Le multipipeline DSPA un multiprocesseur pipeline synchronisé par les données

Yvon Jégou

▶ To cite this version:

Yvon Jégou. Le multipipeline DSPA un multiprocesseur pipeline synchronisé par les données. [Rapport de recherche] RR-0587, INRIA. 1986. inria-00075967

HAL Id: inria-00075967

https://inria.hal.science/inria-00075967

Submitted on 24 May 2006

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

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

INRIA

CENTRE DE RENNES
IRISA

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 587

LE MULTIPIPELINE DSPA
UN MULTIPROCESSEUR PIPELINE SYNCHRONISÉ PAR LES DONNÉES

Yvon JEGOU

Décembre 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

le multipipeline DSPA
un multiprocesseur pipeline synchronisé par les données

the DSPA multipipeline
a data synchronized pipeline multiprocessor

Publication Interne n°321 - Novembre 1986 - 36 pages

Yvon JEGOU

IRISA / INRIA
Campus Universitaire de Beaulieu
35042 RENNES CEDEX

Résumé

Les algorithmes numériques présentent généralement trois types de parallélisme. Le parallélisme de type pipeline provient de l'indépendance qui existe inévitablement entre certains ordres élémentaires d'une séquence d'instructions. Le parallélisme de type vectoriel est interne aux instructions vectorielles dans lesquelles les calculs des éléments des vecteurs sont indépendants. Le parallélisme de type multi-tâche est généralement explicite.

Les supercalculateurs existants favorisent l'exploitation de l'un de ces trois types de parallélisme. Leur efficacité dépend surtout de l'adéquation de la structure des algorithmes exécutés à celui du calculateur.

L'architecture multipipeline que nous présentons dans ce document exploite simultanément ces trois types de parallélisme. Le niveau de parallélisme extrait dans chacune des trois directions -pipeline, vectoriel et multi-tâche- peut être modulé pour adapter la structure du multipipeline à celui de l'algorithme exécuté.

Le faible niveau de complexité des traitements à réaliser pendant le processus de compilation des programmes laisse espérer une *redoutable* efficacité de ce multipipeline sur une famille très large d'algorithmes numériques dont le parallélisme ne pouvait jusqu'à présent être exploité que par des architectures beaucoup plus complexes de type data-flow.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

Abstract

Three different kinds of parallelism can be exhibited from numerical analysis algorithms. Pipeline parallelism is extracted from the inevitable independecies of many elementary orders in an instruction sequence. Vector parallelism in inherent to vector instructions which element computations are independent. Multitask parallelism is usually explicit.

The parallelism that is exploited by existing supercomputers is limited to one among these three models. The efficiency of a computation depends mainly on the adequacy of the computer structure to the algorithm executed.

The multipipeline we present in this document exploites these three types of parallelism simultaneously. Moreover, the level of parallelism that can be extracted from each direction -pipeline, vector, multitask- can be modulated in order to match the structure of the multipipeline to the structure of each algorithm.

The compilation process for a multipipeline remains quite simple; a great efficiency of this architecture is expected on a large family of algorithms which parallelism can be exploited today only by more complex data-flow architectures.

PAPIER RECUPERE ET RECYCLE

Présentation

Un processeur DSPA [Jégou 86a] permet d'exécuter rapidement du code séquentiel même lorsque le comportement de certaines ressources de l'architecture n'est prévisible, ni à la compilation, ni au décodage des instructions, ce qui est fréquemment le cas sur les architectures multiprocesseurs asynchrones. Nous étudions dans une première partie certaines particularités de l'utilisation du DSPA comme processeur élémentaire d'une architecture multiprocesseur de type MIMD à mémoire commune. Mais la programmation de telles architectures entraîne l'exécution d'instructions de synchronisation très coûteuses. Dans de nombreux cas, la gestion d'un parallélisme fin (type SIMD) permet une plus plus grande efficacité des programmes. Ce type de parallélisme est étudié en V.2. En V.3 le traitement des aléas d'accès à la mémoire est généralisé, permettant l'exécution en parallèle d'itérations de certaines boucles séquentielles. Enfin, nous définissons en V.5 le multipipeline DSPA qui intègre toutes ces possibilités et qui permet de moduler le parallélisme offert par les niveaux pipeline et multiprocesseur en fonction des algorithmes.

1 Le DSPA dans une architecture MIMD

Dans les architectures MIMD faiblement couplées, les communications de données entre les processeurs se font par échange de messages plus ou moins volumineux. Le fonctionnement interne des processeurs a finalement peu d'importance dans la définition de telles architectures. Nous nous intéressons ici aux architectures MIMD fortement couplées à mémoire commune. Une telle architecture est composée d'une mémoire principale et de plusieurs processeurs élémentaires (figure 1). Pour permettre des débits suffisants d'échange de données, cette mémoire principale est généralement découpée en plusieurs mémoires qui peuvent elles-mêmes être découpées en plusieurs bancs. Les échanges entre les processeurs et cette mémoire transitent par un réseau d'interconnexion. Les processeurs sont indépendants. Les synchronisations sont explicites. Toutes les mises en oeuvres existantes des instructions de synchronisation peuvent être adaptées aux processeurs DSPA. Par exemple l'instruction lire/écrire que nous avons vu en [Jégou 86a] permet de synchroniser les processeurs à

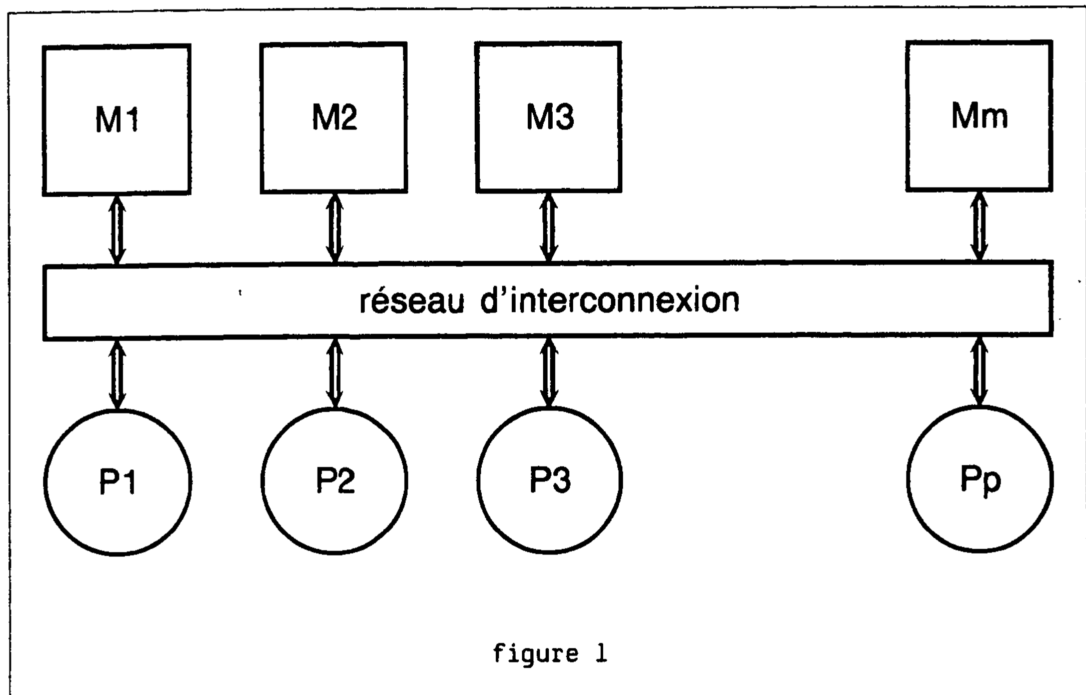

figure 1

différents niveaux. En réémettant la donnée lue par une instruction mémoire lire/écrire vers la mémoire comme donnée, après modification éventuelle (incrémentation, par exemple), le partage de compteurs par les processeurs est possible. Ce partage ne synchronise nullement les processeurs, leur séquencement d'instruction n'étant pas concerné par ces opérations, ni les unités fonctionnelles des processeurs. La cohérence de la variable partagée est prise en charge par le mécanisme de gestion du doublement des écritures mémoire par les lectures. Lorsque la donnée lue est consommée par l'unité fonctionnelle mémoire du processeur sur l'entrée logique d'adresses, une synchronisation de cette unité fonctionnelle est réalisée : elle ne peut émettre de nouvelles requêtes tant que la donnée n'a pas été reçue. Lorsque la donnée lue est consommée par l'unité fonctionnelle de séquencement du processeur, le séquencement est interrompu tant que la donnée n'a pas été reçue, ce qui permet la programmation de véritables test/and/set.

La mise en oeuvre des mécanismes de doublement des écritures par les lectures doit être particulièrement soignée pour éviter certains désagéments. Ces mécanismes peuvent être implantés de plusieurs manière :

o localement à chaque processeur. On peut dans ce cas, soit prévoir un mécanisme global soit un mécanisme spécialisé pour chaque mémoire. Dans ce cas, les aléas interprocesseurs ne sont pas traités, et un système de synchronisation direct des processeurs doit être mis en place, ce qui n'est pas forcément gênant car les processeurs sont totalement indépendants. Cependant, à chaque opération de synchronisation explicite entre les processeurs, il faut s'assurer que toutes les données à écrire correspondant aux instructions d'écriture séquencées avant l'instruction de synchronisation ont été effectivement écrites. Cette contrainte, non seulement entraîne un vidage de tous les pipelines préalablement à la synchronisation des processeurs, mais entraîne également l'introduction de mécanismes globaux de vérification de l'état des pipelines.

o globalement à la mémoire. Le mécanisme de doublement des écritures par les lectures peut être implanté, soit au niveau de chaque mémoire, soit au niveau de chaque banc lorsque ces mémoires sont découpées en bancs physiques. Dans ce cas, chaque mécanisme de traitement des doublements des écritures par les lectures reçoit des requêtes en provenance de tous les processeurs. L'exécution d'un ordre de lecture émis par un processeur peut être suspendue pour cause de dépendance avec un ordre d'écriture émis par un autre processeur, ce qui évite le vidage des pipelines à chaque opération de synchronisation (partage de compteurs, par exemple). Mais, des requêtes indépendantes sont alors artificiellement ordonnées au niveau de chaque banc. L'ordre d'arrivée des requêtes dépend de leur ordre d'émission par les processeurs et du fonctionnement du réseau d'interconnexion. La contrainte minimale que nous imposons à ce réseau est que les requêtes émises par un même processeur vers une même mémoire sont délivrées à cette mémoire dans l'ordre d'émission, contrainte minimale permettant de donner une signification au doublement des écritures par les lectures. Mais cette contrainte n'interdit pas que la situation suivante puisse se produire :

Supposons que le processeur 1 exécute l'instruction a := b ; et que le processeur 2 exécute l'instruction b := a ; la variable a étant située dans

la mémoire M_a et la variable b dans la mémoire M_b. Le processeur 1 émet un ordre de lecture de b suivi d'un ordre d'écriture de a. Le processeur 2 émet un ordre de lecture de a suivi d'un ordre d'écriture de b. Les processeurs étant asynchrones, supposons que la mémoire M_a reçoive la requête d'écriture dans a avant la requête de lecture de a. La requête de lecture est alors suspendue tant que la donnée correspondant à la première instruction n'a pas été reçue. Mais la contrainte minimale de fonctionnement du réseau que nous avons imposé n'interdit pas que la mémoire M_b reçoive également sa requête d'écriture avant sa requête de lecture. Il suffit pour que ce phénomène se produise que le transport d'une requête du processeur 1 vers la mémoire M_a soit plus rapide que le transport d'une requête de ce même processeur vers la mémoire M_b, et inversement pour le processeur 2. Cette situation est bloquante et doit être évitée. Comme la séparation des requêtes des processeurs est gênante dans le mécanisme de détection des aléas, il faut imposer des contraintes plus fortes sur le fonctionnement du réseau (temps de traversée constant, par exemple).

Ce problème nous semble la seule difficulté d'intégration d'un processeur DSPA dans une architecture MIMD, ce processeur offrant par ailleurs une très bonne souplesse pour une telle utilisation.

2 Le DSPA dans une architecture vectorielle multiprocesseur

Nous considérons ici les architectures multiprocesseur qui permettent de répartir l'exécution des instructions vectorielles d'un langage comme Hellena sur les processeurs élémentaires. De nombreuses architectures SIMD entrent dans cette catégorie. Elles peuvent être basées sur un séquencement unique d'instructions qui sont diffusées aux processeurs élémentaires. Ce processeur de commande peut être un processeur DSPA qui exécute une partie des instructions séquencées et qui diffuse certaines instructions directement vers les FIFOs d'instructions des unités fonctionnelles des processeurs élémentaires. Ces architectures sont en réalité monoprocesseur/multi unité fonctionnelle. Un tel exemple de structuration est traité en [Jégou ?] pour le SDSPA. Pour permettre les deux modes de fonctionnement vectoriel et MIMD, chaque processeur élémentaire doit être doté d'un séquenceur d'instructions. Dans la suite,

nous considérons donc que, en mode vectoriel, tous les processeurs élémentaires exécutent la même séquence de code, ce qui permet d'obtenir le même comportement que dans la version multi-unité fontionnelle à condition de prévoir quelques instructions de diffusion. Les processeurs DSPA permettent un séquencement rapide des programmes et un bon recouvrement dans l'exécution des instructions, ce qui limite la nécessité d'introduire un processeur spécialisé dans l'exécution du code scalaire.

Dans un multiprocesseur vectoriel, les processeurs élémentaires se partagent la mémoire principale. Dans une première approche, nous pouvons considérer que cette mémoire est la seule ressource partagée. Soit la suite d'instructions Hellena de la figure 2. L'exécution de chacune de ces

```
M.ligne(i) := M.ligne(i)*v ;
M.colonne(j) := M.colonne(j)+t ;
```

figure 2

instructions est répartie sur les processeurs élémentaires. Mais, ces processeurs ne sont pas indépendants. L'exécution de la deuxième instruction doit prendre en compte certains résultats de la précédente : la détection des aléas dûs au doublement des écritures par les lecture en scalaire doit être généralisée à la détection des aléas dûs à la répartition des instructions sur les processeurs. Cette détection est résolue simplement sur les architectures SIMD classiques : tous les processeurs exécutent la même instruction à un instant donné, et ne sont généralement pas pipeline. Dans notre architecture, nous cherchons à accélérer l'exécution du mode vectoriel en utilisant toutes les possibilités de pipeline de notre processeur scalaire. L'exécution d'un accès vectoriel à la mémoire d'un multi-DSPA peut se faire de deux manières : soit en définissant une instruction de synchronisation explicite des unités fonctionnelles mémoires de tous les processeurs, soit en définissant des requêtes synchrones à la mémoire.

● **instruction de synchronisation.** L'exécution d'une instruction de synchronisation globale par une unité fonctionnelle mémoire d'un processeur provoque la suspension du séquencement de cette unité fonctionnelle tant que toutes les unités fonctionnelles mémoire de tous les processeurs n'ont pas exécuté une telle instruction. Il suffit de produire cette instruction

après le code résultant de la traduction des instructions vectorielles pour garantir que les ordres de lecture d'une instruction sont émis après tous les ordres d'écriture de l'instruction vectorielle précédente. Notons que cette instruction n'est pas liée au fonctionnement vectoriel du calculateur et peut être utilisée partout où une synchronisation des accès mémoire est nécessaire. Cette instruction provoque un rendez-vous de l'ensemble des unités fonctionnelles mémoire. Il est également possible d'imaginer des instructions de synchronisation locales qui provoquent des rendez-vous sur les requêtes de deux processeurs. Les instructions de la figure 2 peuvent être codées comme dans la figure 3 en utilisation un ordre de synchronisation globale sur la mémoire. Les dépendances entre les deux

```
| R : lire,n       → Sq | Sq:InitBV,R →     |
| M :Vlpic,Mli    →*fg | R : lire,v  →*fd | *f : VS*,M,R  → dM |
| M :Vépic,MLi,*f→     | Sq: ItèreBV →     |
| M : Synchro     →    | R : lire,m  → Sq | Sq : InitBV,R →     |
| M :Vlpic,Mcj    →+fg | R : lire,t  →+fd | +f : VS+,M,R  → dM |
| M :Vépic,Mcj,+f→     | Sq: ItèreBV →     |
```

figure 3

instructions Hellena sont respectées dans l'exécution de cette séquence. Cependant, lorsque les vecteurs des adresses des écritures ne sont pas injectifs sur l'espace d'adressage de la mémoire, cette instruction de synchronisation ne suffit pas à imposer un ordre sur le rangement des éléments des vecteurs. Sémantiquement, la signification de telles opérations n'est pas définie en Hellena, ce qui suffit à montrer que cette instruction permet de garantir l'exécution correcte des instructions valides du langage.

● **requêtes synchrones.** Les requêtes synchrones des unités fonctionnelles mémoire provoquent une synchronisation de ces unités fonctionnelles préalablement à l'émission des ordre vers la mémoire. Ces requêtes synchrones permettent donc d'ordonner toutes les requêtes issues de la traduction des instructions vectorielles et parallèles et donc de donner une signification aux cas où deux ordres d'écriture parallèle s'adressent à un même mot mémoire. Lorsque l'unité fonctionnelle mémoire d'un processeur décode une instruction d'accès synchrone (qui peut être vectorielle) à la mémoire, son séquencement est suspendu jusqu'à ce que toutes les unités

fonctionnelles mémoire du multiprocesseur aient décodé une telle instruction. Tous les processeurs exécutent la même séquence d'instructions, ce qui garantit cette synchronisation du séquencement des unités mémoire. Les requêtes mémoires sont alors émises à travers le réseau d'interconnexion, avec arbitrage éventuel des conflits sur ce réseau.

Ces synchronisations permettent de s'assurer que tous les ordres d'écriture d'une instruction vectorielle sont reçus par les mémoires avant les ordres de lecture de l'instruction vectorielle suivante. Le mécanisme de doublement des écritures par les lectures avec contrôle des aléas d'accès fonctionne automatiquement sur les vecteurs répartis sur les processeurs à condition que ce mécanisme soit implanté au niveau des mémoires. Notons que nous avons uniquement synchronisé l'émission des requêtes (composées d'une instruction et d'une adresse), les échanges des données restent asynchrones et le séquencement des processeurs n'est pas interrompu. Le blocage de l'ordre de lecture d'un processeur ne bloque pas nécessairement les échanges des données des autres processeurs. Comme nous l'avons vu dans [Jégou 86a], les processeurs DSPA peuvent exécuter des instructions vectorielles. Ces instructions vectorielles peuvent également exécutées en mode multiprocesseur, la synchronisation des unités fonctionnelles mémoire étant alors réalisée sur chaque élément de vecteur. Par exemple, soit à exécuter l'instruction Hellena de la figure 4 sur un multi-DSPA à 16 processeurs qui exécutent des instructions vectorielles de longueur maximum 8. Cette instruction est alors interprétée par

```
a := b + c*d ;
```

figure 4

l'exécution d'instructions vectorielles sur des vecteurs de longueur maximum 128. Le traitement des aléas d'accès à la mémoire n'est généralisé qu'aux vecteurs de longueur inférieure ou égale à cette valeur (voir les problèmes liés au découpage vertical des instruction de Hellena dans [Jégou 86c]). Ce traitement suffit cependant pour assurer les dépendances entre deux instructions vectorielles successives de Hellena, quelque soit leur longueur. Pour respecter les dépendances de type *écriture après lecture* internes à une instruction vectorielle, il suffit d'intégrer dans chaque processeur une unité fonctionnelle Fi qui retarde le traitement des

ordres d'écriture mémoire tant que tous les ordres de lecture de l'instruction vectorielle n'ont pas été exécutés. Cette unité fonctionnelle Fi est composée d'une très grosse mémoire très intégrée gérée uniquement en FIFO. Elle possède un ordre *de* de mémorisation d'une donnée en provenance d'une sortie d'unité fonctionnelle et un ordre *vers* qui extrait une donnée de la FIFO et l'émet vers une entrée logique d'unité fonctionnelle. Lorsqu'une dépendance de type *écriture après lecture* peut être provoquée par le tronçonnage vertical d'une instruction vectorielle, cette instruction vectorielle est automatiquement transformée à la compilation en une instruction vectorielle qui produit ses résultat dans cette unité Fi suivie d'une instruction vectorielle de lecture sur Fi et d'écriture en mémoire. Par exemple, soit l'instruction Hellena de la figure 5. Pour lever les dépendances de type *écriture après lecture* dûs au

```
A.ligne(i) := A.ligne(j)*A.colonne(k) ;
```

figure 5

tronçonnage vertical, cette instruction peut être transformée automatiquement comme dans la figure 6. Le code de la figure 7 est alors

```
FIFO := A.ligne(j)*A.colonne(k) ;
A.ligne(i) := FIFO ;
```

figure 6

produit sur chaque processeur. Cette unité fonctionnelle permet de

```
| R : lire,n        → Sq  | Sq : InitBV,R →      |
| M : Vlpic,Alj     → *fg | *f : *,M,M    → Fi   |
| M : Vlpic,Ack     → *fd | Fi : de,*f    →      | Sq : ItèreBV  →  |
| M : Synchro       →     | R  : lire,n   → Sq   | Sq : InitBV,R →  |
| M : Vépic,Ali,Fi  →     | Fi : vers     → dM   | Sq : ItèreBV  →  |
```

figure 7

généraliser les instructions vectorielles et les instructions **pour_tout** avec spécification **parallèlement** à des longueurs quelconque sans faire appel à des techniques de génération de code complexes. L'utilisation de l'unité Fi garantit le respect des aléas *écriture après lecture* sur chaque processeur, l'utilisation de l'instruction de synchronisation explicite

(comme dans l'exemple) ou de requêtes d'écriture synchrones garantit le respect des dépendances inter-processeur. Les longueurs des vecteurs sont toutefois limitées par le volume de cette FIFO. Lorsque cette unité fonctionnelle *Fi* est ebsente sur un processeur DSPA, elle peut être simulée dans la mémoire locale.

Comme nous l'avons fait pour les instructions vectorielles des processeurs scalaires, le traitement des longueurs de vecteurs inférieures à la valeur physique maximum et du masquage des instructions vectorielles doit être généralisé au multiprocesseur. Les éléments de chaque vecteur étant répartis sur les processeurs, pour l'exécution d'une instruction vectorielle de longueur quelconque, tous ces processeurs ne possèdent pas le même nombre d'éléments. Cette différence du nombre de requêtes mémoire ne pose aucun problème sur les instructions non synchronisées sur la mémoire, ni pour les instructions de synchronisation explicite qui sont placées en dehors des boucles vectorielles. Par contre, en cas d'utilisation des requêtes synchrones placées dans les boucles issues de la traduction d'instructions vectorielles, tous les processeurs devraient émettre le même nombre de requêtes vectorielles. La solution la plus simple consiste à considérer que chaque processeur connaît le nombre d'accès vectoriels à émettre (information commune à tous les processeur) ainsi que son nombre d'accès propres. Certains processeurs complètent alors leurs accès réels par des accès fictifs qui ont pour rôle unique de permettre la synchronisation des requêtes. Le nombre d'accès fictifs est de un au maximum lorsque les vecteurs sont répartis sur les processeurs de telle façon que l'élément *i* est traité par le processeur *i modulo p*, *p* étant le nombre de processeur (et ceci indépendemment des longueurs des instructions vectorielles de ces processeurs). Pour accéder à ces vecteurs, chaque processeur exécute l'instruction vectorielle avec un pas d'indice *p*. Les descripteurs linéaires doivent être initialisés avec des adresses de base de la forme A+R*i, A étant l'adresse du premier élément du vecteur en mémoire, R sa raison d'accès et i le numéro du processeur. Cette adresse correspond à l'adresse du premier élément accédé par ce processeur. La raison d'accès aux éléments traités par ce processeur est alors R*p. Ce nombre total d'accès doit être également pris en compte dans les instructions de gestion des boucles vectorielles du séquenceur de chaque processeur. La production d'accès fictifs peut provoquer

l'exécution d'une dernière itération vectorielle avec une longueur effective nulle des vecteurs.

Dans les cas de masquage, une telle répartition est également possible. Mais, dans ce cas, le nombre d'éléments à accéder par chaque processeur dépend de son masque propre, et le nombre de requêtes synchrones fictives ne peut être déterminé statiquement. La solution la plus simple consiste à émettre autant de requêtes que s'il n'y avait pas de masquage, les requêtes correspondant aux éléments masqués sont fictives et ne servent qu'à synchroniser les unités fonctionnelles mémoire. Cette solution est très simple mais ne permet pas de réduire le nombre de cycles de séquencement des unités fonctionnelles mémoire au minimum. Le nombre de cycles de commutation effective des mémoires est cependant ramené au nombre d'éléments non masqués. Une solution plus complexe consiste à déterminer par un mécanisme global le maximum du nombre d'accès demandé par chaque processeur. Le séquencement des unités fonctionnelles mémoire est alors ramené au séquencement des accès réels complété par le séquencement du nombre d'accès fictifs nécessaires pour atteindre ce maximum. Cette compression des accès à la mémoire modifie l'ordre d'émission des requêtes appartenant à une même instruction vectorielle. Ceci n'est pas gênant tant que les vecteurs d'adresses en écriture sont injectifs sur l'espace d'adressage, mais, dans ce cas, l'utilisation d'instructions de synchronisation explicites permet une plus grande efficacité. Par contre, lorsque deux adresses du vecteur des adresses d'une instruction d'écriture mémoire sont égale, il peut être nécessaire de définir l'ordre de prise en compte.

Dans le cas général, l'utilisation d'ordres de synchronisation explicites et d'ordres d'accès MIMD aux données nous semble préférable. En effet, la mise en oeuvre des accès synchrones doit être basé sur des mécanismes relativement complexes pour l'émission des requêtes fictives. L'utilisation de ces instructions permet cependant de réduire le nombre de requêtes traitées par les unités fonctionnelles mémoire, ce qui n'est pas négligeable dans le cas des vecteurs courts.

Des instructions spécifiques au fonctionnement multiprocesseur vectoriel doivent être prévues pour accéder aux scalaires dans les instructions vectorielles et pour exécuter les réductions en parallèle. La

présence de scalaires dans les intructions vectorielles étant détecté à la compilation, il suffit que chaque processeur exécute une instruction de lecture scalaire qui provoque la synchronisation des unités fonctionnelles mémoire, mais qu'une seule requête soit émise vers la mémoire, le résultat étant diffusé vers les processeurs. Pour éviter de resynchroniser les unités fonctionnelles mémoire à chaque itération et pour éviter la gestion d'itérations fictives, ces lectures de scalaires peuvent être extraites des boucles vectorielles, les valeurs lues étant sauvées en mémoire locale des processeurs. Pour les réductions, la sommation par exemple, les vecteurs peuvent être répartis de manière classique sur les processeurs, ce qui permet de calculer p contributions au résultat, ou même nxp contributions si chaque processeur calcule localement n contributions comme nous l'avons vu en [Jégou 86a]. Mais, le calcul du résultat final doit prendre en compte ces p contributions. Pour ce faire, on peut, par exemple, produire une séquence d'instructions vectorielles avec masquage progressif des processeurs, les résultats étant échangés par la mémoire. Une autre technique que nous généraliserons dans ce document consiste à prévoir une unité fonctionnelle Com d'échange de données interprocesseur. Dans le cas qui nous intéresse, ces liaisons peuvent être limitées aux chemins de données nécessaires pour effectuer les réductions, ou aux liaisons d'un hyper-cube, par exemple. Comme il n'y a pas, dans ce cas, partage de la ressource commune mémoire, tous les processeurs n'exécutent pas nécessairement la même séquence de code. Par exemple, supposons que chaque processeur possède une unité d'échange Com dotée d'une entrée logique Com, d'une sortie logique Com qui peut exécuter l'instruction de réception |Com: de,i → dest | permettant de recevoir une donnée émise par le processeur dont le numéro diffère uniquement par le bit i vers l'entrée logique dest et l'instruction |Com: vers,i,source→ | permettant d'émettre la donnée reçue de la sortie de l'unité fonctionnelle source vers le processeur dont le numéro diffère uniquement par le bit i. Sur une architecture composée de 16 processeurs, chaque processeur exécute l'une des séquences de code suivante pour effectuer la sommation finale (la première instruction correspondant au calcul de la sous-somme finale de chaque processeur). Le résultat final est calculé par le processeur 0 et peut être, par exemple, rangé en mémoire par une instruction d'écriture en mode MIMD (figure 8). Cette différence dans les codes des différents processeurs ne pose aucun problème, la seule synchronisation des processeurs étant localisée au niveau des unités fonctionnelles mémoire.

processeur 0

```
| +f : +, +f, +f   → +fg | Com : de,0   → +fd |
| +f : +, +f, Com  → +fg | Com : de,1   → +fd |
| +f : +, +f, Com  → +fg | Com : de,2   → +fd |
| +f : +, +f, Com  → +fg | Com : de,3   → +fd |
| +f : +, +f, Com  → destinataire |
```

processeur 8

```
| +f : +, +f, +f   → +fg | Com : de,0   → +fd |
| +f : +, +f, Com  → +fg | Com : de,1   → +fd |
| +f : +, +f, Com  → +fg | Com : de,2   → +fd |
| +f : +, +f, Com  → Com | Com : vers,3, +f →  |
```

processeurs 4 et 12

```
| +f : +, +f, +f   → +fg | Com : de,0   → +fd |
| +f : +, +f, Com  → +fg | Com : de,1   → +fd |
| +f : +, +f, Com  → Com | Com : vers,2, +f →  |
```

processeurs 2, 6, 10 et 14

```
| +f : +, +f, +f   → +fg | Com : de,0   → +fd |
| +f : +, +f, Com  → Com | Com : vers,1, +f →  |
```

processeurs 1, 3, 5, 7, 9, 11, 13 et 15

```
| +f : +, +f, +f   → Com | Com : vers,0, +f →  |
```

figure 8

Les échanges directs interprocesseurs par les unités fonctionnelles d'échange peuvent se faire à travers des FIFOs, ce qui évite une synchronisation fine de ces unités fonctionnelles.

Dans ce multiprocesseur vectoriel, chaque processeur exécute sa propre séquence d'instructions. Lorsqu'une instruction vectorielle ou parallèle est codée sans utiliser de requêtes synchrones ni de requêtes de lecture de scalaires sur la mémoire, la répartition des éléments des vecteurs ou des familles d'instructions produites par les instructions **pour_tout** n'est pas nécessairement définie statiquement à la compilation. Dans une version du langage Hellena où le programmeur pourrait définir sa propre largeur de parallélisme en SPMD (son nombre de processeurs virtuels), l'allocation statique des processeurs virtuels aux processeurs effectifs n'est pas obligatoire. Il est, par exemple, possible de déclarer une variable initialisée à 0 en mémoire. A l'initialisation de l'exécution SPMD, de l'instruction vectorielle ou de l'instruction **pour_tout** et à chaque fois qu'un processeur a terminé l'exécution d'un bloc SPMD, d'un élément de l'instruction vectorielle ou **pour_tout**, il exécute une instruction lire/écrire sur cette variable et émet vers la mémoire la valeur lue incrémentée de un. De cette manière, l'ensemble des numéros des processeurs virtuels est alloué dynamiquement aux processeurs physiques. Cette allocation dynamique équilibre la charge des processeurs. Les conflits d'accès à cette variable commune peuvent provoquer un léger retard de l'exécution des premiers blocs. Ce retard peut être éliminé en allouant statiquement les p premiers blocs aux p processeurs et en initialisant la variable à cette valeur p.

Par contre, il n'est pas possible, comme nous l'avons fait sur un seul processeur DSPA, d'accéder directement à des opérandes vectoriels dans des instructions scalaires, ni de concaténer des opérandes scalaires pour former des opérandes vectoriels. Il faudrait programmer de tels changements de mode en utilisant les unités d'échange interprocesseur, le code scalaire séquentiel étant alors exécuté par un seul processeur.

Ces deux modes de fonctionnement MIMD et vectoriel permettent de coder efficacement les instructions vectorielles, les instructions parallèles et les instructions SPMD du langage Hellena sans introduire de synchronisation explicite des processeurs, et ce par une technique de compilation peu complexe. La synchronisation fine réalisée soit dans les accès vectoriels synchrones (au moins en écriture), soit par des instructions de synchronisation explicites sur les unités fonctionnelles mémoire n'influe pas sur le séquencement des instructions des processeurs.

De nombreuses instructions **pour_tout** avec spécification **séquentiellement** présentent un parallélisme potentiel important que nous avons exploité sur le monoprocesseur DSPA. Nous avons également cherché à étendre l'exploitation de ce parallélisme au multiprocesseur DSPA.

3 Le mode itération du multiprocesseur DSPA

Une spécification **séquentiellement** d'une instruction **pour_tout** indique que, lorsque deux requêtes à la mémoire provoquent un accès à la même adresse et qu'au moins l'une de ces deux requêtes est une écriture, ces deux requêtes doivent être exécutées par valeur croissante de la valeur de contrôle de l'instruction **pour_tout**. Par exemple, soit l'instruction de multiplication de matrice creuse par un vecteur plein de la figure 9.

```
pour_tout i sequentiellement dans [1,n] :
   a(l(i)) := a(l(i)) + v(i)*b(c(i)) ;
fin_pour_tout ;
```

figure 9

Supposons que l(3) soit égal à l(5). Les accès à a(l(3)) et a(l(5)) doivent se faire dans l'ordre de la figure 10. Dans le

```
lecture de a(l(3))
écriture de a(l(3))
lecture de a(l(5))
écriture de a(l(5))
```

figure 10

monoprocesseur DSPA, il suffisait, soit d'accéder au vecteur a par des instruction scalaires, soit d'utiliser une instruction vectorielle lire/écrire. André Seznec a proposé d'étendre la définition de cette instruction lire/écrire au mode multiprocesseur vectoriel synchrone de la manière suivante : lorsque deux requêtes d'un accès vectoriel synchronisé sont destinées au même banc mémoire, il suffit de donner la priorité à (de traiter d'abord) la requête émise par le processeur de plus petit numéro.

Cet ordonnancement statique est simple à réaliser, soit au niveau de l'arbitrage du réseau d'interconnexion (car ces deux requêtes provoquent nécessairement un conflit de destinataire), soit au niveau de chaque banc mémoire. Mais la production automatique de cette instruction (par un compilateur) ne peut se faire que dans quelques cas particuliers. De plus, cette instruction interdit toute compression des accès à la mémoire lorsqu'il y a masquage.

Nous avons donc défini un mode de fonctionnement *itération* du multiprocesseur. Comme pour les instructions vectorielles, le traitement des instances d'une instruction **pour_tout** avec spécification **séquentiellement** est réparti sur l'ensemble des processeurs. La notion de mode itération est liée au fait que, sur les ressources partagées, les requêtes doivent être ordonnées par indice de contrôle croissant, comme on le fait dans le cas du monoprocesseur DSPA. L'implantation de ce mode de fonctionnement peut se faire de la manière suivante :

→ Chaque banc mémoire possède une FIFO de requêtes par processeur dans laquelle sont déposées les requêtes en mode itération.

→ Lorsqu'un processeur a fini d'émettre les requêtes correspondant à la même valeur de contrôle, il diffuse vers tous les bancs un ordre indiquant la fin du traitement de sa valeur de contrôle courante. Cette ordre est une instruction de son unité fonctionnelle mémoire. Comme toutes les requêtes émises dans ce mode, cet ordre est rangé dans les FIFOs des requêtes en provenance du processeur émetteur vers chaque banc.

→ lorsqu'un processeur a fini les traitements d'une certaine valeur de contrôle, il traite les instructions correspondant à la prochaine valeur qui lui est allouée.

→ Après passage dans ce mode de fonctionnement, chaque banc mémoire traite d'abord les requêtes en provenance du processeur 0, c'est à dire en ne prenant que les requêtes de la FIFO du processeur 0. Lorsque l'ordre de fin d'itération est extrait de la FIFO courante, le banc extrait les requêtes de la FIFO des requêtes en provenance du processeur 1 et ainsi de suite cycliquement sur les processeurs.

L'effet obtenu par ce mode itération est que chaque mémoire traite les requêtes dans le même ordre que celui produit par une exécution séquentielle. Ceci garantit la sémantique de l'instruction. Notons que ce système se contente d'ordonner les requêtes. Le mécanisme de doublement des écritures par les lectures continue de fonctionner, et tous les bancs ne traitent pas nécessairement des requêtes de la même itération à un instant donné. Après un délai de startup plus ou moins long, le débit potentiel de la mémoire peut être atteint.

Cependant, les dépendances possibles des instructions traitées sont généralement limitées à quelques accès à la mémoire. Par exemple, dans le cas de la multiplication matrice creuse par un vecteur plein, les seules dépendances possibles se limitent aux accès au vecteur a (figure 9). Par contre, il est fréquent que le résultat d'une lecture mémoire soit utilisé pour calculer l'adresse d'un accès ultérieur, ce qui retarde l'émission de ce deuxième ordre au plus tôt à la réception de la valeur d'adresse. Ces dépendances internes de séquencement peuvent ralentir fortement l'émission des requêtes. Il est possible, dans ce cas, de traiter les instructions parallèles **pour_tout** avec spécification **séquentiellement** par une première séquence d'accès en mode vectoriel (accès aux données sur lesquelles aucune dépendance n'existe qui peuvent également être émis en mode MIMD pour les lectures) suivi d'un passage en mode pipeline et des requêtes à exécuter dans ce mode. Le corps de l'instruction se termine par un retour en mode vectoriel. Ce découpage s'applique à de nombreux cas de programmation numérique rencontrés.

La fin du mode pipeline doit être annoncée soit par l'un des processeurs, soit par tous les processeurs auquel cas tous les processeurs doivent émettre le même nombre d'ordres de fin d'itération.

L'algorithme de multiplication de matrice creuse par un vecteur plein peut être codé comme dans le figure 11 sur le multiprocesseur DSPA. Chaque processeur exécute cette séquence d'instructions. La boucle externe gère des itérations vectorielles (sur des vecteurs de longueur 128, par exemple). Les vecteurs l, c, v et b sont lus dans ce mode vectoriel. Chaque processeur conserve sa portion du vecteur l dans sa mémoire locale à l'adresse Ad. A l'intérieur de cette boucle, il y a passage en mode séquentiel par l'instruction inititLV définie en [Jégou 86a] : cette

Cet ordonnancement statique est simple à réaliser, soit au niveau de l'arbitrage du réseau d'interconnexion (car ces deux requêtes provoquent nécessairement un conflit de destinataire), soit au niveau de chaque banc mémoire. Mais la production automatique de cette instruction (par un compilateur) ne peut se faire que dans quelques cas particuliers. De plus, cette instruction interdit toute compression des accès à la mémoire lorsqu'il y a masquage.

Nous avons donc défini un mode de fonctionnement *itération* du multiprocesseur. Comme pour les instructions vectorielles, le traitement des instances d'une instruction **pour_tout** avec spécification **séquentiellement** est réparti sur l'ensemble des processeurs. La notion de mode itération est liée au fait que, sur les ressources partagées, les requêtes doivent être ordonnées par indice de contrôle croissant, comme on le fait dans le cas du monoprocesseur DSPA. L'implantation de ce mode de fonctionnement peut se faire de la manière suivante :

→ Chaque banc mémoire possède une FIFO de requêtes par processeur dans laquelle sont déposées les requêtes en mode itération.

→ Lorsqu'un processeur a fini d'émettre les requêtes correspondant à la même valeur de contrôle, il diffuse vers tous les bancs un ordre indiquant la fin du traitement de sa valeur de contrôle courante. Cette ordre est une instruction de son unité fonctionnelle mémoire. Comme toutes les requêtes émises dans ce mode, cet ordre est rangé dans les FIFOs des requêtes en provenance du processeur émetteur vers chaque banc.

→ lorsqu'un processeur a fini les traitements d'une certaine valeur de contrôle, il traite les instructions correspondant à la prochaine valeur qui lui est allouée.

→ Après passage dans ce mode de fonctionnement, chaque banc mémoire traite d'abord les requêtes en provenance du processeur 0, c'est à dire en ne prenant que les requêtes de la FIFO du processeur 0. Lorsque l'ordre de fin d'itération est extrait de la FIFO courante, le banc extrait les requêtes de la FIFO des requêtes en provenance du processeur 1 et ainsi de suite cycliquement sur les processeurs.

L'effet obtenu par ce mode itération est que chaque mémoire traite les requêtes dans le même ordre que celui produit par une exécution séquentielle. Ceci garantit la sémantique de l'instruction. Notons que ce système se contente d'ordonner les requêtes. Le mécanisme de doublement des écritures par les lectures continue de fonctionner, et tous les bancs ne traitent pas nécessairement des requêtes de la même itération à un instant donné. Après un délai de startup plus ou moins long, le débit potentiel de la mémoire peut être atteint.

Cependant, les dépendances possibles des instructions traitées sont généralement limitées à quelques accès à la mémoire. Par exemple, dans le cas de la multiplication matrice creuse par un vecteur plein, les seules dépendances possibles se limitent aux accès au vecteur a (figure 9). Par contre, il est fréquent que le résultat d'une lecture mémoire soit utilisé pour calculer l'adresse d'un accès ultérieur, ce qui retarde l'émission de ce deuxième ordre au plus tôt à la réception de la valeur d'adresse. Ces dépendances internes de séquencement peuvent ralentir fortement l'émission des requêtes. Il est possible, dans ce cas, de traiter les instructions parallèles **pour_tout** avec spécification **séquentiellement** par une première séquence d'accès en mode vectoriel (accès aux données sur lesquelles aucune dépendance n'existe qui peuvent également être émis en mode MIMD pour les lectures) suivi d'un passage en mode pipeline et des requêtes à exécuter dans ce mode. Le corps de l'instruction se termine par un retour en mode vectoriel. Ce découpage s'applique à de nombreux cas de programmation numérique rencontrés.

La fin du mode pipeline doit être annoncée soit par l'un des processeurs, soit par tous les processeurs auquel cas tous les processeurs doivent émettre le même nombre d'ordres de fin d'itération.

L'algorithme de multiplication de matrice creuse par un vecteur plein peut être codé comme dans le figure 11 sur le multiprocesseur DSPA. Chaque processeur exécute cette séquence d'instructions. La boucle externe gère des itérations vectorielles (sur des vecteurs de longueur 128, par exemple). Les vecteurs l, c, v et b sont lus dans ce mode vectoriel. Chaque processeur conserve sa portion du vecteur l dans sa mémoire locale à l'adresse Ad. A l'intérieur de cette boucle, il y a passage en mode séquentiel par l'instruction inititLV définie en [Jégou 86a] : cette

```
| M1 :  lire,n         ↣  dS | Sq :  initBV, M1  ↣      |
| M  :  Vlpiv,Dc       ↣  aM | *f :  VV*, M, M   ↣ +fd  |
| M  :  Vlpic,Dv       ↣ *fg | M1 :  Vécrire,Ad,M↣      |
| M  :  Vlbase,Db,M ↣ *fd    | M1 :  init,Ad,Da1 ↣      |
| M  :  Vlpic,D1       ↣  M1 | M1 :  init,Ad,Da2 ↣      |
| M  :mode itération↣        | Sq :  inititLV,R1 ↣      |
| M  :  lbase,Da,M1 ↣ +fd    | M1 :  lpic,Da1     ↣  aM |
| M  :  ébase,Da,M1 ↣        | M1 :  lpic,Da2     ↣  aM |
| M  : fin itération↣        | +f :  +, M, *f     ↣  dM | Sq : itère,R1 ↣ |
| M  :fin mode itéra↣        | Sq :  itèreBV      ↣     |
```

figure 11

instruction initialise le compteur d'itération d'après la valeur courante de vecteur LV. Les processeurs émettent une instruction de passage en mode itération qui provoque une synchronisation des unités fonctionnelles mémoire de tous les processeurs avant d'initialiser la boucle séquentielle. A la fin de ces itérations séquentielles, chaque processeur émet l'ordre de fin de mode pipeline qui provoque le retour en mode vectoriel/MIMD des mémoires dès qu'elles ont extrait cette requête de la FIFO associée à chaque processeur.

A l'exécution de cette séquence en parallèle, aucune contrainte ne limite le débit de séquencement. Le passage en mode pipeline limite l'extraction des requêtes mémoire aux requêtes émises par le processeur 0. Mais, ceci n'empêche pas les autres processeurs d'émettre leurs requêtes : les unités fonctionnelles mémoire ne sont pas synchronisées. La boucle scalaire ne contient que deux accès à la mémoire. Donc, seules deux mémoires reçoivent des requêtes d'une même itération d'un même processeur. Dès qu'une mémoire décode l'ordre de fin d'itération d'un processeur, elle traite immédiatement les requêtes du processeur suivant et qu'elle a déjà reçues selon toute probabilité. En effet, les unités fonctionnelles mémoire sont totalement indépendantes pendant cette phase de calcul et émettent donc leurs requêtes à leur rythme. Il suffit donc que le traitement des ordres de fin d'itération par les mémoires soit suffisemment rapide pour que le débit potentiel des échanges avec la mémoire puisse être atteint. Chaque banc physique traite les accès au vecteur a dans l'ordre

séquentiel. La vitesse d'exécution de ces instructions est limitée uniquement par la présence des dépendances réelles sur les accès au vecteur a dans l'exemple précédent.

Ce traitement type des instructions parallèles **pour_tout** avec spécification **séquentiellement** peut être appliqué dans de nombreux algorithmes numériques qui ne peuvent être exécutés que séquentiellement sur les supercalculateurs existants. Comme nous l'avons indiqué pour le fonctionnement SPMD, il est également possible de répartir dynamiquement les itérations sur les processeurs. Mais il faudrait alors indiquer aux mémoires à quel processeur est associé chaque indice, dans une FIFO par exemple. La sélection de la FIFO suivante au décodage d'un ordre de fin d'itération n'est plus statique, mais déduite la liste d'association des itérations aux processeurs. Comme pour le SPMD, une telle allocation dynamique permet d'équilibrer la charge des processeurs, mais elle interdit l'utilisation d'accès synchronisant les unités fonctionnelles mémoire ainsi que la production d'adresses par postincrémentation (qui demandent une allocation statique).

4 Intégration de la mémoire d'un multiproceseur DSPA

Nous traitons dans ce paragraphe le cas général d'une mémoire d'un multiprocesseur DSPA qui possède les trois modes de fonctionnement MIMD, vectoriel et itération. L'exemple traité possède p processeurs, m mémoires logiques, chacune de ces mémoires étant découpée en n bancs physiques. Les échanges entre les processeurs et la mémoire principale transitent par trois réseaux dinterconnexion : un réseau $p \times m$ de requêtes, un réseau $p \times m$ des données à écrire et un réseau $m \times p$ des données lues. Chacun de ces réseaux doit être capable de diffuser une information reçue sur l'une de ses entrées vers toutes ses sorties. La seule synchronisation des unités fonctionnelles mémoire se limite à la synchronisation de l'émission des requêtes vectorielles synchrones, des requêtes de lecture de scalaires avec diffusion et des requêtes de passage en mode itération sur le réseau de requêtes et au traitement des ordres de synchronisation explicite. L'émission des requêtes MIMD et en mode itération ainsi que l'émission des données à écrire ou des données lues ne provoquent aucune synchronisation. Les mémoires et les processeurs sont indépendants. Les données reçues par

les unités fonctionnelles mémoire sont partiellement ordonnées : celles reçues de la même mémoire le sont dans l'ordre où les requêtes correspondantes ont été émises. Par contre, il n'y a pas de relation sur l'ordre d'arrivée de données en provenance de deux mémoires différentes. Comme nous l'avons fait pour la mémoire multibanc d'un DSPA monoprocesseur, nous intégrons une unité de réordonnement des données lues aux unités fonctionnelles mémoire des processeurs.

Chaque banc d'une mémoire entrelacée exécute ses requêtes dans l'ordre de réception. Mais, comme pour les données lues sur les unités fonctionnelles mémoire des processeurs, les données à écrire ne sont que partiellement ordonnées sur chaque processeur. Une unité de réordonnement des données à écrire doit donc être prévue. La présence de cette unité est nécessaire pour permettre une association simple des données à écrire avec leurs adresses. Dans l'exemple traité, cette unité est placée au niveau de chaque mémoire logique.

La figure 12 présente les mécanismes à mettre en oeuvre tant au niveau des unités fonctionnelles mémoire des processeurs qu'au niveau des mémoires logiques. Dans cet exemple, nous avons représenté une unité fonctionnelle mémoire et une mémoire logique. L'UF mémoire est composée d'un séquenceur des requêtes Sq, d'un séquenceur des données à écrire e_c et d'une unité de réordonnement des données lues Ord. Le séquenceur des requêtes reçoit les instructions à exécuter dans la FIFO Ip et les adresses associées dans les FIFOs associées à l'entrée logique aM. Ce séquenceur gère également des descripteurs d'adresses internes. Il émet ses requêtes sur le réseau des requêtes (le destinataire est généralement déterminé par l'adresse), les informations d'émission des données à écrire vers l'unité d'écriture e_c par la FIFO Rep et les informations de réordonnement des données lues (couples formés d'un numéro de mémoire et du destinataire de la donnée) vers l'unité de réordonnement Ord par la FIFO Rop. Ce séquenceur gère également le traitement local d'erreurs (voir [Jégou 86a]). L'historique des erreurs d'adressage est localisé dans le séquenceur et l'historique des erreurs des données écrites est localisé dans l'unité e_c. L'exécution des instructions de traitement de ces historique provoque des échanges entre le séquenceur et l'unité d'écriture, le résultat étant émis sur la sortie logique M par l'unité de réordonnement Ord.

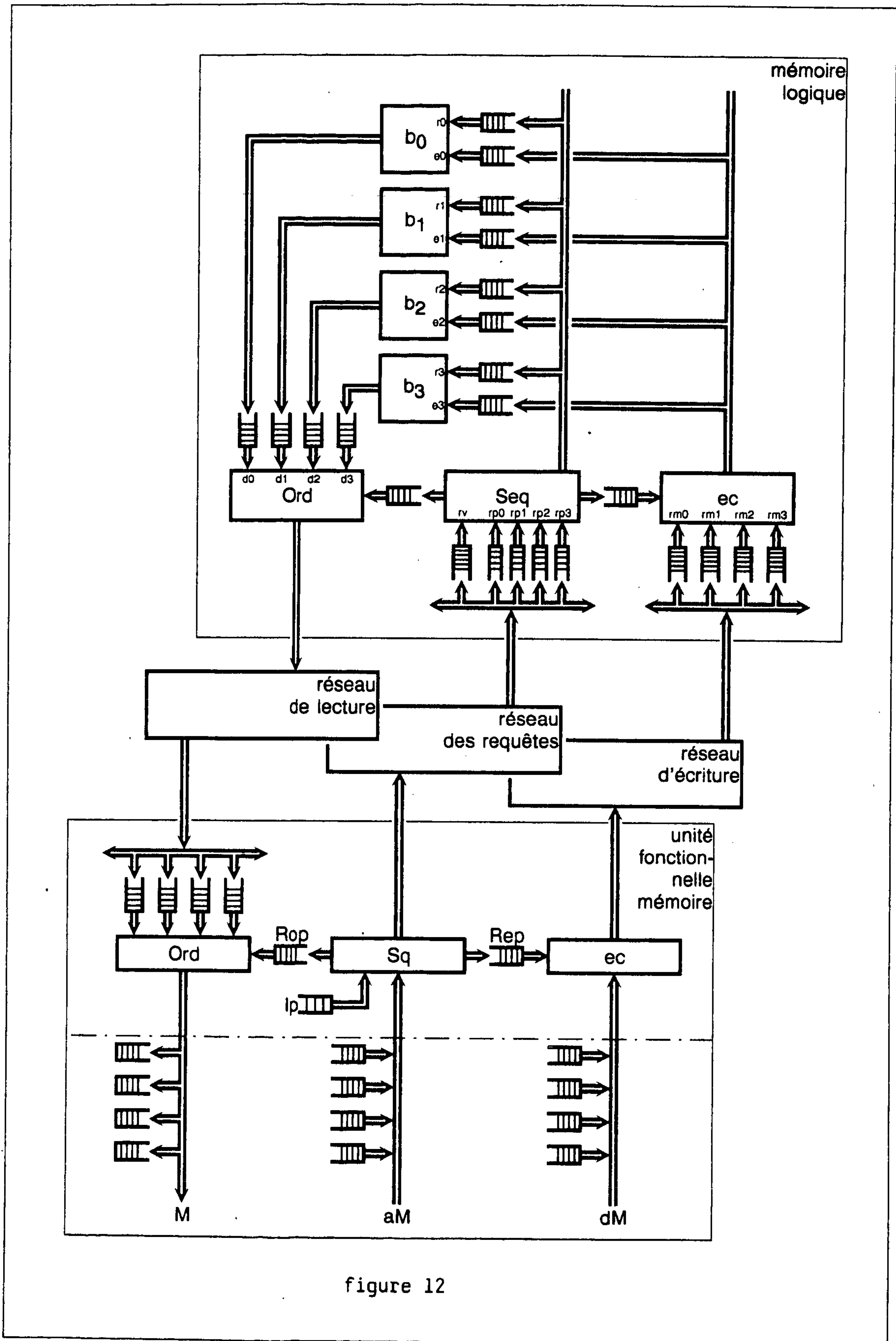

figure 12

L'unité e_c a pour rôle unique d'émettre les données à écrire vers le réseau d'écriture dans l'ordre où les requêtes correspondantes ont été émises. Notons qu'il suffirait de réordonner les requêtes destinées à une même mémoire logique.

L'unité de réordonnement Ord émet les données lues vers les FIFOs du processeur dans l'ordre où les requêtes correspondantes ont été exécutées. La présence de cette unité est nécessaire dans les multiprocesseurs multimémoire. Les données reçues du réseau de lecture sont rangée dans les FIFOs Rm_i associées à chaque mémoire logique.

Ce découpage en unité de séquencement des requêtes, unité d'écriture et unité de réordonnement des données lues se retrouve au niveau de chaque mémoire. Le séquenceur Seq de chaque mémoire prend ses requêtes, soit de la FIFO des requêtes vectorielles/MIMD rv, soit des FIFOs rp_i associées à chaque processeur en mode itération. Le choix de la FIFO qui contient la prochaine requête à traiter dépend du mode de fonctionnement courant. Les changements de FIFOs de requêtes sont provoquées de la manière suivante. Les requêtes de passage en mode itération sont exécutées en mode vectoriel synchronisé (instructions vectorielles qui ont provoqué une synchronisation des unités fonctionnelles mémoire et diffusées à toutes les mémoires) et donc reçues de la FIFO des requêtes vectorielles/MIMD rv. L'exécution d'une requête de passage en mode itération par une mémoire provoque la sélection de la FIFO de requêtes du processeur 0. Le passage de la FIFO de requêtes du processeur i à la FIFO de requêtes du processeurs i+1 (cycliquement sur les processeurs) est provoqué par l'extraction d'une instruction de fin d'itération de la FIFO de requêtes du processeur i. Le retour en mode vectoriel/MIMD est provoqué par le décodage d'une instruction de fin de mode itération dans la FIFO de requêtes de chaque processeur (ou de la FIFO de requêtes du processeur exécutant la dernière itération). Ces synchronisations de commutation de mode sont du type fork-join.

Le séquenceur de chaque mémoire logique émet les requêtes vers les bancs physiques (où sont traités les doublements et les aléas), les informations de réordonnement des données à écrire vers l'unité e_c de la mémoire, et les informations de réordonnement des données lues par les bancs vers l'unité de réordonnement Ord.

Le rôle de l'unité d'écriture e_c est d'émettre vers chaque banc physique les données à écrire dans l'ordre des requêtes. Son fonctionnement se base sur l'ordre des données reçues de chaque processeur.

L'unité de réordonnement émet les données lues sur le réseau de lecture dans l'ordre où les requêtes correspondantes ont été exécutées par le séquenceur de la mémoire (et pas nécessairement dans l'ordre où ces requêtes ont été reçues de manière globale). Les données lues sont émises vers chaque processeur dans l'ordre où il a émis ses requêtes de lecture, qu'elles soient de mode vectoriel/MIMD ou itération.

Chaque banc physique (figure 13) reçoit ses requêtes dans une FIFO de

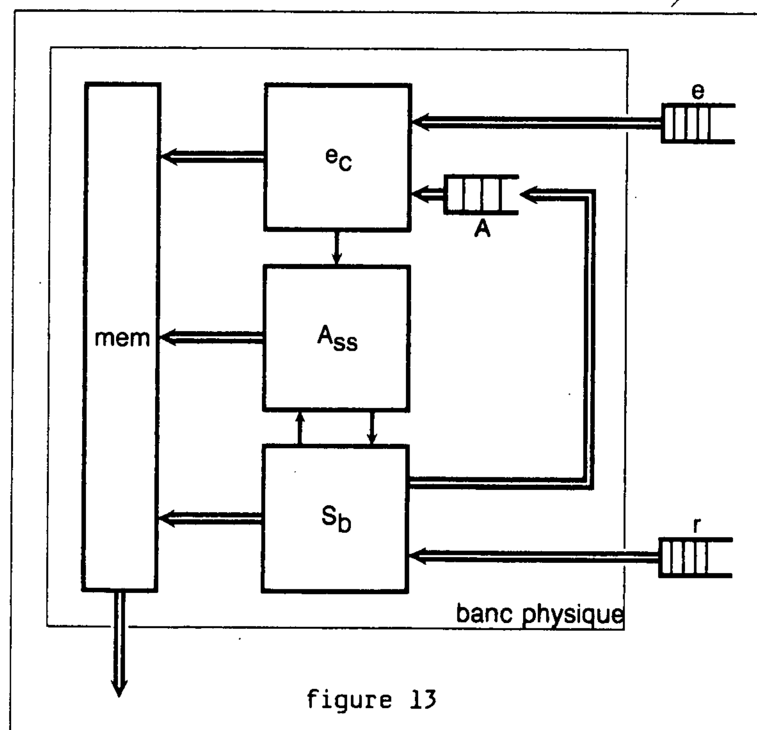

figure 13

requêtes r et les données à écrire dans une FIFO des données à écrire e. Le séquenceur local S_b traite les requêtes, vérifie les dépendances sur la mémoire associative A_{ss} et déclenche les lectures dès que les dépendances éventuelles disparaissent. Le traitement d'une requête d'écriture provoque également une vérification de dépendance (écriture après écriture), puis l'adresse d'écriture est rangée dans la mémoire associative A_{ss}, l'entrée

de cette mémoire est placée dans la FIFO interne A. L'unité locale d'écriture e_c reçoit ces entrées de cette FIFO A et les données correspondantes dans la FIFO e. Cette unité déclenche les écritures en mémoire et libère les entrées correspondantes de la mémoire associative. C'est à ce niveau que peuvent être mémorisées les données récemment écrites pour en accélérer l'accès (voir [Jégou 86a]). La mémoire associative des adresses et la FIFO de mémorisation des entrées pourraient être fusionnées pour former un circuit jouant le rôle de FIFO lors de la réception des adresses d'écriture et des données à écrire, et permettant la comparaison des adresses de lecture aux adresses mémorisées.

Cette description fonctionnelle de la gestion mémoire du multiprocesseur DSPA est très générale. Nous n'avons pas tenu compte du type des réseaux d'interconnexion utilisés (il faut cependant que ces réseaux respectent la contrainte d'ordre partiel exposé précédemment). La notion de mémoire logique est liée au nombre de ports d'accès sur les réseaux du côté de la mémoire. La notion de banc physique permet d'atteindre un certain débit potentiel d'échanges par mémoire logique en utilisant des mémoires de débits inférieurs. De nombreuses simplifications peuvent être apportées, par exemple lorsque chaque banc logique ne contient qu'un seul banc physique.

Une intégration globale de tous ces mécanisme a été définie par André Seznec dans [Seznec 86]. Elle montre comment les réseaux d'interconnexion peuvent être traités dans la philosophie du DSPA, entraînant de nombreuses simplifications des mécanismes exposés, et permettant des échanges virtuellement sans conflits entre les processeurs et la mémoire.

5 Le multiprocesseur DSPA

Nous venons de montrer que le type de séquencement que nous avons défini pour le pipeline synchronisé par les données permet une intégration simple de ce processeur dans une architecture multiprocesseur MIMD ou vectoriel. Nous avons également montré qu'il est possible d'exécuter avec un bon niveau de parallélisme des algorithmes qui ne peuvent être exécutés qu'en séquentiel sur les architectures classiques pipelines ou SIMD. De plus, ces trois modes de fonctionnement MIMD, vectoriel et itération

peuvent parfaitement cohabiter dans une architecture multiprocesseur. La compilation de langages de haut niveau pour une telle architecture ne présente pas de difficulté majeure. Nous en avons exposé les principes en [Jégou 86a] pour le monoprocesseur DSPA. Dans le cas d'un multiprocesseur, les instructions SPMD de Hellena sont produites en mode MIMD. Les instructions vectorielles et les instructions parallèles **pour_tout** sont traduites directement par des instructions vectorielles à deux niveaux : parallélisme des processeurs et instructions vectorielles sur chaque processeur. L'exécution de ces instructions vectorielles et parallèles est répartie sur l'ensemble des processeurs. Lorsque les dépendances de type *écriture après lecture* sont malmenées par le tronçonnage vertical dans les instructions vectorielles ou les instructions **pour_tout** spécifiées parallèles, l'utilisation des unités fonctionnelles Fi des processeurs permet d'imposer l'ordre d'accès aux données en mémoire. Les dépendances de type *lecture après écriture* sur les vecteurs accédés en parallèle par les processeurs sont traitées, soit par des instructions de synchronisation explicite des unités fonctionnelles mémoire, soit par des instructions d'écriture synchrones. Les conditionnelles des instructions parallèles sont, en général, traitées par des masquages des processeurs. Lorsque des spécifications de séquentialité apparaissent sur des instructions parallèles **pour_tout**, une analyse simple des dépendances sur les variables accédées dans l'instruction permet de déterminer les accès sans dépendances (les plus fréquents) qui peuvent être traduits en accès vectoriels, et les accès dépendants qui doivent être rassemblés en mode itération. Cette séparation peut être réalisée dans la majorité des cas où l'instruction **pour_tout** ne contient qu'une seule instruction d'affectation. La compilation de certains cas particulier peut cependant entraîner un traitement séquentiel.

L'efficacité de ce multiprocesseur sur une large famille d'algorithmes est élevée tant que le nombre de processeurs reste faible (<16). Ses performances se dégradent cependant dans un certain nombre de cas parmis lesquels nous pouvons citer :

● **les vecteurs courts** : la répartition des calculs sur les processeurs peut être difficilement équilibrée en présence de vecteurs trop courts. L'efficacité chute rapidement si le nombre de processeurs est élevé.

● **un nombre trop faible de tâches :** lorsque le nombre de tâches à exécuter en parallèle est trop faible, il faut choisir entre une exécution monoprocesseur de chaque tâche au prix d'une inactivité de certain processeurs, et une exécution multiprocesseur de chaque tâche qui peut se heurter à un faible niveau de parallélisme du code de ces tâches.

● **les rebouclages d'unités fonctionnelles :** bien qu'ils présentent un bon potentiel de parallélisme, le parallélisme de certains codes est limité par le fait que certaines unités fonctionnelles doivent traiter des données issues de calculs sur ces mêmes unités fonctionnelles. Le cas le plus critique se situe dans le traitement des accès indirects à la mémoire où une première requête provoque la lecture d'une valeur d'indice consommée par une requête suivante sur cette même unité fonctionnelle. Ces problèmes pourraient être traités, soit en permettant des doublements d'instructions sur les unités fonctionnelles, soit en dédoublant les unités fonctionnelles. Le doublement des instructions sur les unités fonctionnelles ne nous paraît pas souhaitable, d'une part parce qu'elle complique l'intégration du processeur, d'autre part parce qu'elle remet en cause l'ordre logique d'exécution défini à la génération de code et donc les possibilités de mise au point.

6 Le multipipeline DSPA

Le multipipeline DSPA intègre les modes MIMD, vectoriels, itération et le système de communication direct d'un multiprocesseur DSPA. L'objectif recherché est une adaptation plus fine des différents niveaux de parallélisme du multiprocesseur et des algorithmes. Cet objectif est atteint sur le multipipeline par une modulation du parallélisme offert à chaque niveau MIMD, vectoriel et pipeline.

● **les vecteurs courts :** il suffit d'analyser la complexité de l'instruction vectorielle, de déterminer le nombre de processeurs r sur lesquels le calcul d'un élément du vecteur peut être réparti en utilisant le système de communication directe, et de répartir l'exécution de l'instruction vectorielle sur p/r processeurs. Par exemple, soit l'instruction vectorielle Hellena de la figure 14. Le calcul d'un élément de cette instruction peut être réparti sur deux processeurs P_1 et P_2 comme

```
a := b + c*d ;
```

figure 14

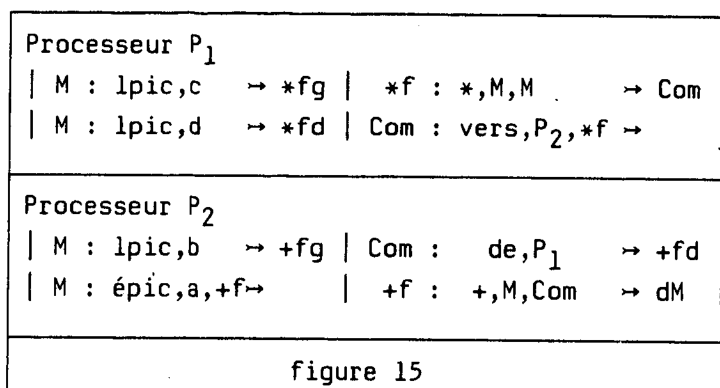

figure 15

dans la figure 15. L'effet produit par cette répartition est celui qu'on pourrait obtenir avec deux fois moins de processeurs qui posséderaient deux fois plus d'unités fonctionnelles et qui pourraient séquencer et décoder deux fois plus d'instructions à chaque cycle : diminution du parallélisme multiprocesseur et augmentation du parallélisme pipeline.

● **un nombre trop faible de tâche :** le multipipeline DSPA permet l'exécution de *r* tâches en parallèle, l'exécution de chacune de ces tâches étant réparti sur *p/r* processeurs (*r* étant une puissance de 2). Pour conserver toutes les possibilités offertes par le multiprocesseur, le mode itération, les instructions vectorielles synchrones, les instructions de synchronisation explicite et les instructions de lecture de scalaires se réfèrent, non plus à l'ensemble des processeurs, mais à des groupes de processeurs.

● **le rebouclage des unités fonctionnelles :** lorsque des rebouclages d'unités fonctionnelles sont détectés, il suffit de répartir la séquence de codes sur deux processeurs en faisant en sorte que l'un des processeurs exécute toutes les requêtes en amont des rebouclages, l'autre processeur exécutant les requêtes en aval. Les instructions sont alors traduites pour *p/2* processeurs.

6.1 les groupes de processeurs

Un multipeline est structuré en groupes de processeurs. Ces groupes sont déterminés en construisant un arbre binaire à p feuilles correspondant aux processeurs (figure 16). Chaque noeud de l'arbre définit un groupe de

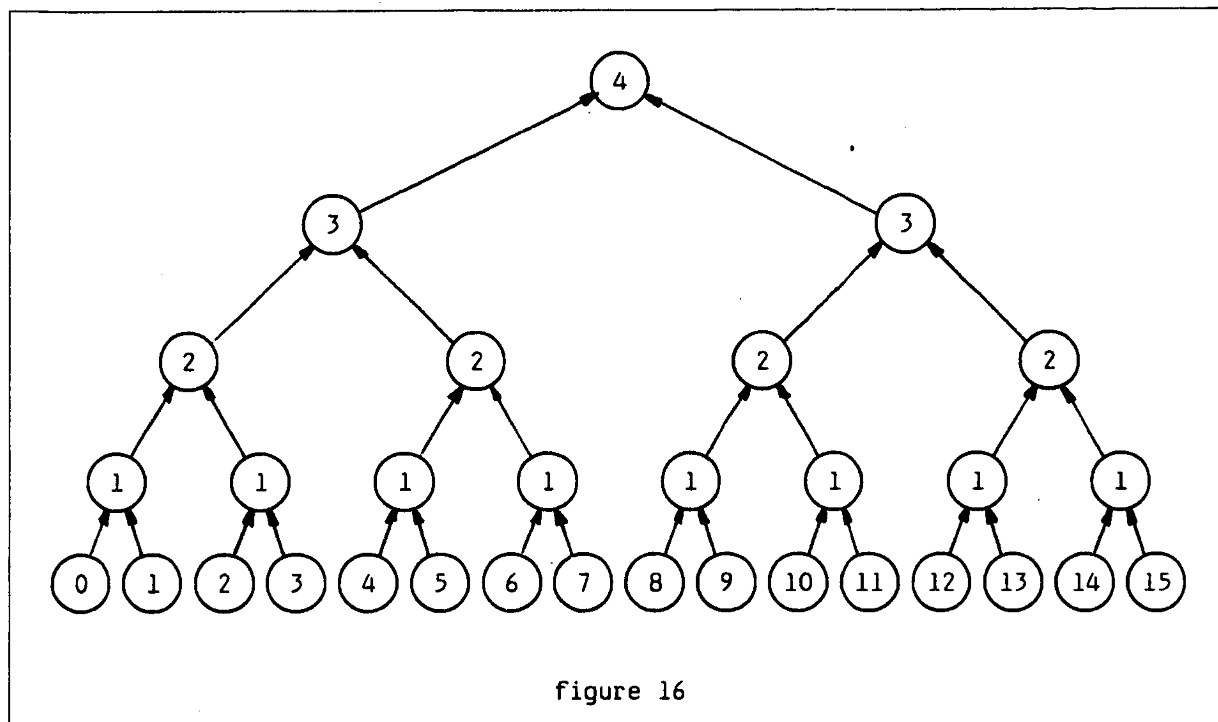

figure 16

processeurs. Chaque processeur appartient à tous les groupes localisés entre la racine et la feuille qui lui correspond. Le groupe racine désigne l'ensemble des processeurs. Un processeur désigne un groupe par la distance entre la feuille et le noeud correspondant au groupe. Un groupe désigné par le numéro i contient donc 2^i processeurs. Dans les instructions vectorielles synchrones et les instructions de synchronisation explicite, le numéro de groupe désigne l'ensemble des processeurs dont les unités fonctionnelles mémoire doivent être synchronisées. Le rendez-vous a lieu lorsque tous les processeurs du groupe ont désigné ce groupe dans une instruction provoquant une synchronisation. Dans une instruction de passage en mode itération, le numéro de groupe désigné définit l'ensemble des processeurs dont les requêtes doivent être réordonnées sur les mémoires. L'exécution d'une instruction de passage en mode itération

provoque une synchronisation des unités fonctionnelles mémoire des processeurs du groupe et la diffusion d'une requête de passage en mode itération des processeurs du groupe vers les mémoires.

Cette notion de groupes de processeurs est également prise en compte dans les instructions de communication directe de deux manières : les désignations de groupes, les désignations absolues à l'intérieur d'un groupe et les désignations d'après les liaisons d'un hyper-cube.

o Les désignations de groupes sont utilisées dans les instructions de diffusion d'une information à tous les processeurs d'un groupe.

o Les désignation absolues à l'intérieur d'un groupe permettent une communication directe entre tous les processeurs d'un groupe.

o Les désignations suivant les liaisons d'un hyper-cube sont généralement utilisées dans les communications entre deux groupes de processeurs.

Le principal intérêt de ces systèmes de désignation des groupes et des processeurs est qu'il permet une production de code indépendante des numéros physiques des processeurs. Par exemple, sur un multipipeline à 16 processeurs, le code d'une tâche compilée pour un groupe de 4 processeurs est exécutable sur les 4 groupes de 4 processeurs de l'architecture. De même, lorsque des rebouclages d'unités fonctionnelles provoquent le partage en un groupe amont et un groupe aval, les processeurs en communication se désignent par le numéro du bit qui diffère dans leur numérotation. Tous les processeurs d'un même groupe exécutent donc le même code binaire.

6.2 les groupes de processeurs dans les mémoires

Les groupes de processeurs doivent être pris en compte au niveau des mémoires pour le traitement du mode itération. Bien qu'il soit possible d'imaginer un fonctionnement permettant un mélange de groupes, nous nous limitons au cas où chaque processeur appartient à un instant donné à un seul groupe. Les changements de groupe sont annoncés explicitement par les instructions de début de mode itération et de fin de mode itération. Nous ne détaillons pas ici la mise en oeuvre du système de gestion des groupes

sur la mémoire qui fera l'objet d'une description détaillée. En résumé, ce mécanisme est basé sur l'utilisation d'une FIFO de requêtes par mémoire et par processeur et d'une FIFO indiquant l'ensemble des processeurs ayant émis une requête dans un même cycle. Le passage en mode itération sur un groupe de processeurs provoque un masquage des processeurs de ce groupe excepté le premier. Les requêtes des processeurs non masqués sont traitées cycle par cycle, celles des processeurs masqués étant mémorisées. Le traitement d'une requête de fin d'itération provoque le masquage du processeur, le démasquage du processeur suivant dans le groupe, le traitement de ses requêtes mises en attente etc... Ce système garantit non seulement le traitement en mode itération mais également le respect de l'ordre de traitement des requêtes après la fin du mode itération.

Ce système de traitement des requêtes sur les mémoires permet de rendre totalement indépendants les groupes de processeurs du multipipeline. Nous résumons ici les possibilités offertes aux programmeurs (et aux compilateurs) :

● on peut décider du nombre de processeurs sur lesquels l'exécution d'une tâche doit être répartie. Le code de la tâche peut utiliser toutes les possibilités offertes par le multipipeline : modes itération, vectoriel et MIMD, communication directe inter-processeur, redécoupage en sous-groupes exécutant des sous-tâches. Le code de la tâche est exécutable sur tous les groupes du multipipeline de la bonne dimension. Les sous-tâches sont exécutables sur tout sous-groupe de bonne dimension du groupe alloué à la tâche.

● Les instructions vectorielles courtes peuvent être réparties sur des sous-groupes du groupe alloué à la tâche qui les exécute.

● Les cas de rebouclages d'unités fonctionnelles peuvent être résolu en formant un sous-groupe amont et un sous-groupe aval du groupe de processeurs chargé de l'exécution de la tâche. Le seul cas où cette répartition est impossible est limité au cas ou un seul processeur est alloué à la tâche.

7 un exemple de programmation du multipipeline DSPA

Soit l'algorithme de multiplication de matrice creuse par un vecteur plein de la figure 17 programmé en Hellena. Sur un multiprocesseur, cet

```
pour_tout i séquentiellement dans [1,n] :
   a(l(i)) := a(l(i)) + v(i)*b(c(i)) ;
fin_pour_tout ;
```

figure 17

algorithme doit être programmé en mode itération. Lorsque l'ensemble des itérations est réparti sur tous les processeurs, il faut éviter les rebouclages dûs à la lecture des indices sur l'unité fonctionnelle mémoire. Pour celà, certaines transformations comme la lecture en mode vectoriel des vecteurs d'indices suivie d'un passage en mode itération scalaire doivent être effectuées à la génération du code. Ces transformations sont difficiles à réaliser de manière automatiques. Sur le multipipeline, il suffit de former un groupe *amont* des processeurs qui lisent les vecteurs d'indices et un groupe *aval* de processeurs qui effectuent les calculs. Supposons une machine constituée de 16 processeurs, chaque groupe est formé de 8 processeurs. Dans les instructions vectorielles et en mode itération, chaque processeur désigne un groupe par $\log_2$(nombre de processeur du groupe). Dans les instructions de communication chaque processeur désigne son correspondant par le numéro du bit qui les diffère (connexion de type hyper-cube). Sur le groupe amont de calcul des adresses, le code de la figure 18 est produit. Les processeurs de ce groupe amont calculent les

```
|Sq:initBS,Ri,R→ |
| M: lpic,c→Ald|R:lire,Ab→Alg |Al :+,R,M →Com|Com:vers,3,Al→|
| M: lpic,l→Ald|R:lire,Al→Alg |Al :+,R,M →Com|Com:vers,3,Al→|
| M: lpic,l→Ald|R:lire,Al→Alg |Al :+,R,M →Com|Com:vers,3,Al→|
|Sq:itèreBS,Ri→ |
```

figure 18

adresses des éléments des vecteurs a et b et les émettent vers les processeurs correspondants du groupe aval de calcul sur les données. Cette

séquence de code est entourée d'instructions de gestion d'itérations séquentielles. Les accès à la mémoire par ce groupe se font en mode MIMD, il n'y a aucune dépendance.
Sur le groupe aval de calcul sur les données, le code de la figure 19 est produit. Toutes les instructions d'accès à la mémoire des processeurs de

```
| M :    initIT,3  →     |
|Sq : initBS,Ri,R →      |
| M :     lpic,V    →*fg | Com:    de,3   →aM |
| M :    lire,Com   →*fd | Com:    de,3   →aM | *f :  *,M,M    →+fd |
| M :    lire,Com   →+fg | Com:    de,3   →aM | +f :  +,M,*f   → dM |
| M :écrire,Com,+f→      |
| M :    finIT,3    →    | Sq :itèreBS,Ri →   |
| M :  finmodeIT,3 →     |
```

figure 19

ce groupe aval sont exécutées en mode itération (sur le groupe 3 contenant 2^3=8 processeurs). Le mélange de modes étant interdit, il n'est pas nécessaire de répéter le mode ni le numéro de groupe dans les instructions. Bien que la charge des processeurs des deux groupes ne soit pas complètement équilibrée, les performances du multipipeline peuvent être très élevées sur cet algorithme. Après un certain délai de startup, les processeurs du groupe d'adresses prennent quelques itérations d'avance sur leurs homologues du groupe des données (aucune dépendance amont). Cet effet provient de l'élimination des rebouclages sur les unités fonctionnelles mémoire des processeurs. Une telle répartition est réalisée très simplement par un compilateur : elle consiste à associer les calculs des adresses à un groupe dit amont, les accès indirects au groupe dit aval puis à distribuer les instructions restantes de manière à équilibrer la charge tout en n'utilisant que des communications dans le sens amont vers aval pour éviter les circuits de dépendance. Une telle répartition n'est pas nécessairement limitée à deux groupes de processeurs. Dans l'exemple traité, il est, par exemple, possible de décomposer chacun de ces deux groupes en deux sous-groupes pour former quatre groupes de processeurs, un seul de ces groupes fonctionnant en mode itération. D'autre part, il n'est pas nécessaire de rèpartir cette instruction sur l'ensemble des processeurs. On peut décider de compiler pour un groupe de taille donné,

ce groupe étant lui-même découpé en un sous-groupe de processeurs amont et un sous-groupe de processeurs aval. Le code produit est alors exécutable sur tout groupe physique de la bonne dimension.

Cette notion de groupe du multipipeline permet également d'équilibrer la charge des processeurs lorsque plusieurs boucles emboîtées d'un programme peuvent être exécutées en parallèle. En effet, la vectorisation systématique sur la boucle interne donne de mauvais résultats lorsque le nombre d'itérations est trop faible (inactivité de certains processeurs). Il faut alors chercher à paralléliser les deux boucles les plus internes. Sur le multipipeline, cette opération est relativement simple. Elle consiste à considérer des groupes constitués de la racine carrée du nombre de processeur, à répartir l'exécution de la boucle externe sur ces groupes et à compiler la boucle interne pour un groupe.

8 CONCLUSION

Les performances de nombreux supercalculateurs existants se dégradent rapidement dès que certaines conditions très strictes ne sont plus respectées. En définissant le langage Hellena [Jégou 86b], nous avons cherché à expérimenter quelques concepts simples permettant une programmation "fiablement" efficace qui ne repose pas sur des processus de compilation complexes.

Puis nous avons étudié les limitations de certaines architectures qui font que seuls certains algorithmes très réguliers permettent d'atteindre les performances de pointe. De cette étude, nous avons extrait un certain nombre de contraintes qui doivent être nécessairement respectées pour pouvoir élargir le spectre des algorithmes exécutables de manière efficace, la contrainte principale étant curieusement la rapidité d'exécution de codes scalaires.

Dans le processeur DSPA que nous avons défini [Jégou 86a], le séquencement des instructions n'est plus le goulôt d'étranglement pour l'exécution de codes séquentiels. Les performances de ce processeur sont uniquement limitées par les performances des diverses unités fonctionnelles le composant. Cette souplesse d'utilisation a été obtenu tout en

conservant des processus de compilation peu complexes. Dans une deuxième étape, ce processeur a été intégré dans un environnement multiprocesseur qui permet des modes de fonctionnement de type SIMD et de type MIMD. Enfin, nous nous sommes intéressés à des algorithmes dont la parallélisation est limitée par la présence de dépendances entre les données. Moyennant la généralisation de certains mécanismes de gestion des mémoires parallèles, nous avons montré que l'exécution de ces algorithmes peut, dans de nombreux cas, être réparti sur tous les processeurs. Le parallélisme de l'exécution est alors limité uniquement par les dépendances effectives rencontrées sur les accès aux variables.

Le multiprocesseur DSPA présente trois niveaux de parallélisme : un parallélisme induit par le pipeline de chaque processeur sur du code séquentiel, un parallélisme de type SIMD obtenu par synchronisation de certains accès parallèles à la mémoire et un parallélisme de type MIMD obtenu par l'exécution de codes indépendants sur les processeurs. Le multipipeline DSPA permet de moduler le degré de parallélisme offert par chacun de ces niveaux.

Le faible niveau de complexité des traitements à réaliser pendant le processus de traduction des programmes laisse espérer une *redoutable* efficacité de ce multipipeline sur une famille très large d'algorithmes numériques dont le parallélisme potentiel ne pouvait jusqu'à présent être exploité que sur des architectures beaucoup plus complexes de type data-flow.

Références

[Jégou 86a]
Jégou Y., *Le DSPA, un pipeline synchronisé par les données*, Publication Interne de l'IRISA n°311, Sept. 86.

[Jégou 86b]
Jégou Y., *Hellena, un langage pour les calculateurs vectoriels*, à paraître dans les publications internes de l'IRISA.

[Jégou 86c]
Jégou Y., *Hellena, principes de mise en oeuvre*, à paraître dans les publications internes de l'IRISA.

[Jégou ?]
Jégou Y., *Le SDSPA, un processeur pipeline universel*, à paraître dans les publications internes de l'IRISA.

[Seznec 86]
Seznec A., Jégou Y., *Address Synchronized Multiprocessor Architecture*, Publication Interne IRISA n°301, Juin 86.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique