorties, typiquement des disques durs parallèles, on peut se permettre d'avoir des accès moins synchrones puisqu'il y a souvent les temps d'accès des disques qui interviennent.

Dans le premier cas, afin d'éviter trop d'interférences avec le reste de la machine, il est bon de prévoir un canal spécifique pour la vidéo. Afin de bénéficier de la répartition de la charge et des données, on distribue la mémoire écran sur les processeurs et dans ce cas on accède aux données vidéo par un canal vidéo par processeur. Du fait de nombre important de processeurs (256 par exemple), le débit sur chaque canal vidéo reste faible et peut tenir sur une liaison série à 25 MHz. Pour lisser le débit et limiter le temps passé par chaque processeur à envoyer les données vidéo, on prévoit de l'ordre de 1 kbit de mémoire tampon avant chaque canal vidéo. Cela permet de factoriser de nombreux accès à la mémoire et d'éviter la conception d'un système de DMA qui irait chercher de lui-même les données vidéo en mémoire. Le remplissage de ce tampon peut se faire sur interruption tous les 4000 cycles vidéo environ (256 PEs et vidéo sur 32 bits), soit 2  $\mu$ s toutes les 20 à 40  $\mu$ s, ce qui est tout-à-fait raisonnable puisque cela n'occupe tout au plus que 10% du temps machine.

Pour ce qui est des entrées-sorties de type disques durs, on peut avoir besoin d'un débit encore plus important, correspondant à plusieurs canaux HIPPI d'un débit de 100 Mo/s chacun pour accéder à plusieurs bancs de disques parallèles du commerce de type RAID par exemple. Ces accès étant asynchrones, on peut se permettre de ne pas concevoir de matériel spécifique pour mais plutôt de se brancher en parallèle sur le réseau de communication qui a largement de quoi soutenir le débit et qui est probablement inutilisé lorsqu'on fait de grosses opérations de transferts sur disques. C'est cette approche qui est prise dans le cas de la MP-1 [Bla90b, Bla90a].

### 5.3.4.3 Les réceptions scalaires

De même qu'on a prévu un mécanisme d'émission scalaire, il faut prévoir le mécanisme de réception scalaire. Mais ce dernier doit être prévu au niveau de l'HyperCom, car il faut extraire de l'information au niveau des PEs pour la faire remonter à contre courant du flot d'instruction qu'on avait pu utiliser pour les émissions scalaires.

Pour ce faire on prévoit quelques pattes de type « drain ouvert » ou « collecteur ouvert » reliées à un bus global, mécanisme ayant l'heureuse propriété de constituer un « *ou* logique câblé » sans court-circuit électrique et donc de permettre de récupérer sur ce bus global la valeur représentant le *ou* de toutes les valeurs présentées par tous les processeurs. 4 pattes permettent de récupérer une valeur sur 32 bits en 8 cycles d'horloge, valeur raisonnable représentant un bon compromis entre nombre de pattes et temps pris pour effectuer une réduction d'une variable parallèle.

On peut donc soit récupérer la valeur contenue sur un processeur si tous les autres ne présentent que des 0, soit récupérer directement la valeur d'un *ou* associatif. Les autres réductions, autre que le *et* déduit du précédent par la loi de MORGAN, sont faites par logiciel en réduisant toutes les valeurs d'abord sur un processeur [HS86] et en récupérant la valeur au niveau du processeur scalaire.

### 5.3.4.4 La gestion des exceptions

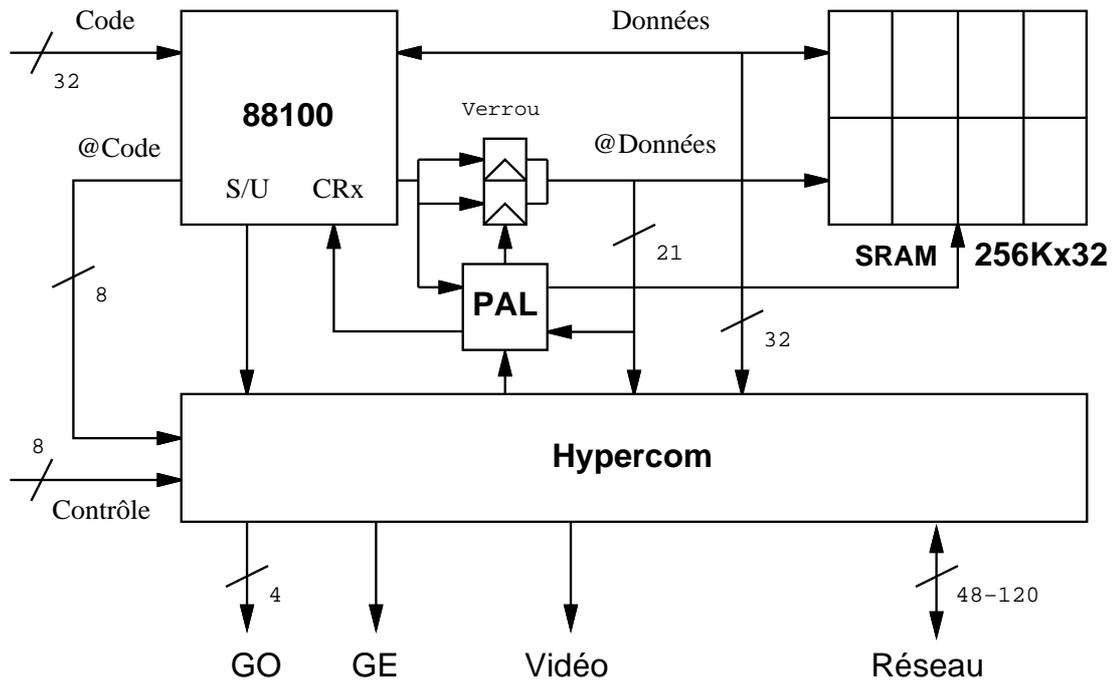
L'absence d'indirection au niveau des registres des PEs pourtant si pratique [Dou89] complique l'écriture des routines d'exception. En effet lorsqu'on a une exception flottante par exemple, on peut corriger le calcul (prenons le cas d'une dénormalisation) et écrire le résultat dans le registre de destination. Mais le numéro de ce registre n'est connu que dans le processeur. Comme le MC88100 n'a aucun moyen de faire une indirection sur ce registre, il faut :

- soit faire remonter l'information au processeur scalaire qui peut générer une instruction avec un nom de registre surchargé<sup>20</sup> ;
- soit plus simplement en sauvant au début de l'exception tous les registres dans une zone de mémoire spécifique que l'on peut modifier par une indirection dans la mémoire locale. La fin de l'exception recharge dans les registres toute la zone mémoire, effectuant ainsi la correction.

Le processeur utilisé contient tout ce qu'il faut pour gérer les exceptions tout en respectant le pipeline. Puisqu'on rajoute un système pipeliné, externe au processeur, d'alimentation en instructions, il faut être capable de sauvegarder l'état du pipeline global de la machine. Cela est fait au niveau du processeur scalaire comme on le verra en 6. Mais pour être capable de faire repartir correctement le pipeline après une exception, il faut connaître la date à laquelle s'est arrêté chaque processeur pour cause d'exception et l'empêcher d'exécuter alors le flot d'instruction du pipeline. Cela est fait au niveau de l'HyperCom qui contient un registre qui mémorise la date lorsqu'une exception survient

---

20. Soit par un mécanisme de multiplexeur identique à celui de l'émission scalaire comme on aura l'occasion de le voir, soit plus simplement par logiciel en ayant un tableau de fonctions pour tous les transferts de registres possibles dont on choisira la bonne fonction à exécuter en fonction du registre de destination. Cette dernière solution est à peine plus longue que la précédente mais ne nécessite aucun matériel supplémentaire.

FIG. 5.1 - *Synoptique d'un nœud de la machine.*

et qui est capable de geler le processeur. Un autre registre de l'**HyperCom** enregistre le poids faible du bus d'adresse du processeur à ce moment-là pour connaître le vecteur d'interruption et donc la raison de l'exception. On aura ainsi tous les éléments pour traiter l'exception correctement.

## 5.4 Conclusion

Nous pouvons donc maintenant présenter sur la figure 5.1 une vue du nœud de notre machine parallèle POMP.

Malgré un certain nombre de contraintes technologiques exposées au départ, on propose une solution pour construire à moindre frais une machine SIMD en évitant d'avoir à concevoir un processeur spécialisé. Même si la machine a un nombre de MFLOPS/dm<sup>3</sup> plus faible qu'en intégrant processeur et mémoire sur un même circuit comme on pensait le faire au début du projet le gain est énorme :

- machine plus réalisable car la conception d'un processeur a souvent provoqué l'échec d'un projet avant même l'étude de la construction d'un prototype pour l'accueillir ;
- l'utilisation de circuits de mémoire du commerce permet d'avoir plus de mémoire disponible et donc une meilleure utilisation des 2 GFLOPS de la machine que si on avait intégré processeur et mémoire sur le même circuit ;
- pour peu que le réseau soit simple à gérer, le circuit de contrôle et de communication du PES peut être simple au point de loger dans un circuit de logique

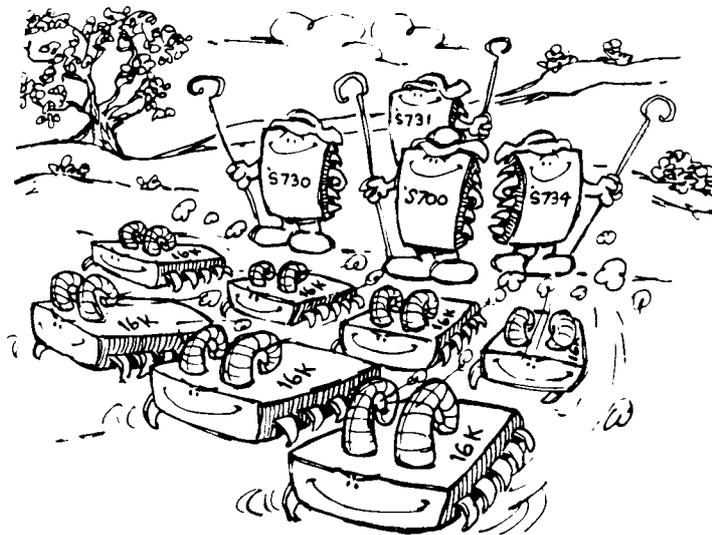
- reprogrammable, ce qui renforce le point précédent ;
- logiciel de base qui vient avec le processeur du commerce qui est choisi ;
- on a une méthodologie pour développer simplement d'autres machines synchrones plus spécifiques comme des machines de traitement du signal ou des machines systoliques.

Maintenant que nous avons défini les bases de notre processeur élémentaire, nous allons nous intéresser aux aspects concernant l'intégration de notre module processeur dans une machine complète et à son utilisation.



# Chapitre 6

## Le contrôle de la machine



**A**FIN de fonctionner correctement, notre machine a besoin d'une part d'un système capable d'exécuter le code scalaire de notre modèle de programmation et d'autre part de contrôler le bon fonctionnement de la machine et d'envoyer les instructions parallèles de bas niveau aux processeurs élémentaires, puisqu'on est dans une machine à architecture SIMD.

## 6.1 Les dispositifs du contrôle

### 6.1.1 Le processeur scalaire

Puisque c'est souvent la partie qui limite les calculs dans une machine, même parallèle [Amd67], on a intérêt à avoir le processeur scalaire le plus rapide possible, dans la mesure du raisonnable.

*A priori* rien ne le distinguant d'une machine séquentielle classique, il vaut mieux récupérer une machine existante plutôt que devoir la développer soi-même, à condition qu'elle soit intégrable dans notre machine. C'est d'ailleurs ce qui est souvent fait, que ce soit en ce qui concerne les ordinateurs vectoriels ou processeurs de tableaux [HSN81] ou les ordinateurs parallèles [Bat82, TR88].

Le problème est ensuite d'avoir une bonne interface entre le processeur scalaire et la partie parallèle de la machine : faible latence d'émission des instructions parallèles, récupération rapide d'une valeur par le processeur scalaire depuis un PE. En effet, lorsqu'on a de petites sections parallèles, ce sont principalement les facteurs limitants. C'est ce qui peut motiver la création d'un processeur scalaire spécialisé.

Néanmoins, ce qui laisse à réfléchir est que la création d'un processeur scalaire implique le développement de l'environnement logiciel nécessaire : compilateurs, assembleur, système d'exploitation, etc. Heureusement, si on part d'un processeur du commerce, une grande partie du travail peut être évitée, si ce n'est totalement.

La solution optimale au niveau programmation est d'utiliser l'hôte pour jouer le rôle de processeur scalaire. Ainsi l'utilisateur n'a pas à faire de modifications importantes de la partie système de ses programmes. Il a l'impression que tous les calculs ont lieu sur sa machine habituelle. En particulier les interfaces graphiques habituelles continuent à fonctionner.

La mise au point en est aussi grandement facilitée : les analyseurs de performance (`gprof`) de l'hôte peuvent être utilisés sur le code commandant la machine parallèle tout comme les débogueurs de l'hôte.

### 6.1.2 Le séquenceur

La nécessité d'avoir un séquenceur provient du besoin d'avoir une adaptation de débit entre le débit d'instructions pouvant être fourni par le processeur scalaire d'une part et celui nécessaire à alimenter les PEs d'autre part.

Si on prend le cas de POMP, on a besoin d'avoir un débit d'instructions parallèles de  $20 \cdot 10^6$  instructions/s (en considérant les MC88100 à 20 MHz utilisés dans notre prototype) alors que dans le meilleur des cas le bus VME limite la bande passante à 10 Mmots de 32 bits par seconde. Si en plus on demande à l'hôte de jouer le rôle de séquenceur, le débit est encore plus catastrophique et les performances sont en conséquence (figure 6.1). Les débits maximaux avec le 68020 sont ceux mesurés sur notre Sun 3/110. Bien évidemment, lorsque le processeur scalaire fait du séquençage, il ne

TAB. 6.1 - *Les instructions de séquençement.*

<i>Méthode sur VME</i>	<i>Débit d'instructions</i>	<i>GIPS sur POMP</i>	<i>Efficacité de la machine</i>
DMA	$10^7$	2,5	50 %
68020 (données dans le cache)	$1,6 \cdot 10^6$	0,41	8 %
68020 (données hors du cache)	$1,23 \cdot 10^6$	0,31	6,1 %

fait pas de calcul scalaire en même temps...

L'idée est d'envoyer des macroinstructions qui sont transformées en microinstructions par le séquenceur<sup>1</sup>. Outre la notion habituelle de microinstructions, le parallélisme introduit la virtualisation : chaque instruction parallèle est à exécuter sur chaque processeur physique autant de fois qu'il y a de processeurs virtuels par processeurs physiques. On peut donc réduire fortement le nombre de macroinstructions à envoyer lorsque le *vp\_ratio* est important. On voit ici que le fait d'avoir la notion de processeurs virtuels ou pas dans le modèle de programmation détermine de manière importante l'architecture de la machine au niveau du contrôle.

Si en plus on a des processeurs élémentaires de petite taille alors qu'on fait des calculs sur des données plus larges que les processeurs, chaque instruction (par exemple sur 32 bits) devra être déroulée en beaucoup d'instructions (par exemple de 1 bit), transférant du travail du processeur scalaire vers le séquenceur.

C'est pourquoi le processeur scalaire d'une CM-2 par exemple, qui est en fait l'hôte de la machine, n'est pas effondré lorsqu'on bénéficie des deux conditions précédentes : *vp\_ratio* important et opérations avec des opérandes « larges » par rapport aux processeurs.

Enfin le coût relatif d'un séquenceur évolue en fonction de la technologie. En particulier si on regarde l'ordinateur ILLIAC IV, on constate qu'il y avait un micro-séquenceur qui générait 200 bits de microcode qui étaient envoyés « à plat » à chaque PE de la machine [BBK<sup>+</sup>68] car le fait d'avoir un décodeur d'instructions factorisé et un bus commun de 200 bits vers tous les PEs coûtait moins que d'avoir un bus commun d'instructions moins large et un décodeur d'instructions dans chaque PE (nécessitant une mémoire supplémentaire donc un coût accru), à l'époque.

Il faut donc trouver un compromis entre le coût et l'efficacité<sup>2</sup>.

### 6.1.3 Le bus d'émission scalaire

Ce bus global permet d'une part l'émission des instructions parallèles et d'autre part l'émission d'une valeur scalaire à tous les processeurs de la machine. Dans certains cas ils peuvent être dédoublés pour des raisons de débit mais en général on peut s'arranger pour mettre les valeurs dans des champs d'instructions parallèles, pour faire des

---

1. À savoir : plutôt que d'envoyer une description fine des instructions au niveau de chaque dispositif interne du processeur, le microcode, on se contente d'envoyer le numéro d'une séquence de microcode, le macrocode. Cela est possible car on s'aperçoit que le nombre de séquences qu'on utilise en pratique est bien inférieur au nombre de microinstructions possibles.

2. Ce doit décidément être aussi un PFI (principe fondamental de l'informatique).

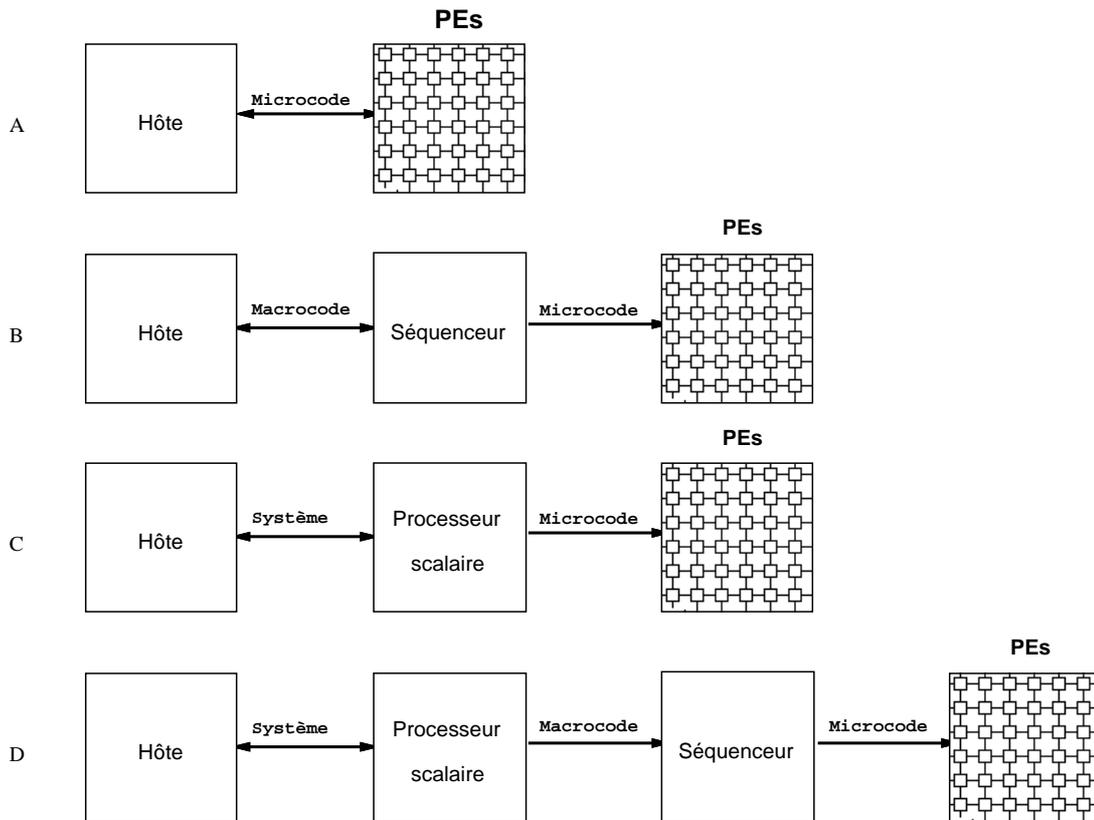


FIG. 6.1 - Différentes organisations possibles du contrôle de la machine.

économies de câblage, de la même manière qu'un processeur peut avoir des champs de constantes numériques dans certaines de ces instructions.

#### 6.1.4 La récupération d'une valeur globale

Comme on a besoin souvent de tester une condition globale, par exemple pour savoir lorsqu'un algorithme itératif a terminé son travail, ou de récupérer une valeur contenue dans un PE, pour la renvoyer à tous les autres par exemple, on a dans la machine un autre bus relié à tous les processeurs capable de récupérer l'information globale ou ne concernant qu'un PE, comme on l'a vu en § 5.3.4.3.

## 6.2 Quelques combinaisons utilisées

Il y a un compromis à trouver en ce qui concerne la localisation de ces tâches de contrôle. Si on considère l'intégration de notre machine dans un ordinateur hôte, on peut faire gérer ces tâches soit par celui-ci, soit par un processeur scalaire indépendant, soit par un séquenceur d'instruction, sachant que toute combinaison des solutions précédentes est possible, comme indiqué sur la figure 6.1.

On peut constater avec les exemples suivants que la tendance est à la diminution du câblage, même au dépend d'une augmentation de la complexité des PEs (utilisations

d'un décodeur d'instructions sur chaque PES).

Mais analysons les stratégies de contrôle scalaire de quelques machines SIMD.

### 6.2.1 L'ILLIAC IV

L'ILLIAC IV entre dans la catégorie B : tout le contrôle de flot est géré par l'hôte (un B6500), le contrôle des PES étant laissé à un séquenceur [BBK<sup>+</sup>68].

Le séquenceur est capable d'exécuter des boucles, faire du calcul d'adresse et gérer les exceptions qui ont lieu sur les PES. En fait il s'agit là d'un véritable processeur qui travaille sur des nombres entiers et qui a entre autre la caractéristique intéressante d'avoir une instruction EXEC permettant une génération automatique d'instructions. Mais quel dommage que cette instruction ne soit pas présente sur les PES ! Elle aurait permis de faire simplement du SPMD.

La machine a 2 bus séparés pour envoyer d'une part les instructions aux PES (200 bits) et d'autre part les opérandes globaux (sur 64 bits). Cela semble assez logique dans la mesure où on envoie à tous les PES le microcode à plat, on ne va pas remultiplexer les opérandes.

### 6.2.2 Opsila

La machine OPSILA entre dans le modèle D : un hôte s'occupe de la partie système et le code scalaire tourne sur un processeur scalaire fait à partir de « tranches » du commerce. Ce dernier fournit des opérations vectorielles générales au processeur d'instruction [Gal85].

Le processeur d'instructions d'OPSILA, qui correspond au séquenceur de la figure 6.1, gère d'une part la virtualisation et d'autre part les conflits d'accès à la mémoire. En effet, chaque PE étant relié à un banc mémoire au travers d'un réseau bloquant contrôlé de manière globale par le processeur d'instruction, il faut parfois sérialiser les accès en cas de conflit.

La virtualisation, qui revient ici à faire du *strip-mining* pour 16 PES, peut porter sur des boucles vectorielles comportant plusieurs instructions (les IVD), ce qui permet l'utilisation des registres des PES, comme dans les machines vectorielles avec registres vectoriels et exploiter pleinement la puissance de chaque PE [Aug85].

### 6.2.3 MasPar MP-1

Le cas de la MP-1 n'est pas clair si on a à sa disposition que [Bla90b, Bla90a]<sup>3</sup>.

Il semble qu'on soit dans le cas C puisque le code scalaire ne tourne pas sur l'hôte mais sur LACU. Même si le processeur possède un jeu d'instruction à la RISC, celui-ci est microcodé pour générer et dérouler en même temps le code pour les processeurs élémentaires.

On peut soit imaginer un couplage VLIW du processeur scalaire et des PES, soit une micromachine indépendante pour la génération de code des PES, soit encore une solution sans recouvrement entre l'exécution scalaire et l'exécution parallèle par simplification

---

3. C'est souvent le problème avec les machines commerciales récentes : il est impossible d'obtenir des schémas ou des sources précis pour des raisons de propriété industrielle.

en réutilisant la micromachine commune processeur scalaire-séquenceur. Il semble que ce soit la dernière solution qui soit celle utilisée.

#### 6.2.4 CM-2

Comme l'application visée est d'avoir beaucoup de processeurs virtuels qui travaillent souvent sur des données de largeur supérieure à celle des processeurs (1 bit), le débit entre le processeur scalaire et le séquenceur n'est plus limitant [TR88].

On peut donc utiliser l'ordinateur hôte comme processeur scalaire, typiquement un SUN-4 qui envoie les instructions parallèles au séquenceur de la CM-2 à travers une carte sur le bus VME du SUN. En fait le processeur scalaire a généralement suffisamment de temps pour gérer simultanément plusieurs partitions de CM-2 à travers plusieurs cartes de séquenceur. On a donc une configuration de type B (figure 6.1).

#### 6.2.5 MPP

L'architecture suit le modèle D de la figure 6.1.

Le processeur scalaire est un ordinateur du commerce (un PDP-11/34) ainsi que l'hôte (un VAX 11/780). Comme ils sont du même constructeur, ils sont facilement intégrables (bus et code identiques) [Bat80a, Bat80b, Bat82].

Le contrôleur-séquenceur de la machine parallèle est microprogrammable et gère aussi les calculs scalaire, le processeur scalaire (le PDP-11) s'occupant plutôt du contrôle de flot global et la gestion de la mémoire.

#### 6.2.6 WaveTracer DTC

Il semble qu'on soit dans la configuration B, d'après les essais que nous avons effectués (la documentation que nous avons eu à notre disposition à ce sujet n'étant pas très explicite [Jac90]).

Le séquenceur est à base d'un processeur 29K d'AMD et est relié à l'hôte, qui sert de processeur scalaire, par une interface SCSI, ce qui rend la connexion à n'importe quel hôte très simple.

Comme les processeurs sont de petite taille et que le modèle possède la notion de processeurs virtuels, la relative faible bande passante du bus SCSI ne semble pas être limitante, même si tout le code scalaire s'exécute sur l'hôte.

L'avantage de ce dernier aspect est, comme on l'a déjà indiqué, que l'intégration de la machine au niveau du système d'exploitation de l'hôte est excellente.

### 6.3 Le contrôle de la machine POMP

Puisque la génération des instructions à partir de l'hôte nous semblait pénalisante, On a étudié 3 autres approches :

- 2 solutions à base de séquenceurs du commerce ou bien « maison », avec l'hôte qui joue le rôle de processeur scalaire ;
- une autre où on a un processeur du commerce qui joue à la fois le rôle du processeur scalaire et du séquenceur.

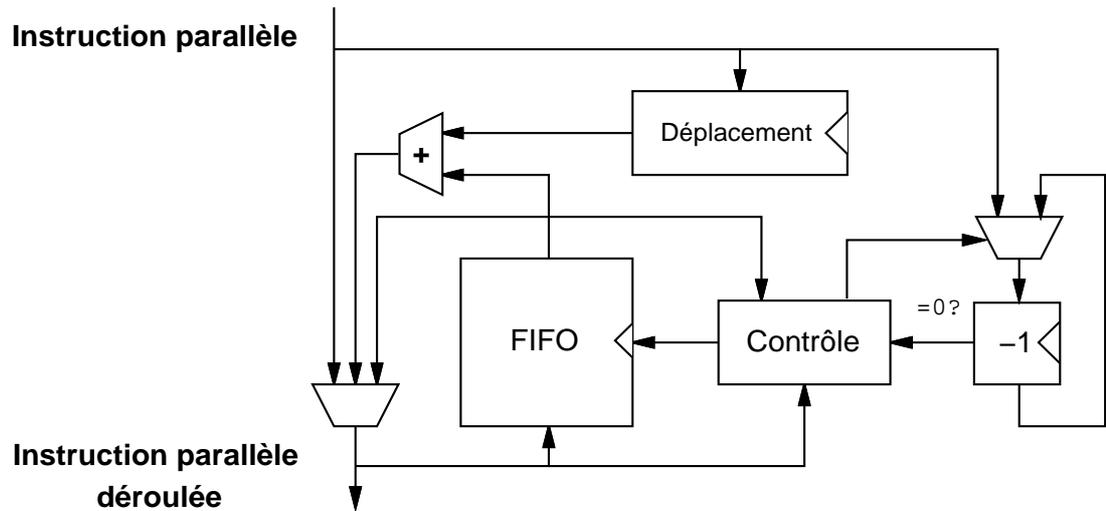


FIG. 6.2 - Un séquenceur de processeurs virtuels maison.

### 6.3.1 L'hôte et un séquenceur

On considère donc une architecture de type B, avec un séquenceur contrôlé par l'hôte.

#### 6.3.1.1 Un séquenceur câblé

Il suffit d'avoir un système qui virtualise sur une seule instruction à chaque fois, comme sur la CM-2 avec PARIS, voire sur plusieurs instructions si on veut augmenter le rendement du processeur scalaire.

On peut réaliser ce système grâce à une simple file rebouclée sur elle-même, un compteur de boucle, un additionneur pour gérer la progression des adresses des processeurs virtuels suivant un déplacement, comme indiqué sur le diagramme de la figure 6.2.

Chaque instruction contient un bit indiquant si son opérande doit être rebouclé à travers l'additionneur pour calculer l'adresse du tour de boucle suivant. Avec des FIFOs du commerce, on peut stocker de 2 à 4K instructions dans 4 boîtiers, ce qui est en général suffisant pour stocker une séquence de programme entre 2 points de synchronisation.

Comme il n'y a pas de mémoire de cartage<sup>4</sup> ou de microinstruction, c'est l'hôte qui envoie les microinstructions à répéter. Comme cela on gagne en souplesse puisque le microcode n'est pas figé. On peut le changer en fonctions des applications. La perte en concision n'est pas très importante dans la mesure où lorsqu'on utilise un processeur RISC du commerce comme PE, les instructions sont déjà très compactées : elles font toutes 32 bits.

Comme la file n'est pas dans le chemin processeur scalaire-PEs, on peut envoyer directement les instructions fournies par le processeur scalaire aux PEs en même temps qu'on remplit la file. Cela permet de gagner du temps précieux dans le cas où on a que peu de processeurs virtuels.

<sup>4</sup> *mapping* en anglais. Le dictionnaire français d'informatique propose le terme officiel « mappage » (sic), mais je préfère « cartage »...

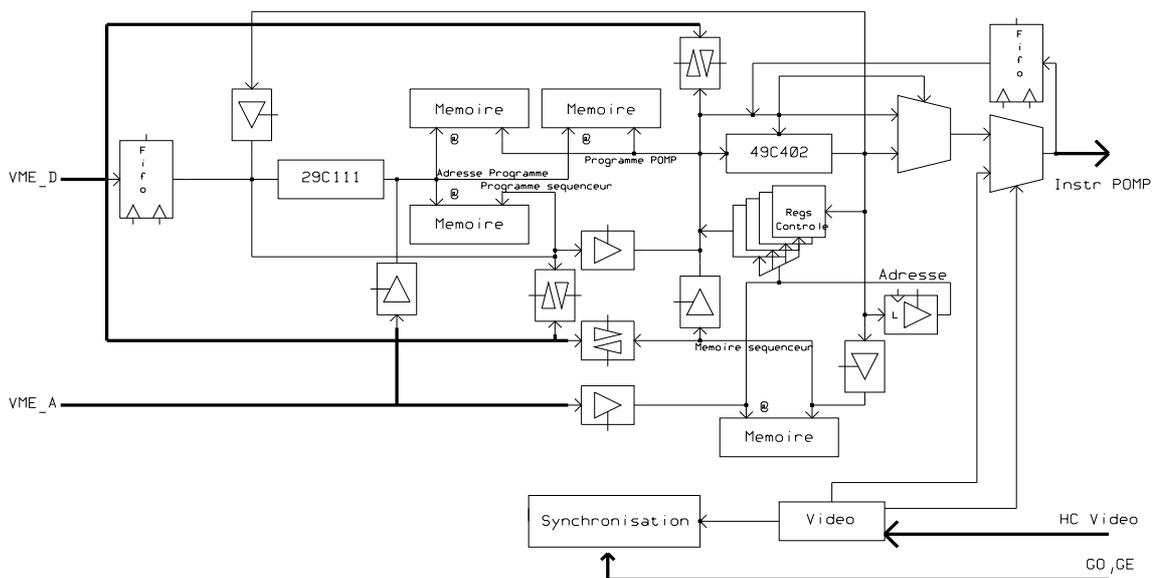


FIG. 6.3 - Un séquenceur en tranche pour POMP.

### 6.3.1.2 Un séquenceur en tranche

Mais dès qu'on se heurte à des problèmes plus compliqués, lorsqu'on ne veut pas avoir les processeurs virtuels traités de manière linéaire dans la mémoire ou qu'on veut que le séquenceur sache gérer les entrées-sorties ou la vidéo, il est clair que le séquenceur extrêmement simple précédent ne peut plus suffire.

En particulier, la gestion d'une exception qui arrive sur un des PEs est très délicate, ce qui invalide la solution précédente.

Heureusement, il existe des composants prévus pour faire ce travail : les séquenceurs et les UAL en tranches, sortes de blocs de base qui permettent de faire des processeurs taillés sur mesure, évidemment aux dépens de l'intégration dans la plupart des cas [Adv88c, Adv88a].

Le schéma résultant est indiqué sur la figure 6.3. Avec de telles tranches, on peut espérer faire un séquenceur possédant un temps de cycle de l'ordre de 60 à 80 ns<sup>5</sup>, selon l'électronique que l'on met autour. Le cycle est donc plus long que celui prévu pour le processeur de la machine (40 à 50 ns).

D'un autre côté, si on regarde un programme d'ordinateur typique, il contient seulement de l'ordre de 1/10 à 1/5 de branchements, donc on peut décider assez arbitrairement de n'autoriser qu'au plus un branchement toutes les 2 instructions<sup>6</sup>. Ainsi, il suffit de fournir 2 instructions pour les PEs par cycle de séquenceur [Ker89]. Cela est réalisé simplement en doublant la mémoire de programme des PEs : une pour les instructions paires et l'autre pour les instructions impaires. Le cycle de base du séquenceur est donc de 80 à 100 ns, selon la fréquence des PEs.

Le séquençage est réalisé en tant que tel par le 29C111, un séquenceur 16 bits, accompagné d'une UAL 16 bits 49C402 pour faire des calculs d'adresse liés à la virtualisation. Les champs d'opérande des instructions des PEs peuvent être remplacés par

5. Moins si on utilise les versions en AsGa, mais est-ce bien nécessaire?

6. Ce qui n'est pas trop pénalisant lorsqu'on a autre chose que des branchements dans le programme...

des valeurs calculées par l'UAL grâce au multiplexeur qui la suit pour permettre soit une émission scalaire, soit une gestion d'adresse virtualisée. Le fait d'avoir un multiplexeur est justifié car cela demanderait plusieurs instructions à l'UAL pour faire ce travail pourtant simple par l'intermédiaire de plusieurs instructions de masquage sur 32 bits. Les boucles virtuelles en tant que telles sont gérées par le compteur de boucle intégré au 29C111.

Toute la gestion de la machine se fait au travers d'un banc de registres, accessible aussi depuis le bus VME pour initialiser le système. Ces registres permettent le contrôle des interruptions vers l'hôte et vers les PES, le contrôle de la vidéo, la gestion du pipeline de la machine, la liaison avec le *ou global* de la machine, la récupération d'une valeur sur un PE, etc.

La vidéo consiste simplement à envoyer une interruption périodiquement et à envoyer les instructions de lecture de la mémoire vidéo. L'adresse étant fournie par le circuit vidéo aux PES à travers le dernier multiplexeur.

Le problème le plus délicat est bien sûr l'arrivée d'une exception imprévue sur un ou plusieurs PES (dénormalisation d'un nombre flottant, par exemple). En effet, il faut alors arrêter toute la machine, traiter la(les) exception(s) là où il y a lieu et faire repartir toute la machine, peut-être à des endroits différents selon les PES, en jouant sur les masques d'activité. Pour conserver une trace du pipeline, les dernières instructions exécutées sont conservées dans une FIFO qui se fige lorsque survient une exception. Ces instructions sont alors disponibles pour une étude par le séquenceur ou le processeur scalaire afin de traiter correctement les exceptions.

Mais le gros problème de ce genre de processeur est qu'il y a énormément de logiciel à développer. Puisqu'on peut faire ce que l'on veut, le jeu d'instructions est aussi arbitraire, l'assembleur aussi, quand au compilateur, en général on ne l'écrit jamais, vue l'ampleur de la tâche. C'est pourquoi on en reste souvent à la programmation au niveau microcode, rebutante mais hélas assez répandue<sup>7</sup>.

### 6.3.2 Un processeur standard

Afin d'éviter tout ce travail fastidieux et répondre à notre objectif initial de simplicité matérielle et logicielle, nous avons étudié l'emploi d'un processeur standard pour faire du séquençage d'instructions.

Un tel processeur n'est pas, *a priori*, qualifié pour ce genre de travail :

- intégration trop importante qui fait qu'on n'a pas accès aux registres ni aux bus internes, accès pourtant utiles lorsqu'on fait un séquenceur ;
- le processeur est pipeliné, contrairement à la plupart des séquenceur en tranche, ce qui fait qu'il répond moins vite aux stimuli extérieurs (interruptions, etc.).

Néanmoins, les processeurs RISC modernes sont plus intégrés et offrent un temps de cycle voisin, voire inférieur, qui peut compenser leur inadéquation.

Il y a des intérêt évidents qui peuvent nous infléchir en sa faveur :

- plus de langage d'assemblage à définir puis à utiliser : il suffit de reprendre celui

---

7. Malheureusement, la mode du microcode revient, tout particulièrement avec la programmation de processeurs comme les i860...

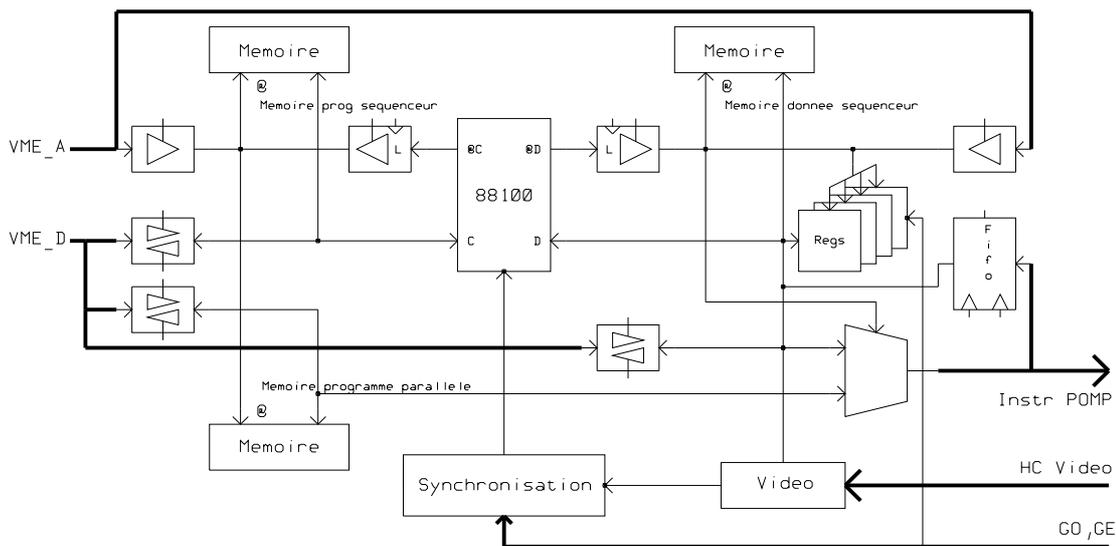


FIG. 6.4 - Un séquenceur à base de MC88100 pour POMP.

du processeur bien sûr, accompagné de tout son environnement de développement matériel et logiciel ;

- le développement matériel du séquenceur ne contient plus le développement d'un processeur qui, même à base de tranches, est assez compliqué.

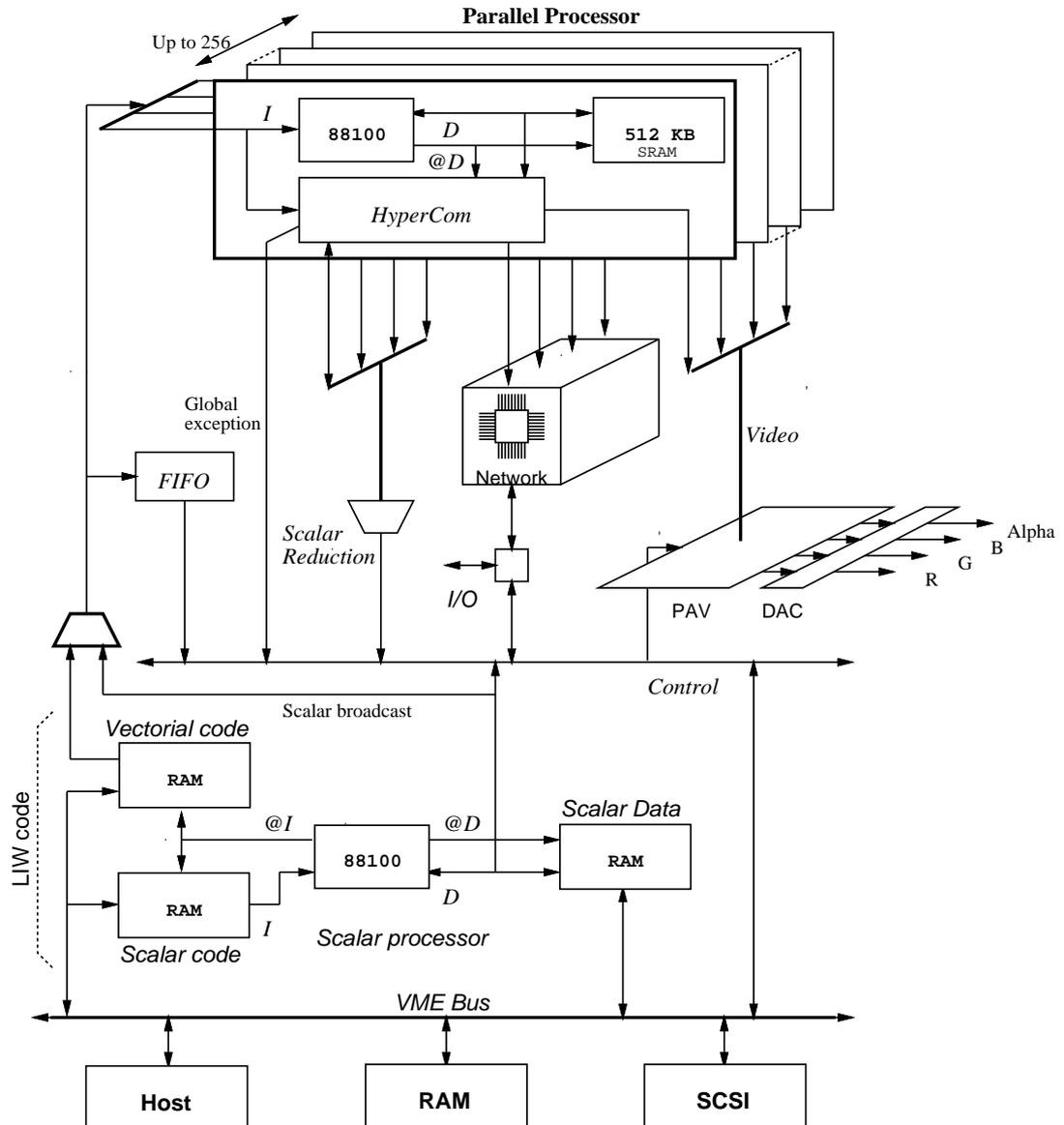
Reste donc à pervertir un processeur du commerce en séquenceur. Après tout, si on est capable de transformer un processeur standard en processeur SIMD, il n'est pas impossible continuer les adaptations. La solution proposée est indiquée sur la figure 6.4 et ressemble finalement assez au schéma précédent. Maintenant on peut donc dessiner le synoptique complet de POMP qui est indiqué sur la figure 6.5.

De même que le séquenceur lisait dans sa mémoire de programme le code pour les PES en même temps qu'il lisait ses propres instructions, le processeur lit les instructions des PES en même temps que les siennes dans deux mémoires séparées mais adressées de manière identique. On est donc en présence d'un couplage VLIW<sup>8</sup> entre le séquenceur et les PES puisque l'instruction lue est une instruction longue commandant les diverses parties de la machine de manière synchrone : une partie scalaire, une partie parallèle et une partie contrôle de l'HyperCom, comme on aura l'occasion de le voir.

Notons que cette méthode entraîne un recouvrement implicite de l'exécution du code scalaire et parallèle, ce qui est un avantage comparé à des machines telles que la MP-1 par exemple puisque cela permet d'exploiter un autre niveau de parallélisme.

Le banc de registres est identique, ainsi que le multiplexeur nécessaire au remplacement de certains champs d'opérande d'instructions. La FIFO pour gérer les interruptions est inchangée. Le multiplexeur lié à la vidéo a disparu dans une phase d'épuration ultérieure à [Ker89] : ce sera le MC88100 de contrôle qui calculera lui-même l'adresse de départ des tampons vidéo à chaque interruption vidéo donnée par le générateur de signaux de synchronisation vidéo.

8. *Very Long Instruction Word*. Effectivement ici le « *very* » est de trop mais il s'agit néanmoins du terme consacré.

FIG. 6.5 - *Synoptique de la machine POMP.*

On se retrouve donc avec une architecture de type C au niveau du contrôle : l'hôte ne sert plus dans ce cas qu'à faire l'interface avec le monde UNIX, exécutant les appels systèmes demandés par le processeur scalaire qui fait aussi office de séquenceur pour les PES.

Le fait d'avoir un code VLIW implique dans notre cas que l'efficacité du mécanisme sera totale s'il y a un bon équilibre entre le nombre d'instructions scalaires et parallèles et que les dépendances entre les 2 sont telles qu'on peut effectivement mettre côte à côte les 2 flots d'instructions. Dans la réalité on s'écarte de cette image idéale :

- soit il y a trop d'instructions scalaires et dans ce cas on est probablement dans le cas de la loi d'AMDAHL [Amd67] car sinon on peut essayer de paralléliser un

peu plus le code scalaire si cela est compatible avec le parallélisme SIMD ;

- soit il y a trop d'instructions parallèles et c'est tant mieux puisque cela signifie qu'on est sur une section très parallèle et donc qu'on utilise POMP au maximum de ses possibilités.

On pourrait peut-être trouver ce couplage trop rigide dans la mesure où le processeur scalaire ne peut pas faire quelque chose de totalement indépendante de la génération d'instructions parallèles telle qu'elle a été définie à la compilation.

Supposons que l'on veuille mettre un système de file, voire même un séquenceur, pour lisser les flots d'instructions scalaires et d'instructions parallèles. On s'aperçoit que soit on pouvait équilibrer la charge à la compilation en appariant mieux les instructions, soit il y a une interdépendance entre les 2 flots et dans ce cas cela se traduira par un système de synchronisation qu'il faudra implanter. Or, à cause du pipeline du processeur scalaire, une telle synchronisation consistant à lire une donnée, agir en conséquence en écrivant une autre donnée prend environ 10 cycles, ce qui est énorme. En effet, le code scalaire comprenant le contrôle de flot scalaire composé, comme nous l'avons vu, généralement de plus d'une instruction de contrôle de flot toutes les 10 instructions impliquera probablement d'avoir autant de points de synchronisation et diminuera de manière très importante l'efficacité de la machine.

Le seul cas évident où cela est nécessaire est pour les opérations d'entrées-sorties asynchrones au niveau d'UNIX. Mais comme c'est l'hôte qui exécute la partie système d'exploitation, le découplage processeur scalaire-processeurs parallèles n'est pas nécessaire.

L'autre intérêt d'avoir un séquenceur indépendant est de pouvoir économiser de la mémoire dans le cas où les codes sont très déséquilibrés, en particulier la partie système d'interface avec l'hôte risque de nécessiter que peu d'instructions parallèles. Si on veut économiser cette mémoire, le plus simple est de déclarer 2 espaces d'adressage d'instructions, un purement scalaire — on envoie des instructions nulles aux PEs — et un autre où on accède bien aux instructions VLIW. Le compilateur peut alors indiquer où charger en mémoire les procédures, selon qu'elles sont purement scalaires ou pas. Néanmoins, cela nécessite de différencier la mémoire et donc de compliquer le séquenceur alors que la mémoire programme est de toute manière peu chère car à un seul exemplaire dans la machine, seulement sur le séquenceur.

Enfin, on avait pensé au début à une solution encore plus économe : un des PEs, en plus de son travail, aurait fait le rôle du séquenceur et du processeur scalaire. Cela revenait à prendre l'architecture que l'on a choisi figure 6.5 en fait et de dire que le processeur scalaire était aussi un PE. Outre le fait le synchronisme des PEs, lorsque le code scalaire s'exécute sur un des PEs, implique que les autres ne font rien pendant ce temps, le problème est lorsqu'une exception survient sur le PE contrôleur, elle perturbe forcément tous les autres PE, ce qui est difficile à gérer.

## 6.4 Conclusion

Nous avons exposé la conception d'un hybride processeur scalaire-séquenceur pragmatique adoptant délibérément la philosophie RISC et, en fait, spécifiquement au VLIW

à contrôle statique :

- on se base sur de la technologie très simple ;
- la synchronisation processeur scalaire-processeurs parallèles est statique ;
- on préfère gâcher un peu de mémoire si cela permet de simplifier beaucoup le matériel ;
- le compilateur doit faire un peu plus de travail pour exploiter au mieux la machine plus simple et donc compenser un peu cette simplicité.

L'aspect lié à l'utilisation de ce séquenceur sera étudié plus en profondeur dans le chapitre 8 au niveau du compilateur et du mécanisme de synchronisation logicielle à réaliser.

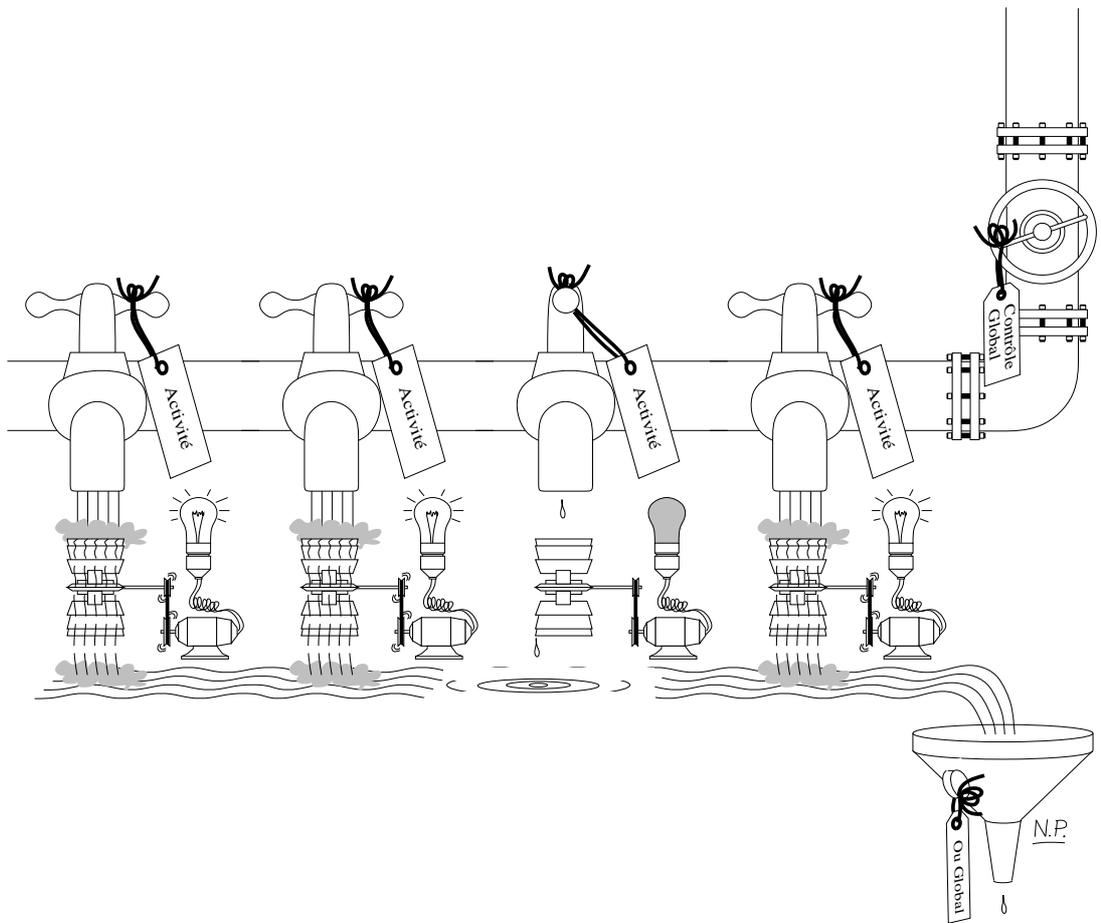
Maintenant que nous avons présenté le système de contrôle scalaire de la machine, on peut regarder les problèmes de contrôle de flot parallèle dans notre machine.





# Chapitre 7

## Le contrôle de flot SIMD



 A nécessité d'avoir un contrôle de flot parallèle efficace sur les machines SIMD n'est plus à démontrer dans le cas d'applications numériques générales. En effet, même si ces machines exécutent la même instruction sur tout un ensemble de données différentes, il est souvent nécessaire de faire des opérations différentes sur des sous-ensembles de données du problème, par exemple pour gérer des conditions aux limites dans le cas de modèles à éléments finis ou pour calculer des fonctions de base telles que la valeur absolue ou le calcul d'un minimum, éviter des erreurs de calcul, sélectionner des données, etc.

Il faut donc rajouter de la souplesse dans l'exécution des instructions que l'on peut voir comme une dessynchronisation bornée dans notre beau modèle synchrone SIMD.

Jusqu'à présent, le seul mécanisme rencontré est celui du *bit d'activité* que l'on trouve sur toutes les machines existantes, y compris sous forme de vecteur de masquage booléen sur la plupart des ordinateurs vectoriels, et le mécanisme du *scatter-gather* qui consiste à compacter, sous la responsabilité du programmeur les sous-ensembles de données pour les traiter séparément, condition par condition :

- la première méthode est assez efficace lorsque le nombre d'imbrications est faible et que les sous-ensembles de données sont équilibrés, ce qui explique qu'il soit limité à 0 (!) dans le cas du langage FORTRAN 90 [MR90], probablement pour des raisons historiquement liées au matériel, hélas ;
- la deuxième méthode est valable lorsque le nombre de données concernées par la condition est faible par rapport à l'ensemble des données. Les sous-ensembles de données sont regroupés dans autant de vecteurs qu'il y a de conditions particulières grâce à une instruction de type *gather* (plus spécifiquement *pack*). Les traitements sont effectués sur ces vecteurs et leurs éléments retrouvent leur position dans le vecteur originel avec une instruction de type *scatter* (plus spécifiquement *unpack*).

Étant plutôt à la responsabilité du programmeur et donc nécessitant une modification plus que superficielle de l'algorithme, ce mécanisme n'est pas considéré dans la suite, bien que tout à fait utilisable par l'intermédiaire d'une communication vers une collection de taille plus petite<sup>1</sup>.

Le développement de machines SIMD à gros grain, c'est-à-dire basées sur des processeurs à 32 ou 64 bits, associé à celui de nouveaux langages à parallélisme explicite de données nous amènent à reconsidérer le problème pour des raisons d'efficacité et à développer en même temps une méthode d'entrelacement du code conditionné pouvant exploiter au maximum une machine SIMD à base de processeurs RISC pipelinés telle que POMP.

Même si on considère qu'un adressage local peut permettre déjà quelques libertés au niveau du contrôle local de l'exécution — ce qui a par exemple justifié le développement de BLITZEN [RB89] par rapport à MPP [Bat80a, Bat80b], le programmeur a besoin d'un domaine de contrôle plus vaste se rapprochant un peu des machines MIMD, ou plus exactement SPMD.

---

1. C'est cette approche qui est prise dans [BS90] où tout conditionnement est compilé en *pack* et *unpack*. Le problème est que si cela équilibre la charge dans le cas de tests imbriqués très creux, cela a pour effet de pénaliser très fortement le cas où il y a peu d'imbrications car les communications nécessaires aux *pack* et *unpack* sont beaucoup plus lentes que la méthode du bit d'activité.

---

```

collection tableau;
    ... /* La taille est d'efinie ailleurs. */
double tableau a,b; /* a et b sont */
    ... /* des variables parallèles. */
void div_par_0()
{
    where (a != 0)
        b = 1/a;
    elsewhere
        b = 0;    /* Pourquoi pas! */
}

```

---

FIG. 7.1 - *Petit exemple en POMPC utilisant le where.*

Dans la suite, on prendra comme argumentation l'exemple de la figure 7.1 où un utilisateur veut programmer la division de 2 vecteurs élément par élément tout en évitant les divisions par 0.

## 7.1 La gestion du contrôle de flot SIMD

### 7.1.1 Décision locale d'exécution d'instructions

Les décisions locales sont à la base du contrôle de flot parallèle des machines SIMD depuis leur existence [SBM62, BBJ<sup>+</sup>62] et il ne semble pas y avoir eu beaucoup d'innovation dans le domaine. Certaines variations ont été proposées, comme le « contrôle global de l'activité<sup>2</sup> » [NFA<sup>+</sup>90], mais le principe reste le même.

### 7.1.2 Vision conceptuelle d'un branchement SIMD

D'un point de vue conceptuel, on peut voir le débranchement d'un processeur d'une machine SIMD comme un saut à une adresse qui sera atteinte dans le futur par le séquenceur global de la machine. Puisqu'il n'y a qu'un seul séquenceur dans la machine, le flot d'instruction est unique et le processeur élémentaire n'a d'autre chose à faire qu'attendre que le flot d'instruction ait atteint l'instruction qu'il demande [HLJ<sup>+</sup>91, page 75].

On peut donc considérer le branchement SIMD comme un saut dans le futur alors qu'un processeur séquentiel (pouvant faire partie d'une machine MIMD) peut brancher son exécution vers n'importe quelle partie (valide) du programme de manière quasi instantanée. Il y a là un exemple de dualité espace-temps :

- un branchement SIMD se fait dans le temps car il faut attendre que l'espace se déplace jusqu'aux PES ;

---

2. Cela ne semble pas très intéressant puisqu'on peut très bien le simuler efficacement à partir du contrôle local contrairement à ce qui y est dit... Mais cette approche peut s'expliquer par la « culture traitement d'image » de [SSDK84].

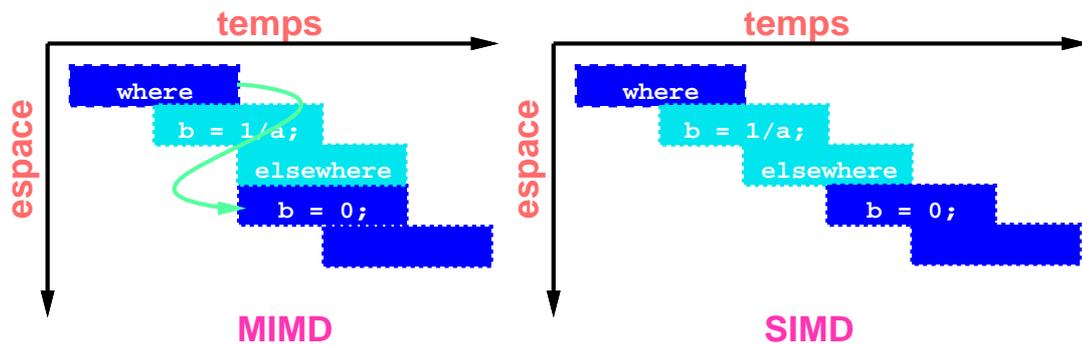


FIG. 7.2 - *Dualité espace d'adressage-temps entre les branchements de machines MIMD et SIMD.*

- un branchement sur une machine séquentielle se fait dans l'espace à une autre adresse du programme en un temps court.

Si on reprend l'exemple précédent où on s'intéresse à un élément où  $a = 0$ , sur une machine MIMD le processeur ira directement à l'instruction correspondant au `elsewhere` (déplacement dans l'espace d'adressage) alors que sur une machine SIMD le processeur attendra que les instructions du bloc `elsewhere` arrivent jusqu'à lui (déplacement dans le temps) comme indiqué sur la figure 7.2. On peut constater qu'à cause du temps de latence du pipeline lors du branchement, le gain du MIMD sur le SIMD n'est pas évident pour de petits blocs conditionnés.

Il semble clair alors qu'un programme possédant beaucoup de débranchements SIMD fera attendre beaucoup de PES et les performances de la machine s'en ressentiront [Fly72] et il va falloir optimiser ce type de contrôle de flot.

Il ne faut pas pour autant négliger l'importance du processeur scalaire qui permet de transporter tous les PES vers un autre point du programme en un temps court, lorsque le débranchement est scalaire.

Afin de mettre en place ce concept, chaque processeur doit posséder une horloge pour savoir quand il faut recommencer à exécuter les instructions qu'on lui envoie ou bien le séquenceur doit envoyer à tous le numéro d'instruction<sup>3</sup> qu'il est en train d'envoyer.

Dans le premier cas, il est impossible de savoir à la compilation ou à l'exécution quelle sera la date à laquelle chaque processeur doit reprendre l'exécution, par exemple à cause de boucles dont l'exécution dépend de paramètres extérieurs au programme<sup>4</sup>.

Dans le deuxième cas, la récursion est difficile à gérer : l'adresse d'une instruction n'est pas suffisante pour pouvoir fournir une information sur le niveau de récursion. En effet un processeur peut exécuter plusieurs fois la même instruction (donc à la même adresse) mais à des niveaux d'imbrication différents, information nécessaire car chaque processeur ne voudra pas forcément exécuter le même nombre d'itérations.

On semble donc obligé de rajouter la notion de pile dans laquelle on sauve l'adresse

3. On suppose que le compilateur est capable de donner un numéro différent pour chaque instruction. Un exemple simple est de considérer l'adresse d'une instruction.

4. En fait cela reviendrait à exécuter le programme ! De toute manière le théorème de GÖDEL s'applique...

de reprise d'exécution à chaque appel de fonction. On ne reprend l'exécution que lorsque la pile ne contient que l'adresse de reprise et que celle-ci est atteinte. Ce problème de récursivité semble avoir été oublié dans [HLJ<sup>+</sup>91]<sup>5</sup>.

Une autre approche implique un système de synchronisations multiple [Jes84] mais celui-ci s'avère inefficace lorsque le cycle des processeurs est bien plus court que le temps de synchronisation.

### 7.1.3 Le compteur d'activité

Cette section présente la méthode développée par Nicolas PARIS pour compiler plus efficacement le contrôle de flot SIMD du langage POMPC, que ce soit pour machines SIMD ou MIMD d'ailleurs.

#### 7.1.3.1 Principe

Dans notre cas, on peut profiter des avantages de la programmation structurée que l'on retrouve dans POMPC et qui sont ici particulièrement intéressants lorsqu'on fait du contrôle de flot SIMD. En particulier, la programmation est faite par blocs syntaxiques :

- pas de « goto » parallèles, ce qui est raisonnable dans le cas d'une machine parallèle SIMD. Les débranchements se font au niveau des blocs ;
- un bloc d'instructions a un début et une fin. Il peut être conditionné et donc soit être exécuté soit inactivé en attente d'un autre bloc actif, selon la condition locale ;
- tout bloc est inclus strictement dans un autre bloc ou est le bloc principal du programme ;
- de par la construction par blocs, le conditionnement s'hérite et donc tout bloc contenu dans un bloc inactif est inactif, en considérant une collection donnée<sup>6</sup>.

On voit qu'il suffit de conserver la notion de bloc avec l'idée de récursion et de supprimer la mention de l'adresse des instructions : il suffit qu'un processeur sache dans combien de blocs imbriqués inactifs il est pour savoir gérer son inactivité et son attente. Cette information est stockée dans le *compteur d'inactivité*, improprement appelé par la suite *compteur d'activité* par soucis de concision.

#### 7.1.3.2 Réalisation des structures de contrôle parallèle de POMPC

Maintenant qu'on a décrit le principe de réalisation des conditions, on peut voir dans la pratique comment sont réalisées les structures de contrôle de POMPC exposée dans le chapitre 4 grâce au mécanisme du compteur d'activité. Mais les méthodes peuvent être déclinées avec d'autres mécanismes, comme pour porter POMPC sur des machines

---

5. À leur défense on argumentera que ces derniers étaient plus impliqués dans la compilation du langage DPC pour machine MIMD, donc leurs considérations étaient différentes.

6. Sauf dans un cas très particulier lié à l'instruction `everywhere` qui permet de rendre actif tous les éléments d'un bloc. Mais cette instruction est réservée à des usages très spécifiques tels que les fonctions systèmes ou les optimisations fines.

TAB. 7.1 - Gestion des *where/elsewhere* avec un compteur.

Ouverture du	<i>Si</i> $CNT \neq 0$ ( <i>inactif</i> ),	$CNT \leftarrow CNT + 1$
<code>where</code> ( <i>cond_évaluée</i> )	<i>Si</i> $CNT = 0$ ( <i>actif</i> ),	$CNT \leftarrow \neg cond\_évaluée$
<code>elsewhere</code>	<i>Si</i> $CNT \leq 1$ ( <i>activable</i> ),	$CNT \leftarrow \neg CNT$
Fermeture du bloc	<i>Si</i> $CNT \neq 0$ ,	$CNT \leftarrow CNT - 1$

avec pile d'activité: « incrémenter le compteur d'activité » devient plus généralement « ouvrir un bloc inactif ».

Lorsque le compteur est à 0, le processeur est actif et exécute les instructions du bloc. Toute autre valeur indique qu'on a plusieurs blocs imbriqués inactifs<sup>7</sup> et donc le nombre de blocs dont doit sortir un processeur pour redevenir actif.

Dans ce qui suit, le compteur d'activité associé à un processeur virtuel est représenté par « CNT » afin de clarifier les explications.

### Compilation d'un *where/elsewhere*

Le cas du couple *where/elsewhere* est finalement assez simple (tableau 7.1) puisqu'un processeur qui était actif et qui entre dans un bloc conditionné ne peut être que dans deux états correspondant à la 2<sup>ème</sup> ligne du tableau: exécution du bloc ou non exécution selon la condition<sup>8</sup>. Cet état est codé par 0 (actif) ou 1 (inactif) dans le compteur: dans le cas d'une condition fautive, le compteur reflète bien le fait que le bloc courant devient inactif. Lorsque le processeur arrive ensuite dans le bloc *elsewhere*, l'activité est tout naturellement inversée comme à la 3<sup>ème</sup> ligne du tableau.

Dans le cas où, avant de rencontrer les blocs *where/elsewhere*, le processeur était déjà inactif, comme ceux-ci sont au même niveau d'imbrication, on se contente d'incrémenter la valeur du compteur (1<sup>ère</sup> ligne), jusqu'à la fermeture du bloc correspondant.

Lorsqu'on sort d'un bloc qui était inactif, on tient à jour le nombre d'imbrications de blocs inactifs comme indiqué à la 4<sup>ème</sup> ligne.

On constate ici que la valeur 1 du compteur a une signification particulière permettant de gérer le passage d'un *where* actif à un *elsewhere* inactif ou inversement. Dans la suite on va être amené à particulariser d'autres valeurs pour compiler du contrôle de flot plus compliqué.

7. Si on n'avait que des *where/elsewhere*, cette valeur indiquerait le nombre de blocs inactifs imbriqués augmenté de 1, mais comme on a d'autres structures plus compliquées que l'on va voir ensuite, cela n'est pas vrai dans le cas général, du moins au niveau de la notion de blocs telle que le programmeur l'a décrite dans son programme. Le compilateur peut-être amené à modifier le compteur ou à rajouter des blocs dans son travail.

8. Les conditions ayant pour valeur dans ce qui suit :

- 0 si la condition est fautive;
- 1 si la condition est vraie.

TAB. 7.2 - Gestion du `whilesomewhere` avec un compteur.

Ouverture du <code>whilesomewhere(cond_évaluée)</code>	<i>Si</i> $CNT \neq 0$ ( <i>inactif</i> ), $CNT \leftarrow CNT + 2$ <i>Si</i> $CNT = 0$ ( <i>actif</i> ), $CNT \leftarrow 2 \times \neg cond\_évaluée$	
Début de boucle	Si toutes les $cond\_évaluée = 0$ ,	$\rightsquigarrow$ Fermeture du bloc
<code>continue</code>	<i>Si</i> $CNT = 0$ ( <i>actif</i> ),	$CNT \leftarrow 1^a$
<code>break</code>	<i>Si</i> $CNT = 0$ ( <i>actif</i> ),	$CNT \leftarrow 2^a$
Fin de la boucle	<i>Si</i> $CNT = 1$ ,	$CNT \leftarrow 0$
	<i>Toujours</i>	$\rightsquigarrow$ Début de boucle
Fermeture du bloc	<i>Si</i> $CNT \leq 1$ ,	$CNT \leftarrow 0$
	<i>Sinon</i>	$CNT \leftarrow CNT - 2$

<sup>a</sup> Valeur relative au bloc du `whilesomewhere` courant. Voir texte.

### Compilation d'un `whilesomewhere` et du `forwhere`

Pour le `whilesomewhere`, il faut gérer en plus les instructions `continue` et `break`. En effet un processeur peut être dans les différents états suivants :

- 1° déjà inactif avant le bloc `whilesomewhere`;
- 2° actif dans le `whilesomewhere`;
- 3° inactif dans le `whilesomewhere` car la condition de départ était fausse;
- 4° inactif dans le `whilesomewhere` suite à un `break`, et ce jusqu'à la sortie du `whilesomewhere`;
- 5° inactif dans le `whilesomewhere` suite à un `continue`, redeviendra actif à l'itération suivante.

Les points 3 et 4 ne sont pas différents une fois le `break` effectué : le processeur restera inactif jusqu'à la sortie du `whilesomewhere`. On peut donc les coder de manière identique.

Par rapport au cas `where/elsewhere`, on a donc l'état 5 à coder en plus. Cela se traduit par réserver un état supplémentaire par la double incrémentation du compteur de la 1<sup>ère</sup> rangée du tableau 7.2.

Les 3 états qui nous suffisent seront codés :

- 0** : le corps du `whilesomewhere` est actif;
- 1** : on est inactif suite à un `continue` (rangée 3) et on sera actif de nouveau à la fin de la boucle (rangée 5);
- 2** : on est inactif à cause d'une condition fausse au départ (rangée 2) ou suite à un `break` (rangée 4) jusqu'à la sortie du `whilesomewhere` (rangée 6).

Il faut enfin arrêter toutes les itérations scalaires (et parallèle, bien sûr) dès que toutes les conditions sont fausses. Cela nécessite l'utilisation de branchements scalaires, exécutés par tous les PES et représentés par le signe «  $\rightsquigarrow$  » sur le tableau 7.2.

Enfin, un cas un peu plus compliqué mais courant qu'il faut considérer est celui de **break** ou de **continue** contenus dans des **where/elsewhere**. Si on applique simplement la méthode du tableau 7.2 (note «<sup>a</sup> ») le **break** ou le **continue** aurait pour effet de « sortir » du **where/elsewhere** au lieu d'agir au niveau du **whilesomewhere**. Il faut donc corriger à la hausse la valeur mise dans le compteur en cas de **break** ou de **continue**.

Dans le cas d'un **whilesomewhere** sans **break** ni **continue**, on peut le compiler le **whilesomewhere** comme un **where** au niveau du compteur d'activité avec un **while** scalaire autour pour tester la condition d'itération globale. Cela permet par exemple d'augmenter l'efficacité sur une machine qui n'a pas le mécanisme des compteurs câblés.

Le **dowhere** associé au **whilesomewhere** découle de ce qui précède de la même manière qu'on passe du **while** au **do** et **while**.

Le **forwhere** étant dérivé du **whilesomewhere** de la même manière que le **for** l'est du **while**, la compilation peut se faire avec le même principe que celui utilisé pour le **whilesomewhere**.

### Compilation d'un **switchwhere**

Enfin, le cas du **switchwhere** se rapproche du **whilesomewhere** au niveau du nombre d'état. Il faut considérer les états suivants :

- 1° déjà inactif avant le bloc **switchwhere** ;
- 2° actif dans un **case** ou un **default** ;
- 3° inactif dans un **case** ;
- 4° inactif dans le **switchwhere** suite à un **break**, jusqu'à la sortie du **switchwhere**.

Le **break** du **switchwhere** se comporte comme le **break** du **whilesomewhere**. Il faut donc lui réserver une identification particulière. On peut choisir comme codage pour les compteurs :

- 0** : le corps du **case** est actif ou on a rencontré un **default** ;
- 1** : on est inactif dans un **case** ;
- 2** : on est inactif à cause d'un **break** jusqu'à la sortie du **switchwhere**.

Cela nous amène à la réalisation suivant le tableau 7.3.

Le cas du **switchwhere** sans **break** peut être optimisé de la même manière qu'un **whilesomewhere** sans **break** ni **continue** par exemple<sup>9</sup>.

### Cas du **return** parallèle

Lorsqu'un **return** est encapsulé dans un bloc de contrôle de flot parallèle, il y a désactivation des PVS de la collection correspondant à la valeur de retour de la fonction qui exécutent le **return**.

Cette désactivation est réalisée en mettant dans les compteurs correspondants la valeur nécessaire pour neutraliser toutes les imbrications de blocs de conditionnement

---

9. On peut en fait considérer ce dernier cas comme n'arrivant qu'exceptionnellement et donc compiler à chaque fois sans se soucier d'une absence éventuelle de **break**.

TAB. 7.3 - Gestion du *switchwhere* avec un compteur.

Ouverture du <code>switchwhere(valeur)</code>	<i>Si</i> $CNT \neq 0$ ( <i>inactif</i> ), $CNT \leftarrow CNT + 2$ <i>Si</i> $CNT = 0$ ( <i>actif</i> ), $CNT \leftarrow 1$
<code>case constante :</code>	<i>Si</i> $valeur = constante$ <i>et</i> $CNT = 1$ ( <i>activable</i> ), $CNT \leftarrow 0$
<code>break</code>	<i>Si</i> $CNT = 0$ ( <i>actif</i> ), $CNT \leftarrow 2^a$
<code>default :</code>	<i>Si</i> $CNT \leq 1$ ( <i>activable</i> ), $CNT \leftarrow 0$
Fermeture du bloc	<i>Si</i> $CNT \leq 1$ , $CNT \leftarrow 0$ <i>Sinon</i> $CNT \leftarrow CNT - 2$

<sup>a</sup> Valeur relative au bloc du `switchwhere` courant, comme dans le cas du `whilesomewhere` du tableau 7.2.

TAB. 7.4 - Cas d'un *return* parallèle dans un conditionnement parallèle.

Ouverture de la fonction	<i>Si</i> $CNT \neq 0$ ( <i>inactif</i> ), $CNT \leftarrow CNT + 1$
<code>return</code>	<i>Si</i> $CNT = 0$ ( <i>actif</i> ), $CNT \leftarrow CNT + r$
Fermeture de la fonction	<i>Si</i> $CNT \neq 0$ , $CNT \leftarrow CNT - 1$

parallèles de la fonction qui contiennent ce `return`, selon un principe similaire au `break`. Il faut considérer les états suivants :

- 1° un PV est actif à l'entrée de la fonction ;
- 2° inactif à l'entrée de la fonction ;
- 3° un PV exécute un `return` qui le désactive jusqu'à la fin de la fonction.

Il faut donc être capable de distinguer les 2 derniers cas pour que le retour de fonction ne désactive pas des PVs en dehors de la visibilité du `return` (c'est-à-dire de la fonction), ce qui est fait en réservant une valeur du compteur à cet effet par l'intermédiaire d'un `where(vrai)` autour du corps de la fonction (figure 7.4).  $r$  est une valeur calculée par le compilateur pour qu'une fois sorti de tous les blocs parallèles conditionnés de la fonction, le  $CNT$  du PV se retrouve à 1, indiquant qu'on a exécuté un `return` avec succès et empêchant une réactivation du PV avant la fin de la fonction.

#### 7.1.4 Un bit d'activité

Nous abordons ici en comparaison l'approche classique de conditionnement SIMD qui fait appel à la notion de bit d'activité.

Puisqu'on se place dans le cas d'un langage parallèle et structuré raisonnable, on n'est pas censé avoir droit à des débranchements conditionnels vers n'importe quel point de la fonction ou procédure courante<sup>10</sup>. On se contente très bien d'avoir un conditionnement parallèle au niveau de blocs d'instructions.

<sup>10</sup>. Fortran n'est donc pas un « langage parallèle et structuré raisonnable ».

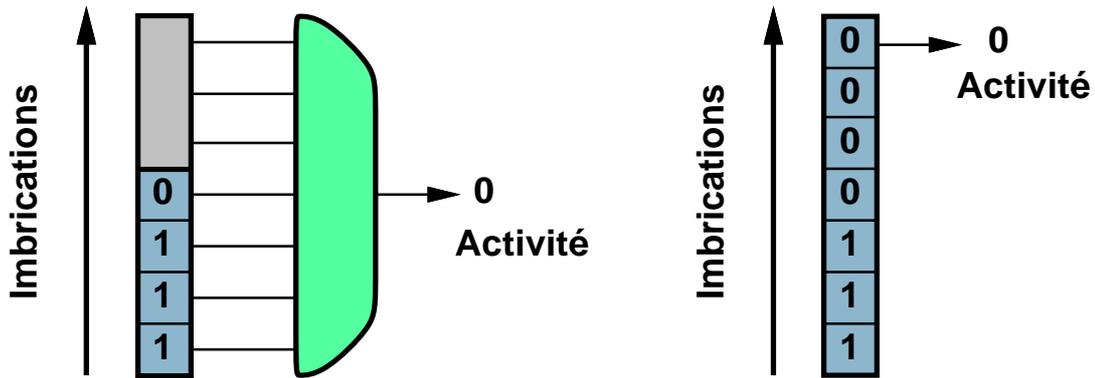


FIG. 7.3 - Pile d'activité et pile d'activité factorisée.

Dans ce cas l'exécution par un processeur parallèle du programme n'a lieu que lorsque le séquenceur exécute le début d'un bloc conditionné d'instructions où la condition locale est vérifiée. On a vu que la notion d'adresse d'instruction devient alors trop précise et on pourrait se contenter de numérotter les blocs.

En fait, comme le conditionnement va d'un début de bloc à une fin de bloc, si on est capable de détecter les début et fin de bloc, il suffit d'associer à chaque processeur parallèle une notion d'activité [SBM62, BBK<sup>+</sup>68, Sto71, KK82] qui est sauvegardée sur une pile à chaque entrée dans un bloc conditionné et altérée dans le bloc par la condition  $a_i$  [Aug85, page 166] [SSKD87, page 392].

On voit ici la différence entre le compteur d'activité et le bit d'activité: alors que dans le premier cas c'est une valeur d'activité (entité virtuelle puisque le compteur est unique) qui est associée à un moment donné à un bloc conditionné, dans le second cas c'est un bit d'activité qui est associé physiquement à chaque bloc conditionné.

Un processeur ne sera actif dans un bloc que si tous les bits d'activité qu'il a sur sa pile, qui résument en quelque sorte son histoire, indiquent une activité vraie, *ie* le processeur est dans une imbrication de blocs où toutes les conditions intermédiaires sont vraies,  $\bigwedge_{i \leq \#pile} a_i = 1$ , en considérant qu'une condition  $a_i$  est vraie si  $a_i = 1$  et fausse si  $a_i = 0$ .

#### 7.1.4.1 Factorisation

Plutôt que d'être obligé de calculer  $\bigwedge_{i \leq \#pile} a_i$  à chaque instruction (figure 7.3, dessin de gauche), on peut stocker sur la pile seulement le fait d'être actif ou non dans tous les blocs imbriqués c'est à dire seulement  $a_{\#pile} \wedge (\bigwedge_{i < \#pile} a_i)$  (figure 7.3, dessin de droite).

Il suffit de tester le haut de la pile pour savoir si l'instruction courante sera exécutée ou non. A chaque fois qu'on entre dans un nouveau bloc conditionnel, on se contente de mettre sur la pile  $cond \wedge a_{\#pile-1}$ .

Notons que la pile d'activité non factorisée n'a pas d'existence réelle telle qu'elle est décrite sur la figure 7.3 avec ses 0 ou 1 au dessus du 1<sup>er</sup> 0 dans la mesure où une fois qu'un processeur est désactivé, il va mettre n'importe quoi en haut de sa pile à chaque nouvelle instruction de conditionnement parallèle. Finalement seul compte l'endroit de la désactivation. C'est ce qui va être utilisé en § ??.

TAB. 7.5 - Fonctionnement de la pile lors d'une condition parallèle.

<b>Avant :</b>	Actif		Inactif	
$v$	$= \#pile + 1$		$< \#pile + 1$	
<b>Condition :</b>	<i>vraie</i>	<i>fausse</i>	<i>vraie</i>	<i>fausse</i>
<b>Dans le bloc :</b>	Actif	Inactif	Inactif	Inactif
$\#pile$	$++^a$	$++$	$++$	$++$
$v$	$++$	<i>Inchangé</i>	<i>Inchangé</i>	<i>Inchangé</i>
$\#pile - v + 1$	<i>Inchangé</i> ( $= 0$ )	$++$	$++$	$++$

<sup>a</sup> Notation empruntée au langage C signifiant une incrémentation.

Il est donc nécessaire d'avoir un pointeur de pile global ou des pointeurs de pile locaux aux processeurs parallèles de  $\lceil \log_2 c \rceil$  bits et d'une zone de stockage de  $c$  bits dans tous les processeurs.

C'est la solution à pointeur de pile global qui est choisie dans la plupart des machines SIMD car il est souvent moins coûteux d'avoir un calcul scalaire du pointeur quitte à l'envoyer à tous les PES plutôt que de calculer un pointeur de pile par PE si ces derniers sont à grain fin.

#### 7.1.4.2 Equivalence entre pile et compteur d'activité

Si on regarde d'un peu plus près le fonctionnement du mécanisme précédent, on constate par récurrence que

$$(\exists v \in [0, \#pile] : a_v = 0) \Rightarrow (\forall i \in [v, \#pile], a_i = 0)$$

Le bas de la pile est éventuellement constitué de 1 (tous les blocs actifs) et à partir d'un éventuel  $v^{\text{ème}}$  bit, tous les éléments de la pile sont nuls (tous les blocs sont inactifs depuis la condition fausse du  $v^{\text{ème}}$  bloc).

Il est donc dommage d'utiliser une pile alors que, si on connaît  $v$  et la valeur du compteur de pile  $\#pile$ , on a toute l'information intéressante. En effet, supposons que lorsqu'aucun bloc n'est inactif (ie tous les blocs imbriqués sont actifs) on pose par convention  $v = \#pile + 1$ , on est dans un ou plusieurs blocs inactifs si et seulement si

$$\#pile - v + 1 > 0$$

En outre cette valeur fournit le nombre de blocs inactifs imbriqués.

Le fonctionnement de la pile lorsqu'on rencontre une condition parallèle est résumé dans la table 7.5 et dans la table 7.6 pour les fins de bloc.

On peut donc remplacer le mécanisme de pile de condition locale (de  $c$  bits) à chaque processeur par un compteur local fonctionnellement équivalent (de  $\lceil \log_2 c \rceil$  bits) à chaque processeur.

Les personnes intéressées par une démonstration plus formelle et plus complète de l'équivalence entre compteur et pile d'activité peuvent se reporter à [?].

TAB. 7.6 - *Fonctionnement de la pile lors d'une sortie de condition parallèle.*

Avant :	Actif	Inactif	
Sortie du bloc :	Actif	Actif	Inactif
$\#pile$	-- <sup>a</sup>	--	--
$v$	--	<i>Inchangé</i>	<i>Inchangé</i>
$\#pile - v + 1$	<i>Inchangé</i>	-- (= 0)	--

<sup>a</sup> Notation empruntée au langage C signifiant une décrémentement.

TAB. 7.7 - *Complexité des méthodes de conditionnement SIMD.*

Méthode de conditionnement //	Temps de calcul		Complexité matérielle	# Comm.
	Séq.	//		
Piles (pointeur global)	$T$	$t$	$Nc + \lceil \log_2 c \rceil$	1
Piles (pointeurs locaux)	$\epsilon$	$t(1 + \lceil \log_L c \rceil)$	$N(c + \lceil \log_2 c \rceil)$	0
Compteurs locaux	$\epsilon$	$t \lceil \log_L c \rceil$	$N \lceil \log_2 c \rceil$	0

### 7.1.5 Arguments de choix

En fait, la granularité de la machine va fortement influencer le choix entre la méthode avec pile et la méthode avec compteur suivant des critères de temps d'exécution et de complexité. La table 7.7 donne une idée de la complexité des méthodes précédentes au niveau du temps et du matériel en fonction de plusieurs paramètres.

$T$  est le temps d'exécution d'une instruction du séquenceur (on suppose que le séquenceur est suffisamment puissant pour pouvoir gérer la pile en une seule instruction),  $t$  est le temps d'exécution d'une instruction sur les processeurs parallèles de  $L$  bits de large.

Même si le tableau précédent donne une indication de la complexité matérielle en fonction du nombre maximal  $c$  d'imbrications de conditions, il ne faut pas perdre de vue que le fait d'avoir un  $c$  élevé dans un modèle d'exécution purement SIMD fait baisser dramatiquement l'efficacité de la machine, donc on s'arrangera pour réduire ce paramètre le plus possible et par conséquent la taille de la pile nécessaire pour stocker l'activité ne sera pas un facteur limitant pour une application se prêtant bien au SIMD.

On constate que lorsqu'on a une machine à grain fin ( $L$  proche de 1) et qu'on veut aller le plus vite possible, il vaut mieux sous-traiter le calcul du pointeur de pile à un processeur central puissant capable de faire l'opération en un cycle plutôt que de la faire localement par tranche de  $L$  bit(s), même si cela coûte l'émission du pointeur de pile vers tous les processeurs. C'est ce qui est fait sur les machines à grain fin comme la CM-2 ou la MP-1.

Si on a une machine à gros grain, c'est clairement la 3<sup>ème</sup> solution qui convient car

1. il est difficile de gérer une pile de bits sur un processeur de plus de 1 bit de large, mieux conçu pour faire des accès d'au moins  $L$ -bits de large, ce qui prouve que dans certains cas les processeurs étroits ont un intérêt;
2. une indirection mémoire coûte souvent relativement cher et est à éviter;

3.  $\lceil \log_L c \rceil = 1$  donc la gestion du compteur se fait très rapidement ;
4. le processeur scalaire est libéré d'une tâche superflue, qui peut alors être utilisé à des opérations plus utiles pouvant aider à atteindre la superlinéarité (voir § 11.3) ;
5. c'est aussi la solution la plus économique en temps, matériel et communication.

Cette méthode aurait donc été employée avec intérêt sur des machines telles que l'ILLIAC IV, GF11 ou OPSILA, et la CM-5 en ce qui concerne les machines actuelles. Dans ce dernier cas le fait d'utiliser des bits d'activités est particulièrement mauvais puisque la machine est basée sur des processeurs à gros grains et que le langage C\* autorise l'imbrication de `where` [Thi91].

De toute manière, si on doit construire une machine SIMD, il faut réaliser un système de compteurs locaux câblés afin de perdre le minimum de temps là où les machines SIMD sont réputées être inefficaces.

Du coup, cette discussion peut très bien s'adapter à la manière de générer du code pour une machine SIMD existante : choisir de compiler des compteurs ou pas lorsqu'il n'y a pas ce mécanisme (voir le chapitre 8) afin de produire le code le plus efficace.

Enfin, on peut réaliser plusieurs jeux de compteurs d'activités pour pouvoir gérer rapidement plusieurs jeux de processeurs virtuels si on veut pouvoir optimiser des collections à faible *vp\_ratio* connu de manière statique. Mais cela complique le développement matériel pour une optimisation qui n'est que très rarement possible, donc ce n'est pas techniquement très intéressant.

Pour mettre en œuvre le mécanisme de contrôle de flot SIMD, quel que soit son type, il faut étiqueter les instructions d'un attribut précisant si telle ou telle instruction est conditionnée ou pas, si elle est sensible au `where` ou au `elsewhere`. Selon le degré de raffinement, on peut compliquer les étiquettes pour qu'elles incluent les fonctions d'ouverture et de fermeture de bloc conditionné, etc. et sont rajoutées sous forme de suffixes aux instructions sous leur forme assembleur (§§ ?? et ??).

Cette étiquette, généralement un ou quelques bits supplémentaires dans le code de chaque instruction, est souhaitable pour faciliter le décodage de l'instruction, quoique non indispensable, car on a souvent besoin d'insérer des instructions non conditionnées dans des blocs conditionnés, ne serait-ce par exemple les instructions de manipulation de ce bit d'activité et permettre ainsi de séparer les instructions des processeurs du contrôle de l'activité.

### 7.1.6 Problématique dans le cas du MIMD

Dans le cas d'une machine MIMD qu'on voudrait programmer avec du parallélisme de données [?, QS90, HLJ<sup>+</sup>91, HQ91], la notion d'activité existe encore car il faut être capable de gérer l'imbrication de `where` concernant des collections, cachée potentiellement par des appels de fonctions, ou des utilisations de collections différentes entrelacées dans l'algorithme que l'on ne peut pas séparer.

Dans ce cas, la compilation revient toujours à utiliser le mécanisme d'activité comme dans [PCMP85]. Mais quel principe utiliser ? Il est clair que le mécanisme de pile à pointeur global est inutilisable car du fait de l'asynchronisme il est beaucoup plus rapide de gérer la pile localement, d'autant plus que la plupart des machines MIMD

sont à gros grain. Reste le choix entre la pile de bits d'activité gérée localement et le système de compteur d'activité.

La pile d'activité nécessitera de toute manière d'être stockée en mémoire car on ne peut pas gérer efficacement une pile dont les valeurs sont stockées en registre. En plus, le fait que le processeur est probablement à gros grain signifie qu'il sera sous-utilisé à chaque manipulation du bit d'activité courant de même que la mémoire car on sera sans doute obligé d'utiliser un mot mémoire pour contenir chaque élément de la pile.

Par contre le compteur d'activité pourra loger dans un registre du processeur et ne nécessitera aucun accès mémoire pour gérer des imbrications de conditions parallèles si on travaille au niveau des processeurs physiques<sup>11</sup>. Bien entendu, dès qu'on utilise des processeurs virtuels, il faut stocker le compteur d'activité de chaque processeur virtuel en mémoire. Néanmoins le compteur n'utilise qu'un mot mémoire et la localité comparée à la méthode à pile permet une meilleure exploitation de la mémoire cache des processeurs.

Par contre il ne semble pas intéressant de réaliser un système matériel de gestion de l'activité car cela nécessiterait des allers-retours incessants entre le compteur d'activité extérieur, le processeur et la mémoire, comparés aux seuls allers-retours entre processeur et mémoire sans système particulier, réduisant fortement l'utilisation du pipeline du processeur dans le cas d'un processeur moderne très fortement pipeliné.

Donc dans le cas d'une machine MIMD, la méthode du compteur d'activité logiciel est clairement la bonne méthode pour compiler des langages à parallélisme de données basés sur la notion de collection lorsqu'on ne peut compiler le contrôle de flot parallèle sous forme de `if` locaux.

## 7.2 Méthodes de contrôle des instructions parallèles

Les discussions précédentes décrivaient des méthodes pour décider des endroits où les processeurs doivent être activés ou désactivés mais nous n'avons pas parlé de la manière de les désactiver physiquement. C'est le problème qui va être abordé maintenant.

### 7.2.1 Le contrôle direct de l'exécution

Il s'agit là évidemment du moyen le plus efficace pour mettre en œuvre le contrôle de flot décrit précédemment.

Si on conçoit un processeur élémentaire, il suffit donc de prévoir une possibilité de non exécution d'instruction en fonction d'un bit d'activité. On retrouve ce mécanisme sous la forme de vecteurs de masques d'activité dans les machines vectorielles japonaise comme le VP-200 [MU84] par exemple pour permettre d'exécuter rapidement des boucles vectorisables contenant des `if` vectoriels.

Si on doit adapter un processeur existant, le plus simple est d'utiliser les fils de contrôle de son bus d'instructions pour faire croire que l'instruction présentée sur ce dernier n'est pas prête lorsqu'on ne veut pas l'exécuter. C'est ce mécanisme qui est utilisé avec d'autres mécanismes sur POMP.

---

11. Notons que cette optimisation n'est pas possible sur un TRANSPUTER puisque cette notion de registre n'apparaît pas.

---

```

collection tableau;
... /* La taille est d'efinie ailleurs. */
double tableau a,b; /* a et b sont */
... /* des variables parallèles. */
void div_par_0()
{
    double tableau rien;

    rien = a; /* On passe par une variable intermediaire */
    where (!(a != 0))
        rien = 1.0; /* afin de ne pas salir a. */
    where (a != 0) /* On peut faire alors */
        b = 1/rien; /* la division en toute securite. */
    where (!(a != 0))
        b = 0; /* Pourquoi pas! */
}

```

---

FIG. 7.4 - L'exemple de la figure 7.1 corrigé pour ne nécessiter que du contrôle local d'écriture en mémoire.

### 7.2.2 Décision locale d'écriture en mémoire locale

Si on n'a pas de mécanisme contrôlant directement l'exécution, on peut prendre un point de vue purement fonctionnel de la chose et dire que si une instruction n'a aucun effet de bord, c'est comme si elle ne s'était jamais exécutée.

Comme l'effet de bord typique d'une instruction dans une machine de VON NEUMANN est d'écrire un résultat en mémoire, on peut se restreindre à supprimer cette écriture de manière conditionnelle.

Beaucoup de machines, comme les CRAY [Rus78] ou GF11 [BDW85], utilisent l'autorisation locale d'écriture en mémoire locale pour effectuer du contrôle de flot SIMD en considérant que le contrôle de l'écriture finale en mémoire du résultat d'une opération suffit.

Cela n'est que moyennement satisfaisant :

- on ne peut pas bénéficier de stockage de variables en registre quand on en a à sa disposition puisqu'on peut faire de l'écriture en mémoire conditionnée mais pas de l'écriture conditionnée en registre ;
- le contrôle de toutes les instructions, surtout celles qui n'écrivent pas un résultat en mémoire, par cette méthode devient aussi infernal que dans le cas où on n'a aucun mécanisme de contrôle SIMD à notre disposition (voir le programme `div_par_0` revu et corrigé sur la figure 7.4).

Si l'ordinateur qui doit exécuter ce programme n'a pas de mécanisme contrôlant l'exécution locale des instructions et qu'on ne veut pas avoir un arrêt de la machine pour « division par 0 », le compilateur retradaira<sup>12</sup> le programme en quelque chose de

---

<sup>12</sup> On suppose le cas optimiste où le compilateur ne refuse pas tout simplement de compiler le programme sous forme parallèle, donc que ce ne sera pas le travail du programmeur.

la forme du programme de la figure 7.4 où chaque **where** n'affecte que l'écriture du résultat des instructions qu'il conditionne et non leur exécution.

Même si cette méthode a été et est toujours utilisée dans certaines machines en argumentant qu'on peut toujours simuler ce contrôle SIMD en un nombre constant d'opérations<sup>13</sup> et que l'utilisation d'un adressage indirect local peut dans certains cas accélérer l'exécution des programmes après réécriture selon le système exposé ci-dessus, il est clair que c'est très inefficace.

En outre cela complique énormément le compilateur qui doit effectuer une véritable recherche opérationnelle à chaque fois qu'il veut compiler un **where** au mieux tout en respectant la sémantique macroscopique du programme et en empêchant la génération d'effets de bord parasites comme les exceptions.

Finalement il n'est guère plus compliqué de réaliser un bit d'activité plutôt qu'un bit d'autorisation d'écriture en mémoire : le bit d'activité ne fait que bloquer (d'un point de vue seulement sémantique ou pas, selon la réalisation) les écritures dans les registres internes du processeur — y compris ceux déclenchant les exceptions — en plus de bloquer les écritures en mémoire.

En outre, si on veut par exemple utiliser un processeur du commerce, il est plus simple d'empêcher l'exécution d'une instruction au moment où elle entre dans le processeur qu'une écriture en mémoire qui peut avoir lieu un temps variable après l'entrée dans le pipeline de l'instruction d'écriture. En effet cette dernière solution obligerait à connaître la latence du pipeline interne du processeur qui est non déterministe si une exception arrive, par exemple.

Néanmoins cette approche à un intérêt dans un domaine différent : les compilateurs générant du code avec exécution spéculative des différentes branches d'un test sur une machine superscalaire ou VLIW. L'analyse du graphe de dépendance est la même et à chaque fois qu'on ne sait pas suffisamment tôt quelle branche choisir, on exécute en parallèle chaque branche et on n'utilise pas les résultats calculés de la branche qui n'est pas choisie. Il faut être capable d'empêcher les effets de bord des branches non exécutées par le principe de réécriture exposé ci-dessus si on veut faire de l'optimisation statique des branchement. Mais certains processeurs superscalaires sont conçus pour gérer ce mécanisme de manière automatique à l'exécution [?, ?].

### 7.2.3 Utilisation de l'adressage local

Un moyen de simuler un contrôle local de l'écriture en mémoire lorsqu'il n'est pas disponible mais qu'on dispose d'un adressage indirect local en mémoire est de transformer toutes les écritures conditionnelles en écritures indirectes écrivant soit au bon endroit en mémoire soit dans une case servant de « trou noir » pouvant accepter toutes les valeurs.

Cette solution de remplacement de dernière minute n'est guère intéressante dans le cas de la construction d'une nouvelle machine puisqu'elle hérite des inconvénient de la méthode du contrôle de l'écriture, ralentie de surcroît par le fait qu'on simule ce mécanisme : cela coûte une indirection supplémentaire avec un calcul spécifique sur l'index. Néanmoins, elle coûte un matériel nul, ce qui peut être un critère clé.

---

13. Constant ne signifie pas hélas petit ou négligeable dans la pratique !

### 7.2.4 Introduction d'instructions SIMD conditionnelles

D'autres approches ont été de compenser l'unicité des instructions sur tous les PEs par l'adoption d'instruction à signification variable en fonction d'un contexte de condition afin de se rapprocher des machines MIMD.

Dans la machine BLITZEN est introduit, en plus du bit de masquage des instructions, un bit  $K$  transformant l'action de certaines instructions [BDR87]. Cela peut être utile lorsqu'on a une machine à base de processeurs peu puissants, à grain fin, et qu'on veut accélérer le calcul sur les nombres flottants, comme les opérations de division ou de valeur absolue par exemple.

Malheureusement, ce bit ne permet de transformer que localement une addition en soustraction, ce qui limite son utilisation au microcode conditionné tel qu'on peut le trouver dans des applications de traitement d'image de bas niveau, la méthode étant peut efficace à conditionner des blocs d'instructions importants et très difficile à implanter au niveau d'un compilateur.

On pourrait étendre la méthode à d'autres microinstruction mais cela ne changerait pas fondamentalement la nature du problème.

Dans la machine GF11 [BDW85], outre le fait que toutes les écritures sont conditionnelles, les opérations diadiques sont conditionnées dans le sens où elles renvoient soit le résultat soit le premier opérande, chaque opérande pouvant lui-même être conditionnellement indirect, c'est-à-dire prendre une valeur en registre ou en mémoire suivant la condition locale. Mais comme le microcode n'est pas figé, on peut toujours reprogrammer des instructions conditionnées.

Même s'il est théoriquement possible de tout faire avec les mécanismes précédents, ils sont trop complexes pour être utilisés de manière automatique par un compilateur — d'ailleurs aucune de ces machines n'a possédé de compilateur « classique » — sur de gros blocs d'instructions et semblent plutôt réservés à l'optimisation manuelle de petit bloc<sup>14</sup>.

Un cas typique de ce genre d'optimisations est donné dans les processeurs scalaires ARM [?], où chaque instruction est conditionnée, et ALPHA [Dig92a], où il existe des instructions faisant des transferts conditionnels de registres. Ces instructions sont utilisées par le compilateur pour limiter localement les débranchements coûteux nécessaires au conditionnement de petits blocs d'instructions et pour faire de l'exécution spéculative pour remplir les trous du pipeline.

On pourrait donc imaginer utiliser ces processeurs pour construire une machine SIMD. Malheureusement le premier n'a ni UAL flottante ni architecture HARVARD tandis que le dernier possède un cache et aussi une architecture non-HARVARD. Le monde n'est pas toujours parfait...

### 7.2.5 Passage en mode SPMD

Tout bien considéré, on peut à juste titre se dire que tout ceci est bien compliqué alors qu'une machine MIMD peut faire cela très simplement. La gestion la plus efficace des conditions en SIMD n'est-elle pas de ne pas les exécuter en MIMD?

C'est le principe des machines SPMD : elles exécutent bien le même programme mais chaque branche de test est exécutée indépendamment sur tous les processeurs. L'incon-

---

14. Qui sont tout simplement les instructions de la machine, composées de microinstructions.

vénient se retrouve au moment des ressynchronisations et au niveau de la réplication des codes.

Au moins 3 machines exploitent cette souplesse supplémentaire, les machines PASM, OPSILA et TRITON/1, toutes 3 SPMD.

Dans le premier cas les processeurs de PASM peuvent exécuter directement des instructions dans leur mémoire locale plutôt que d'exécuter les instructions émises par le séquenceur, en plus du mécanisme de pile d'activité [SSKD87]. L'utilisation des modes SIMD et SPMD est simple car les processeurs voient dans leur espace d'adressage programme à la fois les instructions fournies une à une par le séquenceur et la mémoire locale. Une simple instruction de branchement fait passer d'un mode à l'autre. La machine TRITON/1 [?] lui est très similaire.

Dans le deuxième cas les processeurs d'OPSILA peuvent aussi exécuter des instructions dans leur mémoire locale dans le mode SPMD, selon deux méthodes [AB86] :

- 1° la machine passe globalement en mode SPMD (équivalent à un `fork`) et chaque processeur exécute ses instructions locales jusqu'à la sortie de ce mode (par un mécanisme de type `join`).
- 2° chaque processeur peut exécuter une instruction en mémoire à une adresse locale par l'intermédiaire d'une instruction `CASE`, moins lourde à utiliser pour exécuter de petits blocs conditionnés. Cela se rapproche des instructions `execute` de certains processeurs.

Evidemment cela complique beaucoup l'architecture de la machine puisqu'il faut un chemin de données supplémentaire entre la mémoire locale et le décodeur d'instruction d'un PE et bien entendu cela nécessite d'avoir un décodeur d'instruction par PE.

### 7.2.6 Utilisation des caractéristiques des processeurs modernes

On va voir que l'utilisation de processeurs « modernes » dans une machine SIMD la fait bénéficier de nouveaux avantages en ce qui concerne le contrôle de flot SIMD en soi et les optimisations qu'on peut y apporter, bien que ces processeurs ne soient pas prévus pour cette cause.

Ces méthodes sont complémentaires de la méthode du compteur d'activité et selon les cas sont plus efficaces. Néanmoins elles ne s'adressent qu'au cas de `where/elsewhere` terminaux, c'est-à-dire qui ne contiennent pas d'autre conditionnement parallèle et donc dont on contrôle parfaitement les effets de bord<sup>15</sup>. Cette restriction n'est pas très forte car il s'agit là du cas le plus courant (le seul autorisé dans FORTRAN 90 d'ailleurs).

#### 7.2.6.1 Branchements conditionnels non retardés

Une des caractéristiques des processeurs récents est de posséder ce que l'on appelle des branchements retardés : le branchement n'a lieu que quelques<sup>16</sup> cycles après l'exécution de l'instruction du branchement, comme indiqué sur l'exemple de la figure 7.5.

Le processeur pipeliné qui rencontre l'instruction de branchement non retardé a déjà lu l'instruction `Instr4` alors qu'il n'en avait pas besoin. Il l'efface de son pipeline

15. Cela écarte donc les appels de fonctions dans de tels blocs.

16. Déterministe tout de même et typiquement égal à 2.

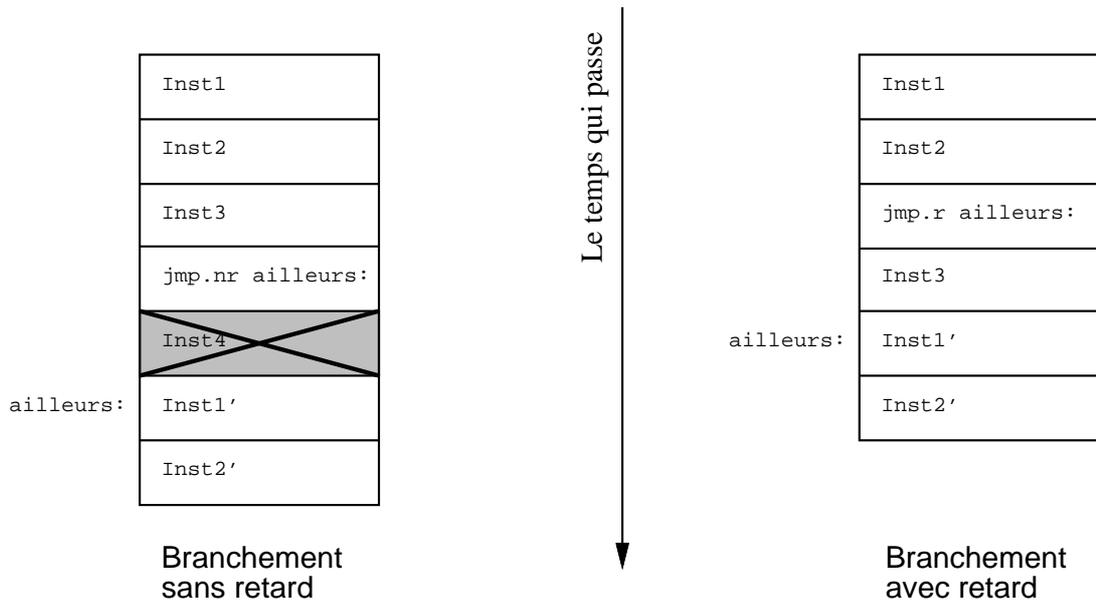


FIG. 7.5 - Exécution d'un branchement non retardé et retardé.

d'exécution tout en continuant l'exécution ailleurs. Il a donc perdu dans cet exemple un cycle.

Par contre le processeur qui a un branchement retardé exécute cette instruction 1 cycle plus tôt et n'a pas besoin d'effacer d'instruction qu'il aurait lu en trop. Ce mécanisme est utile si l'instruction de l'emplacement retardé<sup>17</sup> ne produisait pas une dépendance à travers une condition nécessaire au branchement dans le cas où ce dernier est conditionné.

Même si cette méthode est un peu troublante, elle permet d'augmenter l'efficacité du processeur lorsqu'il effectue beaucoup de branchements et que le désir de saut est connu suffisamment tôt.

Cela revient à pipeliner aussi les instructions de saut : par principe on exécute toutes les instructions suivant l'instruction de saut qui sont présentes dans le pipeline, plutôt que de vider le pipeline avant de recommencer l'exécution à un autre endroit du programme.

Paradoxalement en ce qui nous concerne, un branchement non retardé peut être intéressant car il permet de vider le pipeline d'instruction, et ce conditionnellement si on a une instruction de branchement conditionnel non retardé. En effet dans notre machine SIMD on va pouvoir ainsi décider localement de ne pas exécuter les instructions que nous envoie le séquenceur de la machine. L'application à notre exemple de départ est donnée dans la figure 7.6 : le **where** et le **elsewhere** sont chacun réalisés par un branchement conditionnel sur une condition opposée. Lorsque le branchement est pris, l'instruction suivante est annulée. Peu importe l'adresse de saut bien entendu puisqu'elle n'est jamais utilisée : chaque processeur est nourri de force sur son bus d'instruction.

Mais il peut être astucieux de ne pas mettre n'importe quoi comme adresse de saut. En temps normal, le compteur de programme est incrémenté de 1 à chaque cycle

17. Le « *delay slot* ».

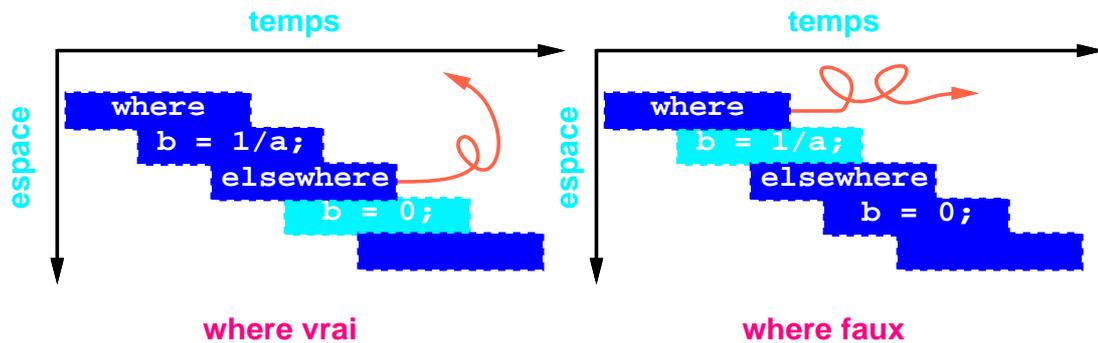


FIG. 7.6 - Utilisation des branchements non retardés pour réaliser le contrôle de flot parallèle.

d'horloge, ce qui nous fournit une bonne horloge temporelle que l'on peut exploiter. Il suffit de s'arranger pour que les branchements ne la perturbe pas et donc sauter à l'adresse courante ajoutée du nombre de cycles de latence. En cas d'exception, le compteur de programme est sauvegardé et est restauré en fin d'exception. C'est donc le « temps utilisateur », c'est à dire le temps passé par la machine à résoudre le problème sans compter les appels systèmes, qui est indiqué par le compteur de programme.

L'avantage par rapport au contrôle externe de l'exécution est qu'on n'a plus besoin de sortir la condition à l'extérieur du processeur, ce qui est pénalisant en temps, pour pouvoir conditionner une instruction. En plus, ce mode de contrôle ne nécessite pas de matériel supplémentaire, bien que typiquement SIMD.

La contrepartie est que la visibilité de la méthode est égale à la latence d'exécution du branchement non retardé, typiquement 1 cycle sur le MC88100, et donc il faudra rajouter autant d'instructions de branchement que nécessaire.

Soient  $\ell_b$  la latence du branchement non retardé,  $\ell_e$  la latence de l'écriture en mémoire<sup>18</sup>,  $\ell_h$  la latence de l'HyperCom,  $\ell_i$  la latence du pipeline entre l'entrée d'une instruction et son exécution<sup>19</sup> et  $n_c$  le nombre d'instructions du bloc à conditionner, on peut calculer le temps d'exécution selon qu'on utilise la méthode du compteur d'activité ou du branchement non retardé, comme indiqué dans le tableau 7.8, où toutes les latences sont exprimées sans dimension, en nombre de cycles d'horloge du processeur.

Les efficacités pipelinée et non pipelinée correspondent au nombre d'instructions divisé par le temps d'exécution en cycle lorsque les « trous » dans le pipeline sont remplis par des instructions utiles mais n'ayant rien à voir avec le bloc conditionné ou bien sont laissés vides, respectivement.

La méthode du branchement non retardé n'introduit pas de trou, si ce n'est ceux résultant de la visibilité du dernier branchement non retardé non rempli. Dans ce dernier cas, on ne peut utiliser la place laissée vide car, étant sous la visibilité du branchement non retardé, elle est conditionnée comme telle. C'est ce qui explique que les valeurs du temps d'exécution en mode pipeliné et en mode non pipeliné soient les mêmes.

On constate que si on peut remplir parfaitement le pipeline, la méthode du compteur

18. L'accès au registres de l'HyperCom permettant le contrôle des conditions par compteur se fait par une instruction d'écriture en mémoire puisque l'HyperCom est sur le bus mémoire du processeur.

19. Etant sur une machine SIMD, cette latence ne tient pas compte de l'accès à une mémoire programme qui n'existe pas ! On gagne donc du temps, un cycle dans le cas du MC88100.

TAB. 7.8 - Comparaison entre deux méthodes de conditionnement SIMD.

	Méthode	
	Compteur externe d'activité	Branchement non retardé
Nombre d'instructions	$n_c$	$n_c + \lfloor \frac{n_c}{l_b} \rfloor$
<i>Mode pipeliné</i>		
Temps d'exécution	$l_e + l_h + l_i + n_c$	$(l_b + 1) \lfloor \frac{n_c}{l_b} \rfloor$
Efficacité	1	$\frac{n_c}{(l_b + 1) \lfloor \frac{n_c}{l_b} \rfloor}$
<i>Mode non pipeliné</i>		
Temps d'exécution	$l_e + l_h + l_i + n_c$	$(l_b + 1) \lfloor \frac{n_c}{l_b} \rfloor$
Efficacité	$\frac{1}{1 + (l_e + l_h + l_i)/n_c}$	$\frac{n_c}{(l_b + 1) \lfloor \frac{n_c}{l_b} \rfloor}$

externe est la meilleure et que sinon l'utilisation du compteur est plus intéressante asymptotiquement, d'un facteur 2 pour la machine POMP actuelle.

Mais il y a beaucoup de tests qui ne concernent que peu d'instructions et, dans ce cas, le facteur de mérite

$$\mathcal{F}_{branch/compt} = \frac{n_c + l_e + l_h + l_i}{(l_b + 1) \lfloor \frac{n_c}{l_b} \rfloor}$$

indique une préférence pour le branchement non retardé, dès que

$$\lfloor \frac{n_c}{l_b} \rfloor \leq \frac{n_c + l_e + l_h + l_i}{l_b + 1}$$

Pour POMP on a :

$$\begin{aligned} l_b &= 1 \\ l_e &= 3 \\ l_h &= 1 \\ l_i &= 1 \end{aligned}$$

donc cela arrive lorsque le nombre d'instructions conditionnées  $n_c \leq 5$ .

L'intérêt de cette technique est donc qu'elle est utilisable sans matériel supplémentaire lorsqu'on a à sa disposition un processeur permettant des branchements non retardés, tels que le MC88100 ou le SPARC (en jouant sur le bit d'annulation [Cyp90b]).

Nous verrons dans la section 8.3 l'algorithme capable de placer temporellement les instructions, basé sur une étude de graphes de dépendance. Un branchement non retardé rajouté par la méthode de conditionnement précédente a une signification très

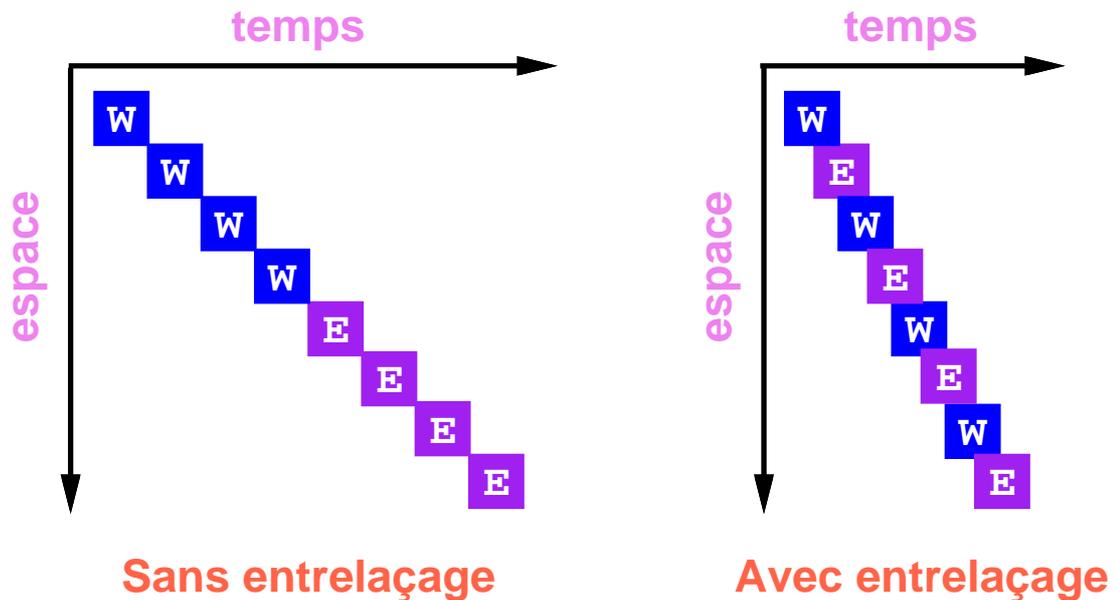


FIG. 7.7 - Estimation qualitative de la perte d'efficacité due au pipeline.

spéciale et ne peut être séparé des  $\ell_b$  instructions qui le suivent. Pour cela, il faut rajouter un suffixe au branchement qui le particularise au niveau de l'algorithme de placement. L'intérêt d'avoir  $\ell_b = 1$  apparaîtra dans la section 8.3.2.2 puisqu'un bloc d'une seule instruction ne peut être divisé par définition, que ce soit ou pas pour un problème de dépendance.

### 7.2.6.2 Entrelacement de blocs alternatifs terminaux

On peut tirer des méthodes précédentes un corollaire d'optimisation. Les processeurs modernes sont de plus en plus pipelinés et bien souvent un programme ne sature pas la bande passante de ce pipeline à cause des dépendances entre données : l'exécution d'une instruction est souvent en attente d'un résultat qui n'a pas encore fini d'être calculé. C'est ce qui explique en grande partie que la performance moyenne d'un processeur soit inférieure à la performance crête. Sur le schéma de gauche de la figure 7.7, cela se traduit par le fait que chaque instruction ne peut commencer avant que la précédente soit terminée (c'est la cas pire) : il y a alors des « trous » dans le pipeline !

On vient de considérer l'exécution d'un bloc conditionné, sans profiter du fait qu'on a souvent affaire avec des blocs associés `where/elsewhere` comme indiqué sur l'exemple.

Plutôt que de laisser chaque bloc `where` et `elsewhere` s'exécuter séquentiellement avec des instructions nulles insérées parfois dans le pipeline, on peut essayer d'entrelacer les instructions de ces deux blocs pour qu'il y ait le minimum d'instructions nulles envoyées par le contrôleur scalaire, à condition bien entendu qu'il n'y ait pas d'effet de bord affectant une autre collection que celle du `where`, donc en particulier ni effet de bord scalaire ni communication provoquant des dépendances. Dans ce cas on exploite au mieux la denrée rare d'une machine SIMD, à savoir le flot d'instructions généré par le séquenceur, ce qui est symbolisé par le schéma de droite de la figure 7.7.

---

```

collection pennou_skoulm;
pennou_skoulm int a,b;
pennou_skoulm float c,d,e;
...
void une_procedure()
{
    where(a >= b)
        c = d*e + 3;    /* Il n'y a pas d'effet de bord scalaire. */
    elsewhere
        c = d*13 + 7;  /* Là non plus. */
}

```

---

FIG. 7.8 - Exemple de conditionnement parallèle à compiler.

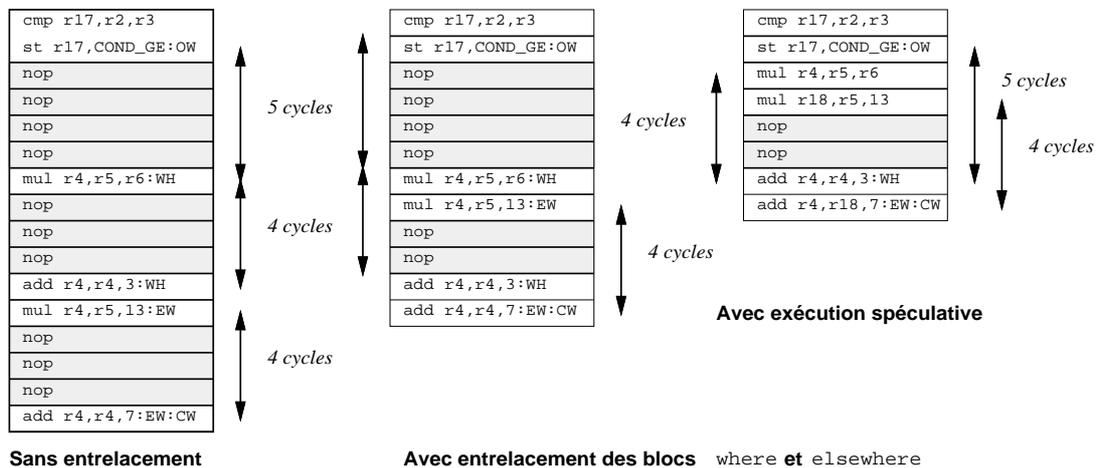


FIG. 7.9 - Programme compilé sans et avec entrelacement.

Si on regarde le petit exemple quantitatif de la figure 7.8 et que l'on compare les différentes manières de le compiler de la figure 7.9, on constate qu'effectivement il y a beaucoup de `nop`<sup>20</sup>. C'est d'ailleurs à peu près le même nombre d'instructions nulles qui seront insérées dans le code d'une machine séquentielle classique avec un compilateur de qualité moyenne.

On voit qu'il est très intéressant de remplir les `nop` du bloc `where` par les instructions du bloc `elsewhere` si cela est possible en respectant le graphe de dépendance temporelle (indiquée par les flèches verticales sur la figure). Le processeur n'exécutera

---

20. La section 8.3.2 décrit plus en détail l'algorithme de placement de ces `nop` permettant de synchroniser globalement le code de la machine tout en respectant les contraintes matérielles.

réellement que les instructions suffixées de `:WH` ou `:EW` selon la condition locale du test<sup>21</sup>. Remarquons que le problème est tout de même différent d’une exécution spéculative puisque si c’était le cas il faudrait assurer une non interférence entre registres — logicielle ou avec un renommage de registres matériel [OMMN90] — car chaque branche étant effectivement exécutée par le processeur.

En outre, on peut appliquer la technique d’exécution spéculative pour remplir encore plus le pipeline : si on a des instructions ne déclenchant pas de gros effets de bords<sup>22</sup>, autant commencer à exécuter des instructions qui seront probablement utiles plutôt que de laisser des instructions nulles « s’exécuter ». L’avantage de cette méthode dans notre cas précis est que la sélection des deux branches est faite par les suffixes et pas par des branchements. On économise donc un branchement par rapport à la version séquentielle. L’inconvénient est qu’il faut bien entendu faire une génération de code plus complexe.

Si on se contente de l’entrelacement des blocs alternatifs terminaux sans exécution spéculative, cela revient à faire de l’optimisation de très bas niveau qui est simple à réaliser. Comme on doit déjà manipuler les fichiers d’assembleur générés par le compilateur C pour ressynchroniser les codes (§ 8.3) et que le compilateur POMPC fait déjà une analyse de dépendance minimale pour optimiser la virtualisation, il suffit que celui-ci rajoute une directive dans le fichier C pour dire au programme de ressynchronisation qu’il a le droit d’entrelacer les blocs alternatifs terminaux.

Ce mode de compilation avec entrelacement des blocs alternatifs terminaux a comme avantage sur l’exécution spéculative faite par un compilateur, outre sa simplicité, le fait que l’exécution des instructions est contrôlée de manière externe alors que si on le fait de manière logicielle, il sera nécessaire de dupliquer des registres, autre denrée rare, pour ne pas à avoir de conflits lors de l’exécution des deux branches simultanément.

Le temps d’exécution de cet exemple passe successivement de 16 cycles à 12 et 8 cycles, sachant que la solution à 12 cycles (entrelacement simple) ne nécessite qu’un algorithme simple d’entrelacement. Si on a un seul `where/elsewhere` sans imbrication et que chaque bloc a un temps d’exécution équilibré, la perte d’efficacité intrinsèque du SIMD lors des branchements conditionnels terminaux peut être rendue négligeable par ces méthodes de compilation dans les cas courants. À titre de comparaison, l’exécution de ce code sur une machine MIMD demande 8 cycles (figure 7.10).

### 7.2.6.3 Utilisation d’un processeur élémentaire VLIW

À supposer qu’on s’éloigne un peu de l’architecture courante de POMP et qu’on s’intéresse à une machine où les PES sont VLIW au lieu d’être superpipelinés<sup>23</sup>, on peut transposer la dualité VLIW-pipeline des processeurs en dualité espace-temps pour le

21. Les suffixes sont présentés en annexe ?? mais on peut dévoiler déjà ceux qui nous intéressent :

- `:OW` ouvre un `where` comme indiqué sur la rangée 1 de la figure 7.1, page 137 ;
- `:WH` rend exécutable une instruction du `where`, c’est à dire lorsque `CNT = 0` ;
- `:EW` rend exécutable une instruction du `elsewhere`, donc lorsque `CNT = 1` ;
- `:CW` ferme un bloc conditionnel, conformément à la 4<sup>ème</sup> ligne de la figure 7.1.

22. Du style division par 0, par exemple.

23. En fait il est probable que les processeurs à venir possèdent ces 2 qualités simultanément, voire plus superscalaires que VLIW.

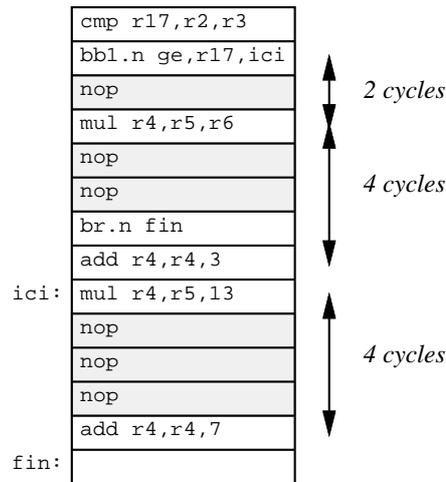


FIG. 7.10 - Le même programme compilé sur une machine MIMD.

contrôle de flot SIMD et l'entrelacement des instructions appartenant à des blocs **where** et **elsewhere**: plutôt qu'entrelacer les instructions dans le temps, on va les entrelacer dans l'espace, sur les unités d'exécution concurrentes présentes dans les processeurs VLIW dans la version non entrelacée.

Si on rajoute le système de contrôle d'activité externe et qu'on peut exécuter au moins 2 instructions à chaque cycle, on pourra simultanément profiter du contrôle de flot **where** et **elsewhere** alors que le processeur VLIW utilisé normalement ne peut exécuter qu'un seul branchement à chaque cycle: on a gagné là aussi en contrôlabilité dans le cadre d'une optique à parallélisme de données.

En général, les processeurs VLIW ne sont pas utilisés à 100%, donc la mise en parallèle des instructions appartenant aux 2 blocs alternatifs terminaux ne sera pas trop pénalisante (voire pas du tout) même si on n'exécute utilement que la moitié de ces instructions (soit le **where**, soit le **elsewhere**), comme le montre la traduction du petit programme POMPC de la figure 7.8 en pseudo-assembleur<sup>24</sup> indiqué sur la figure 7.11, sans ou avec exécution spéculative dans le cas d'un bon compilateur, qui s'exécutent respectivement en 6 et 4 cycles.

Le problème revient à exécuter deux programmes différents (ou plus dans le cas de **where** imbriqués ou de **switchwhere**) en même temps sur une machine VLIW [BQW91], à la différence près qu'ici les programmes sont très particuliers car très peu désynchronisés entre eux à cause du modèle de programmation. La compilation en est grandement simplifiée aussi grâce à la limitation des effets de bord et à la localité accrue des dépendances entre données imposées par le modèle.

Mais la mise en œuvre de cette technique peut être impossible si on ne peut pas influencer extérieurement l'exécution des deux branches, car souvent ces processeurs ont des mémoires caches intégrées et on est à la limite du SPMD. Néanmoins le dernier processeur de la famille CLIPPER [SM91, Int91d] peut se plier à une telle perversion et en général tous les processeurs composés aussi de plusieurs circuits intégrés comme

24. On suppose qu'on a un processeur qui est un MC88100 imaginaire capable d'exécuter deux instructions par cycles et dont la multiplication n'a plus que 2 cycles de latence, ce qui est raisonnable si on considère qu'en contrepartie du nombre d'UAL, celles-ci sont moins pipelinées.

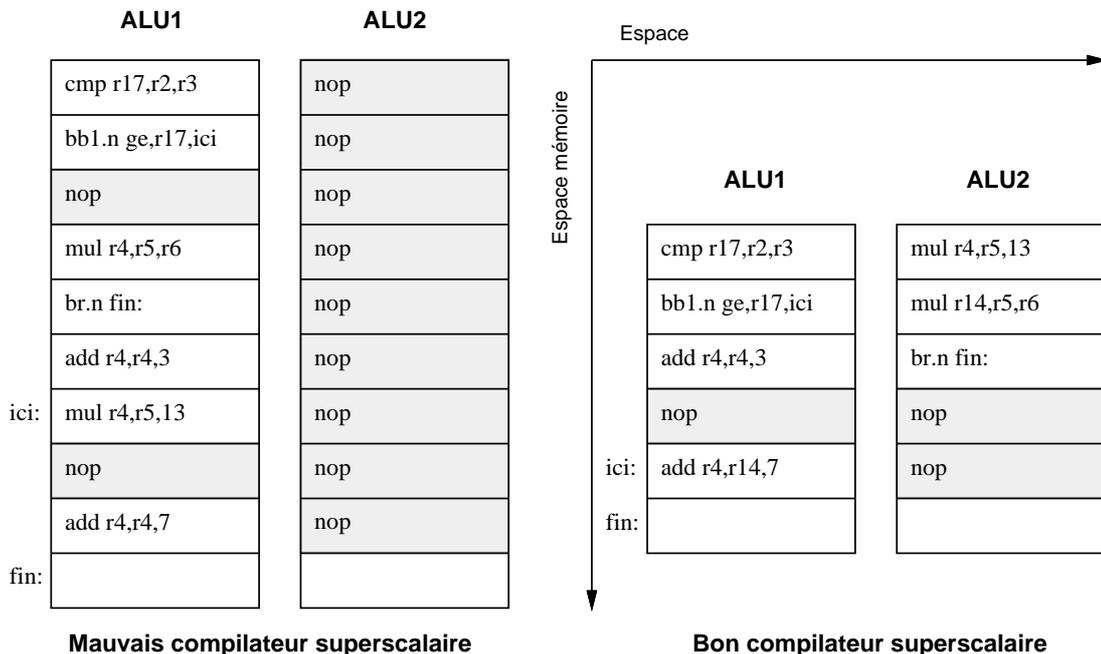


FIG. 7.11 - Exécution d'un test sur une machine non-parallèle VLIW.

certaines versions des HP-PA ou RS/6000 [IBM90], car on peut accéder aux signaux de contrôle fins et vraiment les utiliser comme PES de machine SIMD.

Il faut donc que ce soit le compilateur qui analyse le graphe de dépendance et les effets de bord de chaque branche pour que la sémantique d'exécution soit respectée : tout doit se passer comme si seulement une branche du test avait été exécutée. La manière de compiler est néanmoins la même que pour compiler efficacement un test conditionnel sur une machine VLIW [AN90]. C'est l'approche de l'exécution spéculative : il vaut mieux calculer le plus tôt possible tout ce que l'on peut en parallèle, pourvu que cela ne pénalise pas l'exécution du programme, quitte à s'apercevoir *a fortiori* qu'on a fait du travail inutile.

Si on rajoute un mécanisme externe pour choisir d'exécuter ou non certaines instructions, l'exécution du même programme aura la forme de la figure 7.12 et s'exécutera en 7 cycles. Le fait que ce soit plus lent que même la méthode MIMD la plus simple s'explique par la forte latence du contrôle externe qui nécessite le temps d'une écriture et d'une lecture sur le bus externe du processeur : ramené au nombre d'instructions exécutées,  $l_e + l_h + l_i$  est plus important puisque, du fait du VLIW, plusieurs instructions sont exécutées par cycle.

Le facteur de mérite exposé en § 7.2.6.1 s'applique toujours, si on ne considère non plus  $n_c$  comme étant le nombre d'instructions d'un bloc conditionné mais comme étant le nombre de lignes d'instructions, au sens VLIW, du bloc et qu'on ne transforme pas le code. Il est difficile donc de raisonner en nombre d'instructions efficaces du bloc puisque chaque ligne peut contenir de 0 à  $v$  instructions autres que des `nop`, où  $v$  est le nombre d'instructions exécutables simultanément sur le processeur.

Le contrôle par branchement conditionnel non retardé s'applique toujours à une ligne d'instructions alors que la méthode du contrôle externe concerne les instructions

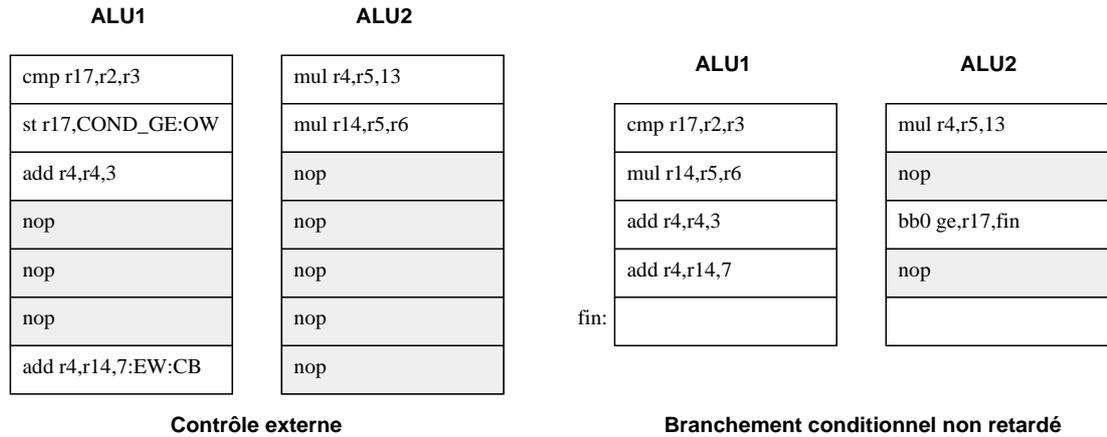


FIG. 7.12 - Exécution d'un test parallèle sur une machine SIMD à PES VLIW.

individuelles. Une solution simple de transformation du code est de faire remonter des instructions du `elsewhere` au niveau du `where` à la place des `nop` dans la mesure du possible. Cette transformation ne nécessite qu'une vision locale des dépendances et des contraintes temporelles du matériel mais permet néanmoins de comprimer l'exécution à 4 cycles (dessin de droite de la figure 7.12), comme dans le cas de l'exécution sur une machine MIMD. Dans ce cas, il faut tenir compte de cette modification du code au niveau de l'algorithme de placement temporel (§ 8.3) par l'intermédiaire de suffixes purement syntaxiques.

Pour conclure dans le cas d'une machine SIMD construite à base de PES VLIW, la méthode du contrôle externe est handicapée par la latence plus importante ramenée au nombre d'instructions exécutées mais permet de contrôler plus finement l'exécution des instructions alors que la méthode du branchement conditionnel non retardé peut être efficace si on la couple avec un système d'exécution spéculative pour compenser le fait que le contrôle se fait de manière plus grossière, au niveau ligne d'instructions.

### 7.3 Conclusion

L'introduction des nouvelles méthodes de compilation et d'optimisation du contrôle de flot SIMD que nous avons exposées permet une compilation plus efficace des parties parallèles contenant du code conditionnel.

Dans le cas des blocs alternatifs terminaux qui constituent une partie importante des blocs alternatifs parallèles rencontrés, les performances peuvent se rapprocher des machines MIMD par une meilleure utilisation des « trous » dans le pipeline. Cela est dû au fait que le contrôle de l'activité rajoute une contrôlabilité se rapprochant de l'exécution d'un débranchement en même temps que chaque instruction et donc augmente l'efficacité du processeur. Un contrôle souple de l'activité permet d'envisager un entrelacement des blocs alternatifs terminaux bénéfique pour exploiter les performances d'un processeur pipeliné sans avoir à dépenser d'instruction supplémentaire pour cet entrelacement ni avoir à faire du renommage de registres qui constituent une denrée assez rare.

Dans le cas de petits blocs conditionnés, la méthode du branchement non retardé

est plus efficace que le contrôle externe et est donc préférable aux autres méthodes.

Enfin, la méthode du compteur d'activité fournit une alternative intéressante à celle de la pile de bits d'activité dans le cas des machines SIMD à gros grain mais aussi pour la compilation efficace de programmes à parallélisme de données pour les machines MIMD, dans la mesure où les compteurs logiciels sont plus simples à gérer et possèdent une meilleure localité dans les caches et un grain plus adaptés aux processeurs.

En ce qui concerne les machines SIMD qui ne sont pas à gros grain, une comparaison avec les méthodes avec pile d'activité contrôlée localement ou globalement permet de trouver un compromis entre le gain de temps, d'espace, la complexité de la machine et du compilateur, induisant ainsi une adaptation à une plus grande classe de machines SIMD.

Ces méthodes de contrôle SIMD permettent donc d'exploiter pleinement les avantages actuels du SIMD, à savoir principalement une machine plus dense en MFLOPS/dm<sup>3</sup> et une factorisation du code, tout en gardant une contrôlabilité raisonnable par rapport aux machines MIMD pour les applications à parallélisme de données.

Maintenant que l'on a étudié ce qui était nécessaire à la gestion du contrôle de flot SIMD, on va pouvoir décrire l'électronique et l'assembleur qui le gère dans les sections 10.1.2.2 et ?? respectivement, après avoir abordé quelques problèmes de génération de code.

# Chapitre 8

## La génération du code



**A**VOIR une machine parallèle pose des problèmes de programmation en amont au niveau du modèle de programmation et du langage bien entendu, mais aussi en aval lorsqu'il faut traduire ces concepts en instructions directement exécutables par la machine. La génération de code pour toute machine programmable est une nécessité, même si elle a tendance à être, hélas, un peu négligée par les architectes<sup>1</sup>. C'est d'autant plus crucial que pour une machine un peu baroque comme POMP il faut concilier les caractéristiques suivantes :

- parallélisme SIMD ;
- synchronisation VLIW du processeur scalaire et des processeurs parallèles ;
- virtualisation offerte par POMPC réalisée de manière efficace ;
- soucis d'économie du travail à accomplir et pragmatisme ;
- récupération maximale de l'environnement de programmation existant autour du processeur ;
- possibilité de pouvoir faire tourner un programme prévu pour POMP sur une autre machine, séquentielle ou parallèle, que ce soit pour mettre au point les programmes ou bien les tester en grandeur nature, voire fournir un nouvel environnement de programmation à une autre machine ;
- offrir un environnement de programmation qui ne fait pas fuir en courant un utilisateur normal.

Il s'agit dans la suite, non pas de décrire le compilateur POMPC pour POMP qui est plus le travail de Nicolas PARIS mais la chaîne de développement et les principes utilisés en aval.

## 8.1 Virtualisation

Comme beaucoup de langages parallèles, POMPC offre la possibilité de virtualiser le parallélisme et de le rendre indépendamment de la taille du parallélisme offert par la machine cible.

Sauf cas particulier comme les machines travaillant directement de mémoire à mémoire comme les STAR 100 [HT72] ou CYBER 205 [Lin82] où la taille des vecteurs importe peu<sup>2</sup>, on est obligé de tronçonner les tableaux parallèles sur lesquels on travaille en entités de taille compatible avec le matériel.

Sur les machines vectorielles de type CRAY-1 [Rus78] qui introduit les registres vectoriels, une opération vectorielle porte sur ces registres et donc est limitée par la taille de ceux-ci. Il faut par conséquent découper les opérations vectorielles pour qu'elles aient la taille des registres vectoriels, découpage qu'on trouve sous le nom de *strip mining*.

---

1. On ne rappellera jamais assez qu'un ordinateur est à la fois un support matériel et un environnement de programmation et donc que l'architecture des ordinateurs devrait englober les deux...

2. « Peu » et non « pas » puisque la taille des vecteurs est en fait limité par la taille du registre codant cette taille, à savoir 64K éléments.

Sur les machines parallèles, les opérations vectorielles sont découpées en instructions qui calculent un élément par processeur physique. Ainsi une suite d'opérations parallèle

$$(A, B, C)$$

sera traduite par exemple en

$$(A_1, \dots, A_n, B_1, \dots, B_n, C_1, \dots, C_n)$$

qui sont des opérations parallèles élémentaires. Mais si  $(A, B, C)$  sont des instructions indépendantes en terme de communications, pourquoi ne pas choisir d'exécuter

$$(A_1, B_1, C_1, \dots, A_n, B_n, C_n)$$

à la place? Cela dépend de la machine cible comme nous allons le montrer.

### 8.1.1 Virtualisation en largeur d'abord

Chaque instruction est exécutée sur tous les processeurs virtuels d'abord, c'est-à-dire sur tous les éléments d'une expression vectorielle, et l'instruction parallèle suivante ne commence pas avant que l'instruction parallèle précédente ait été exécutée pour tous les processeurs virtuels.

Si les processeurs n'ont pratiquement pas de registres adaptés à la taille des problèmes (typiquement les machines de type CM-2 ou MPP qui n'ont que de petits processeurs), les registres serviront tout juste à exécuter chaque instruction élémentaire et on sera obligé de repasser par la mémoire. Dans ce cas autant utiliser la première solution qui est plus simple à mettre en œuvre. On constate qu'on est alors limité par la bande passante mémoire, comme le sont aussi les machines de type CYBER 205 (voir § 5.1.1.4). En outre, certains facteurs extérieurs aux processeurs peuvent aussi faire choisir cette méthode. Par exemple si le débit d'instructions parallèles est insuffisant, on a intérêt à répéter chaque instruction élémentaire  $n$  fois de suite pour limiter la bande passante des instructions. Le fait d'avoir des processeurs à grain fin peut accentuer l'intérêt de la méthode dans la mesure où une instruction parallèle à gros grain (par exemple une addition flottante) doit être exécutée en plusieurs instructions à grain fin répétées. C'est l'approche prise sur la CM-2 et la WAVE TRACER<sup>3</sup>.

Ce modèle de génération de code se prête bien lorsqu'on est dans certaines conditions :

- si on a intérêt à virtualiser au maximum chaque instruction pour ne pas avoir de goulet d'étranglement au niveau du processeur scalaire comme sur la CM-2 ;
- lorsqu'on ne veut décidément pas faire du logiciel : non seulement on n'a pas à dérouler des boucles de plusieurs instructions mais en plus on n'a pas à faire d'analyse de graphe de dépendances entre les instructions puisqu'il n'y a qu'une instruction à chaque fois ;

---

<sup>3</sup>. La MP-1 ne virtualisant pas, du moins en MPL, MASPAR se facilite la vie... Mais pas celle du programmeur !

- quand la machine n'a pas de registres vectoriels comme sur le CYBER 205 ou la CM-2. Dans ce cas, tout ce fait en mémoire et la notion de nombre de processeurs<sup>4</sup> perd de son sens : on est souvent limité par le débit mémoire, donc par le nombre de pattes dédiées au bus de données et finalement au nombre de pattes qu'on peut mettre par carte [Dou89].

Dans le cas de machines parallèles, une couche de logiciel (comme par exemple la machine virtuelle à pile au dessus de la machine DAP [Fla90]) ou du matériel (séquenceur pour les machines CM-2 et WAVETRACER, processeur d'instructions pour OPSILA) sont nécessaires pour découper une instruction vectorielle dont la taille dépasse le nombre de processeurs physiques en autant d'instructions compatibles avec la taille de la machine.

L'inconvénient est que ce type de génération engendre 3 mouvements de données par opération élémentaire : aller chercher les 2 opérandes en mémoire et réécrire le résultat en mémoire. On a souvent affaire à un gâchis conséquent de bande passante mémoire par rapport au cas où on peut utiliser des registres en mémoire. On optimise encore une fois de plus le cas pire aux dépens du cas courant.

Néanmoins, comme on l'a vu, dans certains cas cette approche est justifiée. Dans la première étude de POMP [Dou89] les PEs étaient reliés à la mémoire à travers des bus de plusieurs centaines de bits, rendant l'utilisation de registres superflue. Cela était possible à condition d'intégrer mémoire et PE sur le même circuit intégré, mais ce n'est pas commercialement réaliste pour une petite équipe de chercheurs sans avoir de support structurel industriel.

### 8.1.2 Virtualisation en profondeur d'abord

Par contre, si on a une machine à gros grain possédant des registres, il est fort probable qu'on choisira la 2<sup>ème</sup> méthode de virtualisation. En effet, on pourra passer des données d'une instruction à l'autre bien plus rapidement à travers les registres de la machine qu'à travers la mémoire. En plus, on pourra dérouler la boucle sur  $n$  afin de supprimer des dépendances temporelles trop rapprochées entre instructions qui pourraient limiter les performances. C'est cette approche qui a été choisie lorsqu'on génère du code pour la machine POMP. Cette méthode est aussi celle utilisée sur les machines vectorielles modernes et le déroulage des boucles revient en fait à chaîner plusieurs pipelines opératoires entre eux [KT80] pour reprendre une image du monde des ordinateurs vectoriels. En parallélisation cette opération correspond à une aggrégation de boucles.

À ce niveau, le cas d'OPSILA est un peu particulier : normalement la virtualisation a lieu suivant la première méthode (utilisation des « instructions vectorielles prédéfinies ») mais on peut aussi programmer plus finement la boucle de virtualisation par l'intermédiaire des « instructions vectorielles définissables ».

Supposons que l'on veuille compiler le programme de la figure 8.1 qui calcule une fractale. On constate que pour chaque pixel toutes les variables déclarées dans le `dowhere` sont indépendante et ont une durée de vie limitée. Si on génère le code en profondeur d'abord la boucle de virtualisation peut s'étendre sur tout l'intérieur du

---

4. Correspondant dans le cas d'une machine vectorielle pipelinée à quelque chose entre le nombre d'éléments des registres vectoriels et le nombre d'étages de pipeline des opérateurs.

---

```

collection image int julia(collection image double x,
                           collection image double y)
{
    collection image int k;
    collection image double norme, grand, x0, y0;

    x0 = x;
    y0 = y;
    k = NB_ITER_MAX;
    dowhere
    {
        collection image double x2, y2, xy;

        x2 = x0*x0;    /* On calcule  $z_{n+1} = z_n^2 + c$  */
        y2 = y0*y0;
        xy = x0*y0;
        norme = x2 + y2;
        x0 = x2 - y2 + cr;
        y0 = xy + xy + ci;
    }
    whilesomewhere ((norme < grand) && --k);
    return k;
}

```

---

FIG. 8.1 - Programme de calcul d'un ensemble de JULIA en POMPC.

bloc `dowhere`<sup>5</sup>. Par conséquent les variables parallèles déclarées dans le `dowhere` peuvent déchoir en variables de type registre sur les processeurs physiques. Cela a plusieurs effets bénéfiques :

- les calculs sont faits en registres plutôt qu'en mémoire, ce qui augmente considérablement la vitesse de calcul ;
- la gestion de boucle virtuelle est faite au niveau d'un bloc d'instructions plutôt que répétée au niveau de chaque instruction. Cette diminution du nombre de boucles permet déjà d'augmenter l'efficacité par le code économisé ;
- la taille des blocs d'instructions dans les boucles augmente et le pipeline des instructions de ces blocs plus grands est plus efficace, ce qui accélère encore l'exécution du programme ;
- enfin on fait économie d'autant de variables parallèles, ce qui entraîne un gain d'espace mémoire conséquent, ce qui n'est pas un des moindres avantages lorsqu'on travaille sur des variables très parallèles. Une variable parallèle de  $\mathcal{V}$  éléments n'a plus besoin que de  $N$  éléments, un par processeur. Une fois que cette

---

<sup>5</sup>. Pas l'extérieur car le `whilesomewhere` a un effet de bord scalaire sur le contrôle de flot limitant l'extension de la boucle de virtualisation.

transformation est faite, le compilateur final peut encore classiquement projeter efficacement ces registres sur les registres de la machine s'il y en a qui ont des domaines de vie qui ne se recouvrent pas et donc diminuer le nombre de registres nécessaires<sup>6</sup>.

Au niveau du code généré pour les processeurs parallèle, la déclaration des variables sera donc

```
register double x2,y2,xy;
```

ce qui apporte bien les bénéfices annoncés ci-dessus.

## 8.2 L'infrastructure de génération du code

L'ensemble des programmes nécessaires pour transformer un programme écrit en POMPC en un fichier d'instructions exécutable sur une machine cible peut être divisé en 2 parties : la partie amont qui gère le langage POMPC en particulier et la partie aval qui s'occupe plus précisément des spécificités liées aux machines. L'infrastructure générale est résumée sur la figure 8.2.

Tout programme POMPC est découpé par un analyseur syntaxique et grammatical (basé sur `lex` et `yacc` d'UNIX). En plus des vérifications du langage C, un vérificateur de type contrôle les constructions parallèles et passe la main au générateur de code.

Celui-ci doit prendre en compte les différences liées aux architectures cibles, à savoir principalement scalaire, MIMD ou SIMD, et les différences liées à la présence ou pas de virtualisation toute faite.

À l'heure actuelle les machines programmables en POMPC sont :

- Connection Machine 2, avec le générateur de code C\* de Nicolas PARIS ;
- les machines scalaires UNIX avec du code C intermédiaire (Nicolas PARIS) ;
- MASPAR MP-1, à travers le langage MPL (Nicolas PARIS) ;
- l'iPSC/860 avec le générateur de code C pour machine scalaire modifié par Thierry PORCHER. Ce générateur de code peut s'adapter facilement à la plupart des machines MIMD à mémoire distribuée, étant donné qu'à chaque fois le travail consiste principalement en l'adaptation des routines de communication ;
- en développement une adaptation du générateur de code pour iPSC/860 à la machine ARMEN [FGP91] du LIBr.

On remarque que pour aucune des machines cibles on a eu besoin de faire appel au langage de bas niveau de la machine. Le fait qu'elles acceptent toutes un langage basé sur C, vu encore une fois comme un langage d'assemblage portable et de haut niveau, nous facilite la vie<sup>7</sup> et permet d'adapter le compilateur POMPC vers diverses machines assez simplement.

---

6. C'est ce qui est fait avec le compilateur final que nous utilisons [OAS91].

7. C'est par contre un problème réel pour les traducteurs de FORTRAN 90 vers autre chose puisque FORTRAN 90, outre le parallélisme, permet bien plus de choses qu'on retrouve dans le C. C'est pour cela qu'il n'y a pas de traducteur complet FORTRAN 90 vers FORTRAN 77 mais vers le langage C, comme on l'a vu en § 4.1.1.4.

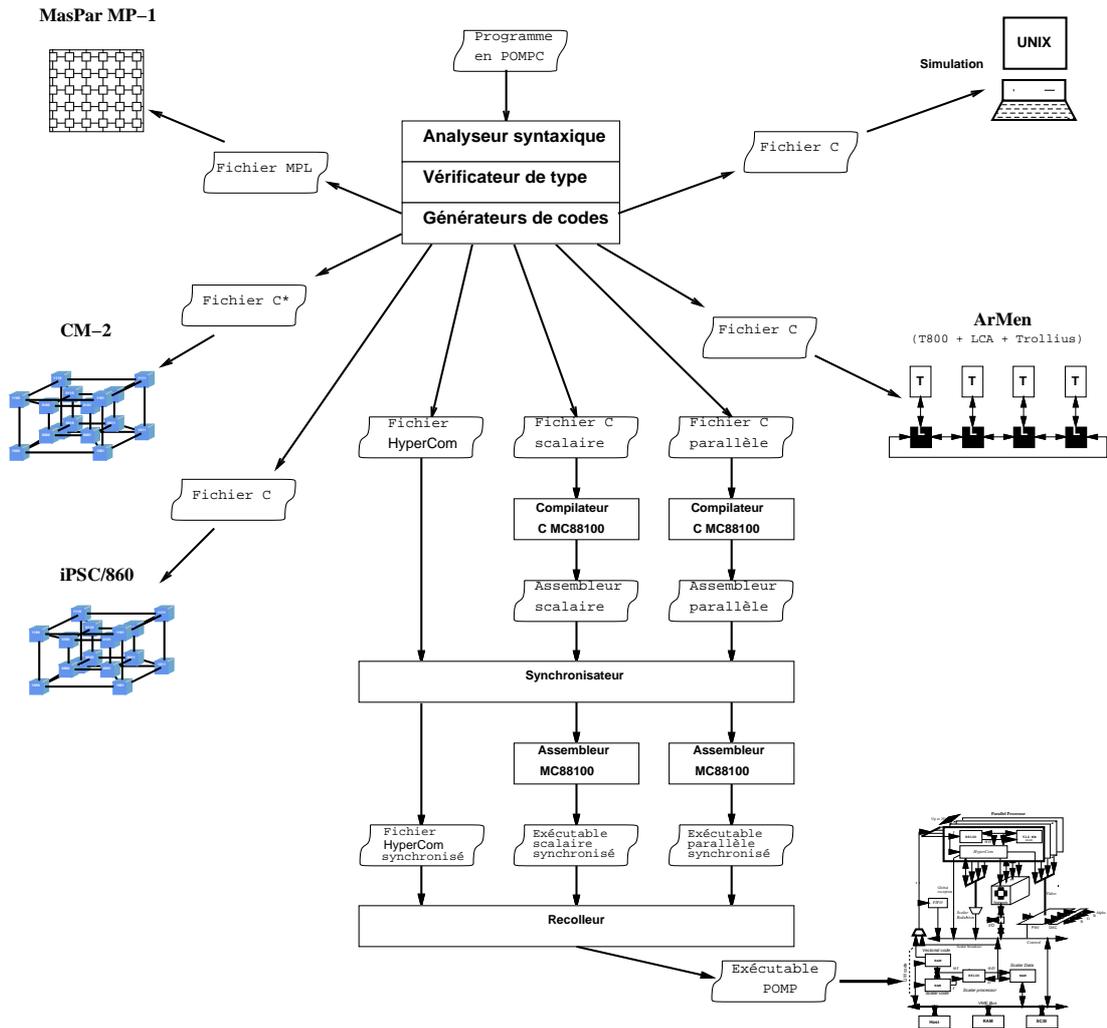


FIG. 8.2 - Infrastructure générale de la chaîne de génération de code à partir de POMPC.

La génération de code pour machines multiprocesseurs, avec le développement actuel des stations de travail de ce type, est envisagée. Elle sera assez simple à faire<sup>8</sup>, basée principalement sur les processus légers d'UNIX SVR4 ou SOLARIS 2.0 [PKB<sup>+</sup>91], car la version actuelle de POMPC pour station de travail peut déjà travailler en multiprocesseur avec segment de mémoire partagée. Le passage de multiprocesseur en multiprocesseur se fait à travers les processus légers.

Pour l'instant, la partie génération de code pour la machine POMP n'est pas terminée puisque la machine ne l'est pas non plus, sachant que la partie logicielle à terminer pour POMP est faible (compilateur et assembleur MC88100 acheté) comparée à la partie amont de l'environnement POMPC. Actuellement, les essais sur POMP se font donc en langage C.

8. La différence principale étant qu'une variable globale de type physique devient un élément de tableau au lieu d'être un scalaire par processeur afin qu'il n'y ait pas de conflit d'accès.

## 8.3 Les problèmes de synchronisation

Ils interviennent à deux niveaux dans la machine POMP. D'une part il faut assurer que les processeurs de la machine restent synchrones, même s'ils ne sont pas fait pour cette tâche et qu'ils sont plongés en milieu hostile (interruptions asynchrones). D'autre part il faut assurer le synchronisme entre le processeur scalaire et les processeurs parallèles puisque c'est celui-là qui nourrit ceux-ci.

### 8.3.1 Synchronisation scalaire-parallèle

Afin de pouvoir utiliser un compilateur C standard d'une part sur le code scalaire et d'autre par sur le code parallèle, il est nécessaire d'insérer des points de repère dans les programmes C générés par le compilateur POMPC que l'on pourra retrouver à coup sûr dans les fichiers assembleur après compilation. Ainsi, en appariant les repères entre le fichier assembleur scalaire, parallèle et le fichier **HyperCom** on sera capable de composer le code final de la machine.

Il y a 2 méthodes simples pour faire cela :

- la première consiste à utiliser des appels à des fonctions bidon (`pc_synchro_1`, `pc_synchro_2`,...) que l'on retrouvera dans le programme après assemblage. Un des avantages est que la méthode semble assez robuste (elle résiste aux déroulements de boucle effectués par le compilateur C que nous possédons) et reste portable. L'inconvénient mineur est qu'il faut juste faire attention dans le cas des branchements retardés au fait que la marque peut concerner l'instruction suivant l'appel à la fonction de marquage. Puisqu'on peut se contenter de pseudo-fonctions de marquages locales au sens du C, leur énumération est locale à un fichier source ou objet ;
- la seconde est d'utiliser les instructions `asm` ou `__asm` qui permettent de rajouter ce que l'on veut *texto* dans un fichier assembleur à partir d'un fichier C. La méthode est moins portable dans la mesure où les instructions de type `asm` ne font pas partie du langage<sup>9</sup>. En plus la méthode est moins robuste puisque les instructions `asm` résistent assez mal au déroulement de boucle<sup>10</sup>, même si dans notre cas il est hors de question d'utiliser cette option du compilateur qui modifie trop le code, vue l'utilisation qu'on fait du compilateur.

Finalement, la méthode des appels de fonction bidon semble être le meilleur choix. À chaque débranchement scalaire possible le compilateur POMPC rajoute juste avant et juste après un appel de fonction de marquage dans chaque fichier C. Il en est fait de même pour les fonctions à effet de bord de type communication, diffusion scalaire ou calcul de condition globale par exemple.

Notons que le fichier parallèle ne contient plus d'instruction de contrôle de flot comme il y en a dans le fichier scalaire mais contient tout de même les début et fin de fonction qui permettent de récupérer le mécanisme de pile du compilateur C. Le

---

9. Le mot `asm` est juste « réservé » [KR88, page 192].

10. On peut se demander où est le bug du compilateur : le fait qu'il essaye de dérouler les boucles en perdant des instructions ou bien en déroulant le code même s'il ne maîtrise pas les effets de bord du code contenu dans les instructions `asm`...

compilateur voit donc le fichier C parallèle comme une série de fonctions contenant chacune des fonctions de marquage qui permettront de refaire le lien avec le contrôle de flot scalaire.

À travers ce mécanisme de fonctions de marquage, en rajoutant des lettres spécifiques aux noms de fonctions, on peut faire passer d'autres informations utiles en plus des informations de synchronisation scalaire-parallèle en vue de faire des optimisations. Ainsi, les marquages début et fin de bloc conditionné, appartenance à un `where` ou à un `elsewhere` permet au synchroniseur de rajouter soit les suffixes `HyperCom` correspondants soit de faire de l'entrelaçage de blocs conditionnés terminaux (§ 7.2.6.2), soit encore faire du conditionnement parallèle à l'aide de débranchements non retardés si les blocs conditionnés sont suffisamment petits (§ 7.2.6.1).

### 8.3.2 Synchronisation SIMD

Même dans le cas d'un ordinateur sans mémoire cache, il y a plusieurs raisons qui font qu'un processeur peut se désynchroniser par rapport aux autres mais elles mènent finalement toutes au même résultat : une instruction ne peut pas être terminée ou commencée à temps pour des raisons de conflits sur des ressources demandées simultanément par au moins deux instructions. On s'éloigne ainsi de la belle image d'ÉPINAL du processeur RISC primitif où chaque instruction est exécutée en 1 cycle quel que soit le contexte.

De manière plus précise dans le `MC88100`, comme dans tous les processeurs de manière générale mais parfois sous une autre forme, on trouve les goulets d'étranglement suivants<sup>11</sup> :

- la dépendance entre instructions partageant des registres identiques peut entraîner des attentes lorsqu'une instruction demande un résultat non encore terminé. Cette dépendance est gérée sur une ardoise centralisée (*scoreboard*) qui prend note de l'utilisation courante de chaque registre du processeur ;
- bus de données de 32 bits seulement alors qu'on peut sensément commencer une instruction d'accès à la mémoire sur 64 bits par cycle d'horloge ;
- le bus d'accès aux registres n'est que de 32 bits alors qu'on peut lancer une instruction de 64 bits (flottant double précision) par cycle en théorie ;
- les instructions n'ayant pas toutes le même temps d'exécution, plusieurs peuvent désirer écrire un résultat dans des registres différents mais en même temps, à travers un bus interne unique alloué à 5 dispositifs par un mécanisme de priorité statique strictement croissante [MOT88a, page 7-28].

La solution de facilité est de faire fonctionner le `MC88100` en mode non pipeliné mais alors les performances sont dégradées et de toute manière il restera la synchronisation scalaire-parallèle à faire.

---

11. La liste peut faire peur mais le manuel [MOT88a] est plutôt clair là dessus et la situation bien expliquée. Hélas, ce n'est pas le cas de beaucoup d'autres processeurs, comme le i860 [INT89b] par exemple...

La bonne solution est de résoudre statiquement les dépendances et de rajouter les instructions nulles là où besoin est. Un compilateur performant est de toute manière obligé de faire cette analyse afin de minimiser le temps passé à l'exécution dans les attentes de résolution dynamique de dépendances entre instructions [RF72, FR72, KMC72, AU77, GM86].

Lorsque le processeur cible possède des mécanismes de résolution des dépendances entre les instructions, les compilateurs du commerce fournissent un programme en langage d'assemblage où la gestion des dépendances n'est pas statique pour deux raisons principales :

- la gestion des dépendances peut être compliquée dans le cas de boucles et de débranchements complexes, appels de fonction, etc. car le compilateur ne connaîtra pas *a priori* le passé de certains blocs d'instructions, ce qui peut avoir des conséquences fâcheuses. Telle fonction se servira d'un argument alors qu'il n'aura pas fini d'être calculé... Pour cela certains processeurs offrent le mécanisme d'ardoise qui évite d'avoir à insérer des temps d'attente préventifs de manière pessimiste<sup>12</sup>.
- la taille du code est réduite si on ne rajoute pas tous les `nop` inutiles et qu'on laisse le système de l'ardoise où sont notées les dépendances à l'exécution les rajouter dynamiquement ;

En ce qui nous concerne, il faut développer un système de résolution statique des conflits et dépendances pour POMP. Refaire un logiciel spécialisé nous semble dommage pour 2 raisons principales

- 1° c'est compliqué à faire et peut faire (et a déjà fait) l'objet de plusieurs thèses rien que sur le sujet [FR72, Fis81, DLSM81, ?, GM86] ;
- 2° le compilateur que nous avons acheté [OAS91] fait déjà l'ordonnancement des instructions pour une utilisation efficace du MC88100, le déroulage des boucles<sup>13</sup> et ce serait dommage de ne pas pouvoir récupérer ces optimisations toutes faites, même si elles ne sont pas optimales par rapport à l'état de l'art du domaine.

L'idée de base va être de récupérer le fichier d'assembleur généré par le compilateur et de résoudre les conflits et les dépendances sans modifier l'ordre d'exécution des instructions. Ceci permet de conserver le travail de la phase d'optimisation de l'ordonnancement effectuée par le compilateur. L'avantage d'avoir un processeur comme le MC88100 est que les instructions sont RISC, donc très simples à analyser, et que leurs effets de bord sont précis et bien définis.

Nous allons maintenant exposer un algorithme de placement temporel après avoir rappelé brièvement quelques notions sur les graphes de dépendances [AU77], en utilisant des notations se rapprochant de [dD91].

---

12. Néanmoins on peut exhiber des cas où la gestion statique des dépendances est plus efficace. Il suffit par exemple que le mécanisme de priorité vu ci-avant provoque une avalanche de dépendances qui n'aurait pas eu lieu si c'était une autre instruction qui avait eu la priorité. Le compilateur, s'il ne peut pas toujours « voir » dans le passé à cause de tous les débranchements et les appels de fonction a néanmoins beaucoup plus de talents pour « voir » dans le futur !

13. Le déroulage est choisi à la compilation comme étant d'un facteur 4 ou 8. Malheureusement, outre sa mise en œuvre primitive, le compilateur ne voit pas toujours certains effets de bord de quelques boucles un peu compliquées écrites en C...

### 8.3.2.1 Graphes de dépendances et exécution

Afin d'analyser un programme, il faut représenter la manière dont les instructions vont s'exécuter. La liste des instructions exécutées *a posteriori* au court du temps est la *trace* du programme.

Malheureusement, dans bien des cas avoir la trace d'un programme revient à l'avoir déjà exécuté afin de déterminer pour chaque test quel branche a été prise, combien de fois a été exécutée chaque boucle, à quel instant, etc. Si on peut deviner dans les cas simples la trace, dans les cas plus complexes où le déroulement du programme dépend de données extérieures (méthodes itératives s'arrêtant par un test de convergence par exemple), la solution est impossible à déterminer exactement sans exécuter le programme avec tous les jeux de données possibles<sup>14</sup>.

C'est pour cela qu'on se contente du *graphe de contrôle de flot*  $\mathcal{G}_c$  qui est construit en disant que chaque branche de test peut être exécutée, quels que soient les antécédents des tests, ou tout au moins si on ne peut pas le déterminer symboliquement dans le cas d'un système de calcul de graphe évolué. Chaque instruction  $I$  du programme est représentée par un nœud relié par un arc à chaque instruction  $J$  (arc noté dans la suite  $I \xrightarrow[\mathcal{G}_c]{+} J$ ) pouvant s'exécuter juste après dans le programme, à savoir 1 arc en temps normal, 2 pour les débranchements, 0 pour un retour de fonction et  $\infty$  pour les déréréférences à des fonctions quelconques. Très probablement, en supposant que tous les débranchements peuvent être pris ou pas on rajoute des arcs et donc des chemins qui n'existent pas dans l'exécution réelle du programme. Mais cette approximation de la trace est compensée par la très forte simplification de la représentation. Une instruction  $J$  peut s'exécuter postérieurement à une instruction  $I$  s'il existe un chemin d'instructions les reliant dans  $\mathcal{G}_c$ , existence que l'on notera par  $I \xrightarrow[\mathcal{G}_c]{+} J$ . Par abus de langage, cette dernière notation pourra aussi signifier tout chemin de  $I$  à  $J$ , voire même tout simplement un booléen indiquant son existence selon le contexte, et durée( $I \xrightarrow[\mathcal{G}_c]{+} J$ ) sera le plus court temps d'exécution pour aller de  $I$  à  $J$ .

Clairement, dans le cas des déréréférences à des fonctions il faut faire une simplification. Afin d'éviter de faire une analyse interprocédurale<sup>15</sup> on va supposer que lorsqu'une fonction est appelée, il n'y a aucune ressource encore réservée au moment où la 1<sup>ère</sup> instruction de la fonction appelée s'exécute et que lorsqu'on revient d'une fonction dans le programme appelant le pipeline soit vide aussi de réservation de ressource. Comme on ne veut pas faire de la parallélisation interprocédurale [TIF86], ce n'est pas une contrainte très forte. Ainsi, en introduisant cette contrainte supplémentaire on s'assure qu'on peut diviser le graphe de contrôle et de dépendance du programme en sous-graphes au niveau de chaque fonction, indépendamment les uns des autres. Chaque

---

14. Ce problème semble en fait se rapprocher des questions que se posent les gens qui font de la « dissipation du calcul », comme quoi faire une machine qui ne dissiperait pas reviendrait à avoir prévu à la construction toutes les solutions pour tous les jeux d'entrée. Néanmoins, c'est clairement intéressant si on veut construire une machine performante sur un petit nombre de *benchmarks*... pour donner la solution sans avoir à la calculer !

15. Ne serait-ce que parce que la philosophie du langage C y est relativement opposée : on doit pouvoir lier ensemble plusieurs objets ayant été compilés séparément sans avoir à recompiler certaines parties des objets. Evidemment, beaucoup d'optimisations globales ne sont pas possible dans ce cas, ce qui amène certains langages à proposer des fonctions de recopies textuelles de fonctions (comme `inline` dans C++) comme un bon compromis si ce n'est que cela augmente le nombre d'instructions à analyser à chaque « appel » [TIF86].

TAB. 8.1 - Quelques dépendances de données

<i>Symptôme</i>	<i>Pathologie</i>	<i>Notation <math>\delta</math></i>
Lecture après écriture	vraie dépendance	$\delta^t$
Écriture après lecture	antidépendance	$\bar{\delta}$
Écriture après écriture	dépendance de sortie	$\delta^o$

fonction possède un graphe de contrôle dont un des nœud n'a pas de prédécesseur, le début de la fonction, et un autre sans successeur, la fin de la fonction.

Afin de préserver la sémantique du programme, il faut préserver certaines dépendances entre données [TF70, KMC72], telles que par exemple « on ne peut pas utiliser un résultat avant qu'il ait été calculé ». Ces dépendances entre instructions peuvent être représentées par un graphe orienté valué où les sommets représentent les instructions du programme et les valeurs associées aux arcs le temps d'exécution des instructions aux sommets initiant ces arcs<sup>16</sup>.

Même si dans notre cas on ne s'intéresse qu'aux dépendances entre les données à l'intérieur du processeur — puisqu'on ne veut pas écrire un paralléliseur automatique — ces dépendances entrent de manière plus générale dans les dépendances entre les données d'un programme.

Celles-ci interviennent au niveau d'éléments mémorisants dans lequel on peut écrire ou lire une valeur. Les principaux type de dépendances sont résumés dans le tableau 8.1.

De manière intuitive, la dépendance vraie ( $\delta^t$ ) impose qu'avant d'utiliser un résultat on l'ait calculé. L'antidépendance impose une pérennité suffisante au résultat : une donnée ne doit pas être écrasée tant que les instructions qui en ont besoin ne l'ont pas utilisée ( $\bar{\delta}$ ). La dépendance de sortie ( $\delta^o$ ) indique qu'une variable ne peut accepter plusieurs résultats que dans un certain ordre dans le temps pour préserver la sémantique<sup>17</sup>.

La représentation de ces dépendances peut être assurée grâce à un *graphe orienté*

16. On suppose ici qu'une instruction fournissant un résultat dans un seul registre a un temps d'exécution unique. Ce n'est en fait pas toujours le cas, en particulier pour les opérations fournissant des double ou le 2<sup>ème</sup> mot de 32 bit est fourni un cycle après. Mais comme ces données sont utilisées par des opération demandant des double dont le 2<sup>ème</sup> mot de 32 bit est demandé aussi plus tard, cela se compense. Les manipulations sordides (mais utiles...) que l'on peut faire en C à coup d'union ou de passage par l'intermédiaire de coercition de pointeurs de type double avec autre chose passent par l'intermédiaire de la mémoire avec les compilateurs actuels et donc le problème n'apparaît pas. Sinon, on peut très bien généraliser la notion de temps d'exécution pour gérer ces problèmes s'ils surviennent.

17. Par exemple, on veut écrire une routine qui trouve le minimum d'un vecteur et qui renvoie l'indice de cet élément. Un moyen de le faire est balayer tout le vecteur en comparant chaque élément au minimum courant et le cas échéant en changeant le minimum courant et son indice. On constate que lors de l'exécution de la routine, l'indice du minimum n'est jamais lu mais toujours écrit. Il faut à tout prix conserver l'ordre d'écriture sous peine d'avoir un résultat faux.

de dépendance  $\mathcal{G}_d$  où un arc relie tout couple d'instructions  $I$  et  $J$  (noté  $I \xrightarrow[\mathcal{G}_d]{\delta} J$ ) où  $J$  dépend de  $I$ . On peut bien entendu accrocher aux arcs tous les renseignements utiles concernant chaque dépendance.

Afin de simplifier ce graphe de dépendance comme on le fait pour la trace, on projette le graphe de dépendance sur les nœuds du graphe de contrôle pour obtenir le *graphe de dépendances quotient*  $\mathcal{G}_q$ . De même que pour le graphe de contrôle par rapport à la trace, on rajoute très probablement des dépendances qui n'existaient pas et on a perdu de l'information sur l'exécution réelle du programme.

La parallélisation automatique d'un programme s'appuie sur une analyse du graphe de dépendance quotient des variables du programme et une réécriture de ce graphe de manière telle que le temps d'exécution soit minimal tout en conservant les dépendances de manière à assurer une équivalence sémantique du programme. Cela garantit que le programme modifié fournira bien le même résultat que le programme de départ.

Indépendamment de la parallélisation, la compilation pour une machine pipelinée éventuellement VLIW ou superscalaire essaye d'exploiter le parallélisme à grain fin, c'est-à-dire le fait qu'un processeur moderne puisse faire plusieurs choses en même temps. Il s'agit là d'optimiser l'organisation des instructions pour que le temps d'exécution du programme soit minimal. La perte d'information ci-dessus due à la projection de la dépendance dans le graphe de contrôle concernait surtout les dépendances entre cases mémoires, ce qui est moins gênant dans notre cas puisqu'on s'intéresse qu'aux dépendances internes au processeur.

Les processeurs superscalaires et certains processeurs pipelinés possèdent un mécanisme d'ardoise (*scoreboard*) permettant de gérer les dépendances par simple réservation d'un registre de destination qui bloque le registre contre toute lecture ou écriture tant que le résultat n'est pas disponible. Ce mécanisme simple permet de démarrer localement le plus tôt possible une des instructions en attente d'un résultat [Tho64]. Dans le cas d'un processeur à contrôle dynamique du parallélisme, on ne sait pas exactement la date d'exécution de chaque instruction mais la sémantique est respectée par le système de réservation.

### 8.3.2.2 Algorithme d'allocation temporelle

Par rapport à de l'optimisation de microcode [DLSM81, BCW89] on n'a pas de verrous de pipeline à gérer ou des bus qui imposent des relations temporelles strictes liées au fait que l'information présente sur un bus est fugitive. Comme on s'intéresse dans notre cas seulement à des registres où l'information est persistante, on peut attendre à loisir avant d'utiliser l'information et le graphe de dépendance au niveau des registres est valué par des inégalités temporelles. Par contre notre graphe de dépendance n'est pas acyclique, ce qui complique le problème.

Puisque l'ordonnancement des instructions restera inchangé, on peut raisonner sur un tableau d'instructions qui représente le stockage des instructions d'une fonction dans la mémoire de code. À chaque case du tableau on rajoute une structure précisant la réservation à faire pour l'instruction correspondante (quels registres sont réservés et pendant combien de temps) et un nombre de retard, qui indiquera le temps d'attente avant d'exécuter cette instruction, initialisé au temps d'exécution de l'instruction isolée pouvant la précéder, à savoir 1 ou 2 selon les cas.

En plus des dépendances sur les registres, il faut ajouter d'autres contraintes, les contraintes « ouvertes » [TF70] : il ne doit pas y avoir de conflit géré par le processeur sur un bus ou sur un registre de pipeline qu'une instruction peut demander un certain temps car sinon, par exemple en cas de retour d'exception, il se peut qu'une des 2 instructions qui était prioritaire ayant été exécutée avant l'exception ne cause plus de conflit et l'autre instruction est terminée à sa place au lieu d'être retardée : l'état du pipeline est modifié et peut par effet boule de neige désynchroniser le reste de la machine. Les conflits à éviter au niveau des ressources fugitives du processeur (par opposition aux registres) sont représentés par un graphe orienté où chaque arc est un conflit à éviter au niveau d'une ressource fugitive entre 2 instructions.

On obtient donc un système d'inéquations ( $\neq$ ) linéaires où chaque inéquation, représentant un arc  $I \xrightarrow[\mathcal{G}_q]{\delta^f} J$  dans le graphe de dépendance  $\mathcal{G}_q$ , doit être vérifiée pour la durée de tout chemin allant  $I$  à  $J$ ,  $I \xrightarrow[\mathcal{G}_c]{+} J$ , durée calculée en additionnant le retard introduit par chaque instruction du chemin et qui représente le temps d'exécution de la portion de programme pour aller de  $I$  à  $J$ . Remarquons qu'une simple analyse en profondeur d'abord convient très bien puisque ce n'est pas la peine de considérer les chemins dont la durée est strictement supérieure à la durée de  $I \xrightarrow[\mathcal{G}_q]{\delta^f} J$  à exclure.

Ces inéquations s'ajoutent au système d'inégalités à vérifier sur les registres, où chaque arc  $I \xrightarrow[\mathcal{G}_q]{\delta} J$  de  $\mathcal{G}_q$  représente une inégalité devant être vérifiée pour la durée de tout chemin  $I \xrightarrow[\mathcal{G}_c]{+} J$ . Pour la même raison, une analyse en profondeur d'abord convient.

Le problème de l'optimisation de la durée d'exécution du programme en jouant sur les retards de chaque instruction compte tenu des contraintes précédentes est non trivial et en plus on ne connaît pas *a priori* quels endroits devront être optimisés plutôt que d'autres en cas de choix exclusif puisqu'on raisonne sur le graphe quotient et non pas sur le graphe de dépendance complet basé sur la trace. C'est pour cette raison qu'il nous semble beaucoup plus simple et de toute manière plus dans la philosophie du projet de présenter une heuristique raisonnable et pragmatique de génération du code aval.

On constate d'abord que les dépendances  $\delta$  sur les registres existent pour pratiquement toutes les instructions (celles qui ne sont pas du contrôle de flot) alors que les conflits  $\delta^f$  sont assez rares.

De plus, les relations  $\delta$  étant des inégalités telles que si une relation  $I \xrightarrow[\mathcal{G}_q]{\delta} J$  est vérifiée, elle le sera encore si on ralentit  $I \xrightarrow[\mathcal{G}_c]{+} J$  alors que si une relation  $I \xrightarrow[\mathcal{G}_q]{\delta^f} J$  est vraie, il se peut très bien qu'elle ne le soit plus si on rallonge la durée de  $I \xrightarrow[\mathcal{G}_q]{+} J$ .

Cela nous amène à proposer un algorithme en 2 passes : d'abord satisfaire toutes les contraintes  $\delta$  en retardant chaque instruction destination d'une dépendance qui n'est pas correctement prise en compte et ensuite satisfaire toutes les contraintes  $\delta^f$  de la même manière. L'algorithme est résumé sur la figure 8.3 et utilise la définition simplificatrice  $I_\Delta$  suivante : soit une instruction  $I$  de la fonction considérée  $\mathcal{F}$  et une

---

```

Faire
  Pour toute instruction  $I$  Faire
    Si  $\exists I_\delta$ 
      Alors  $\text{retard}(I_\delta) := \text{retard}(I_\delta) + 1$ 
Tant que  $\exists (\xrightarrow{\delta})$  non r esolue
  Faire
    Pour toute instruction  $I$  Faire
      Si  $\exists I_{\delta f}$ 
        Alors  $\text{retard}(I_{\delta f}) := \text{retard}(I_{\delta f}) + 1$ 
Tant que  $\exists (\xrightarrow{\delta f})$  non r esolue

```

---

FIG. 8.3 - *Algorithme de planification du code.*

dépendance de type  $\Delta$ , on s'intéresse à une instruction  $J$ , si elle existe, telle que

$$J = I_\Delta \iff \left\{ \begin{array}{l} D = \left\{ x \in \mathcal{F} / (I \xrightarrow{g_c} x) \text{ et } (\text{durée}(I \xrightarrow{g_c} x) < \text{durée}(I \xrightarrow{g_a} x)) \right\} \\ J \in D, \forall K \in D, (\text{durée}(I \xrightarrow{g_c} J) \leq \text{durée}(I \xrightarrow{g_c} K)) \end{array} \right.$$

c'est-à-dire que  $I_\Delta$  est une des instructions les plus « proches » de  $I$  à ne pas respecter la contrainte temporelle due à  $\Delta$ .

Le fait qu'on incrémente le retard d'une unité en plusieurs fois plutôt que directement de la valeur manquante pour respecter la contrainte de dépendance est justifié pour la raison suivante : si on passe en revue toutes les instructions par adresse mémoire croissante, on poussera petit à petit les instructions posant problème. Si  $\text{durée}(I \xrightarrow{g_c} J) < \text{durée}(I \xrightarrow{g_a} J)$ , il se peut très bien qu'on rencontre ensuite une instruction  $K$  qui poussera une instruction  $L$  située sur  $I \xrightarrow{g_c} J$  et donc cela écartera aussi  $I$  et  $J$ . On peut voir cette méthode un peu comme une méthode itérative de type Gauß-Seidel de résolution de système linéaire par rapport à l'inversion directe du système : les contraintes sont relaxées petit à petit de manière équilibrée. Si on avait écarté dès la première itération  $J$  de  $I$  de la valeur liée à la dépendance, en poussant  $L$  on aurait trop éloigné  $J$ .

On est sûr de trouver une solution puisque toute instruction posant problème sera retardée et qu'une fois dépassé la durée de la contrainte la plus forte, toutes les contraintes seront toujours respectées quelles que soient les retards subis par cette instruction. Cela nous donne du même coup une limite supérieure qui est proche de la solution conservatrice consistant à exécuter de manière non pipelinée le programme. La différence étant due principalement au fait qu'on considère le graphe de dépendance quotient au lieu du graphe de dépendance originel.

En ce qui concerne la complexité, une fonction de  $f$  instructions où chaque instruction porte sa dépendance au plus jusqu'à  $d$  cycles nécessite au plus  $2d$  itérations (1 pour  $\delta$  et 1 pour  $\delta^f$ ). Dans le cas pire où toutes les instructions sont des instructions de

---

```

void daxpy(double a, double *x, double *y, int N)
{
    int i;
    for(i = 1; i <= N; i++)
        y[i] = a*x[i] + y[i];
}

```

---

FIG. 8.4 - Routine *daxpy* de la bibliothèque BLAS1.

débranchement conditionnelles<sup>18</sup>, le nombre d'instructions à considérer est  $2^d$ , chaque considération demandant au plus  $r$  tests, avec  $r$  le nombre maximal de ressources utilisées par une instruction. La borne supérieure de la complexité de l'algorithme est donc en  $\mathcal{O}(drf2^d)$ , les boucles principales étant exécutées au plus  $d$  fois, sachant que  $r$  et  $d$  sont des constantes petites. Dans la réalité il faut multiplier  $d$  par la probabilité d'avoir une instruction de branchement et pour la plupart des instructions  $r = 1$  et  $d = 1$ , ce qui rend l'algorithme très rapide. Néanmoins il existe le cas pathologique des divisions en double précision ( $d = 60!$ ) mais son taux d'utilisation est assez faible dans la pratique. Le problème est résolu en modifiant légèrement l'algorithme pour rajouter plus de retard d'un coup après une division.

Le terme en  $2^d$  n'apparaît pas par exemple dans [GM86] car seules des optimisations internes à des blocs  $y$  sont considérées, c'est à dire sans débranchement quelconque et par conséquent l'algorithme donne de moins bons résultats.

Notre algorithme est donc linéaire en fonction du nombre d'instructions. Les algorithmes de placement classiques sont plus complexes [DLSM81, Fis81, GM86], ce qui est normal puisque, outre le placement temporel que nous avons exposés, ils font de la réorganisation dans l'ordre d'exécution, ce qui demande beaucoup plus de travail. Par exemple l'algorithme de [GM86] est quadratique en le nombre d'instructions d'un bloc mais puisqu'il fait de la réorganisation limitée à l'intérieur de blocs, le terme  $2^d$  n'apparaît pas : il n'y a pas de débranchements par définition d'un bloc.

Le tableau 8.2 donne l'exemple de l'algorithme appliqué à la routine *daxpy* d'algèbre linéaire de la bibliothèque classique BLAS1 [?] écrite en C (figure 8.4<sup>19</sup>).

Les colonnes *Data Unit*, FP1, ADD2 et FPLAST contiennent les numéros de cycles relatifs pendant lesquels la ressource du même nom du processeur est occupée. Les instructions de fin de fonctions (*.ef*) marquées « † » sont supprimées pour être remplacées par les instructions marquées par « ¶ ». Ainsi on peut reporter sur le *jmp r1* final le retard permettant de garantir qu'il n'y a plus d'effet de bord une fois sorti de la fonction. Cela nécessite de préciser bien sûr que pour que l'algorithme de la figure 8.3 fonctionne, il faut attribuer à cette instruction finale l'usage fictif de toutes les ressources du processeur 2 cycles après son exécution (temps d'exécution d'un *jmp*) pour être certain que les dépendances seront résolues au retour d'exécution au niveau de l'appelant, évitant ainsi une analyse interprocédurale.

Enfin, il faut tenir compte du fait que certains registres doivent être traités de

---

18. Notons l'improbabilité et l'inutilité d'un tel programme puisqu'il faut au moins des instructions pour calculer chaque test.

19. Par mesure de simplification, la boucle n'a pas été déroulée 4 fois, comme le permet l'option *-OL* du compilateur C que nous possédons.

TAB. 8.2 - Placement temporel de la routine *daxpy*.

<i>Instructions</i>	<i>Registres</i>	<i>Temps d'exécution</i>	<i>Data Unit</i>	FP1	ADD2	FPLAST	<i>Retard</i>
<i>_daxpy:</i>							
subu  r31,r31,48	1	1					0
st.d  r24,r31,32		2	1,2				1
;							
or    r24,r0,r6	1	1					2
st    r30,r31,40		1	1				1
addu  r30,r31,40	1	1					1
or    r25,r0,r5	1	1					1
or    r5,r0,1	1	1					1
cmp   r10,r6,r5	1	1					1
bb1.n lt,r10,@L4		1					1
st    r1,r31,44		1	1				1
@L5:							
ld.d  r8,r4[r5]	3,4	1	1,2				2
fmul.ddd r8,r8,r2	9,10	2		0-3		8,9	4
ld.d  r6,r25[r5]	3,4	1	1,2				2
fadd.ddd r6,r6,r8	6,7	2		0,1	2	5,6	7
st.d  r6,r25[r5]		2	1,2				6
addu  r5,r5,1	1	1					2
cmp   r10,r24,r5	1	1					1
bb1   ge,r10,@L5		2					1
@L4:							
;							
.ef							
ld    r1,r31,44	3	1	1				2
ld    r30,r31,40	3	1	1				1
ld.d  r24,r31,32	3,4	1	1,2				1
addu  r31,r31,48	1	1					1 <sup>¶</sup>
jmp   r1		2					3 <sup>¶</sup>
jmp.n r1		1					†
addu  r31,r31,48	1	1					†
align 4							
data							
;_i    r5      local							
@L8:							
;_a    r2      local							
;_x    r4      local							
;_y    r25    local							
;_N    r24    local							

manière particulière dans le cas du MC88100 par notre algorithme. Par exemple le registre *r0* est spécial: on y lit toujours 0 (une constante souvent utilisée) et on peut l'utiliser en destination comme une poubelle. On peut donc aussi s'en servir à la base d'instructions inutiles. Il n'y a aucune réservation faite sur ce registre, ce qui évite les

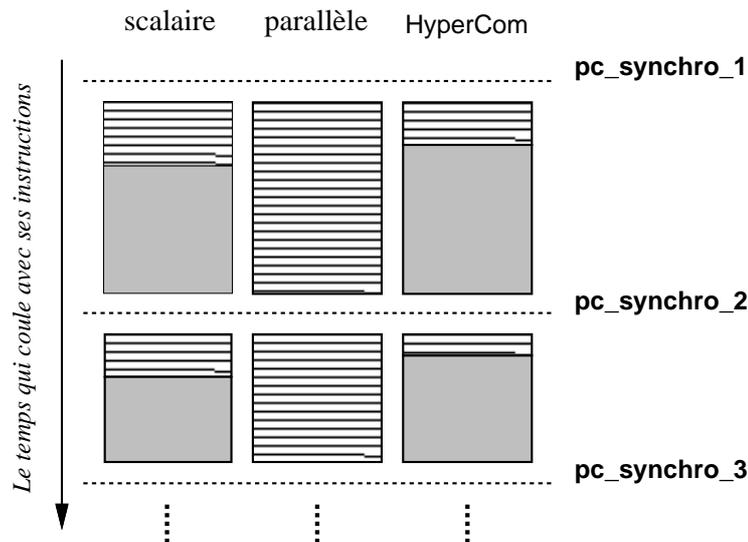


FIG. 8.5 - Allure des flots d'instructions appariés avant placement temporel.

effets de bord. `r0` est donc ignoré par notre algorithme de placement temporel.

Nous avons vu dans les sections 7.1.3 et 7.2.6 un certain nombre de méthodes de contrôle de flot SIMD dont il faut tenir compte au niveau de notre algorithme :

- dans le cas d'un entrelacement d'instructions de blocs alternatifs terminaux, il est clair qu'il ne peut y avoir de dépendances entre instructions des 2 blocs respectifs puisqu'ils sont à exécution mutuellement exclusive ;
- dans le cas d'utilisation de branchements non retardés pour faire le contrôle de flot SIMD, il ne faut pas que l'algorithme de placement sépare un tel branchement des instructions suivantes qu'il conditionne car sinon celui-ci perdrait son effet. Cela revient à attribuer à ces branchements un temps d'exécution de 1 (au lieu de 2 dans le cas du `MC88100`). Comme dans notre cas un branchement ne peut conditionner qu'une seule instruction suivante, on est à l'abri d'un éventuel éclatement lié aux dépendances dans le groupe d'instructions conditionnées par le branchement non retardé. Cela simplifie beaucoup la tâche.

Maintenant qu'on a vu comment placer temporellement du code scalaire, on va pouvoir étendre l'algorithme au code global.

### 8.3.3 Le placement temporel du code global

Entre chaque fonction de marquage, chaque type de code est indépendant : le code scalaire n'a pas de dépendance vis-à-vis du code parallèle et réciproquement. Les dépendances liées aux communications, diffusions scalaires, etc., sont résolues dans des fonctions de bibliothèque séparées optimisées à la main et n'apparaissent pas dans le code de l'utilisateur car justement masqué par l'appel à la fonction de bibliothèque du système<sup>20</sup> qu'a généré le compilateur POMPC.

<sup>20</sup>. On ne fait pas d'analyse interprocédurale comme on l'a déjà mentionné.

Le code tel qu'il est généré avant le placement temporel statique nécessaire à une exécution déterministe sur POMP est représenté sur la figure 8.5. Afin que l'algorithme de placement évoqué ci-avant sur la figure 8.3 continue de fonctionner pour les trois flots d'instructions il faut leur assigner la même longueur. Pour ce faire, on va aligner chaque ensemble entre 2 fonctions de marquage sur le bloc au temps d'exécution le plus long en rajoutant autant de `nop` virtuels que nécessaires à la fin des blocs.

On peut exécuter l'algorithme précédent presque indépendamment sur les 3 types de flots d'instructions. La seule restriction est de garder toujours une même durée d'exécution sur les 3 flots. Cela est fait simplement, outre le rajout d'un « **Pour** code scalaire, code PES, code **HyperCom Faire** » avant chaque « **Pour toute** instruction  $I$  », en modifiant les parties  $\text{retard}(I_\delta) := \text{retard}(I_\delta) + 1$  afin que la durée d'exécution soit vérifiée globalement :

- si on retarde une instruction d'un des flots qui n'est pas le plus long, il suffit de réduire d'1 cycle le `nop` qui le termine, la durée d'exécution de cette portion sera globalement inchangée ;
- si l'instruction que l'on retarde appartient déjà au flot le plus long (qui ne se termine donc pas par un `nop`), on retarde les 2 autres en rajoutant un `nop` virtuel à la fin de leur portion et là la portion globale est bien ralentie d'un cycle.

Enfin, une fois qu'on a établi les retards pour les 3 flots, la génération de code finale consiste simplement sortir les flots d'instructions en prenant soin de transformer chaque instruction retardée de  $\tau$  par  $\tau - 1$  `nop` suivi(s) par l'instruction et les `nop` virtuels engendrés par l'algorithme par de vrais `nop`.

En mettant toutes les données initialisées à la fin de chaque code, on obtient après assemblage de chaque code 3 codes alignés que pourra utiliser le chargeur de programme de POMP.

L'algorithme de placement global a donc la même complexité que l'algorithme de placement où on ne considèrerait qu'un seul flot de code.

## 8.4 L'environnement de programmation

### 8.4.1 La mise au point des programmes

Il faut bien entendu fournir un environnement de développement et de mise au point raisonnable pour faciliter la vie du programmeur mais cet environnement est très coûteux en temps de développement.

C'est pourquoi nous<sup>21</sup> avons continué la politique minimaliste consistant à récupérer le maximum de choses déjà existantes, à savoir l'environnement de l'hôte, au moins dans un premier temps.

A ce niveau, l'intérêt d'une compilation pour CM-2, pour WAVETracer ou pour une station de travail est que comme le processeur scalaire est une station de travail standard, on peut utiliser tout l'environnement logiciel de la station de travail, y compris l'environnement de mise au point ou l'analyseur de performance de type `gprof` d'UNIX avec tout programme parallèle.

---

21. En l'occurrence, pour la section 8.4.1, il s'agit surtout de Nicolas PARIS.

Bien entendu on ne contrôle que la partie scalaire du programme mais ce n'est pas gênant dans l'approche parallélisme de données que nous avons choisie, alors que dans le cas du parallélisme de contrôle on aurait des problèmes pour savoir quelle sémantique donner à un point d'arrêt par exemple.

Sur POMP actuellement ou sur la MASPAR cela n'est pas possible car il faut que cet analyseur fonctionne sur la station de travail qui doit être aussi le processeur scalaire de la machine parallèle, sinon les performances mesurées sont celles du serveur de « service UNIX » que joue l'hôte par rapport à la machine parallèle. Dans ce cas, il faut alors définir un nouvel environnement de mise au point qui puisse tourner sur le processeur scalaire.

#### 8.4.1.1 L'hôte est le processeur scalaire $\rightsquigarrow$ `dbxtool`

Dans le cas où la machine cible a une station de travail comme processeur scalaire, on utilise le débogueur d'origine, à savoir typiquement `dbxtool` sur SUNOS.

Le compilateur POMPC rajoute dans les fichiers du langage de la machine cible (C\* sur la CM, C sur station de travail) les directives `#line` qu'il faut pour que le débogueur affiche bien le fichier POMPC et non le fichier intermédiaire pour la machine cible, quoique cela soit possible pour faire la mise au point du compilateur lui-même, par exemple.

Les principaux problèmes consistent en :

- afficher des variables parallèles dont la notion est inconnue de `dbxtool` ;
- ne pas se tromper de collection à travers les différents appels successifs avec passage de collections en paramètre et permettre un affichage de la pile d'appel des fonctions correct.

Dans le premier cas il suffit de rajouter à `dbxtool` des boutons d'impression appelant des fonctions d'affichage, rajoutées automatiquement au code par `pcc`, qui sont capables, elles, d'afficher les variables après avoir trouvé leur type dans une table des symboles décrite dans la suite. Des fonctions permettant d'afficher graphiquement les variables ont aussi été rajoutées et deviennent rapidement indispensable pour mettre au point agréablement des modèles numériques, par exemple, grâce à l'aspect synthétique de ce mode de visualisation.

Le 2<sup>ème</sup> problème consistant à suivre la propagation des passages de paramètres et des collections à travers les appels fonctionnels est résolu en insérant à l'entrée de chaque fonction du code mettant dans une pile indépendante de celle des appels fonctionnels, gérée par le compilateur final, ces informations et en insérant à la fin de chaque fonction du code nettoyant cette pile.

En plus, une table des symboles est rajoutée au programme non pas sous forme de « .o », car cela est hautement non portable ou alors très long à fouiller, mais sous forme d'exécutable accédé par des fonctions spécialisées.

Ainsi les routines d'affichage sont capable de retrouver toutes les informations concernant la variable dont on indique le nom et celui de la procédure courante et donc d'afficher les informations que demande l'utilisateur. Un exemple des possibilités d'affichage est donnée plus loin sur la figure ??.

Néanmoins cette méthode ne permet pas de tout faire ce que sait faire `dbxtool` en temps normal, en particulier évaluer des expressions parallèles puisque celles-ci sont

vues comme des pointeurs scalaires parmi tant d'autres au niveau de `dbxtool`. Cela demanderait l'écriture d'un évaluateur d'expressions parallèles complet (donc d'un interpréteur) de POMP. Même si cela est faisable assez facilement à partir du travail fait pour le compilateur, cela n'est pas la priorité numéro 1 du projet car la plate-forme actuelle suffit largement à répondre à la plupart des cas considérés.

#### 8.4.1.2 Pour POMP : `bugtool`

Dans le cas de POMP, il n'y avait pas de débogueur disponible répondant à nos souhaits. En effet ces derniers sont prévus d'une part pour fonctionner avec un MC88100 « normalement » utilisé et d'autre part prendraient trop de place mémoire sur notre prototype.

Puisque la mémoire du processeur scalaire est visible depuis l'hôte, on peut très bien faire tourner un débogueur sur l'hôte tout en mettant au point le code du processeur scalaire par interruptions interposées, un peu à l'inverse de ce qui est fait pour les appels systèmes (§ 8.4.2).

Ainsi, un débogueur minimal a été conçu et intégré au chargeur `ld88` développé afin de permettre un affichage de la pile et des variables de POMP de manière symbolique mais aussi de tous les registres du MC88100, lorsqu'on arrête le programme ou qu'il s'arrête sur une erreur.

Dans notre approche minimaliste, l'interface graphique de `dbxtool` a été récupérée et modifiée pour que ce soit notre programme de mise au point `bug` et non pas `dbx`<sup>22</sup> qui soit lancé. Le nom de `bugtool` a suivi.

#### 8.4.2 Intégration dans l'hôte : point de vue système

En ce qui concerne POMP, beaucoup de choses sont faites au niveau de l'hôte du point de vue système. Toutes les fonctions de bibliothèques classique<sup>23</sup> sont venues avec l'environnement de programmation du MC88100, ce qui a permis de gagner un temps considérable par rapport à ce qu'aurait demandé une réécriture de toutes ces bibliothèques<sup>24</sup>.

Mais à un moment donné ces bibliothèques ont besoin de faire appel à des fonctions de plus bas niveau : les fonctions système de la machine, qui constituent dans notre cas le cœur d'un système UNIX. Bien entendu on aurait pu acheter un UNIX tout fait<sup>25</sup> mais pour machine séquentielle monoprocesseur, ce qui ne convenait pas, ou bien réécrire un UNIX SIMD complet, ce qui dépassait totalement la main-d'œuvre disponible sur le projet.

Pour cette raison, on a sous-traité tous les appels systèmes raisonnables à notre SUN 3 : opérations sur les fichiers, `time()` et `exit()` principalement. Cela fait peu par rapport à ce qui est disponible dans un UNIX normal (`man 2`) mais cela suffit pour toutes les applications calculatoires classiques. Un bon critère validant cette affirmation

---

22. Comme c'est le binaire qui a été modifié, il fallait trouver un nom d'exactement 3 lettres pour remplacer « `dbx` ». « `bug` » a fait l'affaire...

23. `man 3` d'UNIX: `stdio` avec `printf` par exemple, les fonctions mathématiques, etc.

24. Un point intéressant à noter est que maintenant les bibliothèques UNIX BSD sont du domaine publique et récupérables par ftp.

25. Mais seulement un UNIX SVR3, hélas, ce qui est nettement moins convivial. Mais là, c'est un problème de « religion »...

est que ces applications sont programmable en FORTRAN, c'est-à-dire avec pas grand chose : pas de réseau, pas de contrôle de la mémoire virtuelle, etc. En plus, comme la machine est monotâche dans sa version actuelle, aucun système de contrôle de processus n'est nécessaire.

Pour que les fonctions systèmes nécessaires soient exécutées sur l'hôte lorsque POMP le désire, il faut que l'hôte en soit avertit. Cela peut être fait de 2 manières :

- 1° l'hôte surveille constamment un registre ou une case mémoire qu'il a en commun avec POMP. Dès que POMP veut faire un appel système, une structure de données décrivant l'appel système avec ses paramètres est mis à un endroit connu par POMP et l'hôte et le registre ou bien la case mémoire change de valeur pour avertir l'hôte : c'est la méthode de scrutation (*polling* en anglais) ;
- 2° plutôt que de scruter une zone commune, on peut envoyer une interruption à l'hôte pour lui dire qu'on a besoin de faire un appel système.

Chaque méthodes a ces avantages et ses inconvénients. La première est plus simple à mettre en œuvre et a un temps de réponse très rapide dans la mesure où l'hôte ne fait quasiment qu'attendre une mise à contribution de POMP. La seconde permet à l'hôte de faire plus de choses le reste du temps où il n'a pas d'appel système à servir. Cela peut être utilisé pour réaliser des entrées-sorties asynchrones (en temps différé) vis-à-vis de POMP. Pour l'instant, c'est la méthode de scrutation qui est utilisée du fait de sa simplicité, même si la méthode par interruption est amenée à la remplacer.

La seule particularité enfin a été de tenir compte de la simplification qui a été faite au niveau du bus VME : le fait qu'on n'autorise que les accès sur des entiers alors que les entrées-sorties ont besoin d'accès au niveau octet. Même lorsqu'on fait une écriture d'un bloc aligné sur des `int` on n'est pas sûr que l'écriture au niveau d'UNIX ne sera pas découpée en sous accès non alignés, dus au fait que l'écriture est faite en plusieurs accès DMA ou bien qu'il y a des temps d'attentes (*socket*, *FIFO*,...), qu'une interruption est survenue au cours de l'appel système ou encore que plusieurs essais sont faits en cas d'échec. Si on ne fait pas attention, cela se traduira par un *Bus Error* au niveau VME mais qui sera récupéré au niveau du noyau UNIX et retraduit en un simple échec de l'entrée-sortie<sup>26</sup>. Pour cette raison, on est toujours obligé, dans le cas d'un `write()` de faire une copie de la zone à écrire depuis POMP vers un tampon intermédiaire sur l'hôte et inversement en cas de lecture.

La sémantique des écritures et lectures UNIX n'est pas exactement respectée dans la mesure où tout accès est tronçonné de manière à loger dans les tampons de l'interface logicielle `ld88` sur le SUN, ce qui n'est pas le cas sur une machine UNIX native, mis à part le cas particulier des accès à des *streams*. Mais cela n'a pas d'importance en ce qui concerne l'utilisation scientifique demandée.

## 8.5 Conclusion

Nous avons présenté une méthode pragmatique permettant de récupérer le maximum des logiciels qui étaient à notre disposition aussi bien du point de vue des pro-

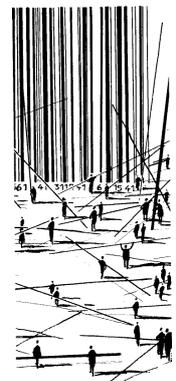
---

<sup>26</sup>. Ce qui laisse bien entendu le constructeur de la machine dans un état perplexe un certain temps au moment de la mise au point...

cesseurs scalaire et élémentaires de POMP, récupération du compilateur C et de l'assembleur grâce à une méthode d'alignement et de placement temporel ainsi que les bibliothèques UNIX, que du point de vue de l'hôte par une perversion de l'environnement de mise au point et l'utilisation d'UNIX pour servir POMP.

Cela permet de réduire fortement les développements logiciels et donc valide l'approche consistant à utiliser des processeurs du commerce pour construire une machine. Cela permet aussi de s'affranchir d'un choix absolu de tel ou tel processeur du fait de la portabilité de la méthodologie, considérant le langage C comme un langage d'assemblage portable.

On peut aborder maintenant le problème délicat d'une machine parallèle : son réseau d'interconnexion entre les processeurs.





# Chapitre 9

## Le réseau d'interconnexion

ES réseaux apparaissent dans les machines multiprocesseur comme étant le point clef d'une architecture, particulièrement lorsque le nombre de processeurs augmente, car ceux-là sont à la base de la coopération entre les processeurs qui mène à une exécution efficace des algorithmes parallèles.

Le problème devient de plus en plus criant avec l'augmentation de vitesse des processeurs : plus ceux-ci sont puissants et plus les réseaux sont mis à rude épreuve. En fait, les performances du réseaux doivent probablement augmenter plus vite que celles des processeurs si on considère que la taille des données des programmes exécutés sur la machine grandira en conséquence et pourra, selon les applications, provoquer encore plus de trafic entre les processeurs.

Dans le cadre de notre projet, il convient donc d'offrir un réseau bien équilibré à notre machine par rapport à la puissance de calcul des processeurs, en tenant compte bien évidemment du coût technologique de chaque réseau. Pour ce faire, nous allons au préalable étudier quelques types de réseaux, étude qui sera d'ailleurs réutilisée dans la section ?? pour une approche SPMD.

Ensuite, nous développerons un type de réseau particulièrement adapté à l'usage qu'on veut en faire sur POMP.

### 9.1 Introduction

On peut étudier les réseaux selon 4 caractéristiques à peu près orthogonales [Fen81] :

**la topologie :** elle décrit le graphe sous-jacent d'un réseau où les sommets du graphe sont des commutateurs ou des processeurs et où les arcs sont des liens de communications. L'idée de base est de faire propager les messages entre les nœuds

à travers les liens, chaque nœud essayant de choisir au mieux quel lien doit emprunter un message pour se rapprocher de son but.

L'étude du graphe permet de connaître le *diamètre* du réseau, c'est à dire le nombre de liens qu'un message doit traverser pour aller d'une entrée à une sortie. Un autre paramètre important du réseau est son *degré*, c'est-à-dire le nombre de liens qui sortent par processeur (qui n'est pas forcément le même pour chaque processeur). La complexité du graphe indique la complexité du réseau de communication bien entendu, mais aussi la facilité de projection de ce graphe dans un espace à 2 ou 3 dimensions (l'espace physique dans lequel nous vivons!) donne une idée de la réalisabilité du réseau. Un réseau dont le degré est très élevé sera probablement plus difficile à construire qu'un réseau dont le degré est plus faible et par conséquent la filasse sera sans doute moindre.

En outre on peut séparer les topologies figées (*statiques*) des topologies qui peuvent évoluer (*dynamiques*) pour relier différents processeurs au cours du temps.

**la stratégie de contrôle :** le contrôle du réseau peut être soit centralisé, soit local :

- dans le cas d'un contrôle centralisé, on peut faire des optimisations sur le choix des chemins possibles pour les messages mais cela prend du temps et nécessite d'avoir un réseau propre autonome uniquement pour collecter l'information de routage vers le contrôleur du réseau et la redistribuer vers les commutateurs. En outre l'optimisation globale du routage nécessite un coût de calcul important qui peut faire perdre l'intérêt de l'optimisation ;
- dans le cas d'un contrôle local, chaque nœud prend des décisions de routage dépendant de conditions locales. Le routage est simple et rapide mais aucune optimisation n'est possible au niveau du réseau tout entier.

Malheureusement certains réseaux ne fonctionnent pas sans routage global, ce qui les rend délicats à mettre en œuvre efficacement.

**la méthode de commutation :** il y a deux méthodes principales de commutation : la commutation de circuits et la commutation de paquets :

- dans le cas de la commutation de paquets, les données sont envoyées par paquets qui traversent le réseau de proche en proche, chaque nœud l'envoyant à un de ses voisins.
- en ce qui concerne la commutation de circuits, un canal de communication est ouvert et réservé par une configuration du réseau. Une fois ouvert, ce circuit permet d'envoyer directement des données depuis une entrée jusqu'à une sortie du réseau. Comme la configuration n'a lieu qu'une fois, cette méthode est intéressante plutôt pour l'envoi de beaucoup de données à la fois.

D'autres systèmes intermédiaires entre ces deux choix ont été développés pour essayer de récupérer les avantages des deux.

**le synchronisme :** les opérations décrites précédemment peuvent être réalisées de manière synchrone ou asynchrone, mais cela a finalement peu d'importance si ce

n'est dans le cas d'un contrôle global où la réalisation synchrone sera probablement plus simple. La notion de synchronisme interviendra probablement plus par contre dans les problèmes fins d'implantation.

L'étude des réseaux revient donc à explorer l'espace du produit cartésien des caractéristiques précédentes, sachant que sont plus particulièrement étudiés les réseaux dynamiques à commutation de circuits et les réseaux statiques à commutation de paquets. Cela est dû au fait que les notions de commutation et de « staticité » ne sont pas très orthogonales : un réseau dynamique peut être vu (et est souvent réalisé !) comme un réseau statique dont certains nœuds seraient des commutateurs de circuits. Tout dépend donc de ce que l'on met aux nœuds : des processeurs ou des commutateurs (routeurs). La phrase « réseaux dynamiques à commutation de circuits » est donc un pléonasme mais nous la conservons par respect des traditions. La nuance peut néanmoins être trouvée si on considère que dans le cas d'un réseau dynamique, la commutation de circuits sous-jacente est contrôlée globalement (décrivant la topologie dynamique par l'intermédiaire de circuits) et ne dépend pas des messages passant à chaque nœud (car véhiculés dans des canaux virtuels dénommés « circuits »).

De plus, on peut compliquer à loisir la classification précédente en mélangeant les types de réseaux, en rajoutant des niveaux de hiérarchie par exemple.

Le choix d'un réseau dépend bien entendu de ce qu'on en attend [WyF84]. Le problème est d'autant plus complexe que non seulement de nombreux paramètres et critères interviennent mais qu'en plus certains sont contradictoires ou problématiques. Parmi les critères à considérer on peut nommer :

- le degré, donc le nombre de liens par nœud, a tendance à augmenter la filasse de la machine ou tout au moins le nombre de pattes de communications de chaque nœud et par conséquent doit plutôt rester faible. En outre, si le degré est fixe, cela évite de particulariser certains nœuds et apporte les avantages de la modularité et de la répétitivité ;
- le diamètre, c'est-à-dire le maximum du plus court chemin entre tout couple de nœuds du réseau, a tout intérêt à être faible pour diminuer la latence du réseau, c'est-à-dire le temps que met un message pour aller de son point de départ à sa destination ;
- le routage doit bien sûr être possible, mais aussi, simple à réaliser, ne dépendant que de l'adresse du nœud de destination et de conditions locales par exemple ;
- la symétrie est importante si on ne veut pas particulariser les nœuds et les algorithmes de routage ou avoir des embouteillages chroniques de paquets à certains endroits ;
- le débit du réseau doit être important pour ne pas (trop) brider les processeurs ;
- un faible coût de démarrage augmentera les performances du réseau pour des paquets plus petits ;
- la largeur des liens doit rester faible car elle détermine avec le degré le nombre de pattes de communications par nœud et la quantité de fils de la machine ;

- le nombre de pattes réservées au réseau sur chaque nœud (le produit degré, largeur des liens) doit rester dans les limites du raisonnable, par exemple afin que la gestion du réseau puisse loger dans un circuit intégré ;
- le réseau doit être performant sur les permutations les plus souvent utilisées, en particulier capable d'émuler les communications sur grille ;
- un réseau facilement extensible permet une évolution de la taille de la machine dans le temps ;
- s'il est capable de fonctionner même si un ou plusieurs nœuds sont en panne c'est un plus ;
- le réseau doit être réalisable dans l'espace réel à 2 ou 3 dimensions<sup>1</sup> ;
- si on veut réaliser une grosse machine, il faut que le réseau puisse être découpé facilement en plusieurs cartes, si possibles identiques pour des questions de coût, et que le nombre de fils entre cartes soit compatible avec les connecteurs disponibles dans le commerce.

Comme pour n'importe quel choix, lorsqu'on a trouvé *la* bonne fonction de coût, on a presque trouvé la solution. Le problème ici est que la fonction de coût dépend beaucoup de l'application visée par la machine, qui peut par exemple n'avoir besoin que de peu de communication et donc seul intervient le coût du matériel<sup>2</sup>.

On peut faire intervenir par exemple, outre le nombre de pattes des circuits intégrés [FWT82, AP91b], la bisection du réseau [?], le nombre de liens dans la machine [RS83], etc. pour tenir compte des coûts technologiques.

Du point de vue de l'utilisateur, on veut que l'émission d'un nombre entier se rapproche le plus possible du temps d'accès à la mémoire ou du temps d'exécution d'une addition, sachant qu'il est plus raisonnable d'intégrer le réseau dans la hiérarchie informatique classique registres/cache/mémoire, après la mémoire.

Dans la suite, les réseaux sont décrits comme reliant des processeurs entre eux par abus de langage alors qu'ils peuvent relier plus généralement processeurs et mémoires, etc. On supposera de surcroît que le nombre maximal d'émetteurs et celui de récepteurs sont identiques par mesure de simplification (« réseaux carrés » plutôt que rectangulaires) mais cela n'enlève rien à leur généralité.

On va s'intéresser aux réseaux à commutation de circuits, aux réseaux à commutation de paquets et aux méthodes hybrides.

## 9.2 Les grandes classes de réseaux

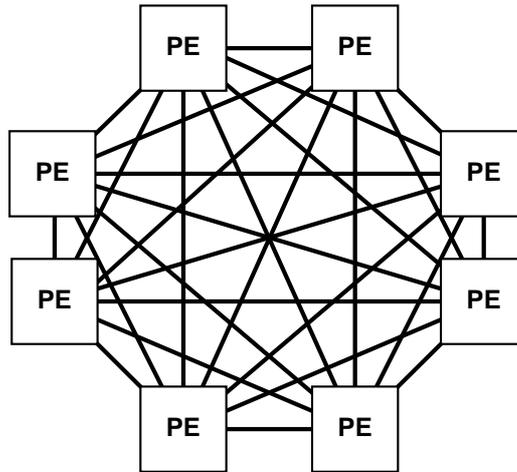
### 9.2.1 La commutation de paquets : un réseau statique

Dans un réseau statique, les processeurs sont situés aux sommets d'un graphe dont la topologie est fixe. Un processeur ne peut envoyer un message qu'à un de ses « voisins »

---

1. Dans le cas d'un circuit imprimé, la dimension est supérieure à 2 (on n'est probablement pas dans un espace de RIEMANN) mais strictement inférieure à 3.

2. On pourrait imaginer par exemple une machine à calculer des fractales de MANDELBROT qui ne nécessiterait pratiquement pas de communications.

FIG. 9.1 - *Synoptique d'un réseau totalement connecté.*

du réseau. Si un processeur veut dans ce type de réseau envoyer un message à un processeur qui ne lui est pas directement connecté, il faut qu'il envoie le message vers un de ces voisins qui se chargera lui-même de rapprocher le message de sa destination.

### 9.2.1.1 Réseau totalement connecté

Il s'agit du réseau le plus simple à imaginer : pour relier des processeurs, on les relie 2 par 2 avec des liens de communications (figure 9.1). Ainsi, chaque processeur peut communiquer simultanément avec tous les autres processeurs.

Malheureusement cela se traduit par  $N(N-1)/2$  liaisons bidirectionnelles pour interconnecter  $N$  processeurs, ce qui est prohibitif pour des  $N$  élevés. En outre, tous les processeurs ont rarement besoin de communiquer simultanément avec tous les processeurs, d'autant plus qu'ils ne seraient probablement pas capables de soutenir le débit. Mais si tel était le cas, cela voudrait dire que le débit de chaque lien serait faible et donc qu'on pourrait remplacer tout ce réseau par un réseau plus économique, avec des liens à débit plus important.

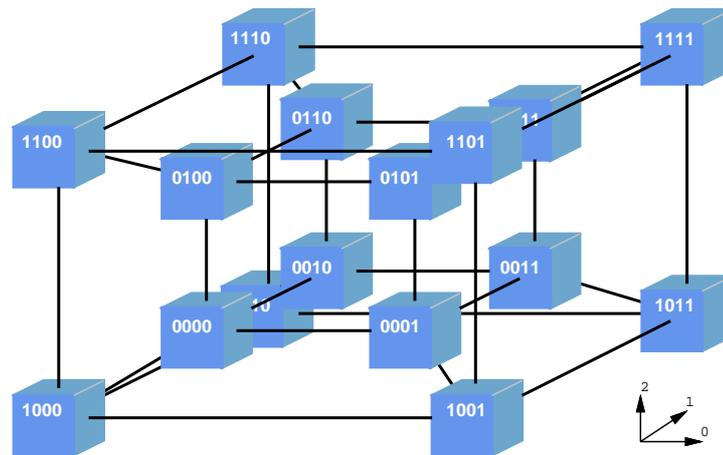
Cela explique que ce réseau ne soit jamais employé avec un nombre important de processeurs. On peut néanmoins jouer sur les constantes en diminuant la largeur des liens de connexion : on augmente la fréquence des liens à débit équivalent, au prix d'une interface réseau plus complexe au niveau de chaque processeur. C'est cette approche qui est étudiée dans [LEH<sup>+</sup>89, Sai91].

### 9.2.1.2 L'hypercube

Un hypercube de dimension  $n$  est un réseau qui généralise le carré (en dimension 2) et le cube (en dimension 3) à des dimensions supérieures.

C'est un réseau où chaque processeur a pour voisin les processeurs dont le numéro représenté en base 2 a un chiffre de différence<sup>3</sup>, comme l'indique la figure 9.2.

3. POMPeusement, on dira qu'ils sont à une distance de HAMMING de 1.

FIG. 9.2 - *Synoptique d'un réseau hypercube.*

L'hypercube a un certain nombre de propriétés intéressantes [SS88] qui l'ont fait adopter dans de nombreuses machines :

- le diamètre du réseau est assez faible :  $n = \log_2 N$ , quoique non optimal ;
- le manque d'optimalité se traduit par de nombreux chemins entre les processeurs, ce qui est intéressant pour limiter les contentions et augmenter la tolérance aux pannes ;
- la construction peut se faire de manière récursive : il est assez simple de construire un hypercube de dimension  $n + 1$  à partir de 2 hypercubes de dimension  $n$  en rajoutant un lien par processeur ;
- on peut utiliser l'hypercube pour simuler des grilles de dimensions diverses, en énumérant des champs de bits du numéro des PES suivant un code de GRAY, utiles pour tous les algorithmes du style résolutions itératives d'équations différentielles qui se contentent souvent de communications locales (dans le sens de voisinage sur une grille).

Par contre, lorsque le nombre de processeurs augmente, il devient de plus en plus difficile de construire le réseau dans un univers à 2 ou 3 dimensions seulement, surtout lorsque les liens sortent d'une carte.

On peut généraliser les hypercubes en utilisant autre chose que la base 2, mais cela revient à remplacer chaque hyperface par un réseau totalement connecté, ce qui n'est pas très intéressant pour des bases élevées, comme on a pu le constater.

### 9.2.1.3 Les grilles et les tores

Par contre, une autre généralisation de l'hypercube est plus intéressante : elle consiste à remplacer chaque arête (qui n'a que 2 processeurs pour  $n > 0$ ) par une ligne de  $k$  processeurs (figure 9.3). Le diamètre et le diamètre moyen sont alors  $nd$  et  $\frac{nk}{2}$

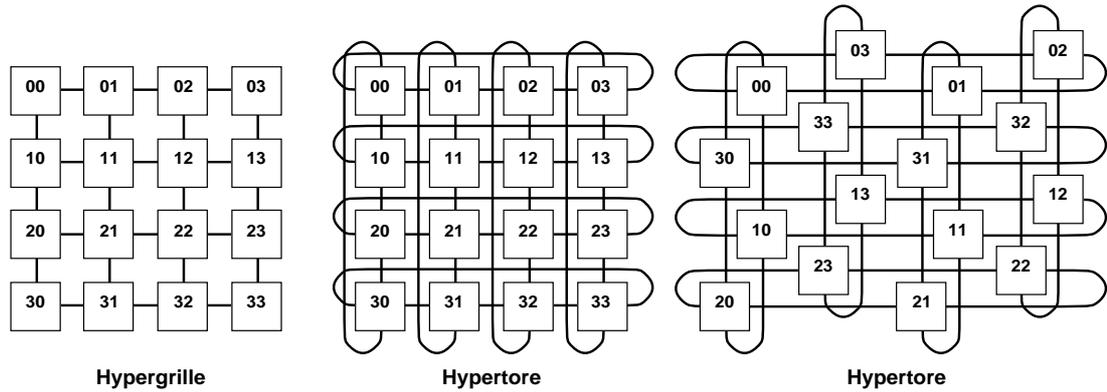


FIG. 9.3 - Schéma d'une hypergrille de dimension 2 et d'un hypertore de dimension 2 (dessiné de 2 manières différentes).

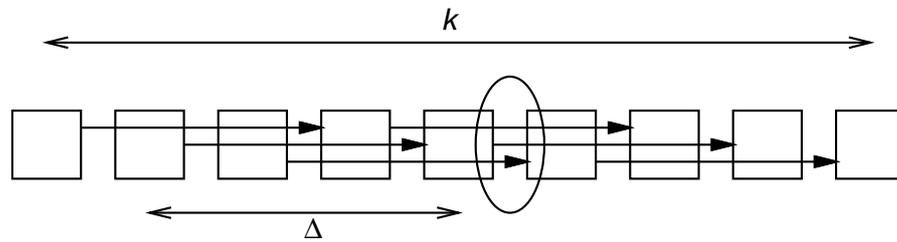


FIG. 9.4 - Réalisation d'une communication sur grille sur un réseau de type grille.

respectivement<sup>4</sup> pour  $k > 2$ .

Si en plus on connecte les deux processeurs qui sont à l'extrémité d'une arête, on obtient un hypertore, ce qui permet de diviser les diamètres précédents par 2. On peut trouver un arrangement permettant de limiter la taille maximale des liens en entrelaçant les nœuds suivant chaque dimension comme indiqué sur le dessin de droite de la figure 9.3.

La comparaison entre les grilles et les tores amène quelques réflexions. En particulier, à section constante, les réseaux ont pratiquement même débit. Le diamètre double de la grille est compensé par le fait que les liens sont de taille double. Par contre dès qu'on veut faire une opération de décalage global torique, comme dans la grille on a perdu la symétrie torique, certains paquets des bords seront obligés de traverser tout le réseau pour aller sur l'autre bord : une opération demandant 1 cycle sur le tore en demande  $k$  sur la grille. Les tores sont donc plus intéressants que les grilles pour les communications de ce type. Par contre en ce qui concerne le partitionnement en plusieurs sous-réseaux, il est très difficile de réaliser un tore partitionnable. Sur ce point, les grilles ont l'avantage.

Un problème paradoxal en ce qui concerne ce type de réseau est qu'il n'est pas très performant pour réaliser des communications sur grille distantes, contrairement aux hypercube [SS88]. Le propos est illustré par la figure 9.4 qui ne montre qu'un décalage

4. Ou encore  $\frac{nk^{n+1}}{k^n-1}$  et  $\frac{nk^{n+1}}{2(k^n-1)}$  respectivement si on considère qu'un processeur n'envoie pas les messages pour lui-même à travers le réseau.

d'une distance  $\Delta$  dans une seule dimension par mesure de simplification. On constate qu'entre 2 nœuds, sauf aux alentours des bords,  $\Delta$  communications veulent utiliser le lien de communication dans cette direction. Si on laisse la situation se résoudre telle quelle par le mécanisme de gestion de conflits, chaque paquet attendra que son lien se libère, ce qui prendra un temps  $\mathcal{O}(k\Delta)$ . Par contre, si on divise la communication en  $\Delta$  communications  $i$  ne concernant que les processeurs émetteurs d'adresse  $a$  avec  $a \equiv i[\Delta]$ , il n'y aura plus de conflit et le temps des  $\Delta$  communications sera en  $\mathcal{O}(\Delta^2)$  seulement<sup>5</sup>.

L'avantage des grilles et tores de faible dimension (1, 2 ou 3) est qu'on peut facilement les réaliser dans l'espace physique et mettre des liens très larges puisque les communications sont de voisinage. La philosophie de [?] est de dire qu'une machine à  $N$  PES est forcément réalisée dans l'espace physique à 3 dimensions, peut communiquer avec un débit qui évolue au moins en  $\mathcal{O}(N)$  alors que la bisection n'évolue qu'en  $\mathcal{O}(N^{\frac{2}{3}})$ . De plus, si on projette un réseau dans un espace à 3 dimensions, on veut éviter que des messages soient amenés à passer plusieurs fois à travers n'importe quelle bisection car cela diminue clairement la bande passante. En utilisant un réseau de type grille de faible dimension, on limite intrinsèquement ce cas. Ainsi, en exploitant au mieux l'espace réel, on peut espérer compenser le diamètre assez important de ce type de réseaux par un débit très élevé entre chaque sommet du graphe. Si on raisonne à bisection du graphe constante, ce qui est raisonnable lorsqu'on a à partitionner un graphe pour le réaliser, l'avantage va donc pour les faibles dimensions.

Par contre si c'est le nombre des pattes allouées au routage par circuit qui est considéré comme constant, l'avantage va aux grilles de grande dimension car le diamètre du réseau est plus faible [AP91b].

On constate que chacune des 2 argumentations précédentes part d'hypothèses raisonnables et arrive à une conclusion contraire ce qui laisse une grande marge de manœuvre à l'esprit... En fait il faut aussi considérer les mécanismes nécessaires pour éviter les interblocages (voir §9.2.4), ce qui n'est fait dans aucune de ces études. En effet, il faut souvent rajouter des tampons dont le nombre croît avec la dimension du tore et déplace donc le choix vers les faibles dimensions. Quoi qu'il en soit on ne peut guère donner de solution miracle à ce sujet.

La comparaison avec un réseau de type hypercube est intéressante. Si on considère un réseau à  $N = 256$  nœuds, l'hypercube possèdera un diamètre moyen de 4 ( $n = 8$ ), un tore 3D  $8 \times 8 \times 4$  un diamètre moyen de 5 et la même grille un diamètre de 10, sachant qu'à bisection constante on peut doubler son débit pour compenser le diamètre.

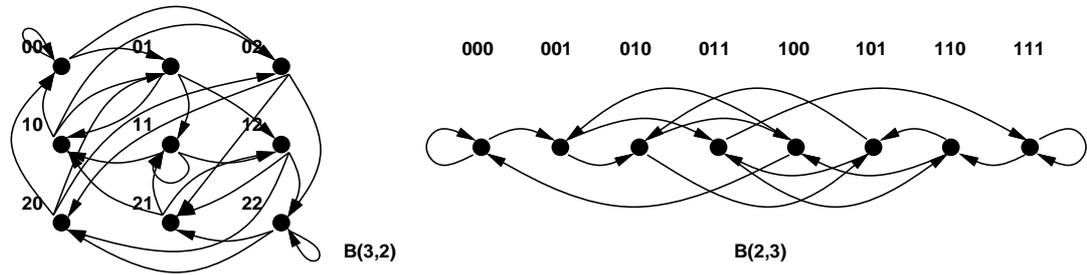
Il semble donc que beaucoup d'ordinateurs actuels s'orientent dans cette voie : INTEL PARAGON (grille 2D), CRAY MPP (grille 3D) par exemple. En plus, on diminue le rapport nombre de signaux utiles au transport d'information sur nombre de signaux de contrôle.

En plus, l'extension du réseau se fait simplement, à dimension constante, sans que le nombre de liens par processeur augmente, par ajout d'hyperplans de processeurs à l'hypergrille ou l'hypercube, ce qui est très intéressant lorsqu'on veut faire évoluer la taille d'une machine.

Enfin, parmi les cas particulier des tores, on retrouve en dimension 1 l'anneau et l'hypercube lorsque le nombre de nœuds suivant chaque dimension est 2.

---

5. On peut voir là une raison à l'existence de la notion de communication pipelinée `xnetp` en MPL pour la machine MP-1 [?] qui possède un réseau en grille 2D.

FIG. 9.5 - Exemple de graphes de DE BRUÏN,  $B(3,2)$  et  $B(2,3)$ .

### 9.2.1.4 Graphes de De Bruijn et de Kautz

Il possèdent la particularité d'être proche de l'optimalité en terme de diamètre et de degré par rapport au nombre de sommets (critère de MOORE) [BP89].

Le graphe de DE BRUÏN  $B(d, D)$  est construit en nommant chaque sommet avec des mots de  $D$  lettres choisies dans un alphabet de  $d$  lettres. Il possède donc  $d^D$  sommets.

Un sommet  $(x_1, x_2, \dots, x_D)$  est relié à tous les sommets  $(x_2, \dots, x_D, \alpha)$ , où  $\alpha$  est n'importe quelle lettre de l'alphabet. Le degré du graphe est par conséquent  $d$ .

On constate alors que le routage consiste à rajouter à chaque étape une lettre de la destination, prise successivement de gauche à droite, pour qu'en  $D$  cycles l'adresse de l'expéditeur se soit transformée en adresse de destinataire.

En ce qui concerne l'émulation de grilles, le problème n'est pas résolu dans le cas général, même s'il existe des cycles hamiltonniens (cycles couvrant l'ensemble de tous les sommets du graphe) permettant d'émuler un anneau.

Rendre le réseau bidirectionnel,  $UB(d, D)$ , revient à autoriser aussi de rajouter une lettre à gauche. Dans ce cas le degré devient  $2d$  mais le diamètre est inchangé. Si on ne réalise pas de routage subtil, cela n'apporte rien par rapport au réseau unidirectionnel dont on aurait par exemple doublé la largeur des liens pour avoir un débit équivalent. Par contre le réseau bidirectionnel augmentant le nombre de chemins possibles augmente la tolérance aux pannes.

Un cas particulier est le réseau *shuffle exchange* basé sur le « battage parfait<sup>6</sup> » *perfect shuffle* [Sto71],  $B(2, D)$ , comme  $B(2, 3)$  indiqué sur le dessin de droite de la figure 9.5. Ce réseau permet de réaliser facilement les mouvements de données utilisés pour calculer des FFTs sur une machine parallèle, faire des tris, des transpositions, etc. Le routage est le même par conséquent que celui dans un réseau de DE BRUÏN.

Comme on peut le voir sur la figure précédente, il existe des rebouclages sur les sommets dont le nom est composé de lettres identiques. Une méthode pour éviter ces liens inutiles est de reprendre la définition du graphe et d'empêcher d'avoir 2 lettres consécutives identiques : ce sont les graphes de KAUTZ qui possèdent des caractéristiques assez semblables.

Le principal inconvénient de ces graphes est que, même si on peut les construire récursivement, ils sont difficile à réaliser en pratique à cause de leur structure irrégulière.

6. Merci à [RJ83] de m'avoir fourni une traduction française. Le nom provient dans les 2 cas de l'analogie avec le battage d'un jeu de carte et l'entrelacement des cartes appartenant à chaque moitié qui en résulte.

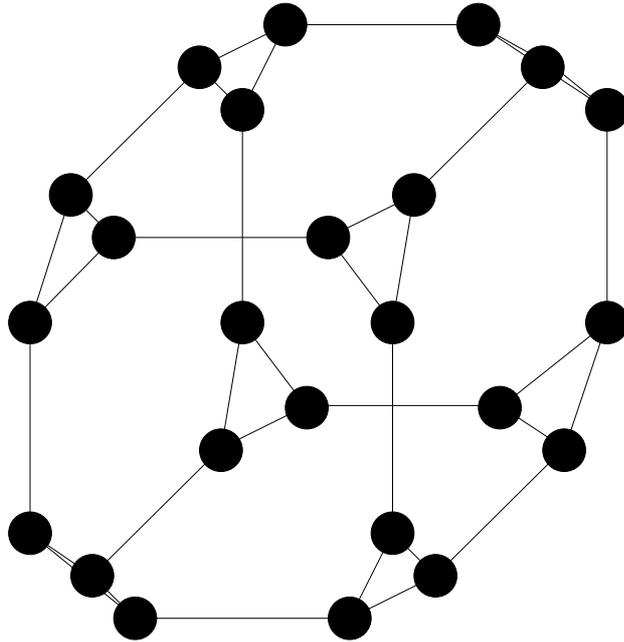


FIG. 9.6 - Réseau de type cube-connected cycles.

C'est pour cela que les réalisations se sont limitées au cas du *shuffle exchange*.

### 9.2.1.5 Autres graphes

Il existe d'autres graphes plus exotiques qui sont d'ailleurs souvent assez difficiles à dessiner. C'est un point important à noter, parce que ce qu'on a du mal à dessiner, en particulier de manière régulière, est encore plus difficile à réaliser !

Plusieurs approches à partir d'hybridations des réseaux précédents ont été faites.

Le *cube-connected cycles* (CCC) par exemple est un hypercube de dimension  $n$  dont on aurait remplacé chaque sommet par un anneau d'au moins  $n$  processeur (figure 9.6), ce qui fait un réseau d'au moins  $n2^n$  PES. L'intérêt est qu'on peut augmenter la dimension du réseau sans augmenter le nombre de liens par processeurs qui reste à 3 [PV81].

On peut citer le réseau *star graph* [AHK87] où chaque nom de PE est constitué d'une suite de  $n$  lettres différentes choisies parmi  $n$  lettres, ce qui forme un ensemble de  $n!$  processeurs. 2 PES sont reliés entre eux si leur noms sont identiques à un échange entre la 1<sup>ère</sup> lettre et une autre lettre près. Le degré est donc de  $n - 1$ . Ce réseau possède de bonnes propriétés de symétrie, de construction récursive (le graphe d'ordre  $n$  étant construit à partir de  $n$  graphes d'ordre  $n - 1$ ), un diamètre raisonnable de  $\lfloor \frac{3(n-1)}{2} \rfloor$ . Néanmoins il semble difficile de le réaliser dès que le nombre de processeurs est important et l'émulation de grilles passe par le routage général.

Enfin on peut citer les cas de graphes aléatoires (!), mais dans ce cas le routage des messages est assez difficile, ou de manière plus concevable, l'exploitation plus poussée des graphes de CAYLEY. Dans ce dernier cas on peut choisir au hasard un sous-ensemble générateur d'un groupe associé aux PES, où l'application de chaque élément du sous-ensemble à un numéro de PE indique le numéro de son voisin [CCD<sup>+</sup>92].

Au sujet de ces graphes exotiques, il peut survenir un problème de codage et de décodage d'adresse très compliqué et coûteux en temps de calcul (voire pouvant nécessiter de grosses tables de routage) s'il n'y a pas de relation simple entre un motif de communication très utilisé (par exemple une communication sur grille) et le réseau de la machine. Cela peut avoir comme conséquence une diminution du débit réel du réseau.

### 9.2.2 La commutation de circuits : un réseau vu dynamiquement

C'est la technique la plus ancienne, probablement parce qu'elle est la base des autocommutateurs téléphoniques dont l'étude et les réalisations remontent bien avant les débuts de l'informatique parallèle moderne et à cause de la commande qui ne demande localement que peu de matériel, voire pas du tout d'« intelligence » lorsque la commande est globale.

Elle consiste à voir le réseau comme un central téléphonique : d'un côté se trouve l'appelant et de l'autre côté l'appelé. Le réseau établit un chemin de communication entre les deux parties qui peuvent échanger des informations : il y a une « commutation de circuits » au début et à la fin de la phase de communication.

L'intérêt est qu'une fois le chemin établi, les communications sont faites de manière instantanée (au temps de propagation près) sans que les informations aient à se frayer un chemin à chaque fois.

L'inconvénient est que, puisque le chemin est constamment établi, cette occupation inutile de matériel peut empêcher l'ouverture de communications supplémentaires entre d'autres processeurs. Il y a un compromis entre le nombre de chemins, le nombre de commutateurs, la probabilité d'avoir des contentions. En effet, s'il existe plusieurs chemins possibles pour aller d'une entrée à une sortie et qu'une fraction d'un chemin est déjà utilisée par une autre communication, il est probable qu'on arrivera à trouver un autre chemin totalement libre et qu'on pourra établir la communication. En outre, le fait d'avoir plusieurs chemins est un argument de tolérance aux pannes.

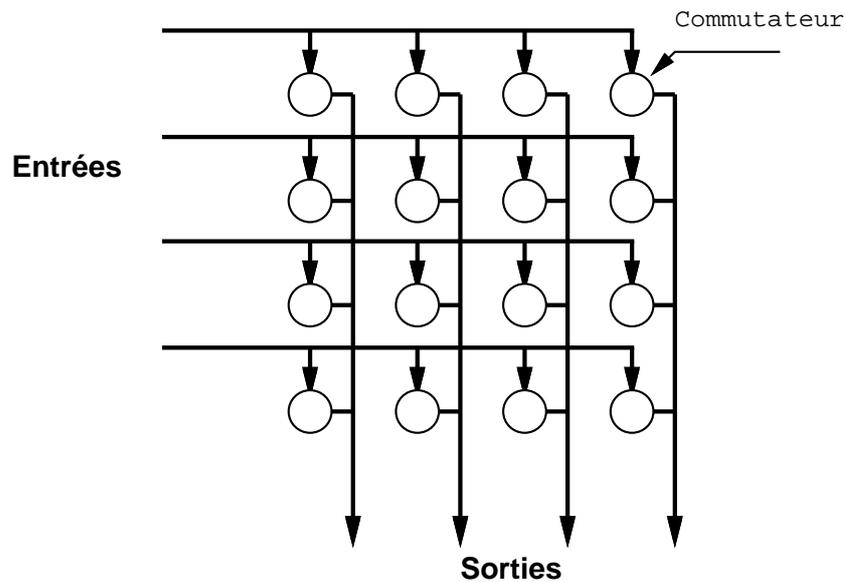
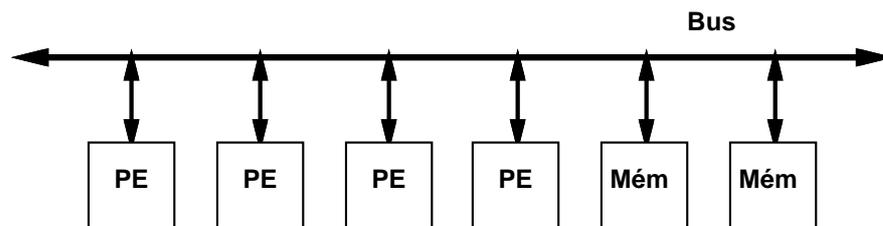
Tous les réseaux dynamiques étudiés sont bloquants au sens strict, c'est à dire qu'il existe des configurations de communications impossibles à réaliser, en particulier la communication de tous les PEs vers un seul. La différence entre les réseaux apparaît pour des configurations moins extrêmes comme les bijections : les réseaux peuvent être bloquants ou pas<sup>7</sup>, accepter seulement un certain type de configuration, etc. Il y a donc un compromis à trouver entre la complexité d'un réseau dynamique, sa latence, son taux de blocage, son degré de « réalisme », son rapport coût/performance.

Afin de réaliser un tel réseau, on part d'un réseau statique où on remplace une partie des processeurs par des circuits de routage : on déroule spatialement sur des routeurs l'algorithme de routage des messages qui s'exécutait séquentiellement (temporellement) sur les processeurs. C'est pourquoi on peut voir apparaître une dualité espace-temps entre les réseaux statiques et dynamiques.

#### 9.2.2.1 Le réseau dynamique complet : *crossbar*

---

7. C'est dans ce cadre qu'on emploie habituellement le terme « bloquant », lorsqu'une bijection ne peut passer en un cycle et que certains chemins ne peuvent pas être établis instantanément (ils sont bloqués).

FIG. 9.7 - *Synoptique d'un réseau à matrice de points de croisement (crossbar).*FIG. 9.8 - *Synoptique d'une machine communiquant à travers un bus.*

Le réseau de type commutateur à croisillons, à matrice de points de croisements ou plus simplement matriciel<sup>8</sup> offre l'avantage de permettre toutes les bijections possibles entre des processeurs et les messages n'ont qu'un commutateur à traverser pour aller d'une entrée à une sortie.

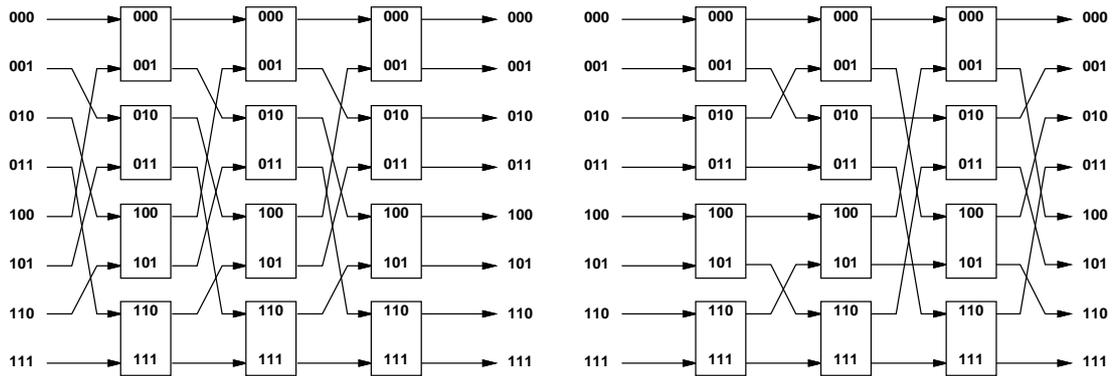
Il est utilisé dans le cas de machines à faible taux de parallélisme, telles que les machines vectorielles multiprocesseurs, car le réseau nécessite  $N^2$  commutateurs pour  $N$  processeurs (figure 9.7), ce qui est considérable lorsque les liens de communications sont larges pour des questions de débit et que le nombre de processeurs croît.

Par contre son contrôle est très simple, donc très rapide, et est bien adapté aux machines vectorielles rapides, qui ont peu de processeurs de toute manière.

### 9.2.2.2 Le bus global

A l'inverse du crossbar se trouve le bus global où tous les processeurs communiquent, en général avec une ou plusieurs mémoires globales, à travers un unique support de données, le « bus » (figure 9.8). Clairement, une seule communication peut être éta-

8. Plus connu sous le nom de *crossbar* en anglais.

FIG. 9.9 - Dessin d'un réseau Oméga et Indirect Binary  $n$ -Cube.

blie à la fois ce qui nécessite d'avoir un bus très large et très rapide si on veut pouvoir y connecter plusieurs processeurs. En fait, le bus se trouve entre la commutation de paquet et la commutation de circuits, selon le protocole utilisé : la séparation des requêtes et des réponses pour augmenter le débit le rapproche par exemple de la commutation de paquets.

Son principal intérêt est sa simplicité, son matériel minimal, en  $\mathcal{O}(1)$ <sup>9</sup>, le fait choisir lorsqu'on a un débit et un nombre de processeurs faibles.

C'est la raison pour laquelle ce système est réservé aux ordinateurs possédant un nombre restreint de processeurs, comme les stations de travail multiprocesseur ou les minisupercalculateurs à mémoire partagée du style ENCORE 90 [Enc91].

Des extensions ont été proposées afin de dépasser ces limitations par l'intermédiaire d'une hiérarchie de bus : une machine est constituée de sous-machines reliées par un bus, chaque sous machine étant elle-même constituée de processeurs reliés par un bus, la machine formant donc un arbre. Le problème principal est alors l'irrégularité qu'on rajoute dans les communications : il sera probablement plus rapide de communiquer entre des processeurs sur le même bus qu'entre processeurs de bus différents. Cette anisotropie est assez difficile à prendre en compte lorsqu'on doit générer du code optimisé pour la machine et le problème de gestion de la cohérence des caches croît d'un ordre de grandeur<sup>10</sup>.

### 9.2.2.3 Les réseaux de type Oméga

Afin de limiter le nombre de commutateurs, un autre approche permet d'avoir des réseaux ne demandant que  $\frac{N}{k} \log_k N$  commutateurs à  $k$  entrées et  $k$  sorties, à comparer aux  $N^2$  commutateurs du réseau à croisillons.

L'idée est de partir d'un réseau statique de type *shuffle exchange* par exemple ( $k = 2$ ) et au lieu de router les messages en  $\log_2 N$  passage à travers le réseau, on va mettre bout à bout  $\log_2 N$  réseaux [WF81], d'où le nom de réseau multiétage<sup>11</sup>. Ainsi

9. Il faut peut-être nuancer cette affirmation en considérant que dans le cas du bus, les barrières d'interface entre les processeurs et le bus jouent le rôle de commutateurs et dans ce cas la complexité serait plutôt en  $\mathcal{O}(N)$ , avec une seule liaison.

10. Ce sont ces mêmes problèmes que l'on retrouve dans la machine CEDAR qui utilise une hiérarchie à base de réseaux [GLK84].

11. Aussi connu sous le nom de MIN: *Multistage Interconnection Network*.

un message arrivera à destination après traversée de tous les étages du réseau Oméga [Law75] (figure de gauche de 9.9).

Notons au passage qu'on optimise ainsi le cas pire. En effet il se peut qu'un message ait pu atteindre sa destination en moins de  $\log_2 N$  passages dans un réseau *shuffle exchange*. Dans le cas du MIN, il est obligé de traverser tout de même tous les étages.

Evidemment, le réseau est plus bloquant que le *crossbar* puisqu'il y a un chemin unique possible pour aller d'un point à un autre et que plusieurs communications peuvent utiliser un certain nombre de liens identiques, sources de conflit. Il faut alors séquentialiser certaines communications pour réaliser la communication complète. Cela suffit néanmoins dans de nombreuses applications<sup>12</sup>.

Un des grands avantages de cette classe de réseaux est que le routage est simple, basé sur le routage dans les graphes de DE BRUÛN, le *destination tag algorithm* (DTA): chaque bit de l'adresse de destination sert à choisir quel chemin sera pris au niveau de chaque commutateur [Law75]. Le routage se fait sur des critères seulement locaux, ce qui est intéressant puisque nul routeur centralisé n'est nécessaire. Le commutateur a donc une structure très simple et n'a pas besoin de mémorisation, ce qui est primordial lorsqu'on n'a pas beaucoup de ressources à sa disposition.

D'autres motifs ont été utilisés pour relier les commutateurs. On peut citer l'*indirect binary n-cube* qui utilise successivement les dimensions d'un hypercube et termine par un battage parfait [Pea77] (figure de droite de 9.9), le réseau FLIP de la machine STARAN [Bat76] qui est un réseau Oméga à l'envers, le réseau Delta [Pat81] ou encore plus généralement les réseaux de type banian<sup>13</sup> [GL73].

Malheureusement pour tous ceux qui avaient inventé des réseaux « différents » de ce type, il a été montré qu'ils étaient équivalents [WF80, BF88]... Néanmoins les considérations à prendre en compte sont liées à la réalisation: le câblage de tel ou tel réseau est-il plus simple, est-il possible de bien partitionner le réseau pour le mettre sur différents circuits intégrés? En ce sens ils sont chacun différents.

Une variation intéressante d'un réseaux de type *indirect binary n-cube* est de factoriser progressivement à chaque étage (*ie* récursivement chaque dimension de l'hypercube) en réduisant le nombre de liens afin de faire un réseau de type *fat-tree* [Lei85]. Cela permet d'économiser du matériel tout en gardant des performances raisonnable. Ce réseau est utilisé dans la CM-5 [Thi91].

#### 9.2.2.4 Clos

Bien avant, des réseaux basés de même sur un empilement de couches de *crossbars* mais étant non bloquants et permettant toutes les bijections possibles ont été étudiés. Un bon exemple est le réseau de CLOS [Clo53], qui n'a pas besoin d'être réarrangé. La figure 9.10 montre la configuration à 3 étages. Les autres configurations s'obtiennent en remplaçant les commutateurs de l'étage central récursivement par des réseaux de CLOS plus petits.

L'idée de base était de limiter le nombre de commutateurs par rapport au réseau à matrice de points de croisement out en ayant les mêmes performances et sans avoir à

---

12. Ne serait-ce que des applications de type téléphone où on table sur le fait qu'il n'y a jamais plus de 5% des abonnés téléphoniques qui téléphonent en même temps.

13. Du nom du figuier indien dont la forme se rapproche. La traduction anglaise en est *banyan*.

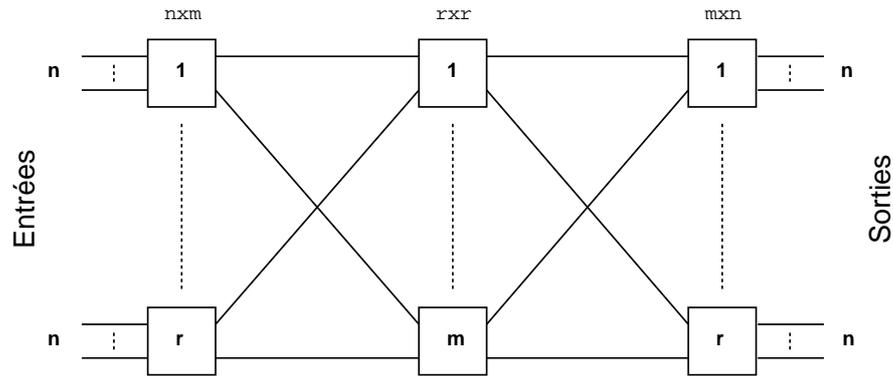


FIG. 9.10 - Synoptique d'un réseau de CLOS à 3 étages.

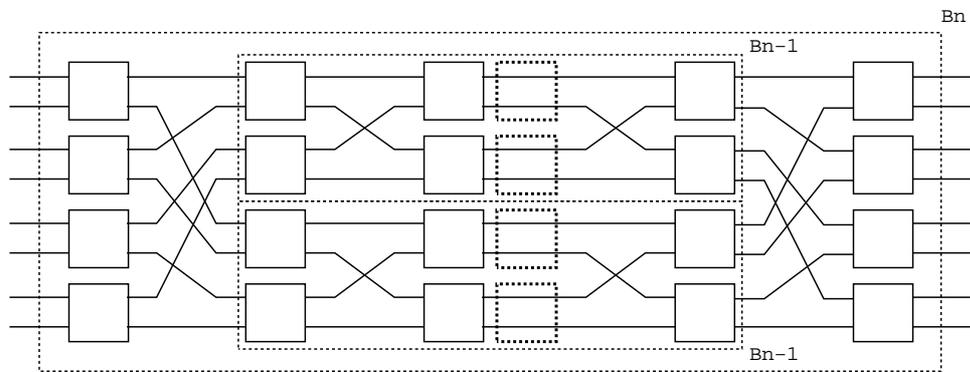


FIG. 9.11 - Synoptique d'un réseau de BENEŠ.

modifier les liaisons déjà établies<sup>14</sup>. Cela est possible lorsque  $m \geq 2n - 1$ .

Un réseau à 3 étages possède  $Nm(2 + \frac{1}{n})$  points de croisement, soit compte tenu de la contrainte sur  $m$  au moins  $N^2(\frac{4}{r} + \frac{2}{rn}) + N(\frac{1}{n} - 2)$  points de croisement à comparer à  $N^2$  pour le commutateur à croisillons.

Le gros problème de ce réseau est qu'il est difficile à configurer globalement et impossible à configurer localement, ce qui lui enlève beaucoup de son intérêt.

### 9.2.2.5 Beneš

Afin de simplifier le réseau précédent lorsque  $n$  et  $r \times n$  sont des puissance de 2 et  $m = 2$ , on remplace les 2 commutateurs du milieu par 2 réseaux de CLOS, et ce récursivement jusqu'à ne plus avoir que des commutateurs  $2 \times 2$ .

Le réseau est devenu seulement réarrangeable puisque la relation  $m \geq 2n - 1$  n'est plus respectée : on compense le nombre moindre de chemins par une nécessité de réarranger les chemins déjà alloués pour permettre l'établissement de nouvelles connexions. C'est le réseau de BENEŠ [Ben62], qui est construit en fait comme un réseau de type Oméga suivi d'un autre à l'envers, un réseau FLIP, comme dessiné de manière un peu

14. Comprendre : « sans interrompre les communications téléphoniques en cours », vue l'application visée.

différente sur la figure 9.11 pour mieux montrer la structure récursive. Il nécessite donc  $\frac{N}{2}(2 \log_2 N - 1)$  commutateurs<sup>15</sup>.

Le problème est qu'il n'y a pas d'algorithme local connu de routage de toutes les bijections et on est obligé de faire le calcul globalement en un temps  $\mathcal{O}(N \log N)$  ou en parallèle en  $\mathcal{O}(\log^2 N)$  [KO90]. Si on peut se permettre de faire attendre une personne au téléphone le temps qu'un ordinateur réarrange le réseau pour le mettre en communication avec son correspondant, cela semble difficile à mettre en œuvre dans le réseau d'un ordinateur parallèle.

Des algorithmes ont néanmoins été trouvés qui s'appliquent aux permutations courantes [Len78, BR88] ou pour un réseau de BENEŠ amélioré [KO90].

Dans la machine OPSILA, un réseau de BENEŠ pouvait être simulé en faisant traverser les données 2 fois dans le réseau Oméga de la machine [Gou82, LM85]. Comme les données devaient traverser une fois dans un sens et une fois dans le sens contraire, cela rend difficile une utilisation pipeliné de ce réseau.

Dans la machine GF11 [BDW85], la configuration du réseau est faite globalement et le routage totalement aléatoire est à la charge du programmeur. Néanmoins cela trouve sa justification dans le fait que beaucoup de communications des applications visées par GF11 possèdent des motifs pouvant être connus à la compilation et donc la programmation du réseau précompilée peut être faite pour chaque phase de calcul.

### 9.2.2.6 Autres réseaux dynamiques

On peut partir d'autres réseaux statiques pour construire des réseaux dynamiques possédant d'autres propriétés.

Par exemple le réseau dynamique construit à partir d'un hypercube [Szy89] ou le réseau *data manipulating functions* (DMF) [Fen74] basé sur une série de décalage.

Mais globalement ils ont des caractéristiques semblables à ceux présentés précédents.

Un autre réseau utilisable est le réseau de tri de BATCHER [Bat68] possédant  $\frac{N}{4} \log^3 N$  commutateurs élémentaires : il suffit d'utiliser l'adresse de destination comme clé de tri pour que les messages arrivent à bon port. Il a l'intérêt d'être non-bloquant, même s'il n'est pas optimal en nombre de commutateurs, et d'être « auto-routant », avec un algorithme similaire au DTA.

On peut aussi mélanger le réseau de BENEŠ et de BATCHER afin de faire mieux en nombre de commutateurs tout en étant routable au niveau commutateur comme le réseau de BATCHER [KO90].

### 9.2.3 Les techniques intermédiaires de propagation

Une tentation logique est d'essayer de récupérer les avantages de chacune des deux classes de réseaux précédentes :

- avoir ceux de la commutation de paquets dans le cas des petits messages (où établir un chemin revient à propager une adresse qui est suivie en même temps par une données), intéressant pour les communications aléatoires ;

---

15. Et non pas  $\frac{N}{2}(2 \log_2 N)$  puisque les deux colonnes de commutateurs du milieu sont reliées directement, une des deux est redondante.

- ceux de la commutation de circuits pour les gros paquets (une fois le circuit établi ce n'est plus la peine de propager une adresse et on peut envoyer autant de données que l'on veut), ce qui est intéressant pour des communications suivant des motifs réguliers.

### 9.2.3.1 La méthode de propagation *virtual cut-through*

La méthode provient directement de la synthèse des méthodes précédentes. Dans le mécanisme classique de commutation de paquets, le *store and forward*, chaque nœud attend d'avoir reçu un paquet avant de l'envoyer vers le destinataire. On comparera cette méthode avec ce qui se passe par exemple sur les réseaux BITNET et EARN<sup>16</sup>.

Pour autant, si on envoie l'adresse en tête du paquet, le processeur qui reçoit le paquet est capable de commencer à le router dès qu'il a l'adresse et ainsi pipeliner en partie la réception et la réémission des portions de paquets.

Lorsqu'un message ne peut plus avancer, il est stocké sur le dernier nœud atteint, libérant ainsi la place sur les liens du réseau [KK79].

Ce qu'il y a de remarquable dans cette technique est le pipeline qui fait que :

- lorsque le réseau est peu chargé il se comporte comme s'il était à commutation de circuit : les données suivent l'adresse qui établit un chemin évanescent sur son passage et les paquets se propagent sans encombre. On fait de la commutation de paquets avec une vitesse de propagation des données comparable à celle de la commutation de circuits ;
- lorsque le réseau est très chargé, les paquets sont gênés dans leur avancement et progressent par à-coup, l'effet pipeline a disparu : tout se passe comme si le réseau était à commutation de paquets avec du *store and forward*. En plus ce mécanisme utilise mieux le réseau car ce n'est pas la peine d'ouvrir un chemin pour l'envoi d'un seul paquet qui aurait pour effet de bloquer beaucoup de ressources le temps du transfert de ce paquet.

### 9.2.3.2 La méthode de propagation *wormhole*

C'est surtout une simplification de la méthode précédente qui a été mise en pratique, par exemple dans les machines SYMULT 2010, nCUBE-2, IWARP et INTEL TOUCHSTONE.

Cette méthode se retrouve aussi au niveau du réseau INTERNET ou FNet sous TCP/IP entre ordinateurs.

Lorsqu'il y a une congestion du réseau, certaines portions de paquets<sup>17</sup> peuvent s'arrêter d'avancer. Plutôt que de stocker tout le message sur le nœud où a lieu le conflit comme dans la méthode précédente, chaque portion de message est stockée là où elle était dans le réseau, le message est donc bloqué et stocké de manière distribuée [?, DS87].

On constate qu'il y a un blocage de plusieurs circuits, un peu comme dans le cas de la commutation de circuit, puisque les morceaux de messages sont stockés généralement *in situ* dans le système de transfert des messages.

16. L'unité de transfert est le fichier : on ne route un fichier ou une lettre électronique (fichier d'un type spécial interprété comme du courrier) que lorsqu'il a été complètement reçu, ce qui nécessite une zone de stockage conséquente au niveau de chaque nœud.

17. On trouve le terme « *flit* » dans la littérature [DS86].

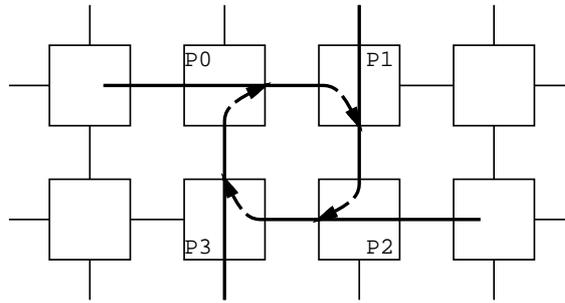


FIG. 9.12 - Exemple d'interblocage.

La technique est donc moins efficace (la congestion apparaît plus rapidement) mais elle nécessite beaucoup moins de mémoire de stockage et est économiquement plus intéressante.

Des améliorations simples ont néanmoins été proposées, comme une duplication des tampons qui permet à des paquets de doubler des messages bloqués à la manière des voitures sur les voies rapides. On crée donc plusieurs canaux virtuels par lien physique et lorsqu'un message est bloqué sur un canal virtuel, il y en a souvent un autre qui est libre pour faire passer un autre paquet. Le débit d'un réseau 2D peut ainsi être augmenté d'un facteur 4 par rapport au simple *wormhole* [Dal92].

#### 9.2.4 Quelques techniques de routage

Dans un réseau à commutation de circuits, la gestion de conflits se fait lors de l'ouverture d'un circuit : si le réseau est bloquant et qu'il y a un conflit, l'ouverture d'un circuit peut être retardée mais on est assuré d'avoir un certain nombre de requêtes qui aboutissent.

Par contre, dans le cas d'un réseau qu'il soit à commutation de paquets (*store and forward*), *wormhole* ou *virtual cut-through*, les conflits ne peuvent être gérés qu'au coup par coup lorsque deux paquets demandent à prendre un même lien ou qu'il y a création de cycles de dépendance, comme indiqué sur la figure 9.12 : chaque paquet attend qu'un autre avance et ainsi de suite circulairement. Si aucun paquet ne peut avancer, c'est un cas d'interblocage ou d'étreinte mortelle (*deadlock*) [MS80].

Pour éviter qu'il puisse apparaître de tels cycles de dépendance entre des ressources du réseau (liens, tampons), plusieurs techniques ont été étudiées, s'appliquant plus ou moins bien à une réalisation concrète, et que nous allons décrire maintenant car les méthodes utilisées pour éviter les interblocage conditionnent directement les performances et les coûts des réseaux statiques.

##### 9.2.4.1 Les algorithmes déterministes

L'avantage des algorithmes déterministes est qu'ils sont simples à mettre en œuvre : tout paquet voulant aller d'un point  $\mathcal{A}$  à un point  $\mathcal{B}$  passera toujours par le même chemin. Dans ce cas, il suffit de s'arranger pour que les chemins formés n'engendrent aucun cycle de dépendance.

L'algorithme le plus simple pour les réseaux de type grille est de faire avancer tous les messages dimension par dimension, une fois que plus aucun paquet n'a plus besoin

d'avancer pour atteindre son but dans la dimension courante on passe à la dimension suivante. Il est clair que lorsque les paquets voyagent suivant une dimension, seuls les liens appartenant à cette dimension sont utilisés, en plus de la non-adaptabilité du système qui augmente les conflits : c'est un gâchis considérable.

Afin d'améliorer cet algorithme, on a autorisé les paquets à voyager suivant les dimensions croissantes, mais les changements de dimension se faisant de manière asynchrone par rapport aux autres paquets. C'est l'algorithme du *e-cube* utilisé parfois pour contrôler les paquets dans les hypercubes. Mais alors, il peut y avoir des cycles et afin d'éviter cela on introduit la notion de *canaux virtuels* [DS87]. Par exemple chaque canal physique d'une grille est divisé en 2 canaux virtuels pour casser les cycles possibles.

Le corollaire immédiat est donc que si on impose à des paquets de passer toujours par le même chemin, les autres chemins possibles sont inutilisés et en cas de congestion du réseau ce manque d'adaptabilité se traduira par une baisse de performance. Par contre le mécanisme de routage non interbloquant est simple, ce qui explique qu'on le retrouve dans la machine INTEL DELTA [Int91c] par exemple.

Une extension à l'algorithme du *e-cube* vient de la remarque que ce dernier revient à empêcher les cycles en n'autorisant pas la moitié des « tournants » possibles (par exemple  $P_1 \xrightarrow{P_2} P_3$  et  $P_3 \xrightarrow{P_0} P_1$  sur la figure 9.12 dans le cas d'un routage en  $x$  puis  $y$ ). Cette contrainte est un peu forte puisqu'il suffit de supprimer un tournant par cycle possible dans chaque dimension. C'est le *turn model* qui permet d'éviter les interblocages simplement et peut même avoir un certain degré d'adaptativité [GN92] et développe (sans le savoir) les idées de [ML89].

#### 9.2.4.2 Routage forcé et routage aléatoire

Il s'agit d'une méthode de routage assez simple mais efficace dans certains cas, compte tenu des avantages :

- le nombre de tampons est très limité puisqu'il n'y en a plus qu'un par lien entrant, ce qui est intéressant lorsqu'on veut mettre le circuit de routage sur une petite surface de circuit intégré [GR89] ;
- il s'adapte sur n'importe quel réseau ayant autant de liens entrant que sortant ;
- comme son nom l'indique, le routage est forcé et donc aucun signal de régulation de flot entre les nœuds du réseau n'est nécessaire.

Pour le mettre en œuvre, il suffit de garantir que lorsqu'un paquet arrive, on puisse toujours en faire quelque chose.

Lorsque le processeur du nœud ne veut pas émettre, il n'y a pas de problème : si un des paquets arrivés dans un des tampons est à destination et peut être accepté par le processeur, il est alors retiré du réseau. Pour les autres paquets, on route les messages en essayant de les rapprocher de leur destination, sinon on les éloigne de leur destination en les envoyant sur des liens choisis de manière aléatoire.

Si le processeur relié au réseau veut émettre, il suffit de le faire lorsqu'il y a au moins un tampon de réception de libre. Ainsi, si l'on considère que les autres tampons arriveront à se vider bien qu'il y ait déjà un lien de sortie occupé, on pourra bien accepter un message de chaque lien entrant. Tout ceci suppose des vitesses d'émission et de réception identiques bien entendu.

Il n'y a pas d'interblocage statique puisqu'on a vu que quand un paquet arrivait on pouvait toujours en faire quelque chose. Par contre il faut éviter qu'un message soit « satellisé » dans le réseau et n'atteigne jamais son but parce que constamment détourné de celui-ci à cause de l'engorgement du réseau (problème de *livelock*). Une méthode simple est de dérouter les messages au hasard lorsqu'il ne peuvent se rapprocher afin que le mouvement stochastique finisse par le rapprocher de son but<sup>18</sup> [KS91a, KS91b].

Une utilisation originale de ce système serait de l'associer avec des réseaux construits sur des liaisons série à haut débit telles que dans [LEH<sup>+</sup>89]. En effet, il est souvent difficile d'acheminer sur ces liaisons des messages de régulation sans perdre de la bande passante et avoir un temps de latence important lorsque cela n'a pas été prévu. Dans le cas d'un réseau surchargé, la vitesse de transmission des paquets sera en fait limitée par le temps de propagation de tous les messages de contrôles de flot. Il faut alors faire une étude comparative sur réseau surchargé entre la bande passante perdue par les messages de régulation et la perte liée à l'allongement des chemins et la diminution du taux d'injection dans le réseau. A faible charge, les deux méthodes se valent puisqu'il n'y a pratiquement pas de conflits, donc peu de messages de régulation et peu d'éloignements. Mais cela nécessite une simulation plus poussée dans chaque cas, car cela dépend beaucoup de la taille et de la topologie du réseau.

### 9.2.4.3 Routage non déterministe

Les algorithmes les plus prometteurs sont ceux qui ne sont pas déterministes puisqu'on peut choisir entre plusieurs chemins, ils sont donc adaptatifs, et sont optimaux lorsque l'algorithme est tel qu'un paquet ne peut que se rapprocher de sa destination, contrairement au routage forcé.

L'idée de base pour faire un routage non-interbloquant *virtual cut-through* ou *worm-hole* est de classer chaque paquet suivant l'octant (si on est dans une grille de dimension 3 par exemple) auquel appartient son vecteur de déplacement. En attribuant autant de canaux virtuels par canal physique qu'il y a de classes et en faisant circuler chaque classe uniquement dans les canaux virtuels correspondants qui forment donc des réseaux virtuels disjoints, on supprime les cycles de dépendance [LH91, FGPS91]. Il s'agit donc d'une généralisation des canaux virtuels de [DS87]. En ce qui concerne les tores, on est obligé de rajouter des canaux virtuels pour gérer le fait que le réseau est torique et qu'on peut avoir des cycles au niveau des canaux virtuels correspondants.

L'adaptativité permet une meilleure utilisation des chemins disponibles qui augmente le débit efficace du réseau, tout particulièrement lorsqu'il est chargé. En effet soit une machine à  $N$  PES sous forme d'une hypergrille de dimension  $n$ . Soit un paquet devant parcourir dans chaque dimension une distance de  $\Delta x_i$ . Le nombre de chemins de distance minimale  $\sum_{i=1}^n \Delta x_i$  est :

$$c_{\Delta x_1, \dots, \Delta x_n} = \frac{(\sum_{i=1}^n \Delta x_i)!}{\prod_{i=1}^n (\Delta x_i)!}$$

en considérant que le problème est équivalent au nombre de combinaisons possibles  $\sum_{i=1}^n \Delta x_i$  billes d'un ensemble de  $n$  paquet de  $\Delta x_i$  billes, chaque paquet possédant des

---

18. En fait c'est une conséquence d'un théorème de DIRAC qui permet de dire qu'au bout d'un temps au plus infini (!) le routage terminera.

billes de couleur propre. Le nombre de chemins possibles est donc en général considérable comparé au chemin unique d'un routage déterministe, ce qui a pour effet d'augmenter la résistance à la congestion et par là le débit.

Par contre on constate que le nombre de canaux virtuels est  $2^n$  pour les hypergrille de dimension  $n$  et  $n2^n$  dans le cas des hypertores. Comme chaque canal virtuel est associé à un tampon, plus la dimension du réseau est importante et plus le nombre de transistors nécessaires aux tampons est important, ce qui peut poser des problèmes d'intégration dès la dimension 3 à l'heure actuelle. Une simplification est de réduire l'adaptativité à certains plans, ce qui a pour effet à se ramener au nombre de canaux virtuels nécessaires pour une grille 2D puisque les problèmes d'interblocage n'apparaissent plus que dans des plans successifs [CK92].

Les réseaux de type hypergrille de dimension plus élevée sont intéressantes dans la mesure où elles offrent plus d'adaptativité en proposant plus de chemins minimaux. Malheureusement, l'adaptativité complète sans interblocage est clairement une limitation que fixe la technologie sur la dimension des réseaux à commutation de paquets non-interbloquants, en plus des considérations sur les bisections [?] et le nombre de pattes [AP91b] déjà évoquées en 9.2.1.3. Il s'agit là d'un autre argument en faveur des réseaux à faible dimension et des grilles.

### 9.3 Réseau de POMP : hybride statique et dynamique

Avant de choisir un réseau pour POMP il faut rappeler les besoins de la machine.

En présentant le modèle de programmation et le langage on a vu qu'on désirait bien entendu des communications aléatoires mais aussi, si possible, des communications régulières rapides, basées sur des relations de voisinage, car il s'agit de communications souvent utilisées.

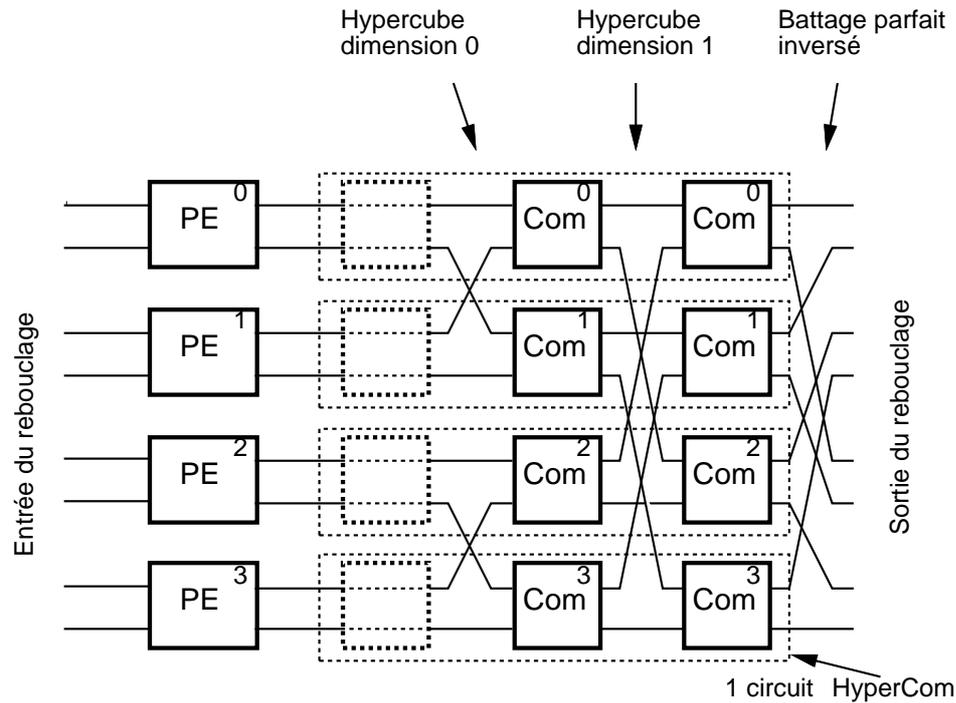
La machine étant SIMD, un réseau à fonctionnement synchrone et à temps de réponse déterministe semble aussi important.

La section ?? a présenté des compromis entre réseau statique et réseau dynamique liés au routage. Nous allons proposer une approche basée sur un réseau possédant 2 modes de fonctionnement : un réseau dynamique pour les communications aléatoires et un réseau statique pour les communications de type grille, qui répond à nos besoins à un coût raisonnable.

#### 9.3.1 Principe

On a vu que le fait d'avoir un réseau dynamique implique un réseau sous-jacent statique. On pourrait avoir l'idée d'être capable d'utiliser ce réseau statique. L'avantage est que, si la topologie du réseau statique est intéressante en soit, on peut l'utiliser directement sans avoir à prendre du temps pour reconfigurer le réseau dynamique. Par contre le réseau dynamique peut être toujours utilisé pour réaliser les communications plus générales tout en conservant une structure très simple au niveau de chaque commutateur.

On peut essayer de récupérer les réseaux statiques qui relient chaque colonne de commutateurs dans un réseau de type *indirect binary n-cube* [Pea77] par exemple : le réseau est simple et basé sur un hypercube et un *shuffle* inversé pour le dernier étage [Kot87], très utile pour les communications sur grille et les FFTs (*shuffle* normal).

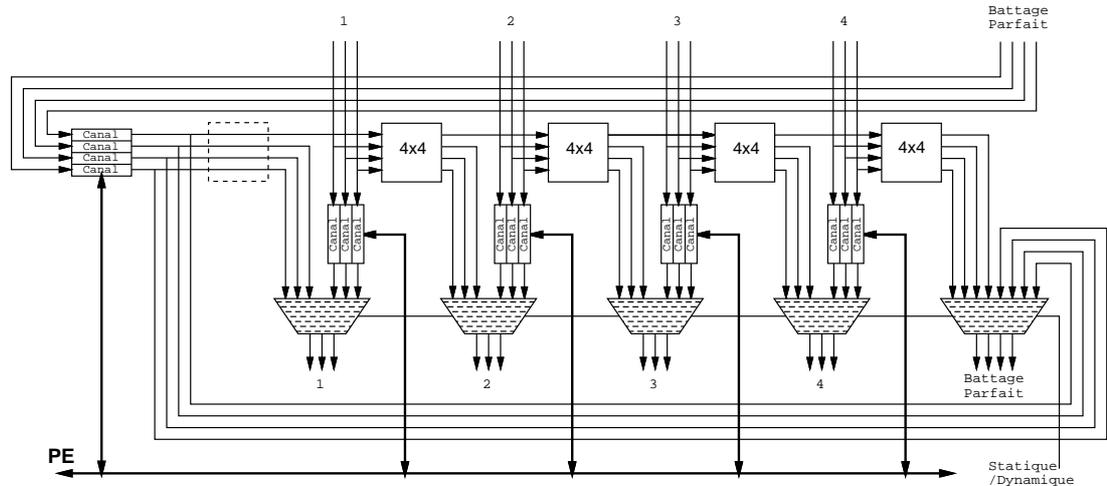
FIG. 9.13 - Synoptique du réseau de POMP pour  $N = 4$  et  $k = 2$ .

Malheureusement chaque colonne n'a que  $N/2$  commutateurs alors qu'on a une machine à  $N$  processeurs (figure 9.13). L'idée est de doubler la taille du réseau pour avoir autant de commutateurs par colonne que de processeurs. Chaque processeur est relié au réseau par 2 liens, ce qui n'est pas gênant car comme on se place *a priori* dans le cas d'une machine à parallélisme massif des données, il est très probable que ces 2 liens seront utilisés constamment pendant les communications globales par 2 communications simultanées appartenant à des processeurs virtuels différents. Ce genre d'extension de réseau n'est pas nouvelle puisque le réseau *Extra Stage Cube* [GBAS82] rajoutait un étage de commutateur pour augmenter la tolérance aux pannes. Mais dans notre cas on augmente la largeur du réseau et non sa profondeur.

Alors que dans le cas d'un réseau dynamique normal on ne peut effectuer qu'un accès à une dimension de l'hypercube à la fois, dans le mode statique on peut accéder à toutes les dimensions à la fois avec un débit global multiplié d'un facteur  $\log_2 N + 1$  ou  $\frac{\log_2 N + 1}{2}$  par rapport au réseau dynamique modifié puisque celui-ci a 2 liens par processeur. Le battage parfait inversé du réseau dynamique peut alors aussi être vu comme un battage parfait normal puisqu'on peut très bien renverser le sens de parcours des données.

Un des autres intérêts du réseau est qu'on peut s'arranger pour mettre tous les commutateurs d'une ligne dans un même **HyperCom**. Comme cela les fils directs restent dans l'**HyperCom** et ne gaspillent pas de la filasse utile pour le réseau statique. Dans la même approche on mettra chaque circuit **HyperCom** à côté du processeur auquel il est connecté.

De plus, une fois qu'on a configuré le réseau statique, les fils de contrôle parallèles aux fils de données qui servaient à l'établissement des chemins de données sont devenus inutiles (réseau statique) et par conséquent peuvent être réutilisés comme fils de

FIG. 9.14 - Synoptique du réseau de POMP pour  $N = 256$  et  $k = 4$ .

données supplémentaires.

Le choix d'avoir un réseau très simple au niveau de chaque commutateur et en particulier ne pas avoir de tampon a été guidé par l'apparition récente sur le marché de circuits intégrés de logique reprogrammable [XIL90].

Si les circuits logiques programmables par mémoire statique ne sont pas vraiment récents, le fait que l'on puisse aussi utiliser de la mémoire de configuration du circuit est nouveau. Ce qui manquait cruellement dans les anciennes versions était la mémoire. Cela est paradoxal dans un circuit qui en contenait beaucoup pour la programmation ! Par conséquent on peut réaliser des tampons avec cette mémoire de manière efficace et donc transformer le circuit programmable en routeur pourvu qu'il soit suffisamment simple. C'est ce qui a justifié, avec la facilité de réalisation et le synchronisme inhérent, le choix d'un réseau à commutation de circuits. Le fait qu'il soit utilisable aussi en tant que réseau statique ne complique guère la conception du circuit.

On peut généraliser le réseau en disant qu'il est basé sur un hypercube en base  $k$  plutôt qu'en base 2, avec tout de même  $k$  puissance entière de 2, ce qui permet de réaliser des réseaux comme celui de la figure 9.14. Le choix du mode se fait par l'intermédiaire des 5 multiplexeurs qui permettent de choisir ce qu'on met aux sorties du circuit : des commutateurs à croisillons pour faire un réseau dynamique (dessin de gauche de la figure 9.15) ou bien des registres à décalage pour contrôler directement le réseau statique (dessin de droite de la figure 9.15).

Lorsqu'on choisit le mode dynamique du réseau ci-dessus, le premier niveau de routage est fait logicielllement par le choix de l'hypercanal dans lequel on écrit. Ce temps de calcul n'est pas négligeable pour le processeur dans la routine de gestion des communications et il est possible de la supprimer en rajoutant un niveau supplémentaire de commutateurs matriciels juste après les hypercanaux, comme ceux indiqués en pointillé sur les figures 9.13 et 9.14.

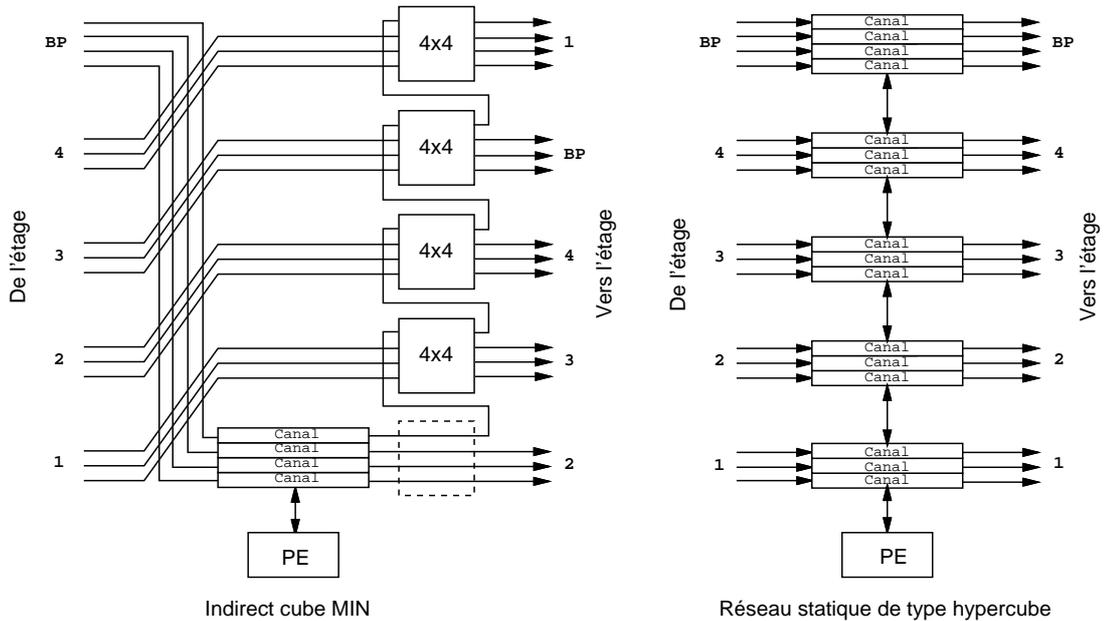


FIG. 9.15 - Les 2 modes du réseau de POMP pour  $N = 256$  et  $k = 4$ .

### 9.3.2 Complexité

#### 9.3.2.1 Fils de données et de contrôle

Le nombre  $\mathcal{F}$  de fils nécessaires pour un lien est la somme du nombre de fils de données  $f_d$  et du nombre de fils de contrôle  $f_c$ . Il est clair qu'on a intérêt à avoir  $f_d$  assez grand si on veut que  $f_c$  soit négligeable dans le coût du câblage. Evidemment, c'est un vœux pieux... Typiquement  $f_c = 2$ , un fil pour indiquer à l'étage suivant qu'on envoie une adresse de routage et un fil de retour pour dire s'il y a conflit ou non dans l'établissement du chemin, ou encore  $f_c = 1$  si on utilise un bit de « start ». Comme on veut aller vite, l'envoi des données avant d'avoir eu une réponse positive d'acceptation implique qu'on ne peut pas utiliser les fils de données retournés à la place de fils de contrôle.

#### 9.3.2.2 Complexité du câblage

Soit  $k$  le nombre d'entrées des commutateurs du réseau, chaque entrée ayant donc  $\mathcal{F}$  fils. Un réseau multi-étage classique pour  $N$  processeurs ayant  $\frac{N}{k} \log_k N$  (section 9.2.2.3), on a besoin dans notre réseau de  $N \log_k N$  commutateurs, ou encore  $N(\log_k N + 1)$  si on a la colonne de commutateurs pour accélérer le travail des PEs. Dans le premier cas on augmente la largeur du réseau et non sa profondeur ce qui fait que si la complexité du réseau a bien augmenté, sa latence (*ie* le nombre d'étages à traverser) est restée constante. La complexité d'un commutateur à croisillons est identique à celle du réseau *indirect binary n-cube* de départ est reste donc de  $f_d k^2$ . Afin de simplifier les notations, on pose le nombre d'étages du réseau  $\mathcal{E} = \log_k N$ . Il en découle qu'une ligne de  $\mathcal{E}$  commutateurs à croisillons nécessite  $\mathcal{E} f_d k^2$  points de croisement.

Pour effectuer les calculs qui suivent on va reprendre la notation exposée sur le

réseau de De Bruijn (section 9.2.1.4). On peut s'arranger par exemple pour qu'un processeur  $\mathcal{P}$  soit numéroté par  $(x_1, x_2, \dots, x_{\mathcal{E}}) \in (\mathbb{Z}/k\mathbb{Z})^{\mathcal{E}}$  avec  $\mathcal{P} = \sum_{i=1}^{\mathcal{E}} x_i k^{\mathcal{E}-i}$ , ce qui revient à dire que  $x_i$  est le  $i^{\text{ème}}$  chiffre de  $\mathcal{P}$  écrit en base  $k$  (le 1<sup>er</sup> est donc celui du poids fort). What?—Unknown command character 'numéro d'un commutateur dans une colonne du réseau.

Si on assimile les processeurs à des commutateurs<sup>19</sup> et qu'on les considère comme des commutateurs de la ligne 0, on peut généraliser la topologie du réseau en disant qu'un commutateur  $(x_1, x_2, \dots, x_{\mathcal{E}-c}, \dots, x_{\mathcal{E}})$  de la colonne  $c \in [0, \mathcal{E} - 1]$  est relié à tous les commutateurs  $(x_1, \dots, \alpha_{\mathcal{E}-c}, \dots, x_{\mathcal{E}})$  de la colonne  $c + 1$ , avec  $\alpha_{\mathcal{E}-c} \in \mathbb{Z}/k\mathbb{Z}$  [Kot87, WF80].

Le dernier étage de commutateurs est relié aux processeurs par un *shuffle* généralisé et donc un commutateur  $(x_1, x_2, \dots, x_{\mathcal{E}})$  de la colonne  $\mathcal{E}$  sera relié à tous les processeurs  $(\alpha, x_1, x_2, \dots, x_{\mathcal{E}-1})$  avec  $\alpha \in \mathbb{Z}/k\mathbb{Z}$ .

Chaque **HyperCom** rassemble  $\mathcal{E}$  commutateurs et sur chaque commutateur un lien d'entrée et un lien de sortie restent dans l'**HyperCom** pour être reliés aux autres commutateurs du circuit ou à l'interface du processeur. Par conséquent il y a  $2(k-1)$  liens (entrées et sorties) qui sortent du circuit par commutateur et pour l'interface processeur, c'est-à-dire  $\mathcal{E}$  dispositifs reliés au niveau de l'hypercube statique. Le nombre de fils qui sortent et entrent dans un **HyperCom** est donc  $2\mathcal{F}(k-1)\mathcal{E}$ , soit  $2N\mathcal{F}(k-1)\mathcal{E}$  pour toutes les dimensions de l'hypercube du réseau.

Pour compter ceux du *shuffle*, c'est un peu plus compliqué : un fil reste dans le même **HyperCom** ssi

$$(x_1, x_2, \dots, x_{\mathcal{E}}) = (\alpha, x_1, x_2, \dots, x_{\mathcal{E}-1})$$

et donc que  $\forall (i, j) \in [1, \mathcal{E}]^2, x_i = x_j$  : les  $k$  commutateurs dont les  $N^{\circ}$  sont composés d'un chiffre identique en base  $k$ . Il y a donc  $k\mathcal{F}(N-1)$  fils qui sortent pour le *shuffle*. Comme il est plus simple de faire un circuit **HyperCom** régulier, on peut décréter que tous les fils sortent et qu'on rajoute une boucle de connexion externe pour les  $k\mathcal{F}$  fils concernés. Dans ce cas on peut considérer que  $k\mathcal{F}N$  fils sortent de tous les **HyperCom** pour le *shuffle*. Pour résumer, le nombre total de pattes de la machine dédiées aux communications est donc  $2N\mathcal{F}((k-1)\mathcal{E} + k)$ , soit  $2\mathcal{F}((k-1)\mathcal{E} + k)$  par **HyperCom**.

### 9.3.2.3 Partitionnement du réseau

Lorsqu'on divise la machine en  $c$  cartes, il est intéressant de savoir le nombre de fils de communication qui entrent et qui sortent au niveau de chaque carte pour savoir si le réseau n'est pas irréalisable.

Afin de subdiviser le réseau en cartes l'approche que l'on prend est d'avoir l'interface processeur du réseau et les commutateurs sur des circuits intégrés identiques afin de ne pas multiplier le nombres de circuits intégrés. C'est donc une approche différente de [FWT82], d'autant plus que notre réseau est pipeliné et donc que le temps de transfert est moins sensible au nombre de circuits intégrés à traverser.

---

19. Ce n'est pas fortuit : un processeur peut choisir d'envoyer un message sur un de ses  $k$  canaux d'interface ce qui lui donne un aspect de commutateur. Ceci dit, si on a rajouté la colonne supplémentaire de commutateurs, c'est celle-ci que l'on prend comme colonne 0.

On peut décider que si la machine possède  $c$  cartes,  $c$  étant une puissance de 2, une carte numérotée  $(b_1, b_2, \dots, b_{\log_2 c})$  en base 2 possède tous les processeurs et commutateurs  $(b_1, b_2, \dots, b_{\log_2 c}, \dots, b_{\log_2 N})$ , donc que le numéro de la carte est le poids fort du numéro binaire de ce qu'elle contient. Cela revient à dire qu'une carte contient des lignes de commutateurs contiguës telles qu'on les a représentée sur la figure 9.13. On peut appliquer un raisonnement similaire à celui fait précédemment sur les commutateurs.

Un commutateur d'une carte a un lien qui sort s'il est connecté à un commutateur dont le poids fort de son numéro n'est pas le même que celui de la carte d'émission. Les  $\lfloor \log_k c \rfloor$  premiers niveaux de commutateurs contrôlent les premiers chiffres de l'adresse en base  $k$  des processeurs. À ce niveau il y a donc  $\mathcal{F} \frac{N}{c} (k-1) \lfloor \log_k c \rfloor$  qui partent de chaque carte étant donné que chaque carte contient  $\frac{N}{c}$  lignes de commutateurs.

Il faut aussi considérer le cas où  $c$  n'est pas une puissance entière de  $k$ , c'est à dire que la  $\lfloor \log_k c \rfloor$  rangée de commutateurs aura seulement  $\log_2 c \bmod \log_2 k$  bits qui auront la possibilité de faire sortir des liens de la carte: il faut donc rajouter  $\mathcal{F} \frac{N}{c} (\log_2 c \bmod \log_2 k) (2^{\log_2 c \bmod \log_2 k} - 1)$  fils à l'expression précédente.

Le cas du *shuffle* est assez délicat et il est difficile de trouver le nombre de fils qui sortent de chaque carte à son niveau sous forme arithmétique simple. Si  $k \geq c$ , tous les bits codant le numéro de carte auquel est relié un commutateur peuvent changer, ce qui correspond à  $\frac{N}{c} (c-1)$  blocs de  $k/c$  liens correspondants aux bits du numéro de commutateur ne concernant pas les bits codant le numéro de carte, soit  $\mathcal{F} N k \frac{c-1}{c^2}$  fils qui sortent par carte, *ie* presque le nombre maximal. Le cas où  $k < c$  est plus subtil car les cartes n'ont pas toutes le même nombre de fils qui sortent. On se contentera de dire que le nombre de fils sortants par carte est inférieur à  $\mathcal{F} N k / c$ , ce qui est très proche de la formule précédente.

Pour résumer, en multipliant par 2 pour avoir le nombre de fils sortants ET entrants, si  $k \geq c$ , le nombre de fils de communication relié à une carte vaut

$$2\mathcal{F}N \left( \frac{(k-1) \lfloor \log_k c \rfloor + (\log_2 c \bmod \log_2 k) (2^{\log_2 c \bmod \log_2 k} - 1)}{c} + k \frac{c-1}{c^2} \right)$$

sinon, dans les 2 cas de toute manière, il est borné supérieurement par :

$$\frac{2\mathcal{F}N}{c} ((k-1) \lfloor \log_k c \rfloor + (\log_2 c \bmod \log_2 k) (2^{\log_2 c \bmod \log_2 k} - 1) + k)$$

La table 9.1 indique un résumé de la complexité de quelques configurations typiques pour  $f_c = 2$  en choisissant pour chaque cas la formule précédente la mieux adaptée. On suppose que la machine à 16 PEs est divisée en 4 cartes et celle à 256 PEs en 16 cartes.

La limitation semble être le nombre de fils qui arrivent et partent par carte. En cela on préférera un réseau avec  $k = 2$  par exemple et  $\mathcal{F} = 4$  ou 6 ( $f_d = 2$  ou 4 respectivement) selon les performances désirées, comme on va le voir par la suite. Le nombre de pattes et de points de croisement est tout à fait compatible avec les plus gros circuits reprogrammables.

### 9.3.3 Usage

L'étude des performances en utilisation normale fait appel à une analyse plus fine du fonctionnement des routines de communication et nécessite peut-être un minimum de compréhension du langage d'assemblage de la machine décrit dans l'annexe ??.

TAB. 9.1 - Complexité de quelques configurations du réseau hybride de POMP.

Réseau			Pattes de com. par PE	points de croisement par PE	Fils par carte
$\mathcal{F}(f_d)$	Étages	$k \times k$			
$N = 16, c = 4$					
4 (2)	4	$2 \times 2$	48	32	$\leq 128$
	2	$4 \times 4$	80	64	$\leq 192$
6 (4)	4	$2 \times 2$	72	64	$\leq 192$
	2	$4 \times 4$	120	128	$\leq 288$
10 (8)	4	$2 \times 2$	120	128	$\leq 320$
	2	$4 \times 4$	200	256	$\leq 480$
$N = 256, c = 16$					
4 (2)	8	$2 \times 2$	80	64	$\leq 768$
	4	$4 \times 4$	128	128	$\leq 1280$
	2	$16 \times 16$	368	1024	$\leq 3840$
6 (4)	8	$2 \times 2$	120	128	$\leq 1152$
	4	$4 \times 4$	192	256	$\leq 1920$
	2	$16 \times 16$	552	2048	$\leq 5760$
10 (8)	8	$2 \times 2$	200	256	$\leq 1920$
	4	$4 \times 4$	320	512	$\leq 3200$
	2	$16 \times 16$	920	4096	$\leq 9600$

### 9.3.3.1 Réseau statique

Il s'agit du mode de fonctionnement le plus simple déclenché par les routines de communication sur grille de POMPC.

Comme le réseau est statique, le temps de latence est constant et il n'y a pas de risque de blocage. Il suffit qu'un processeur distant lise la donnée reçue au bout du temps de latence du réseau, ce qui est prédéfini dans les routines de communication sur grille. Comme on connaît aussi le patron de communication, on sait sur quel processeur doit arriver un message et depuis quel lien, ce qui permet une utilisation maximale du réseau.

Si on veut faire des communications de registre à registre, on offre les instructions suivantes :

`st rs,r0,HC_i`

pour envoyer 32 bits sur l'hypercanal  $i$  depuis le registre  $s$  et

`ld rd,r0,HC_i`

pour recevoir à l'autre bout de l'hypercanal  $i$  dans le registre  $d$  les 32 bits envoyés.

On obtient donc les performances crêtes du réseau, puisque lorsqu'on a plusieurs paquets à envoyer (dû au fait qu'on a un nombre suffisant de processeurs virtuels par exemple) on peut commencer à préparer et envoyer un paquet le temps que le précédent soit transmis.

Mais dans ce cas, on est souvent amené à transférer des tableaux de messages correspondant par exemple aux PVs et se trouvant en mémoire. Dans ce cas avec la méthode précédente on divise la bande passante par 2 car on rajoute une instruction de copie

du registre en mémoire et réciproquement, ce qui est gênant lorsqu'on communique sur tous les hypercanaux à la fois et qu'on aurait besoin de cette bande passante. Or si on regarde ce qui se passe, on fait un passage inutile par le processeur pour aller de l'HyperCom à la mémoire. L'idée est donc de capturer la donnée au vol sur le bus pour simuler un transfert « à la DMA ». On utilise une fausse lecture pour mettre l'adresse adéquate sur le bus par le processeur et on utilise les 8 bits d'instruction supplémentaire pour préciser ce que l'on fait du bus et des données qui s'y trouvent.

Ainsi, écrire dans un hypercanal depuis la mémoire va se faire de la manière suivante :

```
ld r0, rbase, <>:DMA_ECR_HC_i
```

qui signifie<sup>20</sup> que le processeur lit une case mémoire dont l'adresse de base est dans *rbase* additionnée de la valeur *<>* fournie simultanément par le processeur scalaire qui gère la boucle des processeurs virtuels. La valeur de la case mémoire est envoyée dans *r0*, c'est-à-dire à la poubelle, mais l'HyperCom sait qu'il doit capturer la valeur qui passe sur le bus et l'envoyer dans l'hypercanal *i* grâce au suffixe rajouté à l'instruction. Par mesure de simplification, on n'a pas représenté l'instruction d'assembleur scalaire qui s'exécute en même temps.

La lecture d'un hypercanal et l'écriture de la donnée en mémoire se fait de manière similaire par l'instruction suivante :

```
ld r0, rbase, <>:DMA_LECT_HC_i
```

qui a pour effet de mettre l'adresse de la case mémoire adéquate sur le bus de donnée, pendant que l'HyperCom met la donnée reçue dans l'hypercanal *i* sur le bus et force les signaux d'écriture en mémoire par l'intermédiaire du PAL U097 et du signal *ForceEcrMem* de la figure 10.8, page 238.

À cause du pipeline du processeur, l'instruction de pseudo-lecture émise par le PE n'a lieu au niveau du bus de données qu'au bout de la latence de ce type d'instruction. Il faut donc compenser ce délai par un retard équivalent sur les signaux *DMA\_ECR\_HC\_i* dans l'HyperCom. Cela a pour conséquence que si une exception survient pendant un tel accès au réseau, l'HyperCom devra annuler l'accès en cours. S'il s'agit d'une lecture depuis la mémoire, cela ne sera pas grave, par contre s'il s'agit d'une écriture, l'adresse mise sur le bus concernera la routine d'exception et pas du tout l'adresse de la case mémoire liée à l'instruction de communication !

La reprise du programme après une exception se fait de manière triviale en recommençant la communication, puisque les effets de bords de celle-ci sont simples et connus.

Le temps de gestion du réseau configuré statiquement est donc  $2m\mathcal{E}(k-1)$  pour envoyer  $m$  mots de 32 bits dans les  $\mathcal{E}(k-1)$  hypercanaux par processeur. Le débit maximal de gestion est donc celui de la mémoire divisé par 2 (soit 50 Mo/s pour POMP) ce qui est excellent puisque c'est le débit crête de la mémoire (chaque paquet est lu sur un processeur puis écrit sur un autre) et ce au prix d'un mécanisme simple d'espionnage du bus et du rajout de deux instructions suffixes. De surcroît, on peut augmenter les performances en essayant de recouvrir temps de communication et temps de calcul, ou de manière plus simple avec les communications entre processeurs virtuels locaux, ce qui nécessite de diviser l'appel système de communication en 2, un pour l'émission et l'autre pour la réception.

---

20. Voir l'annexe ?? qui décrit le format des instructions de l'assembleur pour plus de précisions.

TAB. 9.2 - Routine d'émission de paquets sur le réseau à commutation de circuits.

---

```

st radresse0,r0,ADR_HC_0
st rdonnée0,r0,DON_HC_0
Ici on envoie des paquets sur les autres hypercanaux.
<Dernière instruction de gestion de l'hypercanal k - 1>:OW_COND_EMIS_HC_0
sub rnpaquets,rnpaquets,1; un paquet de moins peut-être
bcnd ne0,rnpaquets,2; s'il reste des paquets on saute l'instruction qui suit.
st rfin,r0,COMPTE_ACT; sinon on va terminer
ld radresse0,rpaquets,r0
ld rdonnée0,rdonnée,4
add rpaquets,rpaquets,8:CB:OW_COND_EMIS_HC_1
Idem pour tous les autres hypercanaux
ld r0,r0,COMPTE_ACT:GO_1; si l'activité = 2 on termine
On boucle s'il reste encore des paquets à émettre.

```

---

### 9.3.3.2 Réseau dynamique

Pour bien voir le mode d'utilisation du réseau, il faut se replacer dans le contexte de la machine qui est SIMD. On peut difficilement envoyer un paquet sur un hypercanal et se demander s'il a bien pu partir (pas de conflit dans le réseau) et sinon essayer de trouver un autre paquet pour le remplacer après avoir mis en attente le premier paquet. De toute manière, même sur une machine MIMD cette opération est souvent lente comparé au temps de transmission de la donnée dans le réseau et cette réorganisation de paquet au départ n'est pas rentable. Cette réorganisation pourrait être rentable si le matériel du réseau était capable d'une telle fonction. Or cette complexité supplémentaire est incompatible avec le fait qu'on veuille faire un circuit de routage simple.

Pour ce faire, on emploie une méthode différente moins subtile au niveau de l'allocation mais pipelinée pour compenser la perte de subtilité de l'algorithme : on lance l'émission d'un message puis on prépare l'émission du message suivant. Si l'émission a réussi on émet effectivement le paquet suivant, sinon on réemet le paquet précédent. Pour accélérer les opérations, il suffit de pouvoir conditionner les instructions des PEs en fonction d'un bit de conflit au niveau de l'HyperCom. Le PE voit les hypercanaux comme une case où on envoie l'adresse de destination qui déclenche en même temps l'établissement du chemin et une case où on envoie la donnée, case qui déclenche aussi l'émission quand on écrit une valeur dedans.

De même, du côté réception, chaque PE lit une donnée dans un registre rassemblant tous les hypercanaux d'arrivée pour éviter d'avoir à chercher dans quel hypercanal une donnée a bien pu arriver. Le fait qu'un paquet ait été reçu ou non peut être transformé en condition par l'HyperCom, accélérant la gestion de réception des paquets à l'arrivée.

Ainsi l'envoi d'un paquet se fait à la manière de la table 9.2. Les instructions doivent être légèrement réarrangées et la boucle déroulée 2 fois si on ne veut pas introduire des ralentissements dus aux dépendances dans le pipeline mais l'exemple est écrit tel quel pour ne pas (trop) compliquer la compréhension. La clé du mécanisme est le OW\_COND\_EMIS\_HC\_0 qui altère l'exécution des instructions si le chemin programmé à

TAB. 9.3 - Routine de réception de paquets sur le réseau à commutation de circuits.

---

```

ld rdonnée,r0,DON_HC_0:0W_COND_RECU_HC_0
st rpaquets,rdonnée,r0
add rpaquets,rpaquets,4:CB; un paquet de plus peut-être...
Ici on reçoit des paquets sur les autres hypercanaux.
ld r0,r0,TRAFFIC_RESEAU:GO_0

```

---

l'hypercanal 0 a réussi à être attribué. Il suffit de rajouter la boucle scalaire qui s'arrête lorsqu'il n'y a plus aucun paquet à envoyer et qu'ils ont tous été reçus (ie il n'y en a plus en transit). Le contrôle de la fin est géré sur chaque PE par l'écriture de la valeur 2 dans le compteur d'activité<sup>21</sup> qui a l'effet de désactiver jusqu'à la fin (voir la section 7.1.3.2) un PE qui n'a plus rien à émettre.

On constate donc que la routine d'émission de données de 32 bits demande  $8h + 1$  cycles d'horloge par boucle pour gérer  $h$  hypercanaux de sortie. De manière générale, l'émission de paquets de  $m$  mots de 32 bits demandera  $(6 + 2\lfloor \frac{m+1}{2} \rfloor)h + 1$  cycles<sup>22</sup>.

On pourrait encore diminuer le nombre de cycles de la routine en rajoutant un décompteur des paquets à émettre qui contrôlerait la boucle et économiserait 3 cycles par hypercanal à gérer.

Dans la routine précédente on a attribué ici un registre par couple (*adresse,donnée*) d'hypercanal, ce qui est possible dans la mesure où on a un petit nombre  $k$  d'hypercanaux en dynamique. Typiquement  $k = 2$ , donc c'est possible.

Simultanément il faut exécuter la routine de réception des paquets de la table 9.3.

La dernière instruction sert à récupérer globalement au niveau du processeur scalaire le fait qu'il reste ou pas du trafic sur le réseau afin de savoir quand terminer la routine de communication.

La routine de réception est donc plus simple et ne nécessite qu'environ  $2mk$  cycles pour  $k$  hypercanaux de réceptions et des messages de  $m$  mots de 32 bits et  $3k + 1$  pour des `int`<sup>23</sup>.

Étant donné qu'il faut faire un échange constant entre l'activité d'émission et l'activité de réception, il faut rajouter 4 cycles à la somme des temps précédents.

Pour conclure on peut donc écrire que l'exécution d'un cycle de gestion de communication nécessite environ  $g_d = (6 + 2\lfloor \frac{m+1}{2} \rfloor + 2m)k + 5$  cycles d'horloges et  $g_d = 9k + 8$  pour des `int`. Ces valeurs sont intéressantes car elles représentent la durée incompressible de gestion du réseau, quelles que soient les performances du réseau. Cela signifie qu'asymptotiquement pour de longs paquets on atteint un quart de la bande passante du bus des PEs : 1 cycle d'accès à un hypercanal tous les 4 cycles mémoire. Mais comme on se placerait plutôt asymptotiquement au niveau parallélisme massif qui n'intervient pas ci-dessus si ce n'est qu'il permet un déroulage efficace des boucles pour une bonne

---

21. On remarquera au passage l'utilisation du contrôle de flot parallèle fait par une instruction de pseudo-branchement dans le futur (voir section 7.2.6.1).

22. Cette formule est un peu pessimiste puisqu'il y a une queue d'accès à la mémoire de profondeur 3 qui permet d'optimiser un peu le programme ci-dessus pour des paquets plus gros. Mais comme il s'agit ici d'obtenir des ordres de grandeurs, on n'en tient pas compte.

23. La différence pour 32 bits est due au fait qu'on n'a pas le temps de dérouler la boucle.

TAB. 9.4 - Performances de la routine de gestion du réseau dynamique en Mo/s par processeur à 25 MHz.

$k$	Entiers transmis ( $m$ )			
	1	2	4	$\infty$
2	7.69	13.79	21.62	25.00
4	9.09	15.09	23.19	25.00
8	10.00	15.84	24.06	25.00
16	10.53	16.24	24.52	25.00

utilisation du pipeline, c'est plutôt les cas où  $m = 1, 2$  ou  $4$  (un nombre complexe en double précision) qu'il faut considérer et  $k = 2, 4, 8$ , ou  $16$  par exemple dont les performances sont indiquées dans la table 9.4 en exprimant le débit  $4mk \times \frac{f_p}{g_a}$  en Mo/s pour une fréquence de processeur  $f_p = 25$  MHz. On comparera avec le débit du bus de donnée qui est de 100 Mo/s.

Par conséquent, il ne servira à rien d'avoir un réseau plus performant que le débit de gestion : il ne sera pas utilisé !

En cas d'exception, le pipeline de l'HyperCom est perturbé et la communication en cours est annulée. Il suffit de savoir qu'on est dans une routine de communication pour savoir comment relancer la dernière communication qui a été interrompue. Pour cela, on peut mettre à jour un drapeau à chaque entrée et sortie d'une fonction de communication dynamique ainsi que le type de communication plus précis pour qu'une routine soit capable de relancer l'exécution du programme.

### 9.3.4 Performances

Étudions maintenant ce qu'on peut obtenir du réseau de la machine indépendamment de sa gestion.

#### 9.3.4.1 Réseau statique

À chaque coup d'horloge du réseau,  $\mathcal{F}$  bits peuvent être envoyés sur chaque lien du réseau. Le débit par processeur est donc  $(\mathcal{E}(k-1) + k)\mathcal{F}f_r/8$  octets par seconde avec  $f_r$  la fréquence de fonctionnement du réseau si on tient compte du fait qu'il y a  $\mathcal{E}(k-1)$  liens pour l'hypercube et  $k$  pour le *shuffle*, soit  $N(\mathcal{E}(k-1) + k)\mathcal{F}f_r/8$  o/s pour tout le réseau. Le tableau 9.5 indique le débit du réseau pour quelques configurations et quelques fréquences.

Il est à noter qu'*a priori* on saura mieux utiliser le réseau statique pour  $k = 2$  car il permet d'émuler les grilles de manière optimale.

Si on utilise par exemple une communication dans une grille 2D physique, seule la moitié des liens risque d'être utilisée, si on se place dans l'optique  $k = 2$ . Il faut donc en tenir compte pour choisir le réseau en le surdimensionnant car ce type de communication est courant par exemple pour les méthodes de résolutions itératives d'équations différentielles en 2D. En 3D le problème est encore accentué.

Mis à part le problème du temps de latence qui peut intervenir dans le choix, on remarque qu'un réseau avec  $k, \mathcal{F} \leq 4$  est déjà suffisant pour saturer la bande passante

TAB. 9.5 - Débit de quelques configurations statiques en Go/s. En italique est indiqué le débit par PE en Mo/s.

$f_r$	25 MHz			50 MHz			100 MHz			200 MHz		
$k$	2	4	16	2	4	16	2	4	16	2	4	16
$\mathcal{F}$	$N = 16$											
2	0.6	1.0	3.1	1.2	2.0	6.2	2.4	4.0	12.4	4.8	8.0	24.8
	<i>38</i>	<i>62</i>	<i>194</i>	<i>75</i>	<i>125</i>	<i>388</i>	<i>150</i>	<i>250</i>	<i>775</i>	<i>300</i>	<i>500</i>	<i>1550</i>
4	1.2	2.0	6.2	2.4	4.0	12.4	4.8	8.0	24.8	9.6	16.0	49.6
	<i>75</i>	<i>125</i>	<i>388</i>	<i>150</i>	<i>250</i>	<i>775</i>	<i>300</i>	<i>500</i>	<i>1550</i>	<i>600</i>	<i>1000</i>	<i>3100</i>
8	2.4	4.0	12.4	4.8	8.0	24.8	9.6	16.0	49.6	19.2	32.0	99.2
	<i>150</i>	<i>250</i>	<i>775</i>	<i>300</i>	<i>500</i>	<i>1550</i>	<i>600</i>	<i>1000</i>	<i>3100</i>	<i>1200</i>	<i>2000</i>	<i>6200</i>
$\mathcal{F}$	$N = 256$											
2	16.0	25.6	73.6	32.0	51.2	147	64.0	102	294	128	205	589
	<i>62</i>	<i>100</i>	<i>288</i>	<i>125</i>	<i>200</i>	<i>575</i>	<i>250</i>	<i>400</i>	<i>1150</i>	<i>500</i>	<i>800</i>	<i>2300</i>
4	32.0	51.2	147	64.0	102	294	128	205	589	256	410	1178
	<i>125</i>	<i>200</i>	<i>575</i>	<i>250</i>	<i>400</i>	<i>1150</i>	<i>500</i>	<i>800</i>	<i>2300</i>	<i>1000</i>	<i>1600</i>	<i>4600</i>
8	64.0	102	294	128	205	589	256	410	1178	512	819	2355
	<i>250</i>	<i>400</i>	<i>1150</i>	<i>500</i>	<i>800</i>	<i>2300</i>	<i>1000</i>	<i>1600</i>	<i>4600</i>	<i>2000</i>	<i>3200</i>	<i>9200</i>

des PEs avec  $f_r = 25$  ou 50 MHz.

### 9.3.4.2 Réseau dynamique

L'étude du débit du réseau à commutation de circuit est plus compliquée car il survient des conflits dans l'attribution des chemins. On va se placer dans le cas d'un routage de paquets aléatoires comme dans [Pat81]. C'est assez justifié dans la mesure où beaucoup de motifs de communications régulières sont exécutés sous forme de communications sur grille plutôt que sous forme de communications générales.

On considère au préalable un commutateur à croisillons  $k \times k$  et s'intéresse à une sortie en particulier. La probabilité pour qu'un paquet sur une entrée avec une probabilité  $e$  arrive sur cette sortie est  $\frac{e}{k}$  si on considère une répartition aléatoire des adresses. La probabilité pour que le paquet n'aille pas sur cette sortie est donc  $1 - \frac{e}{k}$ . La probabilité pour que cette sortie n'ait aucun des  $k$  paquets qui entrent chacun sur une entrée différente avec une probabilité  $e$  est donc  $(1 - \frac{e}{k})^k$ . On en déduit que si on envoie  $k$  paquets en entrée avec une probabilité  $e$ , la probabilité d'avoir un message sur une sortie est

$$s = 1 - (1 - \frac{e}{k})^k$$

et si on considère la bande passante du commutateur à croisillons au niveau de toutes les sorties, elle est de

$$b = k(1 - (1 - \frac{e}{k})^k)$$

Un réseau multiétage étant composé d'une succession de commutateurs à croisillons, la probabilité d'avoir un paquet sur une entrée d'un commutateur de l'étage  $i$  est égale à la probabilité d'avoir un paquet sur la sortie du commutateur de l'étage  $i - 1$  à laquelle celle-là est reliée. Par conséquent, la probabilité d'avoir un message sur une sortie de

TAB. 9.6 - Performances du réseau dynamique en Go/s (performances par processeur en Mo/s, en italique) pour une fréquence  $f_r$  de 25 MHz.

Réseau			Eff.	Débit pour des			
$f_d$	Étages	$k \times k$		int	double	128 bits	$\infty$
$N = 16$							
2	4	$2 \times 2$	0.45	0.08 <i>5</i>	0.08 <i>5</i>	0.09 <i>5</i>	0.1 <i>6</i>
	2	$4 \times 4$	0.53	0.19 <i>12</i>	0.20 <i>12</i>	0.20 <i>13</i>	0.2 <i>13</i>
4	4	$2 \times 2$	0.45	0.16 <i>10</i>	0.17 <i>11</i>	0.17 <i>11</i>	0.2 <i>11</i>
	2	$4 \times 4$	0.53	0.38 <i>23</i>	0.40 <i>25</i>	0.41 <i>26</i>	0.4 <i>26</i>
8	4	$2 \times 2$	0.45	0.32 <i>20</i>	0.34 <i>21</i>	0.35 <i>22</i>	0.4 <i>22</i>
	2	$4 \times 4$	0.53	0.75 <i>47</i>	0.79 <i>50</i>	0.82 <i>51</i>	0.8 <i>53</i>
$N = 256$							
2	8	$2 \times 2$	0.30	0.77 <i>3</i>	0.85 <i>3</i>	0.90 <i>4</i>	1.0 <i>4</i>
	4	$4 \times 4$	0.37	1.88 <i>7</i>	2.09 <i>8</i>	2.21 <i>9</i>	2.3 <i>9</i>
	2	$16 \times 16$	0.48	9.87 <i>39</i>	10.96 <i>43</i>	11.61 <i>45</i>	12.3 <i>48</i>
4	8	$2 \times 2$	0.30	1.54 <i>6</i>	1.71 <i>7</i>	1.81 <i>7</i>	1.9 <i>8</i>
	4	$4 \times 4$	0.37	3.76 <i>15</i>	4.17 <i>16</i>	4.42 <i>17</i>	4.7 <i>18</i>
	2	$16 \times 16$	0.48	19.73 <i>77</i>	21.92 <i>86</i>	23.21 <i>91</i>	24.7 <i>96</i>
8	8	$2 \times 2$	0.30	3.08 <i>12</i>	3.42 <i>13</i>	3.62 <i>14</i>	3.8 <i>15</i>
	4	$4 \times 4$	0.37	7.51 <i>29</i>	8.35 <i>33</i>	8.84 <i>35</i>	9.4 <i>37</i>
	2	$16 \times 16$	0.48	39.46 <i>154</i>	43.85 <i>171</i>	46.43 <i>181</i>	49.3 <i>193</i>

l'étage  $i$  du réseau est définie récursivement par

$$s_i = 1 - \left(1 - \frac{e_i}{k}\right)^k = 1 - \left(1 - \frac{s_{i-1}}{k}\right)^k$$

Le débit du réseau complet est donc  $kNs_E$  avec  $s_E$  la probabilité d'avoir un message sur une sortie du dernier étage et  $e_1 = s_0 = 1$  pour calculer le débit crête du réseau, ie chaque processeur émet un message sur tous ces hypercanaux.

Les conditions d'utilisations du réseau sont les suivantes : on propage l'adresse du destinataire (codée sur  $\log_2 N$  bits) pour établir le canal virtuel à travers les  $f_d$  fils de données, les fils de contrôle avertissant que c'est une adresse qui passe, à une fréquence de  $f_r$ . L'émission de cette adresse prend  $\lceil \frac{\log_2 N}{f_d} \rceil$  cycles et est suivie par  $\lceil \frac{b}{f_d} \rceil$  cycles d'émission des  $b$  bits de données du message. Comme on envoie le message suivant sans attendre la réponse de réussite du message précédent, la propagation des adresses et des données est pipelinée et le débit maximal. Le débit du réseau est par conséquent

$$\frac{kNs_Ebf_r}{\lceil \frac{\log_2 N}{f_d} \rceil + \lceil \frac{b}{f_d} \rceil}$$

Le débit pour quelques configurations est indiqué sur le tableau 9.6, pour  $b = 32, 64, 128$  et  $\infty$  et seulement une fréquence de fonctionnement de 25 MHz par mesure de concision.

Le passage à la limite pour  $b \rightarrow \infty$  permet d'obtenir le débit où le temps de configuration des adresses est négligeable devant le temps de transfert des données,

par exemple lorsqu'on peut factoriser les transferts entre processeurs virtuels et qu'au niveau des processeurs physiques les motifs de communication sont constants.

Une autre manière d'utiliser le réseau dynamique peut être aussi de l'utiliser de manière à ce que le motif de communication soit constant : on n'envoie une adresse de configuration qu'une seule fois pour relier les processeurs de manière statique mais suivant des motifs plus compliqués que ceux du réseau statique. Cela peut être utile lorsqu'on a un algorithme qui a besoin d'un seul motif de communication et que c'est une configuration acceptable par le réseau. Les performances sont alors intermédiaire entre le réseau statique pur et le réseau dynamique. C'est mieux que le réseau dynamique puisque l'adresse de destination n'est pas obligée de circuler avant chaque message, c'est moins bien que le réseau statique puisque seuls  $f_d$  fils sur les  $F$  de chaque lien sont utilisables et que seuls  $K$  liens sont utilisés, comparé aux  $\mathcal{E}k + k - 1$  du réseau statique. En tout cas l'efficacité du réseau ainsi utilisé redevient 1 ce qui est intéressant.

Dans le tableau 9.6 l'efficacité tient compte des conflits où plusieurs messages ont la même destination. Dans ce cadre l'efficacité d'un *crossbar* complet  $N \times N$  n'est aussi que de 0.63 et donc le réseau à 8 étages avec des commutateurs  $2 \times 2$  n'a que des performances divisées par 2 par rapport à lui mais à un coût matériel moindre dont on peut faire bénéficier la largeur des liens en échange. Si on n'utilise que des bijections acceptables par le réseau, l'efficacité passe évidemment à 1.

### 9.3.5 Conclusion

On a introduit la notion de réseau hybride statique-dynamique qui permet de récupérer les avantages de chacun selon les désirs du programmeur.

La possibilité d'avoir un réseau statique est justifiée par le fait que les communications sur grilles régulières, hypercube ou *shuffle* sont courantes dans les algorithmes typiquement parallèles et qu'il faut les optimiser. Cela est réalisé d'une part en supprimant le temps de reconfiguration et d'autre part en rendant accessible autant d'hypercanaux qu'il y a de liens utiles pour les communications sur grille.

Le fait d'avoir un réseau dynamique se justifie à l'opposé par le besoin de communications totalement aléatoires qu'il faut aussi gérer rapidement par un mécanisme de routage aidant le processeur. Comme les processeurs sont plus sollicités par la gestion des messages qui arrivent de manière aléatoire, le débit qu'ils peuvent supporter est plus faible et on peut se contenter d'un réseau moins puissant que la configuration statique.

Le réseau hybride proposé répond bien à ce qu'on lui demande avec une complexité faible des commutateurs à matrice de points de croisement élémentaires et du système d'ouverture des chemins virtuels dans le réseau. Cette simplicité permet de réaliser ce réseau dans les circuits logiques reprogrammables logiquement de la dernière génération.

Un bon compromis entre les performances statiques, dynamiques, celles des routines de gestion du réseau et la complexité nous fait choisir un réseau composé de 8 étages de commutateurs  $2 \times 2$  ( $k = 2$ ) avec  $f_c = 2$  fils de contrôle et  $f_d = 2$  ou 4 fils de données, suivant que l'horloge du réseau est de 50 ou 25 MHz respectivement. Ainsi les performances des différents modes d'utilisation sont équilibrées et le réseau peut loger dans un circuit reprogrammable par processeur élémentaire et le nombre de fils entre cartes reste du domaine de l'acceptable.

Un autre avantage de la reprogrammabilité est que le réseau n'est pas figé. On peut par exemple le rendre extensible en changeant des paramètres de pattes, changer la topologie du réseau et adapter en conséquence la partie de routage de la machine si besoin est. En particulier on peut promouvoir à l'occasion des signaux de contrôle devenus inutiles en signaux de données.

Si une application a besoin d'avoir un très fort débit sur une topologie possible sur le réseau physique de la machine, on peut très bien proposer à l'utilisateur une bibliothèque de fonctions qui reconfigurera le circuit **HyperCom** (environ 100 ms) pour exploiter au maximum la topologie physique, un peu à la manière dont cela est fait dans le cadre plus général de la machine **ARMEN** [Fil91b]. Il serait d'ailleurs intéressant de proposer ce réseau pour cette machine qui est une plate-forme d'expérimentation de choix.



# Chapitre 10

## Réalisation



« Il est plus simple de faire une machine simd mais alors on ne peut guère utiliser des processeurs du commerce, donc c'est plus compliqué... »

Syllogisme informatique traditionnel.

 A présence d'un tel chapitre pourrait sembler incongru dans une thèse, mais ce serait oublier le but ultime de la recherche un tant soit peu appliquée : avoir une réalisation à la fin de la thèse, en accord avec le cahier des charges initial. Le but de ce chapitre est donc de fournir une base saine pour une reprise du projet par un industriel, avec plus que des idées purement conceptuelles.

Enfin, un industriel, même s'il n'est pas intéressé par l'architecture parallèle de la machine pourra toujours récupérer d'une part les schémas de

- l'interface VME ;
- toute la partie interface avec le MC88100, en particulier le contrôle du pipeline qui est un point délicat lorsqu'on n'utilise pas le cache-MMU MC88200 approprié, dans le cas d'applications « temps réel » ;
- la partie vidéo, qui est assez simple mais qui permet d'atteindre des performances haut de gamme.

## 10.1 Description électrique

Comme on l'a vu dans les chapitres précédents, la machine se compose de trois parties distinctes :

- la machine hôte. Puisque c'est une machine standard du commerce, on n'a rien à développer ;
- le processeur scalaire de la machine qui s'occupe du contrôle de toute la machine ;
- les processeurs parallèles.

### 10.1.1 La carte de contrôle

C'est la carte la plus complexe de la machine mais, comme elle n'est répétée qu'une seule fois, ce n'est pas trop gênant.

#### 10.1.1.1 L'interface VME

C'est la partie assurant une « portabilité » de la machine sur plusieurs hôtes, au travers du bus standard VME [VME87]. En ce qui concerne POMP, une simple interface esclave a été choisie car cela simplifie la conception et parce qu'il n'était pas obligatoire que POMP soit maître sur le bus VME.

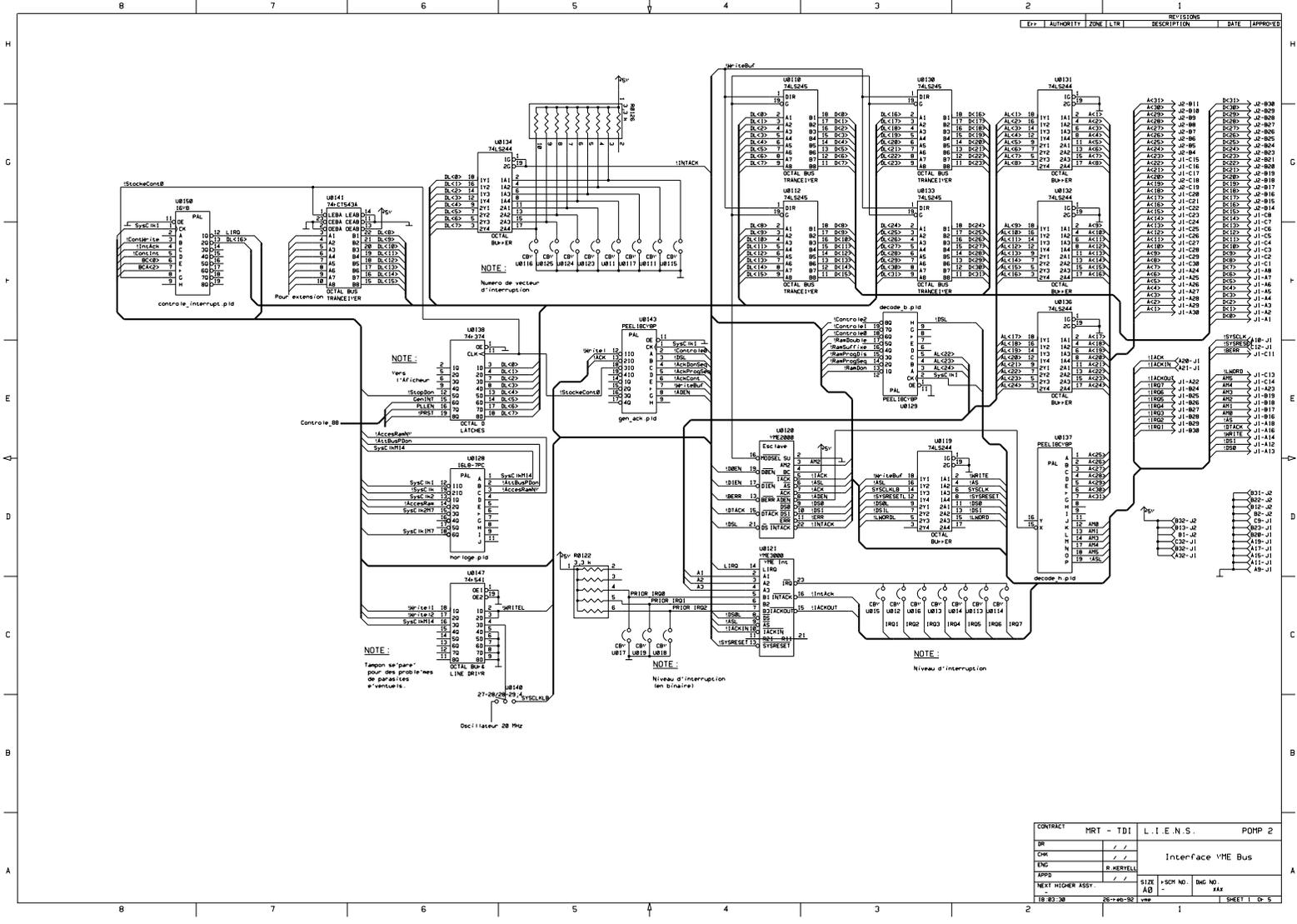


FIG. 10.1 - Schéma de l'interface VME.

Elle a été réalisée à partir de circuits logiques préprogrammés du commerce [PLX88a,

PLX88b] pour le contrôle et de tampons classiques pour les chemins de données<sup>1</sup>. Par mesure de simplification, on suppose que tous les accès à la carte se font par mots de 32 bits, alignés sur des adresses multiples de 4 et sans mode par paquets (*burst*). Cela suppose que l'interface UNIX entre POMP et l'hôte en tienne compte, surtout lorsque `ld88` écrit dans la mémoire du séquenceur (lors d'un `read()` par POMP par exemple) pour qu'il n'y ait que les octets concernés qui soient modifiés, comme on l'a vu en § 8.4.2.

Le schéma de la figure 10.1 décrit l'interface VME :

- le décodage d'adresse est fait par `decode_h.pld` pour la machine et `decode_b.pld` pour les différents module de la machine ;
- le VME2000 est chargé du contrôle du bus VME esclave, accompagné du VME3000 pour la génération des interruptions sur le bus VME ;
- le module VME est aussi chargé avec le circuit `horloge.pld` de générer toutes les horloges de la machine, horloges à 20 MHz<sup>2</sup> mais décalées temporellement d'environ 5–7 ns ou en opposition de phase, qui servent à gérer tous les pipelines de la machine ;
- `gen_ack.pld` s'occupe de rassembler tous les signaux de fin d'accès de cycle VME de tous les modules de la machine. Il échantillonne aussi le signal d'écriture du bus VME car celui-ci n'est pas valide tout le temps du cycle<sup>3</sup> ;
- U0138 stocke le mot de contrôle de la machine écrit par le chargeur `ld88` du SUN (registre `CONTROLE_O`). Le poids faible de cet octet est envoyé à un afficheur hexadécimal de contrôle sur la face avant de la machine qui permet de savoir à tout moment dans quel état est celle-ci.

Ce schéma peut être réutilisé tel quel comme schéma d'application.

### 10.1.1.2 Le MC88100 de contrôle

Le MC88100 scalaire, au cœur de la carte qui contrôle toute la machine, est interfacé avec 4 modules.

#### La mémoire de données scalaire

Il s'agit de la mémoire où sont stockées les variables scalaires de la machine. Pour l'instant elle fait 32 K mots de 32 bits et est accessible aussi bien pour le processeur scalaire que pour la machine hôte, afin de faire les initialisations, le contrôle, les appels systèmes et les entrées-sorties.

---

1. On aurait pu utiliser le VTC, un circuit intégrant plus de chose, malheureusement je n'en ai pas trouvé le fournisseur en France au moment où j'en ai eu besoin... De toute manière l'interface résultante aurait été sur-dimensionnée en performance.

2. En fait on peut aussi faire fonctionner la machine à la fréquence VME, 16,7 MHz, en utilisant le strap U0140.

3. Au départ cela avait été oublié et cela a été une des premières « bogues » de la machine...

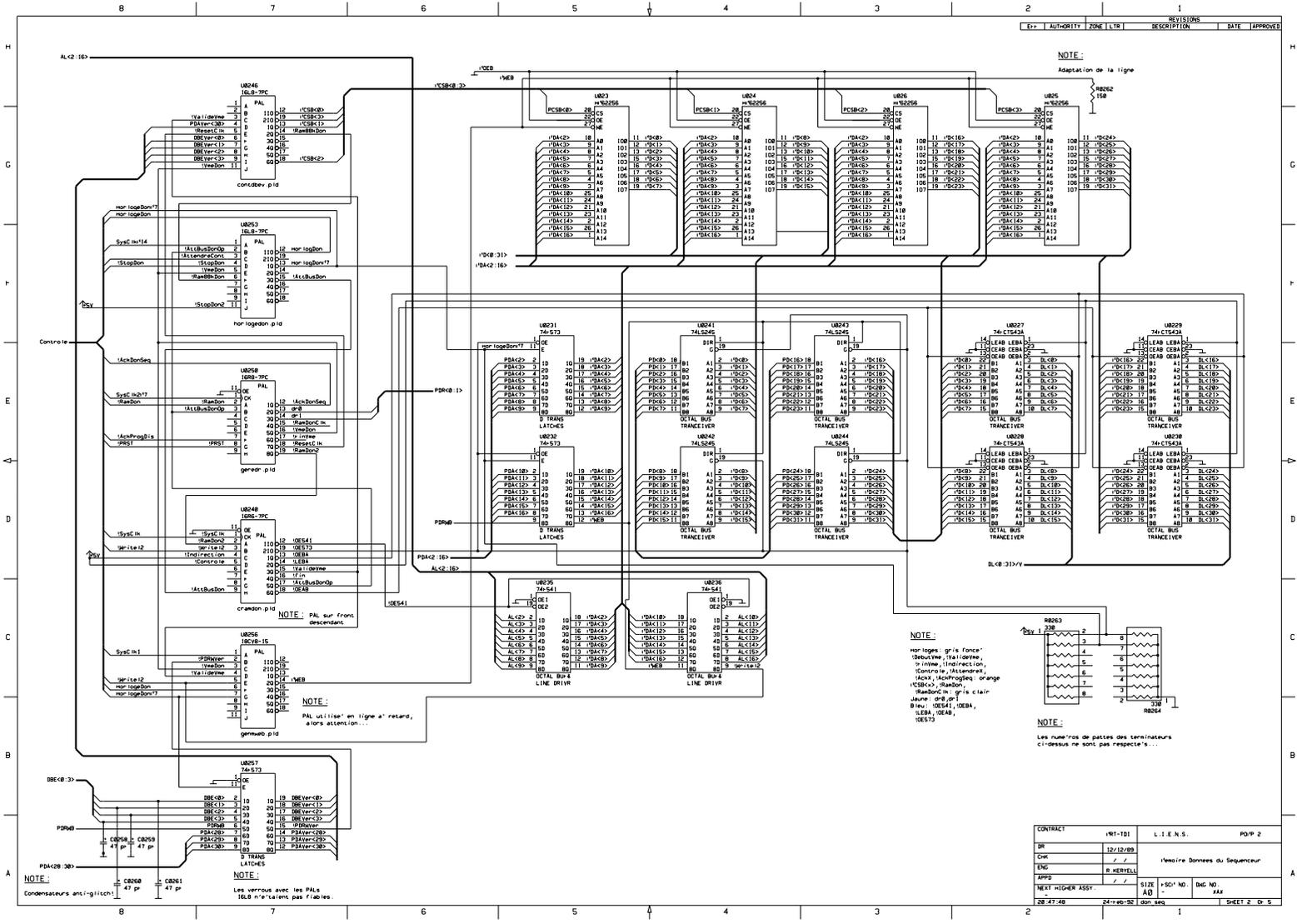


Fig. 10.2 - Schéma de la mémoire de données scalaires.

Le point délicat est le partage du P-bus du MC88100 avec le bus VME, car le P-bus

est pipeliné et il faut savoir arrêter proprement le pipeline. Le contrôle de cette mémoire nécessite pas moins de 5 PALS, formant plusieurs automates qui dialoguent :

- le circuit `germweb.pld` gère le signal d'écriture en mémoire qui doit suivre certaines contraintes temporelles. La programmation de ce PAL doit prendre en compte les caractéristiques physiques de celui-ci.
- Le fait d'utiliser ainsi un PAL comme ligne à retard peut paraître fort critiquable mais on constate que l'utilisation d'une ligne à retard plus précise ne sert à rien puisque c'est de toute manière un PAL, avec toute son imprécision, qui contrôlera la mémoire ;
- `horlogdon.pld` contrôle l'avancement du pipeline des données et avertit le cas échéant qu'il faut arrêter le MC88100, soit parce qu'il y a un conflit avec le bus VME, soit que le bit d'arrêt du bus de donnée `StopDon` est mis dans le registre de contrôle `CONTROLE_0` par l'ordinateur hôte ;
- `geredr.pld` ordonne le déroulement de l'arbitrage et fait attendre le bus de données du MC88100 lors d'un conflit avec le bus VME. Il indique à l'interface VME quand les accès sur le bus VME sont terminés ;
- `cramdon.pld` contrôle toutes les barrières qui rendent la mémoire à double accès. Les signaux engendrés par ce circuit sont en opposition de phase par rapport aux autres pour que les barrières aient le temps d'être configurées pour un accès du bus VME, un demi-cycle avant et un demi-cycle après ;
- l'utilisation de 543 sur le chemin de données VME (U0227—U0230) avec le circuit précédent permet de capturer les valeurs lues pour ne gêner le MC88100 que 2 cycles, indépendamment de la longueur du cycle VME ;
- `contdbev.pld` s'occupe de mettre en service les bons boîtiers de la mémoire en fonction du type d'accès (octets, mots, etc.) et de détecter lorsque le MC88100 veut accéder à celle-ci. Par contre les accès depuis le bus VME se font toujours par mot ;
- le pipeline des adresses et des signaux de contrôle du MC88100 est géré par les verrous U0231, U0232 et U0257 respectivement.

C'est la partie la plus délicate de la machine et la plus bruitée. On peut voir sur le schéma que plusieurs condensateurs ont été rajoutés pour « lisser » les signaux, accompagnés aussi de résistances de terminaison. Un gros problème a été le bruit sur les signaux de contrôle sortant du MC88100 en particulier.

Malheureusement, il reste encore beaucoup de bruit électrique sur la carte, inhérent à la technologie « wrappée » utilisée.

### La mémoire de code scalaire

Elle s'inspire du module précédent, si ce n'est que le MC88100 ne peut pas écrire dedans. Par contre le bus VME peut tout de même lire cette mémoire afin de vérifier si le code a bien été chargé correctement.



nomise les tampons entre le processeur et la mémoire. En outre, on se permet de faire attendre le bus VME lorsqu'il écrit dans la mémoire car cela n'arrive que lors du chargement du programme alors que la machine ne fonctionne pas et donc que l'hôte n'a pas à répondre à un appel système venant de POMP. Les verrous bidirectionnels 546 sont donc remplacés simplement par des tampons bidirectionnels 245. Cela simplifie beaucoup la gestion du module qui est assurée par un seul PAL, `cont_prg.pld`, comme le montre la figure 10.3.

### La mémoire de code vectoriel

Elle est accédée de manière similaire à la mémoire précédente par le processeur et on peut factoriser la partie génération d'adresse, c'est-à-dire réutiliser le bus `MC<2:16>`.

Le partage se fait aussi au niveau du contrôle car le PAL `cont_phc` de la mémoire de programme vectoriel est intimement lié à `cont_prg.pld` de la mémoire de programme scalaire.

La principale différence par rapport à la mémoire programme scalaire est que certains boîtiers mémoire peuvent être mis hors service — par l'intermédiaire des signaux `!CSMemHCBas`, `!CSMemHCHaut` et `!CSMemSufHC` — pour d'une part permettre la surcharge de l'opérande dans une instruction pour les PES et d'autre part permettre aux FIFOs de mémorisation des instructions de prendre la main lors de la gestion des exceptions (voir § 10.1.1.4).

#### 10.1.1.3 Le bus de contrôle de la carte

On a besoin d'un bus accessible depuis le MC88100 scalaire et le bus VME (pour la mise au point et l'initialisation) qui puisse contrôler le bon fonctionnement de la machine, programmer l'interface vidéo et traiter les exceptions parallèles : c'est le bus de contrôle dont le schéma est donné dans la figure 10.4.

Le contrôle est effectué par une série de PALs :

- `arbicont.pld` arbitre le bus de contrôle entre le MC88100 scalaire et le bus VME ;
- `gerecont.pld` s'occupe principalement du multiplexage du bus entre le MC88100 scalaire et le bus VME ;
- `decodcon.pld` fait le décodage du bus de contrôle vers les sous-systèmes, à savoir les 2 afficheurs, le registre de contrôle des interruptions vers l'hôte, la lecture de la queue des instructions envoyées aux PES, le registre de date des exceptions, la vidéo.

Le bus de contrôle est muni de 2 afficheurs hexadécimaux permettant à l'utilisateur de savoir, de manière primitive et obtuse, ce que fait la machine. Ces afficheurs sont bien munis de verrous mais ceux-ci étant trop lents par rapport à POMP, on a rajouté U048.

Le bus d'adresses de contrôle `BCA<2:17>` est contrôlé par les tampons U041–U042 (des 573 à cause du pipeline) du côté MC88100 et U045–U046 du côté VME<sup>4</sup>.

---

4. Des 543 pour relâcher le bus de contrôle en cas de lecture depuis le bus VME pour ne pas ralentir trop le MC88100. Pour l'instant ces 543 sont utilisés comme de simples tampons sur le prototype et donc ce mécanisme n'est pas en fonction.



laissant au logiciel le soin d'émuler des accès moins larges si nécessaires. Le multiplexage des données est assuré par des 543, U043, U0423–U0425 avec le MC88100 et U044, U0426–U0428 avec le bus VME. Le fait que ce soient des 543 réduit le temps d'écriture depuis le MC88100 car la donnée à écrire est mémorisée dans le verrou. Evidemment, cette sorte de cache ne fonctionne que lorsque les cycles d'écriture ne sont pas trop rapprochés.

Enfin, il y a une « boîte aux lettres » d'un octet visible depuis le bus VME et le bus de données du MC88100 scalaire. Cela permet au MC88100 de communiquer avec l'hôte sans conflit alors que, si on utilisait la mémoire de donnée ou un registre sur le bus de contrôle, des conflits pourraient apparaître, ralentissant le MC88100 scalaire qui ralentirait de même les processeurs parallèles. Dans ce cas, l'état du pipeline de la machine ne serait plus le même que celui prévu par le programme générant le code VLIW et le programme exécuté serait faux.

#### 10.1.1.4 Le système de gestion des exceptions et de surcharge

##### Les exceptions

Le problème de toute machine pipelinée est qu'il faut payer la vitesse par un état global plus complexe puisque plusieurs instructions sont à un même instant en cours d'exécution dans la machine, chacune à un stade d'avancement différent.

Lorsque la machine s'arrête sur une exception quelconque, il faut donc mémoriser son état. Chaque processeur contient le nombre de registres nécessaires à la sauvegarde de son état interne. Par contre l'état global de la machine a besoin d'être sauvegardé en plus. Cela est fait par l'intermédiaire d'une file (FIFO) de taille suffisante pour enregistrer autant d'instructions qu'il y a d'étages dans le pipeline depuis l'envoi d'une instruction parallèle par le processeur scalaire jusqu'à son exécution dans les PES.

Ainsi, avec l'état global de la machine, on est en mesure de relancer l'exécution après avoir corrigé si besoin est les instructions ou les opérandes fautifs.

Le schéma du mécanisme retenu pour effectuer cette « photographie » de la machine au moment d'une exception est indiqué sur la figure 10.5.

Le cœur du système est fait de 5 FIFOs (U074–U077, U0713) de 64 octets de profondeur qui sont capables de mémoriser les 40 bits d'instructions envoyées aux PES. En fait on n'a pas besoin d'autant de place mais c'est la taille minimale des FIFOs rapides en standard. Ces FIFOs sont connectées en parallèle sur le bus d'instruction, entrées aussi bien que sorties, afin d'être capables de « rejouer » au retour de l'exception les instructions qui n'ont pas été exécutées.

Afin de simuler une FIFO de profondeur moindre, `contfifo.pld` génère le signal de décalage de sortie décalé d'autant de coups d'horloges par rapport au signal de décalage d'entrée qu'il y a de mots à stocker, c'est-à-dire de nombre d'étages de pipeline dans la machine. Comme cela, le processeur scalaire n'a qu'à examiner le nombre minimal d'instructions.

Ce PAL travail en plus de conserve avec `cont_lect_fifo.pld` afin de permettre la relecture des FIFOs par le processeur scalaire. A chaque fois qu'un mot est lu, il est aussitôt restocké dans les FIFOs pour ne pas être perdu pour la reprise du programme. En effet il faut distinguer 2 phases :

- 1° la phase d'analyse de l'exception et de calcul des modifications à faire sur les opérandes ou les instructions : on lit les FIFOs ;

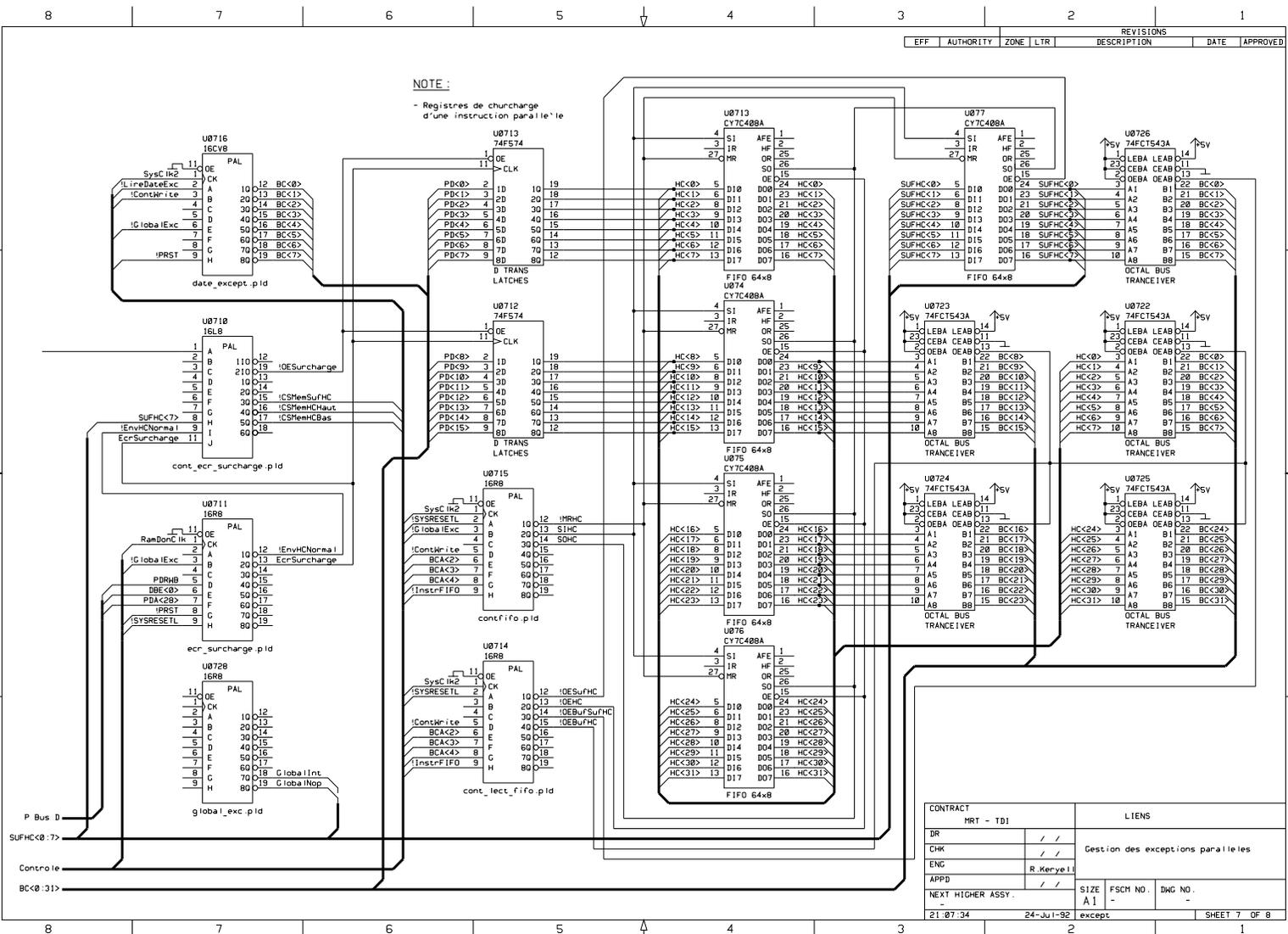


FIG. 10.5 - Schéma de la gestion des exceptions.

20 la phase de reprise où on rejoue pas à pas la reprise des instructions qui n'ont pas

été exécutées : on exécute le contenu des FIFOs, modulo les modifications calculées à la phase précédente.

Toutes les instructions de commande du système d'analyse des exceptions sont envoyées sur le bus de contrôle aux adresses correspondant à `!InstrFIFO` auxquelles répondent les 2 PALS précédents avec un sous-décodage sur `BC<2:4>`, correspondant aux instructions décrites dans la section ??.

Le bus d'instruction parallèle `HC<0:31>:SUFHC<0:7>` est accédé depuis le bus de contrôle à travers les interfaces `U0722-U0726` pour la lecture des FIFOs. Le fait qu'on ait choisi là aussi des 543 est qu'on pourra peut-être avoir besoin de générer une instruction purement synthétique, pas seulement au niveau de la surcharge décrite ci-après des 16 bits de poids faible de l'instruction.

Il reste à décrire `date_except.pld` qui est un compteur de 4 bits avec un verrou de la même taille. Lorsque survient une exception, la valeur du compteur, la « date », est copiée dans le verrou au même moment que cette opération a lieu sur le(s) PE(s) en exception, permettant ainsi de corrélérer la date des exceptions avec l'état du pipeline.

Cette corrélation est indispensable car la machine est pipelinée, même au niveau de l'arrêt de la machine. Cet arrêt n'étant pas immédiat, d'autres instructions exécutées sur d'autres PES que le(s) PE(s) arrêté(s) par une exception peuvent partir aussi en exception. Sans ce mécanisme d'estampage, il serait impossible de savoir jusqu'où le flot d'instruction est allé sur chaque PE et où faire repartir la machine.

### La surcharge des opérandes parallèles

Enfin, le schéma comprend la génération de l'opérande d'une instruction parallèle par le processeur scalaire afin de permettre principalement des émissions scalaires de valeurs ou de factoriser la gestion de pointeurs.

Pour ce faire, les 16 bits de poids faible d'une instruction parallèle fournie par la mémoire de programme parallèle sont remplacés par une valeur fournie par le processeur scalaire.

Ce champ de 16 bits a plusieurs significations suivant l'instruction lue en mémoire [MOT88a, pages 3-1-3-15] mais on peut distinguer les cas suivants :

- le champ de bits `<0:4>` peut représenter un registre source et donc permettre de générer une indirection globale sur un registre de PE ;
- `<0:10>` est un descripteur de champ de bits et permet de générer aussi bien une instruction de décalage selon une valeur globale qu'une instruction de branchement conditionnel où la condition est paramétrable globalement (on choisit par exemple `<=` ou `>=`, etc.) ;
- `<5:15>` peut être un sous-code d'instruction et donc on peut faire une indirection globale sur l'instruction exécutée sur les PES ;
- le champ de bit `<0:15>` peut enfin représenter une valeur immédiate et être utilisée pour faire une émission scalaire, initialiser globalement une variable parallèle ou les compteurs de programme de tous les PES, ou bien encore faire une indirection parallèle où l'adresse est scalaire, etc.

En fait, dans le cas d'une utilisation standard avec POMPC, seules les utilisations du dernier point ont été retenues et correspondent en fait aux émissions scalaires et aux initialisations.

Les autres points servent pour la récupération des exceptions et ne concernent pas l'utilisateur standard.

La surcharge de la partie basse des instructions parallèles est faite grâce au registres de surcharge U0712–U0713 inscriptibles directement depuis le MC88100 scalaire pour éviter une perturbation du pipeline. Ces registres entrent en fonction lorsque le bit 7 du suffixe parallèle est à 1 grâce à `cont_ecr_surcharge.pld`. Ce dernier met hors service simultanément la mémoire correspondant au poids faible de l'instruction parallèle par l'intermédiaire de `!CSMemHCBas`.

Le PAL `ecr_surcharge.pld` contrôle l'écriture dans le registre de surcharge et empêche la mémoire des instructions parallèles d'interférer avec la lecture des FIFOs lors de la récupération d'une exception.

#### 10.1.1.5 Les opérateurs globaux

Cette partie comprend la gestion du *ou global* d'une variable parallèle et du signal d'*exception globale*, comme indiqué sur la figure 10.6.

La section responsable du `Global Or` est cadencée par `gen_stocke_go.pld` et `gen_lire_go.pld`. Il permettent de stocker à chaque cycle les 4 bits de données du `Global Or` fournis par les PES dans les 4 « tranches » de stockage `global_or.pld` de 8 bits selon les signaux `StockeG0x`. Ce registre peut donc récupérer en 8 cycles une valeur globale de 32 bits. Si on veut faire une réception scalaire d'un `double` par exemple, il suffit de faire 2 accès de 32 bits.

C'est le signal `DecalerG0` fourni globalement par les PES qui permet de commencer une réception globale. Le fait d'utiliser un signal global supplémentaire plutôt que de mettre un bit dans le code parallèle de prévision d'une réception scalaire est fait pour simplifier la gestion du pipeline — on laisse plus de liberté aux PES — et savoir de manière sûre si il y a eu au moins un PE qui a voulu émettre une valeur globale. Ce fait est mémorisé dans `gen_lire_go.pld` et remis à 0 en cas de lecture de ce bit de statut ou de la valeur globale.

En plus `gen_lire_go.pld` contrôle la lecture depuis le MC88100 scalaire de la valeur globale. Ce registre est sur le bus de données scalaire et non pas sur le bus de contrôle toujours pour ne pas perturber le pipeline de la machine.

D'autre part 2 PALS sont occupés par la gestion des exceptions dans la machine :

- `status_except.pld` enregistre le fait qu'une exception globale a eu lieu sur au moins un PE mais permet aussi au processeur scalaire de générer une telle exception. Une exception globale peut se répercuter en exception locale suivant un bit de statut à travers le signal `GenIntAussi` ;
- `gere_except.pld` engendre les exceptions vers les processeurs scalaire et parallèles et obéit aussi à l'hôte via `GenINT`. Ce circuit est l'équivalent du PAL de la partie VME (`controle_interrupt.pld`), qui permet au processeur scalaire de faire appel à l'hôte à travers une interruption. On a prévu une entrée `VideoInt` permettant à la partie vidéo d'arrêter la machine pour lui dire qu'elle doit remplir ses tampons vidéo. En même temps un bit de statut est mis à 1

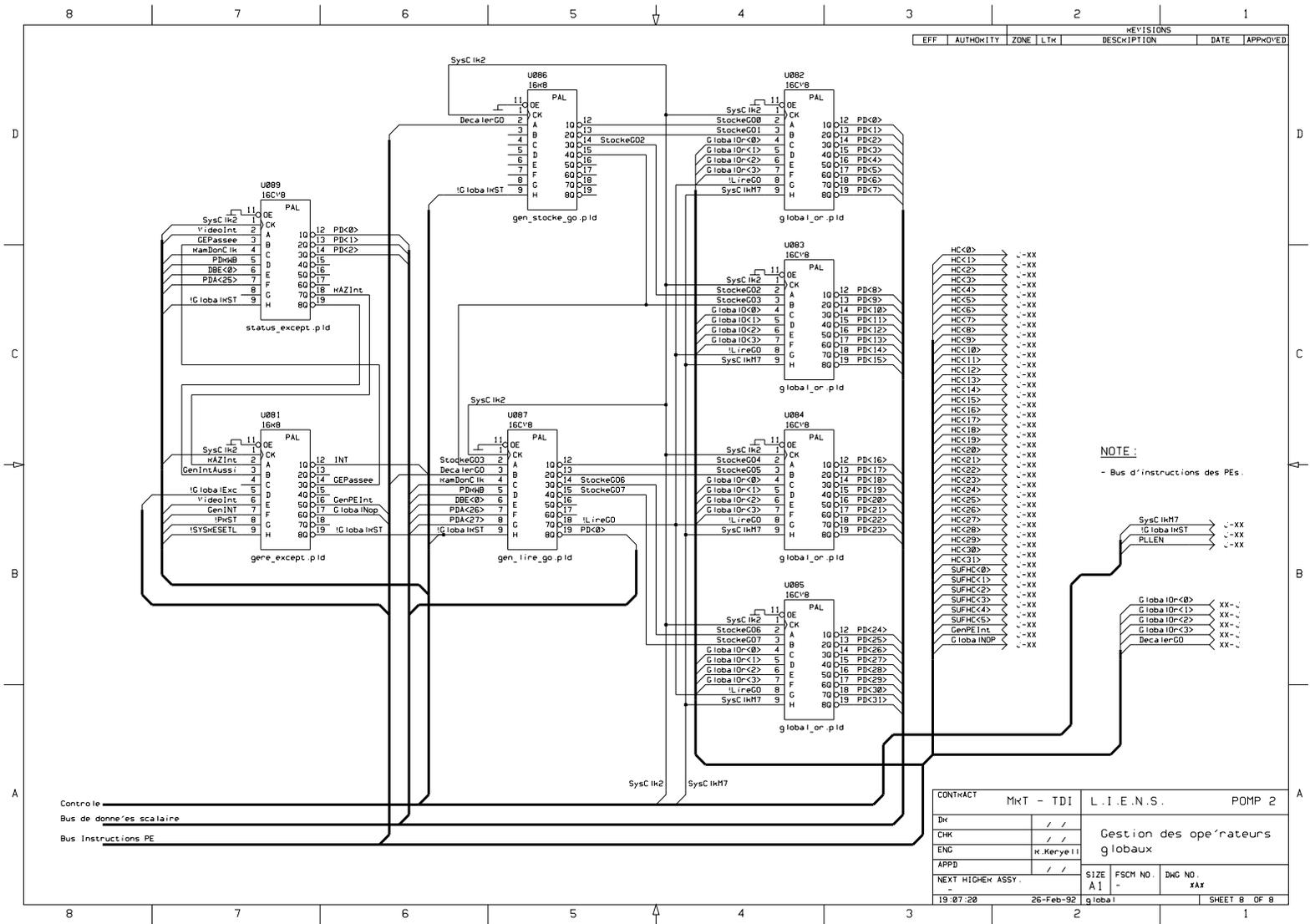


FIG. 10.6 - Schéma de la gestion des opérateurs globaux.

dans status\_except.pld pour indiquer au processeur scalaire qu'il doit lancer

la routine vidéo ;

Enfin on peut voir rassemblés les signaux du bus d'instructions vers les processeurs parallèles à savoir le bus d'instructions des **MC88100** parallèles, les signaux des suffixes d'instruction, le **GlobalNOP** permettant de geler les PEs, **GenPEInt** qui interrompt les PEs et les signaux permettant de récupérer une valeur globale.

Comme actuellement la partie communication entre le processeur scalaire et les PEs par des **HyperCom** n'a pas encore été dessinée, on peut faire des réceptions scalaires par l'intermédiaire du **Global Or** et des émissions par une diffusion scalaire dans un **where**. Pour ce faire, il faut bien sûr identifier chaque processeur par rapport aux autres. Cela est fait simplement en ayant une patte de configuration sur l'**HyperCom** permettant de différencier un PE, la numérotation des autres se faisant par propagation d'une vague de communication.

#### 10.1.1.6 La vidéo

Elle doit être réalisée à l'aide de 4 circuits de transposition PAV, chacun réalisé à partir d'un circuit reprogrammable XILINX vu le nombre nécessaire pour toute la machine, et de palettes graphiques.

Une étude du circuit PAV peut être trouvée dans [Ker88]. Il s'agit d'un circuit transformant les données arrivant au format sériel de chaque processeur en données parallèles à même de pouvoir alimenter les convertisseurs digitaux-analogiques vidéo. On peut voir ce circuit comme un transposeur de matrices 1 bit<sup>5</sup>. Il existe de tels transposeurs comme le Brooktree **Bt710** mais ils sont prévus à l'origine pour être utilisés dans des imprimantes laser et sont à la fois trop lents et impossibles à interfacer en ce qui nous concerne.

La partie conversion digitale-analogique du signal Rouge-Vert-Bleu est laissée à un circuit sur mesure, le **Bt463** qui est capable de générer un signal vidéo jusqu'à 170 MHz à travers une palette graphique gérant la vraie couleur (24 bit)<sup>6</sup>, la surimpression à 4 niveaux, la possibilité d'avoir 16 palettes différentes pour plusieurs fenêtres, ainsi que la gestion d'un curseur. Son interface avec un multiplexage temporel d'un facteur 4 convient pour l'interface avec les 8 circuits PAV.

Les discussions avec les gens de Thomson Digital Image nous ont amené à adopter un 4<sup>ème</sup> canal en plus des classiques Rouge-Vert-Bleu, le canal  $\alpha$  nécessaire en production vidéo pour faire des incrustations douces ou d'autres effets spéciaux.

Il nous faut donc rajouter une vidéo 8 bits pour faire la sortie  $\alpha$ . En fait il existe un circuit de chez INMOS, le **G364**, qui fait l'affaire pour plusieurs raisons que l'on retrouve d'ailleurs pour la plupart dans le circuit précédent :

- gestion du multiplexage temporel à l'entrée ;

---

5. Il s'agit là d'un circuit que l'on retrouve partout dès qu'on veut faire de la conversion entre des formats série et parallèle, en particulier dans la CM-2 où il y en avait pour interfacer les processeurs 1 bit (donc série) avec les coprocesseurs flottant 32 bit, ainsi que dans le GAMMA 60, comme expliqué dans la note 5 de la page 96.

6. L'existence d'une palette graphique n'implique en rien l'absence de « vraies couleurs », tout dépend de ce qu'on met dans celle-ci. Elle peut être en particulier utilisée pour faire la « correction  $\gamma$  » du moniteur vidéo, même si on a intérêt à le faire dans le moniteur pour perdre moins de dynamique [Mat88].

- palette graphique intégrée ;
- gestion d'un curseur ;
- compteur d'adresse ;
- interface avec des VRAM ;
- génération automatique des signaux de synchronisation vidéo.

Les 3 derniers points sont particulièrement intéressants. En effet, on peut voir le mécanisme de remplissage des registres vidéo des **HyperCom** comme le chargement du registre de sortie d'une VRAM macroscopique. Dans ce cas, le contrôleur graphique **G364** interrompt le processeur scalaire pour lui fournir l'adresse de départ de la mémoire vidéo. Le processeur scalaire envoie cette adresse à tous les processeurs qui peuvent remplir les registres vidéo des **HyperCom**.

La seule différence avec une VRAM est qu'ici la latence est plus élevée et qu'une contrainte du circuit ne sera plus respectée : les spécifications du circuit veulent que le chargement des registres vidéo se fasse au plus tôt après la fin du top de synchronisation verticale de la première ligne d'écran. Du coup, pour respecter cette contrainte, on a soit la solution de rajouter de la logique avec un additionneur sur le bus d'adresse du circuit vidéo pour allonger la latence du pipeline, soit faire la génération d'adresse avec quelques instructions supplémentaires dans le processeur scalaire. On n'utilise donc plus le **G364** que comme initiateur du chargement, par interruption, et le processeur scalaire s'occupant de pipeliner d'un niveau supplémentaire le remplissage pour qu'il n'y ait pas de trou en début d'écran.

Le schéma complet n'a pas été dessiné mais un synoptique de la partie vidéo est donnée en figure 10.7.

On constate que l'interconnexion entre les circuits PAVet les palettes graphiques forme un réseau de type *shuffle* que l'on rencontre souvent dans les réseaux d'interconnexion d'ordinateurs parallèles.

#### 10.1.1.7 La sortie audio

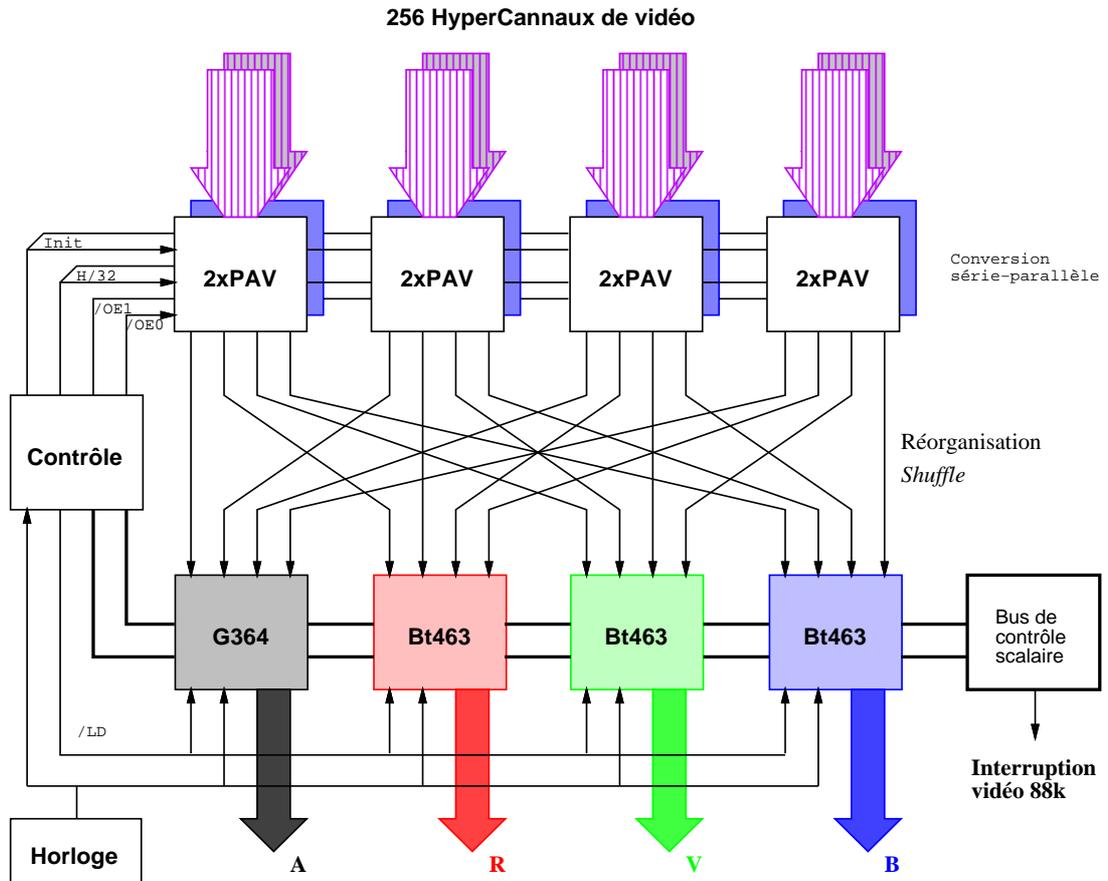
Comme il s'agit là d'une sortie à bas débit, 40 kmot/s, elle est réalisée par une FIFO de 32 bits connectée au bus de contrôle du processeur scalaire et à 2 convertisseurs digitaux-analogiques de 16 bits pour avoir une sortie stéréophonique de bonne qualité.

Le remplissage de cette FIFO se fait régulièrement sur interruption.

#### 10.1.1.8 Les entrées-sorties haut débit

Afin de réaliser le système d'entrées-sorties rapides, on peut utiliser deux méthodes, une ressemblant à la partie vidéo, l'autre se connectant sur le réseau de communication.

Dans le premier cas, cette partie ne diffère pas de la partie vidéo si ce n'est qu'on remplace la partie vidéo par un bus parallèle ou autant d'interfaces HIPPI que nécessaires pour absorber le débit et que les hypercanaux peuvent être utilisés dans les 2 sens. On utilise donc un circuit PAV pour 32 **HyperCom** qui sont utilisés pour faire la conversion série-parallèle entre les PEs et l'entrée-sortie à haut débit.



Si on se limite à un HyperCom par PE pour réaliser l'entrée-sortie parallèle et qu'on a 256 PEs avec une vitesse de décalage de 20 MHz on a déjà un débit de 640 Mo/s, soit de quoi alimenter plus de 6 interfaces HIPPI simultanément.

Dans le deuxième cas, il n'y a pas d'hypercanal supplémentaire à prévoir puisqu'on relie les entrées-sorties au réseau de la machine. Dans ce cas le débit disponible est encore plus important sans avoir à compliquer plus l'HyperCom. Par contre cela peut amener à rajouter des multiplexeurs sur le réseau, ce qui peut poser des problèmes de temps de propagation et d'adaptation d'impédance. Mais un grand avantage est que chaque HyperCom peut communiquer en mode parallèle sur un bus de 8 ou 32 bit, un bit par hypercanal, ce qui évite d'utiliser des PAV de conversion et simplifie donc la conception des entrées-sorties.

La solution choisie dépendra donc du débit d'entrées-sorties rapides voulu pour la machine.

### 10.1.2 Les cartes processeur élémentaire

Comme il s'agit de la partie qui est répétée de nombreuses fois, il est particulièrement important que chaque PE n'occupe que très peu de place et soit facilement répliquable.

### 10.1.2.1 Les modules processeurs

Dans l'étude actuelle, chaque module de processeur élémentaire est constitué de 9 circuits intégrés seulement (figure 10.8), ce qui permet d'avoir une machine possédant une puissance de l'ordre de 2 MIPS ou 1 MFLOPS par circuit intégré.

La mémoire nécessaire à l'initialisation du circuit XILINX n'est pas représentée sur le schéma. De toute manière, comme chaque **HyperCom** est identique, cette initialisation est factorisée et peut être faite par une liaison série globale alimentée par le processeur scalaire, permettant de reconfigurer l'**HyperCom** selon le bon vouloir du programmeur<sup>7</sup>.

Un module de PE se compose de 3 parties :

- un **MC88100** qui est le cœur calculatoire du nœud de la machine ;
- un **HyperCom**, circuit qui pervertit le processeur précédent pour le faire fonctionner en mode SIMD et lui rajoute la fonction de communication nécessaire au parallélisme ;
- une partie mémoire de données.

La partie mémoire est simplifiée par rapport à celle du processeur scalaire (figure 10.2) puisque celle-ci n'est plus partagée entre le processeur et le bus VME. Comme l'**HyperCom** ne peut pas être maître du bus, la mémoire n'est vue que par le **MC88100**.

Outre les boîtiers mémoires, on retrouve les 2 verrous (U099–U0910) mémorisant l'adresse de la case mémoire accédée fournie par le bus pipeliné du **MC88100**, et un PAL (U097) contrôlant les signaux de sélections inversés !PEDBE<0:3> validée pour les accès où PEAD<20> vaut 0.

Ce PAL s'occupe en fait de toute la partie gestion du bus pipeliné du **MC88100**, partie trop rapide pour être reléguée à l'**HyperCom**. C'est un petit dommage dans la mesure où cela aurait pu nous faire économiser 3 circuits intégrés. Evidemment, si on envisage de concevoir un **HyperCom** comme un circuit sur mesure, on pourra effectivement augmenter la densité de la machine de 30 %.

En fait c'est surtout la taille de la mémoire nécessaire à une application qui changera la taille de la machine. En effet, même si lorsqu'on a commencé l'étude on n'avait à notre disposition que des boîtiers mémoire de 256 kbits, on peut proposer aujourd'hui un nœud avec des boîtiers mémoire de 4 Mbits possédant 2 Mo/PE ou par exemple 4 ou 16 Mo/PE si on réalise des PES possédant respectivement 13 ou 21 circuits intégrés. On peut aussi économiser les verrous U099–U0910 si on utilise des mémoires statiques synchrones [Hit91] bien adaptées à notre problème (mais aussi plus chères...).

Enfin, le PAL est responsable aussi du blocage du **MC88100** élémentaire lorsqu'il veut exécuter une routine d'exception afin qu'il n'exécute par des instructions du flot d'instruction global en tant qu'instructions de la routine d'exception. Cela est détecté lorsque le **MC88100** veut accéder une instruction (**CFETCH**) en mode superviseur (**PECSU**), en même temps que l'**HyperCom** note la date à laquelle cela arrive. Le processeur est mis en attente à travers les signaux **PECRO–1** contrôlés aussi par le mécanisme d'activité décrit ci-après à travers le signal **StoppePE**.

---

7. Notons qu'il serait astucieux de faire charger cette mémoire depuis l'hôte ou le processeur scalaire car cela permettrait simplement de coder en « dur » le numéro de chaque PE dans chaque **HyperCom** et d'éviter d'avoir à différencier un PE par une patte externe comme proposé précédemment.



U017–U019 qui tient compte des entrées et des sorties au niveau des blocs inactifs. On peut initialiser et lire le contenu de ce compteur afin de pouvoir changer de processeur virtuel et/ou de collection.

En plus, un mécanisme permet d'arrêter le processeur lorsqu'il part en exception et de stocker le numéro d'exception dans U0122 en attente de la récupération globale par le processeur scalaire.

Comme on peut le constater, la partie à rajouter pour faire le contrôle d'activité SIMD est très simple et peut n'occuper qu'une petite partie d'un circuit logique programmable.

### 10.1.2.3 Une mesure analogique des performances de la machine

Puisque le circuit contrôle l'activité, celui-ci est capable d'informer le monde extérieur pour permettre un calcul global des performances de la machine.

En effet, si à chaque fois qu'un PE effectue une opération flottante il met un fil (MFLOPS sur la figure 10.8) à 1 pendant le cycle correspondant, une moyenne spatiale et temporelle de ce signal sur tous les PES fournit le nombre moyen de MFLOPS<sup>8</sup> de la machine.

Cette moyenne est effectuée très simplement en reliant tous les fils MFLOPS par une résistance à un condensateur qui effectue le moyennage dans le temps. On pourrait penser qu'un tel système ne fonctionnera pas en milieu parasité tel que l'est l'environnement électrique d'un ordinateur. En fait, comme la constante de temps du système de mesure des performances est assez longue (de l'ordre de la seconde) devant les interactions électriques et que le couplage capacitif ou inductif agit au niveau des dérivées des grandeurs physiques concernées (courant ou tension), ces interactions sont négligeables. Pour 256 PES, une valeur de 100 k $\Omega$  pour les résistances et de 2200  $\mu$ F pour le condensateur permet d'obtenir la constante de temps de l'ordre de la seconde.

Pour augmenter la précision du système, on peut moyennner les signaux au niveau de chaque carte de PES avant de les moyennner au niveau global et aller jusqu'à convertir le résultat en digital sur la carte du processeur scalaire si besoin est pour récupérer le résultat au niveau du logiciel. Cela permet en plus d'avoir une indication de puissance par bloc sous forme de barres de tailles variables, très utile dans une optique commerciale où le but est d'avoir le plus de diodes électroluminescentes de toutes les couleurs qui clignotent<sup>9</sup>.

On remarquera au passage que l'analogique permet souvent de trouver une solution élégante à des problèmes du monde digital, lorsque la précision voulue n'est pas prioritaire<sup>10</sup> [PDGO87]. En effet une solution exacte au calcul des performances nécessiterait des compteurs de classes d'instruction dans chaque PE dont on ferait régulièrement une somme globale au niveau du processeur scalaire.

Le système est doublé pour permettre en parallèle le calcul du nombre de MIPS via le signal MIPS qui indique pour chaque instruction et à chaque cycle si celle-ci est

8. Les instructions de conversion entre entier et flottant sont classées comme étant flottantes.

9. Finalement les décorateurs des vieux films de science-fiction vont avoir raison quant à l'aspect « clignotant » des ordinateurs du futur, mais non pas tant à cause de leur vision futuriste des machines qu'à cause de l'influence qu'ils ont dû avoir sur les informaticiens actuels réputés être grands amateurs de science-fiction...

10. Une solution plus astucieuse serait de mesurer la température du circuit et en fonction de la météorologie en déduire le nombre d'instructions utiles exécutées...

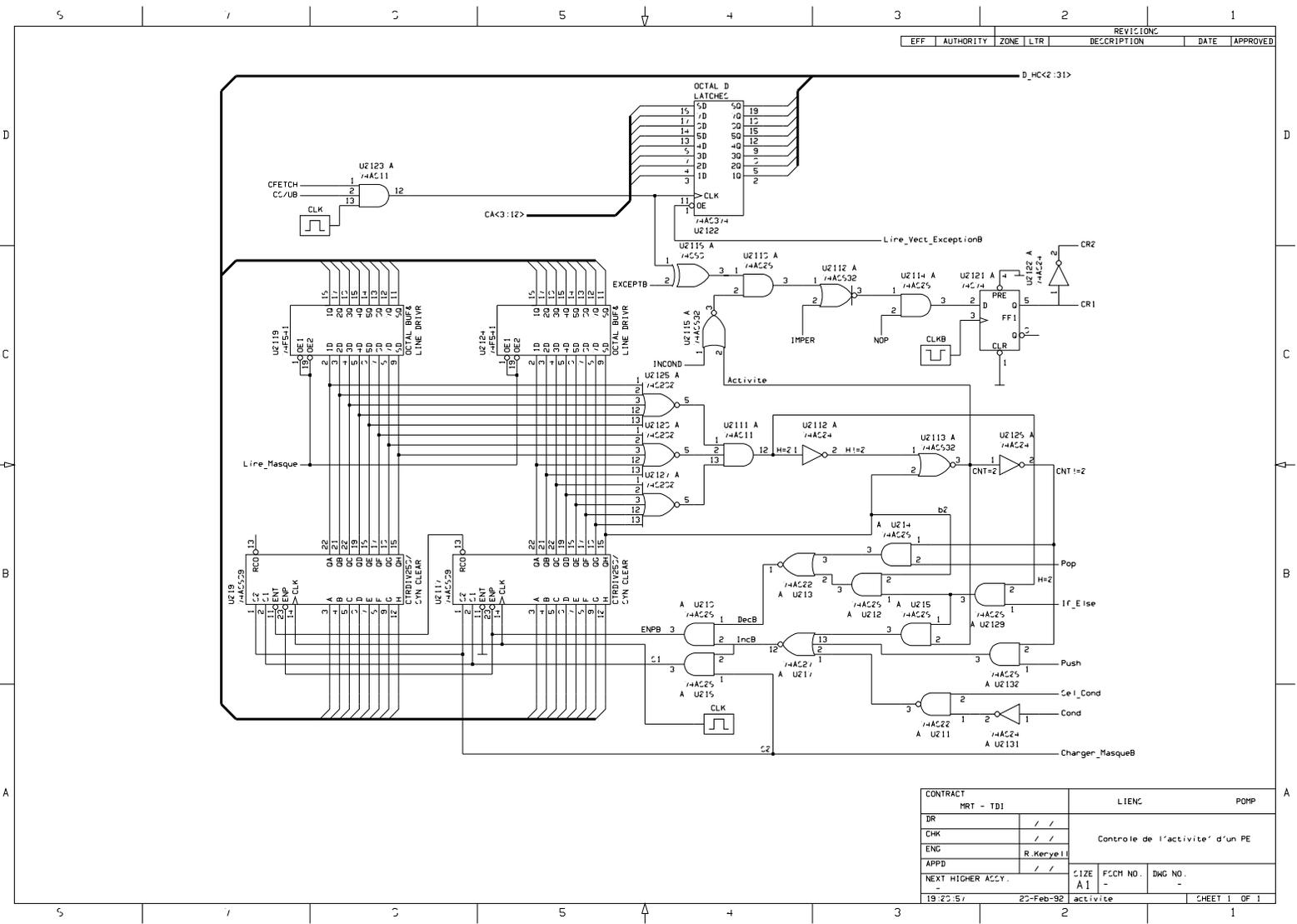


FIG. 10.9 - Schéma équivalent à la gestion de l'activité d'un PE.

une instruction entière. On peut combiner les signaux MFLOPS, MIPS et GlobalNOP en

2 signaux et 2 bits dans la mémoire de code, par soucis d'économie. Néanmoins, ces 2 bits nécessitent d'élargir la mémoire de code parallèle et pour cette raison le mesureur analogique de performances n'a pas été prévu dans notre prototype.

Afin de faire économie de cette mémoire supplémentaire, on pourrait facilement analyser les instructions exécutées par les PEs pour savoir si elles concernent des valeurs entières ou flottantes puisque le format des instructions du processeur RISC est simple mais cela obligerait à rajouter des fils vers l'HyperCom pour cette analyse. En plus, comme il n'existe pas d'instruction « `nop` » dans le MC88100 il ne serait pas évident de distinguer un `or r0,r0,r0` ou toute autre instruction inutile du flot d'instruction comme étant un `nop` rajouté par le compilateur d'une instruction utile.

Pour cette raison, le fait qu'une instruction soit un `nop` est codé dans le signal `GlobalNOP`, nécessaire de toute manière pour le mode SIMD. Il faut au moins avoir 1 bit du code parallèle pour fournir cette information.

#### 10.1.2.4 Le réseau

Etant donné le nombre très restreint de processeurs dans notre prototype, le réseau a été réduit à 1 circuit d'interface de bus avec verrous symétriques 543 permettant de relier les 2 PEs par 2 registres 8 bits.

Mais dans le cas d'un prototype industriel, le réseau est basé sur l'HyperCom qui s'occupe de l'émission d'un message, son routage ou de l'établissement d'un canal de communication et sa réception.

Etant donnés les circuits programmables disponibles actuellement [XIL90], il semble tout à fait raisonnable d'avoir au moins 32 fils du circuit réservés pour l'interconnexion entre les processeurs, ce qui permet de loger tout l'HyperCom dans un circuit de taille comparable à celle du MC88100. On peut même se permettre d'augmenter le nombre de pattes réservées à la communication pour gérer indépendamment les signaux de protocole des signaux de communication et augmenter les débits de communication.

A supposer qu'on se limite à une vitesse d'horloge identique à celle du processeur<sup>11</sup> et que 16 fils sont utilisés en émission, cela fait un débit crête de  $16 \times 20/8 = 40$  Mo/s, tout à fait raisonnable même compte tenu du fait qu'il s'agit là d'un débit crête car le MC88100 doit pouvoir être en mesure de gérer les paquets qui partent et qui arrivent (voir la section 9.3).

#### 10.1.2.5 L'interface avec le reste de la machine

Les PEs peuvent communiquer avec le monde extérieur par l'intermédiaire des liaisons séries contrôlées par les HyperCom, à savoir :

- la sortie vidéo `DonVideo` qui va vers la section vidéo du processeur scalaire ;
- les 4 signaux `GlobalOr<0:3>` qui permettent au processeur scalaire de récupérer des valeurs au niveau d'un ou plusieurs PEs par tranche de 4 bits ;
- les 4 entrées-sorties `ESParal<0:3>` qui permettent de relier des entrées-sorties à haut débit à la machine ;

---

11. On pourrait prendre une toute autre valeur mais ce choix simplifie grandement les problèmes de synchronisation entre le circuit de communication et le bus du PE.

- le réseau `Reseau<0:31>` que l'on peut utiliser en plus comme entrées-sorties parallèle à très haut débit si le système précédent ne suffit pas. En plus cela a l'avantage de pouvoir faire des échanges en mode parallèle plutôt qu'en mode série, vu qu'on dispose de 32 fils, ce qui économise le passage à travers des PAV et simplifie l'interface avec le dispositif d'entrées-sorties.

### 10.1.3 Quelques problèmes rencontrés

#### 10.1.3.1 La technologie employée

La majorité des problèmes rencontrés sont liés aux choix technologiques effectués pour la réalisation du prototype, à savoir l'emploi d'une technologie « *wrapped* ».

Bien que nous ayons utilisé des plaques avec condensateurs de découplages entre plans de masse et d'alimentation spécialement prévu pour les fréquences d'horloge que nous utilisons, les problèmes de diaphonie<sup>12</sup> entre les fils et de propagation des signaux sont TRÈS présents, en plus des problèmes d'adaptation de lignes de transmission.

En fait le problème est non pas au niveau de la vitesse d'horloge en soit (20 MHz) qu'au niveau des temps de transition de la logique utilisée : les PALS de 7,5 ns et les boîtiers logique en technologie FCT de 4 ns génèrent des signaux qui possèdent des temps de transition de l'ordre de la nanoseconde, soit une fréquence fondamentale de l'ordre du GHz.

Avec le wrapping et ces temps de transition, on prend cruellement conscience que les lois de MAXWELL existent toujours alors que la notion de « *logique* » devient de plus en plus floue... On passe plus de temps à rajouter des résistances de terminaisons et des condensateurs pour absorber les « *glitches*<sup>13</sup> » d'une nanoseconde qu'à concevoir le schéma de la machine complète.

Bien que ce genre de problèmes apparaisse amplifié par la technologie dans notre cas, ils survient néanmoins de plus en plus souvent même avec des circuits imprimés avec l'augmentation des performances, entraînant une réapparition des résistances sur les cartes purement digitales à cause des problèmes d'adaptation d'impédance.

Un des problèmes des « *overshoots* » — surtensions provoquées par les inductances des fils qui font que le potentiel de ces derniers dépassent beaucoup la tension d'alimentation — avec les PALS est que ces derniers croient être reprogrammés à cause de la présence des surtensions [Cue92]. En effet, les PALS sont normalement programmés grâce à une surtension sur certaines de leurs pattes. Dans notre cas ils restent alors dans un état bizarre pendant quelques nanosecondes, ce qui ne manque pas de mettre la machine à mal. Ce problème semblent corrigés avec les derniers modèles de PALS rapides qui sont vendus tout récemment.

Mais on espère que le bruit et les problèmes d'adaptation d'impédance seront fortement réduits par l'utilisation des nouveaux circuits logiques rapides incluant une résistance série de 25  $\Omega$  sur chaque sortie. Cela a aussi pour effet d'adoucir les fronts, donc de générer moins de diaphonie, et de réduire le « *ground bounce*<sup>14</sup> ». Néanmoins

12. Il s'agit ici du couplage électromagnétique.

13. Petits parasites très rapides pouvant perturber le fonctionnement normal des circuits logiques.

14. Lorsque un circuit fournit du courant sur ses sorties, une différence de potentiel apparaît aux bornes des fils d'alimentations, modélisés par une résistance et une self en série, qui perturbent les niveaux de référence du circuit et donc toutes les sorties du circuit. Plus le front de sortie est raide et plus ce bruit est important.

ces problèmes ne seront vraiment supprimés que lorsque la machine sera réalisée avec des circuits imprimés plutôt qu'avec des cartes « wrappées ».

On pourrait argumenter que dans ce cas il ne fallait pas essayer de construire une machine si c'était pour essayer d'en faire un exemplaire en technologie wrappée. Le problème des petites équipes comme la notre est que les moyens financiers sont insuffisants pour pouvoir investir dans un environnement logiciel complet permettant la simulation fine de la machine complète avant même d'en avoir construit le moindre morceau.

Un des avantages du wrapping est tout de même son côté construction incrémentale : il est relativement simple, du moins au début tant qu'il n'y a pas encore trop de fils, de faire des modifications au prototype rapidement, de corriger des erreurs alors qu'il faudrait faire réaliser un nouveau circuit imprimé à chaque fois dans le cas contraire. Une solution intermédiaire serait peut-être une technologie mixte du pauvre : on réalise les chemins de données (les bus) en circuit imprimé car il y a peu de risques de se tromper et peu de modifications alors que toute la partie contrôle est wrappée. Comme le nombre de fils wrappés est beaucoup moins important, la diaphonie entre ces fils est plus faible, cette dernière étant très importante dans les gros bus wrappés.

Enfin, cela oblige aussi à construire réellement une machine, sans se contenter d'une simulation informatique, ce qui révèlent tous les détails électriques qui ne sont pas vus lors d'une simulation logique de la machine. En effet, la simulation de la machine ne peut pas être faite au niveau électrique car cela nécessiterait des moyens de calcul beaucoup trop importants.

Et le fait d'avoir une machine réelle a aussi un côté certainement plus motivant et passionnant qu'une simulation sur un ordinateur...

### 10.1.3.2 Extrait du carnet de bord de la machine

En général, on décrit toujours une machine comme fonctionnant du premier coup, sans parler de tous les problèmes de mise au point qui peuvent survenir (et qui surviennent !) alors que ce sont eux qui permettent d'apprendre et de progresser et que c'est peut être la partie la plus intéressante pour une personne qui se passionne pour la vie des machines informatiques. On ne dit jamais ce qui n'a pas marché...

C'est pourquoi un extrait de l'évolution est donné sur la figure 10.10 qui mentionne quelques problèmes que l'on a rencontrés et mis du temps pour les trouver, chose pas très simple dû à l'interaction entre problèmes matériels et logiciels qui sont très liés, plus les problèmes matériels qui agissent entre eux, voire se compensent, ce qui peut être le pire : on en arrive parfois à se demander comment cela a pu fonctionner !

## 10.2 Description physique

L'intégration de la machine dans une petite baie 19" 12U est importante car c'est elle qui a justifié certains choix du projet, tel que le nombre de processeurs, le mode de parallélisme, le réseau, la consommation électrique, le niveau sonore, etc.

Cette partie décrit la manière de faire de POMP une petite machine de bureau.

<b>9-89 :</b> Choix du MC88100 comme PE.	en cas d'exception. 1d88 récupère SIGINT et l'envoie comme interruption au MC88100.	registres, pour raison de métastabilité potentielle ;
<b>11-89 :</b> Début de la construction de la machine.	<b>13-04-1990 :</b>	- glitch sur cont_seq/-arbicont.pld/!SelCont.
<b>17-12-89 :</b> Les premiers programmes commencent à tourner sur le processeur scalaire : POMP sait compter en mémoire. Programme écrit directement en assembleur...	- Début du bus de contrôle données commun au séquenceur et à VME ; - écriture du PAL de génération des interruptions vers le bus VME ;	<b>20-08-1990 :</b> - Les modifications ont été faites ;
<b>01-03-1990 :</b> Bug sur la RAM de données du séquenceur : oubli que !writel n'est plus valable après !ack au VME...	<b>01-05-1990 :</b> Bug d'asynchronisme dans les PALS geredr.pld & cont_prg.pld.	- nouveau PAL de génération du signal d'écriture, gennweb.pld ; - ValideVme arrive un cycle trop tôt => nouveau geredr.pld/RamDon2 nécessaire.
<b>Semaine à partir du 15 :</b> Mise en « rack » de POMP avec Magnolia (notre SUN 3/110 et 20 Mo de mémoire vive.	<b>03-05-1990 :</b> Correction sur decode_b.pld qui passe en 16R8.	<b>30-11-1990 :</b> Avec le 2440 prêté par Tektronix, on voit les glitches de 5 ns sur PCSB<> alors qu'on est en écriture ! Ce problème semble résolu par l'ajout des condensateurs <i>ad hoc</i> ...
<b>30-03-1990 :</b> - Les programmes compilés à partir du 27 (coupure de courant...) ne fonctionnent plus. Pourquoi ? - inversion des fils MCS<0:3> entre decod88.pld et la mémoire de donnée ; - réécriture du chargeur de POMP 1d88. La gestion des exception dans sys.o est aussi modifiée.	<b>09-05-1990 :</b> Lorsqu'on fait attendre le MC88100, l'adresse sur le bus P est la nouvelle et non l'ancienne. Il ne faut donc pas verrouiller les verrous d'adressage lorsqu'on attend. Problème qui concerne la gestion du bus pipeliné du MC88100.	<b>26-04-1991 :</b> - L'oscilloscope et l'analyseur logique reçus fonctionnent ; - il y a des glitches partout, certains d'1 ns !
<b>Début avril :</b> Affichage du contexte	<b>19-07-1990 :</b> - DBEx du séquenceur étaient tout le temps échantillonnés, même en cas de conflit avec le bus VME ; - il faudra améliorer la modification du 03-05-1990 pour passer le signal dans 2 coups de	- problème dans 1d88 quelque part. Mauvaise initialisation ? <b>3-05-1991 :</b> Bug corrigé dans 1d88 pour les accès non alignés dans les entrées-sorties.

FIG. 10.10 - Un extrait du journal de bord de la machine.

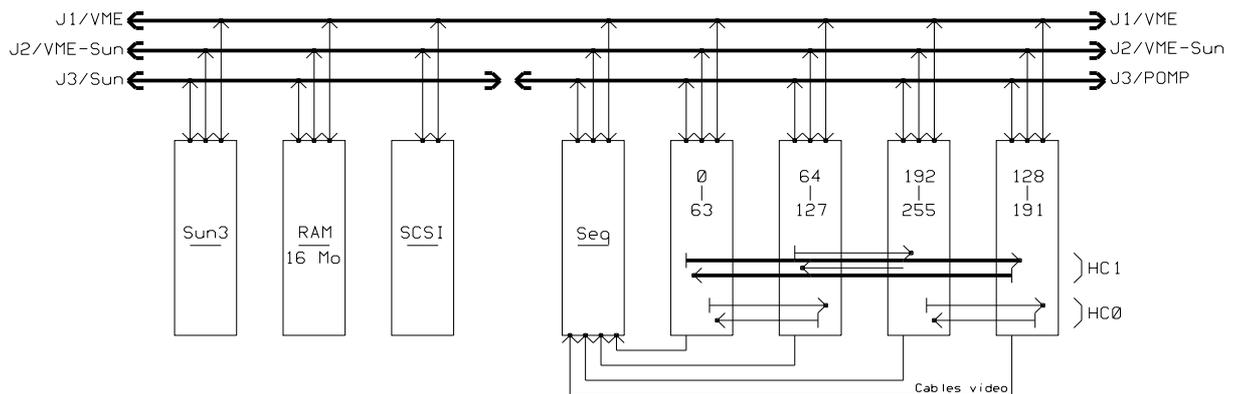
## 10.2.1 Les bus de la machine

Le synoptique du câblage de la machine est indiqué sur la figure 10.11, dans le cas où on considère une machine avec 4 cartes de PES.

### 10.2.1.1 Le bus VME de Sun

Comme on utilise un SUN standard, un 3/110, les cartes sont au format 9U et le bus est composé de la partie VME standard sur J1 et une partie de J2, tandis que le reste des connecteurs, J2 et J3 forment un bus rapide spécifique SUN.

Lorsque nous avons commencé la machine, il n'y avait pas de fond de panier compa-

FIG. 10.11 - *Synoptique du câblage de la machine.*

tibles SUN en vente qu'on aurait pu adapter. Il a fallu en construire un. Par malchance, la spécification de ce bus n'était pas connue de SUN France<sup>15</sup>. Seul un vendeur de fonds de paniers en avait eu les spécifications par un client pour qui il avait fait ce genre de fond de panier et ne voulait pas les donner si on ne lui commandait pas un fond de panier sur mesure. Or le « sur mesure » coûte cher...

Finalement, en auscultant à l'ohm-mètre un SUN qu'on avait démonté, on a pu copier un fond de panier qu'on a wrappé sur un J2 VME standard et un J3 à wrapper, juste pour permettre de rajouter la carte mémoire rapide spécifique au bus SUN, l'interface SCSI se contentant elle du bus VME standard, plus lent.

### 10.2.1.2 Le bus de contrôle et de commande de la machine

Si la partie SUN utilise effectivement les 3 connecteurs, notre carte processeur scalaire n'utilise que le bus VME, il reste J3 et 2/3 de J2 de libre pour faire passer les instructions parallèles et le contrôle entre le processeur scalaire et les processeurs parallèles.

Comme il y a plus de fils disponibles que nécessaire, on pourra toujours démultiplexer les instructions d'un facteur 2 ou 3, lorsqu'on voudra accélérer la machine et qu'on aura des problèmes de transmission.

On peut même utiliser le J1/VME et J2/VME-SUN à tout autre chose que des transferts de type VME si on n'a pas besoin de faire des accès aux cartes des PE depuis le bus VME, ce qui est le cas *a priori*. Dans ce cas, le bus VME ne dépassera pas la carte SCSI. Dans notre prototype, le bus VME traverse néanmoins tout le rack 19" car cela nous laisse plus de liberté quand aux cartes qu'on peut rajouter.

En ce qui concerne la distribution d'horloge, on peut se permettre de la distribuer depuis l'oscillateur principal sur la carte du processeur scalaire vers chacune des cartes regroupant des PEs via autant de câbles coaxiaux de longueurs identiques afin d'avoir un décalage entre les horloges de chaque carte minimal, chaque horloge étant distribuée au niveau d'une carte par un classique arbre en « H ». Comme la machine est totalement synchrone, c'est en minimisant le décalage entre les horloges des PEs qu'ont augmentera le débit du réseau de communication entre les PEs.

15. Y compris le service de maintenance !

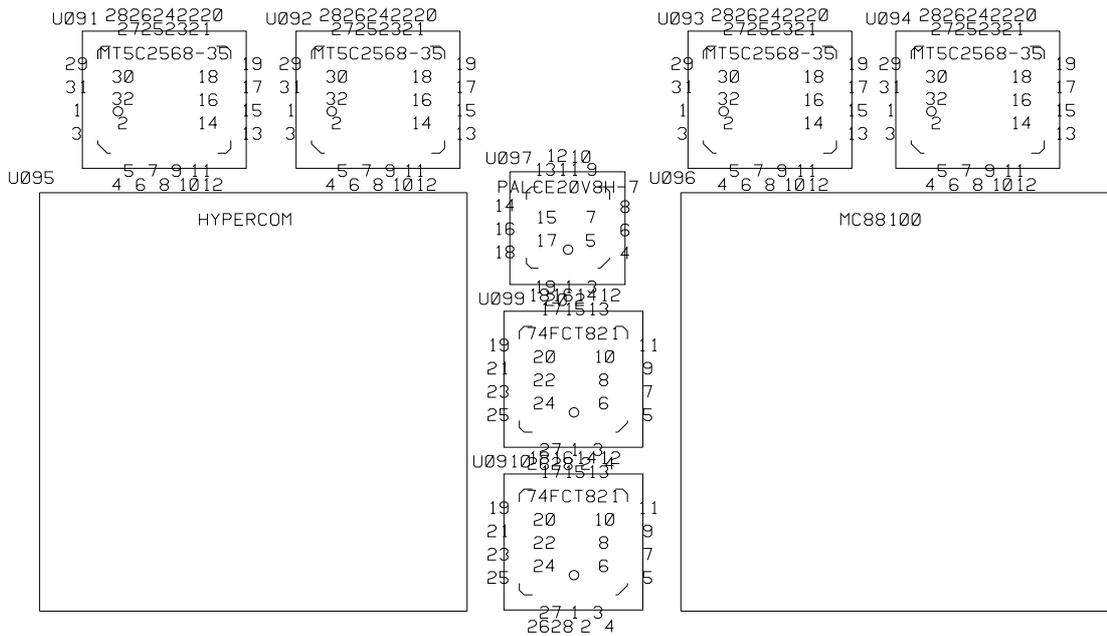


FIG. 10.12 - Schéma d'implantation d'un PE à plat.

### 10.2.2 La répartition des PEs dans la machine

Etant donnée la taille physique d'un PE, on a étudié deux implantations possibles de la machine capables d'intégrer 256 processeurs dans notre rack VME:

- 4 cartes faisant chacune office de fond de panier à 64 petites cartes de PE ;
- 16 cartes comportant chacune 16 PES.

Chaque carte au format triple Europe fournit une surface de l'ordre de 35 cm × 36,6 cm que l'on peut paver de différentes manières.

Il faut bien évidemment se restreindre à des racks de taille commerciale car le développement de racks, baies et fonds de panier spécifiques sort totalement du cadre de notre étude.

Dans le premier cas, ce qui limite est surtout la hauteur des modules. En effet, comme l'hôte et la carte du processeur scalaire occupent 5 emplacements, il reste 16 emplacements VME pour les 4 cartes à 64 PES, soit encore 81,28 mm. Il faut donc que les petits modules aient une hauteur limitée à 6 cm environ, compte tenu de la hauteur réservée aux connecteurs de type DIN 96 qui les lient à la carte mère.

Chaque module mesure 6 cm de haut sur 11,2 cm comme l'indique son implantation figure 10.12. Le problème est qu'on est obligé de réaliser le module en technologie double face, ce qui laisse peu de place à une évolution ultérieure comme du rajout de mémoire sans diminution du nombre de processeurs. Néanmoins on peut placer ces modules selon 3 rangées de PES espacés de 1,5 cm sur les 4 cartes mères.

L'autre solution qui consiste à avoir 16 cartes comportant chacune 16 processeurs disposés à plat permet d'éviter les contraintes de partitionnement strictes aux prix d'une perte de modularité et d'économie d'échelle mais par contre on économise toute la connectique supplémentaire nécessaire aux modules.

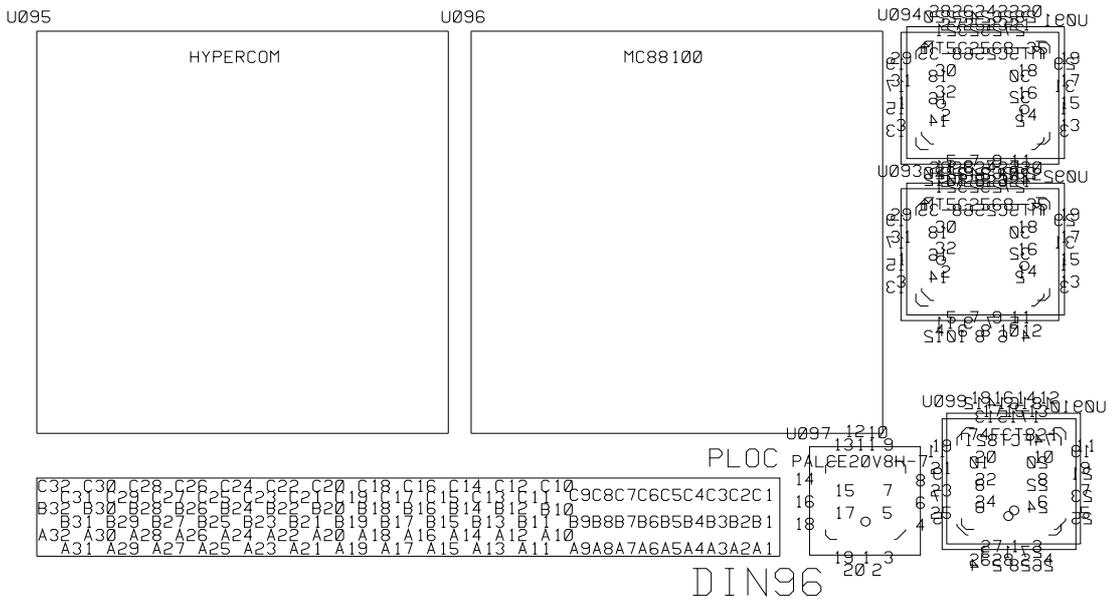


FIG. 10.13 - Schéma d'implantation d'un PE en module.

On peut alors paver les 16 cartes avec les blocs indiqués sur la figure 10.13 mesurant 11,6 cm×6,6 cm. Comme on peut déplacer les composants légèrement on gagne en densité tout en restant en simple face puisqu'on économise en plus la place du connecteur. Si on veut réaliser une machine avec une capacité mémoire doublée ou quadruplée, il suffit de réaliser la carte avec des composants de chaque côté, puis avoir des boîtiers mémoire verticaux pour accroître encore la capacité mémoire.

Pour ces raisons, nous préférons la dernière solution, avoir une machine en 17 cartes, sans compter la machine hôte : une carte processeur scalaire et 16 cartes comportant 16 PEs chacune.

### 10.2.2.1 Le réseau de communication

Il utilise des connecteurs plats entre les cartes car il n'y a pas assez de fils qui peuvent passer au travers des connecteurs J1 et J2 (96 pattes chacun moins les pattes d'alimentation et de masse) laissés libres.

On aura intérêt à utiliser des câbles en nappe formés de câbles coaxiaux dont l'impédance est bien contrôlée et qui réduisent très fortement la diaphonie ainsi que les parasites électriques.

Bien évidemment le motif de câblage dépend du réseau choisi pour la machine, mais dans le cas où on part du réseau type hybride décrit au chapitre 9 on peut proposer le câblage qui va suivre. Cela peut paraître trivial de décrire comment faire un hypercube. En fait, si on veut le réaliser on a des contraintes supplémentaires, en particulier éviter les torsions longitudinales des câbles en nappe, ne pas avoir à réaliser autant de circuits imprimés qu'il y a de nœuds dans le réseau, ne pas avoir de câbles en nappe qui doivent s'interpénétrer (!), qu'il faut respecter.

Déjà, si on rajoute des câbles en nappe, on veut qu'ils soient sur la face avant seulement, et non pas aussi en haut et en bas, de la machine afin qu'elle soit facile à construire et à démonter. Il faut donc déjà projeter l'hypercube dans le plan. On

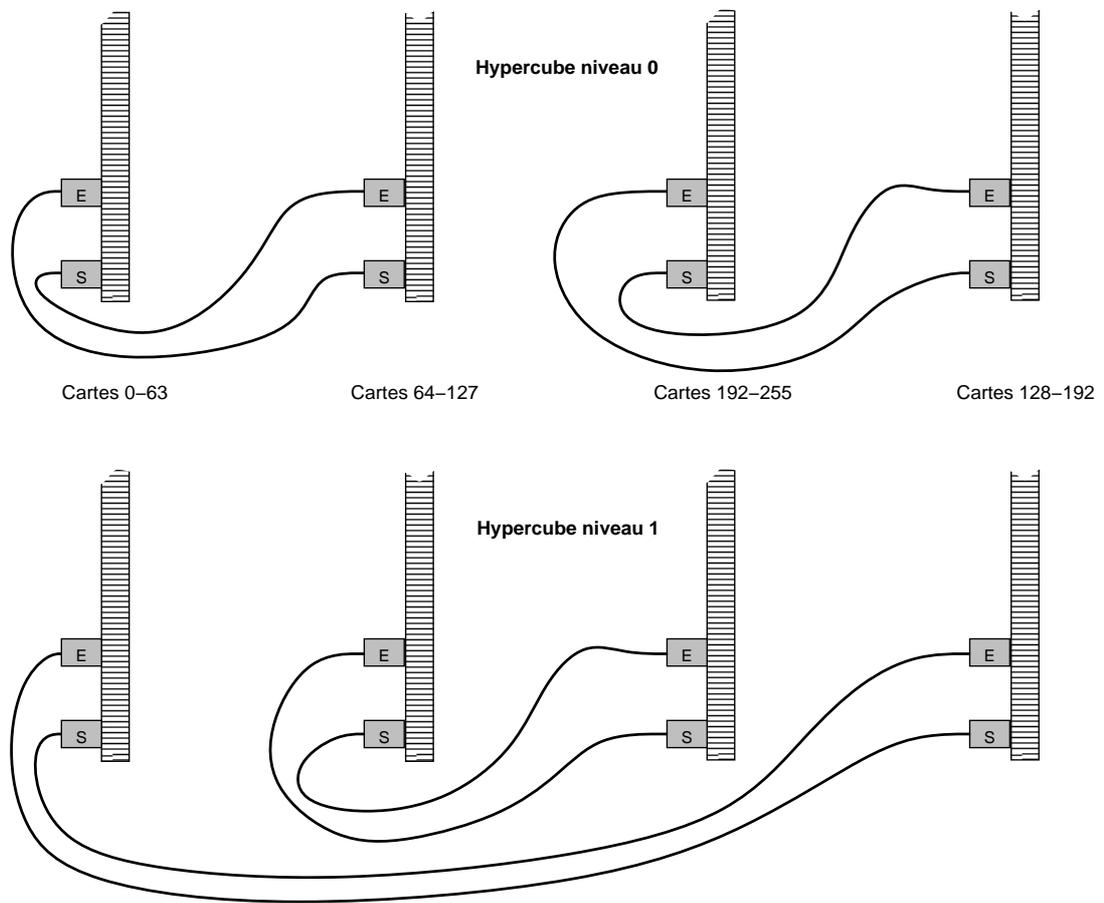


FIG. 10.14 - *Synoptique du câblage du réseau entre carte dans le cas d'un hypercube.*

démontre facilement que cela peut se faire si on range les cartes dans le rack en suivant un code de GRAY pour énumérer les nœuds du réseau [Ker89].

Ensuite il faut respecter la contrainte d'avoir tous les circuits imprimés identiques, donc d'avoir toutes les entrées du réseau au même endroit sur chaque carte et la même chose pour toutes les sorties, chaque sortie étant reliée à une entrée. Cela est fait comme indiqué sur les deux vues en coupe suivant un plan horizontal de 2 niveaux de l'hypercube de la figure 10.14, dans le cas d'une machine réalisée en 4 cartes pour simplifier le dessin. Chaque câble plat de la figure peut être composé en fait de plusieurs câbles plats accolés, ce qui n'enlève rien à la généralité de la méthode.

### 10.2.3 Les entrées-sorties

Le fond de panier VME est sous-utilisé là où il y a des cartes processeurs car celui-ci ne sert qu'à envoyer des instructions, ce qui représente de l'ordre d'1/8<sup>ème</sup> du nombre de broches des 3 connecteurs DIN de 96 broches.

On peut donc très bien utiliser ces broches disponibles pour connecter un sys-

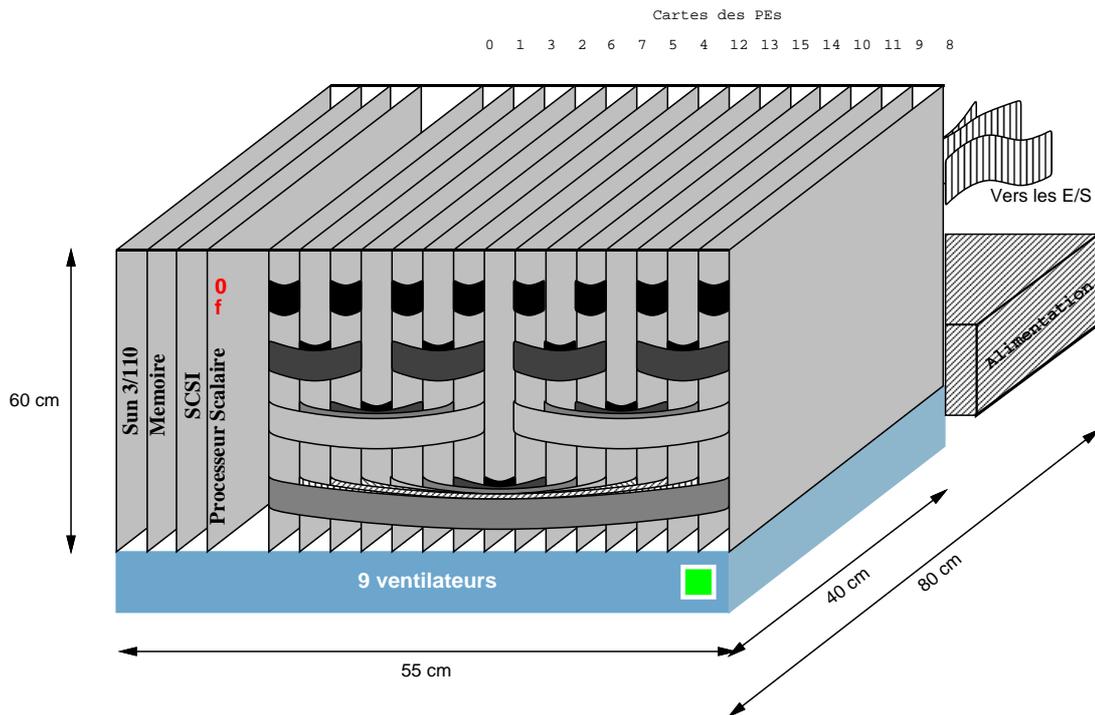


FIG. 10.15 - Organisation de la machine dans sa baie triple Europe.

tème d'entrées-sorties rapide comme des tableaux de disques du commerce (RAID), par exemple.

#### 10.2.4 L'intégration dans la baie

Elle est indiquée sur la figure 10.15.

Il faut remarquer que les cartes des processeurs sont arrangées suivant un code de GRAY pour pouvoir réaliser la connexion et chaque câble plat représenté est constitué d'en fait 2 câbles plats arrangés comme décrit dans la section 10.2.2.1.

La connexion à des entrées-sorties rapides se fait par l'arrière de la machine par autant de câbles plats que de cartes de PEs. Cela nécessite une baie supplémentaire pour la connexion à ces entrées-sorties, mais comme celles-ci occupent une place importante de toute manière si on veut utiliser pleinement le débit d'entrée-sortie, cette baie possède une taille négligeable.

### 10.3 Conclusion

Nous avons exposé une possibilité de construction de POMP ne nécessitant qu'un développement technologique minimal, basé sur des produits existants en standard, une technologie « froide » et en particulier ne faisant pas intervenir le développement de circuit intégré spécifique.

Cela permet de développer une machine qui offre des possibilités intéressantes à

un prix raisonnable (de l'ordre de 3000 FF par nœud<sup>16</sup>), une taille modeste puisque la machine loge dans une baie 19" 9U et une consommation électrique de l'ordre du kw : les objectifs que nous nous étions fixés dans le cahier des charges du projet sont donc pleinement respectés.

Par contre le gros problème provient de notre manque de moyen en environnement de CAO professionnelle pour mener à terme un prototype complètement fonctionnel de la machine : pas d'environnement de spécification matérielle de haut niveau (de type VHDL par exemple), pas de simulateur multi-niveau possédant tous les modèles de circuits récents utilisés permettant de prévoir le comportement du prototype avant tout développement matériel, etc.

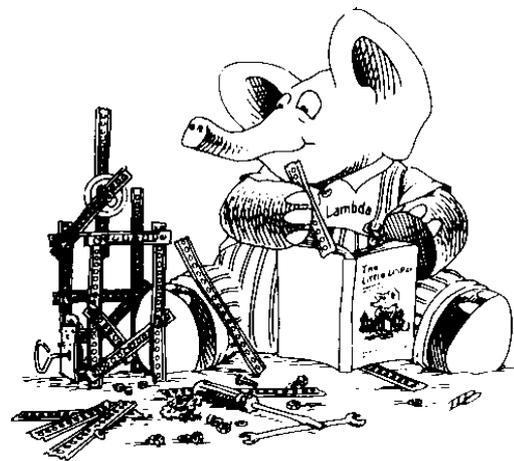
Cela nous a amené à nous engager dans la seule issue qui nous semblait possible et avait déjà été utilisée dans le laboratoire : la conception d'un prototype en wrapping, technologie intéressante car elle permet avec peu de moyens financiers (mais par contre avec plus d'efforts humains) de pouvoir faire de la conception incrémentale. On peut construire des sous-parties de la machine, les tester et de les modifier jusqu'à correction. Malheureusement, on a des problèmes dès qu'on veut utiliser des circuits de logique modernes produisant des signaux à fronts très rapides car cela met en exergue tous les problèmes électriques qui rendent la réalisation peu fiable : diaphonie entre fils, mauvaise adaptation d'impédance, etc. Vus les problèmes rencontrés, on perd en fait tous les avantages qui nous avait fait choisir cette technologie : cela a clairement été un mauvais choix mais nous n'avions guère le choix.

Un autre temps non négligeable a été investi à développer des logiciels d'interface entre les différents outils que nous avons récupérés, tels qu'un programme pour permettre de transformer la description de PALS en un format compatible avec le simulateur décrit dans [Par91] ou encore permettre l'affichage de formes d'onde sous notre éditeur de schémas. Tout ce temps doit pouvoir être économisé à condition d'avoir un environnement de développement homogène et tenu à jour constamment.

On aboutit donc à une situation paradoxale : même pour faire une machine simple et peu chère, il faut un environnement de développement performant et très cher. Cet investissement ne peut être rentabilisé que dans l'optique d'une production minimale, c'est-à-dire avec une finalité industrielle.

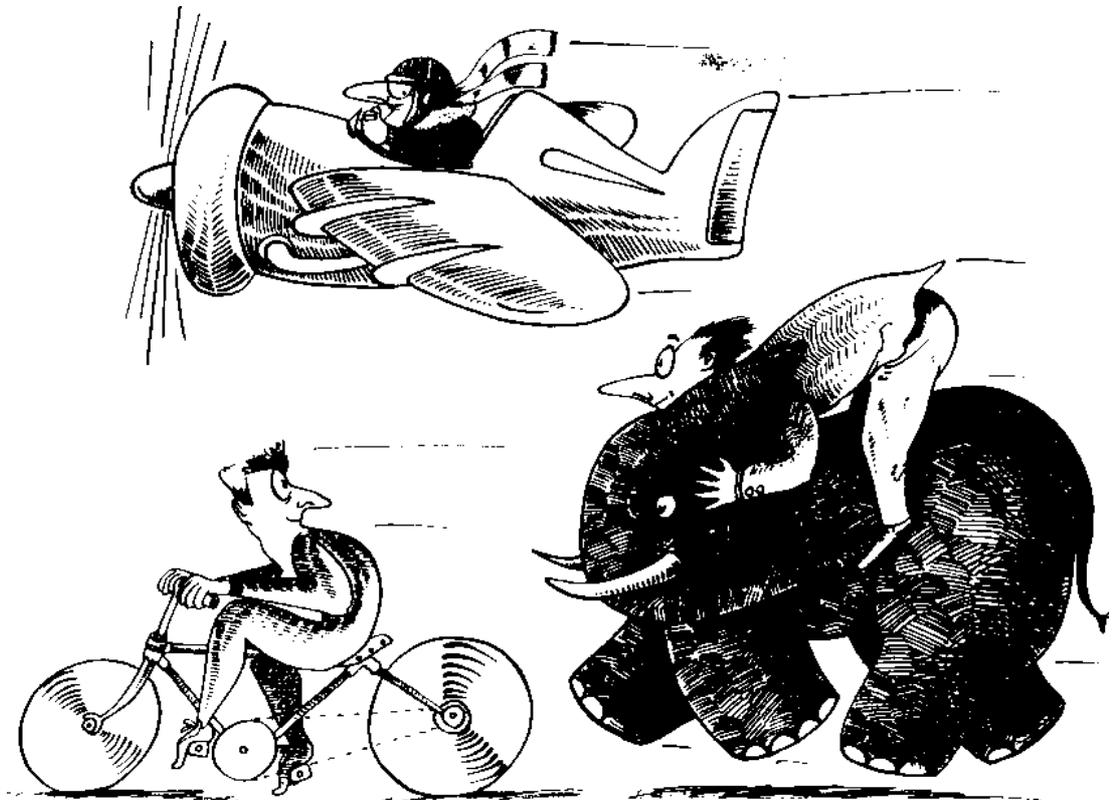
---

16. En fait ce qui coûte le plus cher n'est pas le processeur (50 \$) mais le circuit logique programmable. Donc en production, on aurait intérêt à en faire un circuit spécialisé pour baisser le coût aux environs du millier de francs.



Chapitre **11**

**Divertissement sur la  
superlinéarité**



**S**E chapitre aborde un problème lié aux mesures de performances de certains algorithmes sur les machines parallèles, en particulier la question philosophiquement intéressante de savoir si on peut accélérer un programme plus que le nombre de processeurs utilisés dans la machine exécutant le programme par rapport à une machine à un seul processeur.

Ce qui choque *a priori* dans la présentation de cette question est qu'on a l'impression que cela revient à dire que dans le tout il y a plus que dans la somme des parties, ce qui va à l'encontre du sens commun.

## 11.1 Notions d'accélération et de linéarité

Depuis toujours les humains ont possédé la fâcheuse habitude de tout vouloir comparer et ils se sont empressés de comparer les ordinateurs entre eux dès qu'il y en eut plus d'une seule sorte.

Très simplement, une méthode de mesure intuitive est de comparer les temps d'exécution d'un programme sur les deux machines à comparer. Le rapport entre les deux fournit une indication de rapidité relative, improprement dénommée (au sens physique du terme) accélération :

$$a = \frac{T_1}{T_2}$$

Avec l'introduction des machines parallèles, les informaticiens ont voulu mesurer de la même manière l'efficacité du parallélisme, c'est à dire le gain en vitesse d'exécution relative de leur programme en fonction du nombre de processeurs  $N$  de la machine. Il suffit d'écrire :

$$a = \frac{T_1}{T_N}$$

où  $T_1$  et  $T_N$  sont les temps d'exécution du programme sur les machines à 1 et  $N$  processeurs respectivement.

On appelle cette accélération *absolue* [SN90] lorsqu'on compare le meilleur algorithme séquentiel avec le meilleur algorithme parallèle :

$$a = \frac{S}{T_N}$$

$S$  étant le temps d'exécution séquentiel.

L'intérêt d'avoir la notion d'accélération *relative* est que celle-ci dépend souvent du nombre  $N$  de processeur plus subtilement que de manière linéaire [SN90].

On peut même ramener l'*efficacité* du parallélisme par processeur, c'est la notion de *linéarité*, qui représente la contribution ou l'efficacité de chaque processeur à l'accélération de l'exécution du programme :

$$\ell = \frac{a}{N} = \frac{T_1}{N \cdot T_N}$$

Plusieurs modèles ont été développés pour estimer l'accélération d'un programme qu'on peut obtenir grâce à une exécution sur une machine parallèle.

### 11.1.1 Taille du problème fixe

En ce qui nous concerne, on peut dire en simplifiant que des programmes qui s'exécutent sur des machines parallèles comportent deux parties :

- une partie séquentielle, dont le temps d'exécution est quasi-indépendant du nombre de processeurs parallèles,
- une partie parallèle, dont le temps d'exécution va décroître lorsqu'on va augmenter le nombre de processeurs parallèles.

Considérons les temps d'exécutions sur les machines séquentielle et parallèle :

$$\begin{aligned} T_1 &= s + p \\ T_N &= S + P \end{aligned}$$

Pour simplifier la comparaison, on peut normaliser en fixant

$$T_1 = 1$$

Si on reprend les hypothèses simplificatrices habituelles<sup>1</sup> [Bre74, FLW86, FLW87, SN90], à savoir principalement qu'une instruction du processeur séquentiel sera exécutée sur un processeur élémentaire en le même temps :

$$\begin{aligned} S &= s \\ P &= \frac{p}{N} \end{aligned}$$

c'est à dire que la partie séquentielle est inchangée par le nombre de processeurs tandis que le temps d'exécution parallèle est divisé par le nombre de processeurs, l'accélération devient [Amd67] :

$$a = \frac{1}{s + \frac{p}{N}}$$

et la linéarité :

$$\ell = \frac{1}{p + Ns}$$

On en déduit que quelque soit le nombre de processeurs de la machine on ne dépassera jamais l'accélération maximale

$$a_\infty = \frac{1}{s}$$

et que l'efficacité de chaque processeur tend vers 0. Les machines parallèles semblent donc être d'un intérêt très limité.

Pourquoi, alors, en être déjà à la page 254 d'une thèse qui parle de parallélisme et de machines parallèles ?

Peut-être tout simplement parce qu'on considère des machines ayant de toute manière qu'un nombre assez faible de processeurs ou que l'établissement du problème ne correspond pas toujours aux conditions réelles d'utilisation des ordinateurs parallèles. En outre, la modélisation précédente est assez sommaire et elle suppose en particulier [Fly72] :

- pas de recouvrement d'exécution des parties scalaire et séquentielle,

---

1. Hélas rarement vérifiées mais qui a l'intérêt de prouver ce que l'on veut, sortie du contexte de [Bre74] par exemple...

- pas d'évolution dans la notion de « programmes conventionnels »,
- que la partie « séquentielle » ne peut s'exécuter que sur un seul processeur [Kuc76].

### 11.1.2 Temps d'exécution fixe

Une approche plus psychologique est de dire qu'un utilisateur possédant une machine plus rapide et qui, par exemple, fait de la modélisation va vouloir affiner son modèle, c'est à dire exécuter son programme sur un ensemble de données plus important reflétant mieux son problème. Plusieurs études historiques ont été faites qui vont dans ce sens et montrent qu'en un siècle ce sont les problèmes qui ont évolués avec la rapidité des machines alors que les temps d'attente de l'utilisateur restaient du même ordre de grandeur [GREC91].

Dans ce cas là, ce n'est plus la taille du problème qui est fixe mais plutôt le temps d'exécution qui est du même ordre de grandeur et le but est d'avoir la solution la plus précise possible en un temps donné.

On ne compare plus l'exécution d'un programme sur une machine séquentielle à l'exécution sur une machine parallèle mais l'exécution d'un programme sur une machine parallèle à l'exécution sur une machine séquentielle [Gus88].

$$\begin{aligned} T_1 &= s + p \\ T_N &= S + P \end{aligned}$$

Pour simplifier, on fixe cette fois-ci

$$T_N = 1$$

et on prend comme hypothèses simplificatrices

$$\begin{aligned} s &= S \\ p &= N \times P \end{aligned}$$

d'où finalement :

$$\begin{aligned} a &= 1 + (N - 1)P \\ \ell &= 1 + \frac{1 - P}{N} \end{aligned}$$

On peut donc dépasser l'accélération prévue par AMDAHL à condition de s'attaquer à des problèmes massivement parallèles, ce qui ébranle fortement la croyance anti-parallélisme...

Bien entendu il faut affiner ce modèle en considérant quel est le temps exact que l'utilisateur veut bien allouer à la nouvelle machine pour résoudre son problème plus compliqué et surtout l'évolution de  $S$  et  $P$  en fonction de  $N$ , ce qui peut changer beaucoup d'un problème à l'autre.

Enfin il est probable que ce modèle est plus adapté à comparer des machines très différentes en performance tandis que le modèle à taille de problème fixe doit servir à comparer des machines de performance semblables *a priori*, où les temps d'exécutions ne sont pas très différents.

### 11.1.3 Modèle à mémoire constante par processeur

Un autre modèle intermédiaire a été introduit pour comparer des ordinateurs à nombre de processeurs différents où on considère que la taille des problèmes étudiés suit la mémoire disponible [?, SN90].

L'inconvénient est que cela ne tient compte d'aucun facteur humain : on ne sait pas quel sera le temps d'attente de l'utilisateur puisqu'il n'y a souvent pas de relation simple entre la mémoire utilisée par programme et son temps d'exécution.

Néanmoins ce système de mesure se rapproche du précédent puisque ce qui importe est d'avoir la solution la plus précise possible d'un problème en utilisant toute la mémoire de la machine, en considérant que lorsqu'on rajoute des processeurs à un ordinateur, on rajoute souvent de la mémoire en quantité au moins équivalente, dans le cas de machines à mémoire distribuée.

Comme souvent la demande en calcul parallèle augmente plus vite que celle en mémoire, la proportion de calcul scalaire diminue et par conséquent apparaît la superlinéarité.

## 11.2 La superlinéarité à taille fixée existe-t-elle ?

On a évoqué quelques visions de l'accélération de l'exécution d'un programme due au parallélisme qui toutes se basent sur une division par le nombre de processeurs du temps de calcul de la partie parallèle.

De nombreuses recherches ont montré qu'on peut déjà arriver à avoir des accélérations linéaires sur des problèmes qui semblent intrinsèquement séquentiels, comme des manipulations de listes chaînées [KRS85] par exemple.

Peut-on espérer dépasser cette accélération linéaire dans certains cas ? Comme on s'intéresse au temps réel vécu par l'utilisateur, même dans le cas où

$$a_N = \mathcal{O}(N)$$

on est particulièrement sensible à la constante  $\ell$  si

$$a_N \sim \ell N$$

ce qu'on a tendance à négliger lors des études de complexité [FLW86, FLW87], qui sont une des conséquence du refus de modéliser une certaine réalité<sup>2</sup>.

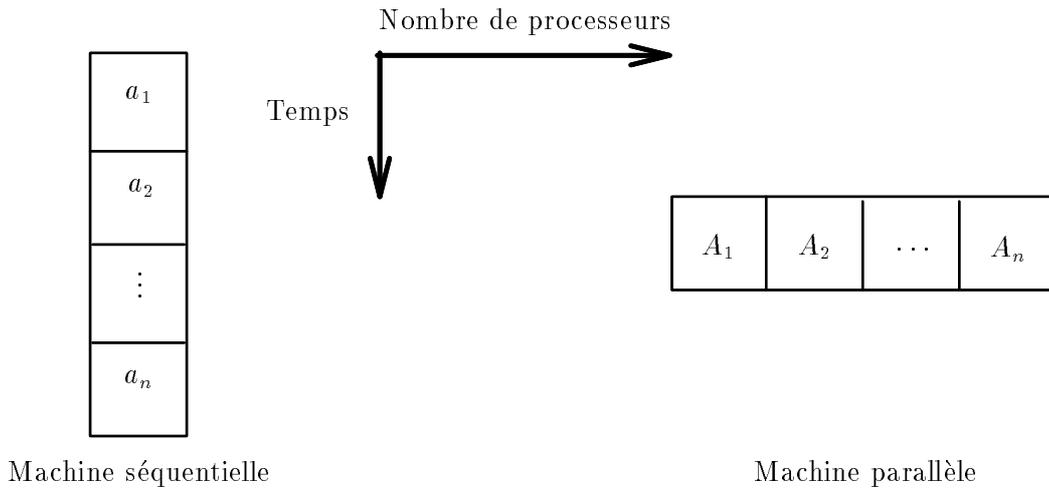
Dans le cas du modèle à temps d'exécution fixé, il peut y avoir superlinéarité car souvent lorsqu'on augmente la taille d'un problème par processeur l'efficacité des processeurs augmente [GREC91]. C'est d'ailleurs souvent la même chose pour une machine monoprocesseur<sup>3</sup>.

Mais dans le modèle à taille de problème fixée, avoir une accélération superlinéaire, c'est à dire  $\ell > 1$ , est impossible car cela correspondrait à avoir  $p > 1$ .

---

2. On trouve par exemple dans [FLW87] « It is important to note that this statement is a theorem, which follow logically from two assumptions : (1) hardware considérations are ignored » (sic!). On peut se demander à juste titre la valeur d'une telle modélisation puisque cet article ne considère en aucun cas des notations d'ordre de grandeur  $\mathcal{O}()$  mais bel et bien des valeurs exactes...

3. Les principales raisons pour que ce ne soit pas vrai sont les limitations des tailles de cache, de registres et de mémoire physique qui créent des discontinuités dans les temps d'exécution.

FIG. 11.1 - *Modèle d'exécution sur machine séquentielle et parallèle.*

En fait, tout provient du fait d'une généralisation abusive de l'équivalence entre une opération globale et  $N$  opérations sur le processeur scalaire, ce qui revient à sortir cette équivalence, aussi connue sous le nom de théorème de BRENT [Bre74], de son contexte.

### 11.2.1 Modèle linéaire

Une supposition classique (dans la littérature) pour calculer l'accélération d'un programme parallèle sur une machine parallèle par rapport à une machine séquentielle est d'émuler l'exécution parallèle sur la machine séquentielle.

Supposons qu'une partie parallèle d'un programme puisse être exprimée en fonction de  $n$  tâches  $A_i$  qui s'exécutent en un même temps

$$\mathcal{P}(A_i)$$

et émulée sur une machine séquentielle par

$$p(a_i)$$

La dualité espace-temps du théorème de BRENT dans le cas d'expressions arithmétiques [Bre74] peut être représentée par la figure 11.1 qui exprime les contraintes sur les temps d'exécution :

$$\frac{T_{p(a_i)}}{T_{\mathcal{P}(A_i)}} \leq n$$

### 11.2.2 Modèle superlinéaire

On ne peut pas étendre cette relation à un ordinateur parallèle car une instruction d'une machine parallèle à  $N$  processeurs ne s'émule pas forcément en  $N$  opérations sur une machine séquentielles.

Cela est dû en particulier au fait que quand on rajoute des processeurs dans une

machine on rajoute aussi de la mémoire<sup>4</sup>, autant de registres et de mémoires caches qui font que la quantité de mémoire rapide augmente par rapport à la machine séquentielle et donc que la machine parallèle peut être plus rapide sur un problème de même taille.

Outre l'ajout de processeurs, on rajoute un réseau qui permet de communiquer entre PES ou entre les PES et la mémoire. Il offre de nouvelles possibilités que n'a pas une machine séquentielle : des exemples de programmes mettant en valeurs ces caractéristiques des PRAM, possédant un « réseau » théorique capable d'échanger des données en parallèle de manière synchrone (en 1 cycle) entre processeurs et mémoire, ont été développés en ce sens, en particulier sur des problèmes de mouvements de données qui donnent des cas d'une superlinéarité de  $2 - \frac{1}{N}$  [ACF92].

Dans la suite, on considère que tous les processeurs parallèles ou séquentiels ont une puissance comparable.

Si on applique un raisonnement espace-temps semblable à une machine SIMD au lieu d'une PRAM<sup>5</sup>, on s'aperçoit que le théorème de BRENT n'est pas valable, puisque la partie séquentielle  $S$  du programme s'exécute sur un seul processeur en même temps que la partie parallèle s'exécute sur les processeurs parallèles SIMD [Ker89, pages 50–51]. On peut donc atteindre une superlinéarité maximale là aussi de  $2 - \frac{1}{N}$  [KNS91].

En fait, si on peut paralléliser encore  $S$  de manière VLIW on peut atteindre une superlinéarité supérieure, selon les applications [Ker89, pages 25–27,51].

De manière générale, il est possible de construire une machine où on peut exécuter certaines tâches d'un programme une fois sur une machine séquentielle ou parallèle. Cela suppose d'avoir un couplage fort entre les processeurs qui exécutent ces tâches (typiquement des bus de diffusion) et donc de voir la machine comme étant VLIW-parallèle.

Le problème est de savoir si un tel programme et une telle machine résiste à la méthode de la simulation sur un seul processeur présentée dans [FLW86] pour montrer que la superlinéarité est impossible. C'est ce que nous allons voir.

Supposons qu'un programme s'exécute sur une machine séquentielle ou MIMD simple en plusieurs tâches (pouvant être très petites) de temps d'exécution égal, programme représenté par

$$p(a_{i,j})$$

auquel correspond l'exécution sur la machine parallèle d'autres tâches de temps d'exécution égal du programme

$$\mathcal{P}(A_{i \leq k}, A_{i > k,j})$$

c'est à dire qu'il peut y avoir factorisation de l'exécution des tâches parallèles  $a_{i \leq k,j}$  en  $k$  tâches  $A_{i \leq k}$  séquentielles alors que les  $(n - k) \cdot N$  tâches restantes  $a_{i > n,j}$  ne le sont pas.

On peut constater sur la figure 11.2 que la notion de puissance<sup>6</sup> n'est pas constante selon l'architecture de la machine : une machine parallèle à composante VLIW sera plus

4. Ce qui peut parfois permettre l'exécution d'un autre algorithme pour résoudre le problème considéré, plus efficace en temps au dépend de l'espace mémoire. Néanmoins on ne considère pas ici ce fait, à la limite du hors-jeu.

5. On peut considérer une machine SIMD comme étant composée d'un processeur scalaire couplé à une PRAM.

6. produit *nombre de processeurs*  $\times$  *temps* [?].

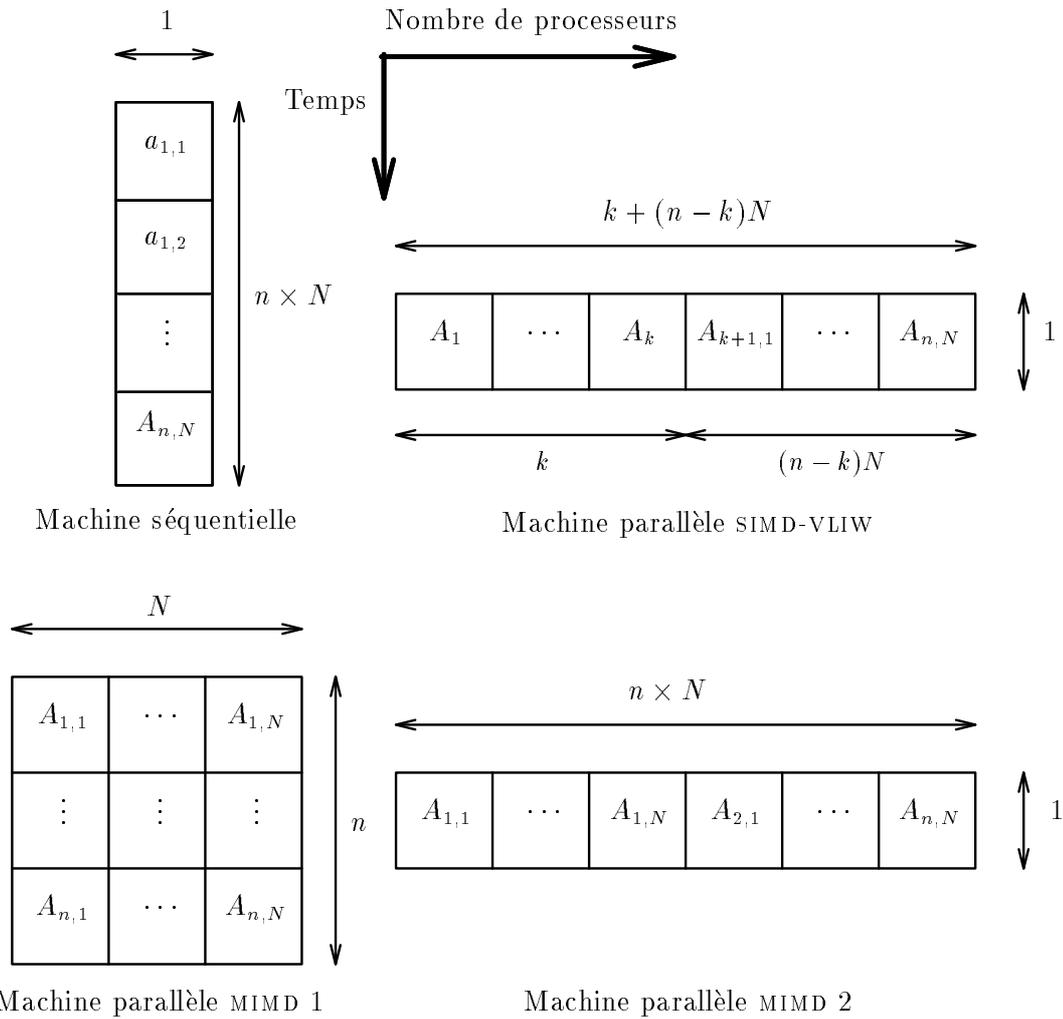


FIG. 11.2 - *Modèle d'exécution comparé avec une machine SIMD-VLIW.*

économique en processeurs. On peut en déduire la linéarité maximale<sup>7</sup> :

$$\hat{\ell} = \frac{nN}{k + (n - k)N}$$

### 11.2.3 Généralisation du modèle

Comme les temps d'exécutions des tâches parallèles et séquentielles ne sont généralement pas égales dans un programme réel, il suffit de décomposer chaque tâche en sous-tâches dont le temps d'exécution est égal au PPCM des temps d'exécution des tâches initiales, ce qui revient à étendre  $n$  et  $k$  dans  $\mathbb{Q}$ .

Mais au fait, de tels programmes et de telles machines existent-ils dans la réalité?

<sup>7</sup> C'est-à-dire lorsque chaque tâche s'exécute en un même temps puisque l'accélération est le temps d'exécution séquentielle par rapport à la plus longue tâche parallèle.

---

```

vecteur double a,b;
    ...
everywhere {
    a = b;
}

```

---

FIG. 11.3 - Programme de copie d'une variable parallèle.

---

```

BOUCLE:
    ld r2,r3[r5]           ; lit un 'el'ement de b
    st r2,r4[r5]           ; et l'affecte à un 'el'ement de a
    sub r6,r6,1            ; d'ecr'emente le compteur de boucle
    bcnd.n ne,r6,BOUCLE   ; reste-t-il du travail à faire ?
    add r5,r5,1            ; incr'emente l'index avec branchement retard'e.

```

---

FIG. 11.4 - Programme de copie de vecteur sur une machine séquentielle.

## 11.3 Une machine superlinéaire

### 11.3.1 Ordinateur SIMD

Nous reprenons l'exemple présenté dans [Ker89, pages 50–51] qui se rapproche des exemples de [ACF92] mais amène la superlinéarité pour une raison totalement différente.

Nous allons utiliser le fait que la plupart des machines SIMD possédant un processeur scalaire associé peuvent exécuter certaines parties séquentielles en même temps qu'elles exécutent des instructions parallèles [KNS91] pour amener une efficacité par processeur supérieure à 1, tel que cela a été mesuré réellement dans le cas de la machine PASM [FCS88].

Supposons que l'on veuille affecter un vecteur<sup>8</sup> à un autre, exprimé en POMPC sur la figure 11.3<sup>9</sup>.

Sur une machine séquentielle, à base de MC88100 pour prendre l'exemple de POMP, cela s'exécute en 5 cycles (figure 11.4) tandis que sur une machine SIMD, toujours à base du même processeur, on l'exécutera en parallèle en 3 cycles (figure 11.5).

---

8. Evidemment, la taille du vecteur n'est pas connue à la compilation, sinon on pourrait dérouler la boucle et l'exemple ne serait plus valable [FLW87].

9. Afin de faire l'économie du contrôle de flot SIMD, on place l'affectation dans un `everywhere` par mesure de simplification.

---

```

BOUCLE:
    ld r2,r3[r5]           "#   sub r6,r6,1
    st r2,r4[r5]           "#   bcnd.n ne,r6,BOUCLE
    add r5,r5,1            "#   nop

```

---

FIG. 11.5 - Programme de copie de vecteur sur une machine SIMD.

---

nop	"#		"#	st r5,r0,REG_BROADCAST
nop	"#	sub r6,r6,16	"#	add r5,r5,1
BOUCLE:				
ld r2,r3,0:BR	"#	bcnd.n ne,r6,BOUCLE	"#	st r5,r0,REG_BROADCAST
st r2,r4,0:BR	"#	sub r6,r6,16	"#	add r5,r5,1
ld r2,r3,0:BR	"#	nop	"#	nop
st r2,r4,0:BR	"#	nop	"#	nop

---

FIG. 11.6 - Exemple sur une machine 1-SIMD-2-VLIW.

Pour une machine à  $256 + 1$  processeurs, notons que POMP a la caractéristique très intéressante en ce qui nous concerne d'avoir des processeurs élémentaires *exactement* identiques au processeur scalaire, on a donc une accélération de  $257 \times 5/3 \approx 428$ , soit une linéarité de  $5/3$ .

Le modèle précédent donne une linéarité, sur cet exemple ( $N = 257$ ,  $n = 5$ ,  $k = 2$ ), de presque  $5/3$  et donc reflète bien la réalité. ce cas.

La superlinéarité provient du fait que la machine SIMD possédant un bus de diffusion peut dans ce cas factoriser sur le processeur scalaire la tâche de contrôle de flot, tâche qui devrait être répétée lors de chaque itération sur une machine séquentielle.

### 11.3.2 Ordinateur VLIW-SIMD

On peut reprendre le petit exemple précédent et l'adapter à une machine SIMD où le contrôleur est de type VLIW, voire parallèle, dans un sens plus général en reprenant une idée de [Kuc76]. Dans ce cas on peut exécuter en parallèle comme indiqué sur la figure 11.6<sup>10</sup> où l'instruction de gauche est exécutée sur les processeurs parallèles et les deux instructions de droite sur le processeur scalaire VLIW. Le programme s'exécute donc asymptotiquement avec une accélération de 645 pour 258 processeurs, soit une superlinéarité de 2,5. Le modèle nous donne une superlinéarité maximale de 2,98 ( $k = 4$  et  $n = 6$ )<sup>11</sup>. On constate que la géométrie du problème a changé: le rajout d'une tâche sur le processeur scalaire pour la diffusion de l'indice a permis de dégénérer une tâche parallèle en tâche séquentielle et augmenter l'efficacité de l'exécution. Evidemment ce genre de réécriture semble très difficile à représenter par un modèle général.

Suivant la nature du problème, on peut avoir des  $k$  plus ou moins grands. En général,  $k = 1$  ou  $k = 2$  pour des  $n = 2$  ou  $n = 3$  sont atteignables assez facilement lorsque le séquenceur fait le contrôle de flot, ainsi que la gestion des adresses globales et des index globaux. On a donc l'usage de ce parallélisme même sans compilateur VLIW sophistiqué, même s'il est vrai que pour atteindre des  $k$  élevés il faudrait utiliser de tels compilateurs.

En ce sens un couplage d'une machine SIMD ou SPMD avec un processeur  $k$ -VLIW (ou plus généralement de manière équivalente un processeur séquentiel  $k$  fois plus rapide), que les processeurs élémentaires de machine SIMD ou SPMD est intéressant, car il y

10. La notation <> permet d'exprimer les diffusions scalaires entre le processeur scalaire et les PES comme on le verra en §??.

11. Les différences sont dues au fait qu'une tâche parallèle n'a pas souvent le même nombre d'instruction qu'une tâche séquentielle qui fait la même chose.

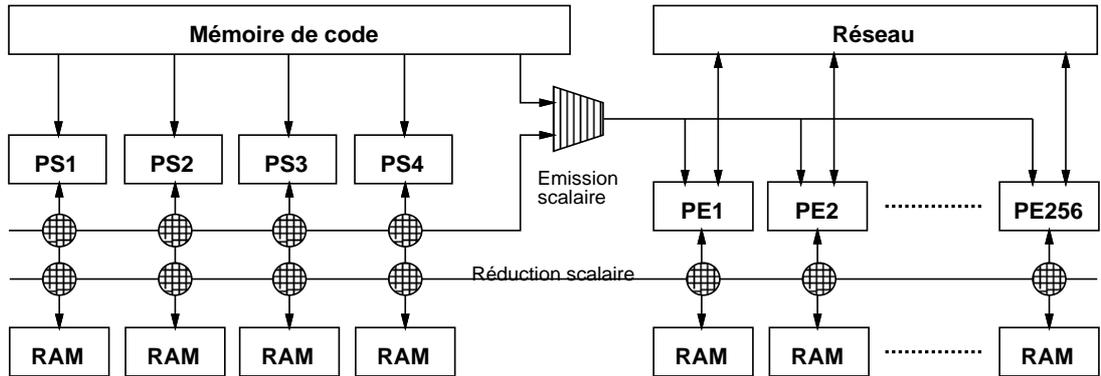


FIG. 11.7 - Architecture 4-VLIW-256-SIMD.

a un effet démultiplication de la puissance du processeur scalaire par les processeurs parallèles. Un exemple d'architecture VLIW-SIMD est montré sur la figure 11.7. En plus, cela peut avoir d'autres avantages dans la mesure où par exemple le processeur scalaire a ses propres registres et ne gaspille pas cette précieuse ressource sur les processeurs parallèles, qu'il a sa propre mémoire et donc que la bande passante mémoire des PEs n'est pas entamée, etc. Autant d'avantages pour un couplage SIMD à ce niveau qui jouent sur les constantes d'efficacité.

Quand POMP a été commencée, il n'y avait pas de processeur VLIW commercial ou de processeur « standard » très rapide. Comme un microprocesseur est plus simple à mettre en œuvre qu'un processeur VLIW en tranche, cela a fait basculer notre choix en ce qui concerne POMP à l'époque [Ker89, pages 24–28].

Après avoir présenté notre petit exemple et quelques compilations sur certaines machines, qu'en est-il de la méthode de simulation présentée dans [FLW87]? Ils font la supposition que le nombre de processeurs est connu déjà à la compilation et non à l'exécution seulement, ce qui est une hypothèse forte allant contre la notion de portabilité des programmes.

Dans ce cas, une itération parallèle est déroulée sur la machine séquentielle autant de fois qu'il y a de processeurs dans la machine parallèle, à savoir  $N$ . Le code de l'exemple 11.4 prend  $3N + 2$  cycles par itération équivalente parallèle à s'exécuter et le gain n'est plus que  $1 - \frac{1}{3(N+1)}$ , c'est à dire que l'exécution n'est plus superlinéaire.

Par contre la comparaison avec l'exécution présentée sur la figure 11.6 est plus intéressante puisque la superlinéarité est tout de même de  $\frac{3}{2} - \frac{1}{2(N+1)}$  malgré l'hypothèse prise de [FLW87].

## 11.4 Conclusion

Nous avons présenté un exemple de superlinéarité exploitant une des caractéristiques des machines SIMD, à savoir qu'elles possèdent un processeur scalaire qui est souvent capable de travailler en même temps que les processeurs parallèles.

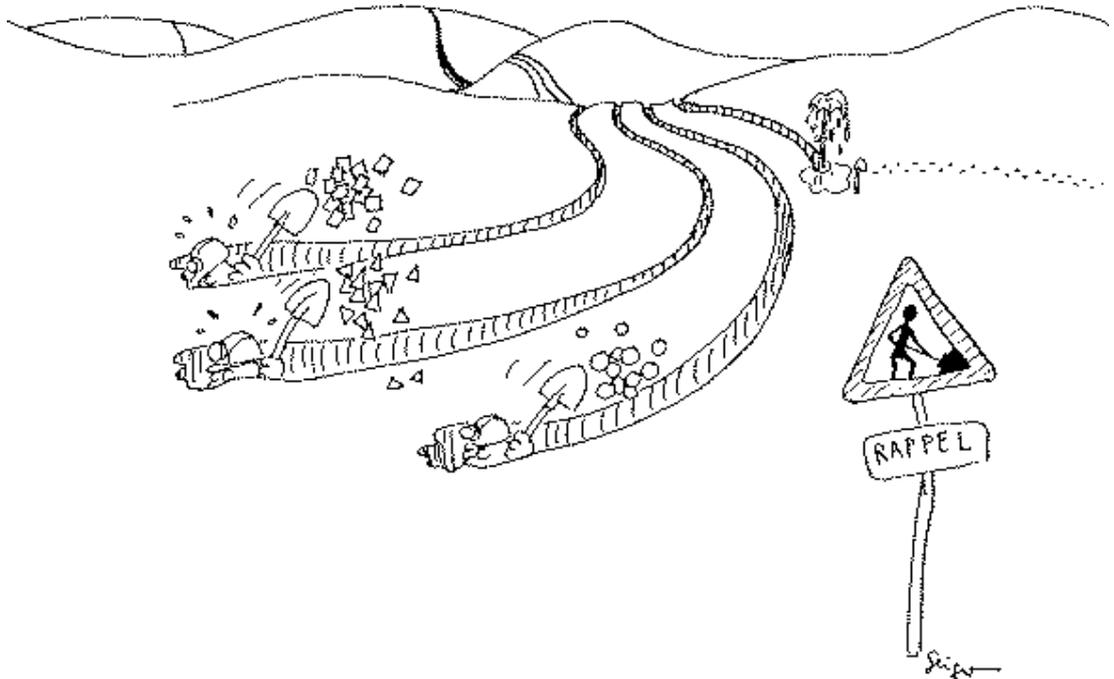
Cette caractéristique a pour effet troublant de permettre une *factorisation* de certaines tâches sur le processeur scalaire qui devraient être répétées autant de fois qu'il y a de PEs si on voulait exécuter le programme sur le seul processeur scalaire. Les

tâches se candidates sont par exemple le contrôle de flot scalaire et les calculs d'adresse globale. Un autre exemple concerne la machine PASM [FCS88].

La conclusion intéressante d'un point de vue philosophique est qu'en rajoutant un processeur de contrôle à des processeurs parallèles de type PRAM la loi d'AMDAHL n'est plus valable, mais qu'une fois qu'on a rajouté ce processeurs scalaire et qu'on en rajoute d'autres, cela revient à augmenter la puissance du métaproscesseur scalaire et la loi d'AMDAHL peut s'appliquer puisqu'on part d'une situation où il y a bien séparation entre partie séquentielle et partie scalaire.

# Chapitre 12

## Vers une machine SPMD ?



Entrées-sorties : on a intérêt à autoriser les les E/S à ne pas rester contraintes par les limites des partitions, pour des raisons de débit, contrairement à la CM-5 où les seuls noeuds d'E/S locaux se congestionnent.

 NOTRE concevoir la machine POMP nous sommes partis d'un certain nombre d'hypothèses au départ, à savoir que nous voulions concevoir une petite machine ne chauffant pas trop mais qui fut assez puissante pour accélérer les opérations de synthèse d'image, le tout conçu avec une économie substantielle de moyens de développement et donc avec pragmatisme, qui nous ont amenés à créer la machine et l'environnement de programmation décrits dans les chapitres précédents. Une des conclusions intermédiaires de notre étude est que finalement, une machine de synthèse d'image programmable n'est finalement qu'un supercalculateur possédant une bonne sortie vidéo.

Une question intéressante que nous nous sommes posés est de savoir à quoi ressemblerait le successeur de POMP si on supprimait les contraintes de place en conservant celles de la dissipation thermique raisonnable et tout de même une bonne densité en MFLOPS/dm<sup>3</sup>, et qu'on concevait la machine dès le départ comme un supercalculateur plus performant de 2 ordres de grandeur mais économique et pragmatique, dans un environnement universitaire ou bien industriel, le tout avec les contraintes technologiques actuelles, à la lumière de notre expérience acquise sur POMP.

On a vu que le modèle de programmation de type SPMD est en plein essor du fait de son principe conceptuel facilement accessible [DGNP88, Ste90, JC91, HQ91] et on peut se demander si une machine SPMD, rassemblant des avantages du SIMD (simplicité de programmation, synchronisation implicite) et du MIMD (liberté de programmation), ne serait pas une bonne candidate pour la prochaine génération de machine.

Ce chapitre va exposer deux approches d'une machine SPMD, la première, celle du « pauvre », l'universitaire pour lequel la conception d'un circuit intégré d'un million de transistors est encore beaucoup de travail<sup>1</sup> et assez utopique mais qui veut montrer que son concept est intéressant, la seconde, celle du « riche », l'industriel qui veut concevoir une machine économiquement compétitive, donc performante, assez facilement réalisable et à un coût minimal.

Il ne s'agit certes pas de faire une autre thèse sur ce sujet encore plus vaste mais plutôt de donner quelques idées et réponses extraites des chapitres précédents, ainsi que de faire des comparaisons avec le projet POMP actuel.

## 12.1 Modèles de programmation et langages

### 12.1.1 POMPC

Étant donné l'héritage logiciel performant du projet POMP, proposer un modèle de programmation et un langage dérivé de POMPC est intéressant car il est facile à adapter à de nombreuses architectures, y compris celle-ci. On pourra dès le début avoir au moins un environnement de programmation et de mise au point du système avant que la machine fonctionne.

Le langage reste un langage orienté collection et hérite des mécanismes de gestion de l'activité nécessaire si on veut pouvoir entrelacer différentes phases de calculs appartenant à différentes collection. En particulier le mécanisme de factorisation à base

---

1. Car il travaille peut-être tout seul dans son laboratoire avec des outils vétustes... Mais il pourra pour se consoler aller voir à ce sujet la très bonne pièce de théâtre « Les palmes de M. SCHULTZ ».

de compteurs présenté dans la section 7.1.3 reste valable.

Par rapport à la version originale développée pour POMP on aura tout intérêt à modifier le langage pour qu'un minimum de MIMD puisse être pris en compte dans le langage afin de pouvoir exprimer et exécuter certains algorithmes typiquement MIMD. Cela peut être fait soit par l'ajout de nouveaux constructeurs, la solution la plus simple, ou par l'exploitation de techniques de compilation avancées [McK92], à commencer par des privatisations, des expansions scalaires pour les boucles parallèles ou des remplacements scalaires pour les mises en registre, la solution la plus compliquée mais la plus transparente à l'utilisateur.

Un premier pas consiste déjà, comme suggéré en § 4.3, en la simple séparation de l'espace de définition de l'espace d'itération, par exemple à partir d'un constructeur de type `forall` [?], peut fournir le concept de base.

### 12.1.2 HPF

Pour les gens qui préfèrent utiliser FORTRAN, proposer HPF (§ 4.1.1.8) est une solution de choix car, étant très probablement amené à devenir un standard *de facto*, il permettra de développer des applications pouvant fonctionner sur plusieurs machines différentes.

L'intérêt du langage est qu'on peut faire assez rapidement un compilateur simple qui fonctionne en n'exploitant que les directives et les constructions parallèles indiquées par le programmeur puis à compliquer de plus en plus le compilateur grâce à des méthodes d'analyse de dépendances et de parallélisation automatique pour qu'il soit capable d'extraire plus de parallélisme, relaxer certaines contraintes fournies par le programmeur, optimiser le placement des données afin de limiter les communications, etc.

En plus, on pourra très probablement trouver de la compétence en milieu universitaire ainsi qu'en milieu industriel, ce qui donne une assurance suffisante en faveur du suivi du langage et des investissements logiciels.

## 12.2 Architecture globale de la machine

Une architecture de type SPMD (§ 2.2.6) nous semble adéquate car elle permet d'exécuter aussi bien des programmes suivant un modèle SIMD ou MIMD. Une machine SPMD parfaite peut être définie ainsi : chaque PE peut exécuter un flot d'instructions locales, se synchroniser avec qui il veut, voire avec un sous-ensemble quelconque de processeurs indépendamment du reste de la machine. En outre, chaque processeur ne voit qu'une seule mémoire globale qu'il peut accéder très rapidement.

Malheureusement, le monde n'étant pas parfait, une telle mémoire globale est difficile à faire (§ 2.4) et on préférera par conséquent un couplage faible. Dans ce cas, les processeurs peuvent communiquer entre eux en un temps qui n'est pas non plus très court mais qui peut être diminué si le programmeur ou le compilateur est capable d'optimiser le placement des données, ce qui est possible sur beaucoup d'applications.

Il en est de même pour les synchronisations : elles prennent un temps non négligeable et il est souvent difficile d'avoir plus d'un point de synchronisation dans toute la machine lorsqu'on a beaucoup de processeurs, ou tout au moins sur des sous-ensembles

de la machine. Néanmoins on peut souvent s'arranger pour limiter les points de synchronisation dans un programme et placer les données et les processus en conséquence.

On choisit donc de voir notre architecture comme étant MIMD avec des mécanismes de synchronisation plutôt que capable d'exécuter des programmes réellement aussi bien en SIMD qu'en MIMD, c'est-à-dire qu'on privilégie le modèle SPMD de programmation plutôt que le modèle réel d'exécution.

### 12.2.1 Pour ou contre le partitionnement ?

Un point à éclaircir avant toute continuation est de savoir s'il est important ou non de pouvoir diviser la machine en sous-machines indépendantes pouvant servir à plusieurs utilisateurs. On admet donc implicitement que la machine est de toute manière multi-utilisateurs.

Les arguments en défaveur du partitionnement de la machine sont principalement :

- la répartition de la charge est mieux gérée par un système d'exploitation qui ne peut allouer qu'une machine complète : la répartition de la charge est implicite. En ce qui concerne le partitionnement en soi, l'allocation des sous-machines par le système d'exploitation est assez complexe et dans certains cas des sous-machines ne peuvent pas être allouées pour des raisons géométriques de communication ou tout simplement de taille de sous-machine disponible, comme dans le cas de la machine PASM [TS85] : il se peut qu'un utilisateur soit obligé d'attendre qu'une sous-machine de taille adéquate soit libérée ;
- il est intéressant d'avoir un nombre de tâches minimales afin d'avoir un recouvrement des temps d'attentes dans les entrées-sorties et les paginations sur disque<sup>2</sup> ;
- enfin le partitionnement en lui-même est compliqué à faire puisqu'il faut être capable de subdiviser les systèmes d'émission scalaire, de synchronisation, le réseau, etc. pour faire des sous-machines les plus indépendantes possibles.

Les arguments positifs sont :

- lorsqu'on n'a pas d'environnement multitâche c'est la seule manière de partager la machine ;
- lorsqu'on a beaucoup de programmes avec un plus faible degré de parallélisme, les processeurs sont mieux utilisés ;
- la politique d'utilisation actuelle des machines partitionnables semble être d'avoir une grosse partition pour faire tourner les programmes en production et d'avoir une ou quelques petites partitions pour la mise au point des programmes en cours de développement ;
- en cas de panne d'un ou de plusieurs processeurs, cela permet un système de tolérance aux pannes assez simple : on élimine un ou plusieurs hyperplans de PES contenant ce PE et on les impose comme limite de partition. Cette méthode peut bien-sûr être étendue dans le cas où plusieurs processeurs sont en panne ;

---

2. Cet argument peut être aussi valable dans le cas d'une machine partitionnable à condition qu'il y ait assez de processus.

- enfin et surtout, l'argument commercial va très fortement en sa faveur.

Malgré la complexité accrue, il semble tout de même utile de proposer une solution partitionnable.

### 12.2.2 Un processeur scalaire est-il nécessaire ?

Le processeur scalaire apparaît comme nécessaire pour plusieurs raisons mais il existe aussi des arguments allant dans l'autre sens, c'est que nous allons montrer.

Le chapitre 11 a montré qu'on pouvait avoir une efficacité accrue en faisant réaliser certaines tâches séquentielles sur le processeur séquentiel d'une machine SIMD/SPMD par rapport à un ordinateur MIMD où il faut exécuter chaque tâche séquentielle sur chaque processeur et qu'il pouvait même en résulter une superlinéarité par rapport à un ordinateur MIMD. Mais disons que le principal effet macroscopique est le recouvrement du temps d'exécution scalaire et parallèle, lorsque cela est possible. Si cela n'est pas possible, les 2 solutions sont équivalentes puisque dans un cas les PEs ne font rien et attendent le processeur scalaire et dans l'autre cas tous les PEs peuvent bien exécuter chacun le code scalaire puisque de toute manière il n'aurait rien d'autre de mieux à faire qu'attendre qu'un des PEs l'exécute et envoie le résultat aux autres.

Les tâches telles que la gestion globale des entrées-sorties ou la gestion du système d'exploitation et le contrôle de la machine se font plus naturellement de manière centralisée. Il participe aussi à la notion d'opérateur global de la machine qui est indispensable pour permettre une synchronisation rapide de la machine (§ 12.6). En particulier, il accélère les mécanismes d'émission d'une variable scalaire vers une variable vectorielle. Un autre cas où un processeur scalaire est très utile est pour la factorisation d'une grosse base de données : elle n'a à être copiée que dans la mémoire du processeur scalaire qui peut être plus grande que celle des PEs.

On peut désormais, avec la vulgarisation des stations de travail multiprocesseur [Cyp91], réserver à un des processeurs le rôle de processeur scalaire et aux autres le rôle d'hôte : gestion du système, des entrées-sorties, autres travaux classiques UNIX, etc. Cela a plusieurs avantages :

- le processeur scalaire devient moins sensible aux interruptions et changement de tâches du système : il est pleinement consacré à la gestion des processeurs parallèles et gaspille le moins possible de leur puissance ;
- on peut imaginer un processeur scalaire composé de plusieurs processeurs de la station de travail pour augmenter ses performances, donc celle de la machine globale (voir la partie 11.3.2), si la partie scalaire est en fait un petit peu parallèle, pas suffisamment pour être exécutée sur les PEs ;
- bus très rapide dans la machine pour permettre une bonne intégration entre l'hôte et le processeur scalaire ainsi qu'un temps de latence faible pour les appels système nécessitant des transferts de données importants ;
- excellente utilisation des programmes système existant et développements logiciels faibles à ce niveau, en particulier on peut utiliser un système de changement de tâche au niveau des PEs commandé par un système détectant un changement de tâche au niveau du processus de l'hôte commandant l'ordinateur parallèle ;

– la généralisation d'UNIX SVR4 permet une telle utilisation du parallélisme.

Dans ce cas, on peut éviter d'avoir à refaire un processeur scalaire. Malheureusement, il vaut mieux avoir un système d'exploitation asymétrique — donc paradoxalement en retard sur l'état de l'art — au niveau de la station de travail multiprocesseur, car sinon les appels systèmes engendrés sur le processeur scalaire risquent de s'y exécuter aussi. Or les progrès des systèmes d'exploitations de ce genre vont vers la symétrie... Mais il est fort probable qu'on puisse ajouter une brisure de symétrie dans la configuration du système.

Néanmoins un processeur scalaire peut être une gêne si on veut pouvoir partitionner la machine en plusieurs sous-machines. Dans ce cas la notion de processeur scalaire disparaît, à moins d'en mettre un par sous-machine [SSDK84, SSKD87, Thi91], voire par PE si on veut accepter un partitionnement quelconque non limité à des sous-machines prédéfinies mais cette solution n'est pas raisonnable.

Si on n'a pas de processeurs scalaire, il faut bien entendu confier au compilateur le soin de faire respecter à la machine le modèle de programmation à parallélisme de données le cas échéant [HQ91]. En particulier des opérations telles que les entrées-sorties ne sont à exécuter qu'une seule fois sur toute la machine, par exemple sur un des processeurs parallèles. Un autre cas où c'est intéressant est pour le partage d'une grosse base de données scalaire qu'on ne veut pas dupliquer chez tous les PEs et qu'on préfère garder sur le processeur scalaire lorsqu'on ne veut pas utiliser un mécanisme de mémoire virtuellement partagée.

De toute manière, même si on n'a pas de processeur scalaire, il est utile d'avoir un processeur s'occupant du système de la machine au démarrage, de son partitionnement, de son test, etc. Cette tâche a tout intérêt à être soumise à une station de travail afin d'éviter le développement de cette machine de contrôle.

Dans la suite, on va considérer qu'on ne réalise pas de processeur scalaire, laissant au logiciel le soin de simuler son existence, soit sur un PE par l'intermédiaire de changements de tâches astucieux pour séparer les domaines d'adressage séquentiel et parallèle par exemple, soit sur un processeur d'entrées-sorties pouvant lui-même être en fait une station de travail si on veut profiter de l'environnement logiciel déjà existant.

## 12.3 Nœud de la machine

Dans ce qui suit on présentera deux versions de machine : une version universitaire où le circuit de communication est en fait un TMS320C40 qui possède de quoi construire un réseau et être relié à un bus global, d'autre part une version industrielle bâtie autour d'un circuit spécialisé à concevoir qui intègre le réseau et la glue de la machine. On développera surtout la version industrielle sans bus global, l'autre version se déduisant par émulation logicielle du matériel absent et l'utilisation du bus global.

### 12.3.1 Les processeurs élémentaires

Les arguments développés dans les sections 5.1.1, 5.1.2 et 5.1.4 restent pertinents et nous poussent à choisir les processeurs parmi les plus puissants du marché à technologie « froide » (§ 5.2.2), à nuancer en fonction de l'infrastructure que l'on veut mettre autour, en particulier du réseau et du coût de l'interface mémoire.

Outre la puissance en MFLOPS théoriques, il faut bien entendu considérer la bande passante des bus de données au niveau registres, cache primaire, cache secondaire s'il existe et enfin mémoire globale du nœud.

Afin de permettre une bonne cohérence entre le cache, la mémoire et le circuit de communication il est primordial que le(s) cache(s) du PE soi(en)t équipé(s) d'un système de assurant leur cohérence.

Le processeur doit aussi être équipé d'une MMU afin de faciliter la compilation et l'exécution de programmes devant être exécutés en multitâche bien-sûr, assurer la protection mémoire entre programmes, c'est-à-dire ce qu'on attend d'une MMU sur n'importe quelle machine mais aussi dans notre cas de permettre une ouverture vers la mémoire virtuellement partagée.

Enfin, comme on en arrive à parler de mémoire globale, il faut parler de son adressage global : 32 bits ne permettant d'accéder que 4 Go, un passage progressif vers un adressage 64 bits s'impose, d'une part pour gérer la mémoire virtuellement globale mais physique, d'autre part pour gérer la mémoire virtuellement globale et virtuelle — virtuelle dans le sens où elle inclut la mémoire de pagination encore plus grande qui est sur les disques de la machine. Cela pousse donc clairement à avoir un parallélisme à gros grain.

### 12.3.2 La mémoire du nœud

Comme discuté précédemment dans cette thèse, un des problèmes cruciaux est le nombre de composants supplémentaires que possède un nœud asynchrone par rapport à un nœud synchrone, principalement dû au fait qu'il faut un système de synchronisation et que le processeur étant le plus rapide possible<sup>3</sup>, il faut lui associer une mémoire conséquente et rapide, plus complexe. Celle-ci doit répondre à deux critères :

- 1° être très rapide pour ne pas pénaliser le processeur ;
- 2° avoir une capacité importante pour permettre une exploitation réelle de la puissance du nœud [GREC91].

Du coup, ce besoin de capacité importante fait choisir des mémoires de type dynamique, de capacité quadruple mais dont la rapidité est divisée par un rapport 4 ou 2 selon le mode d'utilisation<sup>4</sup>.

Mais il est délicat de dire *a priori* quelle solution sera la meilleure entre un nœud avec mémoire dynamique ou mémoire statique, en ce qui concerne le rapport MFLOPS/dm<sup>3</sup>, intimement lié au rapport MFLOPS/nombre de composants. Les sections suivantes discutent de quelques alternatives possibles.

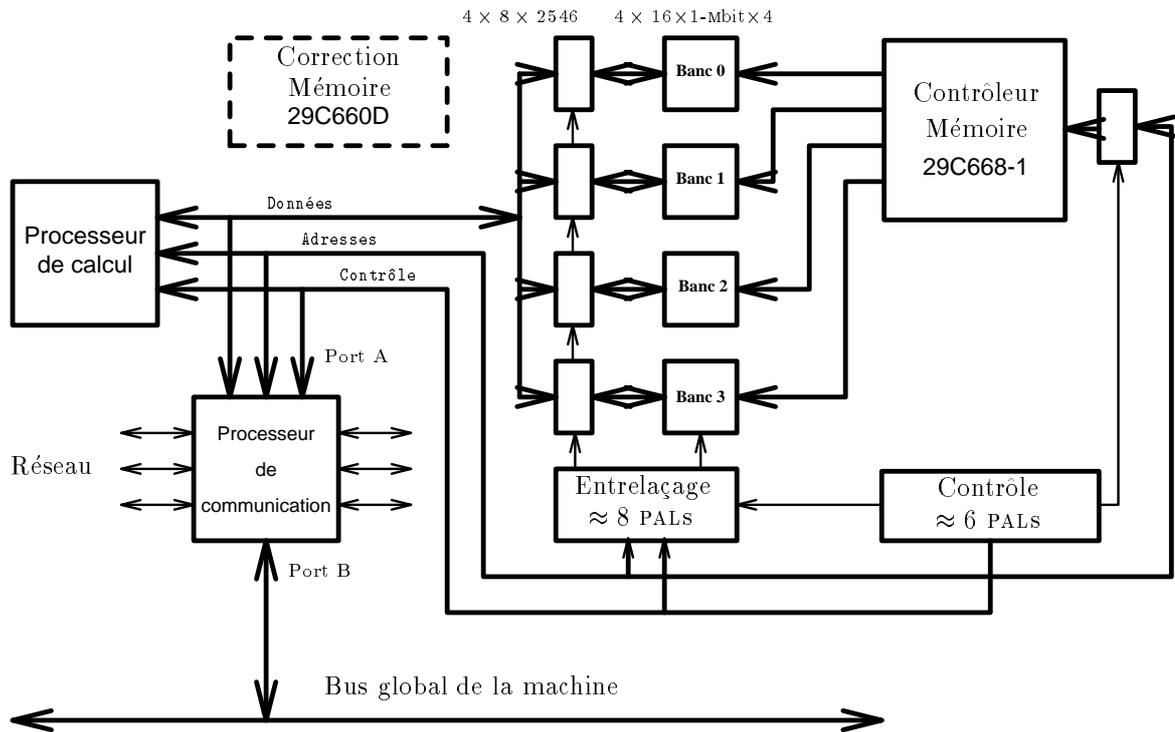
#### 12.3.2.1 Nœud SPMD à base de mémoire dynamique

Qui dit mémoire dynamique dit qu'il faut rajouter du matériel pour la « rafraîchir » à un rythme de l'ordre d'une fois toutes les 1 ou 10 ms car elle n'est pas permanente.

---

3. Dans la limite du raisonnable...

4. Typiquement la comparaison entre une mémoire statique 1M×1 bit de 25 ns de temps de cycle et une mémoire dynamique 4M×1 bit ayant un temps de cycle de 110 ns et 40 ns en mode d'accès dans une même page [Mic91].

FIG. 12.1 - *Synoptique du nœud SPMD avec mémoire dynamique.*

Même s'il existe des circuits intégrés spécialisés dans cette tâche, cela complique la réalisation du nœud.

En plus, comme la mémoire dynamique est lente, il faut utiliser plusieurs astuces classiques pour faire croire au processeur qu'elle est rapide. Le terme « rapide » est d'ailleurs ambigu puisqu'on entend par là aussi bien le temps de latence, c'est à dire le temps que la mémoire met à fournir le premier mot demandé, que le débit, qui est le nombre de données transférées par unités de temps. Si on ne peut pas diminuer la latence, on peut au moins augmenter le débit par deux méthodes [Adv90b] (§ 2.4) :

- lorsqu'on veut une mémoire rapide, on peut utiliser le fait que souvent les accès à la mémoire ont une certaine localité et profiter du fait qu'accéder à une mémoire dans la même « page » est plus rapide qu'accéder dans une page différente. Cela revient à exploiter un principe semblable aux mémoires cache au niveau de la mémoire centrale ;
- pour aller encore plus vite on entrelace plusieurs bancs mémoires entre eux. On accède successivement à chaque banc mémoire, faisant ainsi croire au processeur que le débit de sa mémoire est multiplié par nombre de bancs. Bien entendu, cela ne fonctionne que lorsque les accès se font séquentiellement à des mots contenus dans des bancs différents.

On constate que ces deux méthodes font des suppositions quant aux types d'accès effectués. Souvent ils sont de type remplissage de cache ou accès à des vecteurs consécutifs et on peut espérer une bonne efficacité du système. Mais si les accès à la mémoire

sont plus aléatoires, le processeur ressentira le temps de cycle de la partie mémoire, soit de l'ordre de 200 ns.

Cela rend beaucoup plus complexe la conception et la réalisation de la partie mémoire du nœud et cela conduit à rajouter beaucoup de circuits intégrés. Ceci dit, on peut profiter du grand nombre de composants supplémentaires pour noyer dans la masse un système de correction d'erreurs associé à la mémoire dynamique. Un tel système est de toute manière indispensable pour assurer le fonctionnement d'un ordinateur parallèle à mémoire dynamique car le nombre de circuits de mémoire étant très élevé, la fréquence des erreurs de mémoire augmente en conséquence.

Pour réaliser les corrections d'erreurs de 1 bit par mot mémoire, il existe des circuits basés sur l'utilisation d'un code cyclique redondant, mémorisé en plus des données. On peut mettre soit un circuit de ce type par banc, dans ce cas ces circuits n'ont pas à être très rapides, soit utiliser un circuit sur tout le nœud qui corrige les données qui passent sur le bus de données du processeur. Mais dans ce cas ce circuit doit être très rapide<sup>5</sup>.

Comme la vitesse des bus est importante, on a plus intérêt à mettre le circuit de vérification en parallèle, quitte à ralentir le processeur en cas d'erreur, qu'à faire corriger en continu les données, même lorsqu'elles sont justes, car si cela simplifie la conception, cela augmente la latence.

Enfin, le rafraîchissement et le contrôle de la mémoire sont faits par un ou plusieurs contrôleurs spécialisés suivant sa capacité à gérer la mémoire par banc. Il existe un contrôleur capable de gérer le mode page et la mémoire entrelacée mais malheureusement il n'est pas assez rapide et il faut utiliser 2 de ces circuits.

Le synoptique d'un tel nœud dans sa version universitaire est indiqué sur la figure 12.1, où la mémoire cache associée au processeur n'est pas représentée.

### 12.3.2.2 Nœud SPMD à base de mémoire statique

L'intérêt de la mémoire statique est multiple et c'est d'ailleurs pourquoi nous l'avons choisie pour POMP :

- elle n'a pas besoin d'être rafraîchie ;
- son contrôle est plus simple (pas de circuit supplémentaire) ;
- elle est plus sûre que la mémoire dynamique ;
- son temps de cycle est plus rapide (d'un facteur 4).

Malheureusement, sa capacité est aussi divisée par 4 pour une même génération et son prix est plus élevé.

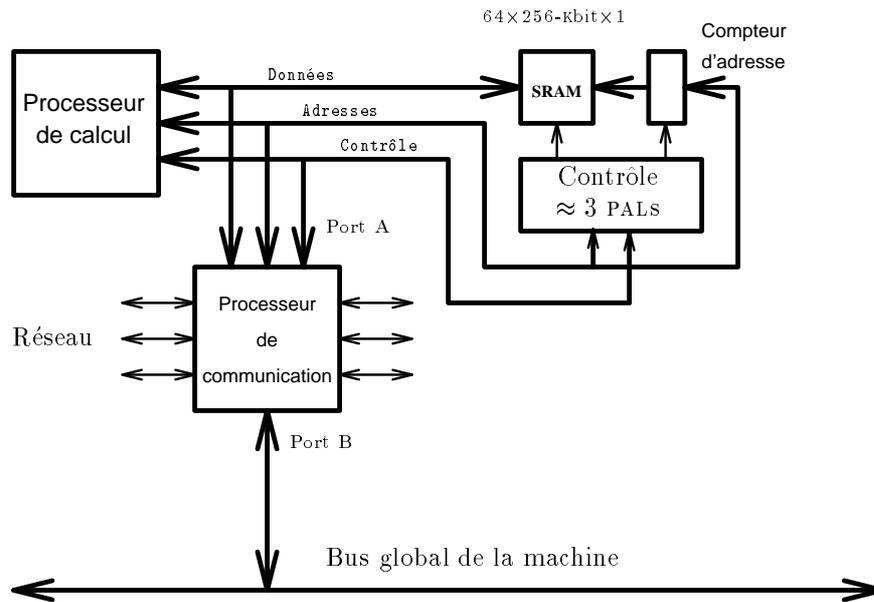
L'utilisation efficace des processeurs les plus rapides va nécessiter l'emploi des mémoires statiques les plus rapides, à moins que pour des raisons de coût on n'utilise une méthode comparable à l'entrelaçage de plusieurs bancs de mémoire comme on l'a vu pour la mémoire dynamique.

Il est bien dommage que les mémoires statiques à mode page rapide n'existent pas car sinon elles suffiraient à remplir la tâche en ayant un coût et une simplicité d'utilisation acceptable<sup>6</sup>.

---

5. Et n'existe pas encore...

6. Mais il est probable que cela ait déjà été breveté !

FIG. 12.2 - *Synoptique du nœud SPMD avec mémoire statique.*

### Mémoire statique non entrelacée

Cela revient à considérer la mémoire du processeur comme étant entièrement de la mémoire cache, comme on a fait sur la machine POMP actuelle.

Le mode par à coup du bus est géré grâce au compteur sur le bus d'adresse de la mémoire (figure 12.2). Cela ne coûte pas plus cher que dans le cas de la mémoire dynamique car de toute manière il fallait un verrou pour s'interfacier avec le bus du processeur (figure 12.1).

Dans l'état actuel de la technologie, les mémoires nécessaires sont d'assez petite capacité (1 Mbit) et le coût du nœud de la machine et le nombre de boîtiers mémoire sont assez importants dès qu'on met beaucoup de mémoire. Néanmoins, comme on économise toute la logique d'entrelacement, la densité n'est que moitié de celle de la mémoire dynamique.

### Mémoire statique à bancs entrelacés

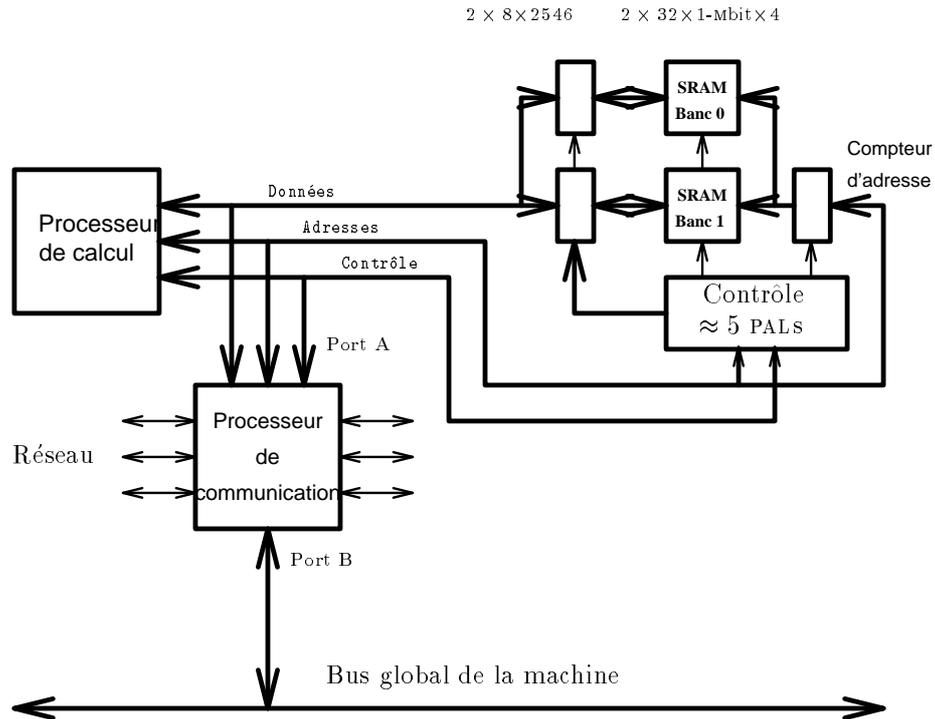
Un entrelaçage d'un facteur 2 est suffisant avec des mémoires statiques moyennement rapides. Si on économise la logique de rafraîchissement, il faut toujours le mécanisme d'entrelacement. Mais ce dernier est assez simple à faire lorsqu'on n'a que 2 bancs à contrôler.

#### 12.3.2.3 Conclusion

Le tableau 12.1 donne une comparaison entre les solutions proposées précédemment avec les performances et le coût de réalisation avec comme hypothèse que le débit crête est de 400 Mo/s dans tous les cas.

Ce tableau est donné en regardant les données actuelles technologiques :

- débit processeur-mémoire de 400 Mo/s sur un bus à 64 bits ;

FIG. 12.3 - *Synoptique du nœud SPMD avec mémoire statique entrelacée.*TAB. 12.1 - *Comparaison entre les différents types de module mémoire.*

Type de mémoire	Latence	Débit aléatoire	Capacité	Nombre de boîtiers mémoire	Nombre de circuits
dynamique entrelacée	200 ns	40 Mo/s	32 Mo	72	120
dynamique entrelacée	200 ns	40 Mo/s	64 Mo	144	192
statique rapide	20 ns	400 Mo/s	2 Mo	64	70
statique entrelacée	40 ns	200 Mo/s	8 Mo	64	88
statique entrelacée	40 ns	200 Mo/s	16 Mo	128	152

- mémoire dynamique de 4 Mbit avec un temps de cycle de 110 ns (temps d'accès de 60 ns) ;
- mémoire statique rapide de 256 Kbit avec un temps de cycle de 12 ns ;
- mémoire statique moyennement rapide de 1 Mbit avec un temps de cycle de 25 ns ;
- des tampons et des PALS avec un temps de traversée de 5 ns, de la logique de type (Bi-)CMOS ;

Les nombres de boîtiers sont donnés approximativement, les schémas n'ayant pas été réalisés.

La solution à base de mémoire statique trop rapide est à éliminer dans une optique industrielle vu son coût et sa capacité même si elle a de bonnes performances pour les accès aléatoires<sup>7</sup>, mais est peut-être à conserver pour un prototype universitaire. On peut donc hésiter entre les autres solutions :

- si on veut un processeur qui puisse être utilisé pleinement avec beaucoup d'indirections, on s'orientera vers une mémoire statique entrelacée de 8 ou 16 Mo selon la place disponible ;
- si on pense que le temps d'accès aléatoire n'est pas critique pour les applications visées, en considérant par exemple le temps d'accès du réseau, on choisira la solution à base de mémoire dynamique, plus complexe mais moins chère à produire.

À titre indicatif, une solution haut de gamme à partir de processeur ALPHA nécessite un bus de 128 bit à 100 MHz, ce qui nous fait pencher vers au moins la solution de la 2<sup>ème</sup> ligne du tableau 12.1 avec peut-être encore le double de mémoire, sachant qu'on fait l'économie du système de correction mémoire, déjà dans le processeur [Dig92a, Dig92b].

Il est probable qu'il faille trouver un compromis avec une interface mémoire plus lente, moins entrelacée, mais aussi moins chère. Jusqu'où peut-on ralentir la mémoire ? Cela dépend vraiment de l'application. Si le cache fonctionne parfaitement, la vitesse de la mémoire sera peut importante. Par contre, si on travaille sur beaucoup de données, de surcroît avec une répartition aléatoire, le débit mémoire est le facteur limitant (comme exposé au début de § 5.1.1.4 et dans [Dou89]). Avoir par exemple un bus mémoire de 128 bits à 20 MHz (mémoire dynamique standard en mode page) permet un débit sur des vecteurs longs de 320 Mo/s, soit dans le cas d'un produit scalaire en double précision 40 MFLOPS par PE et 13 MFLOPS pour une addition vectorielle<sup>8</sup>, ce qui est assez faible. L'entrelacement de la mémoire permet d'augmenter ces performances de manière proportionnelle, dans les limites du débit maximal, bien entendu.

## 12.4 Le système de gestion mémoire

Sur une machine classique, la mémoire physique est divisée en un certain nombre de pages qu'un programme utilisateur peut voir à travers un système de gestion de la mémoire (MMU) comme étant à un autre endroit dans son espace d'adressage virtuel. On distingue donc l'adressage physique, celui de la machine réelle, et l'adressage virtuel, seul accessible à l'utilisateur, qui est un reflet de la réalité que veut bien lui faire croire le système d'exploitation.

Un tel système permet de simuler par exemple plus de mémoire que disponible en stockant certaines pages virtuelles sur disques lorsqu'on n'y accède pas ou bien faire coexister plusieurs programmes pouvant appartenir à différents utilisateur de manière simultanée sans conflit d'adresse d'une part et surtout sans interaction incontrôlée et donc avec un système de protection.

---

7. Ces performances sont surestimées en fait car le mécanisme de gestion de cache du processeur risque de réduire le débit en mode aléatoire.

8. On suppose l'utilisation d'instruction de précharge de cache dans le cas d'un ALPHA par exemple.

Grossièrement on peut dire que plus la taille des pages est petite, plus fin sera le contrôle de la mémoire et mieux elle sera utilisée. Comme on est obligé d'avoir une table permettant de relier les adresses de pages logiques et physiques, lorsque les pages sont petites leur nombre est important et on ne peut les stocker toutes dans la MMU. Il y a donc généralement un système de mémoire cache capable de retenir les conversions les plus souvent utilisées et un mécanisme capable d'aller lire en mémoire superviseur les morceaux de table de conversion, rangées de manière arborescente, pour mettre à jour la mémoire cache [MOT88b].

On constate que la capacité mémoire nécessaire à la MMU se limite à celle de son cache mais implique un système sophistiqué capable de fouiller des tables arborescentes en mémoire.

Dans le cas d'un supercalculateur, le problème est tout autre. En effet, il s'agit généralement de manipuler des structures de données énormes sur lesquelles on fait bien souvent des accès séquentiels. Le fait d'utiliser des pages de petites tailles n'apporte pas grand chose au niveau de l'utilisation de la mémoire. Heureusement, les processeurs du commerce proposent souvent une MMU pouvant travailler avec différentes tailles de page.

Au niveau d'une machine parallèle à mémoire distribuée, la mémoire est utilisée généralement par 2 dispositifs concurrents : le processeur bien sûr et le circuit de communication. Dans le cas de POMP le débit des communications était suffisamment faible pour être géré par le processeur mais si on veut dépasser cette limite il faut rajouter un véritable coprocesseur de réseau. Ce coprocesseur doit accéder à la mémoire avec les mêmes mécanismes de protection que ceux du processeur et donc utiliser une MMU. On n'avait pas ce problème sur POMP, la machine étant monotâche.

Il existe un certain nombre de solutions possibles :

- on rajoute un autre processeur s'occupant de ces problèmes, comme dans le cas de la machine PARAGON. C'est une approche intéressante, même si elle n'offre pas les meilleures performances, ce qui nous la fait retenir pour une version universitaire de notre machine SPMD ;
- la solution la plus simple est d'utiliser la MMU du processeur mais cela ne peut fonctionner que pour de machines assez lentes (typiquement avec le S-Bus) car cela nécessite une interrogation à travers un bus et l'attente d'une réponse. En plus elle demande l'utilisation d'un processeur approprié ;
- la solution la plus performante et la plus intégrée consiste à implanter une vraie MMU dans le circuit de communication.

Une machine haut de gamme aura donc besoin d'une MMU dans son circuit de communication. Malheureusement pour nous, une MMU n'est pas simple à réaliser. Pour essayer de contourner la difficulté, nous allons présenter une version simplifiée adaptée à notre problème.

Si les pages de mémoire gérées par la MMU sont suffisamment grandes, leur nombre est faible et on peut imaginer que leur description puisse loger dans une petite mémoire en faisant ainsi l'économie du système de gestion de tables arborescentes. Malheureusement, l'espace d'adressage virtuel étant beaucoup plus grand que la mémoire physique, le nombre de pages reste encore énorme.

Chaque programme possède au moins 3 segments spécifiques si on reprend la philosophie UNIX : un segment de code, un segment de données et un segment de pile. Un programme nécessitera donc au minimum 3 pages différentes de mémoire. Si la capacité mémoire d'un nœud est  $M$  et que la taille des pages est  $P$ , le nombre de processus simultanés (y compris le système) est  $\frac{M}{3P}$ . De plus, dans le pire des cas, un processus gâche  $3(P - 1)$  octets de mémoire. Cela nous donne des idées afin de trouver un compromis entre le nombre, la taille des pages et la mémoire potentiellement gâchée à réduire.

Si on prend par exemple  $M = 256 \text{ Mo}$ ,  $P = 1 \text{ Mo}$  est une valeur raisonnable car elle permet de partager la mémoire physique en 256 pages seulement qui permet tout de même d'avoir jusqu'à 85 processus résidents en mémoire physique. Le problème est que c'est la mémoire virtuelle qu'on veut partager.

Mais quelle est l'utilité d'une MMU ? Elle sert :

- 1° à faire de la protection mémoire, à savoir à empêcher l'écriture de certaines pages physiques ou encore tout simplement l'accès à certaines pages physiques (non allouée par le système ou appartenant à un autre processus par exemple) ;
- 2° à interdire les accès en dehors de la mémoire physique après traduction de l'adresse virtuelle ;
- 3° à étendre la mémoire physique en l'émulant sur disque ou en la distribuant sur tous les processeurs (cas de la mémoire virtuellement partagée) ;
- 4° faciliter la gestion mémoire au niveau système (ramasse-miettes et démocratisation de l'adressage absolu pour l'utilisateur).

Le 1<sup>er</sup> point ne concerne que les pages physiques et donc ne demande qu'une information par page physique. Le 4<sup>ème</sup> point ne demande rien de particulier, étant une utilisation logicielle des autres points.

Par contre les 2<sup>ème</sup> et 3<sup>ème</sup> points nécessitent de connaître une information sur chaque page virtuelle, ce qui est plus difficile. Les conséquences des points 2 et 3 sont le déclenchement de fonctions systèmes assez lourdes :

- pour le 2<sup>ème</sup> point il s'agit de générer une exception de violation de segmentation éventuellement détournable par l'utilisateur pour le bon fonctionnement de son programme ou tout simplement arrêter le programme sur une erreur ;
- le point 3 nécessite de faire un accès aux entrées-sorties pour la pagination ou bien de lancer des communication afin de faire des transferts de pages entre processeurs.

Vue l'ampleur des tâches, une gestion de ce problème nécessitera de toute manière une aide logicielle<sup>9</sup> et on peut très bien gérer ces défauts de page de manière presque totalement logicielle.

---

9. Néanmoins certaines machines réalisent la mémoire virtuellement partagée de manière totalement logicielle, mais en considérant que la machine KSR a demandé le développement de 12 circuits intégrés différents pour réaliser un nœud, que le protocole impliqué nécessite des milliers d'états, on conçoit que cette approche soit un peu rebutante...

On a juste besoin de savoir très rapidement, donc de manière matérielle, si une page virtuelle est non traduisible, soit une information par page physique en raisonnant sur la négation. Si elle est traduisible on est dans le cas du point 1, si elle n'est pas traduisible une exception est déclenchée. Le noyau du système d'exploitation peut alors faire une recherche arborescente dans les tables de traduction pour savoir à quoi correspond l'adresse fautive. Si cette adresse ne correspond à rien on est dans le cas de la violation de segmentation, sinon on appelle la routine de pagination.

Le mécanisme à réaliser est donc un cache mémorisant la traduction de chaque page physique de la machine. Du fait de la structure habituelle des programmes en 3 segments, il vaut mieux avoir une mémoire complètement associative. En effet, souvent les segments de code et de données ont tendance à avoir la même adresse de poids faible. Une solution un peu plus économe en mémoire serait par exemple d'avoir une mémoire cache à 4 accès avec une politique LRU qui permettrait de gérer au moins les conflits sur les segments de données et de code. Par contre la gestion serait plus compliquée et nécessiterait l'intervention du processeur pour mettre à jour ce cache.

Il faut déterminer un bon compromis sur le nombre d'entrées à mettre dans la mémoire associative limitant à la fois la complexité et augmentant le nombre de pages.

Afin de simplifier le fonctionnement du système et la réalisation du circuit de routage, on considère qu'à un instant donné dans une partition une unique tâche parallèle est exécutée<sup>10</sup>. Cela évite d'une part d'avoir à envoyer avec chaque message le numéro de la tâche et d'autre part surtout d'avoir dans le circuit de communication une MMU devant connaître la configuration mémoire de toutes les tâches présentes, ce qui compliquerait beaucoup sa conception. Les 256 pages et les 85 tâches de l'exemple que nous avons donné sont en fait pessimistes si on adopte la simplification précédente. Il est plus astucieux de raisonner à nombre de pages constant mais à taille de page variable : la nouvelle contrainte est qu'une tâche et le système d'exploitation puisse loger dans les pages disponibles, celles-ci possédant une taille minimale pour augmenter l'efficacité de la gestion mémoire et de la pagination le cas échéant pour de petits programmes. Si par exemple le système d'exploitation fait de l'ordre du Mo, on pourra avoir des pages aussi petites que 4 Ko.

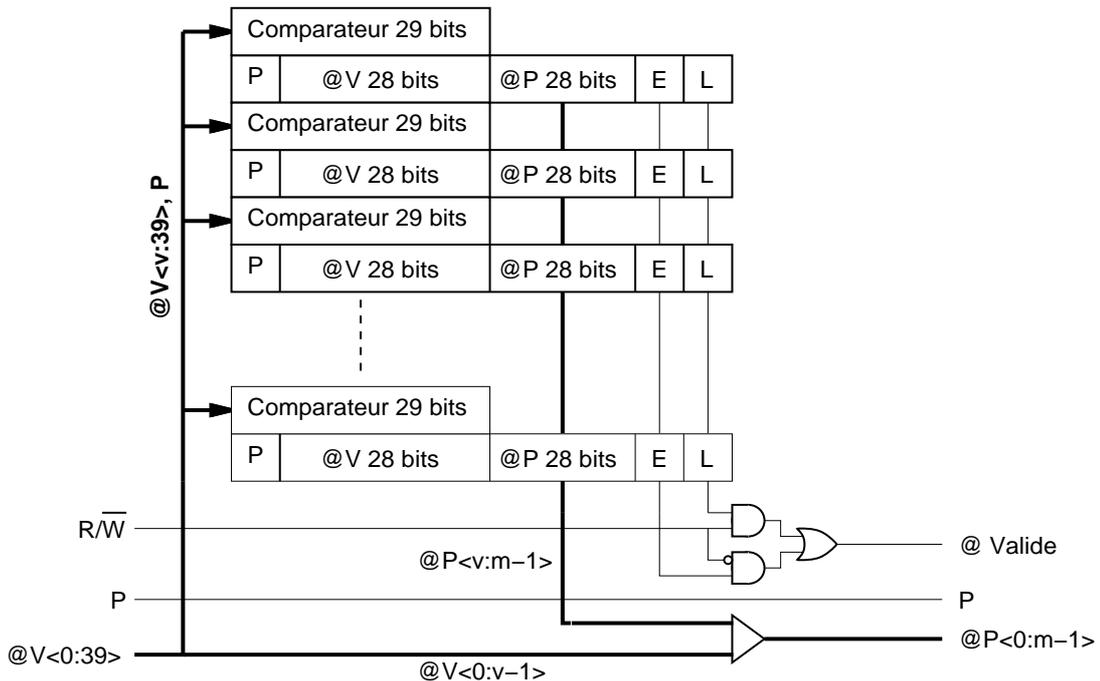
Afin de permettre une évolution de la mémoire de la machine, on rend globalement programmable la taille des pages, comme cela le nombre de pages physiques est toujours le même et notre système d'exploitation continue de fonctionner.

Un synoptique du système est présenté sur la figure 12.4. Un système de registre est capable de mémoriser les causes de violation de segmentation pour les fournir ultérieurement au système d'exploitation. Le système est programmable au niveau de la taille des pages ( $2^v$ ) et de la capacité mémoire du nœud ( $2^m$ ). Sur cet exemple en considérant la MMU à 256 entrées, une adresse virtuelle sur 40 bits et des pages de taille supérieure ou égale à 4 Ko, il faut une mémoire associative à 256 entrées sur 29 bits et une largeur de 30 bits, soit encore 256 comparateurs de 29 bits et 15 kbits de mémoire.

Il se peut qu'un programme ayant démarré avec peu de mémoire et des petites pages soit amené au cours de son exécution à demander beaucoup de mémoire. Dans ce cas, il faut changer sa taille de pages, mais alors il est fort probable que la connexité des

---

10. Notons que cela ne signifie nullement que le programme au niveau de chaque PE soit identique mais simplement que chaque programme a des droits d'accès sur les autres programmes de la même tâche parallèle. Cela autorise donc un fonctionnement purement MIMD si on le désire.

FIG. 12.4 - *Synoptique du système de gestion mémoire du circuit de communication.*

petites pages dans les grandes ne soit plus assurée. Le problème est résolu simplement au niveau du système d'exploitation en collectant toutes les petites pages et en les compactant dans les nouvelles grandes pages. Il n'y a aucun problème d'adressage étant donné que la MMU rend la modification transparente. Le temps de recopie est négligeable vu le débit mémoire important (plusieurs centaines de Mo/s) et que cela n'arrive que quelques fois dans la vie d'un programme.

Remarquons enfin que la taille des pages sur le circuit de communication et sur le processeur peuvent être différentes. Il suffit de s'assurer simplement au niveau système que la traduction des adresses sera bien identique.

Le problème de la mémoire virtuellement distribuée est quelque peu différent puisque celle-ci est gérée directement par la MMU du PE, le circuit de communication se contentant d'obéir au PE et ne provoquant pas de défaut de page. Afin de rassurer les défenseurs de la mémoire virtuellement distribuée inquiétés par la taille des pages exposées précédemment, on retrouve donc une granularité classique à ce niveau [NL91].

Un point que nous n'avons pas abordé est la tolérance aux pannes. Un moyen simple de le réaliser au niveau système est de sauvegarder régulièrement l'état d'une tâche afin d'être capable de la relancer dans cet état si entre temps le système s'arrête, par exemple à cause d'une panne.

Le problème est qu'il faut sauvegarder le minimum d'informations possibles pour que l'opération ne prenne pas trop de temps à chaque fois. Une manière de réaliser cette optimisation est de noter entre chaque point de sauvegarde quelles ont été les pages de mémoire modifiées pour n'avoir que celles-ci à sauvegarder entre chaque point de contrôle. Pour ce faire il suffit de rajouter dans chaque ligne de cache de la MMU du circuit de communication un bit initialement à 0 et mis à 1 dès que le contrôleur

de réseau écrit dans une case mémoire appartenant à la page correspondante. Notons que ce raffinement n'est intéressant que si le processeur du nœud possède un dispositif équivalent.

Le système présenté permet donc une économie de moyen, une gestion efficace des communications en adresses virtuelles, une taille de pages variable pour chaque processus et un système compatible avec un modèle de mémoire virtuellement distribuée. La seule restriction est que la taille des pages est aussi celle des blocs de pagination sur disque, le cas échéant. Mais on a de toute manière intérêt à accéder à de gros blocs sur disque pour des questions de débit.

Si on est prêt à perdre légèrement en performance on peut faire la même simplification que [Dig92a] par rapport à [MOT88b] en ce qui concerne la gestion du cache de la MMU. Dans le premier cas il n'y a aucun automate de remplissage de table de gestion de la mémoire : tout est du ressort du PE qui reçoit une exception à chaque manque de ligne de cache de pagination. Ainsi il suffit de mettre le même système dans le circuit de communication. Dans ce cas on peut utiliser n'importe quelle taille de page puisque toutes les pages physiques n'auront plus à avoir leur traduction mémorisée en permanence. Par contre, à chaque défaut de cache de pagination il y aura une exception qui avertira le PE de modifier l'état du cache de traduction du circuit de communication.

## 12.5 Le réseau

Il s'agit de présenter quelques clés de lecture différentes du chapitre 9. Une machine SPMD a des besoins différents d'une machine SIMD :

- dans la section 9.3, un des critères de choix était le synchronisme qui simplifiait le contrôle de la machine. Dans le cas du SPMD, ce synchronisme est une supposition pessimiste qui diminue au contraire les performances ;
- une autre considération était que la machine devait tenir dans une taille limitée, ce qui évitait de considérer toute extension future. De plus, le réseau était simple à câbler puisqu'on avait relativement peu de processeurs, par l'intermédiaire de câbles plats. Dans le cas d'une version SPMD étendue, on peut se permettre de réserver une salle spécialement pour la machine à condition d'avoir prévue une machine suffisamment extensible ;
- le contrôle du réseau devait être très simple pour être faisable rapidement bien sûr mais aussi loger dans un circuit de logique reprogrammable par PE. Le choix d'un réseau dynamique de type commutation de circuit avec sa résolution des conflits à l'ouverture des communications, l'absence intrinsèque de problèmes d'interblocage, de retardement indéfini, et de satellisation infinie, et son synchronisme en faisait un élément de premier choix (§ 9.3). Dans le cas d'une machine asynchrone développée avec plus de moyens, on peut avoir un réseau asynchrone de type *virtual cut-through* avec un système évitant les problèmes précédents associés à l'interblocage afin d'exploiter plus efficacement le câblage du réseau, au prix évidemment d'un circuit de contrôle beaucoup plus complexe.

L'utilisation de PEs plus performants que ceux de la version actuelle de POMP implique aussi un réseau plus performant exploitant au mieux le fait qu'une machine réelle

est construite dans un environnement à 2 dimensions, parfois à 3. Il ne s'agit pas de faire comme dans la CM-5 un réseau moins performant que dans la CM-2 ramené à la puissance des PES<sup>11</sup>.

Il est probable que le facteur limitant lié à la construction de la machine en plusieurs armoires sera le nombre de connecteurs entre armoires, plus que le nombre de pattes par PE réservées à la communication. Dans ce cas un réseau de type grille semble mieux adapté pour des raisons de bisection du réseau à utiliser au mieux [?] (§9.2.1.3) et de simplification du système évitant les interblocages tout en permettant une adaptativité maximale (§9.2.4.3). Cette simplification permet aussi d'adopter un routage de type *virtual cut-through* plus coûteux en mémoire par lien mais aussi plus efficace qu'un routage de type *wormhole* et même des canaux virtuels supplémentaires pour gérer la tolérance aux pannes si on le désire [LH91].

Comme on l'a vu dans la section 9.2.1.3, les tores se prêtent assez mal au partitionnement en sous-machines indépendantes, ce qui nous fait choisir un réseau de type grille.

Un bon compromis est donc un réseau de type grille 2D ou 3D avec des liens larges pour compenser le diamètre assez élevé de ce type de réseaux, avec une préférence pour la dimension 2 pour les raisons précédentes. Ainsi une machine réalisée sous forme de cartes contenant  $2 \times 2$  PES s'interfacera au fond de panier avec 8 liens de communication. L'utilisation de connecteurs standard de type FUTUREBUS+ [DuP90] permet de choisir des liens de 32 à 64 bits, ce qui représente de 128 à 256 pattes de communication par circuit de communication, ce qui est tout à fait raisonnable. Le choix d'une fréquence de communication de l'ordre de 100 MHz grâce à l'utilisation de câbles coaxiaux plats entre racks [Bel89] et de liaisons adaptées pour pipeliner les transferts de données permet d'obtenir un débit de communication brut de 800 à 1600 Mo par PE<sup>12</sup>. L'utilisation de meilleurs connecteurs [AUG89] peut permettre d'augmenter la fréquence de transmission et donc de diminuer le nombre de pattes à débit équivalent. Un facteur limitant à prendre néanmoins en compte est la dissipation thermique au niveau du circuit de communication liée à la commutation des nombreuses lignes adaptées.

D'un point de vue système, il est utile de proposer un système de DMA couplant la mémoire des PES et le réseau [SHI92] afin d'accélérer les opérations en allégeant les PES des tâches de transfert, contrairement à [?]. Afin d'augmenter encore le débit et d'améliorer l'indépendance du réseau vis-à-vis des tâches de calcul effectuées par les PES, on peut rajouter un système capable de lire dans la mémoire d'un PE distant et d'effectuer automatiquement des communications suivant une planification placée en mémoire.

### 12.5.1 Le partitionnement du réseau

Nous allons développer un système permettant le partitionnement du réseau en sous-ensembles convexes, particulièrement intéressants pour la gestion de entrées-sorties sur

11. Dans le cas de la CM2 [Thi87a] on avait un bit d'accès au réseau pour 16 PES ou 0,5 coprocesseur, soit  $1/32^{\text{ème}}$  de la bande passante d'un PE de 32 bits équivalent et dans le cas de la CM-5 [Thi91] 5 Mo/s pour 128 MFLOPS, soit moins de 1% du débit. Sur la CM-5 les applications devront demander moins de communications à performance équivalente. Évidemment il faudrait considérer des mesures plus précises que ces estimations grossières crêtes mais cela donne un ordre d'idée.

12. Bien entendu dans la réalité le débit est bien plus faible à cause des entêtes, des congestion et du temps de gestion logiciel des communications.

les bords de la machine, avec un mécanisme de protection système du réseau multi-machine.

Il y a plusieurs manières possibles de faire du partitionnement

- la plus simple et de ne pas en faire du tout. Les routines de communication vérifient par exemple avant l'émission ou la demande de réception que l'adresse de destination fait partie de la partition. Cela nécessite donc un passage en mode superutilisateur afin de garantir un certain niveau de protection et une routine de comparaison de limites. Si on réalise un circuit de communication capable d'envoyer automatiquement des paquets de données suivant une série d'informations en mémoire oblige la copie préalable de ces informations en mémoire protégée pour que le système soit sûr qu'une fois les vérifications de validité faites celle-ci ne pourront pas être modifiées par l'utilisateur ;
- l'autre méthode est de mettre des barrières logicielles délimitant topologiquement les partitions de la machine au niveau du réseau. L'intérêt est qu'on optimise le cas courant : celui où il n'y a pas d'erreur de programmation. Il n'y a pas de temps perdu à faire des comparaisons d'adresses et des recopies de descripteurs de paquets. La détection d'erreur est faite en déclenchant une exception sur le nœud où un paquet a voulu franchir une limite de partition.

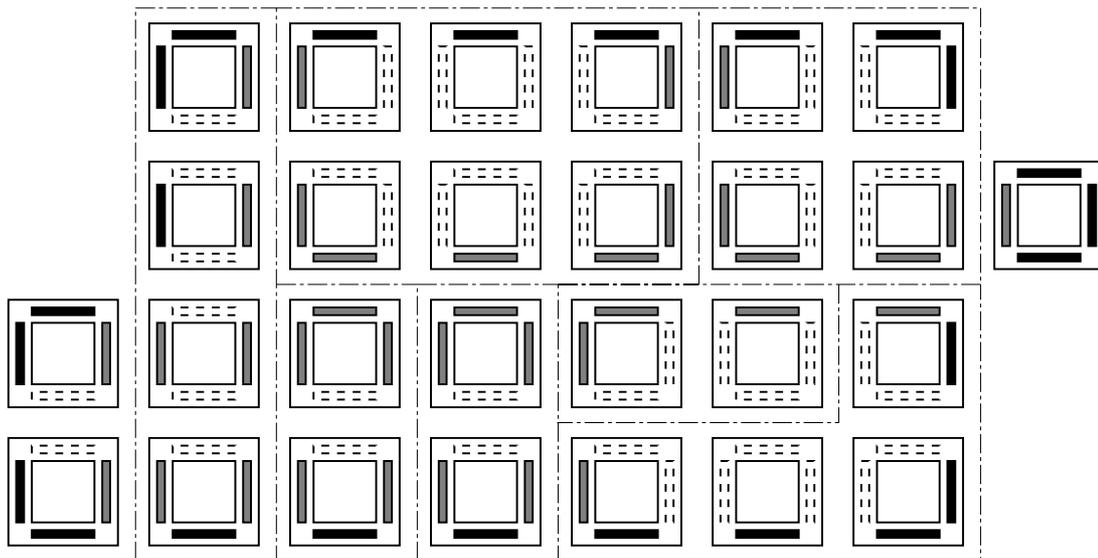
Dans cette étude, nous choisissons la 2<sup>ème</sup> méthode pour son efficacité et sa simplicité. Comme sur toute machine de type UNIX, 2 niveaux de privilège sont établis : le mode utilisateur et le mode superutilisateur (encore appelé superviseur ou tout simplement privilégié) qui ont l'avantage de pouvoir tout faire tout en ayant un codage efficace (1 bit). De même que sur une machine séquentielle l'utilisateur ne voit que la mémoire virtuelle et finalement une certaine virtualisation de la machine, en plus sur une machine parallèle partitionnable, l'utilisateur ne voit que sa partition, ne pouvant pas agir de manière inconsidérée avec les autres parties de la machine : il possède une machine parallèle virtuelle.

Nous verrons comment ces bits de partitionnement seront utilisés aussi pour partitionner les opérateurs globaux dans la section 12.6.

Un point que nous n'avons pas encore considéré est la tolérance aux pannes. L'isolement d'un ou de plusieurs PEs peut être fait par un mécanisme similaire codant le fait qu'il faut ignorer tout ce qui vient des PEs en panne et ne rien leur envoyer.

La figure 12.5 montre un exemple d'utilisation du principe de partitionnement avec des barrières de partitionnement. Chaque barrière possède 3 états (donc codés sur 2 bits) pour indiquer :

- une limite physique infranchissable qui est le bord physique de la machine (état indiqué en noir sur la figure) qui déclenchera une exception pour tout essai de franchissement par un message ;
- une limite logique infranchissable (en grisé) pour des messages utilisateur qui définit les limites de partitions (encadrées par des pointillés) sous peine de déclencher une exception. Ces limites peuvent être franchies par des paquets de niveau superviseur pour la gestion du système et des entrées-sorties ;
- l'absence de toute limite (en blanc et pointillé) entre processeurs d'une même partition, franchissable par tous les types de paquets.

FIG. 12.5 - *Principe et exemple de partitionnement.*

Les bits de partitionnement interviennent dans le système de routage au niveau du choix des liens en restreignant les chemins à longer les limites de partition. Ce système simple permet d'étendre l'algorithme de routage au delà des sous-espaces convexes, afin d'être capable de router des messages entre tout PE et les processeurs d'entrées-sorties disposés comme sur la figure 12.5. Cela a aussi l'avantage de pouvoir autoriser n'importe quelle partition convexe, pas seulement hyperparallélépipédique<sup>13</sup> ce qui est intéressant pour certains algorithmes et surtout pour la tolérance aux pannes (partition en bas à droite sur la figure 12.5).

Enfin, la méthode du bit de partitionnement permet de gérer simplement les diffusions sur des hyperespaces bornés comme nous le verrons plus loin.

### 12.5.2 Les communications régulières

Il s'agit d'une classe de communications couramment utilisée dans beaucoup de programme et il peut être intéressant de les optimiser.

Les communications sur grille peuvent être accélérées d'une part en commandant directement le réseau, puisque le motif déterministe de communication évite l'utilisation de messages contenant une adresse de destination et la logique de routage global utilisant cette adresse, d'autre part en utilisant l'algorithme présenté en §9.2.1.3 et sur la figure 9.4 pour limiter les conflits.

Par contre les communications sur grille suivant une direction parallèle à aucun des axes du réseau peut bénéficier de l'adaptativité du routage général pour augmenter le débit.

Le problème du contrôle direct du réseau dans le cas d'une machine partitionnable est qu'il se peut très bien que des paquets de niveau superviseur, par exemple lorsqu'une partie de la machine veut faire une entrée-sortie, aient à traverser une partition en train

13. Cela provient du fait que la topologie liée au système de routage n'est pas euclidienne mais de norme  $\|x\|_1 = \sum_{i=1}^d |x_i|$ .

de faire des communications sur grilles et donc ne pas pouvoir traverser cette zone car cela impliquerait un comportement non déterministe de la communication régulière.

Plusieurs solutions sont possibles, à savoir entre autres :

- doubler le réseau en un réseau pour routage aléatoire, un autre pour les communications régulières. C'est une solution coûteuse en fils, denrée déjà limitante dans la machine, d'autant plus que la plupart du temps seul un type de communication sera utilisé à un moment donné ;
- séparer le réseau en 2 réseaux comme précédemment seulement lorsqu'on veut effectuer des communications régulières. C'est une solution moins coûteuse mais qui complique tout de même la gestion du réseau ;
- ne pas prévoir de réseau pour les communications régulières, celles-ci étant optimisées logiciellement par une factorisation des paquets avant émission, un recouvrement des temps de calcul et de communication, et l'utilisation de l'algorithme de la section 9.2.1.3 dans le cas de communications sur grille par exemple.

Les 2 dernières solutions sont les moins coûteuses et la dernière étant de loin la plus simple, c'est très probablement un premier choix.

### 12.5.3 Les diffusions

Dans un modèle de programmation SPMD, un PE ou le processeur scalaire peut envoyer une valeur à certains PEs. Une manière évidente de le réaliser est d'utiliser un bus global sur lequel chaque PE écoute les valeurs qui l'intéressent. En fait, on constate qu'on rajoute à la machine déjà bien chargée un bus supplémentaire qui n'est en moyenne que très peu utilisé.

Le problème est très différent dans le cas d'une machine SIMD car la notion de diffusion est la clé de voûte de l'architecture : bien entendu on diffuse des valeurs scalaires mais surtout on diffuse les instructions vers tous les PEs. Le bus de diffusion et le bus d'instructions forment généralement qu'un seul bus, les valeurs scalaires étant enrobées dans des instructions *ad hoc*, comme dans le cas de POMP. Ce bus est donc constamment utilisé, contrairement au cas SPMD.

En plus, avoir un bus global partitionnable au niveau de n'importe quel hypervolume de la machine est difficile à réaliser, car au niveau de chaque fil on a déjà les problèmes que nous exposerons dans la section 12.6.1.1 multipliés par le nombre de fils du bus mais en plus il faut garder le système synchrone entre fils. Même en restreignant le partitionnement par exemple au niveau baie n'est pas très simple.

Il semble donc plus astucieux d'utiliser un mécanisme de diffusion tel que celui utilisé sur la machine PARAGON qui consiste à faire propager la valeur à diffuser à travers le réseau et de laisser sur chaque nœud une copie de cette valeur.

Alors qu'un bus global de diffusion ne permet pas de faire rapidement des diffusions sélectives telles que suivant un axe ou un hyperplan contenant les directions intrinsèques du réseau pour faire des répliques de vecteurs en matrice, etc, le système précédent rend ce genre de diffusions possible en rendant sélectivement anisotrope le système de propagation des valeurs.

Un algorithme simple réalisant le système de diffusion précédent peut par exemple s'inspirer de l'algorithme de routage du *e-cube*<sup>14</sup> [?].

On ordonne les  $n$  dimensions de l'hypergrilles et les bits  $d_i$  d'attribut d'un message de diffusion précisant pour chaque dimension si on doit diffuser ou pas le message.

Soit  $d = (d_1^0, d_1^1, d_2^0, d_2^1, \dots, d_n^0, d_n^1)$  le vecteur de diffusion d'un message  $\mathcal{M}$ . Chaque bit  $d_i^s$  indique si la diffusion aura lieu dans la dimension  $i$  et dans le sens  $s$ .

En un nœud, le message  $\mathcal{M}$  est diffusé dans toutes les dimensions  $i$  et sens  $s$  telles que  $d_i^s = 1$ , avec dans chaque cas le vecteur de diffusion  $d$  modifié pour avoir

$$d_i^s = 0 \quad (12.1)$$

$$\forall j \in [0, i-1] \cap \mathbb{N}^*, \forall u \in \{0, 1\}, d_j^u = 1 \quad (12.2)$$

De par le système de diffusion, tout paquet avec un bit de diffusion à 1 sera recopié sur la demi-droite correspondante. En raisonnant récursivement par dimension croissante on constate que le message de diffusion est bien recopié dans tous les PES concernés. Les équations précédentes restreignent progressivement le champ de diffusion à des demi-hyperespaces (équation 12.1) de dimension inférieure (équation 12.2), ce qui évite toute apparition de cycle et donc d'explosion du mécanisme de diffusion. On peut en outre démontrer que l'algorithme est optimal puisque le front de diffusion se déplace à la vitesse maximale du réseau.

Le fait de permettre des diffusions dans des demi-hyperespaces peut permettre d'économiser la bande passante du réseau dans des applications telles que les factorisations  $LU$  où on a besoin par exemple de diffuser une ligne de matrice sur toutes les lignes inférieures de la matrice.

Dans le cas d'un système convexe mais non étoilé (cas par exemple d'une machine hyperparallélépipédique avec quelques processeurs d'entrées-sorties sur les bords comme sur la figure 12.5), ce système de diffusion n'est pas sûr de fonctionner. Par contre en 2 diffusions successives, la diffusion d'origine suivie d'une autre diffusion de dimension inférieure, la diffusion est complète, dans le cas d'une machine de type exposé précédemment. Le fait de devoir remplacer certaines diffusions par 2 diffusions n'est pas gênant car elles ne concernent que les diffusions privilégiées et seulement celles qui adressent toute la machine avec les processeurs d'entrées-sorties.

Bien entendu, lorsqu'un paquet de diffusion rencontre une limite de partition ou de la machine, il est éliminé sans provoquer d'exception.

Enfin, le système est efficace lors de diffusions multiples puisque le fonctionnement est pipeliné au niveau de chaque message de diffusion. Ce mécanisme sera donc particulièrement adapté à des diffusions de type tous vers tous.

Dans le cas de la version universitaire de la machine, on peut exploiter le bus qui vient avec le DSP utilisé comme coprocesseur de communication pour diffuser des valeurs globalement. Chaque PE de partition émet et reçoit les valeurs sur une adresse propre à la partition et il existe une adresse commune à tous les PES pour réaliser les diffusions pour tous les PES de la machine. Le partitionnement de ce bus est donc logique et non physique. Cela n'est pas très gênant si on considère que le débit n'a pas à être très élevé et que le nombre de PES ne sera jamais très grand.

---

14. Il est intéressant de constater que ce système conçu pour éviter les interblocages peut être utilisé dans notre cas pour éviter une explosion combinatoire d'envois de messages de diffusion. Ces 2 problèmes sont assez semblables dans leur contextes respectifs.

### 12.5.4 Les attributs d'un paquet

Chaque paquet possède un certain nombre de bits d'attributs, en plus des données, pour permettre au réseau décrit précédemment de fonctionner correctement :

- le niveau de privilège du message sur 1 bit (un PE en mode superviseur peut bien entendu émettre un message avec l'attribut utilisateur) ;
- l'adresse relative de la destination, pouvant loger sur 16 bits si on considère qu'on n'aura raisonnablement pas plus de 64 K processeurs dans une première version ;
- une adresse éventuelle locale au PE destinataire stockée sur 48 bits dans une première version ;
- 1 bit précisant si une adresse est présente ou non et donc si à l'arrivée le message doit être stockée dans une file du circuit de communication ou bien dans la mémoire du PE à l'adresse indiquée ;
- 1 autre bit précisant si l'adresse de destination est physique ou virtuelle (une adresse physique provoque une exception lors d'une émission en mode utilisateur) ;
- les attributs de diffusion, 2 bits par dimension, soit par exemple 4 bits en 2D ;
- il faut indiquer si une diffusion en mode superutilisateur traversera ou non les limites des partitions, ce qui est fait par 1 bit ;
- 1 bit permettant de déclencher une exception sur arrivée complète du message. Permet de gérer aussi bien les exceptions à distance entre quelques PE, de réaliser des signaux d'exception globale au niveau utilisateur ou superviseur couplés avec le système de diffusion, de faire fonctionner la machine en mode objets et messages actifs dont un sous ensemble particulièrement intéressant est la mémoire globale virtuelle ;
- le fait qu'il s'agit ou pas d'un message de demande de données pour une réception, codé sur 1 bit ;
- la taille du message en octet sur 8 bits si c'est un message d'émission.

Une demande de réception de niveau utilisateur ne peut exiger une réponse en mode superviseur sous peine d'exception dès l'émission.

En mode superviseur, un PE peut contrôler chaque bit d'attribut d'un message pour faciliter la mise au point, le test et la gestion système de la machine.

En cas d'exception sur violation de privilège de communication, il faut prévoir des registres sur le circuit de communication pour accueillir au moins un des messages fautifs pour diagnostic ultérieur.

On peut rajouter si on le désire aux attributs des bits de correction d'erreur sur le réseau. Cela n'est pas très coûteux matériellement dans la mesure où ce système est déjà présent dans le circuit de communication au niveau du bus mémoire et donc qu'on peut le réutiliser.

Les émissions sur un PE suite à des demandes de réception doivent être prioritaires sur les demandes de communication du PE si on ne veut pas avoir d'interblocage.

La liste des attributs n'est pas exhaustive mais il y a un compromis à trouver si on ne veut pas gâcher de la bande passante.

## 12.6 Les opérations globales

Il est souvent nécessaire de faire appel à des opérateurs globaux rapides tels que les concentrations scalaires associatives ou les synchronisations globales permettant d'effectuer un contrôle de flot global.

En effet, c'est ce qui est très inefficace lorsqu'on programme une machine MIMD suivant un modèle SPMD sans mécanisme physique de synchronisation [Fil91a].

En plus, un système équivalent à un bus scalaire global est nécessaire d'une part pour l'administration système au démarrage et d'autre part pour récupérer une valeur d'un PE plus rapidement ou envoyer une valeur à tous les PEs, provenant soit du processeur scalaire, soit d'un PE.

On peut être amené à avoir la notion d'états utilisateur et superviseur sur ce bus pour accélérer la gestion système sans changement de tâche au niveau de l'utilisateur. Ainsi on peut simuler 2 bus différents sur ce bus global permettant de transférer par moment des informations systèmes de manière transparente aux opérations scalaires des utilisateurs. Un tel système a déjà été présenté dans la section 12.5.3 et nous allons nous intéresser plus particulièrement aux autres types d'opérations.

### 12.6.1 Les synchronisations

Il s'agit d'une tâche courante dans les machines MIMD, tout particulièrement quand on veut faire coopérer des processeurs suivant un modèle tel que le parallélisme de données [Ste90], au point que si un mécanisme réalisant cette opération n'est pas prévu, cette opération peut devenir un goulet d'étranglement [LB80, GE90]. Dans le cas d'une machine à mémoire partagée une réalisation logicielle est assez lente mais dans le cas d'une machine à passage de messages elle est encore plus lente.

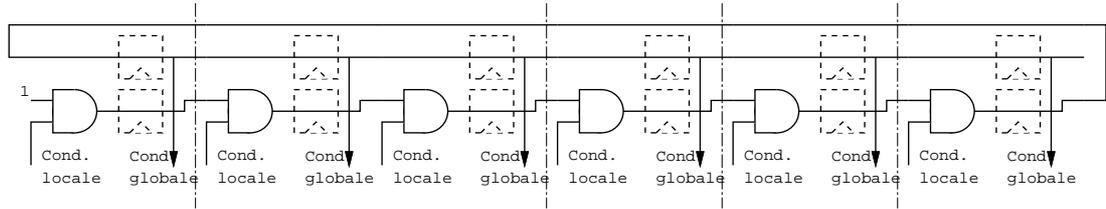
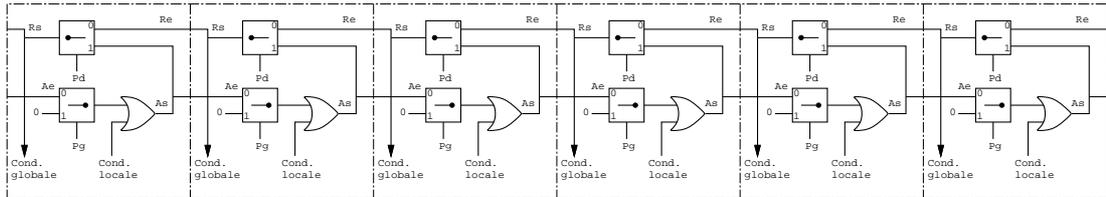
Un certain nombre de méthodes matérielles ont été développées pour accélérer les synchronisation [Fil91a]. Le calcul d'une condition global peut être fait de manière linéaire pipelinée, accélérée en réalisant les calculs sur arbre [Bre74], ou encore par diffusion de messages. On a dans notre cas une contrainte : que le dispositif de synchronisation soit partitionnable comme le reste de la machine. De plus il faut que le système soit facilement extensible, ce qui élimine par exemple les approches de [OD90b, OD90a, BP90] qui nécessitent la centralisation de  $N$  états et une mémoire associative d'une largeur de  $N$  bits.

Bien entendu il va falloir faire un certain nombre de simplification puisque la planification statique du code selon un mode VLIW [DOZ90, ZDO90] n'est pas généralement possible<sup>15</sup>.

Dans tout ce qui suit on calcule une condition globale. Qu'il s'agisse d'un *ou* global ou bien d'un *et* global importe peu en fait, les transformations étant triviales au niveau du logiciel. Il s'agit de réaliser la fonction qui est la plus rapide des 2 en fonction de la technologie disponible. De plus, sauf indication contraire, on considérera que toutes les conditions locales sont temporellement monotones<sup>16</sup> ce qui nous permet de simplifier

15. Parce que tout n'est pas connu à la compilation bien sûr, mais aussi parce qu'il est très difficile de prévoir le temps d'exécution dans un environnement réel où surviennent des interruptions et des exceptions. C'est ce qui est oublié dans [DOZ90, ZDO90], articles qui ne s'intéressent d'ailleurs qu'à des graphes de dépendances vraies et acycliques, ce qui en limite l'utilisation.

16. C'est à dire qu'une condition devenant vraie le restera après pendant toute la phase de calcul global [Fil92].

FIG. 12.6 - Exemple de *et global linéaire pipeliné*.FIG. 12.7 - Exemple de *ou global linéaire partitionnable*.

le système de synchronisation et n'est pas une hypothèse forte en pratique.

On trouvera dans [Fil92] une présentation générale des différentes approches.

### 12.6.1.1 Le *ou* global flottant

Il s'agit du système le plus simple : chaque PE est connecté à un fil global traversant toute la machine. Un PE a le pouvoir de connecter ce fil à un potentiel identique chez tout le monde. Lorsqu'aucun PE ne positionne le fil global, ce dernier revient à un potentiel de repos. Dans ce cas, tout les PEs peuvent savoir en mesurant ce potentiel qu'aucun n'a fixé le potentiel du fil. On a réalisé une fonction logique de *ou* global.

Le problème survient lorsqu'on veut partitionner le système. En effet cela revient à insérer sur le fil entre 2 PEs voisins un interrupteur pouvant diviser le *ou* global en 2 sous-systèmes. Cet interrupteur n'étant pas parfait, il possède une certaine résistance. Cela se traduit par une augmentation du temps de propagation des signaux car la capacité répartie à commuter est plus élevée<sup>17</sup>.

### 12.6.1.2 Un *ou* global pipeliné multidimensionnel

Une amélioration du système précédent est d'amplifier le signal entre chaque PE pour éviter les problèmes d'impédance évoqués précédemment et d'éventuellement pipeliner les opérations. Une telle approche a été utilisée par exemple sur la machine ARMEN [Pot91, Fil91a, Fil92] et le principe est présenté sur la figure 12.6, sous forme de *et* global.

Un tel dispositif permet une synchronisation en un temps  $2(n-1)\tau$  si on considère qu'entre chacun des  $n$  nœuds il faut un temps  $\tau$  pour que le signal se propage.

Une modification pour rendre le calcul global partitionnable par morceau est indiqué sur la figure 12.7. Afin que le système puisse être réalisé sur une structure non torique,

17. En considérant par exemple une chaîne de 1000 PEs, des interrupteurs de  $10\ \Omega$  [QS91] et une capacité de 50 pf par tronçon on obtient un temps  $\tau = RC$  de l'ordre 0,5 ms et ce en négligeant les effets inductifs et les temps de propagation.

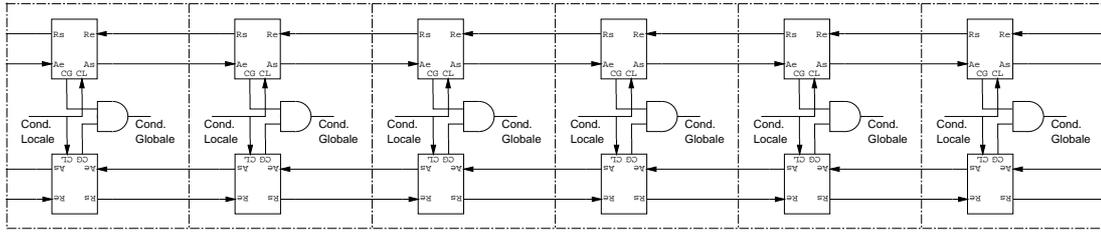


FIG. 12.8 - Ou global linéaire partitionnable symétrique.

le signal de retour a été inversé et se fait donc à contre-courant de la propagation des conditions partielles. Le partitionnement est contrôlé par les bits locaux  $P_g$  et  $P_d$  mis à 1 lorsque le PE courant a une limite de partition respectivement à gauche et à droite.

Le temps de calcul est toujours le même et est dans le pire des cas  $2n\tau$  lorsque le 1<sup>er</sup> PE est le dernier à mettre une condition valide. En effet, soit le temps de propagation nécessaire pour aller d'un PE  $i$  à un PE  $j$

$$t_{i,j} = (n - i)\tau + (n - j)\tau = (2n - i - j)\tau$$

$$\hat{t} = \max_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} t_{i,j} = 2(n - 1)\tau$$

Cette constatation suggère le doublement symétrique de la structure précédente comme indiqué sur la figure 12.8 afin de diviser par 2 le temps de propagation du dernier événement. La condition globale  $C'_{g,d}$  d'une partition  $[g, d]$  sera en fait celle des 2 sous-conditions globales, à savoir le *et* des sous-*ou* ou le *ou* des 2 sous-*et* respectivement calculés de 2 manières différentes

$$C'_{g,d} = \left( \bigvee_{i=1}^n c_i \right) \wedge \left( \bigvee_{i=1}^n c_{n+1-i} \right)$$

qui a pour effet de faire gagner la solution la plus rapide :

$$\hat{t}' = \max_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} (\min(t_{i,j}, t_{n+1-i, n+1-j}))\tau = (2\lceil \frac{n}{2} \rceil - 1)\tau$$

Évidemment, l'inconvénient de la méthode est le doublement du nombre de fils. Si on regarde d'un peu plus près au niveau des fils  $As$  de la figure 12.8 d'un processeur  $i$  appartenant par exemple à une partition  $[g, d]$  qu'on dispose des informations

$$As^d = \bigvee_{j=i}^d c_j$$

et

$$As^g = \bigvee_{j=g}^i c_j$$

or  $C'_{g,d} = As^d \vee As^g$  donc on peut économiser les signaux de retour  $Re$  et  $R_s$  et se contenter des signaux d'aller  $Ae$  et  $As$ .

Le temps de propagation est

$$\hat{t} = \max_{1 \leq i \leq j \leq n} (j - i)\tau = (n - 1)\tau$$

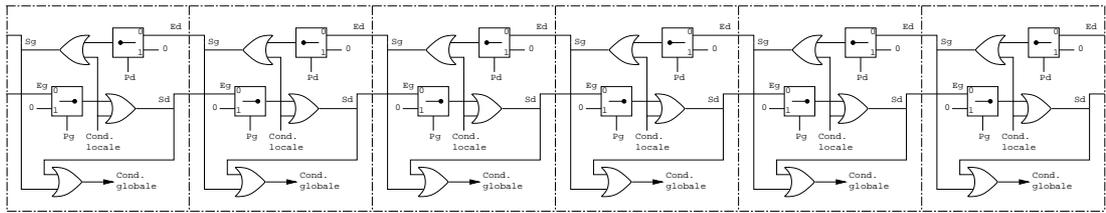


FIG. 12.9 - Ou *global linéaire partitionnable optimal*.

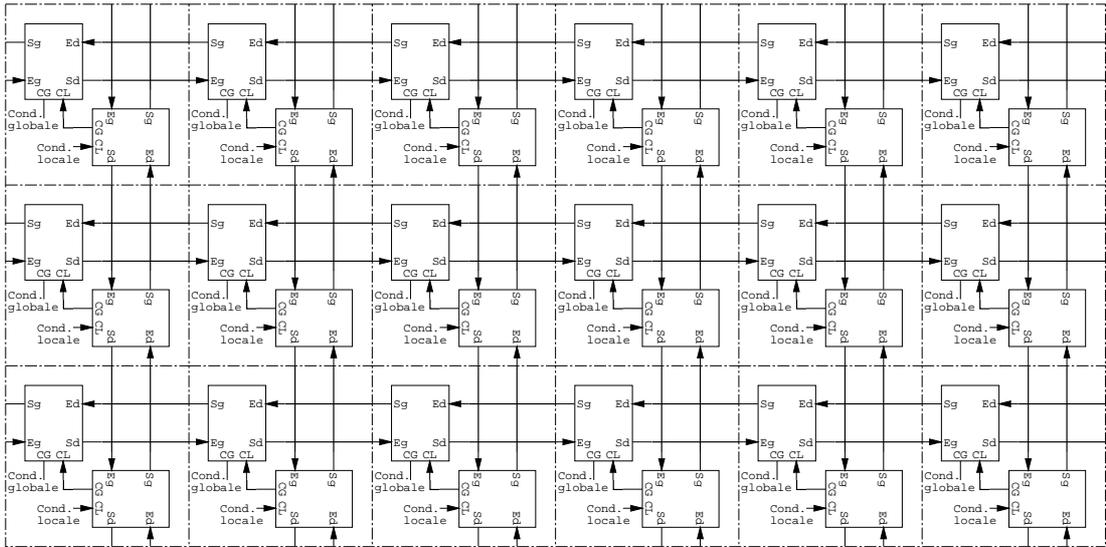


FIG. 12.10 - *Exemple de ou global linéaire partitionnable symétrique en 2D*.

qui est optimal puisque c'est le temps nécessaire à faire communiquer les 2 PEs les plus éloignés. De plus le câblage est optimal pour un réseau linéaire, si on considère qu'on doit envoyer et recevoir des signaux sur 2 fils différents, comme le montre la figure ??.

Dans notre cas, on veut faire une machine partitionnable en hyperparallélépipèdes. Il suffit de répéter dans chaque dimension la structure précédente, le résultat d'une dimension étant utilisé comme source locale pour la dimension suivante. La figure 12.10 présente une telle structure symétrique en 2D. Si la machine est une hypergrille de taille  $n_1 \times n_2 \times \dots \times n_d$  et qu'on considère seulement le temps de synchronisation d'une partition  $p_1 \times p_2 \times \dots \times p_d$ , celui-ci est

$$\hat{t}_{p_1, p_2, \dots, p_d} = \sum_{\delta=1}^d (p_\delta - 1) \tau$$

On peut noter que le temps est en  $\mathcal{O}(\sqrt[d]{n})$  et donc que le passage à des dimensions plus élevées accélère le mécanisme de synchronisation.

La méthode précédente demande donc en dimension  $d$  2 fils entre voisins et  $2d$  bits de partitionnement, un pour chaque bord de chaque dimension.

On va s'intéresser au passage à la limite où  $d = \log_2 N$ .

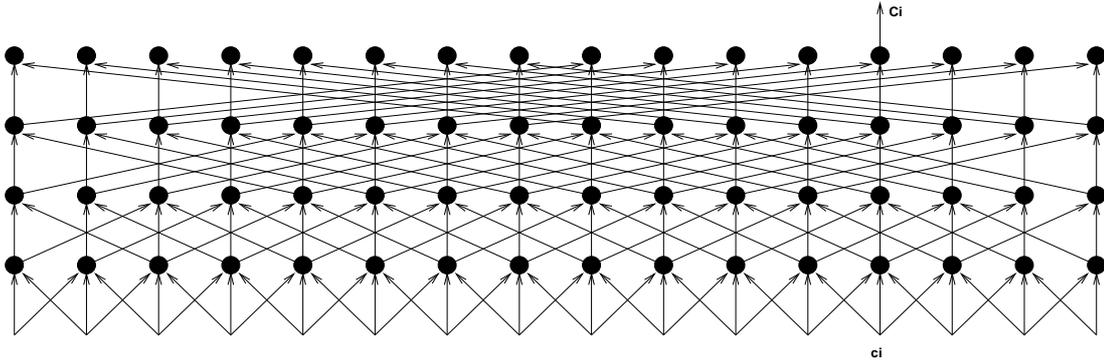


FIG. 12.11 - Opération préfixe parallèle symétrique.

### 12.6.1.3 Opération préfixe parallèle

Si on revient à la définition du problème, on veut calculer par exemple

$$C_i = \begin{cases} c_1 & \text{si } i = 1 \\ (s_{i-1} \wedge C_{i-1}) \vee c_i, & \text{autrement} \end{cases}$$

le plus rapidement possible. Si on dispose par exemple d'opérateurs à 2 entrées on peut utiliser par exemple les méthodes présentées dans [KS73, Kog74b, Kuc76, LF80] (§3.6.5). Dans la suite on va considérer plus particulièrement la méthode de doublage récursif [KS73] à cause de sa régularité de câblage.

Le système est partitionné entre un processeur  $i$  et un processeur  $i + 1$  en disant  $s_i = 0$  au lieu de  $s_i = 1$ . De plus il faut récupérer la condition globale sur toute la partition et pas seulement sur le PE du numéro le plus élevé dans la partition. Il faut donc doubler le câblage de manière symétrique comme indiqué sur la figure 12.11 par rapport à la figure 3.8 de la page 59. Chaque rond du niveau  $j$  ( $j^{\text{ème}}$  ligne à partir du bas) du PE  $i$  ( $i^{\text{ème}}$  colonne) sur la figure est un opérateur qui calcule

$$c_{i,j} = c_{i,j-1} + g_{i,j}c_{i-2^j,j-1} + d_{i,j}c_{i+2^j,j-1}$$

avec  $c_{i,0} = c_i$  la condition locale et  $c_{i,j} = 0$  pour  $i < 1$  ou  $i > n$ . Les coefficients de partitionnement s'expriment de manière simple et sont à calculer par le système une fois pour toute au moment du partitionnement :

$$g_{i,j} = \bigwedge_{k=i-2^{j-1}}^{i-1} s_k$$

$$d_{i,j} = \bigwedge_{k=i}^{i-1+2^{j-1}} s_k$$

Le temps de synchronisation avec la méthode précédente est

$$\hat{t} = \lceil \log_2 n \rceil \tau$$

et de manière générale seulement  $\lceil \log_2 p \rceil \tau$  pour une partition de  $p$  processeurs. Par contre le nombre de pattes de synchronisation par PE est  $4 \lceil \log_2 n \rceil$ . Le réseau sous-jacent

est une restriction au niveau des bords (puisque le réseau n'est pas torique) de type réseau DMF [Fen74], IADM [RFS88] ou gamma [PR82]. La complexité au niveau du câblage est inférieure à celle du double de l'hypercube.

On peut adapter la méthode pour permettre le calcul sur des hypervolume de la même manière que la méthode précédente sur la figure 12.10. Dans ce cas le temps de synchronisation reste identique, modulo les problèmes de divisibilité, car le réseau de la méthode étant basée sur des permutations de type hypercube et les hypergrilles sont incluses dans les hypercubes [SS88]:

$$\hat{t}_{p_1, p_2, \dots, p_d} = \sum_{\delta=1}^d \lceil \log_2 p_\delta \rceil \tau$$

avec  $\hat{t} = \log_2 \prod_{\delta=1}^d p_\delta$  si tous les  $p_\delta$  sont des puissances entières de 2. On ne gagne pas lors du passage sur une hypergrille comme dans la section 12.6.1.2.

En conclusion la complexité est assez élevée même si on considère que chaque lien n'a qu'un bit de large si on veut faire une machine avec beaucoup de processeurs. Par contre comme le calcul est complètement combinatoire, la latence est très faible.

Une simplification du système précédent est de reprendre l'approche de [Ble89a, chapitre 13] pour réaliser le calcul sur un arbre en modifiant l'algorithme pour rendre l'opération symétrique nécessaire à la réduction-diffusion segmentée désirée. On peut améliorer l'intégration du système en mettant plutôt un nœud de l'arbre par PE en changeant quelque peu l'algorithme. Cela ne change rien à la complexité de la méthode sur arbre d'origine qui demande  $\lceil \log_2 n \rceil$  pas de montées et autant pour la phase de redescente. L'algorithme n'est pas purement combinatoire et nécessite une horloge de pipeline. Le temps d'exécution est donc

$$\hat{t} = \lceil \log_2 n \rceil \tau_p$$

où  $\tau_p$  est la fréquence d'horloge du pipeline tenant compte du temps de propagation maximal et du temps de gestion du pipeline. On a donc  $\tau_p > \tau$  et probablement beaucoup plus élevé à cause des marges d'incertitudes. Par contre le nombre de pattes de synchronisation par PE n'est plus que de 3 dans chaque dimension.

Une autre simplification est de remarquer qu'on peut réaliser notre opération préfixe symétrique en 2 opérations préfixe classiques, une dans chaque sens. On pourra utiliser seulement  $2 \lceil \log_2 n \rceil$  pattes de synchronisation mais on est obligé de réaliser le calcul de manière non purement combinatoire, ce qui demandera beaucoup plus de temps.

#### 12.6.1.4 Comparaison

On a présenté 3 méthodes différentes de synchronisation fournissant un compromis entre la complexité, le câblage, la vitesse.

Une méthode pour obtenir encore plus de compromis est d'utiliser un mélange des méthodes précédentes, par exemple la méthode linéaire entre baies ou racks, là où la machine est extensible, et la méthode parallèle préfixe là où l'extension sera nulle, c'est à dire au niveau d'un fond de panier par exemple.

Il est probable que le  $\tau$  de la méthode de la section 12.6.1.2 soit plus petit que dans le cas des méthodes parallèles préfixes car les temps de transfert sont courts de voisin à voisin. C'est un argument en faveur des calculs linéaires. Dans les exemples qui vont

TAB. 12.2 - Comparaison entre les méthodes de synchronisation.

$n_x \times n_y$	Nombre de PEs	Méthode de synchronisation		
		linéaire symétrique	parallèle préfixe	mixte
$1 \times 1$	64	$18\tau$ 270 ns	$6\tau$ 90 ns	$6\tau$ 90 ns
$4 \times 1$	256	$30\tau$ 450 ns	$8\tau'$ 240 ns	$9\tau$ 135 ns
$16 \times 4$	4096	$126\tau$ 1890 ns	$12\tau'$ 360 ns	$24\tau$ 360 ns

suivre, on prendra donc  $\tau = 15$  ns dans le cas général et  $\tau' = 30$  ns pour le cas de l'opération parallèle préfixe sur la grande machine.

Le tableau 12.2 donne une estimation pour quelques configurations de machines 2D composées de  $n_x \times n_y$  racks de  $4 \times 16$  PEs.

Étant donné la technologie actuelle des processeurs, le fait que nos estimations sont pessimistes et qu'il faille rajouter un temps de gestion système, il est fort probable que la méthode la plus simple, même si elle est la plus « lente », suffise même dans le cas d'une grosse machine.

### 12.6.1.5 Considérations de niveau système

Après avoir décrit le matériel nécessaire pour faire des calculs globaux simples mais efficaces, il faut déterminer leur mode d'utilisation pour en faire des barrières de synchronisation ainsi que le nombre nécessaire au bon fonctionnement de la machine.

Dans un modèle de programmation SPMD, et de manière générale MIMD, il faut garantir que les dépendances sont bien respectées à l'exécution entre chaque processeur. Les seules relations étant des communications, le contrôle à effectuer est autour de ces communications. On retrouve les 3 types classiques de dépendance [Kuc76, dD91] multipliés par le fait que la production ou l'utilisation d'une valeur peut être locale ou distante. Par exemple un processeur ayant besoin d'une donnée (donc dépendance vraie  $\delta^t$  doit attendre soit sa production par lui-même, soit son arrivée par le réseau depuis un autre processeur. Toutes les configurations sont résumées dans le tableau 12.3. Les problèmes de dépendances au niveau d'un processeur étant à la charge du compilateur et internes à chaque PE, ils ne font intervenir aucun mécanisme de synchronisation globale. Ensuite il existe les dépendances entre réseau et processeurs qui nécessitent soit une synchronisation au niveau du réseau (on attend que les communications en cours sont finies pour être sûr que les émissions ou réceptions ont bien été faites, synchronisation indiquée par «  $\circ$  » sur le tableau) soit une synchronisation au niveau des processeurs (une barrière de synchronisation entre les PEs assure que ceux-ci ont bien écrit ou lu toutes les variables qu'ils désiraient, opération indiquée par une fourche inversée «  $\nabla$  »).

On utilise des synchronisations globales pour résoudre des dépendances sur variables car il est difficile de réaliser simplement une attente sur une variable au niveau

TAB. 12.3 - Dépendances dans une machine MIMD.

		Vers					
		réseau			processeur		
De	réseau	$\delta^t \rightarrow \circ$	$\bar{\delta} \rightarrow \circ$	$\delta^o \rightarrow \circ$	$\delta^t \rightarrow \circ$	$\bar{\delta} \rightarrow \circ$	$\delta^o \rightarrow \circ$
	processeur	$\delta^t \rightarrow \mathfrak{h}$	$\bar{\delta} \rightarrow \mathfrak{h}$	$\delta^o \rightarrow \mathfrak{h}$	$\delta^t$	$\bar{\delta}$	$\delta^o$

d'un processeur<sup>18</sup>. Synchroniser les processeurs sur une condition globale est donc une approximation pessimiste mais dans la pratique cette approximation n'est pas trop gênante car en général les communications sont regroupées en phases dans le cas d'une application à connotation parallélisme de données. De plus, lorsque l'application le permet (souvent lorsque le degré de parallélisme est élevé), on peut essayer de recouvrir temps de communication et temps de calcul.

### La barrière de synchronisation

Une barrière de synchronisation empêche tous les processeurs de continuer l'exécution de leurs programmes tant qu'ils ne sont pas tous arrivés à l'endroit de la barrière dans le programme.

Le simple *ou* global ne suffit pas pour réaliser une barrière de synchronisation sûre. En effet dans le cas d'un programme contenant 2 barrières il se peut qu'un des PE ait pris de l'avance sur un autre n'ayant pas terminé de tester l'état de la première barrière<sup>19</sup> au point que celui-là ait déjà atteint la barrière suivante : le 2<sup>ème</sup> processeur croit recevoir la reconnaissance de la 1<sup>ère</sup> barrière alors qu'il s'agit de la seconde.

Pour éviter ce genre d'étreinte mortelle il faut coder un état global supplémentaire indiquant le début d'une barrière, ce qui nécessite un *ou* global de plus par exemple.

Lorsqu'un PE atteint une barrière de synchronisation, il annonce à tout le monde son entrée par la condition globale d'entrée en synchronisation et attend que tous les PEs ait reconnus la sortie de synchronisation sur une autre condition globale. L'algorithme est décrit en pseudo-langage informatique sur la figure 12.12 où  $p$  est le numéro de chaque processeur.

Afin de décharger le processeur les manipulations des signaux  $eb$  et  $sb$  peuvent être gérés par un petit automate, celui-là se contentant de démarrer l'entrée en synchronisation et de tester ultérieurement un drapeau indiquant que tous les PEs se sont bien synchronisés [Fil92].

Ces phases d'entrée en synchronisation et de sortie donne l'idée de l'optimisation logicielle proposée dans [GE89, GE90]: si on divise l'appel système « barrière de synchronisation » en un appel d'entrée et un appel de sortie bloquant distinct, on peut élargir la zone entre les 2 en faisant percoler [FR72] des instructions du programmes ne modifiant pas sa sémantique en travaillant à partir du graphe de dépendance. On parle alors de barrières floues et cela revient d'une part à anticiper la synchronisation et d'autre part à retarder l'échéance de la synchronisation. L'intérêt est qu'on recouvre le temps de synchronisation globale avec des phases de calcul lorsque c'est possible, ce

18. On n'étudie pas ici le cas d'une machine à flot de données.

19. Par exemple parce qu'il a été interrompu par une tâche système.

---

```

Initialisation :
     $\forall i \in [1, N], eb_i \leftarrow 0, sb_i \leftarrow 0$ 
    ...
Entrée en synchronisation :
     $eb_p \leftarrow 1$ 
    ...
Sortie de synchronisation :
    tant que ( $\bigwedge_{i=1}^N eb_i \neq 1$ )
        attendre
     $sb_p \leftarrow 1$ 
    tant que ( $\bigwedge_{i=1}^N sb_i \neq 1$ )
        attendre
     $eb_p \leftarrow 0$ 
     $sb_p \leftarrow 0$ 

```

---

FIG. 12.12 - *Algorithme de synchronisation globale.*

qui se traduit par une augmentation de la vitesse d'exécution du programme. Il s'agit d'une optimisation tout à fait comparable à l'introduction des branchements retardés dans les processeurs RISC.

Si un processeur  $i$  a  $eb_i = 1$  et  $sb_i = 1$  constamment il ne participe plus à la barrière mais sans pour autant l'empêcher de fonctionner. Un PE peut donc décider de laisser passer un certain nombre de points de synchronisation avant d'être concerné, ce qui nous permet de réaliser des barrières de synchronisation à planification statique [OD90b, Fil92], comme le montre la figure 12.13. Afin de faciliter ce genre de manipulation, il vaut mieux que la barrière soit contrôlée par le PE lui-même, sur interruption. Mais cela peut aussi être réalisé matériellement par un mécanisme de compteur d'inactivité qui n'est pas sans rappeler celui de la section 7.1.3.

### La fin de communication

On a besoin en particulier d'un signal indiquant que les communications sont terminées, afin de pouvoir être certain que plus aucune instruction de réception particulièrement est en cours<sup>20</sup> que chaque processeur met en fonction dès qu'il a envoyé toutes ses instructions de communications et qu'il attend une synchronisation.

Comme des paquets superviseurs peuvent à tout moment transiter sur le réseau, il faut prévoir un doublement du système de détection de fin de communication : un pour le niveau utilisateur et un, moins utile, pour le niveau superviseur.

Le problème du test de fin de communication est que la condition locale de terminaison n'est pas monotone. En effet, les paquets circulant d'un PE à un autre, un PE

---

20. Dans le cas d'émissions il n'y a pas de problème puisqu'un processeur sait plus facilement qu'il a fini de communiquer. On pourrait faire la même optimisation pour les réceptions mais cela nécessiterait d'avoir en mémoire toutes les requêtes envoyées pour pouvoir vérifier que toutes les données sont revenues.

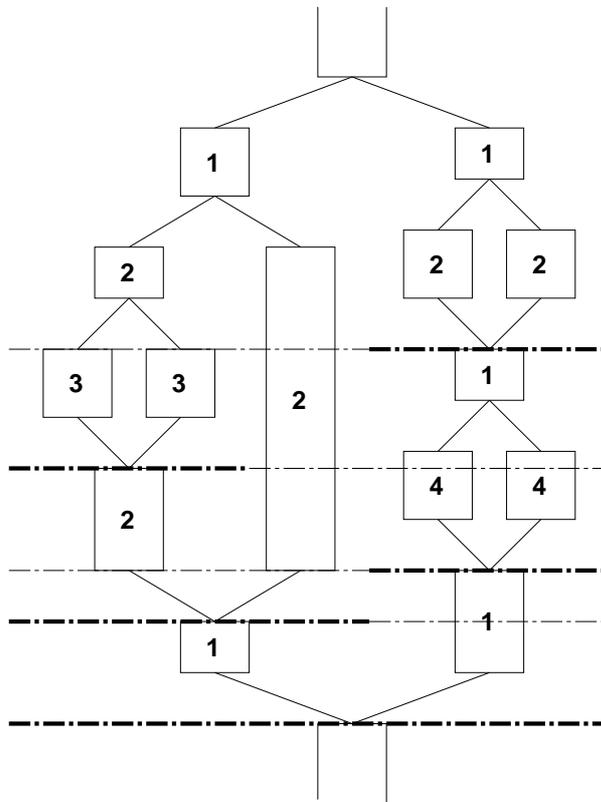


FIG. 12.13 - Barrière à planification statique.

peut ne plus voir de paquet à un moment donné alors qu'un paquet arrivera quelques instants plus tard.

Néanmoins les paquets étant plus larges que les liens, il faut plusieurs cycles pour passer d'un PE à un autre et à ce moment là les PEs sur le chemin constatent tous la présence d'une communication. Si la topologie du réseau est exactement la même que celle du calcul du système de synchronisation il n'y a pas de problème: la condition d'occupation se propage plus vite que les messages. Par contre si ce n'est pas le cas il se peut que le *ou* global de fin de communication détecte temporairement une fin de communication parasite. C'est un cas qui peut arriver par exemple à cause des différences de propagations entre circuits lorsqu'un message se déplace sur une dimension qui n'est pas celle du niveau inférieur du *ou* global linéaire.

Le problème se résout facilement si on est capable de borner le déphasage: il suffit alors de réagir seulement au bout d'un temps supérieur à celui-ci lors de la présence d'une fin de communication globale [Fil92]. Un système avec un compteur programmable peut remplir cette tâche et ce temps d'attente est de toute manière inférieur au temps de calcul du *ou* global qui est faible devant le temps total de communication.

La gestion des entrées-sorties se faisant plutôt de PE à processeur d'entrées-sorties, une synchronisation globale est une approximation trop forte dans ce cas. Il vaut mieux

terminer les phases de communications par un envoi du nombre de paquets étant arrivés pour savoir si tous les paquets sont reçus et donc que la communication est terminée et donc de gérer un protocole point à point.

Si le circuit de communication dispose d'un système capable de lire et réécrire une valeur en mémoire (ce qui est nécessaire pour écrire des valeurs de taille plus petite que le mot mémoire de base dans le cas par exemple d'un nœud avec un ALPHA à cause de la correction d'erreur en mémoire), en couplant la diffusion avec un additionneur capable de modifier une valeur en mémoire on peut réaliser la méthode de synchronisation par compteur-accumulateur utilisée par exemple sur la machine SEQUENT BALANCE [Lub90]. Pour simplifier encore la gestion, on peut mettre un attribut déclenchant une interruption lors du passage à 0 de la case mémoire. Ainsi on peut réaliser un système de synchronisation totalement partitionnable de manière quelconque sur toute la machine, pas seulement sur des sous-hypervolume. La méthode étant néanmoins plus lente car utilisant les diffusions, dans le cas courant on utilisera la méthode avec l'opérateur de synchronisation câblé.

#### 12.6.1.6 Factorisation de barrières de synchronisation par compteur

Les barrières de synchronisations sont nécessaires dans une machine MIMD programmée selon un modèle à parallélisme de donnée lorsqu'on veut exécuter de manière asynchrone les branches alternatives d'un **where**. Une imbrication de **where** se traduira par une exécution de plus en plus asynchrone avec souvent une barrière de synchronisation à la fin d'un **where** s'il contenait des communications. On va s'intéresser à l'optimisation d'un cas particulier indiqué sur la figure 12.14 et qui est donné en exemple dans [GE90].

Il correspond en POMPC par exemple à une imbrication de **where/elsewhere** tels qu'aucun **where** n'est contenu dans un **elsewhere** avec une communication dans chaque branche de manière à ce qu'autant de synchronisations soient nécessaires. Chaque fourche donne naissance à 2 flots parallèles d'instructions qui se synchronisent après pour reformer le flot initial. On constate qu'il y a alors une parfaite imbrication des synchronisation et qu'on peut appliquer la méthode de factorisation à base de compteur présentée en § 7.1.3 : à chaque fourche on augmente le niveau de synchronisations en attente de 1 et à chaque synchronisation on décrémente le compteur de synchronisations en attente. Plutôt que de transmettre le numéro de synchronisation il suffit de transmettre l'information d'incrément ou de décrémentation. Une synchronisation est effective sur un PE lorsqu'il reçoit la synchronisation du numéro qu'il avait avant la fourche.

Le problème est de coder les états d'incrément et de décrémentation. Cela est réalisé de manière simple découlant d'une étude de la figure 12.12. On constate que l'algorithme de synchronisation se termine par  $eb_p \leftarrow 1$  et  $sb_p \leftarrow 1$ , l'ordre important peu. La distinction de cet ordre nous permet justement de coder les 2 états qu'ils nous faut : par exemple l'incrément est codée par une synchronisation supplémentaire se terminant par  $sb_p \leftarrow 1$  puis  $eb_p \leftarrow 1$  alors que la décrémentation ou synchronisation « normale » est terminée par  $eb_p \leftarrow 1$  puis  $sb_p \leftarrow 1$ .

On peut ainsi réaliser simplement à coût matériel nul une barrière à planification dynamique [OD90a] dans ce cas particulier d'imbrication de barrières.

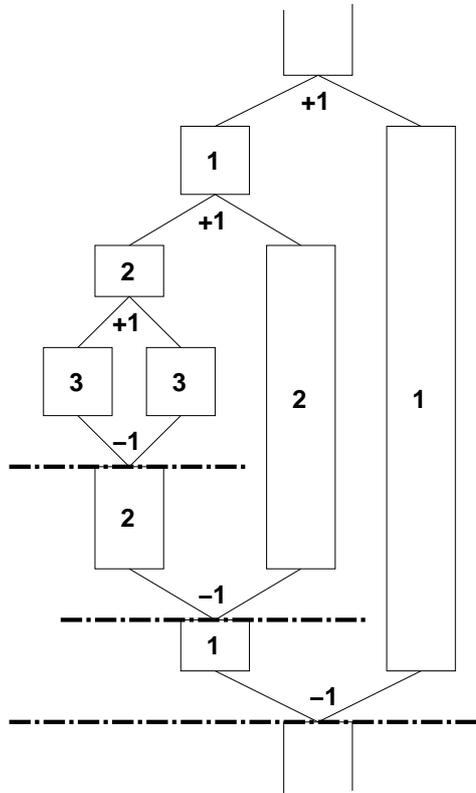


FIG. 12.14 - *Optimisation d'un cas d'imbrication de synchronisations.*

### 12.6.1.7 Conclusion

Il semble utile d'avoir un signal de synchronisation partitionnable, un signal de synchronisation partitionnable de niveau superviseur et un signal de synchronisation globale de niveau superviseur. Chaque signal peut aussi être utilisé pour détecter les fins de communications ou une condition globale<sup>21</sup>.

Cela nous fait  $3 \times 2 \times 4d = 24d$  fils de synchronisation par PE, soit 48 fils par PE, 12 fils entre chaque voisin. On peut éventuellement réduire ce nombre si on est prêt à faire du multiplexage temporel sur les signaux globaux superviseurs, moins mis à l'épreuve *a priori*.

On peut rajouter un signal d'exception partitionnable afin d'accélérer les changements de tâches mais étant donné que le réseau peut déclencher des exceptions à distance et que le changement de tâche est souvent assez lourds, on peut peut-être économiser la réalisation de système.

---

21. Sachant que la fin de communication privilégiée partitionnable détecte en fait les fins de communication de niveau utilisateur, condition utile pour autoriser les changements de tâche, après un arrêt de toutes les émissions.

### 12.6.2 Les concentrations associatives

Les autres opérations globales importantes sont les concentrations associatives qui permettent d'appliquer un opérateur associatif à un sous-ensemble d'une variable parallèle. Comme ces opérations sont moins souvent utilisées que les opérations telles que le *ou* global et les synchronisation, il ne semble pas nécessaire de les réaliser matériellement mais seulement de faciliter leur réalisation logicielle sur le réseau de la machine.

Une concentration sur un grand nombre d'éléments aura intérêt à être faite suivant un arbre binaire, voire  $2d$ -aire dans le cas d'une hypergrille de dimension  $d$ , afin d'éviter les conflits mentionnés sur la figure 9.4 de la page 190. Notons que cette optimisation fonctionne aussi bien avec les concentrations régulières qu'aléatoire mais elle est plus simple à mettre en œuvre pour les concentrations régulières<sup>22</sup>.

Comme on ne veut pas réaliser dans le circuit de communication un opérateur capable de traiter tous les types d'opérations associatives, y compris en nombres flottants, les opérations sont à la tâche du processeur. Afin d'éviter un accès inutile en mémoire, on rajoute un file d'attente<sup>23</sup> au niveau du circuit de communication pour recevoir des paquets possédant l'attribut de concentration. Ainsi lors d'une routine de concentration associative chaque processeur est en attente d'une valeur de type prévu à la compilation, près à la combiner avec une valeur locale et à l'envoyer à son père sur l'arbre de réduction ou bien la diffuser vers tous les PES par exemple.

## 12.7 Mise au point de la machine

Le problème de la mise au point d'une machine parallèle est encore plus crucial dû au fait du grand nombre de processeurs et doit donc être pris en compte dès les premières phases de conception.

Le problème principal provient du développement simultané du logiciel et du matériel alors que le matériel est souvent en retard sur le logiciel surtout en début de développement. Généralement on ne peut pas faire fonctionner la machine sur une application réelle, même avec des performances dégradées. Cela est dû au fait que souvent une machine est composée de différentes cartes par exemples et que celles-ci ne sont pas toutes réalisées en même temps.

Nous allons développer une technique simple et économique permettant de tester, configurer, charger le système d'exploitation et effectuer des entrées-sorties dégradées.

Le test d'une machine au plus bas niveau se fait en général à l'aide de quelques fils de « *scan path* » qui traverse tous les circuits de la machine. Grâce à ces fils on peut envoyer de manière série des données et en recevoir. On peut donc charger des registres de test dans les circuits et relire des valeurs. La mise au point est en général de bas niveau et il se peut que le chemin soit interrompu au niveau d'un circuit hors d'usage, isolant ainsi une partie de la machine.

D'autre part il faut développer un système capable de faire du test de plus haut niveau : charger des routines de test que chaque PE pourra exécuter localement. Mais

---

22. Dans le cas d'une réduction massive mais non régulière, on peut générer un arbre à partir d'opérations préfixes parallèles pour pouvoir exécuter plus rapidement les concentrations.

23. D'après une idée de Nicolas PARIS qui propose aussi un système d'exception en cas de débordement.

en général le chargement d'un programme sur une machine parallèle et la réalisation des entrées-sorties nécessite d'avoir une interface sur le monde extérieur qui fonctionne, ce qui n'est pas au début des développements.

En ce qui concerne le test de haut niveau, l'approche de la machine PARAGON est d'avoir une interface ETHERNET par PE. Comme cela on peut se connecter depuis le monde extérieur sur chaque PE. Cela nécessite d'avoir localement suffisamment de mémoire permanente pour faire les 1<sup>ers</sup> tests et faire fonctionner le protocole TCP/IP, ce qui n'est pas rien. De plus cette interface ETHERNET est redondante une fois que la machine et son réseau rapide est lancée. Ce qu'il faudrait serait un petit processeur possédant une interface de type ETHERNET qui soit capable de prendre en main le contrôle de la carte, soit bon marché et petit.

Or de tels circuits font actuellement leur apparition chez plusieurs constructeurs sous licence [Ech91, Ech92]. Ils possèdent une interface proche d'ETHERNET à 1,25 Mbits/s sur différents supports physiques dont la paire torsadée, 3 processeurs de 8 bits intégrés (2 s'occupent du réseau, un est dédiée à l'application), de la mémoire volatile, de la mémoire permanente réinscriptible localement, de la mémoire morte contenant le logiciel de gestion du réseau, des entrées-sorties capables d'agir sur le monde extérieur.

Bien que prévu pour le marché très grand public, il s'avère que ces circuits ont des caractéristiques proches de l'idéal dans notre cas. Il suffit de prévoir 11 pattes du circuit de communication pour être capable de se connecter au petit circuit NEURON de 32 pattes. Chaque circuit de chaque PE est relié à tous les autres circuits, y compris un circuit de commande globale. Le fonctionnement du système se fait en 3 phases distinctes :

- 1° dans un premier temps le logiciel de test est diffusé sur tout les circuits. Chaque circuit peut se tester lui-même bien sûr mais surtout tester le circuit de communication du nœud auquel il est relié. Le circuit global sous contrôle de la station de travail servant de console collecte le résultat des tests et peut déjà éliminer des cartes ;
- 2° ensuite la station de travail diffuse le logiciel de téléchargement sur les circuits NEURON. Pendant ce temps là le PE, considérons un ALPHA pour l'exemple, a chargé son EEPROM contenant un logiciel capable de charger en mémoire un programme depuis le circuit NEURON à travers le circuit de communication. La station de travail peut commencer à diffuser le système sur chaque nœud et faire des tests exhaustifs avant de démarrer le système d'exploitation ;
- 3° enfin, le réseau sur paire torsadée est utilisé comme système d'entrées-sorties primitives après diffusion du logiciel de communication. Cela permet d'utiliser la machine normalement, si ce n'est que les performances des entrées-sorties avec le monde extérieur sont limitées à 1,25 Mbits/s. Cela permet néanmoins de faire fonctionner raisonnablement des applications utilisant du graphique par exemple si on considère.

Une manière d'augmenter le débit du système serait de partager la machine pour la connecter à plusieurs réseaux primitifs différents possédant chacun un contrôleur de réseau.

En ce qui concerne la mise en œuvre du système global de commande, il suffit d'acheter une carte toute faite de développement du circuit qu'on mettra dans un

ordinateur individuel ou une station de travail pour commander toute la machine. L'environnement de programmation semble assez complet puisque le développement se fait entièrement en ANSI C comprenant quelques `#pragma`.

On constate donc que les développements matériels et logiciels sont très faibles comparés aux avantages de la méthode : un petit circuit intégré par carte pour réaliser le test, le contrôle système et les entrées-sorties primitives. Ce réseau correspond en fait au réseau de diagnostic de la CM-5 [?, Thi91].

Dans le cas de la version universitaire, c'est le bus global qui joue le rôle de réseau de diagnostic, comme c'est le cas sur POMP.

## 12.8 Extensibilité

Un autre terme à la mode mais aussi assez significatif est celui d'extensibilité<sup>24</sup> qui cautionne les possibilités d'évolution de la machine en nombre de processeurs.

L'extensibilité doit non seulement être possible physiquement, c'est-à-dire que les possibilités de rajouter des PEs et d'augmenter le réseau aient été prévues mais aussi que les performances de la machine complète évoluent correctement avec le nombre de PEs : cela ne sert à rien de rajouter des processeurs si l'ordinateur n'est pas globalement plus puissant.

L'avantage de choisir un réseau de type hypergrille de dimension  $d$  est que le degré des nœuds reste constant ( $2d$ ) quel que soit le nombre de PE  $N$ . Le diamètre évolue raisonnablement ( $\mathcal{O}(\sqrt[d]{N})$ ) ainsi que le débit crête ( $\mathcal{O}(dN)$ ).

De plus un tel réseau permet de rajouter des processeurs par incréments de  $\mathcal{O}(N^{\frac{d-1}{d}})$  seulement, à comparer au doublement successif de la taille dans le cas d'un hypercube.

On retrouve donc dans ce réseau les compromis exposés dans la section 9.2. Néanmoins, comme les performances du réseau augmentent moins vite avec  $N$  que la puissance brute de la machine, il est clair que les applications demandant moins de communications seront favorisées tout particulièrement sur une grosse machine avec beaucoup de processeurs.

## 12.9 Conclusion

Les machines SPMD, intermédiaires entre les machines SIMD et MIMD, semblent hériter des avantages des deux, en théorie, ce qui les rend particulièrement intéressantes.

Elles peuvent exécuter efficacement un programme basé sur modèle à parallélisme de données, par rapport à une machine MIMD, grâce aux opérateurs globaux tels que le matériel de synchronisation, le système de diffusion vers tous les processeurs ou les concentrations associatives.

Par rapport à une machine SIMD, les tests parallèles complexes sont exécutés de manière plus efficace puisqu'on permet dans une certaine mesure les dessynchronisations mais on peut toujours utiliser la machine en mode MIMD si on a un problème à résoudre ayant clairement un parallélisme de tâche par exemple ou encore exploiter du parallélisme de données intermédiaire tel que les `doacross` [Cyt86].

---

24. Terme pouvant traduire l'anglais « *scalability* » dont on trouve une excellente définition dans [?]: “*We thought hard about issues of scalability: making a machine whose size would be limited only by the dollars a customer could spend, not by any architectural or engineering constraint*” .

Dans une machine SIMD on optimise le cas pire de la synchronisation à chaque instruction alors que bien souvent il s'agit d'une attention pessimiste. Dans une machine MIMD à l'inverse on optimise un autre cas pire : celui où on n'a jamais besoin de se synchroniser.

Au niveau des bus des PES on peut évoquer 2 critères, la quantité et la qualité :

- la quantité est facilement perceptible : plus on envoie de données ou d'instructions et plus la machine sera efficace. On sera capable de traiter plus de données ou d'instructions ;
- la qualité est plus subtile mais en l'occurrence il s'agit des possibilités d'adressage. Si l'adressage est global, on pourra accéder aux données ou aux instructions de manière beaucoup plus souple et augmenter la qualité des traitements. La qualité de l'adressage des instructions est à mesurer dans le rapport débit d'adresses d'instructions sur débit d'instructions.

Une machine SPMD essaye de faire la part des choses en offrant la liberté de l'adressage au niveau des instructions et un système plus ou moins général et efficace de synchronisation, pouvant concerner dans le meilleur des cas des sous-ensembles indépendants de processeurs.

Le point qu'on a mentionné dans l'introduction est qu'on désire avoir une concentration en MFLOPS/dm<sup>3</sup> ou en MFLOPS par circuit intégré maximale tout en conservant un programmabilité de la machine raisonnable.

Actuellement, la concentration en MFLOPS semble plus élevée pour une machine SIMD à gros grain que pour une machine SIMD à grain fin, une machine SPMD ou MIMD, mais très prochainement les puissances volumiques iront en faveur du MIMD à cause des progrès en intégration microélectronique. Quelques concentrations approximatives estimées sur des performances sur nombres flottant en simple précision sont indiquées sur la table 12.4 à titre indicatif, sachant que les performances crêtes ne sont pas très représentatives, d'après [Sab92, ?, Cra91a, Cra91b, Int91c, Dig91] sans compter les dispositifs d'entrées-sorties. Les CM-2 et CM-200 dont les performances sont présentées ici n'utilisent que les coprocesseurs flottants et non pas les processeurs 1 bit (mode « *slice-wise* »). La machine étudiée dans ce chapitre, dénommée POMP+ sur le tableau suppose une intégration de 4 PES par carte, 8 cartes par rack, 4 racks par baie, 8 baies et en tout 1024 processeurs ALPHA à 200 MHz par exemple, le tout relié par un circuit de communication complexe factorisant la glue de chaque PE.

On constate qu'en prenant un processeur performant d'aujourd'hui pour réaliser une machine MIMD on dépasse la densité du SIMD même à gros grain mais au prix du développement d'un circuit d'interface et de communication complexe et coûteux selon les standards actuels.

Néanmoins on peut voir cette tendance à l'évolution du SIMD vers le SPMD ou le MIMD dans les supercalculateurs comme celle qui a fait évoluer le STAR 100 vers le CYBER 205 [Lin82] :

- dans la première machine les unités fonctionnelles du processeur se partageaient la plupart des portes logiques, c'est à dire qu'une porte servait souvent à plusieurs tâches suivant l'opération à faire. Cela permettait de réduire le nombre de circuits intégrés nécessaires pour faire la machine ;

TAB. 12.4 - Quelques comparaisons de puissance volumique crête.

<i>Machine</i>	<i>Puissance</i> (GFLOPS)	<i>Volume</i> (m <sup>3</sup> )	<i>Puissance volumique</i> (MFLOPS/dm <sup>3</sup> )	SIMD
CM-2	28	3,19	8,79	■
CM-200	40	3,19	12,56	■
CM-5	1000	2250	0,44	
CRAY Y-MP C90	16	4	4	□
CRAY Y-MP EL	0,532	1,5	0,35	□
DELTA	46	7,37	6,25	
MP-1	1,6	0,74	2,16	■
POMP	6,4	0,27	23,46	■
POMP+	205	6	34	

- dans la machine suivante, la microélectronique ayant progressé, il n'était plus nécessaire de faire ces partages de portes car dans chaque boîtier on pouvait mettre beaucoup plus de portes, augmentant le nombre de MFLOPS par boîtier.

Dans une machine SIMD, il y a encore un partage de la génération des instructions et dans notre cas il est beaucoup plus simple d'avoir une mémoire statique. Mais dans le cas d'une machine SPMD ou MIMD, il faut d'une part réaliser toute la partie décodage et récupération des instructions et d'autre part pouvoir initialiser celles-ci, à savoir charger le code dans la mémoire de chaque processeur, ce qui demande d'avoir des multiplexeurs supplémentaires sur le bus d'instructions ou le bus commun aux données et aux instructions, et une mémoire de code par exemple.

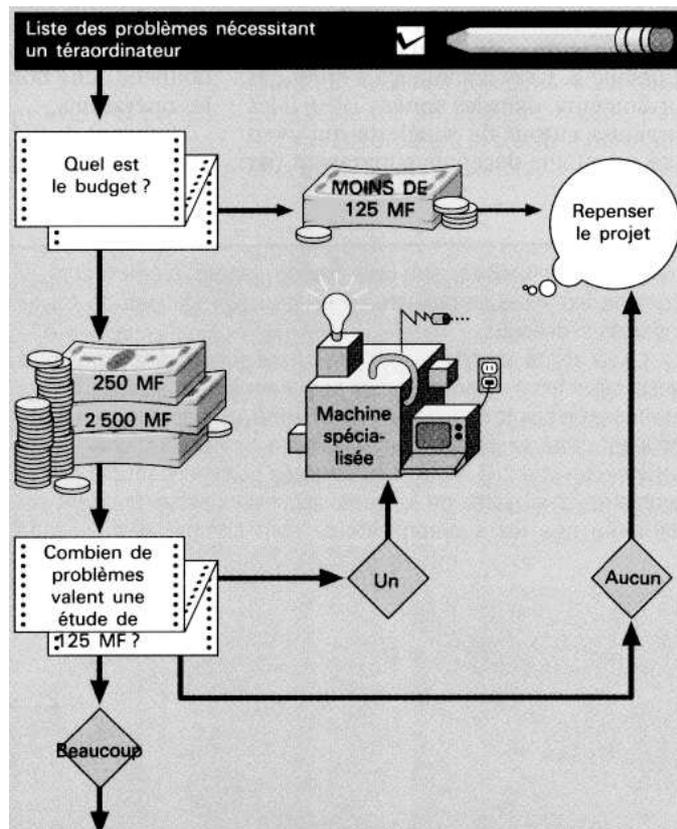
De même qu'au début de leur existence les machines massivement parallèles étaient SIMD sans adressage local aux données [Ung58, SBM62, Bat80a, ?], qu'ensuite elles ont acquis la qualité de pouvoir adresser librement leurs mémoires locales [BBK<sup>+</sup>68, BDR87, BDHR90, Bla90b], il semble raisonnable qu'elles puissent acquérir leur liberté totale d'expression apportée par l'avancée technologique. Il reste encore le problème des synchronisations arbitraires rapides à résoudre de manière simple pour les machines à venir.

Pour en revenir à un de nos objectifs initiaux sur le projet POMP, si notre but est de faire le maximum de MFLOPS/dm<sup>3</sup>, il est clair que pour avoir une machine SPMD performante on sera obligé de développer un gros circuit intégré de contrôle capable d'interfacer le PE et la mémoire avec le réseau de la machine, ce qui nous fait sortir du cadre de l'étude visée : concevoir une petite machine simple avec peu de moyens. En ce sens notre objectif de faire une petite machine avec peu de développements est atteint à travers le SIMD à gros grain développé dans cette thèse.

Si par contre le but est de construire une machine puissante dont la densité n'est pas préoccupante parce qu'elle ne doit pas loger sous un bureau, on peut construire une machine SPMD avec un DSP comme gestionnaire de réseau et de la logique discrète pour réaliser la machine en version « universitaire » ou bien encore une machine SPMD avec autant de TRANSPUTERS qu'il faut pour atteindre les performances, si c'est la simplicité qu'on recherche.

# Chapitre 13

## Conclusion



« *Tout le monde connaît l'utilité de l'utile mais rares sont ceux qui savent l'utilité de l'inutile.* »

Zhuangzi (*taoïste du IV<sup>ème</sup> siècle av. JC*),  
Le monde des hommes, *chapitre 4.*

**N**OUS sommes partis des hypothèses de départ qui étaient de faire un ordinateur accélérant les opérations classiques utilisées en synthèse d'image tout en possédant une taille et une consommation électrique raisonnables compatibles avec un bureau de chercheur<sup>1</sup>. En outre le développement de la machine devait être possible avec peu de moyens, humains et matériels, ce qui nous a amené à adopter une attitude pragmatique.

De bonnes performances en synthèse d'image impliquent une machine ayant des caractéristiques comparables aux supercalculateurs actuels :

- les algorithmes graphiques peuvent souvent être divisés en phases qui peuvent être pipelinées logiciellement sur un ordinateur standard ;
- ces algorithmes demandent de bonnes performances en nombres flottants y compris pour les divisions ;
- non spécialisation matérielle pour ne pas être restreint à un petit nombre d'algorithmes ;
- bonnes propriétés de programmation par opposition à la spécialisation.

Le refus d'une spécialisation et d'un développement coûteux à base de technologies « chaudes » nous a orienté vers le parallélisme massif permettant d'obtenir de bonnes performances par le biais de la répétition de petits processeurs de calcul plutôt que d'un gros processeur puissant. Cela a fourni l'argumentation de notre travail : POMP un Petit Ordinateur Massivement Parallèle.

Le choix du mode de parallélisme a été fonction du parallélisme à grain fin inhérent à de nombreux algorithmes graphiques ou scientifiques qui est le parallélisme de données — celui qui traite de collections d'éléments tels que vecteurs, matrices, pixels, lignes, etc. — en ce qui concerne le modèle de programmation et fonction de la performance volumique en calcul de la machine cible pour qu'elle puisse loger sous un bureau. Cela nous a mené tout naturellement à la définition d'une machine de type SIMD en même temps que l'autre partie de l'équipe définissait le langage POMPC, extension parallèle de C permettant d'exprimer ce parallélisme suivant une sémantique synchrone. Ce langage a été de conserve adapté à d'autres machines, parallèles ou non.

La largeur des processeurs élémentaires de la machine a été choisie égale à celle des données traitées classiquement, à savoir 32 et 64 bits, permettant ainsi d'exploiter implicitement ce parallélisme facile en ciblant un parallélisme moyen en nombre de processeurs mais massif en nombre de bits de calcul efficace. On peut actuellement

---

1. Que nous avons supposé parisien afin d'augmenter les contraintes d'encombrement stérique.

observer le triomphe des architectures RISC qui deviennent de plus en plus performants (superscalaires, superpipelinés, etc.) pour les machines séquentielles et le développement de machines parallèles à gros grain basées sur ces processeurs. La machine que nous proposons va dans cette optique en utilisant en plus l'aspect SIMD qui pose son originalité.

Le couplage de la machine est faible, c'est à dire que les processeurs communiquent par un réseau et ne partagent pas de mémoire commune. Cela simplifie l'architecture tout en permettant une meilleure exploitation logicielle de la localité des données.

Malheureusement, bien que conceptuellement plus simples, les machines SIMD s'accommode mal des processeurs commerciaux et nécessitent le développement de processeurs spécifiques, source de travaux supplémentaires et de nombreux déboires potentiels surtout dans le cas de processeurs larges. En outre il faut développer toute une partie contrôle de la machine qui peut ne pas exister dans le cas du parallélisme MIMD.

De plus ce développement de tout le matériel *ex nihilo* implique la création de tout l'environnement logiciel *ad hoc*, ce qui rend finalement le concept bien compliqué à mettre en œuvre.

Contrairement à d'autres approches passées, l'utilisation de processeurs élémentaires du commerce ne table pas sur un système global de synchronisation après chaque instruction qui ralentirait la machine mais sur l'utilisation de processeurs RISC dont le fonctionnement déterministe est assuré par un algorithme simple de planification statique du code parallèle et du code scalaire après compilation avec les outils standard fournis avec le processeur.

Différentes méthodes de contrôle de flot SIMD ont été développées afin de permettre une exploitation efficace de notre machine même dans le cas de code possédant beaucoup d'alternatives parallèles. Certaines de ces méthodes sont utilisables telles quelles sur d'autres architectures, pas seulement SIMD.

La stratégie de contrôle est celle du couplage fort VLIW entre le processeur scalaire et les processeurs élémentaires : il est à la fois simple et efficace et permet un comportement déterministe de la machine, tout en économisant le développement d'un séquenceur de code parallèle. Le processeur scalaire est en fait le même que celui des processeurs élémentaires car cela nous permet de simplifier la conception et l'intégration des 2 parties. La majeure partie de la gestion système est reportée sur un hôte UNIX ce qui simplifie encore une fois le développement, y compris celui de l'environnement de mise au point.

Ce type de couplage entre processeur scalaire et processeurs élémentaires nous permet d'ailleurs d'exhiber des cas de superlinéarité, c'est-à-dire des cas où une machine à  $n$  processeurs accélère d'un facteur strictement supérieur à  $n$  certains calculs.

De manière indépendante il a fallu développer un nouveau type de réseau simple et efficace dans notre cas où on considérait une machine avec un nombre moyen et borné de processeurs. Il est issu du constat classique qu'un programme parallèle nécessite souvent des phases de communications régulières de type grille et moins souvent des phases de communications irrégulières. Il s'agit donc d'accélérer dans la mesure du possible les premières communications sans diminuer les secondes tout en simplifiant au maximum le réseau et sa gestion.

Le point de départ est un réseau multi-étage de type hypercube en accord avec le synchronisme de la machine et la simplicité du routeur nécessaire pour les communications aléatoires. Afin de pouvoir le partitionner en un circuit par processeur et de

pouvoir utiliser le réseau statique sous-jacent le réseau dynamique est multiplié. Le fait d'avoir plusieurs réseaux dynamiques ne pose pas de problème de performance car on suppose que le degré de parallélisme des applications est suffisant pour permettre un bon usage du réseau. Des doublements assez semblables avaient déjà été étudiés pour la tolérance aux pannes mais nous l'exploitons ici dans un autre but : l'optimisation d'une classe de communications régulières (décalages, communications sur hypercube, battage parfait). En outre il s'avère que ce réseau est très simple au point de pouvoir loger dans un circuit de logique reprogrammable, ce qui rajoute 2 intérêts :

- 1° aucun circuit spécifique n'est à développer pour notre machine et notre
- 2° la machine est dotée d'une possibilité de reconfiguration matérielle si besoin est pour lui rajouter d'autres fonctionnalités.

La méthodologie développée et utilisée dans cette thèse nous permet donc de nous détacher des contingences matérielles fines de l'architecture des ordinateurs parallèles telles que la conception des processeurs élémentaires, d'un processeur scalaire, d'un contrôleur, d'un séquenceur, etc. en récupérant et en exploitant le maximum de ressources déjà existantes : processeurs, mémoires, compilateurs commerciaux, bibliothèques de fonctions système et mathématiques, etc. Cette approche rationnelle pour la conception d'une classe de machines parallèles nous a permis d'éviter le développement de circuits intégrés spécifiques ou d'un compilateur ciblé.

Néanmoins cette approche pragmatique ne sacrifie pas les performances de la machine pour des applications scientifiques nécessitant de gros calculs en nombres à virgule flottante puisque les performances prévues sont meilleures en fait que la première étude de la machine qui consistait à concevoir aussi des processeurs spécifiques.

Enfin, à titre de comparaison, nous avons développé une version MIMD modernisée de POMP dans une technologie actuelle. Un modèle d'exécution SPMD est intéressant car il permet de mieux exploiter d'autres types de parallélisme bien sûr mais aussi le contrôle de flot SIMD. l'étude inclue le développement d'une méthode de partitionnement de la machine et du réseau, un système de gestion mémoire, un mécanisme de synchronisation globale optimisée et un système de mise au point simple de la machine. Les conclusions sont qu'il semble encore difficile de faire une machine parallèle SPMD offrant la même densité qu'une machine SIMD comme POMP, mais qu'il est prévisible qu'à moyen terme ce sera le cas. De toute manière, le point noir de l'approche actuelle SPMD est qu'elle nécessite le développement d'un circuit intégré complexe de communication pour atteindre ces concentrations en calcul, développement allant totalement à l'encontre de nos hypothèses initiales de simplicité.

Peu d'estimations sérieuses de performances sont données dans cette thèse car le fait d'avoir un compilateur pour d'autres machines assez similaires nous font penser que les applications que nous avons écrites, telles que celles données en annexe ??, auront des performances en accord sur notre machine avec celle que nous avons mesurées sur d'autres machines. D'autre part, une estimation fine des performances est très difficile et demande un effort que nous ne pouvions fournir, vue la taille de notre équipe.

Enfin la conclusion des développements matériels de cette thèse est qu'il devient de plus en plus difficile de réaliser une machine en dehors d'un environnement industriel motivé et financier adéquat. En particulier le choix initial d'une technologie de construction « wrappée », même pour un prototype réduit, nous apparaît *a posteriori*

comme très mauvais vues les raideurs des fronts et les fréquences d'horloges utilisées par les processeurs de la machine. Cela a entraîné un retard considérable dans le projet et abouti à un prototype incomplet peu fiable. Malheureusement, c'était le seul choix possible étant donnés les moyens à notre disposition pour essayer de faire un prototype. Il semble que l'époque où on pouvait construire une machine parallèle à partir d'une « manipulation de coin de table » avec peu de moyens soit dépassée.

On peut essayer de développer des coopérations avec des industriels afin d'associer les compétences, mais cela est difficile étant donné les intérêts respectifs à plus long terme et à court terme des parties concernées. De plus, il faut probablement plus qu'une collaboration étroite avec un industriel, peut-être créer une entreprise propre pour éviter de court-circuiter un projet interne qui démotiverait les chercheurs et ingénieurs de l'entreprise ou ne serait pas dans les visions commerciales de la direction<sup>2</sup> et d'autre part avoir une motivation suffisante de ses membres.

Certains pourront dire qu'il n'était pas nécessaire d'essayer de réaliser un prototype, voire même d'imaginer une hypothétique réalisation et qu'il suffit d'avancer des arguments bien étayés, et donc de se contenter des développements logiciels, voire conceptuels. On peut répondre plusieurs choses :

- avoir un prototype montre qu'on avait une architecture réalisable et pas seulement une « machine papier » ;
- une machine qui fonctionne est un argument de poids pour avoir des financements et/ou convaincre un industriel ;
- en réalisant une machine on apprend énormément de choses qu'on ne trouve pas dans la littérature ;
- il est nécessaire d'acquérir cette expérience si on veut l'enseigner après ou le réinvestir, que ce soit en l'exportant en milieu industriel ou dans un autre projet.

En ce sens il nous semble important, voire vital<sup>3</sup>, de continuer les recherches en ce domaine, même s'il est parfois ingrat sur le plan de la théorie la plus pure...




---

2. Par exemple une entreprise plutôt spécialisée dans le matériel militaire ou de gestion...

3. En constatant que — malheureusement — les superordinateurs sont devenus aujourd'hui une arme économique d'une toute autre envergure puisqu'ils permettent à celui qui les possède et les maîtrise d'avoir une avance considérable dans des domaines de conception de nouveaux produits, par exemple.



Chapitre **13**

**Bibliographie**



- [AB86] M. AUGUIN et F. BOERI. The OPSILA Computer. Dans INRIA, éditeur, *Parallel Algorithms & Architectures*, pages 143–153. North-Holland, 1986. Cité pages 71, 107, 149.
- [ABD90] Michel AUGUIN, Fernand BOERI, et Jean Paul DALBAN. Synthèse du projet Opsila. *Techniques et Sciences Informatiques*, 09(02):79–98, 1990. Cité page 94.
- [ABM88] Brian APGAR, Bret BERSACK, et Abraham MAMMEN. A Display System for the Stellar<sup>TM</sup> Graphics Supercomputer Model GS1000<sup>TM</sup>. Dans *Computer Graphics (SIGGRAPH '88)*, pages 255–262. Association for Computing Machinery, août 1988. Volume 22, Number 4. Cité page 22.
- [ABT82] Robert G. ARNOLD, Robert O. BERG, et James W. THOMAS. A Modular Approach to Real-Time Supersystems. *IEEE Transactions on Computers*, C-31(5):385–398, mai 1982. Cité page 10.
- [ACF92] Selim G. AKL, Michel COSNARD, et Alfonso G. FERREIRA. Data-Movement-Intensive Problems: Two Folk Theorems in Parallel Computation Revisited. *Theoretical Computer Science*, 95(2), 1992. Cité pages 10, 258, 259.
- [Adv88a] Advanced Micro Devices. *32 Bit Microprogrammable Products Am29C300/29300*, 1988. Cité pages 108, 125.
- [Adv88b] Advanced Micro Devices. *Am29000 32-Bit Streamlined Instruction Processor*, 1988. Cité page 109.
- [Adv88c] Advanced Micro Devices. *Am29C111, CMOS 16 bits Microprogram sequencer*, janvier 1988. Cité page 125.
- [Adv90a] Advanced Micro Devices. *Am29050 Streamlined Instruction Processor*, advance information 15039 rev. a/0 édition, septembre 1990. Cité page 109.
- [Adv90b] Advanced Micro Devices. *Dynamic Memory Design*, data book/handbook édition, 1990. Cité page 271.
- [AHK87] Sheldon B. AKERS, Dov HAREL, et Balakrishnan KRISHNAMURHY. The Star Graph: An Attractive Alternative to the  $n$ -Cube. Dans *International Conference on Parallel Processing*, pages 393–400. The Institute of Electrical and Electronics Engineers, Inc., Academic Press, 1987. Cité page 193.
- [AJ88] Kurt AKELEY et Tom JERMOLUK. High-Performance Polygon Rendering. Dans *Computer Graphics (SIGGRAPH '88)*, volume 22(4), pages 239–246. ACM, août 1988. Cité pages 15, 18.
- [All91] Alliant Computer System Corporation. *The Campus/800 System from Alliant*, 1991. BR05-7915M. Cité page 37.
- [ALS90] Eugene ALBERT, Joan D. LUKAS, et Guy L. STEELE JR.. Data Parallel Computers and the FORALL Statement. Dans *The 3<sup>rd</sup> Symposium on the Frontiers of Massively Parallel Computation*, pages 390–396. The Institute of Electrical and Electronics Engineers, Inc., octobre 1990. Cité page 90.
- [Amd67] Gene M. AMDAHL. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. Dans AFIPS, éditeur, *Proceedings of the Spring Joint Computer Conference*, pages 483–485, 1967. Cité pages 6, 28, 30, 119, 128, 254.
- [AMT89] AMT — Active Memory Technology Ltd. *Massively Parallel Computers — DAP Series Specification*, 1989. Cité page 94.

- [AN90] Alexander AIKEN et Alexandru NICOLAU. A Realistic Resource-Constrained Software Pipelining Algorithm . Dans *Third Workshop on Programming Languages and Compilers for Parallel Computing*, pages 1–17. Preliminary Proceedings to be published by Pitman/MIT Press, août 1990. Cité page 157.
- [AP87] Todd R. ALLEN et David A. PADUA. Debugging Fortran on a Shared Memory Machine. Dans *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727. The Pennsylvania State University Press, 1987. Cité page 45.
- [AP91a] Seth ABRAHAM et Krishnan PADMANABHAN. MVAMIN: Mean Value Analysis Algorithms for Multistage Interconnection Networks. *Journal of Parallel and Distributed Computing*, 12(3):237–248, juillet 1991. Cité page 39.
- [AP91b] Seth ABRAHAM et Krishnan PADMANABHAN. Performance of Multicomputer Networks under Pin-out Constraints. *Journal of Parallel and Distributed Computing*, 12(3):237–248, juillet 1991. Cité pages 187, 191, 204.
- [Ata89] Abdelghani ATAMENIA. *Architectures Cellulaires pour la Synthèse d'Images*. Thèse, Laboratoire d'Informatique Fondamentale de Lille — Université des Sciences et Techniques de Lille Flandres Artois, juin 1989. Cité page 22.
- [AU77] Alfred V. AHO et Jeffrey D. ULLMAN. *Principles of Compiler Design*. Addison-Wesley Publishing Company, 1977. Cité pages 168, 169.
- [Aug85] Michel AUGUIN. *Etude et Réalisation de Structures de Calculs Parallèles — Application aux Méthodes Numériques*. Thèse d'état, Université de Nice, mai 1985. Cité pages 40, 94, 122, 141.
- [AUG89] AUGAT. *EHTM Electronically Invisible Interconnect*, 1989. Cité page 281.
- [Bac78] John BACKUS. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–641, août 1978. Cité pages 3, 6, 17, 111.
- [Bar90] Anthony C. BARKANS. High Speed High Quality Antialiased Vector Generation. Dans *Computer Graphics (SIGGRAPH '90)*, pages 319–326. Association for Computing Machinery, août 1990. Volume 24, Number 4. Cité page 15.
- [Bat68] K. E. BATCHER. Sorting networks and their applications. Dans *The Proceedings of AFIPS 1968 SJCC*, pages 307–314. AFIPS Press, 1968. Cité page 199.
- [Bat76] Kenneth E. BATCHER. The Flip Network in STARAN. Dans The Institute of ELECTRICAL et Inc. ELECTRONICS ENGINEERS, éditeurs, *The Proceedings of the 1976 International Conference on Parallel Processing*, pages 65–71, 1976. Cité page 197.
- [Bat80a] Kenneth E. BATCHER. Architecture of a Massively Parallel Processor. Dans *SIGARCH 80*, pages 168–173. The Institute of Electrical and Electronics Engineers, Inc., 1980. Cité pages 101, 123, 133, 303.
- [Bat80b] Kenneth E. BATCHER. Design of a Massively Parallel Processor. *IEEE Transactions on Computers*, C-29(9):1–9, septembre 1980. Cité pages 123, 133.
- [Bat82] Kenneth E. BATCHER. Bit-Serial Parallel Processing Systems. *IEEE Transactions on Computers*, C-31(5):377–384, mai 1982. Voir [GMSF87a]. Cité pages 119, 123.
- [BBES92] Ingo BARTH, Thomas BRÄUNL, Stephan ENGELHARDT, et Frank SEMBACH. *Parallaxis Version 2 User Manual. Second Edition – Release 2.1x 2/92*, Univesität Stuttgart Fakultät Informatik, février 1992. Récupérable par `ftp anonymous` sur la machine `ftp.informatik.uni-stuttgart.de`. Cité page 78.

- [BBJ<sup>+</sup>62] J. R. BALL, R. C. BOLLINGER, T. A. JEEVES, R. C. MCREYNOLDS, et D. H. SHAFFER. On the Use of the SOLOMON Parallel-Processing Computer. Dans *Proceedings of the Fall 1962 Eastern Joint Computer Conference*, pages 137–146, décembre 1962. Cité pages 101, 134.
- [BBK<sup>+</sup>68] George H. BARNES, Richard M. BROWN, Maso KATO, David J. KUCK, Daniel L. SLOTNICK, et Richard A. STOKES. The ILLIAC IV Computer. *IEEE Transactions on Computers*, C-17(8):746–757, août 1968. Cité pages 29, 40, 70, 94, 107, 120, 122, 141, 303.
- [BC63] David R. BENNION et Hewitt D. CRANE. *All-Magnetic Circuit Techniques*, volume 4, pages 53–133. Academic Press, 1963. Cité page 93.
- [BCW89] François BODIN, François CHAROT, et Charles WAGNER. Microcode Optimization for the PCS Processor. Rapport Technique 473, IRISA, mai 1989. Cité pages 33, 172.
- [BDHR90] Donald W. BLEVINS, Edward W. DAVIS, Robert A. HEATON, et John H. REIF. BLITZEN: A Highly Integrated Massively Parallel Machine. *Journal of Parallel and Distributed Computing*, 8(2):150–160, février 1990. Cité page 303.
- [BDR87] Donald W. BLEVINS, Edward W. DAVIS, et John H. REIF. Processing Element and Custom Chip Architecture for the BLITZEN Massively Parallel Processor. Rapport Technique TR87-22 Révision 1, Microelectronics Center of NC, 1987. Cité pages 94, 101, 148, 303.
- [BDW85] John BEETEM, Monty DENNEAU, et Don WEINGARTEN. The GF11 Supercomputer. Dans *SIGARCH 85*, pages 108–115. The Institute of Electrical and Electronics Engineers, Inc., 1985. Cité pages 63, 146, 148, 199.
- [Bel89] Belden — Wire and Cable. *Master Catalog*, 1989. pages 172–174. Cité page 281.
- [Ben62] V. E. BENEŠ. On a Class of Multistage Interconnection Networks. *The Bell System Technical Journal*, XLI(5):1481–1492, septembre 1962. Cité page 198.
- [Ber91] Robert BERNECKY. Fortran 90 Arrays. *ACM SIGPLAN Notices*, 26(2):83–97, février 1991. Cité pages 55, 72.
- [BF88] J.-C. BERMOND et J.-M. FOURNEAU. Independent connections: an easy characterization of baseline-equivalent multistage interconnection networks. Dans *International Conference on Parallel Processing*, pages 187–190. The Institute of Electrical and Electronics Engineers, Inc., Academic Press, 1988. Cité page 197.
- [BIT89] Bipolar Integrated Technology, Inc. *B2130/B3130/B4130 Single Chip Floating Point Processors*, advance information édition, décembre 1989. Cité page 108.
- [BJR88] Meera BALAKRISHNAN, Rajiv JAIN, et C. S. RAGHAVENDRA. On Array Storage For Conflict-Free Memory Access For Parallel Processors. Dans *Proceedings of the 1988 International Conference on Parallel Processing*, pages 103–107. The Pennsylvania State University Press, 1988. Cité page 39.
- [Bla90a] Tom BLANK. The Design of the MasPar MP-1, A Cost-Effective Massively Parallel Computer. Dans IEEE, éditeur, *IEEE Comcon Spring 1990*, février 1990. Cité pages 94, 95, 113, 122.
- [Bla90b] Tom BLANK. The MasPar MP-1 Architecture. Dans IEEE, éditeur, *IEEE Comcon Spring 1990*, février 1990. Cité pages 36, 40, 94, 95, 113, 122, 303.
- [Ble89a] Guy E. BLELLOCH. *Scan Primitives and Parallel Vector Models*. PhD thesis, Laboratory for Computer Science — Massachusetts Institute of Technology, octobre 1989. MIT/LCS/TR-463. Cité pages 50, 60, 96, 292.

- [Ble89b] Guy E. BLELLOCH. Scans as Primitive Parallel Operation. *IEEE Transactions on Computers*, 38(11):1526–1538, novembre 1989. Cité pages 60,96.
- [Bod90] François BODIN. Data Structure Analysis in C programs. Dans *International Workshop on Compilers for Parallel Computers*, pages 11–23. ENSMP CAI - UPMC MASI, décembre 1990. Cité page 90.
- [Bou91] Luc BOUGÉ. Discipline de Programmation Synchronisée pour les Architectures à Mémoire Distribuée. Journée d'Etude de Programmation pour les Machines Parallèles, PRC C<sup>3</sup>, 24 juin 1991. Cité page 43.
- [BP89] J-C. BERMOND et C. PEYRAT. De Bruijn and Kautz Network: a competitor for the hypercube. Dans INRIA F.ANDRÉ, J.P. Verjus, éditeur, *Hypercube and Distributed Computers*, pages 279–293, 1989. Cité page 192.
- [BP90] Carl J. BECKMANN et Constantine D. POLYCHRONOPOULOS. Fast Barrier Synchronization Hardware. Dans *Proceedings of Supercomputing 90*, pages 180–189. IEEE, novembre 1990. Cité page 287.
- [BQW91] Hester BAKEWELL, Donna J. QUAMMEN, et Pearl Y. WANG. Mapping Concurrent Programs to VLIW Processors. Dans *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26(7) de *SIGPLAN Notices*, pages 21–27, juillet 1991. Cité page 156.
- [BR88] Rajendra BOPANA et C. S. RAGHAVENDRA. On Self Routing in Beneš and Shuffle Exchange Networks. Dans *Proceedings of the 1988 International Conference on Parallel Processing*, pages 196–200. The Pennsylvania State University Press, 1988. Cité page 199.
- [Brä89] Thomas BRÄUNL. Structured SIMD Programming in Parallaxis. *Structured Programming*, 10(3):121–132, juillet 1989. Cité page 78.
- [Bre74] Richard P. BRENT. The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM*, 21(2):201–206, avril 1974. Cité pages 93,254,256,257,287.
- [BS90] Guy E. BLELLOCH et Gary W. SABOT. Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, février 1990. Cité pages 50,133.
- [Bul57] Compagnie des Machines Bull. *Gamma 60*, 1957. 105.150. Cité pages VI,31,95.
- [Bul60] Compagnie des Machines Bull. *Schémas du Calculateur Logique — Gamma 60*, septembre 1960. 105.148. Cité pages VI,31.
- [Bul92] Réseaux et Systèmes d'Information — Bull. *Naissance d'un Ordinateur (Bull DPS 7000) à livre ouvert*, mars 1992. Cité page 37.
- [BW86] Gary BISHOP et David M. WEIMER. Fast Phong Shading. Dans *Computer Graphics (SIGGRAPH '86)*, volume 20(4), pages 103–106. Association for Computing Machinery, août 1986. Cité page 22.
- [Cal91] D. CALLAHAN. Recognizing and Parallelizing Bounded Recurrences. Dans *Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 169–185. Springer-Verlag, août 1991. LNCS 589. Cité pages 7,96.
- [Cas85] Serge CASTAN. Architectures adaptées au traitement d'images. *Techniques et Sciences Informatiques*, 04(05):431–445, 1985. Cité page 94.
- [CC92] Patrick CASSAM-CHENAI". *Algèbre Fermionique et Chimie Quantique*. Thèse, Université Pierre et Marie Curie — PARIS VI, mai 1992. Cité page VI.

- [CCD<sup>+</sup>92] Lowell CAMPBELL, Gunnar E. CARLSSON, Michael J. DINNEEN, Vance FABER, Michael R. FELLOWS, Michael A. LANGSTON, James W. MOORE, Andrew P. MULLHAUPT, et Harlan B. SEXTON. Small Diameter Symmetric Networks from Linear Groups. *IEEE Transactions on Computers*, 40(2):218–220, février 1992. Cité page 193.
- [CCYHJ<sup>+</sup>85] Robert P. COLWELL, III CHARLES Y. HITCHCOCK, E. Douglas JENSEN, H. M. Brinkley SPRUNT, et Charles P. KOLLAR. Instruction Sets and Beyond: Computers, Complexity, and Controversy. *Computer*, 18(9):8–19, septembre 1985. Voir [GMSF87b]. Cité page 108.
- [Cel91] Cellule d'Évaluation des Microprocesseurs Rapides de l'IRISA. *Journée Microprocesseurs Rapides : les Microprocesseurs RISC de 2<sup>ème</sup> Génération — Architecture et Compilation*, décembre 1991. Cité page 24.
- [Che83] Steve CHEN. Large-scale and High-Speed Multiprocessor System for Scientific Applications — CRAY X-MP-2 Series. Dans *Proceedings of NATO Advanced Research Workshop on High Speed Computing*, pages 46–58. IEEE Computer Society Press, juin 1983. Voir [Hwa84]. Cité pages 7, 36.
- [CK91] Elizabeth CORCORAN et Tom KOPPEL. Trends in Computing: Calculating Reality. *Scientific American*, janvier 1991. Cité pages 3, 25.
- [CK92] Andrew A. CHIEN et Jae H. KIM. Planar-Adaptative Routing: Low-cost Adaptative Networks for Multiprocessors. *The 19<sup>th</sup> Annual International Symposium on Computer Architecture Conference Proceedings*, 20(2):268–277, mai 1992. Cité page 204.
- [CKM91] Christophe CHAILLOU, Sylvain KARPF, et Michel MÉRIAUX. Une Architecture Massivement Parallèle pour la Synthèse d'Images en Temps Réel: la Machine I.M.O.G.E.N.E.. Dans *Troisième Symposium sur les Architectures Nouvelles de Machines — Recueil des Communications*, pages 55–74. PRC ANM – CNRS, juin 1991. Cité page 22.
- [Cla82] James H. CLARK. The Geometry Engine: A VLSI Geometry System for Graphics. Dans *Computer Graphics (SIGGRAPH '82)*, volume 15(3), pages 127–133. Association for Computing Machinery, juillet 1982. Cité page 18.
- [Clo53] Charles CLOS. A Study of Non-Blocking Switching Networks. *The Bell System Technical Journal*, XXXII:406–424, mars 1953. Cité page 197.
- [CMZ92] B. CHAPMAN, P. MEHROTRA, et H. ZIMA. Programming in Vienna Fortran. Rapport Technique ACPC/TR 92-3, ACPC — Austrian Center for Parallel Computation, mars 1992. Cité page 69.
- [Col90] Robert COLLINS. *CM++ Manual*. Department of Computer Science, University of California Los Angeles, avril 1990. Récupérable par ftp `anonymous` sur la machine `polaris.cognet.ucla.edu`. Cité page 78.
- [Cor91] Elizabeth CORCORAN. Les Superordinateurs. *Pour la Science*, (161), mars 1991. Cité pages VII, 25.
- [Cra91a] Cray Research, Inc. *The CRAY Y-MP C90 Supercomputer System*, 1991. MCPB-104-1191. Cité pages 36, 39, 110, 302.
- [Cra91b] Cray Research, Inc. *The CRAY Y-MP EL Supercomputer System*, 1991. MCPB-102-0991. Cité page 302.
- [Cue92] Jean-Christophe CUENOD. Senior Engineer, DEC, janvier 1992. Conversation privée. Cité page 242.

- [Cur63] William A. CURTIN. *Multiple Computer Systems*, volume 4, pages 245–230. Academic Press, 1963. Cité page 30.
- [Cyp89] Cypress Semiconductor. *CMOS/BICMOS Data Book*, February 1989. Chapter 6: Introduction to RISC (CY7C601). Cité page 109.
- [Cyp90a] Cypress Semiconductor. *BiCMOS/CMOS Data Book*, mars 1990. Cité page 108.
- [Cyp90b] Cypress Semiconductor. *SPARC RISC User's Guide*, février 1990. Cité page 152.
- [Cyp91] Cypress Semiconductor. *The Time for Multiprocessing is Now*, juillet 1991. Cité page 268.
- [Cyt86] Ron CYTRON. Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract). Dans *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844. IEEE, août 1986. Cité page 301.
- [Dal92] William J. DALLY. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, mars 1992. Cité page 201.
- [dD91] Benoît Dupont de DINECHIN. *Parallélisation de Code et Architectures Superscalaires à Contrôle Statique*. Thèse, Université Pierre et Marie Curie — PARIS VI, décembre 1991. Cité pages 38, 39, 169, 293.
- [DGNP88] F. DAREMA, D.A. GEORGE, V.A. NORTON, et G.F. PFISTER. A Single Program-Multiple-Data Computational Model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11–24, avril 1988. Cité pages 65, 265.
- [Dic92] A. DICKINSON. Future Supercomputing Requirements for Weather Forecasting and Climate Prediction. Dans *Supercomputing Europe '92*, pages 50–52, février 1992. Cité page 65.
- [Dig91] Digital Equipment Corporation. *DECmpp 12000 — Massively Parallel Processing Systems*, 1991. Cité page 302.
- [Dig92a] Digital Equipment Corporation. *DECChip<sup>TM</sup> 21064-AA RISC Microprocessor Preliminary Data Sheet*, avril 1992. Récupérable par `ftp anonymous` sur la machine `gatekeeper.dec.com`. Cité pages 39, 109, 111, 148, 275, 280.
- [Dig92b] Digital Equipment Corporation. *Introduction to Designing a System with the DECChip<sup>TM</sup> 21064-AA Microprocessor*, avril 1992. Revision 1.0. Récupérable par `ftp anonymous` sur la machine `gatekeeper.dec.com`. Cité page 275.
- [DKMS90] Jack DONGARRA, Alan H. KARP, Ken MIURA, et Horst SIMON. 1990 Gordon Bell Prize Winners. *IEEE Software*, 8(3):92–102, mai 1990. Cité pages 7, 68, 95.
- [DLSM81] Scott DAVIDSON, David LANDSKOV, Bruce D. SHRIVER, et Patrick W. MALLETT. Some Experiments in Local Microcode Compaction for Horizontal Machines. *IEEE Transactions on Computers*, C-30(7):460–477, juillet 1981. Cité pages 169, 172, 175.
- [dM88] Patrice Ossona de MENDEZ. Conception d'un Unité d'Arithmétique et Logique SIMD optimisée pour le calcul flottant. Diplôme d'étude approfondie, Paris XI, 1988. Cité page 112.
- [Dou89] César DOUADY. *Visualisation graphique et architecture parallèle: Conception et réalisation*. Thèse, Paris XI, mai 1989. Cité pages 8, 18, 94, 99, 101, 104, 105, 111, 112, 114, 162, 163, 275.

- [DOZ90] Henry G. DIETZ, Matthew T. O'KEEFE, et Abderrazek ZAAFRANI. An Introduction to Static Scheduling for MIMD Architectures. Dans *Third Workshop on Programming Languages and Compilers for Parallel Computing*, pages 1–25. Preliminary Proceedings to be published by Pitman/MIT Press, août 1990. Cité page 287.
- [DS86] William J. DALY et Charles L. SEITZ. The Torus Routing Chip. *Distributed Computing*, 1(4):187–196, 1986. Cité page 200.
- [DS87] William J. DALY et Charles L. SEITZ. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, mai 1987. Cité pages 200, 202, 203.
- [DuP90] Du Pont Electronics. *METRAL<sup>TM</sup> Connectors*, 1990. Cité page 281.
- [Ech91] Echelon Corporation. *LONTALK Protocol — LONWORKS Engineering Bulletin*, septembre 1991. Cité page 300.
- [Ech92] Echelon Corporation. *NEURON 3120 CHIP and NEURON 3150 CHIP — Advance Information*, janvier 1992. Cité page 300.
- [ED85] Phillip EIN-DOR. Grosch's Law Re-Revisited: CPU Power and the Cost of Computation. *Communications of the ACM*, 28(2):142–151, février 1985. Cité page 103.
- [Enc91] Encore Computer Corporation. *Infinity 90 Series — the Mainframe Alternative*, 1991. 316-0017 15M-PP-0891. Cité pages 37, 196.
- [FCS88] Samuel A. FINEBERG, Thomas L. CASAVANT, et Thomas SCHWEDERSKI. Non-Deterministic Instruction Time Experiments on the PASM System Prototype. Dans *Proceedings of the 1988 International Conference on Parallel Processing*, pages 281–285. The Pennsylvania State University Press, 1988. Cité pages 260, 262.
- [Fen74] Tse-Yun FENG. Data Manipulating Functions in Parallel Processors and Their Implementations. *IEEE Transactions on Computers*, C-23(3):309–318, mars 1974. Cité pages 199, 291.
- [Fen81] Tse Yun FENG. A Survey of Interconnection Networks. *Computer*, 14(12):12–27, décembre 1981. The Institute of Electrical and Electronics Engineers, Inc. Cité page 184.
- [FGP91] J.-M. FILLOQUE, E. GAUTRIN, et B. POTTIER. Efficient Global Computations on a Processor Network with Programmable Logic. Dans *PARLE '91 Parallel Architectures and Languages Europe*, volume 505(I), pages 69–82. Lecture Notes in Computer Science, Springer-Verlag, juin 1991. Cité page 165.
- [FGPS91] S. A. FELPERIN, L. GRAVANO, G. D. PIFARRÉ, et J. L. C. SANZ. Fully-Adaptative Routing: Packet Switching Performance and Wormhole Algorithms. Dans *Proceedings Supercomputing '91*, pages 654–663. The Institute of Electrical and Electronics Engineers, Inc., novembre 1991. Cité page 203.
- [FHK<sup>+</sup>92] Geoffrey FOX, Seema HIRANANDANI, Ken KENNEDY, Charles KOELBEL, Uli KREMER, Chau-Wen TSENG, et Min-You WU. Fortran D Language Specification. Rapport Technique, Department of Computer Science, Rice University, Houston, USA, janvier 1992. Récupérable par `ftp anonymous` sur la machine `cs.rice.edu` dans le fichier `public/HPFF/fd.ps.Z`. Cité page 68.
- [Fil91a] J.-M. FILLOQUE. Synthèse de Contrôleurs sur une Architecture à Couche Logique Reconfigurable. Dans *Actes des 1<sup>ères</sup> Journées ArMen*, pages 89–122. Laboratoire d'Informatique de Brest, UBO-ENSTBr, juin 1991. Cité pages 287, 288.

- [Fil91b] Jean-Marie FILLOQUE, éditeur. *Actes des 1<sup>ères</sup> Journées ArMen*. Laboratoire d'Informatique de Brest, UBO-ENSTBr, ENSTBr, juin 1991. Cité page 218.
- [Fil92] Jean-Marie FILLOQUE. *Synchronisation répartie sur une machine parallèle à couche logique reconfigurable*. Thèse, Université de Rennes I, novembre 1992. Cité pages 287, 288, 294, 295, 296.
- [Fis81] Joseph A. FISHER. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478–490, juillet 1981. Cité pages 169, 175.
- [FJL85] J.M. FRAILONG, W. JALBY, et Jacques LENFANT. XOR-Schemes: a Flexible Data Organization in Parallel Memories. Dans INRIA, éditeur, *Cours et séminaires, conférences, Saint-Cyprien*, pages 33–45, octobre 1985. Cité page 39.
- [Fla90] Peter FLANDERS. Virtual Systems Architecture on the AMT DAP. Dans SPRINGER-VERLAG, éditeur, *CONPAR 90 - VAPP IV*, pages 774–785, September 1990. Cité page 163.
- [FLW86] V. FABER, Olaf M. LUBECK, et Andrew B. WHITE, Jr. Superlinear Speedup of an Efficient Sequential Algorithm is not Possible. *Parallel Computing*, 3(3):259–260, 1986. Cité pages 254, 256, 258.
- [FLW87] V. FABER, Olaf M. LUBECK, et Andrew B. WHITE, Jr. Comments on the paper “Parallel Efficiency can be Greater than Unity”. *Parallel Computing*, 4:209–210, 1987. Cité pages 254, 256, 260, 262.
- [Fly66] Michael J. FLYNN. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, décembre 1966. Cité pages 25, 26, 29, 32, 33.
- [Fly72] Michael J. FLYNN. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, septembre 1972. Cité pages 25, 28, 29, 30, 31, 33, 34, 82, 94, 95, 135, 254.
- [FP81] Henry FUCHS et John POULTON. Pixel-Plane: a VLSI-oriented design for a raster graphics engine. *LAMBDA/VLSI Design*, 2(3), 1981. Cité page 19.
- [FPE<sup>+</sup>89] Henry FUCHS, John POULTON, John EYLE, Trey GREER, Jack GOLDFEATHER, David ELLSWORTH, Steve MOLNAR, Greg TURK, Brice TEBBS, et Laura ISRAEL. Pixel-Plane 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. Dans *Computer Graphics (SIGGRAPH '89)*, volume 32(4), pages 79–88. ACM, juillet 1989. Cité pages 21, 36.
- [FR72] Caxton C. FOSTER et Edward M. RISEMAN. Percolation of Code to Enhance Parallel Dispatching an Execution. *IEEE Transactions on Computers*, C-21(12):1411–1415, décembre 1972. Cité pages 168, 169, 294.
- [FWT82] Mark A. FRANKLIN, Donald F. WANN, et William J. THOMAS. Pin Limitation and Partitioning of VLSI Interconnection Networks. *IEEE Transactions on Computers*, C-31(11):1109–1116, novembre 1982. Cité pages 187, 208.
- [Gal85] Yannick Del GALLO. *Etude et Réalisation de l'Unité de Commande du Calculateur Vectoriel « OPSILA »*. Thèse de 3<sup>ème</sup> cycle, Université de Nice, novembre 1985. Cité page 122.
- [Gau92] Bernard GAULLE. *Notice d'utilisation du style french.sty*, janvier 1992. Version 3.00, Récupérable par ftp anonymous sur la machine spi.ens.fr. Cité page VII.
- [GBAS82] III GEORGE B. ADAMS et Howard Jay SIEGEL. The Extra Stage Cube: A Fault-Tolerant Interconnection Network for Supersystems. *IEEE Transactions on Computers*, C-31(5):443–454, mai 1982. Cité page 204.

- [GE89] Rajiv GUPTA et Michael EPSTEIN. Achieving Low Cost Synchronization in a Multiprocessor System. Dans *PARLE '89 Parallel Architectures and Languages Europe*, volume 365, pages I,71–84. Lecture Notes in Computer Science, Springer-Verlag, 1989. Cité page 294.
- [GE90] Rajiv GUPTA et Michael EPSTEIN. High Speed Synchronization of Processors Using Fuzzy Barriers. *International Journal of Parallel Programming*, 19(1):53–72, février 1990. Cité pages 33, 287, 294, 297.
- [GF86] Jack GOLDFEATHER et Henry FUCHS. Quadratic Surface Rendering on a Logic-Enhanced Frame-Buffer Memory. *IEEE Computer Graphics and Applications*, 6(1):48–59, janvier 1986. Cité page 22.
- [GGK<sup>+</sup>80] Allan GOTTLIEB, Ralph GRISHMAN, Clyde P. KRUSKAL, Kevin P. MCAULIFFE, Larry RUDOLPH, et Marc SNIR. The NYU Ultracomputer — Designing a MIMD, Shared-Memory Parallel Machine. Dans *SIGARCH 82*, pages 27–42. The Institute of Electrical and Electronics Engineers, Inc., 1980. Cité page 53.
- [GGSS89] Nader GHARACHORLOO, Satish GUPTA, Robert F. SPROULL, et Ivan E. SUTHERLAND. A Characterization of Ten Rasterization Techniques. Dans *Computer Graphics (SIGGRAPH '89)*, pages 355–368. Association for Computing Machinery, juillet 1989. Volume 23, Number 4. Cité page 15.
- [GL73] L. Rodney GOKE et G.J. LIPOVSKI. Banyan Networks for Partitioning Multiprocessor Systems. Dans The Institute of ELECTRICAL et Inc. ELECTRONICS ENGINEERS, éditeurs, *The Proceedings of the First Annual Symposium on Computer Architecture*, pages 21–28, 1973. Cité page 197.
- [Gla63] H. H. GLAETTLI. *Digital Fluid Logic Elements*, volume 4, pages 169–243. Academic Press, 1963. Cité page 93.
- [GLK84] D. D. GAJSKI, D. H. LAWRIE, et D. J. KUCK. Cedar. Dans *Proceedings of the COMPCON '84*, pages 306–309. The Institute of Electrical and Electronics Engineers, Inc., 1984. Cité page 196.
- [GM86] Phillip B. GIBBONS et Steven S. MUCHNICK. Efficient Instruction Scheduling for a Pipelined Architecture. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, 21(7):11–16, juillet 1986. Cité pages 168, 169, 175.
- [GMSF87a] D.D. GAJSKI, V. M. MILUTINOVIĆ, H. J. SIEGEL, et B.P. FURHT. *Tutorial: Computer Architecture*. IEEE Computer Society Press, 1987. Cité page XXIV.
- [GMSF87b] D.D. GAJSKI, V. M. MILUTINOVIĆ, H. J. SIEGEL, et B.P. FURHT. *Tutorial: Computer Architecture*. IEEE Computer Society Press, 1987. Cité page XVIII.
- [GN92] Christopher J. GLASS et Lionel M. NI. The Turn Model for Adaptive Routing. *The 19th Annual International Symposium on Computer Architecture Conference Proceedings*, 20(2):278–287, mai 1992. Cité page 202.
- [Got84] Allan GOTTLIEB. Avoiding Serial Bottlenecks in Ultraparallel MIMD Computers. Dans The Institute of ELECTRICAL et Inc. ELECTRONICS ENGINEERS, éditeurs, *COMPCON'84*, pages 354–359, 1984. Cité page 53.
- [Gou82] Michel GOUGET. *Etude et Réalisation d'un Réseau de Permutation OMEGA-BENEŠ et de sa Commande*. Thèse de docteur-ingénieur, Université de Nice, juillet 1982. Cité page 199.
- [GP85] Daniel D. GAJSKI et Jih-Kwon PEIR. Essential Issues in Multiprocessor Systems . *Computer*, 18(6):9–27, juin 1985. Voir [GMSF87b]. Cité page 25.

- [GR89] Cécile GERMAIN-RENAUD. *Etude des mécanismes de communication pour une machine massivement parallèle : MEGA*. Thèse, Université Paris XI, décembre 1989. Cité page 202.
- [GREC91] John GUSTAFSON, Diane ROVER, Stephe ELBERT, et Michael CARTER. The Design of a Scalable, Fixed-Time Computer Benchmark . *Journal of Parallel and Distributed Computing*, 12(4):388–401, août 1991. Cité pages 98,255,256,270.
- [GSMN90] Andreas GEYER-SCHULZ, Josef MATULKA, et Gustaf NEUMANN. A  $\text{\LaTeX}$  Document Style Option for Typesetting APL. *TUGboat*, 11(4):644–651, novembre 1990. Cité page 72.
- [Gus88] John L. GUSTAFSON. Reevaluating Amdahl’s Law . *Communications of the ACM*, 31(5):532–533, mai 1988. Cité page 255.
- [HA90] Paul HAEBERLI et Kurt AKELEY. The Accumulation Buffer: Hardware Support for High-Quality Rendering. Dans *Computer Graphics (SIGGRAPH '90)*, pages 309–318. Association for Computing Machinery, août 1990. Volume 24, Number 4. Cité page 15.
- [Han75] Per Brinch HANSEN. The Programmung Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, juin 1975. Voir [GMSF87a]. Cité page 70.
- [Hel76] Don HELLER. Some Aspects of the Cyclic Reduction Algorithm for Block Tri-diagonal Linear System. *SIAM Journal on Numerical Analysis*, 13(4):484–496, septembre 1976. Cité page 58.
- [Hen84] John L. HENNESSY. VLSI Processor Architecture. *IEEE Transactions on Computers*, C-31(12):1221–1246, décembre 1984. Voir [GMSF87b]. Cité page 108.
- [HFYT91] HORNING, FORSYTH, YETTER, et THAYER. How ICs impact workstations. *IEEE Spectrum*, page 61, avril 1991. Cité page 109.
- [HH80] James H. CLARK et Mark R. HANNAH. Distributed Processing in a High-Performance Smart Image Memory. *Lambda*, 1(4):369–374, 1980. Cité page 18.
- [Hil85] W. Daniel HILLIS. *The Connection Machine*. the M.I.T. press, 1985. Cité pages 95,96,101.
- [Hit91] Hitachi. *HM62A9128/8128 Series : 131072-Word  $\times$ 9(8)-bit Synchronous Cache SRAM*, mars 1991. Cité page 237.
- [HJG<sup>+</sup>82] John L. HENNESSY, Norman JOUPPI, John GILL, Forest BASKETT, Alex STRONG, Thomas GROSS, Chris ROWEN, et Judson LEONARD. The MIPS Machine. Dans *IEEE Compecon Spring '82*, pages 2–7, 1982. Voir [GMSF87b]. Cité page 108.
- [HKK<sup>+</sup>91] S. HIRANANDANI, K. KENNEDY, C. KOELBEL, U. KREMER, et C.-W. TSENG. An Overview of the Fortran D Programming System. Dans *Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 18–34. Springer-Verlag, août 1991. LNCS 589. Cité page 68.
- [HKMP91] Philippe HOOGVORST, Ronan KERYELL, Philippe MATHERAT, et Nicolas PARIS. POMP or How to Design a Massively Parallel Machine with Small Developments. Dans *PARLE '91 Parallel Architectures and Languages Europe*, volume 505(I), pages 83–100. Lecture Notes in Computer Science, Springer-Verlag, juin 1991. Cité page 8.

- [HLJ<sup>+</sup>91] Philip J. HATCHER, Anthony J. LAPADULA, Robert R. JONES, Michael J. QUINN, et Ray J. ANDERSON. A Production-Quality C\* Compiler for Hypercube Multicomputers. Dans *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26(7), pages 73–82, juillet 1991. SIGPLAN Notices. Cité pages 33, 74, 134, 135, 144.
- [Hoa78] C.A.R. HOARE. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, août 1978. Voir [GMSF87a]. Cité page 46.
- [Hor82] R. Michael HORD. *The ILLIAC IV, The First Supercomputer*. Computer Science Press, 1982. Cité pages 29, 94.
- [HQ91] Philip J. HATCHER et Michael J. QUINN. *Data-Parallel Programming on MIMD Computers*. Scientific and Engineering Computation. The MIT Press, 1991. Cité pages 44, 45, 48, 49, 74, 90, 144, 265, 269.
- [HS86] W. Daniel HILLIS et Guy L. STEELE JR.. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, décembre 1986. Cité pages 60, 96, 114.
- [HSN81] Kai HWANG, Shun-Piao SU, et Lionel M. NI. *Vector Computer Architecture and Processing Techniques*, volume 20, pages 115–197. Academic Press, 1981. Cité pages 34, 36, 119.
- [HT72] R. G. HINTZ et D. P. TATE. Control Data STAR-100 Processor Design. Dans *IEEE Comcon Fall '72*, pages 1–4, 1972. Voir [GMSF87b]. Cité pages 36, 161.
- [Hwa84] Kai HWANG. *Tutorial: Supercomputer: Design and Applications*. IEEE Computer Society Press, 1984. Cité page XXIX.
- [Hyp90] Hyperstone Electronics GmbH. *hyperstone 32-Bit-Microprocessor User's Manual*, avril 1990. Cité page 109.
- [IBM90] IBM Product Design and Development Advanced Workstations Division. *RISC System/6000 Technology*, 1990. Cité page 156.
- [IDT89] Integrated Device Technology Inc. *High Performance CMOS Data Book Supplement*, application note an-09 édition, 1989. Cité page 108.
- [IJT90] François IRIGOIN, Pierre JOUVELOT, et Rémi TRIOLET. Overview of the PIPS Project. Dans *International Workshop on Compilers for Parallel Computers*, pages 199–212. ENSMP CAI - UPMC MASI, décembre 1990. Cité page 90.
- [INM89] Inmos. *The Transputer Databook*, 1989. Cité pages 37, 106.
- [INM91] inmos — SGS-THOMSON. *The T9000 Transputer Products Overview Manual*, first édition, 1991. Cité pages 37, 106, 109.
- [Int88] Intel Corporation. *iWarp Architecture Overview*, mai 1988. Cité page 33.
- [INT89a] Intel. *i860 64-Bit Microprocessor*, advance information édition, avril 1989. Cité pages 17, 109.
- [INT89b] Intel. *i860 64-Bit Microprocessor Programmer's Reference Manual*, 1989. Cité page 168.
- [Int90] Integrated Device Technology Inc. *1991 IDT RISC R3000A, R3001m R3051 Family Product Information*, 1990. Cité page 109.
- [Int91a] Integrated Device Technology Inc. *Third Generation 64-bit Super-pipelined RISC Microprocessor*, octobre 1991. Cité page 109.
- [Int91b] Intel. *i860 XP Microprocessor Data Book*, preliminary édition, mai 1991. Cité page 109.

- [Int91c] Intel Corporation. *A Touchstone DELTA System Description*, février 1991. Advanced Information. Cité pages 37, 40, 202, 302.
- [Int91d] Intergraph Advanced Processor Division. *CLIPPER C411 CPU — Integer Unit*, advanced information édition, janvier 1991. Cité pages 109, 156.
- [Ive62] Kenneth E. IVERSON. *A Programming Language*. John Wiley and Sons, Inc., 1962. Cité pages 55, 72.
- [Jac90] James H. JACKSON. The Data Transport Computer<sup>TM</sup>: A 3-Dimensional Massively Parallel SIMD Computer. Rapport Technique, Wavetracer Inc., 1990. Cité pages 36, 40, 94, 123.
- [JC91] Dz-Ching JU et Wai-Mee CHING. Exploitation of APL Data Parallelism on a Shared-memory MIMD Machine. Dans *Third 91M SIGPLAN Symposium on Principles & Practice of Parallel Programming*, volume 26(7) de *SIGPLAN Notices*, pages 61–72, juillet 1991. Cité pages 72, 265.
- [Jeg87] Yvon JEGOU. Le langage vectoriel Hellena. Rapport Technique 368, IRISA, juin 1987. Cité page 71.
- [Jes84] Frank JESCHONNEK. A New Type of Paralel Computer Using Microprocessors. Dans *Parallel Computing 83*, pages 527–532. North-Holland, 1984. Cité page 136.
- [Jor85] Harry F. JORDAN. *Parallel MIMD Computation: HEP Supercomputer and Its Application*, Chapitre HEP Architecture, Programming and Performance, pages 1–40. Janusz S. Kowalik, the mit press édition, 1985. Cité page 65.
- [Kar87] Alan H. KARP. Programming for Parallelism. *Computer*, 20(5):43–57, mai 1987. Cité pages 43, 46, 61.
- [Ker88] Ronan KERYELL. POMP : Vidéo & Entrées-Sorties. Diplôme d'étude approfondie, Paris XI, septembre 1988. Cité pages 95, 234.
- [Ker89] Ronan KERYELL. POMP2 : D'un Petit Ordinateur Massivement Parallèle. Rapport de magistère, LIENS — Ecole Normale Supérieure, octobre 1989. Cité pages 8, 125, 127, 247, 258, 259, 262.
- [Ker92] Ronan KERYELL. Le Contrôle de Flot dans les Machines SIMD. Dans *RenPar4 — 4<sup>èmes</sup> rencontres de Parallélisme*, pages 100–103, Université des Sciences et Technologies de Lille — Villeneuve d'Ascq, France, mars 1992. Cité page 30.
- [Kil91] Michael F. KILIAN. Object-Oriented Programming for Massively Parallel Machines. Dans *Proceedings of the 1991 International Conference on Parallel Processing*. CRC Press Inc., août 1991. volume II. Cité pages 50, 51, 63.
- [KK79] Parviz KERMANI et Leonard KLEINROCK. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Network*, 3(4):267–286, 1979. Cité page 200.
- [KK82] Svetlana P. KARTASHEV et Steven I. KARTASHEV, éditeurs. *LSI modular computer systems*, volume 1, pages 130–135. Prentice-Hall, 1982. PEPE, the Parallel Element Processing Ensemble. Cité page 141.
- [KKLW80] David J. KUCK, Robert H. KUHN, Bruce LEASURE, et Michael WOLFE. The Structure of an Advanced Retargetable Vectorizer . Dans *The Proceedings of COMPSAC'80*. The Institute of Electrical and Electronics Engineers, Inc., 1980. Voir [Hwa84]. Cité page 64.
- [KLGSL90] Kathleen KNOBE, Joan D. LUCAS, et Jr. GUY L. STEELE. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, février 1990. Cité page 68.

- [KM89] Les KOHN et Neal MARGULIS. Introducing the Intel i860 64-Bit Microprocessor . *IEEE Micro*, pages 15–30, août 1989. Cité page 109.
- [KMC72] David J. KUCK, Yoichi MURAOKA, et Shyh-Ching CHEN. On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup. *IEEE Transactions on Computers*, C-21(12):1293–1310, décembre 1972. Cité pages 96, 97, 168, 171.
- [KMR87] Charles KOELBEL, Piyush MEHROTRA, et John Van ROSENDALE. Semi-Automatic Domain Decomposition in BLAZE. Dans *Proceedings of the 1987 International Conference on Parallel Processing*, pages 521–524. The Pennsylvania State University Press, 1987. Cité page 71.
- [KNS91] Shin-Dug KIM, Mark A. NICHOLS, et Howard Jay SIEGEL. Modeling Overlapped Operation between the Control Unit and Processing Elements in an SIMD Machine. *Journal of Parallel and Distributed Computing*, 12(4):329–342, août 1991. Cité pages 30, 258, 260.
- [KO90] David M. KOPPELMAN et A. YAVUZ ORUÇ. A Self-Routing Permutation Network. *Journal of Parallel and Distributed Computing*, 10(2):140–151, octobre 1990. Cité page 199.
- [Kog74a] P. M. KOGGE. Maximal Rate Pipelined Solutions to Recurrence Problems. *IBM Journal of Research and Development*, 18(3):138–148, mars 1974. Cité pages 55, 58.
- [Kog74b] P. M. KOGGE. Parallel Solution of Recurrence Problems. *IBM Journal of Research and Development*, 18(3):138–148, mars 1974. Cité pages 7, 55, 58, 59, 291.
- [Kog81] P. M. KOGGE. *The Architecture of Pipelined Computers*, Chapitre Pipelines with feedbacks, pages 57–66. McGraw-Hill Book Company, 1981. Cité pages 55, 58.
- [Kor85] Cary D. KORNFELD. *Architercural Elements for Bitmap Graphics*. PhD thesis, Stanford University — Department of Electrical Engineering & Xerox Corporation, juin 1985. Rapport Technique Xerox CSL-85-2. Cité page 17.
- [Kot87] S. C. KOTHARI. *Multistage Interconnection Networks fo Multiprocessor Systems*, volume 26, pages 155–199. Academic Press, 1987. Cité pages 204, 208.
- [KPDL<sup>+</sup>79] Jr. KENDALL PRESTON, Michael J.B. DUFF, Stephano LEVIALDI, Philip E. NORGREN, et Jun-Ichiro TORIWAKI. Basics of Cellular Logic with Some Applications in Medical Image Processing. Dans *Proceedings of the IEEE*, volume 67(5), pages 826–856, mai 1979. Voir [GMSF87a]. Cité page 94.
- [KR88] Brian W. KERNIGHAN et Dennis M. RITCHIE. *The C programming language*. Prentice-Hall, second edition (ansi c) édition, 1988. Cité page 167.
- [KRS85] Clyde P. KRUSKAL, Larry RUDOLPH, et Marc SNYR. The Power of Parallel Prefix. *IEEE Transactions on Computers*, C-34(10):965–968, octobre 1985. Cité pages 49, 55, 59, 60, 88, 96, 97, 256.
- [KS73] Peter M. KOGGE et Harold S. STONE. A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, C-22(8):786–792, août 1973. Cité pages 20, 55, 58, 291.
- [KS82] David J. KUCK et Richard A. STOCKES. The Burroughs Scientific Processor (BSP). *IEEE Transactions on Computers*, C-31(5):363–376, mai 1982. Cité pages 39, 60, 94.

- [KS91a] S. KONSTANTINIDOU et L. SNYDER. The Chaos Router : a Practical Application of Randomization in Network Routing. *ACM SIGARCH Computer Architecture Newsletter*, 19(1):79–88, mars 1991. Cité page 202.
- [KS91b] S. KONSTANTINIDOU et L. SNYDER. Chaos Router : Architecture and Performance. Dans *The 18<sup>th</sup> Annual International Symposium on Computer Architecture*, volume 19(3), pages 212–221. ACM SIGARCH Computer Architecture Newsletter, mai 1991. Cité page 202.
- [KT80] Edward W. KOZDROWICKI et Douglas J. THEIS. Second Generation of Vector Supercomputers . *Computer*, 13(11):71–83, novembre 1980. Voir [Hwa84]. Cité page 163.
- [Kuc68] David J. KUCK. ILLIAC IV Software and Application Programming. *IEEE Transactions on Computers*, C-17(8):758–770, août 1968. Cité pages 55, 59, 60.
- [Kuc76] David J. KUCK. *Advances in Computers*, volume 15, Chapitre Parallel Processing of Ordinary Programs, pages 119–179. Academic Press, 1976. Cité pages 255, 261, 291, 293.
- [KV90] David KIRK et Douglas VOORHIES. The Rendering Architecture of the DN10000VS. Dans *Computer Graphics (SIGGRAPH '90)*, pages 299–307. Association for Computing Machinery, août 1990. Volume 24, Number 4. Cité page 15.
- [Law75] Duncan H. LAWRIE. Access and Alignment of Data in an Array Processor . *IEEE Transactions on Computers*, C-24(12):1145–1155, décembre 1975. Cité pages 38, 39, 196, 197.
- [LB80] Stephen F. LUNDSTROM et George H. BARNES. A Controllable MIMD Architecture. Dans *Proceedings of the 1980 International Conference on Parallel Processing*, pages 19–27. The Institute of Electrical and Electronics Engineers, Inc., 1980. Voir [GMSF87a]. Cité page 287.
- [LC91] L.-C. LU et M. CHEN. Parallelizing Loops with Indirect Array References or Pointers. Dans *Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 201–217. Springer-Verlag, août 1991. LNCS 589. Cité page 90.
- [LEH<sup>+</sup>89] LITAIZE, ELKHLIFI, HAMMAMI, LALAM, MZOUGHFI, SAINRAT, et SALINIER. Serial Multiport Memory Multiprocessors. Dans *PARLE '89 Parallel Architectures and Languages Europe*, volume 365, pages I,34–51. Lecture Notes in Computer Science, Springer-Verlag, juin 1989. Cité pages 188, 203.
- [Lei85] Charles E. LEISERSON. FAT-TREES: Universal Networks for Hardware-Efficient Supercomputing. Dans *Proceedings of the International Conference on Parallel Processing*, pages 393–402. The Institute of Electrical and Electronics Engineers, Inc., août 1985. Cité page 197.
- [Len78] Jacques LENFANT. Parallel Permutations of Data: A Beneš Network Control Algorithm for Frequently Used Permutations. *IEEE Transactions on Computers*, C-27(7):637–647, juillet 1978. Cité page 199.
- [Lep89] Éric LEPRÊTRE. *Architectures Cellulaires pour la Synthèse d'Images*. Thèse, Laboratoire d'Informatique Fondamentale de Lille — Université des Sciences et Techniques de Lille Flandres Artois, juin 1989. Cité page 22.
- [LF80] Richard E. LADNER et Michael J. FISHER. Parallel Prefix Computation. *Journal of the ACM*, 27(4):831–838, octobre 1980. Cité pages 55, 291.

- [LG92] François LÉVÊQUE et Matthieu GLACHANT. Diversité Génétique : la Gestion Mondiale des Ressources Vivantes. *La Recherche*, 23(239):114–123, janvier 1992. Cité page 44.
- [LH91] Daniel H. LINDER et Jim C. HARDEN. An Adaptative and Fault Tolerent Worm-hole Routing Strategy for  $k$ -ary  $n$ -cubes. *IEEE Transactions on Computers*, 40(1):2–12, janvier 1991. Cité pages 203, 281.
- [Lin82] Neil R. LINCOLN. Technology and Design Tradeoffs in the Creation of a Modern Supercomputer . *IEEE Transactions on Computers*, C-31(5):349–353, mai 1982. Cité pages 3, 30, 36, 100, 161, 302.
- [LM85] Jacques LENFANT et Claude MICHEL. Les Réseaux d'Interconnexion. Dans INRIA, éditeur, *Cours et séminaires, conférences, Saint-Cyprien*, pages 11–21, octobre 1985. Cité page 199.
- [Lub90] Boris D. LUBACHEVSKY. Synchronization Barrier and Related Tools for Shared Memory Parallel Programming. *International Journal of Parallel Programming*, 19(3):225–250, juin 1990. Cité page 297.
- [LV82] Duncan H. LAWRIE et Chandra R. VORA. The Prime Memory System for Array Access. *IEEE Transactions on Computers*, C-31(5):435–442, mai 1982. Cité pages 35, 38, 39, 94.
- [Mar92a] Roland MARBOT. *Architecture des Systèmes de Traitement de l'Information : Impact de l'Évolution des Techniques Microélectroniques*. Rapport d'habilitation à diriger des recherches, Université Pierre et Marie Curie — PARIS VI, mars 1992. Cité page 4.
- [Mar92b] Philippe MARQUET. *Langages explicites à parallélisme de données*. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille — Université des Sciences et Technologies de Lille, février 1992. Cité pages 63, 77.
- [Mat88] Philippe MATHERAT. *Contribution à l'augmentation de puissance des architectures de visus graphiques*. Thèse d'état, Paris VI, mai 1988. Cité pages 24, 234.
- [McK92] Kathryn S. MCKINLEY. *Automatic and Interactive Parallelization*. PhD thesis, Department of Computer Science — Rice University, Houston, Texas, USA, avril 1992. Cité pages 91, 266.
- [Mel89] Charles MELEAR. The Design of the 88000 RISC Family. *IEEE Micro*, pages 26–38, avril 1989. Cité page 109.
- [Mic91] Micron Technology, Inc. *MOS Data Book*, 1991. Cité pages 22, 270.
- [ML89] W. G. P. MOOIJ et A. LIGTENBERG. Architecture of a Communication Network Processor. Dans *PARLE '89 Parallel Architectures and Languages Europe*, volume 365, pages I,238–250. Lecture Notes in Computer Science, Springer-Verlag, juin 1989. Cité page 202.
- [MOT88a] MOTOROLA. *MC88100 RISC processor user's manual*, 1988. Cité pages 109, 168, 231.
- [MOT88b] MOTOROLA. *MC88200 Cache/Memory Management Unit User's Manual*, 1988. Cité pages 275, 280.
- [MOT89] MOTOROLA. *DSP96002 — 96-Bit General-Purpose IEEE Floating-Point Digital Signal Processor (DSP)*, 1989. Cité page 109.

- [MR87] Piyush MEHROTRA et John Van ROSENDALE. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5(3):161–168, novembre 1987. Cité page 71.
- [MR90] Michael METCALF et John REID. *Fortran 90 Explained*. Oxford Science Publications, 1990. Cité pages 64, 67, 133.
- [MS80] Philip M. MERLIN et Paul J. SCHWEITZER. Deadlock Avoidance in Store-and-Forward Networks — I: Store-and-Forward Deadlock. *IEEE Transactions on Communications*, COM-28(3):345–354, mars 1980. Cité page 201.
- [MU84] Kenichi MIURA et Keiichiro UCHIDA. Facom Vector Processor System: VP-100/VP-200. Dans *Proceedings of NATO Advanced Research Workshop on High Speed Computing*, volume F7, pages 59–73. IEEE Computer Society Press, 1984. Voir [Hwa84]. Cité pages 36, 145.
- [Mye91] Ware MYERS. Massively parallel systems break through at Supercomputing 90. *Computer*, 24(1):121–126, janvier 1991. The Institute of Electrical and Electronics Engineers, Inc. Cité page 6.
- [NCR84] NCR. *Geometric arithmetic parallel processor NCR45CG72*, 1984. Cité pages 94, 105.
- [NEC88] NEC. *RISC Microprocessors Vr3000 & Vr3010 MIPS RISC Architecture, User's Manual*, 1988. Cité page 109.
- [NEC91] NEC Supercomputer. *SX-3R series*, 1991. Cat. ° E51473 92022001KP. Cité page 36.
- [NFA<sup>+</sup>90] Wayne G. NATION, Samuel A. FINEBERG, Mark D. ALLEMANG, Thomas SCHWEDERSKI, Thomas L. CASAVANT, et Howard Jay SIEGEL. Efficient Masking Techniques for Large-Scale SIMD Architectures. Dans *Proceedings of the Third Symposium on the Frontiers of Massively Paralle Computation*, pages 259–264. IEEE Computer Society Press, octobre 1990. Cité page 134.
- [Ni90] Yang NI. *Contribution à l'étude des architectures massivement parallèles pyramidales: Réalisation d'un circuit VLSI multi-processeur et définition d'un modèle de contrôle dynamique*. Thèse, Paris XI, février 1990. Cité page 102.
- [NL91] Bill NITZBERG et Virginia LO. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52–60, août 1991. Cité pages 40, 98, 279.
- [OAS91] OASYS. *Green Hills Compiler Family Cross Development Guide*, janvier 1991. Version 1.8.5. Cité pages 164, 169.
- [OD90a] Matthew T. O'KEEFE et Henry G. DIETZ. Hardware Barrier Synchronization: Dynamic Barrier MIMD (DBM). Dans *Proceedings of the 1990 International Conference on Parallel Processing*, pages I-43–I-46. IEEE, août 1990. Cité pages 287, 297.
- [OD90b] Matthew T. O'KEEFE et Henry G. DIETZ. Hardware Barrier Synchronization: Static Barrier MIMD (SBM). Dans *Proceedings of the 1990 International Conference on Parallel Processing*, pages I-35–I-42. IEEE, août 1990. Cité pages 287, 295.
- [OMMN90] Brett OLSSON, Robert MONTROYE, Peter MARKSTEIN, et MyHong NGUYENPHU. RISC System/6000 Floating-Point Unit. Dans *RISC System/6000 Technology*. IBM Product Design and Development Advanced Workstations Division, 1990. Cité pages 32, 154.

- [OPP91] Luis F. ORTIZ, Ron Y. PINTER, et Shlomit S. PINTER. An Array Language for Data Parallelism: Definition, Compilation, and Application. *The Journal of Supercomputing*, (5):7–29, mai 1991. Cité page 61.
- [Par88] Nicolas PARIS. Définition de POMPC (draft). Rapport préliminaire, décembre 1988. Cité page 79.
- [Par89] Nicolas PARIS. *Développement d'outils de conception de circuits intégrés et application à la réalisation d'une architecture de visualisation*. Thèse, Université Paris XI, mai 1989. Cité pages 17, 18.
- [Par91] Nicolas PARIS. MOD2MAG User's Manual. Rapport Technique LIENS-91-17, Laboratoire d'Informatique de l'École Normale Supérieure, novembre 1991. Récupérable par `ftp anonymous` sur la machine `spi.ens.fr` dans le fichier `pub/reports/liens/liens-91-17.A4.ps.Z`. Cité page 250.
- [Par92] Nicolas PARIS. Définition de POMPC (Version 1.99). Rapport Technique LIENS-92-5, Laboratoire d'Informatique de l'École Normale Supérieure, mars 1992. Récupérable par `ftp anonymous` sur la machine `spi.ens.fr` dans le fichier `pub/reports/liens/liens-92-5.A4.ps.Z`. Cité page 79.
- [Pat81] Jamak H. PATEL. Performance of Processor-Memory Interconnections for Multiprocessors. *IEEE Transactions on Computers*, C-30(10):771–780, octobre 1981. Cité pages 197, 215.
- [PCMP85] Ronald H. PERROTT, Danny CROOKES, Peter MILLIGAN, et W. R. Martin PURDY. A Compiler for an Array and Vector Processing Language. *IEEE Transactions on Software Engineering*, SE-11(5):471–478, mai 1985. Cité pages 71, 82, 144.
- [PDGO87] W. J. POPPELBAUM, A. DOLLAS, J.B. GLICKMAN, et C. O'TOOL. *Unary Processing*, volume 26, pages 47–92. Academic Press, 1987. Cité pages 5, 93, 239.
- [Pea77] Marshall C. PEASE III. The Indirect Binary  $n$ -Cube Microprocessor Array. *IEEE Transactions on Computers*, C-26(5):458–473, mai 1977. Cité pages 197, 204.
- [Per79] R. H. PERROTT. A Language for Array and Vector Processors. *ACM Transactions on Programming Languages and Systems*, 1(2):177–195, octobre 1979. Voir [GMSF87a]. Cité pages 63, 70, 81.
- [PH89] Michael POTMESIL et Eric M. HOFFERT. The **Pixel Machine**: A Parallel Image Computer. Dans *Computer Graphics (SIGGRAPH '89)*, volume 32(4), pages 69–78. ACM, juillet 1989. Cité page 22.
- [PHE77] Jr. PHILIP H. ENSLOW. Multiprocessor Organization—A Survey. *ACM Computing Surveys*, 9(1):122–126, mars 1977. Voir [GMSF87a]. Cité pages 4, 25.
- [Pig20] A. PIGOU. *The Economics of Welfare*. Macmillan, 1920. Cité page 44.
- [PKB<sup>+</sup>91] M. L. POWEKK, S. R. KLEIMAN, S. BARTON, D. SHAH, D. STEIN, et M. WEEKS. SunOS 5.0 Multithread Architecture,. Rapport Technique, Sun Microsystems, Inc., septembre 1991. Cité pages 46, 165.
- [PLX88a] PLX Technology Corporation. *VME 2000 — VMEbus Slave Module Interface Device*, mai 1988. Cité page 221.
- [PLX88b] PLX Technology Corporation. *VME 3000 — VMEbus Interrupt Generator*, mai 1988. Cité page 221.
- [PM92] Douglas M. PASE et Tom MACDONALD. MPP Fortran Programming Model. Rapport Technique, Cray Research, Inc., février 1992. Cité page 40.

- [Pot91] Bernard POTTIER. *ArMen : Une machine parallèle intégrant un réseau de circuits logiques programmables*. Thèse, Université de Rennes I, juin 1991. Cité page 288.
- [PP82] David A. PATTERSON et Richard S. PIEPHO. Assessing RISCs in High-Level Language Support. *IEEE Micro*, pages 9–18, novembre 1982. Voir [GMSF87b]. Cité page 108.
- [PR82] D. S. PARKER et C. S. RAGHAVENDRA. The Gamma Network: A Multiprocessor Interconnection Network with Redundant Paths. Dans *9<sup>th</sup> Annual Symposium on Computer Architecture*, pages 73–80, avril 1982. Cité page 291.
- [PV81] Franco P. PREPARATA et Jean VUILLEMIN. The Cube-Connected Cycles: A Versatile Network for Parallel Computation. *Communications of the ACM*, 24(5):300–309, mai 1981. Cité page 193.
- [PW89] Richard S. PIEPHO et William S. WU. A Comparison of RISC Architectures. *IEEE Micro*, pages 51–62, août 1989. Cité page 109.
- [QS90] Michael J. QUINN et Bradley K. SEEVERS. Implementing a Data Parallel Language on a Tightly Coupled Multiprocessor. Dans *Third Workshop on Programming Languages and Compilers for Parallel Computing*, pages 1–19. Preliminary Proceedings to be published by Pitman/MIT Press, août 1990. Cité page 144.
- [QS91] Quality Semiconductor, Inc. *QUICKSWITCH<sup>TM</sup> — Applications Notes & Data Sheets*, avril 1991. Cité page 288.
- [Rap83] Michèle RAPHALEN. *Parallélisation d'algorithmes d'analyse de données*. Thèse de 3<sup>ème</sup> cycle, Université de Rennes I, novembre 1983. ° A/816/139. Cité pages v, 94.
- [RB89] Jonathan B. ROSENBERG et Jonathan D. BECHER. Mapping Massive SIMD Parallelism onto Vector Architectures for Simulation. *Software — Practice and Experience*, 19(8), août 1989. Cité pages 94, 95, 133.
- [RCCT90] Randall D. RETTBERG, William R. CROWTHER, Philippe P. CARVEY, et Raymond S. TOMLINSON. The Monarch Parallel Processor Hardware Design. *Computer*, 23(4):18–30, avril 1990. The Institute of Electrical and Electronics Engineers, Inc. Cité page 37.
- [Rei60] George W. REITWIESNER. *Binary Arithmetic*, volume 1, pages 231–308. Academic Press, 1960. Cité page 93.
- [RF72] Edward M. RISEMAN et Caxton C. FOSTER. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, décembre 1972. Cité pages 28, 168.
- [RFS88] Darwen RAU, Jose A. B. FORTES, et Howard Jay SIEGEL. Destination Tag Routing Techniques Based on a State Model for the IADM Network. Dans Computer Society Press of the IEEE, éditeur, *The 15<sup>th</sup> Annual International Symposium on Computer Architecture Conference Proceedings*, pages 318–325, février 1988. Cité page 291.
- [RJ83] Robert RANNOU et Yvon JEGOU. Tris pour Machines Synchrones. *Techniques et Sciences Informatiques*, 2(6):427–444, 1983. Cité page 192.
- [RJH92] RONG-JAYE et Yu-Song HOU. Non-Associative Parallel Prefix Computation. *Information Processing Letters*, 44(2):91–94, novembre 1992. Cité page 59.
- [Rou90] Frank ROUSÉE. *Conception d'une cellule de calculs arithmétiques pour la synthèse d'image*. Thèse, Université de Rennes I, octobre 1990. Cité page 33.

- [RS83] Daniel A. REED et Herbert D. SCHWETMAN. Cost-Performance Bounds for Multicomputer Networks. *IEEE Transactions on Computers*, 32(1):83–95, janvier 1983. Cité page 187.
- [Rus78] Richard M. RUSSEL. The CRAY-1 Computer System. *Communications of the ACM*, 21(1):63–72, janvier 1978. Voir [GMSF87a]. Cité pages 30, 34, 36, 100, 103, 111, 146, 161.
- [Sab92] Gary SABOT. Optimized CM Fortran Compiler for the Connection Machine Computer. Dans *Proceedings of the 25<sup>th</sup> Hawaii International Conference on System Sciences*, pages 161–172. The Institute of Electrical and Electronics Engineers, Inc., janvier 1992. Cité pages 95, 96, 98, 101, 102, 302.
- [Sai91] Pascal SAINRAT. *Réseau d'Interconnexion du Multiprocesseur M3S — Étude et Mise en Œuvre*. Thèse, Université Paul Sabatier, Toulouse, juin 1991. Cité page 188.
- [SBM62] Daniel L. SLOTNICK, W. Carl BORCK, et Robert C. MCREYNOLDS. The SOLOMON Computer. Dans *Proceedings of the Fall 1962 Eastern Joint Computer Conference*, pages 97–107, décembre 1962. Cité pages 36, 94, 134, 141, 303.
- [SHI92] Toshiyuki SHIMIZU, Takeshi HORIE, et Hiroaki ISHIHATA. Low-Latency Message Communication Support for the AP1000. Dans *Conference Proceedings of the 19<sup>th</sup> Annual International Symposium on Computer Architecture*, volume 20(2), pages 288–297. ACM SIGARCH Computer Architecture Newsletter, mai 1992. Cité page 281.
- [Sie91] Siemens Components, Inc. *MIPS R4000 — Introduction to the MIPS R4000 Microprocessor*, 1991. Cité page 109.
- [Sla92] Michael SLATER. Wave of High-End Processors Due : Most in 40–80 SPECmarks Range, 16–36K On-Chip Cache. *Microprocessor Report*, 6(2):9–15, février 1992. Cité page 109.
- [SM91] Howard SACHS et Harlan MCGHAN. *The C400 Chipset Architecture*. Intergraph Advanced Processor Division, janvier 1991. Cité pages 109, 156.
- [SN90] Xian-He SUN et Lionel M. NI. Another View on Parallel Speedup. Dans *Proceedings Supercomputing '90*, pages 559–568. The Institute of Electrical and Electronics Engineers, Inc., novembre 1990. Cité pages 253, 254, 255.
- [Sny88] Lawrence SNYDER. A Taxonomy of Synchronous Parallel Machines. Dans *Proceedings of the 1988 International Conference on Parallel Processing*, pages 281–285. The Pennsylvania State University Press, 1988. Cité pages 26, 48.
- [SP89] François SILLION et Claude PUECH. A General Two-Pass Method Untegrating Specular and Diffuse Reflection. Dans *Computer Graphics (SIGGRAPH '89)*, volume 23(3), pages 335–344. Association for Computing Machinery, juillet 1989. Cité page 13.
- [SS88] Youcef SAAD et Martin H. SCHULTZ. Topological Properties of Hypercubes. *IEEE Transactions on Computers*, C-37(7):867–872, juillet 1988. Cité pages 188, 190, 292.
- [SSDK84] Howard Jay SIEGEL, Thomas SCHWEDERSKI, Nathaniel J. DAVIS, IV, et James T. KUEHN. PASM: A Reconfigurable Parallel System For Image Processing. *ACM SIGARCH Computer Architecture Newsletter*, 12(4):7–19, septembre 1984. Cité pages 94, 105, 134, 269.

- [SSKD87] Howard Jay SIEGEL, Thomas SCHWEDERSKI, James T. KUEHN, et Nathaniel J. DAVIS, IV. *Tutorial: Computer Architecture*, Chapitre An Overview of the PASM Parallel Processing Sytem , pages 387–407. IEEE Computer Society Press, 1987. Cité pages 37, 94, 105, 107, 141, 149, 269.
- [Ste90] Guy L. STEELE JR.. Making Asynchronous Parallelism Safe for the World. Dans *Conference Record of the Seventeenth Annual ACM Symposium Principles of Programming Languages*, pages 218–231. ACM SIGACT SIGPLAN, janvier 1990. Cité pages 43, 44, 45, 265, 287.
- [Sto70] Harold S. STONE. The Organization of High-Speed Memory for Parallel Block Transfer of Data. *IEEE Transactions on Computers*, C-19(1):47–53, janvier 1970. Cité pages 18, 38, 40.
- [Sto71] Harold S. STONE. Parallel Processing with the Perfect Shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, février 1971. Cité pages 141, 192.
- [Sto73] Harold S. STONE. An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations. *Journal of the ACM*, 20(1):27–38, janvier 1973. Cité pages 58, 59, 60.
- [Sto77] Richard A. STOCKES. Burroughs Scientific Processor. *High Speed Computer and Algorithm Organization*, pages 85–89, 1977. Voir [GMSF87a]. Cité page 94.
- [Szy89] Ted SZYMANSKI. On the permutation capability of a circuit-switched hypercube. Dans *International Conference on Parallel Processing*, pages 103–110. The Institute of Electrical and Electronics Engineers, Inc., Academic Press, 1989. Cité page 199.
- [Tar83] Robert Endre TARJAN. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983. Cité page 55.
- [TF70] Garold S. TJADEN et Michael J. FLYNN. Detection and Parallel Execution of Independent Instructions. *IEEE Transactions on Computers*, C-19(10):889–895, octobre 1970. Cité pages 32, 171, 172.
- [Thi87a] Thinking Machine Corporation. *Connection Machine Model CM-2 Technical Summary*, avril 1987. HA87-4. Cité pages 36, 94, 280.
- [Thi87b] Thinking Machine Corporation. *Connection Machine Model CM-2 Technical Summary*, avril 1987. pages 35–41. Cité page 73.
- [Thi90] Thinking Machine Corporation. *C\* Programming Guide*, novembre 1990. Version 6.0. Cité page 75.
- [Thi91] Thinking Machine Corporation. *The Connection Machine CM-5 Technical Summary*, octobre 1991. Cité pages 33, 37, 40, 60, 144, 197, 269, 280, 301.
- [Tho64] James E. THORNTON. Parallel Operation in the Control Data 6600. Dans *Proceedings of the 1964 Fall Joint Computer Conference*, pages 33–40, 1964. Cité page 172.
- [Tho86] Marc THORIN. *Manuel ADA — Langage Normalisé Complet*. Masson, 1986. Cité page 46.
- [TIF86] Rémi TRIOLET, François IRIGOIN, et Paul FEAUTRIER. Direct Parallelization of Call Statement. Dans *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, volume 21(7), pages 176–185. ACM SIGPLAN Notices, juillet 1986. Cité pages 90, 170.
- [Tom67] R. M. TOMASULO. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, 11(1):25–33, janvier 1967. Cité page 32.

- [TR88] Lewis W. TUCKER et George G. ROBERTSON. Architecture and Applications of the Connection Machine. *Computer*, 21(8):26–38, août 1988. Cité pages 40, 94, 119, 123.
- [TS85] David Lee TUOMENOKSA et Howard Jay SIEGEL. Task Scheduling on the PASM Parallel Processing System. *IEEE Transactions on Software Engineering*, SE-11(2):145–157, février 1985. Cité page 267.
- [Tuc90] Russ TUCK. *Porta-SIMD: An Optimally Portable SIMD Programming Language*. PhD thesis, University of North Carolina at Chapel Hill, mai 1990. Cité pages 26, 78.
- [Ung58] S. H. UNGER. A Computer Oriented Toward Spatial Problems. Dans *Proceedings of the IRE*, pages 1744–1750, octobre 1958. Cité pages v, 36, 94, 303.
- [VME87] VMEbus International Trade Association. *The VMEbus Specification*, septembre 1987. Conforms to: ANSI/IEEE STD1014-1987, IEC 821 and 297. Cité page 221.
- [Wav91] Wavetracer Inc. *The multiC Programming Language — User Documentation*, pub-00001-001-1.01 édition, septembre 1991. Cité page 76.
- [WF80] Chuan-Lin WU et Tse-Yun FENG. On a Class of Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-29(8):694–702, août 1980. Cité pages 197, 208.
- [WF81] Chuan-Lin WU et Tse-Yun FENG. The Universality of the Shuffle-Exchange Network. *IEEE Transactions on Computers*, C-30(5):324–332, mai 1981. Cité page 196.
- [WLH81] William A. WULF, Roy LEVIN, et Samuel P. HARBISON. *HYDRA/C.mmp An Experimental Computer System*. McGraw-Hill Book Company, 1981. Cité page 93.
- [WyF84] Chuan-Lin WU et Tse yun FENG. *Tutorial: Interconnection Networks for Parallel and Distributed Processing*. IEEE Computer Society Press, 1984. Cité page 186.
- [XIL90] XILINX. *XC 4000 Logic Cell<sup>TM</sup> Array Family*, 1990. Technical Data. Cité pages 206, 241.
- [ZBC<sup>+</sup>92] H. ZIMA, P. BREZANY, B. CHAPMAN, P. MEHROTRA, et A. SCHWALD. Vienna Fortran — A Language Specification Version 1.1. Rapport Technique ACPC/TR 92-4, ACPC — Austrian Center for Parallel Computation, mars 1992. Cité page 69.
- [ZD91] Belkacem ZERROUK et Anne DÉRIEUX. Une Architecture Parallèle SIMD à Instruction Longue. Rapport Technique 1413, IRISA, avril 1991. Cité page 94.
- [ZDO90] Abderrazek ZAAFRANI, Henry G. DIETZ, et Matthew T. O’KEEFE. Static Scheduling for Barrier MIMD Architectures. Dans *Proceedings of the 1990 International Conference on Parallel Processing*, pages II-187–II-194. IEEE, août 1990. Cité page 287.



# Table des figures

2.1	Schématisation des étapes entrant dans la synthèse d'une image. . . . .	14
2.2	Partie visualisation d'un ordinateur sans matériel spécialisé pour faire du graphique. . . . .	16
2.3	Synoptique de la station de travail IRIS. . . . .	19
2.4	Synoptique de la gestion des pixels dans les machines PIXEL-PLANE. . .	21
2.5	Synoptique de la PIXEL MACHINE. . . . .	23
2.6	Synoptique d'une machine SIMD et sa représentation sous forme de flots.	27
2.7	Amélioration de la bande passante d'instructions grâce au pipeline. . .	28
2.8	Architecture typique de machine SIMD avec ses flots. . . . .	29
2.9	Architecture typique de machine MIMD avec les flots correspondants. .	31
2.10	Synoptique d'une machine MISD avec les flots correspondants. . . . .	32
2.11	Principale classification du grain du parallélisme. . . . .	35
2.12	Comparaison entre le couplage fort et le couplage faible. . . . .	38
3.1	Une machine $\equiv$ un tableau de processeurs. . . . .	49
3.2	Résoudre un problème $\equiv$ avoir un processeur de tableaux. . . . .	50
3.3	Les deux types de communications générales. . . . .	53
3.4	Exemple de communication où des données dans des processeurs vir- tuels (PV) en gris foncé restent dans les processeurs physiques (PP). . .	54
3.5	Concentrations ou réductions associatives. . . . .	56
3.6	Comparaison entre une réduction exécutée séquentiellement ou paral- lèlement suivant un arbre binaire. . . . .	57
3.7	Les types de communications scalaires. . . . .	58
3.8	Addition préfixe parallèle. . . . .	59
4.1	Petit exemple de C*. . . . .	74
4.2	Exemple de collections et de déclarations de variables en POMPC. . . . .	81
4.3	Petit exemple à base de valeur absolue parallèle. . . . .	83
4.4	Sémantique du <b>where</b> sur fond de laplacien dans une méthode de l'échi- quier. . . . .	86
4.5	Exemple de fonction parallèle générique. . . . .	87
4.6	Virtualisation explicite dans une routine de compaction de vecteur. . .	89
5.1	Synoptique d'un nœud de la machine. . . . .	115
6.1	Différentes organisations possibles du contrôle de la machine. . . . .	121
6.2	Un séquenceur de processeurs virtuels maison. . . . .	124
6.3	Un séquenceur en tranche pour POMP. . . . .	125

6.4	Un séquenceur à base de MC88100 pour POMP. . . . .	127
6.5	Synoptique de la machine POMP. . . . .	128
7.1	Petit exemple en POMPC utilisant le <b>where</b> . . . . .	134
7.2	Dualité espace d'adressage-temps entre les branchements de machines MIMD et SIMD. . . . .	135
7.3	Pile d'activité et pile d'activité factorisée. . . . .	141
7.4	L'exemple de la figure 7.1 corrigé pour ne nécessiter que du contrôle local d'écriture en mémoire. . . . .	146
7.5	Exécution d'un branchement non retardé et retardé. . . . .	150
7.6	Utilisation des branchements non retardés pour réaliser le contrôle de flot parallèle. . . . .	151
7.7	Estimation qualitative de la perte d'efficacité due au pipeline. . . . .	153
7.8	Exemple de conditionnement parallèle à compiler. . . . .	154
7.9	Programme compilé sans et avec entrelacement. . . . .	154
7.10	Le même programme compilé sur une machine MIMD. . . . .	156
7.11	Exécution d'un test sur une machine non-parallèle VLIW. . . . .	157
7.12	Exécution d'un test parallèle sur une machine SIMD à PES VLIW. . . . .	158
8.1	Programme de calcul d'un ensemble de JULIA en POMPC. . . . .	164
8.2	Infrastructure générale de la chaîne de génération de code à partir de POMPC. . . . .	166
8.3	Algorithme de planification du code. . . . .	174
8.4	Routine <b>daxpy</b> de la bibliothèque BLAS1. . . . .	175
8.5	Allure des flots d'instructions appariés avant placement temporel. . . . .	177
9.1	Synoptique d'un réseau totalement connecté. . . . .	188
9.2	Synoptique d'un réseau hypercube. . . . .	189
9.3	Schéma d'une hypergrille de dimension 2 et d'un hypertore de dimension 2 (dessiné de 2 manières différentes). . . . .	190
9.4	Réalisation d'une communication sur grille sur un réseau de type grille. . . . .	190
9.5	Exemple de graphes de DE BRUÛN, $B(3,2)$ et $B(2,3)$ . . . . .	192
9.6	Réseau de type <i>cube-connected cycles</i> . . . . .	193
9.7	Synoptique d'un réseau à matrice de points de croisement ( <i>crossbar</i> ). . . . .	195
9.8	Synoptique d'une machine communiquant à travers un bus. . . . .	195
9.9	Dessin d'un réseau Oméga et <i>Indirect Binary n-Cube</i> . . . . .	196
9.10	Synoptique d'un réseau de CLOS à 3 étages. . . . .	198
9.11	Synoptique d'un réseau de BENEŠ. . . . .	198
9.12	Exemple d'interblocage. . . . .	201
9.13	Synoptique du réseau de POMP pour $N = 4$ et $k = 2$ . . . . .	205
9.14	Synoptique du réseau de POMP pour $N = 256$ et $k = 4$ . . . . .	206
9.15	Les 2 modes du réseau de POMP pour $N = 256$ et $k = 4$ . . . . .	207
10.1	Schéma de l'interface VME. . . . .	222
10.2	Schéma de la mémoire de données scalaire. . . . .	224
10.3	Schéma de la mémoire programme du processeur scalaire. . . . .	226
10.4	Schéma de la partie gérant le bus de contrôle. . . . .	228
10.5	Schéma de la gestion des exceptions. . . . .	230

10.6	Schéma de la gestion des opérateurs globaux. . . . .	233
10.7	Synoptique de l'interface vidéo de POMP. . . . .	236
10.8	Schéma de la carte d'un processeur élémentaire. . . . .	238
10.9	Schéma équivalent à la gestion de l'activité d'un PE. . . . .	240
10.10	Un extrait du journal de bord de la machine. . . . .	244
10.11	Synoptique du câblage de la machine. . . . .	245
10.12	Schéma d'implantation d'un PE à plat. . . . .	246
10.13	Schéma d'implantation d'un PE en module. . . . .	247
10.14	Synoptique du câblage du réseau entre carte dans le cas d'un hypercube. . . . .	248
10.15	Organisation de la machine dans sa baie triple Europe. . . . .	249
11.1	Modèle d'exécution sur machine séquentielle et parallèle. . . . .	257
11.2	Modèle d'exécution comparé avec une machine SIMD-VLIW. . . . .	259
11.3	Programme de copie d'une variable parallèle. . . . .	260
11.4	Programme de copie de vecteur sur une machine séquentielle. . . . .	260
11.5	Programme de copie de vecteur sur une machine SIMD. . . . .	260
11.6	Exemple sur une machine 1-SIMD-2-VLIW. . . . .	261
11.7	Architecture 4-VLIW-256-SIMD. . . . .	262
12.1	Synoptique du nœud SPMD avec mémoire dynamique. . . . .	271
12.2	Synoptique du nœud SPMD avec mémoire statique. . . . .	273
12.3	Synoptique du nœud SPMD avec mémoire statique entrelacée. . . . .	274
12.4	Synoptique du système de gestion mémoire du circuit de communication. . . . .	279
12.5	Principe et exemple de partitionnement. . . . .	283
12.6	Exemple de <i>et</i> global linéaire pipeliné. . . . .	288
12.7	Exemple de <i>ou</i> global linéaire partitionnable. . . . .	288
12.8	<i>Ou</i> global linéaire partitionnable symétrique. . . . .	289
12.9	<i>Ou</i> global linéaire partitionnable optimal. . . . .	290
12.10	Exemple de <i>ou</i> global linéaire partitionnable symétrique en 2D. . . . .	290
12.11	Opération préfixe parallèle symétrique. . . . .	291
12.12	Algorithme de synchronisation globale. . . . .	295
12.13	Barrière à planification statique. . . . .	296
12.14	Optimisation d'un cas d'imbrication de synchronisations. . . . .	298



# Liste des tableaux

3.1	Généralité contre contrôlabilité. . . . .	43
3.2	La notion de synchronisme dans les langages et les architectures parallèles. . . . .	44
3.3	Petit dictionnaire vectoriel-parallèle. . . . .	45
4.1	Opérateurs de contrôle de flot en POMPC. . . . .	83
4.2	Différentes sortes de communications en POMPC. . . . .	85
5.1	Complexité des opérations préfixes parallèles . . . . .	97
5.2	Comparatif entre quelques processeurs du marché. . . . .	109
6.1	Les instructions de séquençement. . . . .	120
7.1	Gestion des <code>where/elsewhere</code> avec un compteur. . . . .	137
7.2	Gestion du <code>whilesomewhere</code> avec un compteur. . . . .	138
7.3	Gestion du <code>switchwhere</code> avec un compteur. . . . .	140
7.4	Cas d'un <code>return</code> parallèle dans un conditionnement parallèle. . . . .	140
7.5	Fonctionnement de la pile lors d'une condition parallèle. . . . .	142
7.6	Fonctionnement de la pile lors d'une sortie de condition parallèle. . . . .	143
7.7	Complexité des méthodes de conditionnement SIMD. . . . .	143
7.8	Comparaison entre deux méthodes de conditionnement SIMD. . . . .	152
8.1	Quelques dépendances de données . . . . .	171
8.2	Placement temporel de la routine <code>daxpy</code> . . . . .	176
9.1	Complexité de quelques configurations du réseau hybride de POMP. . . . .	210
9.2	Routine d'émission de paquets sur le réseau à commutation de circuits. . . . .	212
9.3	Routine de réception de paquets sur le réseau à commutation de circuits. . . . .	213
9.4	Performances de la routine de gestion du réseau dynamique en Mo/s par processeur à 25 MHz. . . . .	214
9.5	Débit de quelques configurations statiques en Go/s. En italique est indiqué le débit par PE en Mo/s. . . . .	215
9.6	Performances du réseau dynamique en Go/s (performances par processeur en Mo/s, en italique) pour une fréquence $f_r$ de 25 MHz. . . . .	216
12.1	Comparaison entre les différents types de module mémoire. . . . .	274
12.2	Comparaison entre les méthodes de synchronisation. . . . .	293
12.3	Dépendances dans une machine MIMD. . . . .	294
12.4	Quelques comparaisons de puissance volumique crête. . . . .	303

# Index

En travaux... Pour l'instant seule la bibliographie est à peu près indexée. [?]



## Symboles

- écologie, [LG92]
  - écriture en mémoire
    - contrôle de l'exécution, 146
  - élimination
    - gaussienne, [Hel76]
  - émissions, 53
  - équation
    - aux dérivées partielles
    - résolution en parallèle, [BBJ<sup>+</sup>62]
  - évaluation
    - des performances, [FLW86,FLW87,FR72, Fly66,KMC72,Kuc76], 253–263
    - à taille de du problème fixe, [Amd67]
    - à taille mémoire fixe, [SN90]
    - à temps constant, [GREC91]
    - à temps d'exécution fixe, [GREC91, Gus88,SN90]
    - débranchements conditionnels et parallélisme, [RF72]
    - expressions arithmétiques, [Bre74]
    - problèmes de déplacement de données, [ACF92]
    - réévaluation de la loi d'Amdahl, [Gus88]
  - incrémentale
    - d'expressions affines, [FP81]
    - d'expressions quadratiques, [FPE<sup>+</sup>89, GF86]
  - parallèle
    - d'expressions arithmétiques générales, [Bre74]
    - évolution technologique, [Mar92a]
- 
- A**
- 
- accélération, 253, *voir* évaluation des performances
    - à taille du problème fixe, 254
    - à taille mémoire fixe, 256
    - à temps d'exécution fixe, 255
  - acquisition de données d'expériences, [Mar92a]
  - activité, 134, 140
  - Actus, [PCMP85,Per79]
  - ADA, [Tho86]
  - adaptation
    - d'impédance, 242
  - addition
    - en arithmétique binaire, [Rei60]
  - additionneur
    - complexité, [LF80]
  - adresse
    - physique, 275
    - virtuelle, 275
  - affichage, 15
  - algèbre fermionique, [CC92]
  - algorithme, [Tar83]
    - à parallélisme de données, [HS86]
    - graphique, [Ata89,Lep89]
    - recherche, [Ive62]
    - scan*, [Ive62]
    - Tomasulo* (de), [Tom67]
    - tri, [Ive62]
  - algorithmique
    - parallèle, [Ive62]
  - alignement
    - de tableaux, [CMZ92,FHK<sup>+</sup>92,ZBC<sup>+</sup>92], 68
  - alignement de données, [HKK<sup>+</sup>91]
  - allocation
    - de tâches
      - sur PASM, [TS85]
      - sur une machine VLIW, [BQW91]

allocation de la mémoire  
  sur machine SIMD  
  ILLIAC IV, [Kuc68]  
  sur machines SIMD  
  CM2, [KLGLS90,Sab92]  
Alpha, [Dig92a,Dig92b]  
Alto, 17  
AMT DAP, [AMT89]  
analyse  
  de dépendances  
  interprocédurale, [Bod90,IJT90,TIF86]  
  de flot de données, [Bod90]  
  dynamique  
  de dépendances, [LC91]  
  factorielle de données  
  parallélisation, [Rap83]  
  lexicale, [AU77]  
  syntaxique, [AU77]  
*anti-aliasing*, voir anticrénelage  
anticrénelage, 15  
AP1000  
  communications, [SHI92]  
APL  
  impression en  $\text{\LaTeX}$ , [GSMN90]  
APL, [Ber91]  
arbre  
  de recherche, [Tar83]  
  de tri, [Tar83]  
  recouvrant, [Tar83]  
Architecture  
  mémoire dynamique, [Adv90b]  
  MIMD, [RCCT90]  
architecture, **13–40**  
  graphique, [CKM91,Mat88]  
  cellulaire, [Ata89,Lep89]  
  de type bitmap, [Kor85]  
  ordinateurs  
  tutoriel, [GMSF87a,GMSF87b]  
  pour le traitement d'images, [Cas85]  
  processeur  
  MIPS, [HJG<sup>+</sup>82,Hen84]  
  RISC I, [PP82]  
arithmétique  
  binaire, [Rei60]  
ArMen, [FGP91,Fil91a,Fil91b,Fil92,Pot91]  
automate  
  à états finis, [AU77]  
  cellulaire  
  sur ArMen, [Fil91b]  
automate à états finis  
  complexité, [LF80]  
  opération préfixe parallèle  
  non associative, [RJH92]

autoplanification de boucle, [BP90]

---

**B**


---

barrière  
  de pipeline, 27  
  de synchronisation, [LB80]  
  autoplanification de boucle, [BP90]  
  floue, [GE89,GE90]  
  statique, [OD90a,OD90b]  
  sur machine à mémoire partagée, [Lub90]  
battage parfait, [Sto71]  
*benchmark*  
  à mémoire constante, [SN90]  
  à temps constant, [GREC91,SN90]  
  SLALOM, [GREC91]  
binaire  
  arithmétique, [Rei60]  
bit  
  d'activité, 134, 140, 147  
  factorisation, 141  
BITBLT, 17  
BLAZE, [KMR87]  
BLITZEN, [BDHR90]  
blocs alternatifs  
  terminaux  
  entrelacement, 153  
*bounded recurrence*, [Cal91]  
branchement, voir débranchement  
  conditionnel non retardé  
  pour le contrôle de flot SIMD, 149  
**break**, 138, 139  
bretzel liquide, voir le dessin en tête de l'index  
BSP, [KS82,LV82,Sto77]  
bus  
  de POMP, 244  
  Sun, 244  
  VME, [PLX88a,PLX88b,VME87], 221  
  VME, 244

---

**C**


---

C\*  
  ancienne version, [Thi87b]  
  nouvelle version, [Thi90]  
câble  
  coaxial plat, [Bel89]  
cache, 28  
calcul  
  déterministe, [PDGO87]  
  logique, [Ive62]  
  parallèle, [AJ88,HH80]  
  probabiliste, [PDGO87]

- unaire, [PDGO87]
  - Campus/800, [All91]
  - canaux
    - virtuels (roulage), [DS86,FGPS91,GN92]
    - multiples, [Dal92]
  - carré magique, [BJR88]
  - case, 139
  - CDC 6600, [Tho64]
  - Cedar, [GLK84]
  - chemin
    - le plus court, [Tar83]
  - chimie quantique, [CC92]
  - circuit
    - moulé, [Gla63]
  - circuits logiques
    - reconfigurables, [Pot91]
  - classification de données
    - parallélisation, [Rap83]
  - CM Fortran, [ALS90,Sab92]
  - CM-2, [Thi87a]
  - CM-2, [TR88]
  - CM-5, [Thi91]
  - code
    - de GRAY, [SS88]
  - cohérence de caches, [Cyp91,MOT88b]
  - collection, [BS90,Ble89a]
  - communication, 45, **52–61**
    - générale, 52
    - scalaire, **57**
    - sur grille, 54
  - commutation
    - virtual cut-through*, [KK79]
    - wormhole*, [DS87]
  - compaction
    - de microcode
      - comparaisons, [DLSM81]
      - globale de microcode, [Fis81]
  - comparaison
    - d'architectures RISC, [PW89]
    - langage
      - APL et Fortran 90, [Ber91]
  - compilateur
    - autovectorisant
      - Paraphrase, [KKLW80]
    - C, [OAS91]
    - de langage parallèle, [HQ91]
    - de silicium, [Par91]
    - SIMD, [PCMP85]
    - vectorel, [PCMP85]
  - compilation
    - avec planification statique
      - pour ordinateur MIMD, [ZDO90]
    - introduction, [AU77]
  - machines parallèles (pour)
    - mémoire distribuée, [McK92]
  - optimisation, [AU77]
  - pour machine SIMD
    - forall**, [ALS90]
  - pour machine VLIW, [BCW89]
  - pour machines SIMD
    - CM2, [KLGLS90]
  - pour machines MIMD, [DGNP88,HLJ+91, KMR87]
  - pour machines SIMD
    - CM2, [Sab92]
  - pour machines super-scalaires, [AN90]
  - pour machines VLIW, [AN90,BQW91]
- complexité
- additionneur, [LF80]
  - algorithmique, [Tar83]
  - automate à états finis, [LF80]
  - contrôle de flot
    - SIMD, 143
  - opération préfixe parallèle, [LF80], *voir*
    - opération préfixe parallèle non associative, [RJH92]
- PE algorithmique, 55
- compteur d'activité, [Ker92]
- concentration
  - associative, 55
- confluence, [Fly66,Fly72,Tho64], **27**
- connecteur
  - haute performance, [AUG89]
  - millimétrique, [DuP90]
- Connection Machine, [Hil85]
- continue**, 138, 139
- contrôle, **119–130**
  - de l'écriture en mémoire, [Ker92]
  - de l'exécution
    - écriture en mémoire, 146
    - d'instructions parallèles, 145
- contrôle de flot
  - SIMD, [Ker92]
  - SIMD, [BDW85,NFA+90], **133–159**
    - avec un branchement conditionnel non retardé, 149
  - complexité, 143
  - en mode SPMD, 148
  - pack* et *unpack*, [BS90]
  - pour machine MIMD, 144
- couplage, [Len78], **37**
  - faible, 266
  - fort, [BJR88,FJL85,KS82,LB80,LV82, Law75]
- CRAY Y-MP C90, [Cra91a]
- CRAY Y-MP EL, [Cra91b]

CRAY-1, [Rus78]  
 CRAY-X-MP-2, [Che83]  
 CYBER 205, [Lin82]

---

**D**

---

débranchement  
   conditionnel  
     non retardé, [Ker92]  
 débranchement conditionnel  
   inhibition du parallélisme, [RF72]  
 débranchements conditionnels et parallélisme,  
   *voir* évaluation des performances  
 décomposition  
   de données, [CMZ92,FHK<sup>+</sup>92,HKK<sup>+</sup>91,  
     ZBC<sup>+</sup>92]  
 décomposition semi-automatique de domaines,  
   [KMR87]  
 décompositions  
   de données, 68  
 dépendance, [DLSM81,Fis81,KMC72,Mar92b,  
   dD91]  
   analyse interprocédurale, [Bod90,IJT90,  
     TIF86]  
   entre données, 28, 170  
 Dataparallel C, [HQ91]  
*debug*, *voir* mise au point  
 DECChip 21064-AA, [Dig92a,Dig92b]  
 Delta, [Int91c]  
 diaphonie, 242  
 diffusion, 284  
 distribution  
   de tableaux sur des processeurs, [???,  
     CMZ92,FHK<sup>+</sup>92], [ZBC<sup>+</sup>92]  
 distribution de données, [HKK<sup>+</sup>91]  
 diversité génétique, [LG92]  
 division  
   en arithmétique binaire, [Rei60]  
 doacross, [Cyt86]  
 doublage récursif, [KS73,Sto73]  
*double buffering*, 15  
 dowhere, 139  
 DPS 7000, [Bul92]  
 DSM, [NL91]  
 DSP  
   DSP96002, [MOT89]  
 DSP96002, [MOT89]

---

**E**

---

efficacité, 253  
 elsewhere, 137  
 ensemble, [Tar83]

entrées-sorties  
   parallèles, [CMZ92,ZBC<sup>+</sup>92]  
 entrelacement  
   blocs alternatifs  
     terminaux, [Ker92], 153  
 EPEX/FORTRAN, [DGNP88]  
 étreinte mortelle, *voir* interblocage  
 EVA, [Mar92b], 77  
 everywhere, 84  
 everywhere, 136  
 exécution  
   spéculative, 147  
 expression arithmétique  
   générale  
     évaluation parallèle, [Bre74]  
     parallélisation, [KMC72]  
 externalité, [LG92]

---

**F**

---

factorisation  
   du bit d'activité, 141  
   LU, [Sto73]  
 ferrite  
   tore, [BC63]  
*fetch & op*, [GGK<sup>+</sup>80,Got84]  
*Fetch & Add*, 66  
 FLIP2, [Dou89]  
 flot, [Fly66,Fly72]  
 fluide  
   logique à base de fluide, [Gla63]  
 FMP, [LB80]  
 FMP Fortran, [LB80]  
 FORALL, 68, 69  
 forall  
   compilation pour machine SIMD, [ALS90]  
 FORTRAN  
   détermination du parallélisme dans des  
     programmes, [KMC72]  
   vectorel, [KS82]  
 Fortran 8x, [KLGLS90]  
 Fortran 90, [Ber91]  
 FORTRAN D, [HKK<sup>+</sup>91], 68  
 forwhere, 139  
 french.sty, [Gau92]  
 front de calcul, [KMC72]  
*fuzzy barrier*, [GE89,GE90]

---

**G**

---

génération  
   de code, [Sab92]  
 GAMMA 60, [Cur63]

GAMMA 60, [Bul57,Bul60]  
 GAMMA 60, 31, 34  
 GAPP, [NCR84]  
 gather  
   *gather*, **53**  
*gather*, 45, 133  
 gestion  
   de la mémoire, 275  
 get  
   *get*, **53**  
 GF11, [BDW85]  
*glitches*, 242  
 glossaire, 45  
 grain fin, **36**  
 grain moyen, **37**  
 graphe  
   de CAYLEY, [CCD<sup>+</sup>92]  
   de dépendance, [AU77,Kuc76]  
   de dépendances, 170  
 graphe de dépendance, [Mar92b,dD91]  
 GRAY  
   code, [SS88]  
 gros grain, **36**  
 GROSCH (loi de), [ED85]  
*ground bounce*, 242  
 groupe  
   linéaire, [CCD<sup>+</sup>92]

---

***H***

---

Hellena  
   langage vectoriel, [Jeg87]  
 hiérarchie mémoire, **27**  
 HPF, 69  
 HPF, 266  
 hypercube  
   propriétés, [SS88]

---

***I***

---

IBM 360/91, [Tom67]  
 ILLIAC IV, [BBK<sup>+</sup>68,Hor82]  
 imagerie médicale, [KPDL<sup>+</sup>79]  
 IMOGENE, [CKM91], 22  
 indirection (parallélisation en présence de),  
   [Bod90,LC91]  
 Infinity 90, [Enc91]  
 instruction  
   conditionnée, [Ker92]  
   SIMD  
     conditionnelle, 148  
 interblocage, [DS86,DS87,FGPS91,KS91a,KS91b,  
   LH91,ML89], [MS80]

routage  
   adaptatif plan, [CK92]  
   *turn model*, [GN92]  
 interprocédural  
   analyse et parallélisation, [Bod90,IJT90,  
     TIF86]  
 introduction, **3–11**  
   aux ordinateurs parallèles, [GP85]  
   aux réseaux de communication, [Fen81]  
   aux supercalculateurs, [KT80]  
   aux systèmes multiprocesseurs, [GP85]  
 ordinateur  
   multiprocesseur, [PHE77]  
   sur les réseaux de communication, [WyF84]  
 Iris, 18  
 iWarp, [Int88]

---

***J***

---

J, [Ber91]  
 jeu d'instructions  
   complexité et controverse, [CCYHJ<sup>+</sup>85]

---

***L***

---

langage, **63–90**  
 SIMD  
   Porta-SIMD, [Tuc90]  
 à parallélisme de données  
   C\* (ancienne version), [Thi87b]  
   C\* (nouvelle version), [Thi90]  
   Parallation LISP, [BS90]  
 ADA, [Tho86]  
 ALP, [OPP91]  
 APL, [Ber91]  
   sur ordinateur MIMDRP3, [JC91]  
 BLAZE, [MR87]  
 C, [KR88]  
 C\*, [HLJ<sup>+</sup>91]  
 CM Fortran, [ALS90,Sab92]  
 CM++, [Col90]  
 Concurrent Pascal, [Han75]  
 de VON NEUMANN, [Bac78]  
 EPEX/FORTRAN, [DGNP88]  
 EVA, [Mar92b], 77  
 fonctionnel, [Bac78,OPP91]  
 Fortran, 64–70  
 Fortran 8x, [KLGLS90]  
 Fortran 90, [Ber91,MR90]  
 HPF, 69  
 HPF, 266  
 J, [Ber91]  
 LSD2, [Mar92b]

- multiC, [Wav91]
  - parallélisme de données
    - APL, [Ive62]
  - parallèle
    - APL (impression), [GSMN90]
    - Dataparallel C, [HQ91]
    - FORTRAN D, [FHK<sup>+</sup>92,HKK<sup>+</sup>91], 68
    - MPP Fortran, [PM92]
    - Vienna FORTRAN, [CMZ92,ZBC<sup>+</sup>92], 69
  - POMPC, 79–90
  - PomPC, 265
  - POMPC, [Par92]
  - SIMD
    - Actus, [PCMP85,Per79]
    - ParallaxisParallaxis, [BBES92,Brä89]
    - TRANQUIL, [Kuc68]
  - SPMD
    - BLAZE, [KMR87]
    - FMP Fortran, [LB80]
  - vectorel
    - Actus, [Per79]
    - Hellena, [Jeg87]
  - LARC, [Cur63]
  - L<sup>A</sup>T<sub>E</sub>X et APL, [GSMN90]
  - liaisons séries à haut débit, [LEH<sup>+</sup>89,Sai91]
  - linéarité, 253
  - logique
    - à base de fluide, [Gla63]
    - asynchrone, [DS86]
    - cellulaire, [KPDL<sup>+</sup>79]
    - en circuits magnétiques, [BC63]
    - self-timed*, [DS86]
  - loi
    - d'AMDAHL, [Amd67]
    - d'AMDAHL, 254
    - de GROSCH, [ED85]
    - de MAXWELL, 242
  - LONTALK, [Ech91,Ech92]
  - LSD2, [Mar92b]
  - LU, voir factorisation LU
- 
- M**
- M3S, [LEH<sup>+</sup>89,Sai91]
  - Mémoire, [Mic91]
  - mémoire
    - cache, 28
    - d'écran, 16
    - hiérarchie, 27
    - parallèle
      - à transfert par bloc, [Sto70]
    - partagée, [LEH<sup>+</sup>89,Sai91]
  - statique
    - synchrone, [Hit91]
  - vidéo-RAM, 16, 22
  - virtuellement distribuée, [NL91]
  - Machine
    - SIMD
      - BLITZEN, [BDR87]
  - machine
    - Campus/800, [All91]
    - Delta, [Int91c]
    - Infinity 90, [Enc91]
    - MEGA, [GR89]
    - MIMD, [All91,Enc91,Int91c]
    - POMP, [Ker89]
    - systolique
      - iWarp, [Int88]
      - PCS, [Rou90]
  - magnétique
    - logique en circuits magnétiques, [BC63]
  - masquage, [Ker92]
  - MAXWELL
    - loi de, 242
  - MC88100, [MOT88a,Mel89]
  - MC88200, [MOT88b]
  - microcode, [BCW89], 120
    - optimisation
      - comparaisons, [DLSM81]
      - optimisation globale, [Fis81]
  - microprogrammation, [Ive62]
  - MIMD
    - MIMD, 30
    - MIMD, [Fly66,Fly72]
    - MIMD, 25
  - MISD
    - MISD, 32
    - MISD, [Fly66]
    - MISD, 25
  - mise au point
    - de programmes en FORTRAN sur machine à mémoire partagée, [AP87]
  - MMU, 275
    - MC88200, [MOT88b]
  - MOD2MAG, [Par91]
  - modèle
    - à parallélisme de données, [HQ91]
    - CSP, [Hoa78]
    - de programmation
      - parallélisme de données, [Ble89a]
      - parallélisme de données asynchrone avec vérification des effets de bord à l'exécution, [Ste90]
      - parallèle, [Kar87]
      - parallèle orienté objet, [Kil91]

parallélisme de données, [Kil91,Mar92b,Par92]  
 SPMD, [DGNP88,HLJ+91,KMR87]  
 de VON NEUMANN, [Bac78]  
 SPMD, [CMZ92]  
 modèle de programmation, **43–61**  
 MP-1, [Bla90a,Bla90b,Dig91]  
 MPP Fortran, [PM92]  
 multiplication  
 en arithmétique binaire, [Rei60]  
 multiprocesseur, [All91,Cur63,Enc91]  
 multirocesseur, [Cyp91]

---

$\mathcal{N}$

---

NEURON  
 circuits 3120 & 3150, [Ech92]  
 notation  
 LUKASIEWICZ, [Ive62]  
 polonaise, [Ive62]  
 NYU Ultracomputer, [GGK+80,Got84]

---

$\mathcal{O}$

---

ombrage, 15  
 de GOURAUD, 15  
 de PHONG, 15  
 opération  
 parallèle préfixe, [Ive62]  
 préfixe  
 parallèle, [Kuc76]  
 préfixe parallèle  
 segmentée, [BS90]  
 opération préfixe parallèle, [Ble89a,Ble89b,Cal91,HS86], [KMC72,KRS85,KS73,Kog74a,Kog74b,Kog81,Sto73], **58**  
 complexité, [LF80]  
 non associative  
 complexité, [RJH92]  
 réalisation, [Ble89a,Ble89b]  
 segmentée, [Ble89a,Ble89b]  
 sur des listes, [HS86,KRS85]  
 optimisation  
 code pour ordinateur parallèle, [KMC72]  
 code pour processeur pipeline, [GM86]  
 compilation, [AU77]  
 des communications, [KLGLS90]  
 microcode  
 comparaisons, [DLSM81]  
 optimisation globale  
 microcode, [Fis81]  
 Ordinateur  
 Monarch, [RCCT90]

OPSILA, [Gal85,Gou82]  
 ordinateur  
 MIMD  
 Pixel Machine, 22  
 SIMD  
 BSP, [LV82]  
 taxinomie, [Tuc90]  
 confluent  
 CDC 6600, [Tho64]  
 IBM 360/91, [Tom67]  
 déterministe, [PDGO87]  
 de VON NEUMANN, [Bac78]  
 graphique, [Cla82,HH80,Mat88], 13–23  
 Alto, 17  
 FLIP2, [Dou89]  
 IMOGENE, [CKM91], 22  
 Iris, [AJ88,Cla82], 18  
 Pixel Machine, 22  
 PIXEL-PLANE, [FP81,FPE+89,GF86], 20  
 LIW  
 SMAL, [ZD91]  
 massivement parallèle  
 développement, [Mye91]  
 MIMD  
 AP1000 (communications), [SHI92]  
 MIMD, [Cur63]  
 Bull DPS 7000, [Bul92]  
 GAMMA 60, [Cur63]  
 GAMMA 60, 31, 34  
 LARC, [Cur63]  
 NYU Ultracomputer, [GGK+80,Got84]  
 PILOT, [Cur63]  
 PIXEL-PLANE, [FPE+89]  
 RP3, [JC91], 66  
 RW-400, [Cur63]  
 temps réel, [ABT82]  
 MIMD  
 M3S, [LEH+89,Sai91]  
 MIMD  
 Cedar, [GLK84]  
 MIMD multiplexé  
 GAMMA 60, [Bul57,Bul60]  
 multiprocesseur  
 introduction, [GP85,PHE77]  
 OPSILA, [Aug85]  
 parallèle, 23–40  
 IMOGENE, [CKM91], 22  
 performance, [DKMS90]  
 pipeliné, [Kog74a,Kog81]  
 prix et performances, [ED85]  
 probabiliste, [PDGO87]  
 SIMD, [Ung58]

- AMT DAP, [AMT89]  
 BLITZEN, [BDHR90]  
 BSP, [KT80]  
 BSP, [KS82,Sto77]  
 CM-2, [TR88,Thi87a]  
 CM-5, [Thi91]  
 Connection Machine, [Hil85]  
 DAP, [Fla90]  
 GF11, [BDW85]  
 ILLIAC IV, [BBK<sup>+</sup>68,Hor82,Kuc68]  
 MP-1, [Bla90a,Bla90b,Dig91]  
 PIXEL-PLANE, [FP81,FPE<sup>+</sup>89,GF86],  
   20  
 POMP, [Dou89,Ker92]  
 PROPAL 2, [Rap83]  
 PROPAL II, [RJ83]  
 SMAL, [ZD91]  
 SOLOMON, [BBJ<sup>+</sup>62,SBM62]  
 temps réel, [ABT82]  
 UNIFIELD I, [PDGO87]  
 WaveTracer DTC, [Jac90]  
 SPMD, [GE89,GE90], **265–303**  
   ArMen, [FGP91,Fil91a,Fil91b,Fil92,  
   Pot91]  
   FMP, [LB80]  
   PASM, [FCS88,NFA<sup>+</sup>90,SSDK84,SSKD87,  
   TS85]  
   PASM, [KNS91]  
 taxinomie, [Fly66,Fly72]  
 Thémis, 13  
 vectoriel, [HSN81], **34**  
   conception, [Lin82]  
   CRAY Y-MP C90, [Cra91a]  
   CRAY Y-MP EL, [Cra91b]  
   CRAY-1, [KT80]  
   CRAY-1, [Rus78]  
   CRAY-X-MP-2, [Che83]  
   CYBER 205, [Lin82]  
   Cyber 205, [KT80]  
   SX-3R, [NEC91]  
   VP-100/200, [MU84]  
 STAR-100, [HT72]  
 VLIW, [dD91]  
   StaCS 2.2, [dD91]  
 ZX81, 17  
 ordonnancement, [dD91]  
 ordonnancement d'instructions  
   dans un pipeline, [GM86]  
   pour processeur superscalaire, [FR72]  
   pour processeur VLIW, [DLSM81,Fis81]  
*overshoot*  
   avec les PALS, 242  
*owner*
- compute*  
   *rule*, voir règle des écritures locales
- 
- P***
- 
- pack*, 133  
 Palette  
   graphique, 234  
 Paraphrase, [KKLW80]  
 parallélisation, [Kuc76]  
   automatique, [McK92]  
   d'expressions arithmétiques, [KMC72]  
   de boucles avec pointeur ou indirection,  
     [Bod90,LC91]  
   de boucles **DO**, [KMC72]  
   de récurrence, [Cal91,KMC72]  
   interactive, [McK92]  
   interprocédurale, [McK92]  
   pour machines MIMD  
     **doacross**, [Cyt86]  
 parallélisation automatique, [dD91]  
   interprocédurale, [IJT90,TIF86]  
 parallélisme, [Kuc76,PKB<sup>+</sup>91], 23–40  
   à grain fin, 36  
   à grain moyen, **37**  
   à gros grain, **36**  
   dans les programmes FORTRAN, [KMC72]  
   de données, [HKK<sup>+</sup>91,OPP91]  
     FORTRAN D, [FHK<sup>+</sup>92]  
   de tâches, [Hoa78,Tho86]  
   introduction, [GP85]  
   modèle de programmation, [Kar87]  
 parallélisme de données, [Ble89a]  
   algorithmes, [HS86]  
   asynchrone  
     vérification des effets de bord à l'exé-  
     cution, [Ste90]  
   imbriqué  
     compilation pour machines SIMD, [BS90]  
 parallèle préfixe, voir opération préfixe pa-  
   rallèle  
 Parallation LISP, [BS90]  
   compilation pour machines SIMD, [BS90]  
 ParallaxisParallaxis, [BBES92,Brä89]  
 ParaScope, [HKK<sup>+</sup>91]  
 partitionnement  
   d'un ordinateur, 267  
 PASM, [FCS88,NFA<sup>+</sup>90,SSDK84,SSKD87,TS85]  
 PCS, [Rou90]  
 percolation  
   de code, [FR72]  
*perfect shuffle*, voir battage parfait  
 performance

- évaluation, *voir* évaluation des performances
  - graphique, 15
  - machine scalaire contre machine parallèle, [Amd67]
  - ordinateur
    - parallèle, [DKMS90]
  - performances, [Fly72]
  - permutations courantes, [Len78]
  - physique
    - des particules, [Mar92a]
  - PILOT, [Cur63]
  - pipeline, [Fly66,Fly72,Kog74a,Kog81], **27**
    - graphique, 18
    - logiciel, [AN90]
    - optimisation du code, [GM86]
    - ordonnancement d'instructions, [GM86]
  - PIPS, [IJT90]
  - pixels*, 15
  - Pixel Machine, 22
  - PIXEL-PLANE, [FP81,FPE<sup>+</sup>89,GF86], 20
  - planification
    - statique
      - pour ordinateur MIMD, [ZDO90]
  - pointeur (parallélisation en présence de), [Bod90, LC91]
  - POMP, [Dou89,Ker92]
  - POMPC, *voir* Langage POMPC
  - PomPC, 265
  - Porta-SIMD, [Tuc90]
  - préfixe, *voir* opération préfixe parallèle
  - prix de ordinateurs et performances, [ED85]
  - Processeur
    - AMD29000, [Adv88b]
    - AMD29050, [Adv90a]
    - hyperstone, [Hyp90]
    - i860, [INT89a,INT89b,Int91b,KM89]
    - R3000, [Int90,NEC88]
    - scalaire, 223
    - SPARC, [Cyp89,Cyp90b]
    - T9000, [INM91]
    - Transputer, [INM89]
  - processeur
    - 88110, [Cel91]
    - Alpha, [Dig92a,Dig92b]
    - comparaison, [Sla92]
    - comparaisons, [PW89]
    - de tableaux, [BBK<sup>+</sup>68,Hor82]
    - de traitement du signal
      - DSP96002, [MOT89]
    - DECChip 21064-AA, [Dig92a,Dig92b]
    - flottant, [OMMN90]
    - géométrique, [Cla82], 13
    - i860, [PW89]
    - MC88100, [MOT88a,Mel89,PW89]
    - R4000, [Int91a,Sie91]
    - processeur
      - R4000, [Cel91]
      - RS6000, [IBM90,OMMN90]
    - scalaire, 268
    - SIMD
      - GAPP, [NCR84]
      - SPARC, [Cel91,PW89]
    - superscalaire, [IBM90,OMMN90,RF72, TF70]
    - unaire, [PDG087]
    - Viking, [Cel91]
    - virtuel, [Fla90]
  - Processeurs élémentaires
    - de machine SIMD
      - BLITZEN, [BDR87]
  - processus légers, [PKB<sup>+</sup>91]
  - programmation
    - fonctionnelle, [Bac78]
  - PROPAL 2, [Rap83]
  - PROPAL II, [RJ83]
- 
- $\mathcal{R}$**
- R4000, [Int91a,Sie91]
  - réalisation, **221–251**
  - réception, **53**
  - récurrance, [Cal91,KMC72,KS73,Kog74a,Kog74b, Kog81]
    - linéaire, [Hel76,Sto73]
  - récurrance linéaire, [KS82]
  - récurrances
    - linéaires, [Kuc76]
  - réduction, [Ive62]
    - cyclique
      - mesure de convergence, [Hel76]
  - Réseau
    - aleatoire, [CCD<sup>+</sup>92]
    - autosynchrone, [RCCT90]
    - Beneš, [Gou82]
    - cube-connected cycles (CCC), [PV81]
    - FLIP, [Bat76]
    - multi-étage, [Bat76,RCCT90,WF81]
    - Indirect Binary n-Cube*, [Pea77]
    - Omega, [Gou82]
    - Shuffle Exchange*, [WF81]
  - réseau, **184–218**
    - à matrice de points de croisement, 194
    - autoroutant, [BR88,KO90]
    - banian, [GL73]
    - battage parfait, [Sto71]

- Beneš, [BDW85,Len78]
    - autoroutant, [BR88]
    - contrôle des permutations courantes, [Len78]
  - BENEŠ, [Ben62]
  - CLOS, [Clo53]
  - commutateur à croisillons, 194
  - crossbar*, [FWT82], 194
  - data manipulating functions* (DMF), [Fen74]
  - de tri, [RJ83]
  - de tri bitonique, [Bat68]
  - dynamique
    - à chemins redondants, [PR82]
    - gamma, [PR82]
    - Extra Stage Cube*, [GBAS82]
    - fat-tree*, [Lei85]
    - grille, [GR89]
    - hypercube, [KS91a,KS91b]
      - propriétés, [SS88]
    - hypergrille, [DS87,Dal92,FGPS91,KS91a,KS91b,LH91], [ML89]
      - routage adaptatif plan, [CK92]
  - IADM
    - routage DTA, [RFS88]
  - interblocage, **201–204**
  - LON TALK, [Ech91,Ech92]
  - multi-étage, [AP91a,Bat68]
  - multiétage, [Ben62,Clo53,Dal92,FWT82,Fen74,GBAS82], [GL73,Kot87,Law75,RFS88,WF80]
    - autoroutant, [BR88,KO90]
    - cube généralisé, [SSKD87]
    - extra stage cube*, [SSKD87]
  - Oméga, [LB80,Law75]
  - partitionnement, [FWT82]
  - performance, [AP91a,RS83]
    - avec nombre de pattes contraint, [AP91b,FWT82]
  - recombinant, [GGK<sup>+</sup>80,Got84], 53
  - shuffle exchange*
    - autoroutant, [BR88]
  - statique, [RS83]
  - tore 2D
    - expérience, [SHI92]
  - virtuel, [LH91]
  - réseaux
    - delta, [Pat81]
    - multi-étage, [Pat81]
    - performances
      - comparaison entre delta et matriciel, [Pat81]
  - racine carrée
    - en arithmétique binaire, [Rei60]
  - radiosité, [SP89]
  - rapport
    - coût/performance, [DKMS90]
  - réception
    - associative, 55
  - recouvrement des sections SIMD et séquentielles, [KNS91]
  - recursive doubling*, [KS73,Sto73]
  - réduction
    - associative, [Kuc68], 55
  - Registre
    - CONTROLE\_0**, 223, 225
  - renommage de registres, [OMMN90]
  - ressources
    - vivantes, [LG92]
  - return**
    - parallèle dans un conditionnement parallèle, 139
  - routage
    - adaptatif, [FGPS91,GN92,KS91a,KS91b,LH91]
    - adaptatif plan, [CK92]
    - aléatoire, [GR89,KS91a,KS91b]
    - algorithme du *e-cube*, [DS87]
    - bounding-box, [ML89]
    - canaux
      - virtuels, [DS86]
    - déterministe, [DS86,DS87]
    - DTA
      - pour réseau IADM, [RFS88]
    - forcé, [GR89]
    - non adaptatif, [DS86,DS87]
    - partial virtual cut-through*, [AP91b]
    - store and forward*, [MS80]
    - sur tore, [DS86]
    - tolérant aux fautes, [LH91]
    - turn model*, [GN92]
    - virtual cut-through*, [AP91b,KK79]
    - wormhole*, [DS87,Dal92,FGPS91,LH91]
  - wormhole*, [DS86]
  - RP3, [JC91], 66
  - RS6000, [IBM90,OMMN90]
  - runtime*, voir routines d'exécution
  - RW-400, [Cur63]
- **S**


---
- saut, voir débranchement
  - scan
    - segmenté, [BS90]
  - scan*, voir opération préfixe parallèle
  - scatter
    - scatter*, **53**

- scatter*, 45, 133  
*scheduling*, voir planification  
*scoreboard*, [Tho64]  
 send  
     *send*, **53**  
 SIMD  
     SIMD, **29**  
 SIMD, [Fly66,Fly72,KNS91]  
 SIMD, 25  
 simulateur  
     niveau comportemental, [Par91]  
     niveau interrupteur, [Par91]  
     niveau transistor, [Par91]  
 Simulation  
     de machines SIMD  
         BLITZEN, [RB89]  
 SISD, [Fly66,Fly72], **27**  
 SISD, 25  
*slice-wise*, [Sab92]  
*smart memory*, [AJ88,HH80], 18  
 SOLOMON, [BBJ<sup>+</sup>62,SBM62]  
 soustraction  
     en arithmétique binaire, [Rei60]  
 SPMD  
     SPMD, **33**  
 SPMD, [KNS91], 266  
 STAR-100, [HT72]  
 stockage de tableaux, [BJR88,FJL85,KS82,  
     Kuc68,LB80,LV82], [Law75]  
*strip-mining*, 46  
 structure  
     de données, [Tar83]  
*summary graph*, [Bod90]  
 Supercalculateur  
     introduction, [Cor91]  
 supercalculateur, [Che83,Cra91a,Cra91b,HSN81,  
     HT72,Lin82], [MU84,NEC91,Rus78]  
     comparaison, [KT80]  
     introduction, [CK91,KT80]  
 superlinéarité, 253–263  
     exemple, [KNS91]  
     impossible, [FLW86,FLW87]  
     possible, [ACF92,GREC91,SN90]  
 Superordinateurs  
     tutoriel, [Hwa84]  
 superscalaire, [RF72,TF70]  
 surrelaxation  
     pour la résolution d'équation au déri-  
         vées partielles, [BBJ<sup>+</sup>62]  
 SVM, [NL91]  
*switchwhere*, 139  
 SX-3R, [NEC91]  
 synchronisation, [GE89,GE90], 266  
     globale, [Fil91a,Fil91b,Fil92,LB80]  
     SIMD du code, 168  
 synchroniseur, [Fil91a,Fil91b,Fil92]  
 synthèse  
     logique, [Fil91b]  
 synthèse d'image  
     architecture, [Ata89,Lep89]  
     radiosité, [SP89]  
 système  
     d'exploitation  
         comparaison entre MACH, TROLLIUS  
             et HELIOS, [Fil91b]  
     linéaire tridiagonal  
         doublage récursif, [Sto73]  
         réduction cyclique, [Hel76]  
     mémoire, [BJR88,FJL85,LB80,Law75]  
 système mémoire, [KS82,LV82,Len78,Sto70,  
     dD91]  
 systolique, [BCW89,Int88,Rou90]  
 système d'exploitation, [PKB<sup>+</sup>91]
- 
- T*
- 
- T9000, [INM91]  
 table  
     des symboles, [AU77]  
 tas, [Tar83]  
 taxinomie, [Fly66,Fly72,Sny88]  
     des langages parallèles, [Mar92b]  
     des ordinateurs SIMD, [Tuc90]  
 temps  
     utilisateur  
         mesure, 151  
*Threads*, [PKB<sup>+</sup>91]  
*Tomasulo*  
     algorithme de, [Tom67]  
 tore  
     ferrite, [BC63]  
*trace scheduling*, [Fis81]  
 Traitement  
     d'images, [KPDL<sup>+</sup>79]  
 traitement  
     d'image, [SSKD87]  
 TRANQUIL, [Kuc68]  
 tri, 60  
     bitonique, [Bat68]  
     parallèle, [Ble89a,Ble89b]  
     pour machine synchrone, [RJ83]  
     sur machine SIMD, [Rap83]  
 tutoriel  
     sur l'architecture des ordinateurs, [GMSF87a]

---

**u**

---

unaire  
  processeur, [PDGO87]  
UNIFIELD 1, [PDGO87]  
*unpack*, 133

---

**v**

---

vérification des effets de bord à l'exécution  
  parallélisme de données  
    asynchrone, [Ste90]  
vecteur  
  de dépendance, [KMC72]  
  de masque, 145  
vidéo-RAM, 16, 22  
Vienna FORTRAN, [CMZ92,ZBC<sup>+</sup>92], 69  
*virtual cut-through*, [KK79]  
VLIW  
  ordonnancement d'instructions, [DLSM81,  
    Fis81]  
*vp\_ratio*, 46  
VP-100/200, [MU84]

---

**w**

---

WaveTracer DTC, [Jac90]  
**where**, 133–159  
**whilesomewhere**, 138, 139  
*wrapping*, 242–243

---

**x**

---

*XOR-scheme*, [FJL85]

---

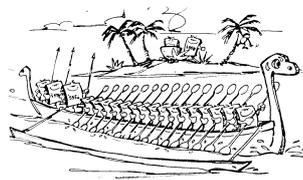
**z**

---

ZX81, 17

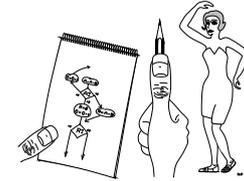


# Table des matières

<b>De quelques remerciements</b>		III
<b>1 Introduction</b>		<b>2</b>
1.1 Le développement des ordinateurs dits « parallèles »		3
1.2 Les origines du projet POMP		7
1.3 Plan du mémoire de thèse		8
1.4 But de cet ouvrage		10
1.5 Typographie du mémoire de thèse		11
<b>2 Des architectures graphiques au parallélisme</b>		<b>12</b>
2.1 Les machines graphiques		13
2.1.1 La problématique		13
2.1.2 Quelques solutions		15
2.1.2.1 Un matériel très réduit		16
2.1.2.2 Les processeurs spécialisés de type BITBLT		17
2.1.2.3 IRIS		18
2.1.2.4 La machine Pixel-Plane		20
2.1.2.5 La PIXEL MACHINE		22
2.2 Les machines parallèles		23
2.2.1 La nature du parallélisme		25
2.2.2 SISD		27
2.2.3 SIMD		29
2.2.4 MIMD		30
2.2.5 MISD		32
2.2.6 SPMD		33
2.2.7 Les ordinateurs vectoriels		34
2.3 Le grain du parallélisme		35
2.3.1 Gros grain		36
2.3.2 Grain fin		36

2.3.3	Grain moyen . . . . .	37
2.4	Le couplage entre processeurs et mémoires . . . . .	37
2.4.1	Contre-danse anglaise : couplage fort . . . . .	38
2.4.2	Boudoir . . . . .	39
2.5	Conclusion . . . . .	40

### 3 Le modèle de programmation



3.1	Au sujet de quelques problèmes de langage . . . . .	45
3.2	Quelques modèles de programmation . . . . .	46
3.2.1	Le parallélisme de tâches . . . . .	46
3.2.2	Le modèle flot de données . . . . .	47
3.2.3	Le parallélisme de données . . . . .	47
3.2.4	Le modèle SPMD . . . . .	48
3.3	Un modèle de programmation pour POMP . . . . .	48
3.3.1	Parallélisme de données . . . . .	48
3.3.2	Virtualisation . . . . .	49
3.4	Parallélisme orienté objet . . . . .	50
3.5	Contrôle de flot . . . . .	51
3.5.1	Contrôle de flot scalaire . . . . .	51
3.5.2	Le contrôle de flot parallèle : une dessynchronisation bornée . . . . .	51
3.6	Interactions complexes = communications . . . . .	52
3.6.1	Communications générales . . . . .	52
3.6.1.1	Émissions . . . . .	53
3.6.1.2	Réceptions . . . . .	53
3.6.2	Communications sur grille . . . . .	54
3.6.3	Concentrations ou réductions associatives . . . . .	55
3.6.4	Communications scalaires . . . . .	57
3.6.5	Opérations parallèles préfixes . . . . .	58
3.7	Conclusion . . . . .	61

### 4 Le langage : POMPC



4.1	Les langages de programmation parallèle . . . . .	63
4.1.1	Les langages de type Fortran . . . . .	64
4.1.1.1	Fortran 77 . . . . .	65
4.1.1.2	HEP Fortran . . . . .	65
4.1.1.3	EPEX/Fortran . . . . .	65
4.1.1.4	Fortran 90 . . . . .	66
4.1.1.5	CM-Fortran . . . . .	68
4.1.1.6	Fortran D . . . . .	68

4.1.1.7	Vienna Fortran . . . . .	69
4.1.1.8	HPF . . . . .	69
4.1.2	Les langages de type Pascal . . . . .	70
4.1.2.1	Concurrent Pascal . . . . .	70
4.1.2.2	Actus . . . . .	70
4.1.2.3	BLAZE . . . . .	71
4.1.2.4	Hellena . . . . .	71
4.1.3	APL . . . . .	72
4.1.4	Les langages de type C . . . . .	73
4.1.4.1	Le vieux C* . . . . .	73
4.1.4.2	DPC . . . . .	74
4.1.4.3	Le nouveau C* . . . . .	75
4.1.4.4	MultiC . . . . .	76
4.1.4.5	MPL . . . . .	77
4.1.4.6	EVA . . . . .	77
4.1.5	Les langages de type C++ . . . . .	78
4.1.5.1	Porta-SIMD . . . . .	78
4.1.5.2	CM++ . . . . .	78
4.1.6	Parallaxis : un langage basé sur Modula 2 . . . . .	78
4.2	POMPC . . . . .	79
4.2.1	L'expression du parallélisme : les collections . . . . .	80
4.2.2	Le contrôle de flot parallèle . . . . .	82
4.2.3	Les communications . . . . .	84
4.2.4	Placement des données . . . . .	86
4.2.5	Les fonctions parallèles . . . . .	87
4.2.6	Notions de virtualisation . . . . .	88
4.2.7	Les bibliothèques . . . . .	89
4.3	Conclusion . . . . .	90

## 5 Les processeurs élémentaires



5	<b>Les processeurs élémentaires</b> . . . . .	<b>92</b>
5.1	Caractéristiques générales . . . . .	93
5.1.1	Largeur des processeurs . . . . .	93
5.1.1.1	Efficacité calculatoire . . . . .	94
5.1.1.2	Efficacité algorithmique . . . . .	96
5.1.1.3	Utilisation de la mémoire . . . . .	98
5.1.1.4	Importance du débit mémoire et utilisation des registres . . . . .	99
5.1.1.5	Adressage local et global de la mémoire locale . . . . .	101
5.1.1.6	Coût du réseau . . . . .	103
5.1.1.7	Coût de la machine . . . . .	103
5.1.2	Coût technologique et balance processeurs-mémoire . . . . .	104
5.1.3	Caractéristiques SIMD . . . . .	105
5.1.4	Caractéristiques supplémentaires . . . . .	105
5.1.5	Conclusion sur les caractéristiques . . . . .	106

5.2	Le développement d'un processeur : un mal nécessaire? . . . . .	106
5.2.1	Processeurs en tranche . . . . .	108
5.2.2	Processeurs de type RISC . . . . .	108
5.3	Le module processeur élémentaire de POMP . . . . .	111
5.3.1	Le système mémoire . . . . .	111
5.3.2	Le MC88100 . . . . .	112
5.3.3	Les diffusions scalaires . . . . .	112
5.3.4	Le circuit de contrôle et de communication . . . . .	113
5.3.4.1	La gestion de l'activité . . . . .	113
5.3.4.2	Les entrées-sorties parallèles . . . . .	113
5.3.4.3	Les réceptions scalaires . . . . .	114
5.3.4.4	La gestion des exceptions . . . . .	114
5.4	Conclusion . . . . .	115

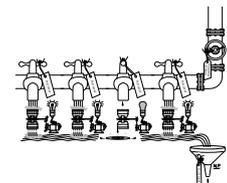
## 6 Le contrôle de la machine



118

6.1	Les dispositifs du contrôle . . . . .	119
6.1.1	Le processeur scalaire . . . . .	119
6.1.2	Le séquenceur . . . . .	119
6.1.3	Le bus d'émission scalaire . . . . .	120
6.1.4	La récupération d'une valeur globale . . . . .	121
6.2	Quelques combinaisons utilisées . . . . .	121
6.2.1	L'ILLIAC IV . . . . .	122
6.2.2	Opsila . . . . .	122
6.2.3	MasPar MP-1 . . . . .	122
6.2.4	CM-2 . . . . .	123
6.2.5	MPP . . . . .	123
6.2.6	WaveTracer DTC . . . . .	123
6.3	Le contrôle de la machine POMP . . . . .	123
6.3.1	L'hôte et un séquenceur . . . . .	124
6.3.1.1	Un séquenceur câblé . . . . .	124
6.3.1.2	Un séquenceur en tranche . . . . .	125
6.3.2	Un processeur standard . . . . .	126
6.4	Conclusion . . . . .	129

## 7 Le contrôle de flot SIMD



132

7.1	La gestion du contrôle de flot SIMD . . . . .	134
7.1.1	Décision locale d'exécution d'instructions . . . . .	134
7.1.2	Vision conceptuelle d'un branchement SIMD . . . . .	134
7.1.3	Le compteur d'activité . . . . .	136

7.1.3.1	Principe . . . . .	136
7.1.3.2	Réalisation des structures de contrôle parallèle de POMPC	136
	Compilation d'un <code>where/elsewhere</code> . . . . .	137
	Compilation d'un <code>whilesomewhere</code> et du <code>forwhere</code> . . . . .	138
	Compilation d'un <code>switchwhere</code> . . . . .	139
	Cas du <code>return</code> parallèle . . . . .	139
7.1.4	Un bit d'activité . . . . .	140
	7.1.4.1 Factorisation . . . . .	141
	7.1.4.2 Equivalence entre pile et compteur d'activité . . . . .	142
7.1.5	Arguments de choix . . . . .	143
7.1.6	Problématique dans le cas du MIMD . . . . .	144
7.2	Méthodes de contrôle des instructions parallèles . . . . .	145
	7.2.1 Le contrôle direct de l'exécution . . . . .	145
	7.2.2 Décision locale d'écriture en mémoire locale . . . . .	146
	7.2.3 Utilisation de l'adressage local . . . . .	147
	7.2.4 Introduction d'instructions SIMD conditionnelles . . . . .	148
	7.2.5 Passage en mode SPMD . . . . .	148
	7.2.6 Utilisation des caractéristiques des processeurs modernes . . . . .	149
	7.2.6.1 Branchements conditionnels non retardés . . . . .	149
	7.2.6.2 Entrelacement de blocs alternatifs terminaux . . . . .	153
	7.2.6.3 Utilisation d'un processeur élémentaire VLIW . . . . .	155
7.3	Conclusion . . . . .	158

## 8 La génération du code



160

8.1	Virtualisation . . . . .	161
	8.1.1 Virtualisation en largeur d'abord . . . . .	162
	8.1.2 Virtualisation en profondeur d'abord . . . . .	163
8.2	L'infrastructure de génération du code . . . . .	165
8.3	Les problèmes de synchronisation . . . . .	167
	8.3.1 Synchronisation scalaire-parallèle . . . . .	167
	8.3.2 Synchronisation SIMD . . . . .	168
	8.3.2.1 Graphes de dépendances et exécution . . . . .	170
	8.3.2.2 Algorithme d'allocation temporelle . . . . .	172
	8.3.3 Le placement temporel du code global . . . . .	177
8.4	L'environnement de programmation . . . . .	178
	8.4.1 La mise au point des programmes . . . . .	178
	8.4.1.1 L'hôte est le processeur scalaire $\rightsquigarrow$ <code>dbxtool</code> . . . . .	179
	8.4.1.2 Pour POMPC : <code>bugtool</code> . . . . .	180
	8.4.2 Intégration dans l'hôte : point de vue système . . . . .	180
8.5	Conclusion . . . . .	181

<b>9</b>	<b>Le réseau d'interconnexion</b>	<b>184</b>
9.1	Introduction . . . . .	184
9.2	Les grandes classes de réseaux . . . . .	187
9.2.1	La commutation de paquets : un réseau statique . . . . .	187
9.2.1.1	Réseau totalement connecté . . . . .	188
9.2.1.2	L'hypercube . . . . .	188
9.2.1.3	Les grilles et les tores . . . . .	189
9.2.1.4	Graphes de De Bruijn et de Kautz . . . . .	192
9.2.1.5	Autres graphes . . . . .	193
9.2.2	La commutation de circuits : un réseau vu dynamiquement . . . . .	194
9.2.2.1	Le réseau dynamique complet : <i>crossbar</i> . . . . .	194
9.2.2.2	Le bus global . . . . .	195
9.2.2.3	Les réseaux de type Oméga . . . . .	196
9.2.2.4	Clos . . . . .	197
9.2.2.5	Beneš . . . . .	198
9.2.2.6	Autres réseaux dynamiques . . . . .	199
9.2.3	Les techniques intermédiaires de propagation . . . . .	199
9.2.3.1	La méthode de propagation <i>virtual cut-through</i> . . . . .	200
9.2.3.2	La méthode de propagation <i>wormhole</i> . . . . .	200
9.2.4	Quelques techniques de routage . . . . .	201
9.2.4.1	Les algorithmes déterministes . . . . .	201
9.2.4.2	Routage forcé et routage aléatoire . . . . .	202
9.2.4.3	Routage non déterministe . . . . .	203
9.3	Réseau de POMP : hybride statique et dynamique . . . . .	204
9.3.1	Principe . . . . .	204
9.3.2	Complexité . . . . .	207
9.3.2.1	Fils de données et de contrôle . . . . .	207
9.3.2.2	Complexité du câblage . . . . .	207
9.3.2.3	Partitionnement du réseau . . . . .	208
9.3.3	Usage . . . . .	209
9.3.3.1	Réseau statique . . . . .	210
9.3.3.2	Réseau dynamique . . . . .	212
9.3.4	Performances . . . . .	214
9.3.4.1	Réseau statique . . . . .	214
9.3.4.2	Réseau dynamique . . . . .	215
9.3.5	Conclusion . . . . .	217

## 10 Réalisation

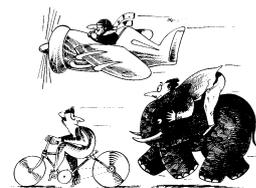
10.1	Description électrique . . . . .	221
10.1.1	La carte de contrôle . . . . .	221
10.1.1.1	L'interface VME . . . . .	221
10.1.1.2	Le MC88100 de contrôle . . . . .	223
	La mémoire de données scalaire . . . . .	223



220

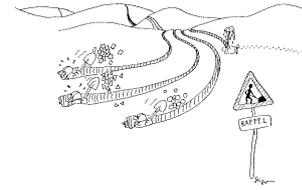
La mémoire de code scalaire . . . . .	225
La mémoire de code vectoriel . . . . .	227
10.1.1.3 Le bus de contrôle de la carte . . . . .	227
10.1.1.4 Le système de gestion des exceptions et de surcharge . . . . .	229
Les exceptions . . . . .	229
La surcharge des opérandes parallèles . . . . .	231
10.1.1.5 Les opérateurs globaux . . . . .	232
10.1.1.6 La vidéo . . . . .	234
10.1.1.7 La sortie audio . . . . .	235
10.1.1.8 Les entrées-sorties haut débit . . . . .	235
10.1.2 Les cartes processeur élémentaire . . . . .	236
10.1.2.1 Les modules processeurs . . . . .	237
10.1.2.2 Le contrôle de l'activité . . . . .	238
10.1.2.3 Une mesure analogique des performances de la machine . . . . .	239
10.1.2.4 Le réseau . . . . .	241
10.1.2.5 L'interface avec le reste de la machine . . . . .	241
10.1.3 Quelques problèmes rencontrés . . . . .	242
10.1.3.1 La technologie employée . . . . .	242
10.1.3.2 Extrait du carnet de bord de la machine . . . . .	243
10.2 Description physique . . . . .	243
10.2.1 Les bus de la machine . . . . .	244
10.2.1.1 Le bus VME de Sun . . . . .	244
10.2.1.2 Le bus de contrôle et de commande de la machine . . . . .	245
10.2.2 La répartition des PEs dans la machine . . . . .	246
10.2.2.1 Le réseau de communication . . . . .	247
10.2.3 Les entrées-sorties . . . . .	248
10.2.4 L'intégration dans la baie . . . . .	249
10.3 Conclusion . . . . .	249

**11 Divertissement sur la superlinéarité**



11.1 Notions d'accélération et de linéarité . . . . .	253
11.1.1 Taille du problème fixe . . . . .	254
11.1.2 Temps d'exécution fixe . . . . .	255
11.1.3 Modèle à mémoire constante par processeur . . . . .	256
11.2 La superlinéarité à taille fixée existe-t-elle? . . . . .	256
11.2.1 Modèle linéaire . . . . .	257
11.2.2 Modèle superlinéaire . . . . .	257
11.2.3 Généralisation du modèle . . . . .	259
11.3 Une machine superlinéaire . . . . .	260
11.3.1 Ordinateur SIMD . . . . .	260
11.3.2 Ordinateur VLIW-SIMD . . . . .	261
11.4 Conclusion . . . . .	262

**12 Vers une machine SPMD ?**



**264**

12.1 Modèles de programmation et langages . . . . . 265

    12.1.1 POMPC . . . . . 265

    12.1.2 HPF . . . . . 266

12.2 Architecture globale de la machine . . . . . 266

    12.2.1 Pour ou contre le partitionnement? . . . . . 267

    12.2.2 Un processeur scalaire est-il nécessaire? . . . . . 268

12.3 Nœud de la machine . . . . . 269

    12.3.1 Les processeurs élémentaires . . . . . 269

    12.3.2 La mémoire du nœud . . . . . 270

        12.3.2.1 Nœud SPMD à base de mémoire dynamique . . . . . 270

        12.3.2.2 Nœud SPMD à base de mémoire statique . . . . . 272

            Mémoire statique non entrelacée . . . . . 273

            Mémoire statique à bancs entrelacés . . . . . 273

        12.3.2.3 Conclusion . . . . . 273

12.4 Le système de gestion mémoire . . . . . 275

12.5 Le réseau . . . . . 280

    12.5.1 Le partitionnement du réseau . . . . . 281

    12.5.2 Les communications régulières . . . . . 283

    12.5.3 Les diffusions . . . . . 284

    12.5.4 Les attributs d'un paquet . . . . . 286

12.6 Les opérations globales . . . . . 287

    12.6.1 Les synchronisations . . . . . 287

        12.6.1.1 Le *ou* global flottant . . . . . 288

        12.6.1.2 Un *ou* global pipeliné multidimensionnel . . . . . 288

        12.6.1.3 Opération préfixe parallèle . . . . . 291

        12.6.1.4 Comparaison . . . . . 292

        12.6.1.5 Considérations de niveau système . . . . . 293

            La barrière de synchronisation . . . . . 294

            La fin de communication . . . . . 295

        12.6.1.6 Factorisation de barrières de synchronisation par comp-  
teur . . . . . 297

        12.6.1.7 Conclusion . . . . . 298

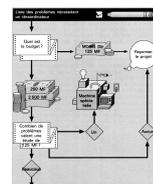
    12.6.2 Les concentrations associatives . . . . . 299

12.7 Mise au point de la machine . . . . . 299

12.8 Extensibilité . . . . . 301

12.9 Conclusion . . . . . 301

**13 Conclusion**



**304**

*TABLE DES MATIÈRES*

LXIII

**Bibliographie**



XIV

**Index**

XLII

Vous êtes ici . . . . . LXIII

## Résumé

contrôle de flot SIMD !

L'utilisation de plus en plus courante de gros modèles de simulation numérique et de la visualisation graphique des données nécessite d'avoir toujours plus de puissance de calcul.

Afin d'augmenter la puissance d'un ordinateur encore plus rapidement que l'amélioration de la vitesse des composants élémentaires, tels que les processeurs, une solution est le recours au parallélisme : on conçoit une machine à partir de plusieurs processeurs plutôt qu'un seul, espérant ainsi résoudre plus rapidement certains problèmes.

Le point de départ de notre étude est une machine suffisamment puissante pour permettre de la synthèse d'image de haute qualité mais qui conserve les avantages d'une station de travail ordinaire, c'est-à-dire être assez petite et consommer peu pour pouvoir loger sous un bureau et posséder une entrée/sortie vidéo efficace. Le refus de la spécialisation nous mène naturellement à étudier la machine comme étant un ordinateur assez ordinaire (programmable) mais performant.

Les objectifs de notre Petit Ordinateur Massivement Parallèle (POMP) prennent en compte les contraintes et les originalités liées à l'étude du matériel informatique dans le cadre d'un laboratoire de recherche universitaire. Même s'il s'agit de développer des concepts de portée suffisamment générale, les aspects liés à la réalisation d'un prototype industrialisable ont été pris en compte : le prototype étudié n'utilise que des circuits du commerce et des technologies de connexion disponibles chez les sous-traitants électroniques.

Le modèle de programmation choisi est de type parallélisme de données. Il permet une structuration du parallélisme à travers un modèle synchrone et s'adapte bien aux modèles numériques habituels et à la synthèse d'image en particulier. Une comparaison est faite entre quelques langages à parallélisme de données pour permettre de dégager les concepts nécessaires à la réalisation d'un tel langage. Le langage POMPC, extension du langage C, est présenté comme un langage à parallélisme de données explicite permettant de programmer une grande classe de machines parallèles dépassant le cadre de la machine POMP.

Le choix du modèle d'exécution est guidé par la densité de la machine et une bonne adéquation au parallélisme des données des problèmes évoqués. Cela nous mène au mode SIMD. Alors qu'habituellement SIMD signifie la conception d'un processeur spécialisé à grain fin, nous proposons de pervertir un processeur du commerce à gros grain permettant d'obtenir de meilleures performances en calcul flottant à volume équivalent. De nouveaux modes de contrôle de flot SIMD sont développés pour permettre une utilisation maximale des processeurs et des cas de superlinéarité sont dégagés.

Le contrôle de la machine est basé sur un couplage fort de type VLIW entre le processeur scalaire et les processeurs parallèles qui en fait une machine simple à concevoir.

La génération de code utilise C comme langage intermédiaire de haut niveau, ce qui nous permet de récupérer un environnement de développement complet déjà existant. Les 2 fichiers C de codes scalaire et parallèle sont synchronisés par des pseudo-fonctions permettant de mettre en correspondance les codes dans une phase ultérieure. Ces fichiers sont compilés par le compilateur C standard du processeur puis analysés par le synchroniseur. Celui-ci utilise un algorithme itératif simple pour faire la planification des instructions en étudiant le graphe de dépendance au niveau de l'assembleur et générer le code synchronisé VLIW-SIMD de la machine qui est ensuite assemblé par la chaîne standard de compilation.

Le choix du réseau est contraint de par sa réalisation à base de circuits reprogrammables du commerce qui nous évite d'avoir un circuit complexe à concevoir.

Un bon compromis semble être un hybride entre un réseau en hypercube et un réseau multi-étage qui peut être vu suivant les deux modes selon les instructions de communications proposées dans le langage POMPC.

Cette approche pragmatique permet néanmoins de concevoir une machine offrant plus de 5 GIPS et 2 GFLOPS, consommant de l'ordre d'un kW et conservant la taille d'une grosse station de travail.

À titre de comparaison nous développons une machine SPMD se libérant de la contrainte de taille mais nécessitant un circuit de communication plus complexe. Pour ce faire, une méthode de partitionnement de la machine et de synchronisation partitionnable est exposée.

Enfin nous présentons 2 applications programmées en POMPC et issues du monde de la physique : une méthode de résolution d'équations différentielles elliptiques par une méthode multigrille et un programme de simulation de gaz sur réseau.