APPLICATION

D'un Petit Ordinateur Massivement Parallèle synthèse du projet POMP

Ronan Keryell

Centre de Recherche en Informatique
École des Mines de Paris
35, rue Saint-Honoré
77305 FONTAINEBLEAU Cedex
FRANCE
`keryell@cri.ensmp.fr`

RÉSUMÉ. Le projet POMP *a eu comme objet le développement d'une machine parallèle* SIMD *avec son environnement de programmation. Les applications visées nous ont amenés à choisir un modèle de programmation à parallélisme de données et à développer un langage de programmation adapté. Contrairement aux approches* SIMD *habituelles, l'utilisation d'un processeur* RISC *du commerce évite le développement d'un processeur spécifique et le choix d'un mode de contrôle* VLIW *avec le même processeur évite la conception d'un contrôleur-processeur scalaire complexe et permet des optimisations statiques. Une optimisation du contrôle de flot* SIMD *est abordée dans le cas de l'imbrication de* `where` *ainsi que dans le cas d'un bloc terminal. Enfin, un réseau hybride statique et dynamique est présenté permettant d'exploiter au mieux les 2 types de communications rencontrés.*

MOTS-CLÉS : architecture, parallélisme de données, SIMD*, contrôle de flot* SIMD*, réseaux d'interconnexion statiques et dynamiques, compilation parallèle,* POMP*,* POMPC.

ABSTRACT. The POMP *project includes an* SIMD *parallel machine and its dataparallel programming environment,* POMPC*. Instead of developping a specific* SIMD *processor, we use a commercial* RISC *processor for the processor element. The control of the whole machine is done by another* RISC *processor in a* VLIW *way. A new method of dynamic scheduling of* SIMD *control flow and an optimization for small blocks are introduced. At last, an hybrid static and dynamic interconnection network is used to speed-up regular and irregular communications.*

KEY WORDS : architecture, dataparallelism, SIMD*,* SIMD *control flot, static and dynamic interconnection networks, parallel compilation,* POMP*,* POMPC.

1. Introduction : des architectures graphiques au parallélisme

Les ordinateurs sont de plus en plus amenés à gérer des images, outils puissants de représentation, qui ne valent plus dix mille mots comme au temps du proverbe chinois mais plutôt des milliards de mots pour une image de simulation numérique en 3 dimensions et en couleur.

Dans ce cadre, le projet POMP (Petit Ordinateur Massivement Parallèle [1]) a débuté au Laboratoire d'Informatique de l'École Normale Supérieure afin d'augmenter la puissance des machines utilisées en synthèse d'image et en visualisation scientifique, telles que nous avions pu en concevoir auparavant dans le laboratoire.

En fait, on s'est rendu compte que la phase d'affichage de l'image en soi prenait de moins en moins de temps par rapport à la phase d'élaboration des données (simulation numérique) ou de rendu réaliste (synthèse d'image, pouvant aussi inclure la simulation de phénomènes physiques). Il s'agissait donc d'optimiser la phase la plus consommatrice en calcul sans négliger l'aspect affichage et proposer la machine dans un format compatible avec une station de travail et ayant une consommation électrique du même ordre de grandeur.

Le gain en performance devait être amené par l'utilisation de plusieurs processeurs travaillant en parallèle, le *parallélisme*, plutôt que via la conception d'un gros processeur très puissant, probablement très pipeliné et plus lié aux contingences technologiques.

D'un point de vue matériel, le parallélisme SIMD [2] avait déjà été employé dans un de nos projets précédents et avait l'avantage d'être à la fois simple (un flot unique d'instructions) et efficace sur les problèmes d'affichage : parallélisme au niveau pixel où une même opération doit souvent être faite sur de nombreux points de l'images d'une part, mais aussi sur plusieurs bits de ces pixels.

D'un point de vue logiciel, on rencontre très souvent dans les applications scientifiques en général et dans les modèles de simulation numérique en particulier des algorithmes qui manipulent des ensembles de données (typiquement vecteurs, matrices, rayons,...) en répétant la même opération sur chaque donnée : il s'agit du *parallélisme de données*.

Ce modèle est proche du concept de *smart memory* utilisé par le matériel gérant les pixels [AKE 88] et il nous a semblé naturel de *projeter* tous ces parallélismes sur une machine unique SIMD qui gérerait ces parallélismes par pipeline logiciel et multiplexage temporel.

Le choix du SIMD a été fait à travers plusieurs critères, plus ou moins objectifs [3] :

- architecture simple : un seul séquenceur fournit les instructions exécutées par tous les processeurs ;
- le modèle de programmation est synchrone et les programmes sont donc faciles à mettre au point ;
- le programme de la machine est unique puisque c'est le même qui s'exécute sur chaque nœud ;

1. En anglais libre : *Perversion Of Many Processors*.
2. *Single Instruction stream, Multiple Data streams*.
3. Car la notion de simplicité est très subjective et très culturelle !

— l'aspect synchrone de l'architecture facilite le couplage avec l'interface vidéo ;
— l'écriture d'un compilateur pour une telle machine est plus facile.

La section 2 présentera le modèle de programmation et le langage POMPC. Dans la section 3.1 nous rompons avec l'idée bien pensante que le SIMD implique du parallélisme à grain fin en utilisant un des processeurs RISC les plus puissants du marché. Le contrôle de la machine est fait sur un mode VLIW à la manière présentée dans la section 4 et le contrôle de flot SIMD dans la section 5. Le principe de génération de code est abordé dans la section 6 tandis que la section 7 expose le principe du réseau de la machine. Enfin, la section 8 survole la réalisation de la machine.

2. Programmation

2.1. *Modèle de programmation*

Le choix du modèle de programmation permettant d'exprimer correctement et efficacement les algorithmes cibles doit permettre de privilégier quelques classes d'architectures. Il s'agit donc d'un des premiers choix à effectuer dans la définition d'une machine.

Parmi toutes les classes de parallélisme, une classe de parallélisme se dégage plus particulièrement de nos applications : elles ont toutes à traiter un grand nombre de données en parallèle.

2.1.1. *Parallélisme de données*

Le parallélisme de données est particulièrement adapté s'il s'agit d'effectuer une même opération sur toutes les données, ou tout au moins un sous-ensemble des données, ce qui est notre cas.

Actuellement, il s'agit là du parallélisme qu'on sait le mieux exploiter et c'est celui qui a été choisi pour POMP. Contrairement à d'autres machines il faut être capable de gérer des variables parallèles de taille quelconque, indépendante du nombre de processeurs élémentaires (PEs) de la machine, comme dans les langages parallèles de haut niveau.

On choisit une vue locale du parallélisme, plus proche du matériel et donc plus facilement exploitable. Le parallélisme est exprimé sous la forme de machines virtuelles de forme adaptée au problème à traiter et composées de processeurs virtuels (PVs). Chaque PV contient un élément de chaque donnée parallèle et la virtualisation permet de libérer l'utilisateur de la taille physique et de la géométrie de la machine (topologie des processeurs physiques), un peu comme la mémoire virtuelle libère le programmeur de la taille de la mémoire physique d'un ordinateur.

À chaque pas de calcul parallèle macroscopique vu par l'utilisateur, en fait chaque PE s'occupe de l'opération qu'il doit faire sur chacun de ses PVs. Ainsi, la machine peut gérer des données de taille arbitrairement grande en parallèle, dans les limites de la mémoire disponible, de la même manière que le strip-mining dans les ordinateurs vectoriels.

2.1.2. *Communications explicites*

Cette notion de PV est une vision exacerbée de la règle des écritures locales *owner compute rule* puisqu'on fixe implicitement le PV sur lequel chaque calcul sera effectué et non simplement le PE.

Il faut être capable de différencier les accès locaux à un PV, par exemple lors de calculs sur des variables parallèles élément par élément, des accès à des valeurs contenues sur d'autres PVs qui nécessitent des communications, pour tout autre type d'interaction telles que *scatter*/*gather* ou décalages sur grille par exemple.

Afin de simplifier la phase de compilation, toute phase de communication est précisée par une syntaxe particulière. Un des effets intéressant est de bien signifier à l'utilisateur que les communications sont coûteuses et donc qu'il faut les limiter si on veut avoir des performances raisonables. De plus les notions de variables parallèles et de variables séquentielles sont bien distinctes.

Néanmoins à la restriction ci-dessus près et contrairement à d'autres modèles de parallélisme, on offre à l'utilisateur un espace de nommage global des variables : on ne voit pas dans la programmation la notion de tableaux locaux aux processeurs physiques qu'ils faudrait gérer explicitement.

2.2. *Le langage POMPC*

Aucun langage ne satisfaisant à notre connaissance l'objectif désiré de portabilité, de syntaxe simple et de sémantique parallèle claire, la conception d'un nouveau langage a été décidée. Cela avait comme intérêt la prise en compte de nos *desirata* ainsi qu'une maîtrise totale du compilateur.

Pour des raisons d'efficacité de compilation, d'habitude de programmation et d'expérience, un langage impératif basé sur C avec des extensions parallèles a été conçu : POMPC. Comme on avait une expérience de la CM-2, le langage a visé une amélioration du langage C*, langage utilisé sur cette machine.

Comme dans le langage C*, le pragmatisme a mené à l'expression explicite du parallélisme par rapport à un travail de parallélisation et à la résolution des communications à l'exécution plutôt que des optimisations plus statiques liées à une analyse fine des dépendances, voire interprocédurale.

Cela a permis d'obtenir rapidement un compilateur complet du langage, sans avoir à choisir un sous ensemble du langage C en particulier.

2.2.1. *Les collections d'objets*

Dans un algorithme, on ne rencontre souvent qu'un petit nombre de classes de parallélisme, typiquement un espace de parallélisme représentant un espace de modélisation discrétisé, des matrices de même taille ou encore des vecteurs de taille identique.

La déclaration de classes d'équivalence de parallélisme, les *collections*, permet de typer, aussi bien du point de vue de la taille que de la forme, les objets parallèles manipulés par la machine et d'inférer simplement la sémantique SIMD des opérations.

```
collection [100,100]une_matrice;                                    1
collection [20]vecteur;                                             2
une_matrice int ecran;                                              3
une_matrice double potentiel,p1,p2;                                 4
vecteur int coord_x,coord_y;                                        5
int couleur;                                                        6
{           ...                                                     7
    potentiel = p1 + p2;                                            8
    where (p1 < 0)                                                  9
        p1 = -p1;                                                   10
    [coord_x,coord_y]ecran <- couleur;                              11
    potentiel = 0.25*((<-[.,.+1]potentiel) + (<-[.,.-
                   1]potentiel) + (<-[.+1,.]potentiel) +
                 (<-[.-1,.]potentiel));                             12
}                                                                   13
```

Figure 1. *Déclarations de variables parallèles et utilisation.*

La figure 1 montre la déclaration de matrices parallèles et l'addition de 2 matrices (ligne 8). Leur taille et leur forme sont ici connues à la compilation mais POMPC autorise aussi bien des déclarations dynamiques de collections, y compris celles concernant des variables parallèles globales, contrairement à C*.

Une collection peut être passée en paramètre, ce qui permet d'écrire des fonctions parallèles génériques.

2.2.2. *Le contrôle de flot*

Comme POMPC est un surensemble de C, il contient tout le contrôle de flot séquentiel de C.

Un certain nombre d'opérateurs de contrôle de flot parallèle ont été introduits : les `where`, `forwhere`, `switchwhere`, `whilesomewhere` qui étendent respectivement au domaine parallèle des `if`, `for`, `switch` et `while`. Les lignes 9 et 10 de la figure 1 montre l'utilisation d'un `where` pour prendre la valeur absolue d'une variable parallèle.

Pour des raisons d'efficacité et surtout de sémantique du modèle les `goto` parallèles n'ont pas été introduits.

Par contre il est possible d'imbriquer des opérateurs de contrôle de flot et la sémantique d'un tel opérateur est que *sa visibilité ne porte que sur les opérations concernant sa collection.*

2.2.3. *Les communications*

Elles sont exprimées par l'opérateur `<-` et les adresses de communications sont précisées à gauche de la variable, afin d'éviter tout conflit avec des variables parallèles de tableaux.

La figure 1 présente un cas de communication irrégulière indexée par une variable parallèle (ligne 11) et un cas de communications régulières pour le calcul d'un laplacien en 2D (ligne 12).

Il est intéressant de faire la distinction entre communications régulières et communications irrégulières car souvent le matériel permet une exécution plus rapide des cas réguliers. Les communications régulières s'apuient sur l'opérateur de coordonnée courante $\boxed{\cdot}$ utilisé à la ligne 12.

Enfin, des opérations de réductions sont fournies (un produit scalaire s'écrit par exemple `+<- v*u`) ainsi que des bibliothèques d'opérations préfixes parallèles dont la puissance algorithmique n'est plus à démontrer [BLE 89].

2.2.4. *Le placement des données*

Un des problèmes critiques en programmation de machine parallèle concerne le placement des données sur les processeurs physiques : si certains motifs de communications concernent plus certaines directions d'une collection que d'autres, on aura intérêt à réduire les communications dans ces directions en augmentant la taille des blocs de données sur les processeurs dans ces directions.

De telles considérations ont mené au développement de langages comme FORTRAN D [FOX 92] ou Vienna FORTRAN [ZIM 92] et plus récemment à HPF et sont prises en compte en POMPC au moment de la déclaration d'une collection, avec en plus des options pour le rebouclage torique lors d'accès à des variables parallèles.

3. Architecture

Après avoir aperçu le modèle de programmation de la machine et le langage POMPC, nous pouvons détailler l'architecture SIMD de la machine, à commencer par un point crucial : le choix du processeur élémentaire.

3.1. *Les processeurs élémentaires*

3.1.1. *Caractéristiques générales*

Gros grain contre grain fin Un des premier choix à faire est celui de la largeur des processeurs et souvent on suppose que si on a N processeurs ayant une puissance p, la machine aura une puissance $P \leq Np$ [BRE 74].

Or les applications visées nécessitent de faire des calculs sur des variables de taille raisonnable contrairement à des applications de traitement d'images binaires comme dans [UNG 58]. Étrangement, il apparaît que toutes les machines SIMD commerciales sont à grain fin et qu'aucun projet de machine SIMD à gros grain (processeurs larges) n'ait abouti à une commercialisation [KER 92]. Cela va même plus loin puisque selon [FLY 72], l'ILLIAC IV, pourtant SIMD à gros grain, n'est pas une machine SIMD. Néanmoins, il s'agit bien là d'une machine SIMD à gros grain, de même que OPSILA [AUG 90] ou PASM [SIE 84] (en fait ces 2 dernières peuvent être aussi SPMD).

Pourtant, pour de nombreuses raisons le gros grain est plus performant que le grain fin. Au niveau des calculs élémentaires, un processeur 32 bits pipeliné produira un résultat de multiplication de 2 nombres de 32 bits par cycle alors qu'il faudra 1024 cycles pour faire la même opération sur un processeur 1 bit. C'est ce qui a fait rajouter des coprocesseurs flottants sur la CM-2 et l'abandon pur et simple des processeurs 1 bit par les compilateurs *slice-wise* sur cette machine [SAB 92]. Il est intéressant de constater que sur un nœud le flottant occupe 40 % des transistors sans la mémoire contre 5 % pour les processeurs et donc que les processeurs 1 bit sont très mal équilibrés avec le reste du nœud.

D'un point de vue purement algorithmique, des problèmes efficaces de type opérations préfixes parallèles de taille n sont résolus en un temps $\mathcal{O}(\frac{n}{p} + \log p)$ sur une PRAM de taille p [KRU 85] et donc avec une efficacité $\mathcal{O}(\frac{1}{1+p\log p/n})$, ce qui pousse à avoir un nombre minimal de processeurs.

En ce qui concerne la mémoire, moins de processeurs permet plus de mémoire par processeur ce qui permet de faire fonctionner des algorithmes impossibles avec moins de mémoire, comme ceux nécessitant de grosses *look-up tables* locales.

Un autre point important est la qualité de l'adressage local de la mémoire. En effet, accéder à la mémoire localement nécessite des bus d'adresse larges, que les processeurs soient petits ou gros. Si un circuit intégré peut contenir 1 gros processeur ou n petits processeurs et que ce circuit est dans les deux cas relié à une mémoire identique de M bits, dans le 1$^{\text{er}}$ cas on a besoin de $\lceil \log_2 M \rceil$ pattes pour le bus d'adresse. Dans le 2$^{\text{ème}}$ cas, soit on fait du multiplexage temporel de n adresses sur ce même bus et les performances sont faibles, soit on a n mémoires de M/n bits et donc n bus de $\lceil \log_2(M/n) \rceil$ bits de large, ce qui est très performant mais irréalisable. C'est ce qui explique en partie les faibles performances de la CM-2 en adressage local en plus du fait que les adresses étant envoyées en format série par les processeur 1 bit nécessitent une conversion en format parallèle.

D'autres arguments peuvent être trouvés dans [KER 92] et on peut résumer cette discussion en constatant qu'utiliser des processeurs à gros grain permet d'utiliser le *parallélisme intrinsèque* trivial d'une application scientifique avant toute autre sorte de parallélisme : le fait qu'on manipule des nombres sur 32 ou 64 bits et donc qu'on a tout intérêt à commencer par exploiter ce parallélisme facile par une machine à gros grain. Un processeur de 32 bits peut être vu comme un processeur parallèle au niveau des bits optimisé pour traiter de tels nombres, comme le montre l'exemple de la multiplication sur la CM-2 cité ci-avant.

Coût technologique et balance processeur-mémoire Un point important est d'avoir la machine la plus efficace à un coût donné. Au niveau d'un nœud il s'agit tout particulièrement de bien équilibrer la taille mémoire avec le processeur. Comme fonction de coût on peut prendre le critère de compacité puisqu'il s'agit d'un facteur limitant important de notre cahier des charges, qui est constant sur toute la machine.

On a tout intérêt à avoir un coût égal entre processeur et mémoire car pour chaque application :

— soit on s'est trompé et on a trop de mémoire qui est inutilisée mais on sait qu'on

ne pouvait mettre plus du double de processeurs ;
— soit la machine pèche par manque de mémoire, mais on ne pouvait pas faire mieux que la doubler.

Avec un tel choix de mettre autant de surface de mémoire que de processeur, on ne se trompe au pire d'un facteur 2 en performance [4].

Bien entendu, de telles considérations seraient caduques si on pouvait intégrer processeur et mémoire sur le même circuit intégré mais cela est difficile aujourd'hui dans un cadre universitaire, même si c'est l'optique étudiée au début du projet.

Caractéristiques SIMD Bien entendu, le grain ne fait pas tout et il faut aussi que les processeurs aient de bonnes caractéristiques SIMD qu'on peut résumer par :

— une architecture HARVARD [5] pour permettre d'envoyer des instructions indépendamment des données locales, celles-ci résidant dans la mémoire de chaque PE ;
— un comportement synchrone pour toute la machine, à moins de synchroniser toute la machine à prix fort en temps comme dans PASM [SIE 84]. Dans les deux cas cela empêche aussi l'utilisation d'un cache ;
— un système de contrôle de l'activité permettant d'arrêter certains processeurs sur des conditions locales, comme on le verra plus loin ;
— des systèmes de gestion de communications, d'entrées-sorties et d'exception.

En outre, on désire avoir toutes les sucreries habituelles qui permettent d'avoir des processeurs rapides : pipeline et confluence pour augmenter le débit d'instructions, VLIW [6] ou superscalaire, architecture RISC pour simplifier le processeur et le compilateur, etc.

3.1.2. *Le développement d'un processeur : un mal nécessaire ?*

Le problème est qu'il n'existe pas de processeur commercial SIMD à gros grain. C'est bien dommage car des processeurs RISC ont pourtant presque les caractéristiques requises et un certain nombre d'avantages : leur puissance augmente constamment avec les nouvelles versions et on évite une « dérive technologique » du projet, on bénéficie du travail de nombreuses équipes de développement, les frais de développement sont limités à l'intégration dans le projet, l'environnement logiciel complet et enfin le pipeline et les branchements non retardés permettent une gestion inédite du contrôle de flot SIMD comme on va le voir.

Il nous a semblé raisonnable d'étudier la perversion d'un processeur du commerce pour POMP. Une telle approche avait été faite dans PASM mais les processeurs CISC n'allant pas tous à la même vitesse devaient être synchronisés après chaque instruction [7].

4. D'un point de vue philosophique, on peut voir cela comme une généralisation de la « loi d'AMDAHL » [AMD 67] telle qu'elle est faite dans [HEN 90, pages 8–11] mais appliquée à la compacité de la machine.
5. C'est à dire qui possède deux bus distincts pour les instructions et pour les données.
6. *Very Long Instruction Word* : architecture à mot d'instruction très long où chaque champ commande une unité fonctionnelle simultanément.
7. Un comble pour une machine SIMD !

3.2. *Le processeur élémentaire de POMP*

3.2.1. *Un 88100*

Un bon candidat pour le projet était le processeur MC88100 : RISC, HARVARD avec un bus à fonctionnement simple, pas de cache intégré mais avec coprocesseur flottant intégré.

Avec des performances d'environ 10 MFLOPS et 21 MIPS pour la version à 25 MHz utilisée dans notre prototype, cela laisse espérer une performance globale maximale de 2,5 GFLOPS sur la base de 256 processeurs[8].

La particularité de la machine est que le processeur est nourri de force par un séquenceur via son bus d'instructions dont les adresses et les signaux sont simplement *ignorés*.

Afin de soutenir les performances sans cache, la mémoire de la machine est statique. Elle est de capacité moindre que la mémoire dynamique et sont prix est plus élevé. Néanmoins, elle ne représente que la moitié du prix de la machine et permet des économies de circuits d'interface. Comme le bus du processeur est pipeliné, un verrou est rajouté sur le bus d'adresse de données.

3.2.2. *Diffusions scalaires*

Elles sont réalisées en surchargeant au niveau du séquenceur des champs de certaines instructions envoyées aux PEs. La tâche est facilitée par le fait que dans un processeur RISC ces champs sont souvent au même endroit dans les instructions.

Les affectations de variables parallèles par une variable scalaire sont réalisées en écrasant le champ de valeur d'une instruction de chargement immédiat et les indirections locales à partir d'une adresse globale de manière identique sur une instruction d'adressage avec valeur immédiate.

On peut aussi créer des indirections sur d'autres champs pour faire par exemple des indirections sur les numéros de registre, ce qui peut être utile pour écrire des routines d'exception. Mais cela n'a pas été réalisé dans notre approche « RISC » de la machine.

3.2.3. *Contrôle et communications*

Bien entendu, ce processeur commercial n'est pas totalement adapté à une machine SIMD. Il faut donc rajouter un minimum d'électronique autour, rassemblée principalement dans notre circuit HYPERCOM.

Ce circuit est doublé d'un petit PAL rapide qui gère le pipeline du bus, l'activité bas niveau et le blocage en cas d'exception.

L'HYPERCOM gère les communications entre les processeurs à travers des liens de type série, les hypercanaux. Pour des raisons d'économie, les entrées-sorties rapides sont reliées aussi à ces liens.

Comme l'interface vidéo est primordiale dans l'architecture, elle est aussi implantée au niveau de ce circuit. Néanmoins, pour éviter les interférences, un hypercanal séparé

8. Atteinte typiquement sur des calculs de fractales. On considère que la puissance crête de 6,25 GFLOPS de la machine ne signifie pas grand chose...

par PE lui est réservé. La mémoire écran peut ainsi être répartie sur tous les PEs pour équilibrer la charge de la machine.

Si on suppose une machine à 256 PEs, le débit sur chaque lien suffit. Avec un tampon vidéo de 1 Ko par HYPERCOM et un lien à 25 MHz, le remplissage peut se faire tous les 4000 cycles vidéo environ, soit 2 μs toutes les 20 μs, ce qui est raisonnable.

Comme la machine est synchrone, le calcul d'un état global de la machine ne nécessite qu'une fonction telle que le *ou* global d'une variable parallèle. Ce signal est doublé par un fil d'exception globale permettant de dérouter le séquenceur lorsqu'une exception survient sur un PE.

Dans ce dernier cas, il faut que le processeur scalaire corrige l'exception sur le ou les PEs qui sont en exception. Comme la machine est pipelinée, cela ne peut se faire instantanément. Pour cela l'HYPERCOM a un système qui rend sourd au flot d'instructions son PE fautif et note la date de l'exception. En plus il échantillonne le bus d'adresse d'instructions du PE (dont c'est la seule utilité dans la machine) pour noter le numéro de l'exception. Ainsi, le processeur scalaire a tous les éléments pour refaire démarrer la machine après avoir corrigé les exceptions par classe (date et type).

Plutôt que d'avoir un seul signal de *ou* global on en a 4, ce qui permet d'accélérer d'un facteur 4 les réceptions scalaires qui sont aussi utilisées pour les entrées-sorties dans une première version de la machine.

3.3. *Conclusion*

Malgré un certain nombre de contraintes technologiques, le fait de pervertir un processeur RISC du commerce apporte beaucoup :

- machine plus réalisable car moins dépendante d'un pari technologique ;
- le circuit HYPERCOM peut être simple au point de pouvoir loger dans un circuit reprogrammable de type LCA (XC4000) ;
- l'environnement logiciel de base du processeur permet de limiter la programmation en assembleur.

Le synoptique du nœud est représenté sur la figure 2.

4. Le contrôle de la machine

Afin de faire fonctionner la machine, il faut d'une part exécuter le code scalaire du programme et d'autre part séquencer le code parallèle pour les PEs. Ces tâches sont attribuées dans la littérature respectivement au processeur scalaire et au séquenceur.

4.1. *Dispositifs de contrôle*

Généralement, 4 dispositifs sont impliqués dans une machine SIMD :

- un hôte ou « frontal » qui est une machine commerciale dont on récupère l'environnement système et qui interface le processeur scalaire avec le monde

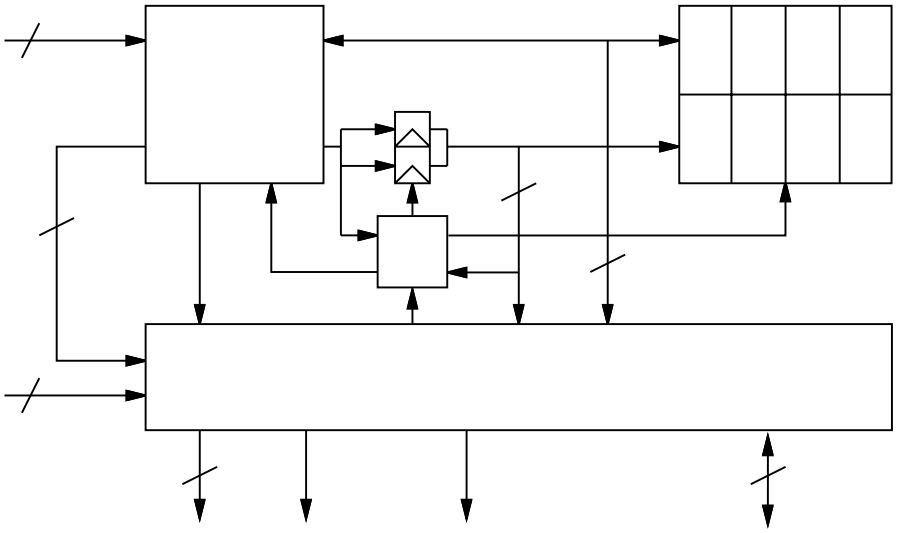

Figure 2. *Synoptique du processeur élémentaire.*

réel ;
- un processeur scalaire qui exécute la partie scalaire du programme et qui lance des instructions parallèles contenues dans le flot d'instructions scalaires ;
- un séquenceur qui récupère ces instructions parallèles pour en faire des micro-instructions directement compréhensibles par les PEs ;
- et bien sûr les PEs qui exécutent le code parallèle.

En fait pour des raisons de coût ou pour tenir compte de certaines particularités de machine SIMD on est parfois amené à rassembler certains de ces dispositifs. Pratiquement toutes les combinaisons ont été utilisées.

L'avantage du séquenceur est qu'il permet un débit d'instructions parallèles plus faible et est utilisé par exemple dans la CM-2 où l'hôte et le processeur scalaire ne font qu'un et sont une station de travail.

L'avantage d'un processeur scalaire indépendant est qu'il permet de contrôler plus rapidement les PEs.

La conception d'un séquenceur part en général de processeurs « en tranche » très rapides spécifiques au séquencement qu'on arrange de façon adaptée au problème.

L'inconvénient est que cela revient à construire un petit ordinateur pipeliné rapide qui lit et génère le code de manière parallèle pour assurer le débit étant donnée la performance des PEs. Le seul séquenceur apparaît aussi compliqué que le reste de la machine. En outre, comme les possibilités de combinaisons sont immenses, il n'y a pas de compilateurs commerciaux mais seulement des assembleurs paramétrables. La tâche de micro-programmation est de fait ingrate pour l'architecte.

Afin de simplifier la machine, la suppression d'un séquenceur a été retenue.

4.2. *Un contrôle VLIW*

Avec les processeurs RISC actuels les temps de cycle sont du même ordre que ceux des micro-séquenceurs en tranche. Il est donc envisageable de pervertir un autre processeur RISC pour en faire un séquenceur.

Comme l'hôte, le processeur scalaire et le séquenceur utilisent tous un processeur RISC, il est tentant de tout projeter sur un seul processeur, celui de l'hôte. Malheureusement il est difficile de greffer du matériel supplémentaire à une machine existante et d'autre part toute la partie séquenceur ne bénéficierait pas au système d'exploitation de l'hôte. Pour ces raisons, seuls processeur scalaire et séquenceur ont été fondus.

Un mode de contrôle VLIW a été choisi pour sa pureté : à chaque fois que le processeur scalaire lit une instruction pour lui dans sa mémoire de code scalaire une instruction parallèle est lue dans la mémoire de code parallèle et envoyée aux PEs. Puisque là aussi le déterminisme temporel est crucial, il faut prendre un processeur sans mémoire cache.

Deux cas peuvent se présenter :

— soit il y a trop d'instructions scalaires par rapport aux instructions parallèles. C'est que le code est trop séquentiel et on tombe dans le cas d'AMDAHL [AMD 67] : un ordinateur parallèle ne peut accélérer que du code parallèle qui est ici minoritaire. Le code séquentiel s'exécute toujours aussi lentement et domine dans ce cas le temps d'exécution et apporte de faibles performances ;

— soit il y a trop d'instructions parallèles et c'est tant mieux car on utilise pleinement la machine.

Ce couplage simple nous semble donc satisfaisant.

En choisissant un processeur identique à ceux du PE on simplifie la réalisation car les processeurs seront plus facilement synchronisables et les codes scalaire et parallèle semblables. Les codes sont implicitement synchrones ce qui permet de faire des optimisations statiques à la compilation.

Le synoptique complet de la machine est donné sur la figure 3. On peut voir la mémoire d'instructions qui est doublée. Une instruction parallèle contient en fait 40 bits, 32 bits pour les MC88100 des nœuds et 8 bits pour les HYPERCOM des nœuds. Cette instruction est en fait liée à celle du processeurs scalaire pour former une instruction VLIW de 72 bits.

La mémoire de données du processeur scalaire est par contre unique et ne contient que les données scalaires.

Un multiplexeur permet de surcharger certains champs d'instructions parallèles à partir d'une valeur sur le bus de données scalaires pour autoriser des émissions scalaires. Comme l'émission des instructions vers les PEs est pipelinée, en cas d'exception il faut mémoriser l'état du pipeline et les instructions qui ont déjà été envoyées aux PEs. Cela est fait par une file d'attente (FIFO) qui est analysée par la routine d'exception qui pourra renvoyer les instructions fautives aux PEs concernés.

En ce qui concerne la vidéo, les hypercanaux sont rassemblés et les données passent dans 8 transposeurs de matrices de bits de taille 32×32 [KER 88, KER 92] pour faire la conversion entre 256 hypercanaux au format série à 25 MHz en une vidéo RVBα au format parallèle 4×8 bits à 100 ou 200 MHz. Ces signaux sont ensuite envoyés dans

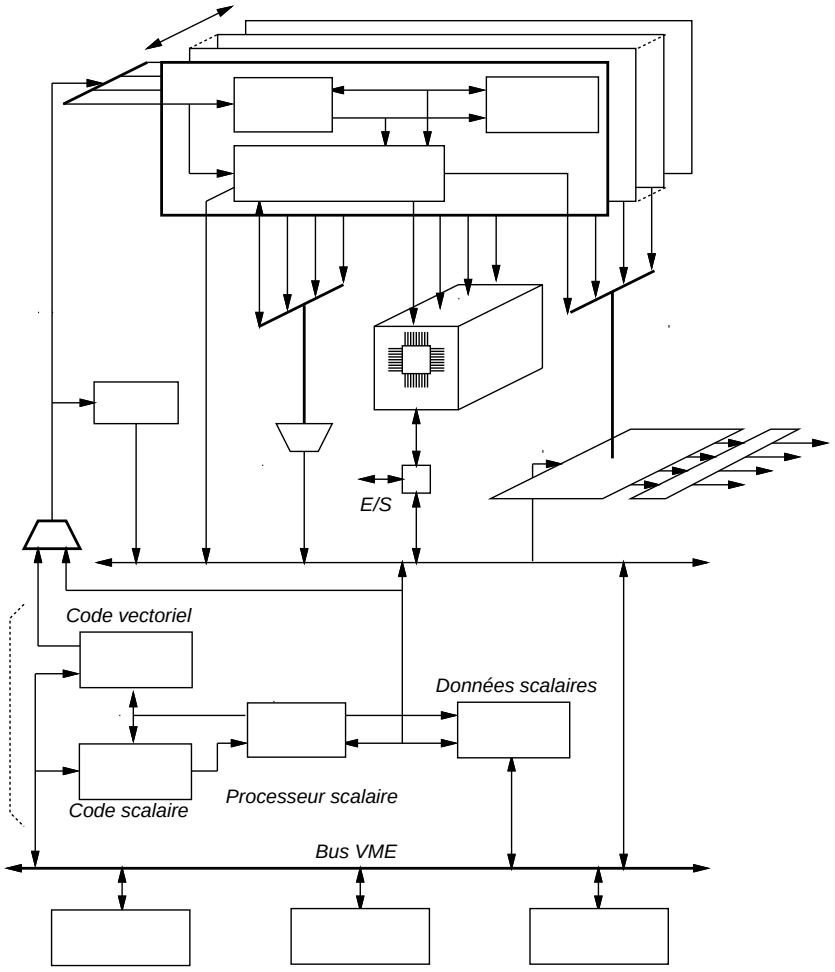

Figure 3. *Synoptique de la machine* POMP.

des convertisseurs digital-analogique avec palette de couleurs pour obtenir les signaux vidéo classiques. Les transposeurs sont réalisés là aussi avec des circuits logiques programmables.

Les entrées-sorties rapides sont connectées au réseau global de POMP. L'utilisation d'un seul hypercanal par PE permet d'alimenter 8 interfaces HIPPI de 100 Mo/s chacune, et donc jusqu'à 128 de ces interfaces en utilisant tout le réseau (16 hypercanaux/PE).

Enfin, la connexion de la carte du processeur scalaire à l'hôte (un SUN 3/110) se fait par un bus VME.

5. Le contrôle de flot SIMD

L'intérêt d'une machine SIMD est qu'on peut effectuer la même opération sur toutes les données d'un problème. Pourtant, bien souvent une application nécessitera de ne pas appliquer toujours la même opération sur les données, dans le cas par exemple de conditions aux limites d'un problème aux dérivées partielles.

Cette nécessité semble incompatible avec la notion de SIMD qui suppose un synchronisme parfait des PEs. Le principe utilisé depuis les débuts des machines SIMD [SLO 62] est d'empêcher l'exécution d'une instruction en fonction d'une condition locale, l'« activité ». D'un point de vue conceptuel alors que le branchement dans une machine MIMD ou tout simplement séquentielle est un déplacement dans l'espace d'adressage, le branchement dans une machine SIMD est un branchement dans le temps : un PE, ne pouvant pas influencer le flot d'instructions qui lui arrive, ne peut qu'attendre que l'instruction qui l'intéresse lui parvienne dans le futur.

Des méthodes de contrôle de flot SIMD ont été beaucoup étudiées pour la vectorisation automatique de boucle contenant du contrôle de flot, principalement à travers le principe d'*if-conversion* [ALL 83]. Afin de permettre l'imbrication d'opérateurs de contrôle de flot SIMD, il faut sauvegarder les activités des niveaux supérieurs dans une pile et l'activité courante est le *et* logique de toutes les activités englobantes.

Des travaux ont été faits de manière à optimiser le contrôle de flot statiquement [KEN 90] donc sans considérer la récursivité ou l'interprocéduralité non connue à la compilation. En outre le langage POMPC autorise comme d'autres langages orientés collection d'entrelacer des instructions concernant différentes collections, ce qui est très difficile à compiler statiquement. Il a donc fallu développer des méthodes de contrôle dynamique efficace dans le cas d'un langage structuré par bloc comme POMPC [KER 93].

5.1. *Optimisation de blocs conditionnels imbriqués*

Si on regarde d'un peu plus près la méthode de pile d'activité telle qu'elle est utilisée sur la CM-2 ou la MP-1 on s'aperçoit qu'une fois qu'on a mis une condition fausse dans la pile (un « 0 », *faux*), toutes les instructions concernant cette collection dans des blocs conditionnés imbriqués ne sont pas exécutées, comme symbolisé sur la figure 4.

Les valeurs stockées sur la pile n'ont plus aucune importance une fois un 0 mis sur la pile. La pile est donc un gâchis de matériel.

5.1.1. *Factorisation*

La seule information utile dans cette pile est le nombre de blocs conditionnels imbriqués inactifs, c'est à dire le nombre de blocs dont doit sortir un PE pour redevenir actif.

Soient `push`(*cond*) et `pop` les 2 opérations contrôlant la pile $(a_i)_{i\in\mathbb{N}}$. On peut analyser leur fonctionnalité suivant f_0, la place du premier 0 sur la pile, et s la taille

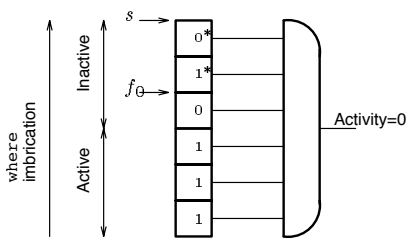

Figure 4. *Exemple de pile d'activité.*

Table 1. *Sémantique des opérations* `push` *et* `pop` *sur la pile d'activité.*

Opération	Comportement	Précondition	Action
`push`(*cond*)	$s \leftarrow s+1$	$f_0 \neq s+1$	$f_0 \leftarrow f_0$
	$a_s \leftarrow cond$	$(f_0 = s+1) \wedge (cond = 0)$	$f_0 \leftarrow s$
		$(f_0 = s+1) \wedge (cond = 1)$	$f_0 \leftarrow s+1$
`pop`	$if^a(s>1), s \leftarrow s-1$	$f_0 \neq s+1$	$f_0 \leftarrow f_0$
	`return`(a_s)	$f_0 = s+1$	$f_0 \leftarrow s+1$

[a] Notons que si le programme est correct, cette condition est toujours vraie.

courante de la pile comme sur la figure 4. L'activité du PE est définie par $\mathcal{A} = \bigwedge_{i=0}^{s-1} a_i$. Le PE est actif si $\mathcal{A} = 1$ et inactif si $\mathcal{A} = 0$.

Par définition les PEs sont tous actif à l'initialisation, donc $s = 1$, $a_0 = 1$ (actif), $f_0 = s + 1$ quand il n'y a pas de 0 dans aucun élément de la pile. Pour des raisons de simplicité un `pop` sur une pile vide retourne *vrai*.

La table 1 résume la sémantique de la pile d'activité. Un PE est actif si et seulement si $f_0 = s + 1$, lorsqu'il n'y a aucun 0 dans la pile. En fait, il est plus intéressant de faire le changement de variable $c = s + 1 - f_0$ parce que seule une comparaison avec 0 est nécessaire. Cette forme est plus facile à réaliser matériellement et même souvent en logiciel [KER 89, KER 92]. Les manipulations de base sur c sont les mêmes que sur f_0: incrémenter or décrémenter, écrire ou lire la valeur (table 2).

Lorsque $c = 0$, `push`(*cond*) peut être simplifié en $c \leftarrow \neg cond$. Une preuve plus détaillée de l'équivalence entre pile d'activité et compteur d'activité pour le contrôle de flot parallèle peut être trouvée dans [BOU 92, LEV 93].

Une méthode utilisant des compteurs gérant l'activité est présentée dans [BEC 92] dans une optique à flot de données, mais la méthode ne permet pas de gérer la récursivité.

Table 2. *Sémantique du* `push` *et du* `pop` *avec un compteur d'activité.*

Opération	Précondition	Action
`push`(*cond*)	$c \neq 0$	$c \leftarrow c+1$
	$(c=0) \wedge (cond=0)$	$c \leftarrow 1$
	$(c=0) \wedge (cond=1)$	$c \leftarrow 0$
`pop`	$c \neq 0$	$c \leftarrow c-1$
	$c = 0$	$c \leftarrow 0$

Table 3. *Réalisation du* `where`/`elsewhere` *avec un compteur d'activité.*

Opération	Précondition		Action
`where`(*cond*)	$c \neq 0$	*(idle)*	$c \leftarrow c+1$
	$c = 0$	*(actif)*	$c \leftarrow \neg cond$
`elsewhere`	$c \leq 1$	*(activable)*	$c \leftarrow \neg c$
	$c \not\leq 1$		$c \leftarrow c$
fin du `where`	$c \neq 0$	*(inactif)*	$c \leftarrow c-1$
/`elsewhere`	$c = 0$	*(actif)*	$c \leftarrow 0$

5.1.2. *Application dataparallèle*

On peut appliquer la méthode au contrôle de flot parallèle de POMPC. On ne présente que la réalisation matérielle et logicielle du `where` et du `switchwhere` mais cela est généralisable aux autres opérateurs [KER 92].

`where` Les opérateurs de bases sont la paire `where`/`elsewhere` qui est trouvée dans la plupart des langages à parallélisme de données de FORTRAN 90 à C*.

Le `where` est équivalent au `push` mais on doit traduire le `elsewhere`. Un PE est actif dans un `elsewhere` si et seulement si le PE était inactif à cause du *dernier* `where`, *i.e.* le niveau d'inactivité $c = 1$. La valeur 1 peut être vue comme une valeur spéciale codant un état « activable » dans le bloc `where` ou `elsewhere`.

Une réalisation est présentée sur la table 3.

`switchwhere` La compilation d'un `switchwhere`, l'extension parallèle du `switch` du langage C, possède aussi plusieurs états. Un PE peut être :

1° inactif avant le `switchwhere`;

2° actif dans un `case` (après avoir reconnu une valeur) ou dans un `default`;

3° inactif dans un `case`, dans l'attente de reconnaître la valeur d'un `case`.

4° inactif dans un `switchwhere` à cause d'un `break`, jusqu'à la sortie du `switchwhere`.

Table 4. *Réalisation du* `switchwhere` *avec un compteur d'activité.*

Opération	Précondition		Action
`switchwhere`(*value*)	$c \neq 0$	*(inactif)*	$c \leftarrow c+2$
	$c = 0$	*(actif)*	$c \leftarrow 1$
`case` *`constante`* `:`	$(c = 1) \wedge$ (*valeur* = *`constante`*)		$c \leftarrow 0$
`break`	$c = 0$	*(actif)*	$c \leftarrow 2$ [a]
`default :`	$c = 1$	*(activable)*	$c \leftarrow 0$
Fermeture du `switchwhere`	$c \leq 1$		$c \leftarrow 0$
	$c \not\leq 1$		$c \leftarrow c-2$

[a] Doit être relatif au bloc `switchwhere` courant, si le `break` est inclus dans un ou plusieurs `where`/`elsewhere`.

Le cas du `break` est similaire à celui du `whilesomewhere`. Un exemple de codage des états utilisés est $c = 1$ pour l'état 3 et $c = 2$ pour l'état 4, comme indiqué sur la table 4.

5.1.3. *Coût matériel*

La méthode du compteur nécessite $\lceil \log_2 c \rceil$ bits par PE si au plus c niveaux de blocs conditionnels parallèles sont imbriqués. Si chaque PE a un opérateur de L bits, un PE a besoin de $\lceil \frac{1}{L} \log_2 c \rceil$ cycles de durée t pour faire une opération sur le compteur.

La pile d'activité ne nécessite que des manipulations de 1 bit sur chaque PE et prend un temps t mais nécessite un pointeur de pile pour gérer la pile. Puisque l'exécution est SIMD, toutes les piles sont synchrones et le pointeur de pile peut être :

— centralisé sur le processeur scalaire qui distribue sa valeur aux PEs;
— distribuée avec des pointeurs locaux qui évoluent de manière synchrone.

Dans le premier cas, cela prend un temps T sur le processeur scalaire et ce temps est négligeable sur les PEs. Dans le second cas, un temps $t\lceil \frac{1}{L} \log_2 c \rceil$ est nécessaire pour contrôler le pointeur de pile sur chaque PE. La complexité matérielle est c pour une pile d'éléments de 1 bit dans chaque cas, plus $\lceil \frac{1}{L} \log_2 c \rceil$ bits pour le pointeur de pile global dans le premier cas et $N\lceil \frac{1}{L} \log_2 c \rceil$ bits pour les pointeurs de pile locaux dans le second cas, pour une machine à N PEs.

La complexité des 3 méthodes précédentes est résumées dans la table 5.

Comme POMP est un ordinateur SIMD à gros grain ($L = 32$), $\lceil \log_2 c \rceil = 1$ et par conséquent la méthode la plus efficace en temps et en espace est bien la méthode du compteur. Mais pour les machines à grain fin c'est la méthode à pile globale qui est utilisée (CM-2, MP-1).

La méthode des compteurs intéresse aussi les machines parallèles MIMD lorsqu'on doit gérer dynamiquement l'activité, par exemple pour la compilation de langages à

Table 5. *Complexité du compteur d'activité comparée aux méthodes à pile.*

Conditionnement parallèle	Complexité en calcul scalaire	Complexité en calcul parallèle	Complexité matérielle	Nombre de diffusions
Pile (pointeur global)	T	t	$Nc + \lceil \log_2 c \rceil$	1
Pile (pointeurs local)	ϵ	$t(1 + \lceil \frac{1}{L} \log_2 c \rceil)$	$N(c + \lceil \log_2 c \rceil)$	0
Compteurs d'activité	ϵ	$t\lceil \frac{1}{L} \log_2 c \rceil$	$N\lceil \log_2 c \rceil$	0

collections, tel que POMPC. En fait, cette méthode est utilisée dans le compilateur pour les machines iPSC et ARMEN.

Une autre méthode plus générale à compteur est utilisée dans une approche flot de données [BEC 92] pour du contrôle dynamique d'exécution mais il n'y a pas de possibilité de récursivité ou d'appels de fonctions non connues à la compilation.

5.2. *Optimisation de blocs conditionnels terminaux*

La méthode précédente optimise les imbrications de blocs conditionnels mais il existe aussi les blocs conditionnels terminaux, qui ne contiennent aucun autre opérateur de conditionnement ou d'appel de procédure ou fonction. Comme c'est un cas qui est courant, il faut optimiser ce cas.

5.2.1. *Branchement conditionnel non retardé*

Les processeurs pipelinés possèdent au moins un des 2 types de branchement suivants : les branchements non retardés (« normaux ») et les branchements retardés. Les premiers nécessitent de vider le pipeline des instructions en cours de décodage pour conserver la sémantique du branchement tandis que les derniers changent la sémantique en retardant l'instant où est pris le branchement.

Dans une philosophie SIMD le branchement conditionnel non retardé peut être utilisé pour faire du vidage de pipeline conditionnel, c'est à dire inhiber sélectivement l'exécution d'une instruction.

Le processeur choisi pour POMP a une profondeur de pipeline de branchement de 1 ce qui signifie que seule l'instruction suivant le branchement pourra être sautée ou pas selon une condition locale. Cela peut sembler restrictif mais la méthode évite la traversée du pipeline nécessaire pour mettre à jour le compteur externe en fonction de la condition locale.

La méthode du branchement conditionnel non retardé a une visibilité faible et ne peut contrôler d'autres branchements, donc des appels de fonctions ou d'autres blocs conditionnés. C'est pour cela qu'elle se limite aux petits blocs terminaux.

Dans POMP cette méthode est plus intéressante que celle du compteur pour les blocs conditionnés d'au plus 5 instructions [KER 92].

5.2.2. *Entrelacement de blocs alternatifs terminaux*

Il ne s'agit pas d'une nouvelle méthode, mais plutôt d'une optimisation de la méthode du compteur. Si on a 2 blocs terminaux ne concernant qu'une collection et qu'il n'y a pas de communication, on sait que chaque code s'exécutera de manière exclusive. On peut donc entrelacer les 2 codes qui s'exécuteront correctement car chaque instruction est étiquetée au niveau de l'HYPERCOM comme appartenant à un `where` ou un `elsewhere`.

L'intérêt est que souvent dans une application il y a des trous dans le pipeline d'un processeur scalaire à cause des dépendances entre instructions. L'idée est que si on peut remplir les trous du bloc `where` avec des instructions du bloc `elsewhere` on peut aller jusqu'à diviser la taille du code et donc avoir du code SIMD aussi efficace que du code MIMD sur des `where`/`elsewhere` terminaux.

La méthode est plus efficace que l'exécution spéculative compilée dans le cas d'une machine séquentielle puisqu'aucun registre supplémentaire n'est nécessaire. Elle est aussi plus simple.

6. Génération de code

L'architecture minimale ayant été décrite, on peut survoler la génération de code. Pour des raisons de concision, seuls les aspects proches du matériel sont considérés ici.

6.1. *Virtualisation*

Selon les architectures, on a intérêt à avoir la boucle qui énumère les éléments de variables parallèle au niveau d'une instruction (virtualisation en largeur d'abord) ou bien au niveau d'un bloc d'instructions parallèles (virtualisation en profondeur d'abord). Cette énumération est nécessaire si les variables ont une taille supérieure à celle de la machine, ce qui est souvent le cas.

Si les processeurs ont peu de registres ou que le débit d'instructions est faible, on a tout intérêt à virtualiser en largeur : c'est le cas de la CM-2, du CYBER-205.

Si par contre on a beaucoup de registres, on veut bénéficier de « l'effet registre » et donc limiter les accès à la mémoire : c'est le cas du CRAY-1 ou des machines MIMD.

Sur POMP, c'est la virtualisation en profondeur d'abord qui est par conséquent choisie.

6.2. *Infrastructure de génération du code*

Elle est résumée sur la figure 5. Le principe est de récupérer le maximum de l'environnement logiciel du MC88100 : compilateur C, assembleur, éditeur de lien, bibliothèques mathématiques et systèmes.

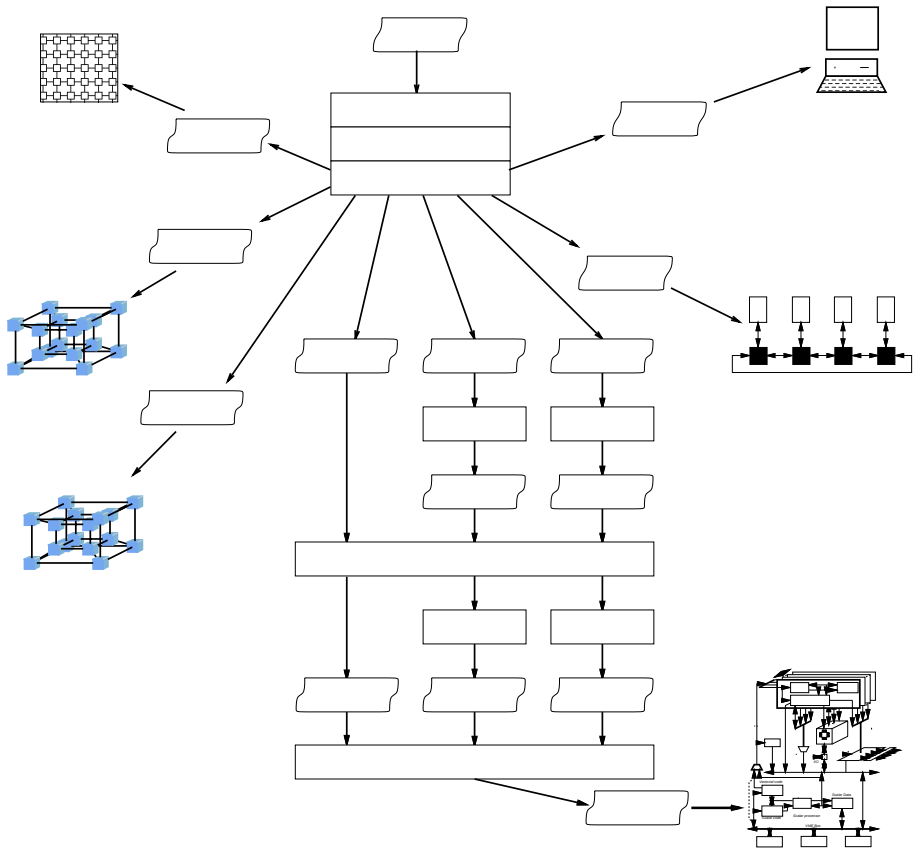

Figure 5. *Architecture de la génération de code à partir de* POMPC.

Le compilateur POMPC est multicible et génère des fichiers en langage de type C selon l'architecture de la machine (C* pour la CM-1, MPL pour la MP-1) et des codes C pour POMP : du code séquentiel et du code pour les PEs, plus des instructions de contrôle pour l'HYPERCOM.

Le langage C est donc utilisé ici comme un macro-assembleur portable sur différentes machines.

6.3. *Synchronisation du code*

Comme la machine est SIMD, donc synchrone, mais que les processeurs ne le sont pas *a priori*, il faut ressynchroniser le code à la compilation.

D'une part il faut synchroniser le code entre processeur scalaire et PEs (mode VLIW), d'autre part il faut synchroniser le code entre les PEs (mode SIMD). L'écriture d'un compilateur complet pour POMP ne serait pas une approche pragmatique car les

problèmes de planifications d'instructions sont trop complexes pour une petite équipe de recherche.

Les points essentiels de synchronisations sont indiqués par le compilateur POMPC sous forme de pseudo-fonctions (par exemple `pc_synchro_1587`) qui étiquettent les codes au niveau des appels de fonction, des émissions et réceptions scalaires, des bornes de blocs conditionnés.

Puisque ces informations se retrouvent au niveau du langage d'assemblage, un programme de synchronisation à ce niveau convient parfaitement, d'autant plus qu'un analyseur syntaxique pour ce langage est particulièrement simple. De plus toute la génération et l'optimisation fine du code est déjà toute faite par le compilateur natif du processeur.

Le code scalaire, parallèle et celui de l'HYPERCOM sont d'abord mis en correspondance au niveau des blocs. Ensuite le mécanisme d'allocation temporelle statique utilise une méthode de relaxation utilisant une diminution progressive et répartie des contraintes. Par mesure de simplification, aucune réorganisation du code n'est faite et on fait confiance au compilateur C. Le seul travail à faire est « d'écarter » dans le temps les instructions l'unes de l'autres en rajoutant des instructions nulles pour respecter les contraintes conjoncturelles de dépendances entre registres et les contraintes structurelles du processeur (liées au nombre borné de ressources). Ce problème n'a pas lieu dans le cas d'une machine « normale » à MC88100 car ce dernier résoud toutes les dèpendances par un mécanisme de *scoreboard*.

En fait, les contraintes conjoncturelles sont les plus fréquentes et elles ont la particularité que si elles sont vérifiées elles le resteront même si on éloignent 2 instructions. La relaxation est faite en 2 passes :

1° on satisfait les dépendances de données en retardant pour chaque instruction une seule instruction conflictuelle jusqu'à la solution en itérant sur le graphe de dépendance quotient ;

2° on satisfait ensuite les contraintes structurelles de la même manière.

Comme l'influence d'une instruction est courte, un parcours en profondeur d'abord des graphes suffit. L'algorithme est linéaire en le nombre d'instructions [KER 92].

6.4. *Environnement de programmation*

6.4.1. *Mise au point des programmes*

L'intérêt d'avoir un processeur scalaire qui est une station de travail permet de récupérer le déboggueur symbolique de la machine cible et de l'adapter à un rôle plus parallèle. Cela est possible en rajoutant à tout programme POMPC des routines capables d'afficher des variables parallèles sous forme graphique ou textuelle. Ces routines sont cachées par les alias du déboggueur et sont capables de rechercher dans une table des symboles parallèles portable les adresses et les collections des variables considérées.

Malheureusement, si c'est le cas sur la CM-2 ou en simulation sur station de travail, sur POMP ou sur MP-1 cela n'est pas possible car le processeur scalaire n'est pas la station de travail.

Dans le cas de POMP, un petit déboggueur a été développé pour tourner sur POMP. Il mime le comportement du `dbx` de SUN et par conséquent on peut récupérer tout l'emballage graphique de `dbxtool`. Le fonctionnement se fait par l'intermédiaire d'échanges de messages entre le chargeur du SUN et la partie du code sur POMP.

6.4.2. *Développement système*

L'idée de base était de récupérer tout l'environnement système de l'hôte. Comme le compilateur MC88100 était livré avec les bibliothèques UNIX classiques, il n'y avait qu'à écrire les appels systèmes.

Tous les appels systèmes indispensables sont en fait sous-traités au SUN qui les exécutent par l'intermédiaire de la mémoire scalaire de POMP qui est aussi visible par le SUN. Les données éventuelles échangées passent par cette mémoire.

La seule simplification faite au niveau matériel est que la mémoire n'est accessible au niveau du SUN que par mots de 32 bits. Les autres accès, plus rares, sont simulés logiciellement.

7. Réseau d'interconnexion

Le dernier point mais néanmoins primordial dans une machine parallèle est le réseau, qui permet aux processeurs d'échanger des informations. Souvent il s'agit là du facteur limitant les performances.

7.1. *Quelques classes de réseaux*

Les réseaux peuvent être répartis en 2 classes : les réseaux statiques et les réseaux dynamiques. Les premiers ne permettent des communications qu'entre certains PEs suivant un schéma établi alors que les derniers permettent de relier les PEs selon différents motifs.

Ils peuvent aussi être répartis selon leur mode de commutation : par paquets, où les messages se propagent de nœud en nœud, ou bien par circuits, où les messages passent dans des circuits ouverts et l'allocation du chemin n'est faite qu'une fois.

Comme notre architecture est synchrone, on s'est orienté vers un réseau synchrone mais avec contrôle réparti, plus simple à réaliser.

Au niveau du modèle de programmation on a séparé les communications régulières, de type grille, des communications irrégulières pour des questions d'optimisation. Il serait intéressant de trouver un réseau permettant une optimisation de ces 2 cas.

Malheureusement, les communications régulières se passent très bien de routage et sont plus efficaces sur un réseau statique de topologie proche, alors que les communications dynamiques nécessitent un mécanisme de routage dynamique pour être rapides.

Un réseau qui pourrait etre configuré comme statique ou bien dynamique au cours des calculs et dont la conception et le routage soit simple serait parfait. On peut préciser

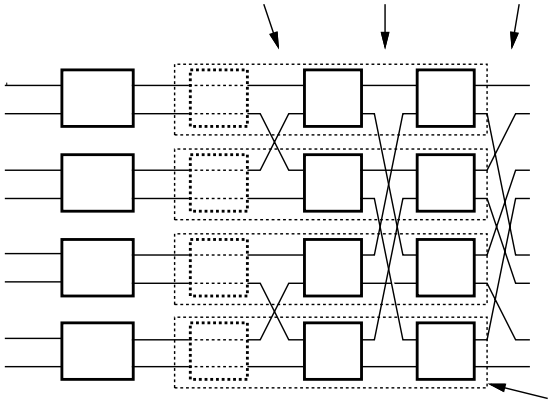

Figure 6. *Synoptique du réseau de* POMP *pour* $N = 4$ PE*s et* $k = 2$ *liens par commutateurs.*

que le réseau n'a pas besoin d'être particulièrement extensible car le tout ne doit pas dépasser la taille d'une station de travail.

7.2. *Un réseau hybride statique et dynamique*

En ce qui concerne le réseau dynamique, un réseau multi-étage de type figuier banian [9] est intéressant car la conception est simple et éprouvée, ainsi que le routage DTA.

Un cas particulier est le réseau de type *indirect binary n-cube* [PEA 77] qui permet de voir le réseau statique sous-jacent à ce réseau dynamique, l'énumération des liens d'un hypercube en l'occurrence. Un tel réseau est constitué de commutateurs à croisillons de 2×2 liens par exemple et possède $\frac{N}{2} \log_2 N$ commutateurs.

Malheureusement, le seul hypercube du réseau statique n'a que $\log_2 N - 1$ dimensions alors que pour N processeurs on pourrait utiliser pleinement ce réseau s'il avait $\log_2 N$ dimensions.

L'idée est donc de doubler ce réseau en ayant non plus un lien par PE mais 2. Le fait d'avoir 2 liens n'est pas gênant dans la mesure où on s'attaque à des problèmes massivement parallèles : il y aura souvent au moins 2 PVs par PE et donc les 2 liens seront utilisés en parallèle.

L'architecture du réseau hybride est représentée sur la figure 6. Tel que les commutateurs sont arrangés dans les HYPERCOM, seuls les liens de l'hypercube sortent des PEs ainsi que le battage parfait. Le réseau est bien entendu généralisable avec des commutateurs à croisillons à $k \times k$ liens.

9. Car sa forme rappelle les racines adventives retombantes de ce figuier indien.

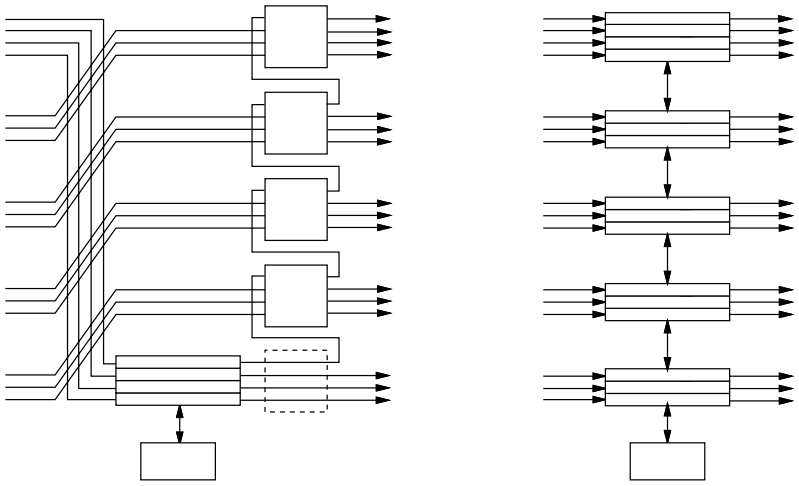

Figure 7. *Les 2 modes du réseau de* POMP *pour* $N = 256$ PE*s et* $k = 2$ *liens par commutateurs.*

Selon un bit de mode interne aux HYPERCOM, le réseau est vu comme multi-étage de type *indirect binary n-cube* (communications générales) ou comme un réseau de type hypercube de base k à $\log_k N$ dimensions (communications régulières à travers les codes de GRAY) plus un battage parfait, utile pour les calculs de FFTs par exemple. Les 2 configurations internes sont représentées sur la figure 7.

Les simulations du réseau montrent que si on a des liens de $\mathcal{F}$ fils de large à 25 MHz comme le reste de la machine et 256 PEs, avoir k et $\mathcal{F} \leq 4$ suffit car sinon on sature le bus mémoire du MC88100 ou la routine de gestion du réseau.

En considérant les hypothèses conservatrices $k = 2$ et $\mathcal{F} = 2$ et une fréquence de 25 MHz, le réseau statique permet 16 Mo/s/PE soit 16 % de la bande passante mémoire crête et le réseau dynamique sur des communications aléatoires de 4 Mo/s/PE.

Tout le réseau est fait en logique reconfigurable. Cela permet en fait d'adapter la taille de la machine en utilisant un nombre de pattes constant par PE. En fait, si une application nécessite d'optimiser un certain motif de communication, le programmeur peut très bien les implanter dans l'HYPERCOM. D'autres fonctionnalités plus complexes peuvent être rajoutées comme cela est fait dans ARMEN [POT 91].

8. Réalisation

La mise en boîtier prévue de la machine utilise un code de GRAY pour disposer les cartes afin d'éviter que les câbles du réseau ne se croisent.

La conception d'un petit prototype de la machine à 3 processeurs à été entrepris en technologie « wrappée » pour des raisons de facilité de mise au point incrémentale.

Cela est important lorsqu'on ne dispose d'aucun outil de simulation logicielle.

Malheureusement, cette technologie est assez mal adaptée aux signaux extrêmement rapides qui sont caractéristiques des cartes à processeurs RISC modernes. Du coup, la technologie devient inadaptée à la mise au point car tout fil trop long rajouté devient une inductance potentielle : les parasites et la polyphonie s'installent... Alors que la partie scalaire était réalisée en seulement un mois, énormément de temps (étalé sur un an et demi) a été consacré à la mise au point de la machine pour n'aboutir qu'à une partie scalaire qui marche souvent.

Alors qu'il n'y a encore quelques années les « manipulations de coin de table » étaient relativement simple pour une petite équipe, l'arrivée de processeurs rapides rend ce travail impossible avec peu d'infrastructure. C'est bien dommage car c'est en réalisant une machine que l'on acquiert de nombreuses connaissances difficilement accessibles autrement et que l'on conserve son pragmatisme.

9. Conclusion

Le projet POMP a amené de nombreux résultats dans plusieurs domaines, même si la machine n'est pas terminée.

Au niveau logiciel un nouveau langage portable à parallélisme de données a été développé et des compilateurs vers plusieurs machines écrits. Au niveau des applications, un certain nombre d'applications ont été développées en POMPC pour la machine (codes de différences finies, méthodes multi-résolutions, gaz sur réseau, etc.) qui ont démontré concrètement l'intérêt d'un tel langage associé à une machine parallèle comme POMP.

À la limite du matériel et de la compilation, une nouvelle méthode de planification dynamique de contrôle de flot à parallélisme de données a été mise en pratique.

L'architecture choisie est de type SIMD car elle permet une bonne compacité en performances de calcul tout en étant adaptée à nos applications. Mais le parallélisme à gros grain étant souvent plus efficace que le parallélisme à grain fin, l'adaptation de processeurs RISC du commerce a été faite.

En plus de la méthode du compteur d'activité, une méthode de contrôle de flot basée exploitant une gestion fine du pipeline associée à une méthode de compaction a permis d'augmenter le rendement de la machine.

Un couplage de type VLIW a permis de simplifier au maximum la partie contrôle puisque processeur scalaire et séquenceur ne sont plus qu'un autre processeur RISC.

Afin de rendre l'exécution du code déterministe à moindre frais, la chaîne de compilation du processeur est récupérée avec un passe terminale de planification statique pour synchroniser le code.

Les innovations apportées dans de nombreux domaines permettent globalement la conception d'une machine efficace, contrôlable et plus facilement réalisable car aucun circuit ou processeur spécifique n'est nécessaire à sa conception. Les principes peuvent être utilisés pour le développement d'autres machines synchrones générales et d'autres environnements de programmation.

10. Remerciements

L'auteur tient à remercier plus particulièrement son ancien collègue de bureau Nicolas PARIS avec qui il a partagé son enthousiasme fébrile et son temps sur ce projet vaste et passionnant. Sans lui ce travail n'aurait pas été possible.

Bien entendu, les autres membres de l'equipe ayant touché de près ou de loin au projet ne sont pas oubliés avec toutes leurs discussions positives : Philippe MATHERAT, Philippe HOOGVORST, César DOUADY, Patrice OSSONA DEMENDEZ, Théodore PAPADOPOULO et Pierre CHICOURRAT.

Il faut remercier aussi Luc BOUGÉ et son équipe, spécifiquement Jean-Luc LEVAIRE, pour leur discussions pointilleuses en ce qui concerne la sémantique des langages SIMD et leur contrôle de flot, et de manière générale leur intérêt pour le domaine. Luc BOUGÉ a en outre consacré beaucoup de temps (probablement ingrat) à rassembler des chercheurs du domaine et les rappeler à l'ordre régulièrement pour faire ce numéro spécial de TSI.

Merci enfin aux relecteurs anonymes de la revue pour leurs remarques et leurs corrections, ainsi qu'à toute la rédaction de TSI, même si l'adaptation du style LaTeX et la création du style BibTeX adéquat n'a pas été de tout repos.

Ce travail a été réalisé lorsque l'auteur était dans le Laboratoire d'Informatique de l'École Normale Supérieure, 45 Rue d'ULM, 75005 PARIS, FRANCE. Ces recherches et le projet POMP ont été financés partiellement par le Ministère de la Recherche et de la Technologie, le Centre National de la Recherche Scientifique (CNRS), le Laboratoire d'Informatique de l'École Normale Supérieure (LIENS), le Département de Mathématique et Informatique (DMI) de l'ENS, l'École Normale Supérieure, Thomson Digital Image (TDI), le Programme de Recherche Coordonnées en Architectures Nouvelles de Machines (PRC-ANM) et moins directement le Laboratoire d'Informatique de l'École des Mines de Paris de par l'accès libre à sa bonne bibliothèque et sa moins bonne photocopieuse...

11. Bibliographie

[AKE 88] Kurt AKELEY et Tom JERMOLUK. « High-Performance Polygon Rendering ». Dans *Computer Graphics (SIGGRAPH '88)*, volume 22(4), pages 239–246. ACM, août 1988.

[ALL 83] J. R. ALLEN, Ken KENNEDY, Carrie PORTERFIELD, et Joe WARREN. « Conversion of Control Dependence to Data Dependence ». Dans *Conference Record of the Tenth Annual ACM Symposium on Principles Of Programming Languages*. Association for Computing Machinery, janvier 1983.

[AMD 67] 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.

[AUG 90] Michel AUGUIN, Fernand BOERI, et Jean Paul DALBAN. « Synthèse du projet Opsila ». *Techniques et Sciences Informatiques*, 09(02):79–98, 1990.

[BEC 92] Carl J. BECKMANN et Constantine D. POLYCHRONOPOULOS. « Microarchitecture Support for Dynamic Scheduling of Acyclic Task Graphs ». Dans *The 25 th Annual In-*

ternational Symposium on Microarchitecture, volume 23(1-2), pages 140–148. ACM SIG MICRO Newsletter, décembre 1992.

[BLE 89] 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.

[BOU 92] Luc BOUGÉ et Jean-Luc LEVAIRE. « Control structures for data-parallel SIMD languages: semantics and implementation ». *Future Generation Computer Systems*, 8(3-4):363–378, 1992.

[BRE 74] Richard P. BRENT. « The Parallel Evaluation of General Arithmetic Expressions ». *Journal of the ACM*, 21(2):201–206, avril 1974.

[FLY 72] Michael J. FLYNN. « Some Computer Organizations and Their Effectiveness ». *IEEE Transactions on Computers*, C-21(9):948–960, septembre 1972.

[FOX 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`.

[HEN 90] John L. HENNESSY, David A. PATTERSON, et David GOLDBERG. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[KEN 90] Ken KENNEDY et Kathryn S. MCKINLEY. « Loop Distribution with Arbitrary Control Flow ». Dans *Proceedings of Supercomputing '90*. The Institute of Electrical and Electronics Engineers, Inc., novembre 1990.

[KER 88] Ronan KERYELL. « POMP : Vidéo & Entrées-Sorties ». Diplôme d'étude approfondie, Paris XI, septembre 1988.

[KER 89] Ronan KERYELL. « POMP2 : D'un Petit Ordinateur Massivement Parallèle ». Rapport de magistère, LIENS — Ecole Normale Supérieure, octobre 1989.

[KER 92] Ronan KERYELL. « *POMP : d'un Petit Ordinateur Massivement Parallèle SIMD à Base de Processeurs RISC — Concepts, Etude et Réalisation* ». Thèse, Laboratoire d'Informatique de l'Ecole Normale Supérieure — Université Paris XI, octobre 1992.

[KER 93] Ronan KERYELL et Nicolas PARIS. « Activity Counter : New Optimization for the Dynamic Scheduling of SIMD Control Flow ». Dans *1993 International Conference on Parallel Processing*, pages II–184–II–187, août 1993.

[KRU 85] Clyde P. KRUSKAL, Larry RUDOLPH, et Marc SNYR. « The Power of Parallel Prefix ». *IEEE Transactions on Computers*, C-34(10):965–968, octobre 1985.

[LEV 93] Jean-Luc LEVAIRE. « *Contribution à l'étude sémantique des langages á parallélisme de données; application à la compilation* ». Thèse, LIP — ENS Lyon, Université de Paris 7, février 1993.

[PEA 77] Marshall C. PEASE, III. « The Indirect Binary n-Cube Microprocessor Array ». *IEEE Transactions on Computers*, C-26(5):458–473, mai 1977.

[POT 91] Bernard POTTIER. « *ArMen : Une machine parallèle intégrant un réseau de circuits logiques programmables* ». Thèse, Université de Rennes I, juin 1991.

[SAB 92] Gary SABOT. « Optimized CM Fortran Compiler for the Connection Machine Computer ». Dans *Proceedings of the 25th Hawaii International Conference on System Sciences*, pages 161–172. The Institute of Electrical and Electronics Engineers, Inc., janvier 1992.

[SIE 84] 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.

[SLO 62] 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.

[UNG 58] S. H. UNGER. « A Computer Oriented Toward Spatial Problems ». Dans *Proceedings of the IRE*, pages 1744–1750, octobre 1958.

[ZIM 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.