POMP
un Petit Ordinateur Massivement Parallèle SIMD

Ronan Keryell, Philippe Matherat et Nicolas Paris

—

Laboratoire d'Informatique de l'Ecole Normale Supérieure

45 rue d'Ulm 75005 PARIS
Tél : (33.1) 43.26.58.85

E.mail : ...@ens.ens.fr
...@FRULM63.BITNET

—

30 mars 1990

Résumé

La machine POMP est une petite machine SIMD générale. Elle utilise des processeurs 32 bits connectés entre eux par un hypercube. Un modèle de programmation est défini et permet la création du langage POMPC, extension de C pour cette machine.

1 But de l'architecture

Pour bien comprendre la motivation de nos choix pour ce projet, commençons par introduire notre préoccupation première.

1.1 Les origines du projet : les architectures graphiques

Nous abordons le problème des architectures parallèles sous le point de vue de l'augmentation de puissance des machines graphiques. Notre équipe a depuis longtemps comme préoccupation la définition d'architectures graphiques. Le point crucial de nos études était davantage basé sur l'organisation intrinsèque d'une mémoire écran que sur des accélérateurs de calcul tri-dimmensionnel [Cla80,CH80,JHH80]. Nous avons ainsi étudié les couplages unité centrale–mémoire écran, qui ont conduit aux architectures du 9365 [Mat80], de Flip [Par89] et de Flip2 [DDPP86]. La problématique de ces architectures se situe en deux points :

- une organisation de la mémoire en bancs, avec une architecture de type *tranches* de 1 bit d'opérateurs pour chaque banc,
- des opérateurs câblés et spécialisés pour des fonctions très particulières (BitBlt,...) et intrisèquement graphiques.

1.2 Les conclusions sur ces architectures graphiques

Dans ces machines, l'organisation générale de la mémoire nous plaît car elle répond bien dans le cadre des architectures de visualisation, à une problématique plus générale : le goulot d'étranglement entre processeur et mémoire dans les machines de Von Neuman. L'organisation de la mémoire se trouve généralement au cœur de la discussion sur les architectures parallèles.

Par contre, la spécialisation des opérateurs, même si elle nous a semblé raisonnable dans le cadre d'entrées–sorties haut débit, nous semble un inconvénient si nous désirons élargir notre point de vue. En effet, si nous désirons étendre notre structure de *smart memory*[1] sur des architectures plus complexes et plus générales, nous devons bannir toute spécialisation. Ceci est d'autant plus vrai dans le cadre de la synthèse d'image, dans lequel les algorithmes évoluent constamment, ainsi que l'organisation de la représentation des données.

En conséquence, même si la finalité de notre projet est d'offrir une machine pour l'image de synthèse, nous avons à cœur de faire avant tout une machine générale; non pas qu'elle puisse être nécessairement efficace sur tout algorithme (nous cherchons à proposer une machine originale), mais qu'elle se présente sous la forme d'une machine informatique programmable dans le sens le plus général du terme.

[1] L'association d'opérateurs et de mémoires en couplage étroit.

1.3 Les choix de base pour notre machine

Notre démarche peut donc clairement s'inscrire dans les différents mouvements que l'on rencontre dans les architectures parallèles.

Nous visons un couplage étroit entre de nombreuses unités de calcul et leur mémoire locale. Nous sommes donc assez voisin de certaines machines cellulaires [FP81,FEG88].

Nous affirmons clairement une indépendance vis à vis de l'algorithmique. Nous voulons offrir un environnement général de programmation assez classique (langage de haut niveau) et ne pas être classé dans la catégorie des machines de traitement d'images. Nous sommes donc plus proche de la Connection Machine [Hil85]. Nous pousserons même plus loin cet esprit de généralité en préférant a priori une structure à grain plus classique, mieux à même de répondre aux besoins des calculs numériques scientifiques, en particulier ceux de la synthèse d'image. Il nous semble plus judicieux d'adopter un grain plus large (entier 32 bits, flottants 32 et 64 bits).

Il nous reste enfin à choisir le type de parallélisme. Notre but initial est d'essayer d'étendre le concept de *smart memory* à des problèmes nettement plus compliqués que celui de la visualisation d'images. L'aspect SIMD ressort très nettement de cette structure et nous continuerons à l'utiliser au regard de ces nouveaux arguments :

- la programmation grandement simplifiée par l'unicité du code à écrire,
- la synchronisation naturelle des processeurs lors des communications,
- un excellent rapport de puissance globale en terme de Mflops/dm^3.

Maintenant que le cadre de notre recherche est placé, nous pouvons présenter les choix préliminaires de notre architecture.

2 Choix préliminaires de l'architecture

Voici les grandes lignes de l'architecture :

- processeurs 32 bits,
- couplage haut débit avec une mémoire locale,
- 64 à 256 processeurs,
- Petite machine (station de travail),
- Réseau d'interconnexion genre hypercube,
- vidéo couplée en temps réel au tableau de processeurs.

La définition plus fine de la machine va dépendre de comment on désire la programmer. Pour cela, on a besoin de définir maintenant un modèle de programmation, un langage de programmation et enfin la manière de générer le code pour cette machine. En effet, la structure de chaque processeur du tableau doit dépendre des débits de données

nécessaires, qui sont très fortement liés à la qualité du code généré[2]. Une des questions de bases pour ce genre d'architecture est de savoir si on peut se servir efficacement de registres pour effectuer des opérations sur de grands tableaux. Pour y répondre, nous allons donc consacrer les deux prochaines sections à ce but.

Nous préférons fixer dès maintenant nos ambitions au niveau de la programmabilité avant de fixer les détails de l'architecture, plutôt que d'être bridé a posteriori.

3 Le modèle de programmation

On peut énumérer les besoins de la programmation comme suit :

- faire des calculs séquentiels classiques,
- faire des calculs vectoriels classiques,
- convertir une variable scalaire en une variable vectorielle,
- compacter une variable vectorielle en une variable scalaire,
- faire interagir des variables vectorielles,
- contrôler l'exécution du programme,
- conditionner les calculs vectoriels.

3.1 la partition des données

La base du modèle de programmation va reposer sur le partitionnement des donnnées en 2 catégories :

- les données scalaires résidant sur le processeur scalaire,
- les données vectorielles réparties sur le tableau de processeurs.

On désire faire des calculs entre ces différentes catégories de données et on aimerait que les expressions de ces calculs dans la syntaxe du langage de programmation soient semblables. Il est alors naturel de déclarer l'appartenance de chaque variable à une des deux catégories : le meilleur endroit pour déclarer cette information est évidemment lors de la déclaration de la variable. En plus de son type, on doit maintenant préciser sa catégorie. Par défaut, une variable sera scalaire.

Pour les variables vectorielles se posent deux problèmes :

- La taille des vecteurs doit être indépendante du nombre de processeurs physiques. Ce problème est résolu dans la Connection Machine grâce au mécanisme des processeurs virtuels.

[2]Une amélioration de la génération du code nous a permis de gagner jusqu'à un facteur 8 dans certains cas.

- Il faut pouvoir faire coexister et interagir différents formats de vecteurs dans un même programme. Ce problème est résolu dans la Connection Machine par la possibilité de manipuler plusieurs ensembles de processeurs virtuels (*vp_set*).

Il nous semble intéressant d'offrir simultanément plusieurs formats de vecteurs. On définit alors une partition naturelle de l'ensemble des vecteurs d'un programme. Nous appellerons par la suite *collection* chaque classe de cette partition. Il faut pour chaque variable déclarer (si elle n'est pas scalaire) son appartenance à une collection.

Les vecteurs d'une même collection vont avoir nécessairement la même taille. Ils vont partager également la même activité. De cette manière, les collections vont représenter des ensembles de vecteurs possédant les mêmes caractéristiques sémantiques.

Pour bien préciser la sémantique de l'activité d'un vecteur, nous allons proposer un modèle de processeurs virtuels.

3.2 le modèle des processeurs virtuels

La machine est composée d'un processeur scalaire et d'autant de tableaux de processeurs qu'il y a de collections. Chaque tableau de processeurs possède un vecteur qui définit l'activité de chacun de ses éléments.

Dans ce modèle, on distingue deux types d'interactions :

- les interactions directes : les calculs entre scalaires et entre vecteurs d'une même collection. L'activité de l'interaction est alors parfaitement définie.

- les interactions indirectes entre variables qui ne sont pas localisées sur les mêmes unités de calcul : interactions entre scalaires et vecteurs, interactions entre vecteurs de collections différentes. Dans ce cas, il y a effectivement communication. Il faut alors préciser dans chacun des types de communication, l'activité de l'interaction.

Nous préférons que les communications apparaissent explicitement dans le langage de programmation, car elles vont avoir beaucoup d'impacts sur la performance d'un programme.

Un programme dans ce modèle doit en principe consister en une série de codes, un pour le processeur scalaire et un pour chaque tableau de processeurs virtuels (un par collection). Il faut en plus préciser la synchronisation entre tous ces codes : la méthode la plus élégante consiste à mélanger tous ces codes en un seul. La synchronisation est alors naturellement effectuée par l'ordre d'énumération des instructions dans le programme. Il faut néanmoins pouvoir sans ambiguïté déterminer pour chaque instruction le tableau de processeurs virtuels (l'activité) dont elle dépend. Cette opération va pouvoir être réalisée grâce à un mécanisme de typage, qui va permettre d'établir l'appartenance à une collection de proche en proche depuis les variables jusqu'aux instructions atomiques en passant par les expressions.

3.3 Le typage des variables, des expressions et des instructions atomiques

L'appartenance à une collection est précisée directement lors de la déclaration de chaque variable. Des règles simples vont permettre de composer les types de telle sorte qu'un vecteur ne puisse être affecté à un scalaire et que deux vecteurs de collections différentes (c'est à dire résidant sur des tableaux de processeurs virtuels différents) ne puissent être composés. Une seule communication peut intervenir de manière non-explicite : c'est la conversion d'un scalaire en un vecteur. Cette communication ne pose aucun problème de performances, car elle est réalisée par un *broadcast* dans le flux de code commun à tous les processeurs.

Toute instruction atomique est constituée d'une expression et hérite de celle-ci son appartenance à une collection.

Maintenant que le cadre est défini, nous pouvons présenter le langage de programmation de la machine : RMPC [Par90].

4 Le langage RMPC

Nous avons décidé de prendre comme langage de programmation une extension du langage C. Nous avons choisi C parce que c'est le langage de haut niveau le plus proche des capacités du matériel. Tout en apportant la puissance du symbolique, il permet d'accéder pleinement aux potentialité du matériel. Ce langage est considéré pour nous comme le *macro-assembleur*[3] de RMP. Notre ambition est d'offrir une plate-forme efficace pour le développement de langages ultérieurs de plus haut niveau. Nous voulons que cette plate-forme offre néanmoins les possibilités de programmation des langages de niveau résonnable (C, Pascal,...).

Nous cherchons également la souplesse du développement en environnement C :

- compilation séparée (éventuellement avec d'autres langages),
- environnement Unix (appels système, accès au réseau, `lex`, `yacc` et `dbx`, etc.).

Enfin ce choix nous permet d'envisager des simulateurs de la machine en C et un émulateur en CPARIS sur la Connection Machine.

RMPC, de ce point de vue, est assez similaire à C* [Tmc87], un des langages de programmation de la Connection Machine. RMPC laisse beaucoup moins de liberté au programmeur par le fait que les mémoires locales ne sont pas vue comme une seule et même mémoire. On ne peut déclencher de communications dans RMPC sans qu'elles soient explicitement apparentes dans le texte du programme.

4.1 Collections et typage

Une collection est définie de la manière suivante :

```
collection pixel;
```

[3] Certains considérent C comme un *macro-assembleur* portable

`pixel` devient alors un symbole du langage. Il est utilisé de deux manières :

- dans la déclaration de chaque variable de cette collection :

```
pixel int a,*b,d[4];
```

- dans l'accès à la structure concentrant les informations de la collection :

```
a = ?pixel->position;
```

Cette structure contient des informations sur la taille et la géométrie des vecteurs ainsi que deux vecteurs : l'activité du processeur et un vecteur d'indices qui précise la position de chaque élément d'un vecteur.

Par le mécanisme de typage décrit dans la section précédente, on remonte l'information de l'appartenance à une collection jusqu'aux instructions atomiques. L'exécution de celles-ci pour chaque élément du vecteur dépend de l'activité de la collection correspondante. La modification de l'activité d'une collection va être réalisée grâce à l'opérateur **where** qui est un *if* vectoriel et que nous allons décrire dans la prochaine section.

4.2 Contrôle de flot

Nous avons recherché à étendre au maximum le contrôle de flot pour le vectoriel. La seule manière de différencier le comportement des processeurs est l'emploi de l'activité. La construction **where** sera la clé de voûte du contrôle de flot vectoriel.

4.2.1 Where : le *if* vectoriel

Par l'opérateur **where** nous allons modifier l'activité d'une collection sur tout un bloc d'instructions, d'après un vecteur booléen de cette collection. Le corps du **where** sera systématiquement éxécuté : en effet, seules les opérations sur la collection en question verront leur activité modifiée. Les autres opérations seront éxécutées (systématiquement si elles sont scalaires, ou suivant leurs propres activités si elles sont vectorielles). Le **where** n'est en fait qu'un modificateur temporaire de l'activité d'une collection. L'activité initiale est restaurée à la fin du bloc. Elle doit être sauvegardée dans une pile.

L'activité d'un élément donné d'une collection est initialement à *vrai* tant qu'on ne rentre pas dans un bloc conditionné par un **where** invalidant l'élément en question. A partir de cet instant, tous les blocs imbriqués dans celui-la seront inactifs. Si on étudie l'état de la pile d'activité du processeur, on constate qu'elle est remplie depuis le bas avec des *vrai* jusqu'au bloc invalidant l'élément; on trouve alors des *faux* dans la pile jusqu'à son sommet. Par mesure de simplicité et d'économie de mémoire, nous remplaçons la pile de booléens par un compteur qui dénombre la quantité de *faux* au sommet de la pile. On peut interpréter ce nombre comme étant le nombre de sorties de blocs conditionnés par un **where** qu'il faut franchir avant de redevenir actif. L'ouverture d'un **where**, **elsewhere** et la fermeture du bloc vont agir sur la valeur du compteur :

- à l'ouverture du **where**. Si le compteur n'est pas nul (on est déja inactif), on incrémente le compteur (on note le fait qu'on entre dans un bloc supplémentaire où on est inactif). Sinon, on incrémente le compteur si le booléen est faux (on devient inactif) ou on garde le compteur à 0 si le test est positif (on reste alors actif).

- au **elsewhere**. Si la valeur du compteur est plus grande que 1 (on était déjà inactif à l'ouverture du **where**), on reste inactif avec cette valeur. Si celle-ci vaut 1 (on était actif à l'entrée du **where**, mais le test était négatif), le compteur passe alors à 0 (on redevient actif). Si la valeur est 0 (on est actif), alors le compteur passe à 1 (on devient inactif pour la durée du corps du **elsewhere**).

- à la sortie du **where**. Le compteur est systématiquement décrémenté lorsqu'on est inactif.

Maintenant que le constructeur **where** est défini, nous allons présenter les autres opérateurs du contrôle de flot.

4.2.2 Les boucles

Une boucle est basée sur deux événement : un saut en arrière et un test. Une boucle vectorielle aura sa fin conditionnée sur un vecteur de booléens. Pour simuler le comportement attendu d'une boucle vectorielle la boucle est répétée jusqu'à ce que tous les éléments du vecteur de test soient faux. Le corps de la boucle est alors effecuté à l'intérieur d'un **where** conditionné par ce vecteur de test.

Le comportement de **break** et **continue** est parfaitement simulé par des manipulations sur le compteur. De la même manière, **return**, pour une fonction renvoyant un vecteur, est simulé par une manipulation sur le compteur.

4.2.3 Le constructeur `switch`

Il est de la même manière simulé à l'aide de manipulation sur le compteur d'activité. On peut récapituler le contrôle de flot dans le tableau 1.

4.3 Communications

Il existe 7 types de communications :

- émissions scalaires,
- réceptions scalaires,
- concentrations scalaires associatives,
- émissions vectorielles,
- réceptions vectorielles,

constructeur scalaire	*constructeur vectoriel*
`if`	`where`
`else`	`elsewhere`
`while`	`whilesomewhere`
`do`	`dowhere`
`for`	`forwhere`
`break`	`break`
`continue`	`continue`
`switch`	`switchwhere`
`case`	`case`
`default`	`default`
`return`	`return`

Tableau 1: le contrôle de flot de RVPC.

- communications sur grille et *scans*[4].
- la primitive de communication `transmit`.

La primitive de communication `transmit` permet de programmer finement le comportement du processeur lors de l'arrivée d'un message. Les communications sur grille et les scans apparaissent sous la forme d'une bibliothèque de fonctions. Elles ne font pas partie à priori du langage, mais elles sont néanmoins nécessaires à l'exploitation de la topologie du réseau.

Les autres communications utilisent l'opérateur `<-`, qui explicite la communication d'une manière générale. L'opérateur `[. .]` permet de préciser une adresse, id est le réarrangement d'un vecteur suivant un vecteur d'indice : `a[.b.]` spécifie le vecteur `a` d'une collection quelconque réarrangée suivant le vecteur d'indices `b` sur une autre collection. Le résultat est un vecteur dont les éléments ont le même type que ceux de `a` et sur la collection de `b`

4.4 les communications scalaires

Elles interviennent entre le processeur scalaire et le tableau de processeurs. La figure 1 présente les différents types communications scalaires.

4.5 les communications vectorielles

- Soit c'est l'émetteur qui a l'initiative de la communication (l'adresse a la même collection), il s'agit alors du *send* de la Connection Machine :

[4]Ce sont des opérations de balayages suivant un opérateur associatif. La réalisation utilise une structure d'arbre.

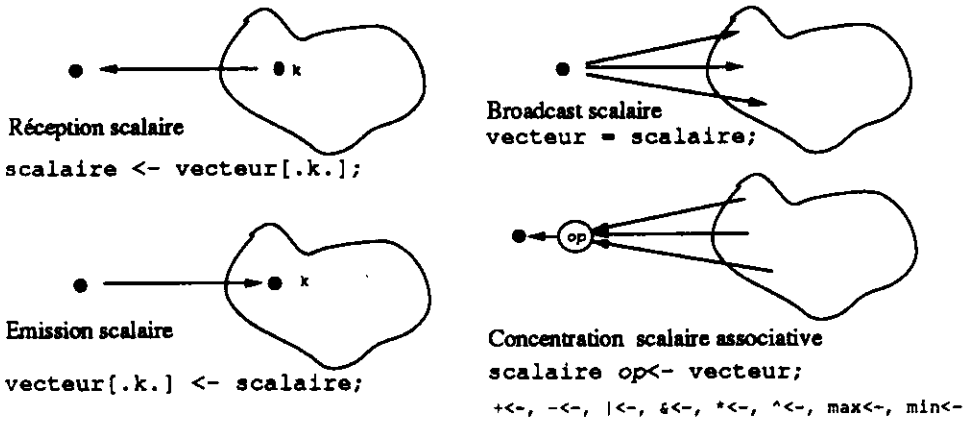

Figure 1: les communications scalaires.

```
destination [. adresse .] op<- source
```

Comme plusieurs données peuvent être reçues par le même processeur, on peut préciser un opérateur associatif (**+**, *****, **&**, **|**, **min** et **max**).

- Soit c'est le récepteur qui a l'initiative de la communication (l'adresse a la même collection), il s'agit alors du *get* de la Connection Machine :

```
destination <- source [. adresse .]
```

Dans tous les cas, l'activité est celle du vecteur d'adresse. On constate qu'on a bien, de part et d'autre de la flèche, des vecteurs de collections identiques.

4.6 Gestion des processeurs virtuels et collection processeur

Comme la taille des vecteurs n'est pas connue à priori, il y a plus d'un processeur virtuel par processeur réel pour toute collection. Chaque variable vectorielle prend la forme sur chacun des processeurs d'un tableau dont la taille est le nombre de processeurs virtuels par processeur réel (le *vp_ratio* de la Connection Machine). Chaque instruction sera répété pour chaque jeux de données. La gestion des processeurs virtuels est réalisée à l'aide d'une boucle *for* énumérant tous les processeurs virtuels. Les instructions pour chaque processeurs virtuels sont exécutées si le compteur d'activité est nul.

Dans certains cas, on peut désirer avoir une collection qui ne subisse pas la gestion des processeurs virtuels : cette collection particulière porte le nom de **processor**. Ainsi chaque variable sur cette collection existe une et une seule fois par processeur physique. Dans ce cas particulier, un vecteur peut être totalement en registre.

Pour limiter le coût de gestion des processeurs virtuels, le générateur de code cherche à factoriser cette gestion. On est amené à distinguer 2 types d'instructions :

- les instructions à effet de bord scalaire : elles modifient des scalaires (affectations scalaires, incrémentations scalaires et les appels de fonctions, qui peuvent avoir des effets de bords et pour lesquelles on désire avoir un appel atomique même si elles traitent des vecteurs).

- des instructions purement vectorielles sur la même collection.

Le générateur de code peut agglomérer des instructions vectorielles sur la même collection. De proche en proche, des sections de codes de tailles importantes peuvent être agglomérées dans la même boucle de gestion de processeurs virtuels. D'une part, cela réduit son coût, d'autre part toute variable vectorielle allouée dans un bloc totalement aggloméré peut être simplifiée. En effet, sur chaque processeur, on doit allouer un tableau pour chaque variable. Comme ce tableau n'est utilisé qu'à l'intérieur de la gestion des processeurs virtuels, chaque élément du tableau ne sera utilisé qu'à une seule itération de la boucle. Il n'est alors plus nécessaire de différencier les éléments du tableau (c'est une variable temporaire) : le compilateur déclasse la variable et lui attribue la collection **processor**. Elle peut alors résider en registre.

Ce point délicat est très important, car avec un minimum de précautions, on peut programmer des blocs de calculs critiques dont tous les intervenants résident en registre : le coût de la gestion des processeurs virtuels devient marginal.

Ce point a une retombée formidable sur notre architecture : une opération est en général effectuée en registre.

Dans le cas contraire si toutes les opérations nécessitent une indirection sur ses 3 opérandes, cela signifie qu'il faut compter 3 accès à la mémoire par opération : la puissance de la machine est alors uniquement déterminée par la bande passante de la mémoire par la règle très simple : le débit mémoire en mots de 32 bits divisé par 3. Cela signifie que, rapporté au nombre de connexions possibles sur une carte dans le cas d'utilisation de boîtiers mémoires existants, on pourra obtenir au mieux 300 MIPS par carte VME triple Europe [Dou89]. La seule méthode pour éviter ce goulot d'étranglement consiste à intégrer sur le même substrat à la fois le processeur et sa mémoire. Cette solution est certes ambitieuse et innovante (c'est pourquoi nous l'avons envisagée), mais nécessite des développements gigantesques :

- développement d'un processeur performant en MIPS et en MFlops,
- développement d'un compilateur pour ce processeur,
- maîtrise d'une technologie de mémoire dynamique.

De plus nous serions restés limités par la technologie : la mémoire par processeur serait d'au plus 2 Mbit et ce en utilisant des technologies très récentes (4 Mbit).

Heureusement nous pouvons utiliser des registres, ce qui nous permet de profiter de l'avancée des RISCs. En trouvant un processeur RISC d'architecture Harvard (pour sa séparation code (global)–données (locales)), possédant une unité flottante intégrée, on peut profiter pleinement des derniers développements de la technologie RISC tant sur le plan matériel, que logiciel.

5 Architecture finale

5.1 Choix de la technologie

Nous avons choisi de ne pas intégrer la mémoire au processeur et d'utiliser un processeur du commerce. Ce choix est crucial quant à l'aboutissement du projet au sein d'une petite équipe telle que la nôtre.

Afin de dépasser le débit processeur–mémoire de 3,6 Go/s [Dou89], nous avons dû diviser la machine en plusieurs cartes contenant des modules. En fait, l'aspect modulaire nous plait bien; nous avons gardé l'association processeur–mémoire sous la forme de modules enfichables sur une carte-mère, un peu à la manière des *TRAM* d'INMOS [Inm89], mais en plus rapide.

La machine se présente alors sous la forme d'un rack VME contenant un Sun 3/110 (3 cartes), la carte de séquencement de la machine et 4 cartes "épaisses" contenant chacune 64 modules montés verticalement. La machine tient donc largement sous un bureau standard.

Elle offre une puissamce de calcul de l'ordre de 5 GIPS et 2,5 GFlops tandis que la consommation électrique reste de l'ordre du kW. Un refroidissement à air suffit donc largement et une climatisation est superflue.

5.2 Choix du processeur

Nous avons cherché un processeur disponible sur le marché qui, tout en étant rapide, puisse être facilement utilisé en mode SIMD. Il va de soi qu'en choisissant un processeur commercial, on bénéficie de tout son support logiciel existant, ce qui est à notre avis la clé de voûte du projet.

Nous avons retenu les processeurs CMOS pour leur bon rapport efficacité/dissipation et parmis ceux-ci nous avons écarté les ALUs "en tranche" et les DSP, principalement car ils ne disposent pas de très bon compilateur à cause de leur architecture très "ouverte".

On peut citer:

- le SPARC : Sun, Cypress [Cyp89], Fujitsu, etc.
- le R3000 : Mips [Mip89], $\int dt$ [Idt89], Nec [Nec88], Siemens, etc.
- le MC88100 : Motorola [Mot88] et SGS-Thomson,
- l'i860 : Intel [Int89a,Int89b].

Seuls les 2 derniers microprocesseurs ont une ALU flottante intégrée, ce qui est indispensable pour une bonne intégration. Mais si l'i860 est plus rapide, il possède un cache intégré et n'a pas une architecture Harvard contrairement au 88100.

L'emploi de caches conduit à la désynchronisation fatale de la machine, puisqu'on est en SIMD.

Enfin l'architecture Harvard est très pratique car elle permet une mise en commun de tous les bus d'instructions des processeurs facilement, sans un acrobatique multiplexage de bus, coûteux en temps et en circuits sur chaque module.

C'est pourquoi notre choix s'est porté sur le 88100 qui nous semble actuellement un bon compromis entre performance et adaptabilité à notre projet.

5.3 Le séquencement des instructions

Il s'agit d'alimenter les processeurs du réseau avec un débit d'instructions de 100 Mo/s[5].

Afin de résoudre les problèmes de débit du bus avec l'hôte, une alternative s'offre à nous :

- avoir un séquenceur qui se contente de dérouler le microcode de bas niveau correspondant aux instructions de haut niveau que l'hôte lui envoie, comme sur la Connection Machine,
- avoir un séquenceur qui contient tout le programme, l'hôte ne servant qu'au système d'exploitation.

Le problème lié au premier point est que ce sera la puissance de l'hôte qui conditionnera la puissance de la machine, problème sensible sur la Connection Machine en *Lisp.

Pour ce qui est du $2^{ème}$ point, il faut que le séquenceur soit capable d'exécuter toutes les instructions de haut niveau de la partie scalaire du programme. Par contre on n'a plus de goulot d'étranglement sur le bus de l'hôte.

Heureusement pour nous, l'arrivée de microprocesseurs rapides sur le marché peut parfois rendre caduque l'utilisation de séquenceurs "en tranches", plus performants mais moins pratiques au niveau de la programmation de haut niveau.

C'est pourquoi le second choix nous semble préférable car clairement plus simple. Afin de conserver une structure orthogonale au niveau de la programmation, nous avons pensé utiliser le même processeur, pour le séquencement, que les processeurs du réseau.

5.4 Les communications

5.4.1 Le réseau d'interconnexion

Le problème du réseau est peut-être même plus important que celui des processeurs, d'après notre expérience sur la Connection Machine. En effet, sur ce genre de machine à mémoire locale, on passe généralement plus de temps à communiquer qu'à faire du calcul. Cela devient d'autant plus sensible qu'on utilise des processeurs rapides et donc que le rapport $\frac{\text{temps de communication}}{\text{temps de calcul}}$ augmente.

On en déduit que pour augmenter les performances de la machine, il faut:

- avoir des communications rapides, c'est à dire une vitesse de transmission élevée, un réseau de faible diamètre[6] et un fort débit,

[5] Les instructions sont codées sur 40 bit et sont émises à 20 MHz.

[6] On entend par là la distance maximale entre 2 processeurs en termes de liaisons.

- faire les communications en parallèle avec le calcul, ie avoir un routeur indépendant et rapide.

Mais cela ne sert à rien de prendre un débit supérieur à celui que peut traiter un processeur. Avec un débit mémoire locale de 80 Mo/s, un débit soutenu de 20 Mo/s de communication par processeur semble être un maximum. Cela correspond à 8 hypercanaux, qui sont des liaisons série à 20 Mbit/s.

Après le choix du débit reste le choix de la topologie. Les deux principales topologies candidates sont l'hypercube et le réseau de De Bruijn. On trouvera une vue d'ensemble de la problématique dans [BP89].

Bien que le réseau de De Bruijn soit très tentant, en ce qui nous concerne, nous préférons l'hypercube pour des raisons technologiques. En effet, comme un hypercube de dimension $n+1$ peut être engendré récursivement par la somme de 2 hypercubes de dimension n, une partition sur plusieurs cartes sera plus simple et nécessitera moins de fils entre les cartes que par exemple un réseau de De Bruijn et sera donc plus facilement câblé, pour un même degré[7]. Mais ce problème sera peut-être bientôt caduque lors d'utilisation de multiplexeurs–démultiplexeurs à haut débit qui simplifieront le câblage, [LEH*89].

En outre, le fait que les liaisons ne soient pas bidirectionnelles dans le réseau de De Bruijn laissent prévoir quelques surprises en cas d'engorgement du réseau. C'est un problème crucial pour une machine SIMD où, lorsqu'on communique, tous les processeurs communiquent !

Enfin, il est relativement utile de pouvoir émuler des routages plus simples sur un hypercube, comme des grilles, ce qui ne peut se faire que par le routeur général dans le cas d'un réseau de De Bruijn. Alors dans ce cas, on perd l'intérêt d'utiliser un mode de communication simple, eg de type *NEWS*[8].

En conclusion, sur le prototype à 256 processeurs sur 4 cartes, on opte pour un réseau en hypercube de dimension 8, ce qui permet d'atteindre un débit crête de 5 Go/s avec une partition simple sur 4 cartes, tandis que si on veut faire une machine réduite monocarte 64 processeurs, il semble plus intéressant d'avoir un réseau de De Bruijn.

5.4.2 Les moyens de communication des processeurs

Ils peuvent communiquer par 3 moyens:

- les hypercanaux, qui constituent le réseau d'interconnexion,
- le `Global Or` (GO), signal commun à tous les processeurs, qui sert au séquenceur à récupérer une condition globale à tous les processeurs ou une donnée sur un seul processeur,
- un hypercanal entre le séquenceur et chaque processeur permettant d'envoyer ou de recevoir des données du réseau de processeurs.

[7] Ou encore le nombre de liaisons connectée à chaque processeurs. C'est donc le paramètre qui fixe le débit de communication du réseau.

[8] Type de communication par voisinage *ONSE* (Ouest, Nord, Sud, Est) sur la Connection Machine.

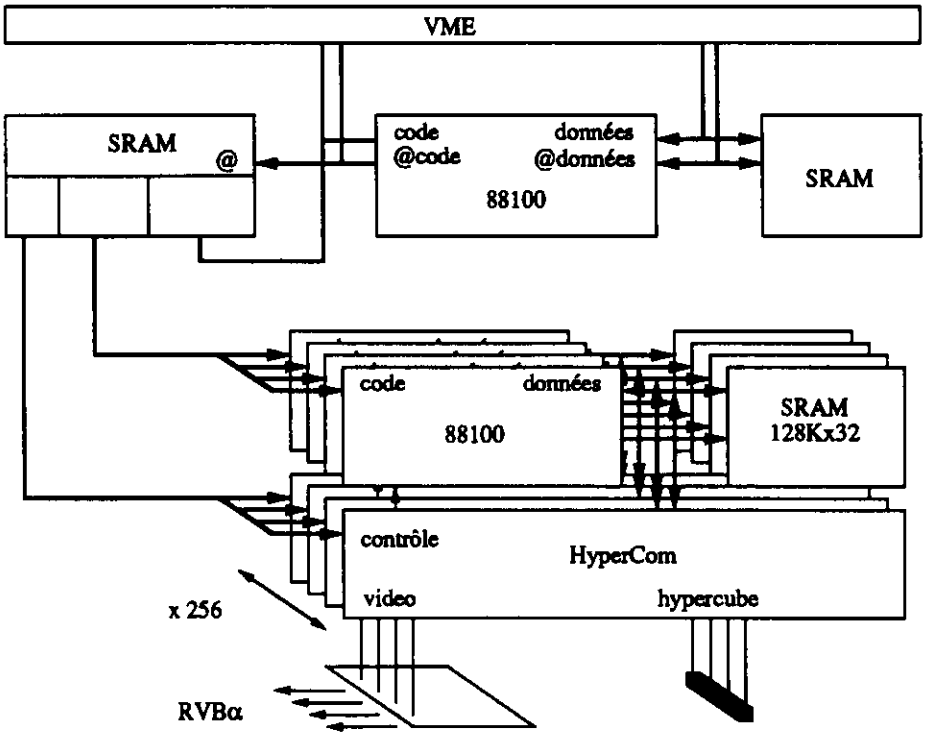

Figure 2: synoptique de RVP.

5.4.3 Les entrées–sorties

D'autres hypercanaux sont réservés à la vidéo et à des entrées–sorties rapides tels que des disques durs.

Un hypercanal par processeur représente un débit d'entrée–sorties, pour une machine à 256 processeurs, de 640 Mo/s, soit par exemple un écran de 1024 × 1280 en 64 bit par pixel.

5.5 Structure du nœud

On peut désormais définir la structure du nœud qui est la brique élémentaire de la machine.

Réaliser chaque nœud sous forme de module permet d'avoir une modularité analogue a l'intégration totale, qui serait optimale, tout en gardant la simplicité de conception puisqu'il n'y a qu'un circuit intégré assez simple à réaliser.

5.5.1 Synoptique de la machine

On peut rassembler chaque nœud dans un module suffisament petit pour pouvoir en positionner verticalement 64 sur une carte VME triple Europe.

La structure simplifiée du nœud au sein la machine est représentée sur la figure 2. Une description plus détaillée peut se trouver dans [Ker89].

5.5.2 HyperCom

Ce circuit intégré de 120 pattes s'occupe principalement des 4 tâches suivantes.

Les communications Il contient pour cela 12 hypercanaux, qui sont des registres à décalage, et un encodeur prioritaire qui prend la responsabilité de l'aiguillage des paquets qui transitent sur le réseau.

La vidéo et les entrées–sorties Il s'agit principalement d'hypercanaux auxquels on a rajouté 128 octets de mémoire tampon en émission et en réception, afin de comprimer dans le temps les transferts de données, effectués sur interruption régulière.

L'activité des processeurs locaux C'est un dispositif capital car il permet d'exécuter les **where** et autres instructions de contrôle de flot de RVC.

HyperCom décide alors si le processeur exécutera les instructions conditionnelles suivantes ou bien au contraire attendra, en fonction du compteur d'activité locale.

La gestion des exceptions Il est capital de gérer correctement les exceptions car des exceptions mineures sur le 88100 doivent être prises en considération[9].

Détaillons un peu ce qui ce passe lors d'une exception sur un processeur:

1. le processeur est mis en attente, la date de l'exception est notée et le numéro de l'exception est capturé par l'HyperCom,
2. l'HyperCom avertit le séquenceur de l'arrivée de l'exception par le signal commun `Global Exception`,
3. le séquenceur exécute la routine d'interruption permettant de gérer les exceptions sur la grille,
4. il corrige les données adéquates dans le(s) processeur(s) en exception,
5. grâce à la date de l'exception et du contenu du pipeline, relance les instructions que n'ont pas exécutées le(s) processeur(s) partis en exception,
6. toute la machine reprend l'exécution du programme principal.

5.6 La génération finale de code

La figure 3 montre le processus de génération du code. Il est à noter l'emploi de logiciels existants en particulier du compilateur C 88100. Ainsi l'effort de développement logiciel se trouve réduit au strict minimum.

Il est intéressant ici de constater qu'on a une machine qui est superlinéaire, c'est à dire que, si elle a par exemple n processeurs, elle peut fonctionner plus de n fois plus vite qu'une machine mono-processeur.

[9] Comme le cas du flottant dénormalisé, du calcul sur des infinis, etc. par rapport à [Iee85].

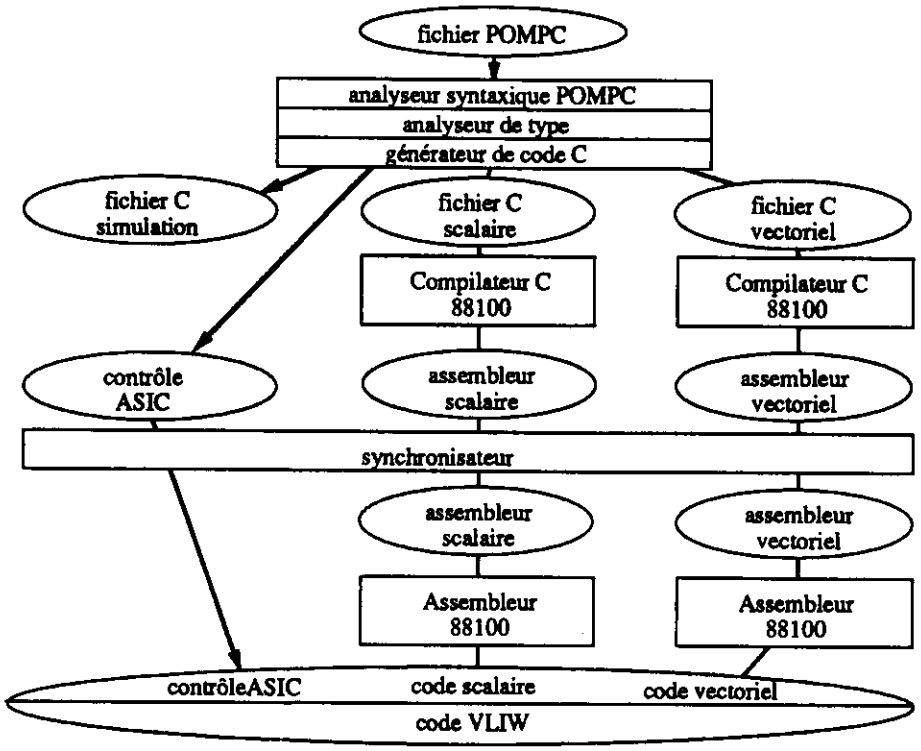

Figure 3: le processus de génération de code.

Cela est dû au fait qu'en factorisant les bus d'instruction, tous les calculs d'adresses et les contrôles de flot sont faits sur le processeur de séquencement, libérant ainsi les processeurs du réseau pour faire des calculs ou des transferts mémoires, par exemple.

La superlinéarité théorique maximale de 2 peut même être obtenue pour des cas, certes simples, tels que les transferts de blocs sur les processeurs, comme **bcopy**. On a dans ce cas une machine SIMD à 257 processeurs qui va aussi vite qu'un machine MIMD à 512 processeurs.

6 Avancement du projet

6.1 Avancement matériel

Un prototype de la carte séquenceur a été réalisé sur une carte à wrapper au format triple Europe et a permis de tester nos premiers programmes.

Une carte avec 2 processeurs est en cours de réalisation, avec une vidéo réduite, compte tenu de la bande passante de 2 hypercannaux.

L'hôte est un Sun 3/110 qui nous permet de communiquer avec l'extérieur et de bénéficier de tous les avantages Unix. En particulier on bénéficie de tout l'environnement de programmation et la plupart des appels systèmes de l'hôte sont utilisables, permettant ainsi de faire croire au programmeur qui utilise PVP qu'il est sur un Sun[10].

[10] Nous n'avons bien entendu pas réalisé l'appel système **fork** puisque c'est une machine SIMD...

6.2 Avancement logiciel

L'analyseur syntaxique de RVP-C fonctionne depuis novembre 1989.

Le simulateur fonctionne depuis décembre 1989 et permet de générer du code C standard, afin de faire tourner les programmes RVP-C sur les machines du LIENS et sur notre prototype monoprocesseur.

On a ainsi quelques petits programmes de démonstration, dont un simulateur électrique, que nous avions écrits pour la Connection Machine.

Un générateur de CPARIS est en cours d'écriture afin de pouvoir programmer la Connection Machine en RVP-C. Cela permettra d'exécuter des programmes sur une autre machine massivement parallèle avant d'avoir une machine à 257 processeurs.

Ensuite le générateur de C pour RVP sera écrit, vers mai 1990.

7 Conclusion

7.1 perspectives industrielles

Le projet POMP est soutenu actuellement par le Ministère de la Recherche et des Technogie en collaboration avec Thomson Digital Image (TDI).

TDI, qui a plus une expérience algorithmique qu'architecturale, est très intéressé par la forte puissance de calcul de POMP. Nous comptons développer avec eux une application importante de synthèse d'image à base de lancer de rayons et de radiosité qui sera un véhicule de test efficace pour toutes nos idées.

Enfin, nous établissons des contacts avec des industriels en vue de fabriquer et commercialiser une machine avec au moins 257 processeurs.

7.2 perspectives de recherche

Nous espérons pouvoir fournir une plate-forme consistante à des équipes de recherche intéressées par l'étude du parallélisme ou des super-ordinateurs, là où à notre sens la Connection Machine a un peu échoué à cause de son prix trop élevé.

Ainsi Luc Bougé utilise actuellement le simulateur POMPC et s'intéresse avec un stagiaire de DEA à la preuve de programmes SIMD.

Au niveau de l'environnement logiciel il reste beaucoup de travail à faire, puisque nous aimerions avoir à court terme un débogueur symbolique mais aussi pouvoir fournir à moyen terme à des utilisateurs peu familiarisés avec le parallélisme une bibliothèque de sous-programmes classiques —accessibles à partir de langages standards— permettant de ne pas avoir à connaître l'architecture interne de POMP dans un grand nombre de cas.

Références

[BP89] J-C. Bermond and C. Peyrat. De Bruij and Kautz Network: a competitor for the hypercube. In INRIA F.André, J.P. Verjus, editor, *Hypercube and*

Distributed Computers, 1989.

[CH80] James-H. Clark and Mark-R. Hannah. A high performance smart image memory. *Lambda*, 1(3), 1980.

[Cla80] James-H. Clark. A VLSI geometry processor for graphics. *Computer*, 59—69, July 1980.

[Cyp89] *CMOS/BICMOS Data Book.* Cypress Semiconductor, février 1989. Chapitre 6: Introduction to RISC (CY7C601).

[DDPP86] César Douady, Daniel Dure, Nicolas Paris, and Alfred Permuy. *FLIP : 2 architectures de multi-fenêtrage graphique.* Technical Report, Cray Research France, 1986. Projet soumis au Concours Cray 1986 ayant obtenu le prix spécial du jury.

[Dou89] César Douady. *Visualisation graphique et architecture parallèle: Conception et réalisation.* Nouvelle Thèse, Paris XI, Mai 1989.

[FEG88] Henry Fuchs, John Poulton John Eyle, and Trey Greer. Coarse-Grain and Fine-Grain Parallelism in the Next Generation Pixel-Planes Graphics System. In *Internationnal Conference and Exhibition on Parallel Processing for Computer Vision and Display*, 12—15 Janvier 1988.

[FP81] Henry Fuchs and John Poulton. Pixel–Plane: a VLSI–oriented design for a raster graphics engine. *VLSI Design*, 2(3), 1981.

[Hil85] W. Daniel Hillis. *The Connection Machine.* the M.I.T. press, 1985.

[Idt89] *High Performance CMOS Data Book Supplement.* 1989.

[Iee85] IEEE 754-1985 Binary Floating Point Arithmetic (ANSI/IEEE). 1985.

[Inm89] *The Transputer, Development and* iq **systems** *Databook.* Inmos-ST, 1989.

[Int89a] *i860 64-Bit Microprocessor Programmer's Reference Manual.* Intel, 1989.

[Int89b] *i860^{TM} Processor Performance, Release 1.0.* Intel Corporation, mars 1989.

[JHH80] James-H.Clark and Mark-R. Hannah. Distributed processing in a high performance smart image memory. *Lambda*, 1(4), 1980.

[Ker89] Ronan Keryell. *POMP2: D'un Petit Ordinateur Massivement Parallèle.* Rapport de Magistère, LIENS — Ecole Normale Supérieure, Octobre 1989.

[LEH*89] LITAIZE, ELKHLIFI, HAMMAMI, LALAM, MZOUGHI, SAINRAT, and SALINIER. Serial multiport memory multiprocessors. In *PARLE '89 Parallel Architectures and Langages Europe*, 1989.

[Mat80] Philippe Matherat. *Le circuit* **EF9365**, *notice de description technique.* Technical Report, Thomson–Efcis, 1980. référence : SP01–F.

[Mip89] *Journée AFUU du 7 Décembre 1989, Présentation RISComputer 6280.* 1989.

[Mot88] *MC88100 RISC processor user's manual.* MOTOROLA, 1988.

[Nec88] *RISC Microprocessors Vr3000 & Vr3010 MIPS RISC Architecture, User's Manual.* 1988.

[Par89] Nicolas Paris. *Développement d'outils de conception de circuits intégrés et application à la réalisation d'une architecture de visualisation.* Nouvelle Thèse, Univesité Paris XI, Mai 1989.

[Par90] Nicolas Paris. *Définition de POMPC (Version 1.5).* Technical Report, LIENS, Février 1990.

[Tmc87] *Connection Machine Model CM–2 Technical Summary.* Technical Report HA87–4, Thinking Machine Corporation, April 1987. Pages 35—41.